System design document for Adlez

Version:
Date:
Author:

This version overrides all previous versions.

1 Introduction

1.1 Design goals

The application must follow the MVC design pattern. The design should therefore be modular so that different parts of the application can be exchanged without affecting other parts. In particular, the visual representation must be completely separated from the game logic and data. Also, subsystems such as file handling should be communicated with using interfaces so that also they can be substituted without ramifications on the rest of the application.

The model package should also strive to have as loose coupling as possible within itself. It should also be the basis for all tests of the application.

1.2 Definitions, acronyms and abbreviations

- MVC: An architectural pattern that divides the application into three parts, the Model containing the logic, the View containing the graphical user interface based on the Model and the Controller that handles the user's input and updates the Model.
- Screen:
- World object: An object in the game such as a wall, obstacle, character etc.
- Area: An area in the game which contains world objects.
- Player: The character that is controlled by the user of the application.
- LibGDX: Game-development application framework.
- JSON: Short for JavaScript Object Notation. Data format used for storing information in an organized and easy understandable manner.
- Game loop: Common design pattern for games. Involves a method that continuously loops and updates game logic and data, graphics, acknowledges user input etc. One (1) game loop is one loop of this method.
- DeltaT: Short for delta time, the time since the last update of the game.

2 System design

2.1 Overview

The application is based on a MVC structure. The application is split into different screens that are managed by a screen manager. The screen manager loads the data for the individual screens when they are to be displayed.

The game screen is the screen that handles the core gameplay of the game and can be seen as a sort of top level controller. It contains the game loop which is responsible for updating the game model and rendering the view. The update is done through controllers for either individual objects such as an enemy or families of objects such as obstacles. User input is handled by the PlayerController class which registers what key was pressed and proceeds with performing appropriate actions such as telling the player model to move as well as saving the game.

2.1.1 Model structure

The game is split into areas which hold the state for all the game objects except for the player. The area handler holds the state for each area. The state for the player is held by the class Adlez that also manages the state of the player together with the current area being played. All changes of the model therefore goes through the class Adlez. The game screen retrieves the game data from Adlez. Adlez is the top level class for the model and since only one instance is needed it appropriately uses the singleton pattern.

2.1.2 The model functionality

The model’s API can be accessed through the Adlez class. To avoid a very large API in a single class, methods are spread out throughout the model and can all be accessed through Adlez.

2.1.3 Areas

Adlez world is divided up in different smaller areas where each area contains the game’s different objects such as enemies and obstacles. The areas are handled through an area handler which handles every object that an area needs. With the help of the area handler Adlez can save the game effectively since it handles every object.

The different components to fulfill an area, such as obstacles, are implemented separately as model and view. While the area handler handles all the models (obstacles, enemies…), the game screen will render the objects graphics.

The areas maps are drawn using the program Tiled, which is a tile map maker. We are using it as some sort of background, so it will not have any logical affections.

When all components are ready, Aldez has a setup for each area that sets all objects on their place.

2.1.4 Collision handling

Collisions are handled at the end of each game loop by a collison handler, more specifically the CollisionHandler class. Only one instance is needed of this class too so it also uses the singleton pattern. Each object in the world inherits from the WorldObject class, which contains position and size information about the object. The CollisionHandler handles collisions by seeing all objects in the world as WorldObjects and then checking for each object if it overlaps with another. If there is an overlap, a collision has happened. The collision handler then proceeds with invoking the object in question’s *onCollide(Collidable other)* method which then performs the appropriate actions depending on what kind of object *other* is.

The only objects that don’t get checked for collisions at the end of each game loop are characters. This is due to complications that occured when characters moving in intercardinal directions (e.g. northeast) collided with other objects. Collision checking for characters is instead done using the observer design pattern. The collision handler first registers itself as an observer to every character. Directly after a character has moved it notifies its observers, which includes the collision handler.

2.1.5 Attacks

When a character attacks, an attack object is created and added to the world as any other world object. Say for example that the player performs a melee attack. This is handled by the creation of an attack object that gets situated right in front of where the player is facing. If the collision handler then finds that, say, an enemy has collided with the attack, the enemy’s *onCollide(Collidable other)* method will be invoked with the attack object as the parameter. The enemy’s method will register that the collision was with an attack from the player and do what needs to be done, in this case decrease the enemy’s health depending on the attack’s damage.

A strength of this way of handling attacks is that an attack can go on and live their own lives in the worl, so to speak. An attack representing a bullet for example could travel in the world independent of what else is happening. If it then eventually collides with something, the onCollide method will perform the appropriate actions.

2.1.6 Interactions

Interactions are handled in a similar way to attacks. If the player presses the interaction button, an interaction object is created and placed in front of where the player is facing. If the interaction doesn’t collide with anything that can be interacted with, nothing happens. If it does though, say with a chest, the onCollide method of the chest will handle the transaction of the chest’s contents to the player.

