Assignment 1 Group 8008

Joost Verbraeken 4475208
Eric Cornelissen 4447859
Nick Winnubst 4145747
Cornel de Vroomen 4488628
Michael Tran 4499638

Exercise 1 - The Core

Question 1

1. First from the requirements we will look at the nouns to derive the classes. This will only be found in the Must, Should and Could haves. For a lot of these nouns we don't have a separate class. For example controls, screen and score do not have their own class. Some of these classes like some powerups still need to be made.
2. When the CRC cards are created for every class we identify the relationships and responsibilities by looking at the verbs in our requirement

Game
Purposes:
- Is able to start the whole process of playing the game.
- Starts up the window.

Collaborators:
- Screen
- All the scenes

Doodle
Purposes:
- Is able to move left and right by an user pressing left or right arrow.
- Will constantly be pulled down by gravity.
- Collides with platforms, powerups and enemies.
- The background moves up when the Doodle jumps higher than ½ of the screen.

Collaborators:
- Platform
- Powerup
- Enemy
- Background

Platform

Purposes:

- Will make the Doodle jump jump when it collides with the platform.
- Will be created inside of Blocks.
- Sometimes have enemies or powerups on them.
- Sometimes move horizontally or vertically.
- Sometimes break when jumped on.

Collaborators:

- Doodle
- Block
- Enemy
- Powerup

Powerup

Subclasses:

jetpacks, springs, propellor hat, spring boots, helicopter, shield, trampoline and spaceship.

Purposes:

- Will help the Doodle be stronger.
- Are placed on platforms

Collaborators:

- Doodle
- Platform

Enemy

Purposes:

- Will kill a Doodle when colliding with it.
- Are placed on platforms or will fly in the air.

Collaborators:

- Doodle
- Platform

Block

Purposes:

- Create and contain platforms, enemies and powerups.

Collaborators:

- Platform
- Enemy
- Powerup

Our actual implementation looks a bit like the CRC cards stated above, but there are also some differences. Like already stated in point 2: controls, screen and score do not have their own class and also some powerups still need to be made at this moment. For most of the classes

we've also created factories which will create the needed class. For all the collision methods a separate class 'Collisions' is added.

Question 2

The main classes that are the basis of the game are ServiceLocator, Game, World, Doodle and Platform.
ServiceLocator is a class that contains all the Factories which are needed to create all objects in the game. The ServiceLocator is created by game and at initialization will register all the possible object factories in our program. The World, Doodle and Platform are all created through the factories contained in the ServiceLocator. Every class has access to this object and uses it to create other classes.
The Game is the main object that contains a lot of the important data needed, for example the scenes are stored here. Also Game contains everything that has to do with the window of the program. So it has the width and the height, and also creates the window at the start of the program.
World is one of those scenes contained in Game. World has a set with all the elements which are in the game at that moment. The World class renders and repaints all the elements every frame.Lastly the World class has some important data about the game, for example the gravity factor and the speed factor of the Doodle.
The Doodle, one of the elements contained in World, is the object controlled by the player. In here the left and right key are observed and will be responded to when pressed. So the most important part of the Doodle class is the movement inside of the World. Left, right, up and down will be done here. Also the collisions with other objects inside the scene, like Platforms, Powerups and Enemies are registered here.
A Platform is an object which is also placed inside the World. The only thing the Platform is responsible for is sitting there and being collidable. Because when the Doodle collides with the Platform, the Doodle 'jumps' up.

Question 3

The other classes, for example Powerup or Sound, are not as important as the ones stated above because the ones from question 2 are at the basis of the game. The important classes like Game or Doodle are really needed to create a playable game, a Powerup or Sound will make the game only more fun.
These non-main classes are still very important to keep structure in the code. You could probably put Sound or Collisions into other classes, but these classes will then be larger and too cluttered.

Question 4

The document "AllCreatedDocumentsAssignment1" has this Class diagram of the main classes.

It's also at the following link:
https://github.com/jverbraeken/DoodleJump/blob/develop/UMLMainClasses.svg

Question 5

The document "AllCreatedDocumentsAssignment1" has this sequence diagram of the main classes.
At the following link is the sequence diagram of the main classes is also found:
https://github.com/jverbraeken/DoodleJump/blob/develop/UML/sequenceDiagram.svg

Exercise 2 - UML in Practice

Question 1

Composition: class A is composed of (i.a.) class B -> B cannot exists without class A
- Example: a heart and a ventricle

Aggregation: class A uses class B -> B can exist independently from class A.
- Example: a student and a laptop

Question 2

We do not have any parametrized classes in our code, because parametrized classes are normally only used to define data structures like sets, lists, maps, etc. Because our program does not demand any custom data structures we did not implement any of them. However, we do use a custom data structure from Guava (a LoadingCache) to optimize memory consumption to make sure the game will run on devices with very little memory as well.

We would use parameterized classes in our UML if we do need custom data structures that are not available in Java or in any of the "famous" libraries like Apache Commons or Guava. The reason we would parametrize these data structures is because the structure should be abstracted from its data (i.a. To avoid code duplication and non extensibility). That is, the data structure does not need to know anything about its data, not even which class the data is, it only needs to provide some structure for the data.

