System design document for Pyromaniacs

Version: 2.1

Date: 2012-05-16

Author: Andreas Rolén, Adrian Bjugård, Joakim Persson, Viktor Anderling

This version overrides all previous versions.

Table of Contents

1 Introduction

1.1 Design goals

The design must 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.
- 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, area bombs, bomb power.
- Abilities, different abilities that can be upgraded at the shop by buying upgrades. These abilities are health points, number of bombs etc.
- 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

The system will use the MVC pattern. In our implementation we use a global access point from the model (see 2.1.2). Every main view is connected to its own controller class. It is also common that the main views which the controllers talk to manage their own sub-views for different containers.

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 a controller and a view. (For an overview of the the different states in the application see figure 1). This design works well with the slick framework and enables us to split up the application into smaller and more manageable pieces. However one problem that occurs is that sometimes two states need 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).

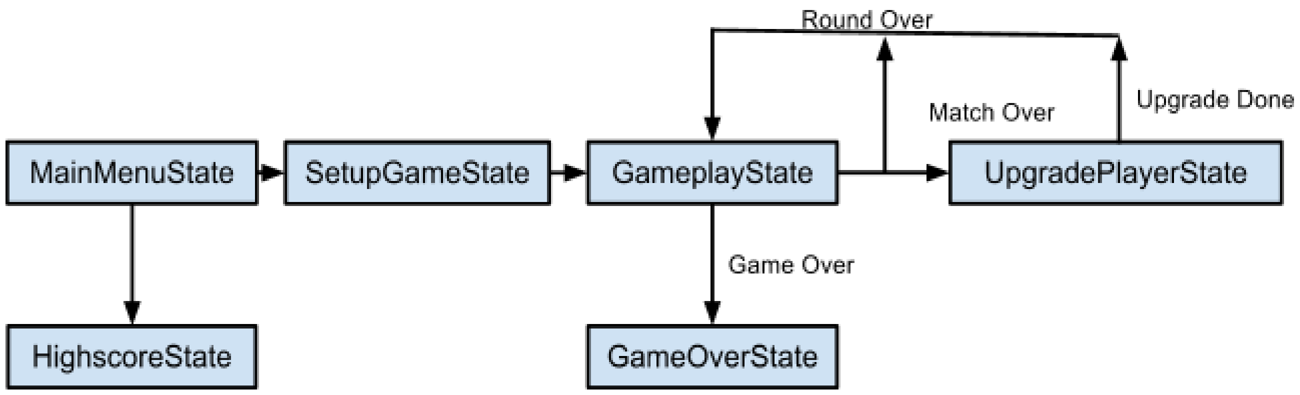


Figure 1: Game Flow

2.1.1 Rules

The Pyromaniac Game rules could vary by implementing different game modes. We handle it by having an IGameLogic 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 GameLogic 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 IPyromaniacModel. At run an unique instance of the PyromaniacModel 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 continually 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 asks the model for the values it needs. However if we replace 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 wherein we read from a .txt file that have been formatted in a special way. 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 Input System

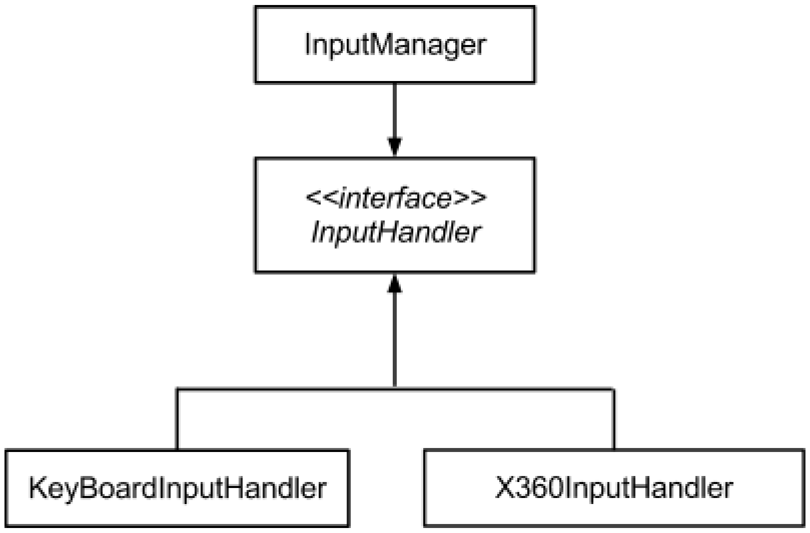


Figure 2: Input System

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.

