System Design Document for AwesomeGame

Version: 1.0

Date: 27/5-18

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

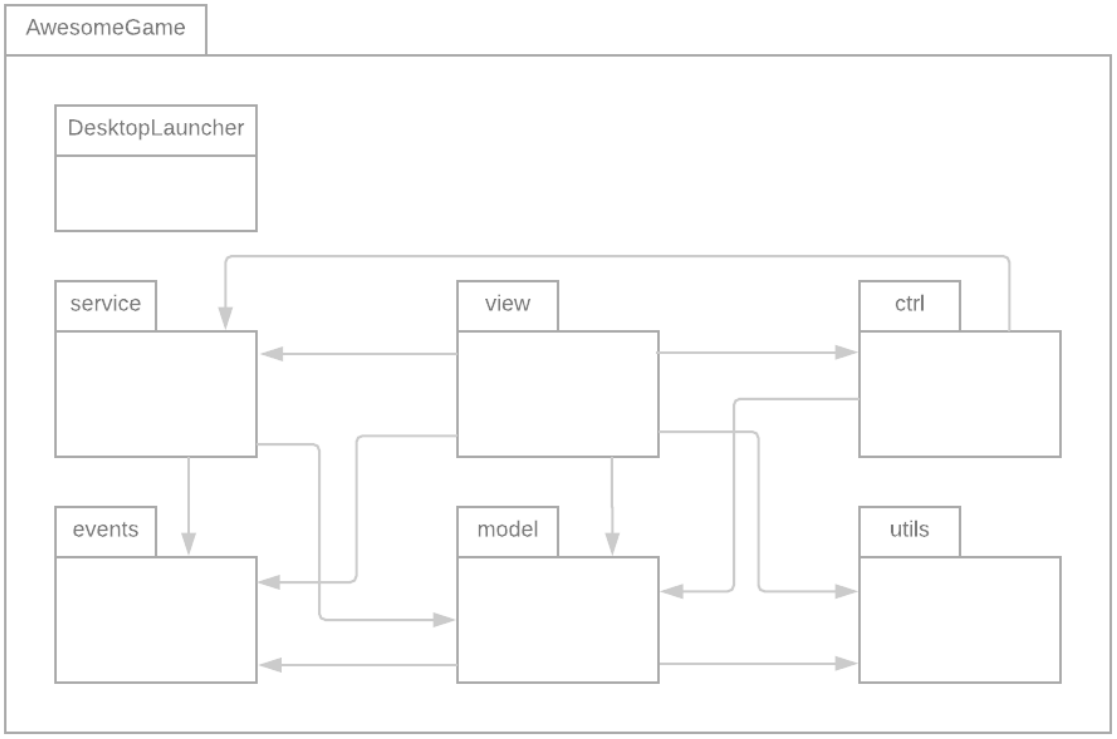
Most definitions and terms regarding the core game:

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).



DesktopLauncher: Contains the main method and serves as the entry point for the application
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 work between the model and view
service: Handles classes with LibGDX connections
util: Contains general utility classes that can be reused whenever you want

2.1 General observations

Flow

The flow of the application starts in the class DesktopLauncher which is a class that is responsible for setting up the GDXWrapper. From the GDXWrapper, parts of the model are initialized and a starting screenstate is set to the mainmenu.

The model functionality

A game class called AwesomeGame will be passed around through the different screenstates which contains game logic.

Maps

Tilemaps are stored using the service class MapStorage. Maps can also be loaded on the fly and are accessed with ease from this class.

Enemies

Actions taken by enemies such as monsters and Bosses are handled by the class EnemyCtrl. Enemies are able to perform two actions; Attacking and moving. Checks related to these actions are also performed here.

Player character

The actions of the character that represents the player are performed using the keyboard. Movement is tied to the arrow keys. The item slots are each tied to a specific key, A and S respectively. Keyboard input is read through the class GameCtrl and is then delegated to PlayerCtrl which calls the relevant methods in PlayerCharacter.

