System design document for Prjkté

Version: 2.0
Date: 2016-05-29

Autors: Arvid ”Alle” Björklund
Axel ”Mouseman” Bergrahm
Frej ”Täcket” Karlsson
Ludvig ”Frej” Ekman

This version overrides all previous versions.

1 Introduction

1.1 Design goals

The design must be loosely coupled to make it possible to switch different components like physics and graphics engine. This also makes it easier to maintain, since you will be able to rewrite a part of the program without breaking anything else. It should also be easy to write tests, since we are doing a test-driven development.

1.2 Definitions, acronyms and abbreviations

- **Game loop** – One iteration of the game were for example: A player moves or picks up an item.
- **Game object** – An object that appears on the screen, can be anything from a dead cat to a raging smurf.
- **Character** – Extends from Game object. Is a controlable object that can deal damage and move.
- **State** – What current action a character is doing, i.e. punch, walk, jump.

2 System design

2.1 Overview

The game will use something similar to a MVC pattern. On the top we have a game loop that constantly updates itself. The game loop tells the Game class to update itself. We have several different controllers (scenes) that handels different usecases. For example selecting characters and fighting in an Arena. Arena also has a subcontroller handling more complex use cases. Most models extends GameObject, this includes playable characters and items which have similar characteristics.

The view paints the different gameobjects and background (the 'Map').

2.1.1 Aggregates

The program consists of several different root classes that handels the communication. The communication between these all pass through the top class 'Game'.

Other controllers has their own models. Arena also has a separate subcontroller that has responsibility for controlling the characters and their movement.

2.1.2 Unique identifiers and global look-ups

Imgage files used for sprites and animations use their filepath as an identifier. Also concrete implementations of Characters and Maps have unique names since they are used as an identifier when using their coresponding factory classes.

2.1.3 Events paths

As earlier described the game is updated from a gameloop that updates itself. This loop then tells the game to update itself which in turn tells the active scenes controller to do the same. The Scenes check if relevant keyboardkeys are pressed via the Input class. After this it updates the models and tells the view to paint both the map and gameobjects present.

The models are updated both by setters and by updating the models certain steps, not caring what state the models currently are in.

2.2 Software decomposition

2.2.1 General

Package diagram. For each package an UML class diagram in appendix.

2.2.2 Decomposition into subsystems

Both the AssetsManager, which handels creating and loading Textures and animations, and the KeyInput, that checks if a certain key has been pressed, can be viewed as being their own subsystems.

```
public interface IAssetsManager {

    public void addAnimation(AnimationObject filename);
    public void addTexture(String filename);

    public Object getTexture(String filename);

    public Object [] getAnimation(String filename);

    void finishLoading();
}

package com.saints.gamecode.interfaces;

import com.saints.gamecode.Direction;

public interface IKeyInput {

    boolean isKeyPressed(Direction direction);
}
```

2.2.3 Layering

The top layer is the game loop, responsible for updating the game constantly. Under that is the Game, which is responsible for updating and selecting the active scene. The scene layer compose of the character and mapSelectControllers aswell as the arena. The arena has the sub controller that handels Characters aswell as the healthbar.

2.2.4 Dependency analysis

To combat dependency issues there are observable patterns in use between the scenes and Game. Charactercontroller is also observable and is being observed by Arena.

CharacterController also shares a list with Arena where all Entities are contained. This is since Arena handels the view and need to know what objects CharacterController wants to be in the view.

Thus all classes and packages have a clear top down command chain and there are no circular dependencies.

2.3 Concurrency issues

NA

2.4 Persistent data management

NA

2.5 Access control and security

NA

2.6 Boundary conditions

NA

Appendix

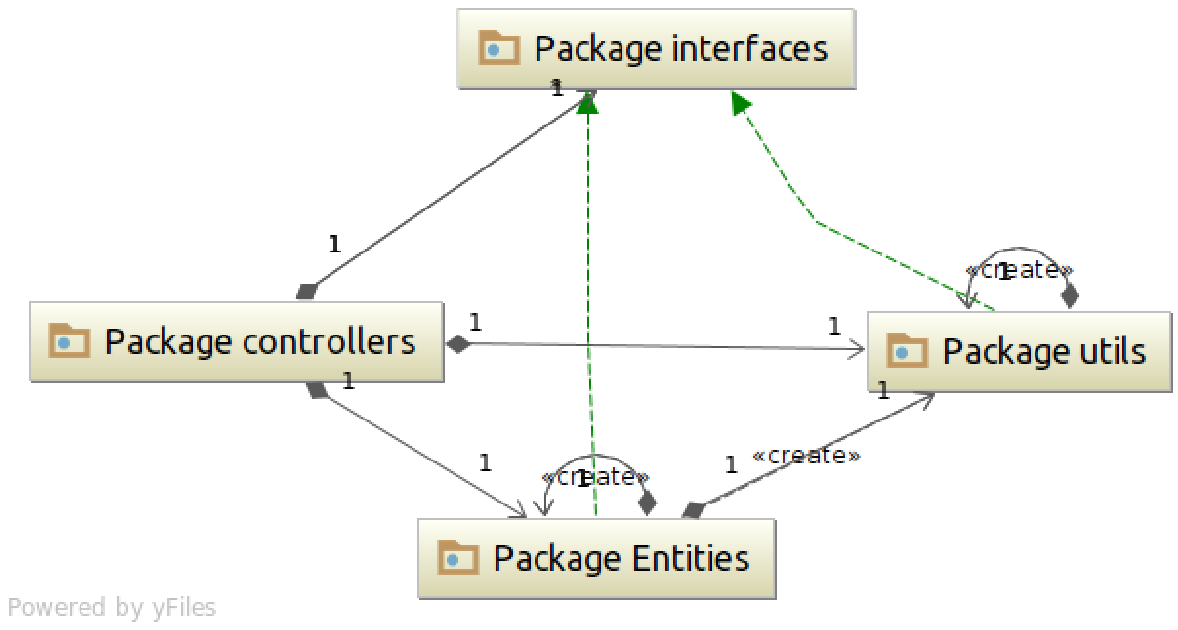

Figure 1: Package diagram.