2.1.6 Audio system

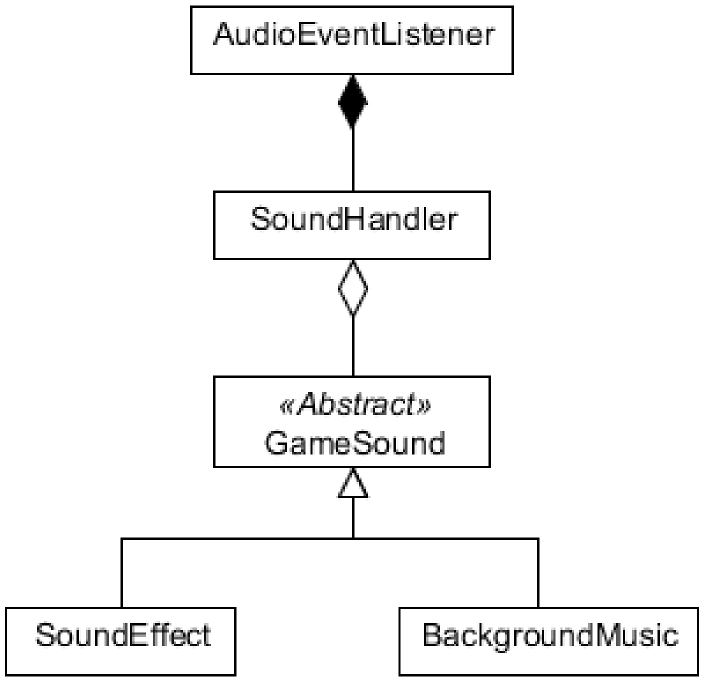


Figure 3: Audio System

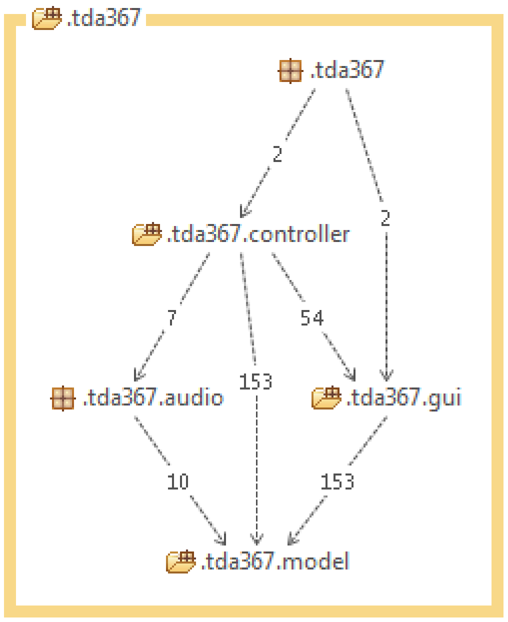
All the sound classes can be found in the Audio-package. The abstract class GameSound represents any type of one audio file. The difference between the two implementations of GameSound is how they are played. The implementation SoundEffect will only play once when told to "play", while BackgroundMusic loops. It may seem unnecessary to use multiple classes for this sole purpose, though the different GameSounds are handled differently in methods inside the SoundHandler class.
The SoundHandler class initiates and stores all instances of GameSounds in the game while also contributing with methods such playing playlists and setting the games different volumes. The AudioEventListener listens to commands and translates them to method calls which is passed on to the SoundHandler. Classes in the model that is connected to events that should play a sound will have AudioEventListener listening to that instance and they fire a property change with an Event that symbolises the event. This way, other possible listeners may be implemented that listen to these same events. An example would be if we decided to write our own framework for the GUI. Then we could use the sent events as an alert when different objects need to be redrawn.

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.

- controller.input.osdependant, the package including the OS Dependants input.
- controller.utils, the package including the controller utils.
- gui, the package including all the GUI related classes.
- gui.utils, the package including the gui utils.
- model, the package including all modelrelated classes.
- model.constants, the package including all the constants.
- model.gamelogic, the package including all the game logic.
- model.highscore, the package including the score and highscore.
- 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.