Question 3

All the UML Diagrams can be found in our “AllCreatedDocumentsAssignment1” because some are too large to clearly see all the classes we also provide the links to our Github repository below:

UML diagram containing only the classes
https://github.com/jverbraeken/DoodleJump/blob/develop/UML/diagram_compressed.png
UML diagram containing the methods and fields:
https://github.com/jverbraeken/DoodleJump/blob/develop/UML/diagram.png
This extensive UML diagram was asked not for in the exercise explicitly, but because (almost) all PowerPoint slides did use fields and methods we decided to include this diagram to be sure.

We chose to use an interface driven design so that everyone could start working (almost) immediately as all methods and classes are defined already and thus can be used. It also provides a great amount of flexibility because every object implementing an interface can be swapped for another one without changing all the code.
In our program the preference is given to interfaces instead of (abstract) classes. Classes can inherit from multiple interfaces so that the existence of a method is guaranteed, but cannot inherit from multiple (abstract) classes. Classes that use other classes do not care about the implementation of the other class, they only want a certain result and are willing to provide some information to the other class to return the result.
However, there is one abstract class, namely *AGameObject*, that provides an implementation for methods concerning hitboxes. The implementations of this method is for all game object the same and using an abstract class greatly reduces code duplication.

Our program embraces the principle of encapsulation. By default is everything private.
Fields are always private because they are implementation details of a class and thus not relevant for other classes.
Constructors are private because we don’t want random classes to suddenly hack new objects into the game. Instead we want to control the way objects are created so that we can, possibly at a very advanced stage, for example introduce the flyweight design pattern (see SpriteFactory). To control the creation of new classes we use factories for every game object. Actually, we don’t even want someone to hack a new factory in the game, so the constructors of factories are private as well. The reason we want to do this, is to prevent duplicate factories which are both inefficient and could introduce some problems when using multiple threads. Factories must be instantiated at some point in the program though. To accomplish this we use a combination of the service locator pattern and dependency injection and modified the result.

- The service locator class provides all services, i.a. Factories, to all classes that want to use it. This way we achieve loose coupling as the only reference to the services that classes have is a single reference to the service locator. Aside from that it’s also

possible to swap the implementation of a class for another implementation very easily (we use interfaces everywhere, so this is possible).
- Normally the service locator is a singleton which results in strong coupling between classes and the service locator. To achieve loose coupling between the classes and the service locator we use dependency injection to provide the service locator to the classes that want to use it. Additional benefits of using dependency injection are better testability (as singletons are very difficult to mock) and services swappable at runtime.
- Traditionally service locators are implemented with data-structures that bind the name of a service with the service itself. A service is requested by using the name of the service. However, this method is very unsafe as the programmer can make a typo in the name of the service it wants to request resulting in a crash on runtime. Instead we provide a method in the service locator for each service it provides. It results in many methods, but also in compile-time safety as the program does not compile if the programmer makes a typo.

Java AWT sprites are not passed to clients directly, but instead wrapped into a Sprite object that provides some detail about the sprite (such as it dimensions) and allows us to swap awt for another framework without too much trouble.

There is no object containing code to draw something except for the Renderer. Every class that want to draw something on the screen call certain methods in the Renderer that will take the responsibility for the actual drawing itself. The reason is we might want to support Android as well that cannot handle AWT. By abstracting the rendering code from the classes we can easily implement the code to support i.a. the Android OS.

Classes that want to get mouse/keyboard input have register themselves to the InputManager that will send them notifications if something happens. In exchange the classes that register themselves to InputManager must implement an interface that allows InputManager to send notifications to them. This observer pattern follows the principle “Tell, don’t ask” and has as benefit that classes don’t have to ask the InputManager each time if something has happened and thus the performance is better.

Exercise 3 - Logging requirements

1. Functional requirements

For the game Doodle Jump, four categories of functional logging requirements can be identified using the MoSCoW method.

1.1 Must haves

- The possibility to log to the console.
- The possibility to log to a log file.
- A log should contain a message.
- The message should be the last part of a log.
- A log should contain a timestamp for the event.
- The timestamp should be the first part of a log.
- Each log (except stacktrace) should be on one line.
- Different fields (message, timestamp, etc.) should be separated by a ‘|’.
- An unspecified, error, info, warning log.

1.2 Should haves

- A log should contain the class of origin of the event.
- A stacktrace log.

1.3 Could haves

- Colored logs in the console.

1.4 Won’t haves

- Log different type of events (e.g. keyboard, mouse) to different files.

2. Non-functional requirements

For the game Doodle Jump, four categories of non-functional logging requirements can be identified using the MoSCoW method.

2.1 Must haves

- The game started.
- The game paused.
- The game resumed.
- The game ended.
- The game restarted.
- The Doodle died.

2.2 Should haves

- The game catches an exception.
- The game loads a sprite.
- The game loads a sound.
- The player pressed the start button.
- The player pressed the pause button.
- The player pressed the resume button.
- The player pressed the start again button.
- The player pressed the menu button.
- The player pressed the left-arrow key.
- The player pressed the right-arrow key.
- The player clicked the left-mouse button.

2.3 Could haves

- A new scene is shown.
- A powerup is created.
- An enemy is created.
- The player pressed any key on the keyboard (includes the key pressed).
- The player pressed any button on the mouse (includes the button & location).
- The frames per second of the game.

2.4 Won’t haves

- The Doodle collides with an ingame object.