System Design Document (SDD) for Blast in the Past

TDA367

Group 21:
Bjarne Gelotte
Jonas Graul Sagdahl
Mattias Johansson

Version: 2.0
Date: 2015-05-31
Authors: Bjarne Gelotte, Jonas Graul Sagdahl, Mattias Johansson

This version overrides all previous versions.

1. Introduction

1.1 Design goals

The project structure should be categorized in different packages following the MVC-model. In essence the group strives to create a modular project as possible. In order to achieve this, many classes are based on interfaces and use dependency injection.

The packages should be loosely coupled and have a strong coherence. If a package were to be deleted, the rest of the application should still work fine without too much effort. Apart from the MVC-packages there will also be other packages such as utils and tests.

1.2 Definitions, acronyms and abbreviations

HP - Health Points, signifies the amount of damage a character can absorb before dying.
2D - Two Dimensions/Two-dimensional
AI - Artificial Intelligence
Ammo - Ammunition
Boss - A particularly tough and dangerous enemy
Power-up - An item which gives the PC certain positive abilities like increased damage or faster movement for example. Expires after a short time (less than a minute).
NPC - Non-Player Character, a character that is not being controlled by the user.
PC - Player Character
GUI - Graphical User Interface

2 System design

2.1 Overview

The application structure is based on the MVC-model. It also includes a package for utilities, a package for data handling and a package for testclasses. The application uses the framework LibGDX, mainly for graphics and music. The Vector2-class from LibGDX is used for object movement and is the only part of the framework associated with the model.

2.1.1 Running the application

The main class is DesktopLauncher in the desktop package. It instantiates the BPController and sets it as the running LibGDX application. The application runs by LibGDX calling the render method in BPController. Contradictory to the method name it doesn't render anything, it just calls update methods in the model and view packages.

2.1.2 The model functionality

The models functionality is divided into several packages. Each type of game object is contained within a package and implements an interface. BPModel, which works as the main model class, handles the instantiation of game objects, as well as updating them and eventually removing them.

2.1.3 Input handling

The input is handled first by InputHandler in the controller package. For convenience, InputHandler uses LibGDX to listen to input. If a method in InputHandler is called, e.g. keyDown or mouseMoved, InputHandler calls a method with the same name in BPController. For example, if keyDown is called in InputHandler, it sends the keycode forward to keyDown in BPController. BPController in it's turn checks which controller is currently active and sends the keycode forward to that controller. The active controller tells the model and view what to do, depending on the input.

2.1.4 Levels

Levels are a part of the model and are managed by LevelManager. Only one level has been implemented, but the structure is made so that it should be easy to add more levels (using interface and abstract class).

2.1.5 Weapons and Projectiles

The Weapon class handles firing of projectile and reloading. Projectiles are created when a Character fires it's weapon. Before they are instantiated in the weapon class, the projectiles available are only represented by an int (ammunition).

2.1.6 Characters

Characters have a list with the projectiles currently in the world that they have fired. It might seem contrived (and it is) but there are good reasons for it. Basically it all comes down to the fact that the gameplay is more important than having a realistic model. What this allows the model to do is to selectively ignore collisions between projectiles and certain characters. The biggest "issue" this solves is the issue of the player being shot by projectiles which they fired (by running into them for example). While it is unlikely to be a problem with the current implementation, if someone wanted to implement a weapon which fires slower moving projectiles and/or increase the potency of the movementspeed power-up, the issue would be more likely to arise. There are other ways this list could come in handy, like enemies being able to ignore collisions with projectiles fired by other enemies.

Characters use the observer pattern when they fire their weapon or when they die. They notify observers that something has changed.

