System Design Document for AwesomeGame

Version: 1.0

Date: 23/5-18

Author: Therese Sturesson, Philip Nilsson, Farzad Besharati, Linus Wallman

This version overrides all previous versions.

1. Introduction

AwesomeGame is a game designed to run on a desktop PC. The game itself is designed as a top-down adventure-based game where you run around in a world and defeat enemies with the help of items. This document describes the core and construction of the game application, AwesomeGame, that is specified in the requirements and analysis document (RAD).

1.1 Design goals

The goal with *AwesomeGame* was to make an open-world that runs on desktop PCs that functions like an old-school adventure game (Zelda 1, as an example). We wanted the player to use different weapons and items to defeat different types of enemies and win the game by defeating a final boss.

1.2. Definitions, Acronyms and Abbreviations

Technical definitions:

- JRE: Java Runtime Environment
- PE: Portable Executable, see references
- Jar: package file format used to aggregate Java class files

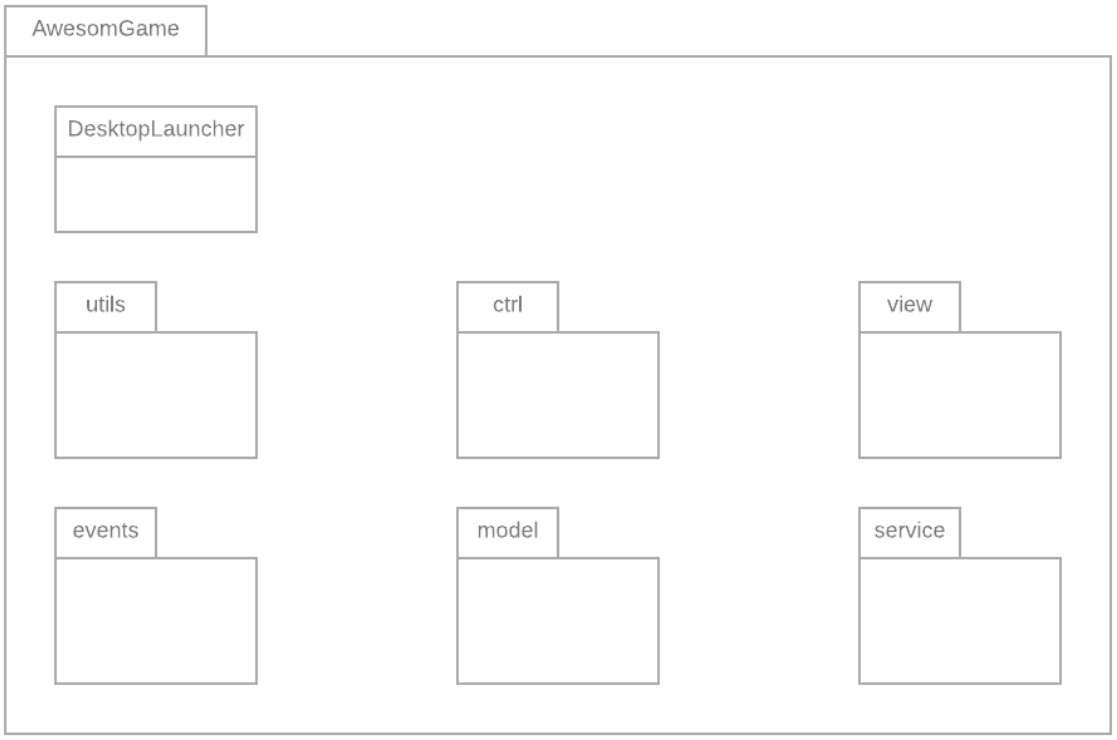
Most definitions and terms regarding the core game:

1. Player: Representative of the person playing the game
2. AI: Representative of entities that aren't the Player
3. World: The environment in which the game takes place
4. Tile: An even-sided square, the world consists of these
5. Game Object: All renderable objects in the world, this could be a Character or something like a bomb.
6. Character: An entity in the world, could be the AI or the Player
7. Player Character (PC): A Character representing the Player
8. AI Character: A Character representing the AI

2. System Architecture

Only the machine that the program is installed on is relevant for running the program, there are no external connections or programs required. This means the application run on a single desktop computer. To run the application, the user will need to have JRE installed on the machine. To start the application you need to click on a specific file you have when you get the game. To stop the application you need to either click on the exit button in the main menu screen or click on the red button with the cross in the right upper corner on the window. The application will use an EventBus (instead of Observers).

The application is decomposed into the following top level packages (arrows for dependencies).



(SÄTT DIT PILAR)

DesktopLauncher: Are the main- and entery class for the game
event: Contains classes with events for the eventBus
view: Contains several different views used in the GUI
core: Contains all the model classes
ctrl: Contains all the controller classes that works between the model and view
service: Handles classes with LibGDX connections
util: Have general utilities classes that can be reused whenever you want

2.1 General observations (Linus)

Flow

(Any general principles in application? Flow, creations, …
The most overall, top level, description of the system. Describe the high level overall flow of some use case)

Dependency Analysis

screenshot from STAN java would be nice here

(Och sedan en liten analys över bilden och dependencies vi har)

3. Subsystem Decomposition

AwesomeGame is divided into 6 packages

- Controllers
- Model
- View
- Events
- Services
- Utils

3.1. Application

3.2. Model

This package contains classes describing objects that have no connection to other frameworks, such as LibGDX. The logic for the actual game engine resides here.