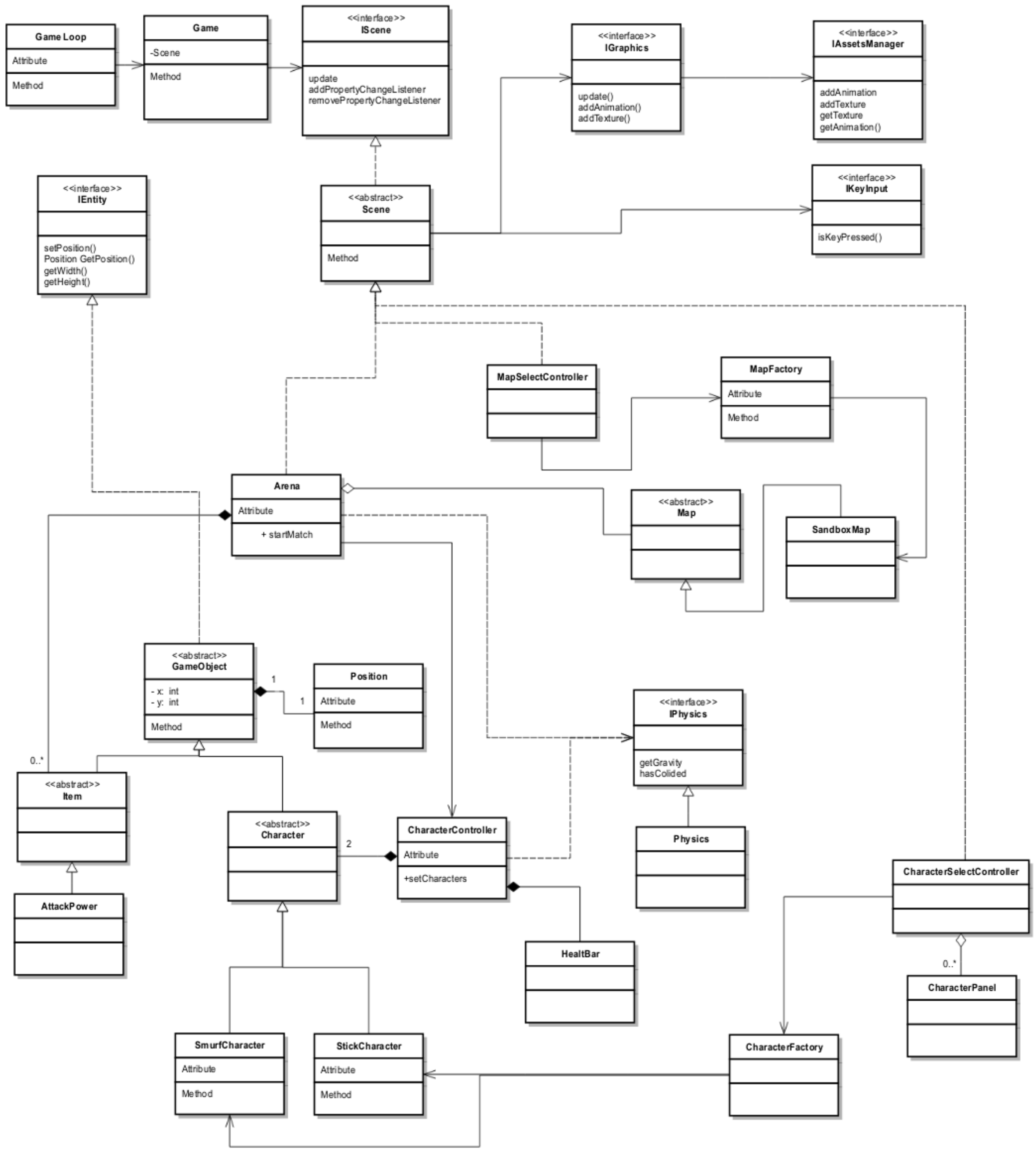
Game Loop
Attribute
Method
Game
-Scene
Method
<<interface>>
IScene
update
addPropertyChangeListener
removePropertyChangeListener
<<interface>>
IGraphics
update()
addAnimation()
addTexture()
<<interface>>
IAssetsManager
addAnimation
addTexture
getTexture
getAnimation()
<<interface>>
IEntity
setPosition()
Position GetPosition()
getWidth()
getHeight()
<<abstract>>
Scene
Method
<<interface>>
IKeyInput
isKeyPressed()
MapSelectController
MapFactory
Attribute
Method
Arena
Attribute
+ startMatch
<<abstract>>
Map
SandboxMap
<<abstract>>
GameObject
- x: int
- y: int
Method
1
1
Position
Attribute
Method
<<interface>>
IPhysics
getGravity
hasColided
0..*
<<abstract>>
Item
<<abstract>>
Character
2
CharacterController
Attribute
+setCharacters
Physics
CharacterSelectController
AttackPower
HealtBar
0..*
CharacterPanel
SmurfCharacter
Attribute
Method
StickCharacter
Attribute
Method
CharacterFactory

Figure 2: UML.