2.1.7 Ammunition

Ammunition is used to represent several projectiles which the player can pick up and add to his/her ammunition count for a specific weapon. Ammunition has been implemented this way because the ammunition for a certain weapon is simply stored as an integer. A projectile is only created when the weapon is fired, implementing the ammunition class to work in a similar fashion is therefore logical. Having an ammunition class and a projectile class instead of making the projectile class fulfill two roles also avoids any problem with deciding what should happen when a character collides with a projectile (take damage or pick up ammunition). As mentioned it also separates concerns, each class fulfills one role.

2.1.8 Power-ups

Each time the model updates, the power-up bonuses are applied. The bonuses are applied with functions which adds a bonus value to the standard player values for movement speed and such. Before the player is updated the bonuses are added and the power-ups bonuses are then removed after the player has been updated. The reason for implementing power-ups this way is because removing power-ups becomes simple (they are simply removed from the list of active power-ups). It also avoids issues which can occur when picking up a new weapon while a power-up that affects weapons is active.

After an enemy is killed they have a chance to drop a power-up which the player character will be able to loot. Once it's looted (simply by walking onto it) the power-up is activated for a limited time.

2.1.9 Loot

Enemies can drop loot when they die. The dropped objects are sent to BPModel with the use of propertyChangeEvents and BPModel adds the objects to a list which contains all loot. The loot system takes advantage the Strategy pattern which simply means that loot implementations are not tied to enemy classes, the same class of enemies can have different types of loot. Enemies contain an object of the type LootInterface and a concrete implementation of the LootInterface is sent as a parameter in the enemy constructor.

2.1.10 Factories

The project follows the Factory Pattern (MSDN, 2002) where the idea is to make the creation of new objects as general as possible, by hiding exactly how the object is created. It is used for several generic entities such as weapons and characters.

2.1.11 Utilities

Since no use of framework code is allowed in the model package, the project follows the Adapter pattern to retain some of the logic needed in the game. The utility package contains adapter classes for Rectangle and CollisionLayer from the LibGDX library.

2.1.12 View, GameStates

In the view package lies all graphical representations of the model, including characters, projectiles and GUI.

The project uses State Design Pattern when handling game states. There are five different game states (or view states): MainMenu, PlayState, GameOverState, InGameMenu and HighScoreState. The active state is controlled by GameStateManager. The view uses the observer pattern, i.e. it's listening for changes in the model. When new objects are created (e.g. Projectile, Ammunition, etc.) the view creates the corresponding view object.

2.1.13 Testing

To achieve good test classes, most model classes are based on interfaces in order to use dependency injection. This means that mock classes can be implemented when needed, to achieve unit tests. The goal is to test only one class at a time, without strong dependencies to another.

2.2 Software decomposition

2.2.1 General

The application is decomposed into the following packages:

- **model** - contains the core with all logic. Model part of MVC.
- **view** - graphical representation of the model. View part of MVC.
- **controller** - handles input and delegate tasks to model and view. Controller part of MVC.
- **utils** - contains adapters for rectangle and map (LibGDX), which are used by model
- **data** - classes for i/o, data handling
- **tests** - test classes mostly to test the model's functionality

2.2.2 Decomposition into subsystems

High scores are handled through filestreams using java.io in the HighScoreHandler class. On initialization, the program reads from a local .sav file containing the game high scores. If there's no such file, the program will create one with default values.

If a player manages to get enough points to get into the high score chart, the points and a three letter name will be written to the .sav file.

2.2.3 Layering

The layering is visualized in Figure 1 and Figure 2. Packages higher up in a figure are at a higher level than the packages below.