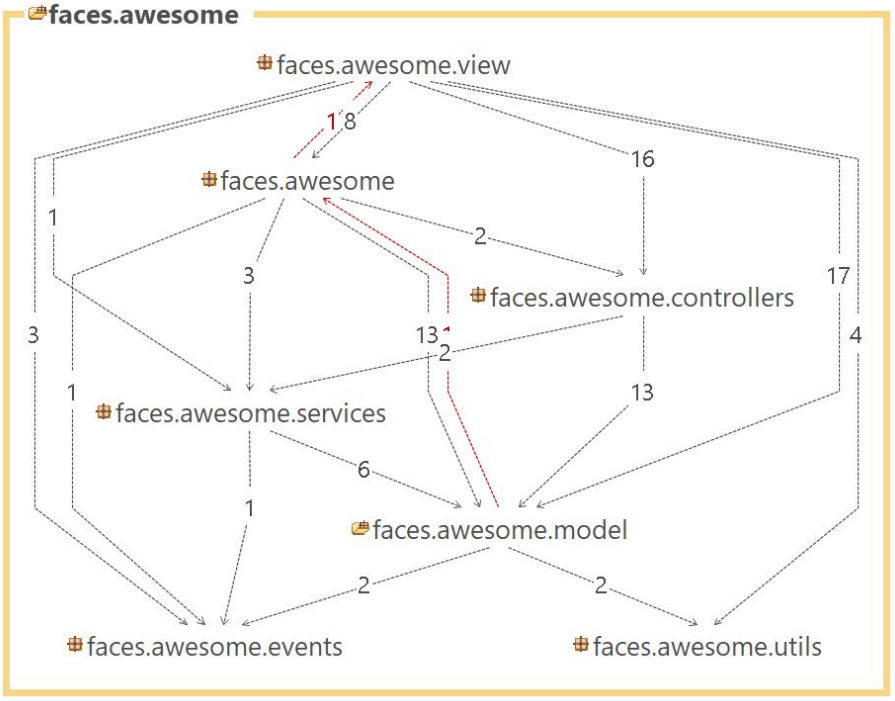
State notification

User input is handled using LibGDX's internal "inputprocessor". When a button is pressed or when mouse is pressed down the class that implements the inputprocessor will handle the input and delegate it to the right controller.

Lookups

All asset related lookups are done through an assetmanager which holds all assets and can be fetched using a unique key.

Dependency Analysis



By using the program STAN you can do a dependency analysis on the program. Here you can see that the program has two circular dependencies. One from the awesome folder to the view and on from the model to the awesome folder. Since the first view (main menu screen) has to be initialized somewhere in the model when the game starts it is a dependency that is hard to remove. The other one (model to awesome folder) are when the mapsegment class (in the model) needs to have the player and the world so the easiest way was to let the class get the gdxwrapper class and from there mapsegment gets the player and the world. It is not the best solution but for the moment it is the only solution we had.

3. Subsystem Decomposition

AwesomeGame is divided into 6 packages

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

3.1. 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.

- DropTable
    - This class is called when an enemy dies and sometimes drops an item like a heart or a bomb the player can pick up.

- 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 factories (for an example CharacterFactory) that are used to instansiate objects 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 this list are allowed to attack. It does so by performing target checking. 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.

3.2. Controllers

This package contains all the controller classes that work between the model and the different views.

- GameScreenCtrl
    - This controller reads all input from the keyboard and delegates it 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 GameScreenCtrl to the PlayerCharacter class in the model when the player tries to use an item.

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

- ScreenSwitcher
    - Handles the switching of screen states

- ScreenSwitcherListener
    - An observer interface for switching between screens.

3.3. 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.

- MainMenuScreen
    - This is the screen from which the user can access different functionalities such as displaying game settings, displaying credits, and starting the game.

- CharacterView
    - Handles character animation by tying animations to different character states such as running or standing.

- GameObjectView
    - Handles the rendering of GameObjects. Necessary to use during some instances of item rendering.

- GameOverScreen
    - Handles rendering of the Game Over screen. When you lose the game this screen will be presented.

- GameWonScreen
    - Handles rendering of the Game Won screen. When you win the game this screen will be presented.

- CreditScreen

        - A credit screen that will be presented after the game won or game over screen.

- ItemView
    - Handles the general rendering of items

- ScreenRepository
    - Handles the instantiation of the different types of available game screens.

3.4. Services

The Service package contains classes that pertain to the model but contain fields related to LibGDX. So basically they have code with game logic but since they are connected to the framework they are not allowed to reside in the model packages and are therefore moved to the service package. The packages contains four classes.

- AssetManager
    - The class manages game assets such as items, textures and animations.
- MapStorage
    - Stores all the maps that are used in the game.
- Tiles
    - Performs different kinds of checks on the map and also populates the map with enemies on predesignated spawn points.
- WorldMap
    - This class holds the current map that is used and sets a new map when needed.

3.5 Events

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

3.6 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
    - This class represents a simple clock. Is often used together with the timer class to prevent an event from occurring too often.
- AwesomeTimer
    - This class represents a simple timer. Is often used together with the clock class to prevent an event from 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

STAN java