- GameObject
    - An abstract superclass used mostly when rendering non-permanent items such as bombs. All renderable items are derived from this class. Has a Facing, a Position and a name.

- Characters
    - This package contains a class for the player; PlayerCharacter. One class for enemies; Enemy, and one for Boss enemies; BossEnemy. All of these classes are subclasses of the abstract superclass Character. Character contains fields such as baseDamge, health and maxHealth as well as a position.

- Item
    - This package contains three abstract superclasses; BaseInstant, BaseConsumable and BaseItem. These classes contain fields and methods pertaining to each type of item: Consumable, Instants and Permanents, the most important being the use()-method. Below is an example from BaseItem:

```
public abstract class BaseItem implements Item {
   protected String name;

   public String getName() {
       return name;
   }

   public abstract void use(Position pos, Facing facing);
}
```

Contained here are also a few concrete subclass implementations such as Sword, Bomb and heart. Here's the Sword implementation as an example:

```
public class Sword extends BaseItem {
   public Sword() {
       name = "Sword";
   }
  /**
   * Swing the sword side to side on the three blocks in front of you
   * @param pos current player position
   * @param facing current player facing
   */
   @Override
   public void use(Position pos, Facing facing) {
       List<Enemy> enemies = new ArrayList<>();
       int x = pos.getX(), y = pos.getY();

       if (facing == Facing.SOUTH)
           enemies = MapSegment.getPlayerTargets(x-1, y-1, x+1, y-1);
       else if(facing == Facing.NORTH)
           enemies = MapSegment.getPlayerTargets(x-1, y+1, x+1, y+1);
       else if(facing == Facing.EAST)
           enemies = MapSegment.getPlayerTargets(x+1, y+1, x+1, y-1);
       else if(facing == Facing.WEST)
           enemies = MapSegment.getPlayerTargets(x-1, y+1, x-1, y-1);

       for (Enemy enemy : enemies) {
           enemy.decreaseHealth(5);
       }
   }
}
```

- Facing
    - An enum that represents a characters current facing in the world. Possible values are equivalent to cardinal directions.

- Inventory
    - This class represents the items that are carried by an instance of PlayerCharacter. It contains methods which allow the PlayerCharacter to manipulate items. Instansiated as *inventory* in PlayerCharacter

- Factory
    - This package contains all the factories (for an example CharacterFactory) in the model.

- MapSegment
    - This class processes small chunks of the map defined as a 32x16 matrix of positions. Holds a list of characters in the current segment and performs different checks to see if enemies in the current segment are allowed to attack and checks for targets. Also makes sure that enemies in the segment stay within the current segment. And so forth...

- Position

    - A class that represent a position in the world. Has an x and y value. This class are used by everything that is placed in the world.

- WorldPosition
    - A class that contains a position and a string (for the map name) and is used for example when you need to return both in a method.

3.3. Controllers

This package contains all the controller classes that works between the model and view.

- GameCtrl
    - This controller takes in all the input from the keyboard and delegates the input to either another controller or to the model.

- PlayerCtrl
    - This controller does all the checks for the player when it tries to move. The controller gets an input from the GameCtrl and checks if the player can move to that position. It also delegates the input from the GameCtrl to the PlayerCharacter class in model when the player tries to use an item.

- EnemyCtrl
    - This controller does all the checks for the enemy when it tries to move or tries to attack.

(FINNS FLER CONTROLLERS)

3.4. View

This package contains several different views used in the GUI.

- GameScreen
    - In this screen you play the game. You can move around you character on the map, use items, fight enemies and kill the boss. This screen handles the rendering for the game.

(FINNS FLER CONTROLLERS)

3.5. Services

The Service package contains classes that is modell classes but have connections with LibGDX. So basically they have code with game logic but since they have connection with

the framework they are not allowed to be in the model packages and therefore are moved to the service package. The packages contains four classes.

- AssetManager
    - The class manage game assets like items, textures and animations.
- MapStorage
    - Stores all the maps that are used in the game.
- Tiles
    - Does different kinds of checks on the map and also populate the map with enemies.
- WorldMap
    - This class holds the current map that is used and sets a new map when needed.

3.6 Events

The Events package contains all the event classes that is used through the eventbus.

3.7 Utils

The Utils package have general utilities classes (that is not really game logic) that can be reused whenever you want. This packages contains two classes.

- AwesomeClock
    - The class is a simple clock class. Are often used together with the timer class to prevent an event to occurring too often.
- AwesomeTimer
    - The class is a simple timer class. Are often used together with the clock class to prevent an event to occurring too often.

4. Persistent Data Management

All persistent data is stored in flat test files, the files are:

- .jpg/.png containing images that we then convert into textures
- .pack
- .tsx are xml files that contain information regarding certain tiles, such as if they’re solid or if they teleport you somewhere, it also list tile ID’s
- .tmx files contain the layout of maps. By importing different tilesets you use tile ID’s in a giant matrix to create your map

5. Access Control and Security

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

6. References