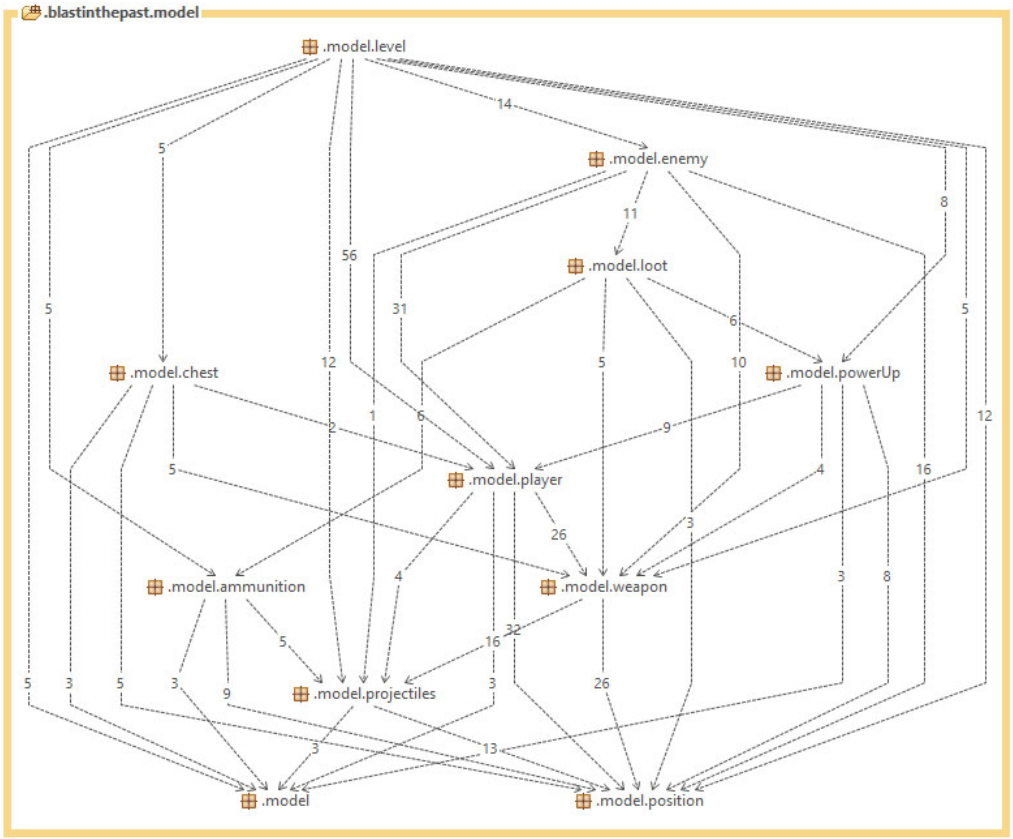

Figure 1. Model package layering.