2.1.7 Movement

Movement incorporates deltaT (delta time) so that even if the game’s update frequency drops, a character will still move the same distance.

2.1.8 Enemy AI

Enemy AI is for now handled very simply in that an enemy will start moving towards the player if the player gets within a certain proximity. The enemy will then attack if it gets close enough to the player. The frequency at which an enemy can attack is handled by the time variable *cooldown*. This variable gets incremented with the value of deltaT every game loop. Only if the

the variable has reached a certain limit is the enemy allowed to attack. When the enemy has attacked, the variable gets reset and the enemy has to again wait for a while until it’s allowed to attack.

2.1.9 Enemy factory

The enemy factor’s purpose is to have easy access to different enemies where each enemy has stats assigned to it. This design saves much time and space for implementation of enemy related components.

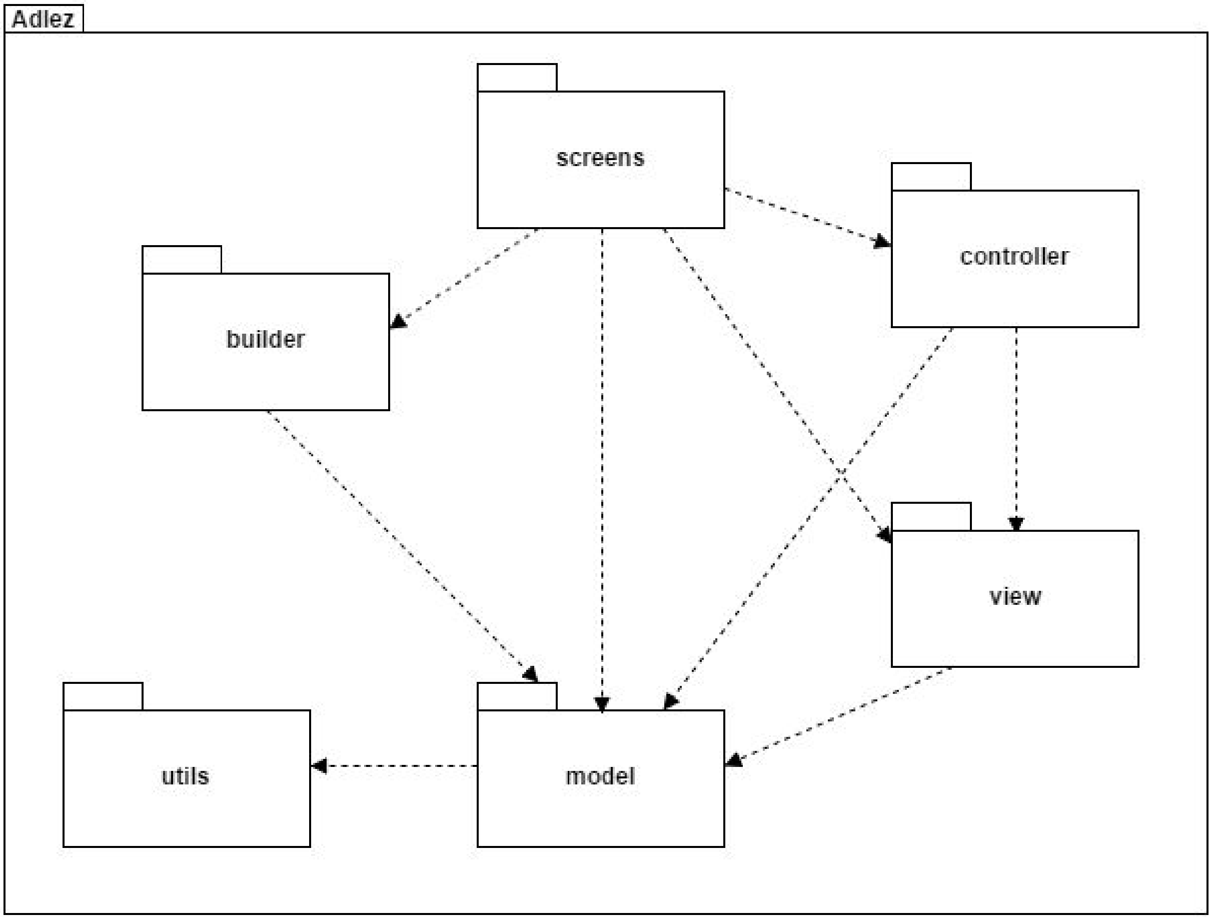
2.1.10 Saving/loading game

2.1.11 Area switching

2.2 Software decomposition

2.2.1 General

The application is decomposed into the following top level packages.