2.2.2 Dependency analysis

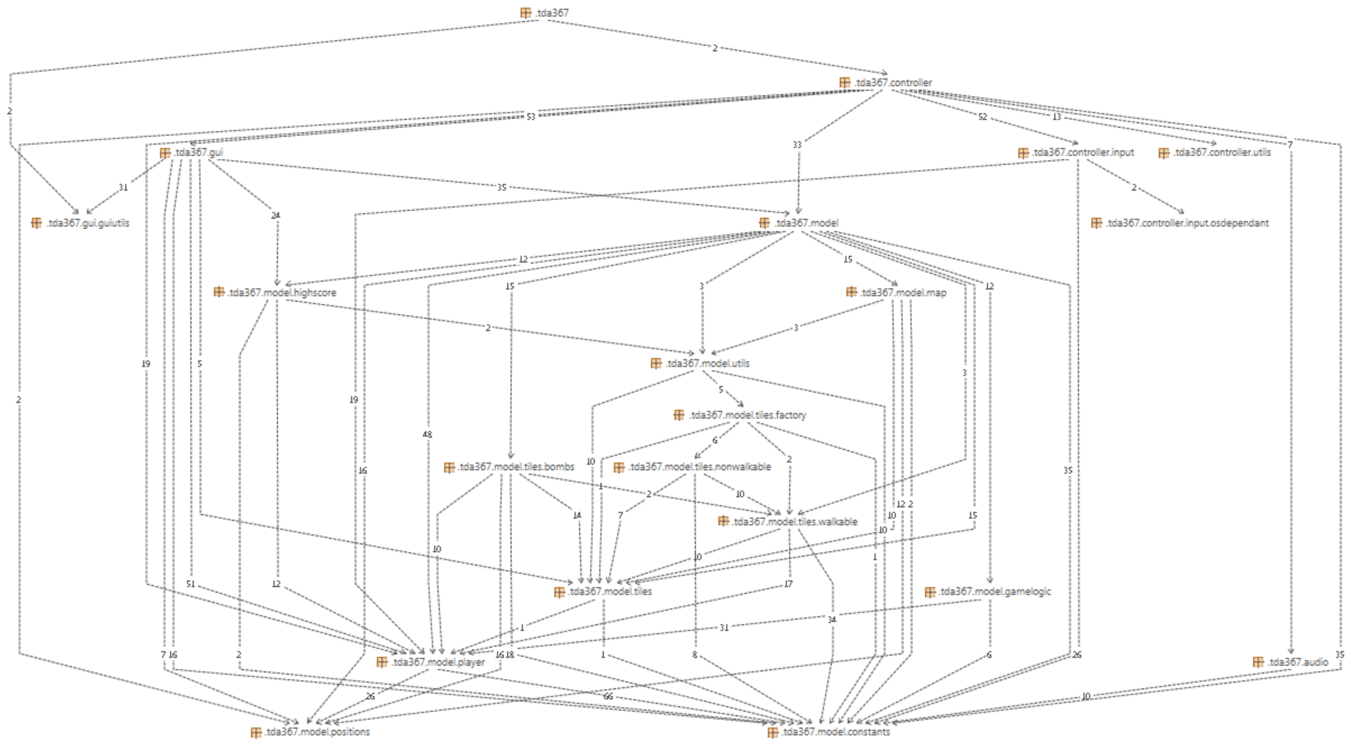


Figure 4: Dependency analysis

Dependencies are as shown in Figure. There are no circular dependencies between the packages.

2.3 Concurrency issues

The Timers that are used implement threads, these threads are synchronized and don't create any errors. How Slick2D uses threads can be read in more detail at their website. The PyromaniacModel class method for collecting the instance is also synchronized.

2.4 Persistent data management

All the models txt files for the map loading system is located in a maps folder inside the res folder and. The maps are converted into tile matrices by a special subsystem (see 2.1.4) at run-time.

Files generated by the application are stored in a "gen" folder which is created by the system if it didn't previously exist. An example of a generated file by the system is the highScore.data file containing the saved highscores.

2.5 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/