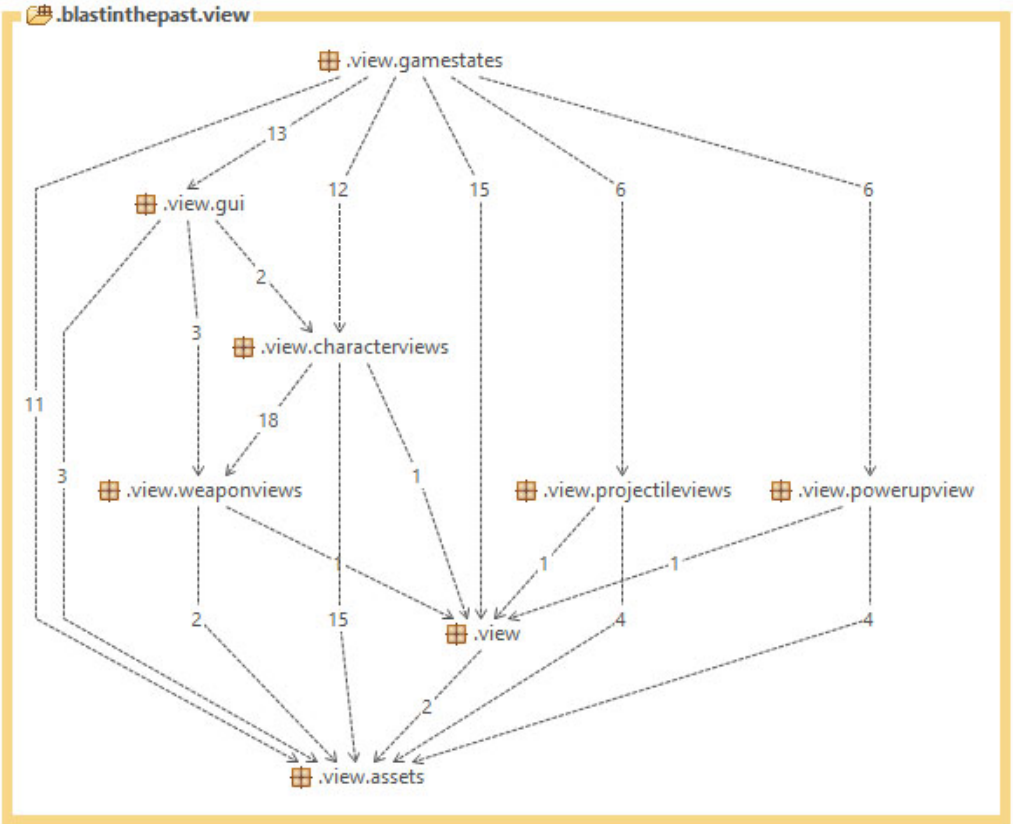

Figure 2. View package layering.

2.2.4 Dependency analysis

As shown in *Figure 1*, *Figure 2* and *Figure 3*, there are no circular dependencies between packages.

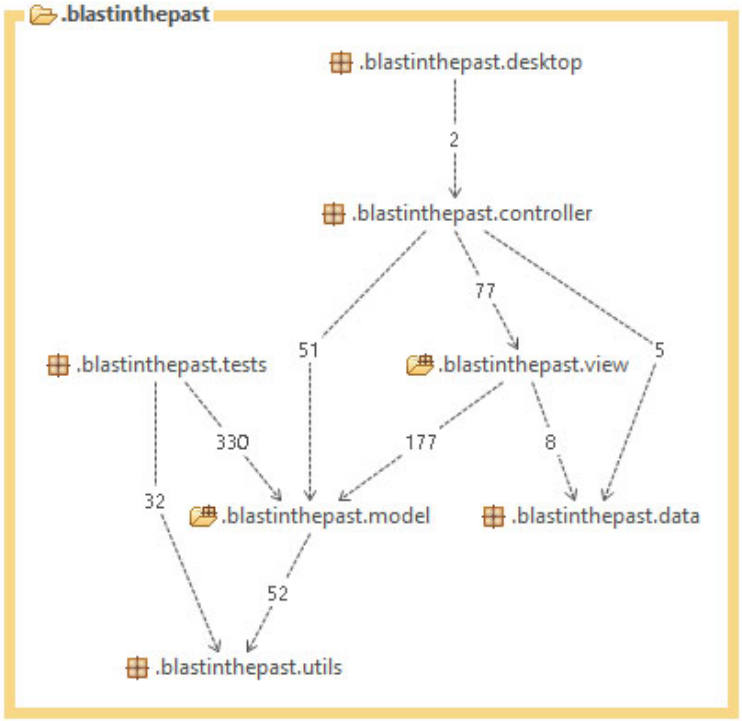

Figure 3. High level package structure and dependencies.

2.3 Concurrency issues

The program runs on a single thread so no concurrency issues.

2.4 Persistent data management

HighScoreHandler handles filestreams through java.io, see more under *2.2.2 Decomposition into subsystems*.

2.5 Access control and security

N/A

2.6 Boundary conditions

N/A. The application is launched and exited as a normal desktop application.

3 References

MSDN, 2002. *Factory Pattern*. Retrieved from https://msdn.microsoft.com/en-us/library/ee817667.aspx

MSDN, 2003. *Model-View-Controller*. Retrieved from https://msdn.microsoft.com/en-us/library/ff649643.aspx

Wikipedia, 2015. *Graphical user interface*. Retrieved from http://en.wikipedia.org/wiki/Graphical_user_interface