- Screens includes the ScreenManager and all the screens for the different views of the game such as the main menu, the game screen, the inventory.
- Builder includes the AreaHandler and the AreaBuilder which handles the saving and loading of the game.
- Model, controller and view includes what there names say.
- Utils include general things that possibly can be reused.

2.2.2 Decomposition into subsystems

The system that handles the saving and loading of the game is a subsystem with the following interface.

```
public interface AreaIO {
   public void savePlayer();
   public void saveAreaHandler();
   public IPlayer loadPlayer();
   public AreaHandler loadAreaHandler();
}
```

2.2.3 Layering

NA.

2.2.4 Dependency analysis

2.3 Concurrency issues

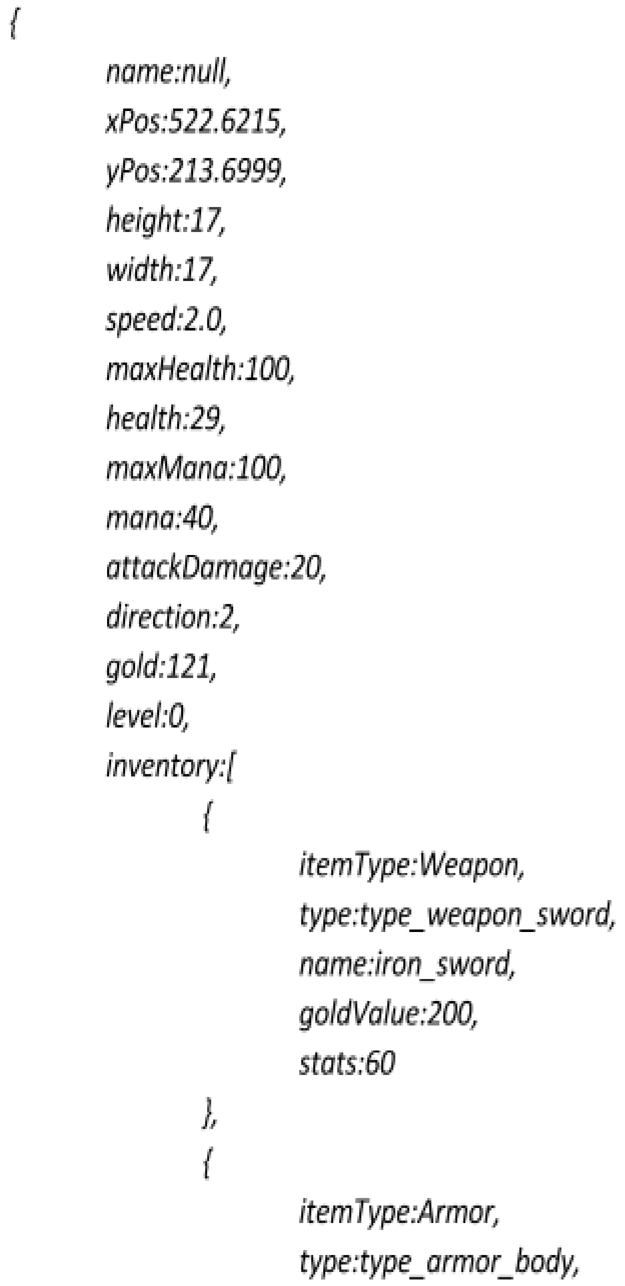
The game runs on a single thread.

2.4 Persistent data management

All the data for the game is located in two classes, the AreaHandler which has the current state for each Area and the Player that has the current state for the player. The changeable data for these are stored in two JSON files,

- One file containing the data of each area such as the enemies' locations, the obstacles and the chests.
- One file containing the data for the player, such as health and items.

The persistent data management can easily be exchanged since it uses an interface with the methods, saveAreaHandler(), loadAreaHandler(), savePlayer() and loadPlayer().
Part of an example of the data saved from the player can be seen below.



```
{
        name:null,
        xPos:522.6215,
        yPos:213.6999,
        height:17,
        width:17,
        speed:2.0,
        maxHealth:100,
        health:29,
        maxMana:100,
        mana:40,
        attackDamage:20,
        direction:2,
        gold:121,
        level:0,
        inventory:[
                {
                        itemType:Weapon,
                        type:type_weapon_sword,
                        name:iron_sword,
                        goldValue:200,
                        stats:60
                },
                {
                        itemType:Armor,
                        type:type_armor_body,
```

2.5 Access control and security

Only one user and one kind of access and no need for security for the application.

2.6 Boundary conditions

No need, the application launched and exited as normal desktop application.

3 References

- LibGDX: https://libgdx.badlogicgames.com/
- MVC: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
- JSON: http://www.json.org/

APPENDIX