System design document for
“The Boss”

Version: 0.1

Date: 2015-05-18

Authors: Johan Carlshede, Johan Iversen, Johannes Hildén, Oskar Willman

This version overrides all previous versions.

1 Introduction

1.1 Design goals

Our goal is to create a procedurally generated platform game, that can easily be extended with new levels, enemies and items. The game should be functional with just some basic classes and files for enemies, rooms etc. More eventual additions should be seamlessly integrated.

1.2 Definitions, acronyms and abbreviations

1.2.1 LibGDX

LibGDX is a java library used in the development of both 2- and 3-dimensional games. It is very extensive and offers a large amount of functionality that is relatively easy to use. Among the functionality there is a create and a render method to run the game-world, a physics engine to control different bodies in the rendered world and a system to easily add sprites and animation to the bodies in the world.

1.2.2 MVC

MVC is an acronym for Model-View-Controller, which is a software architectural pattern commonly used when implementing user interfaces. Essentially, it splits the system into three separate parts, the model, the view and the controller, all which complete tasks unique to each part.

1.2.3 Box2D

Box2D is a physics engine written in C++ and ported to Java, and supported by LibGDX. The engine deals with objects of the class Body. Bodies are placed in an ArrayList in a World object,
and are affected by simulated gravity and collisions between them.

2 System design

2.1 Overview

The application will use the MVC model.

2.1.1 The model functionality
The map, current room, and everything in it except the hero can be accessed in the Map class. The hero is accessed by itself. Both the map and the hero class is using the singleton pattern.

2.1.2 Global look-ups
We use the singleton pattern for the Map class and the Hero class to enable uniform access across the entire application. We also use static access to everything in the class WorldConstants, where we store the world, all the current bodies, and a few constants.

2.1.3 Enemies
Enemies are controlled with simple AIs. The AI is composed of different behavior objects which implement behavior interfaces depending on what kind of behavior it is. All enemies have behavior variables which are set to the specific behavior the enemy should have. This makes it easy to create new enemies with new specific behaviors.

Enemies are created by classes implementing the interface EnemyFactory. This returns an EnemyController containing an Enemy model and an EnemyView

2.1.4 Rooms
All rooms are kept in a single list in the Map class. Each individual room holds all enemies and chests in it, as well as access to a TiledHandler which is responsible for creating and destroying the room. Rooms are created by the RoomFactory class.

2.1.5 Event handling
The event handling within the game is restricted to when different bodies in the world touch, or stop touching. To keep track of this we use a LibGDX class called “Contactlistener”. The Contactlistener class works like an event listener that gets called everytime two bodies in the world either touch or stop touching. When it is called it receives the two bodies that touch or stop touching as input.

2.2 Software decomposition

2.2.1 General
The application is decomposed into the following modules, see Figure X.

- Controller, the control classes for the MVC model and view
- Model, the model classes of the MVC.
    - Creatures, holds all creature related model classes
    - Enums
    - Items, holds all item related model classes
    - Map, holds all map related model classes
    - Factory classes
- Utils, utility classes

    - AIBehaviors, holds all classes used in AI movement
    - WorldConstants, static class with variables deciding gravity and other globally available values.
- View, all view classes in MVC.

2.2.2 Layering

See figure X below. Higher up means higher layer.

2.2.3 Dependency analysis

(STAN, borde fixa alla ringar innan vi lägger in det)

2.3 Concurrency issues

There are no apparent concurrency issues since the game runs on a single thread, apart from destruction of instances of Body in the World class. Instead of destroying and rebuilding the bodies when entering a new room, or destroying bodies tied to enemies when they are killed, all of these events have to be flagged for destruction when the World class is not handling its' physics simulations.

2.4 Persistent data management

The game is made to be played in short sessions, where every session is different from the last. As a result, not much data will be saved. What will be saved are some basic statistics like the number of played sessions and completed games. The number of completed games will decide if certain weapons will have a chance to appear in the current session. This data is preliminarily saved in a simple .txt file.

2.5 Access control and security

This game is a single player desktop application, without any connection to any network. The game will also basically start over every time you open it. This will make any sort of personal account redundant. The program also has no use for personal data, so no security measures needs to be taken into consideration.

2.6 Boundary conditions

The game is started started like a normal desktop application and a session is started pressing the enter key in the game's main menu. One session will last until a) the player character dies, b) the player beats the boss, or c) the player closes the application or presses the escape key and selects "quit session" to end the current session and return to the main menu.

3 References

APPENDIX