Assignment 1

23 september 2016 - Group 666

Exercise 1 - The Core

1. RDD - From requirements to classes

Step 1: Noun phrases

These are the noun phrases from the requirements divided into their categories.

Obvious classes	Uncertain candidates	Nonsense
Game	Start screen	Start
Player	Game over screen	Momentum
Asteroid	Screen	Keys
Projectile	Texture	Direction
Entity	Points	Random spot
UFO	Level	Lives
	Mode (Arcade and survival)	Hyperspace
	Enemies	Size

Step 2: Candidate classes

These are the classes that should be implemented. Most uncertain candidates and nonsense can become attributes or responsibilities, but not classes.

Game	Player	Asteroid
Projectile	Entity	UFO

Step 3: CRC (Class-responsibility-collaboration) cards

These are the CRC cards. On these cards the super and subclasses are visible as well as the responsibilities of each class and the collaborators.

Game	
Superclass(es):	
Subclasses:	
Open Screens	Entity
Start	Player
End	
Create entities	
Destroy entities	

Player	
Superclass(es): Entity	
Subclasses:	
Turn	Game
Accelerate	Entity
Have lives	Projectile
Respawn	
Shoot	

Asteroid	
Superclass(es): Entity	
Subclasses:	
Have a size	Entity
Split	
Have points	

Projectile	
Superclass(es): Entity	
Subclasses:	
Have a lifetime	Entity

UFO	
Superclass(es): Entity	
Subclasses:	
Have points	Entity
Spawn	Projectile
Shoot	
Have a size	

Entity	
Superclass(es):	
Subclasses: Player, Asteroid, Projectile, UFO	
Have a location	Game
Have speed	Player
Wrap around	Asteroid
Hit entities	Projectile
	UFO

Comparison:

The result resembles the actual implementation very well. The actual implementation has a couple of helper classes, namely the Spawner and Launcher class. The Launcher class was needed to create a game environment (to be able to run an update every 60th of a second). The Spawner class removed some of the many responsibilities from the Game class and it improved the overview of the project. We have also created the utility class Display to display text and lives on the screen. These classes were not thought of in the making of the requirements, but they are necessary to implement the them.

2. Description of *main* classes

The main classes in the implementation are: Game, AbstractEntity, Player, Asteroid, Bullet and Saucer.

Responsibilities

The responsibilities of Game are: Managing the score, the entities, the player, the spawner, the display and the game modes. To manage the score it has an addScore() method and sends the score to the display. To manage the entities it has a create(), destroy() and checkCollisions() method and it has getters. To manage the player it manages the entities and it has a getter. To manage the spawner it updates the spawner every tick. To manage the display it updates the correct screen according to its game mode.
The responsibilities for AbstractEntity are: Collisions, location, drawing, radius, speed, death and wrap around. For the collisions, death and drawing it refers to the subclasses. For the location, the radius and the speed it has getters and setters and a method called distance() and speed(). For the wrapping around it has a method called wrapAround().
The responsibilities for Player are: lives, invincibility, hyperspace, acceleration, deceleration and firing. To manage its lives Player has the gainLife() method and getters and setters. For the invincibility and hyperspace it has methods called invincibility() and hyperspace() and getters. Acceleration and deceleration are handled in accelerate() and slowDown() method. Firing is done in the fire() method.
The responsibility for Asteroid is size. Size has getters and setters.
The responsibility for Bullet is friendliness. Friendliness has getters and setters.
The responsibilities for Saucer are: Size, direction, path and shoot. The Size has getters and setters. The direction and path have setters and shooting is done in the shoot() method.

Collaborations

Game collaborates with the most classes, because its update() method is called by the launcher every tick. So Game sends this update through to every object that needs to be updated that tick. For example, the entities are updated when the game mode is “GAMEMODE_GAME”. Most other classes don’t call other classes that often. The player and saucer create some bullets and they update the score. Furthermore the spawner creates a couple of entities, but it is not a main class.

3. Reflection on non-main classes

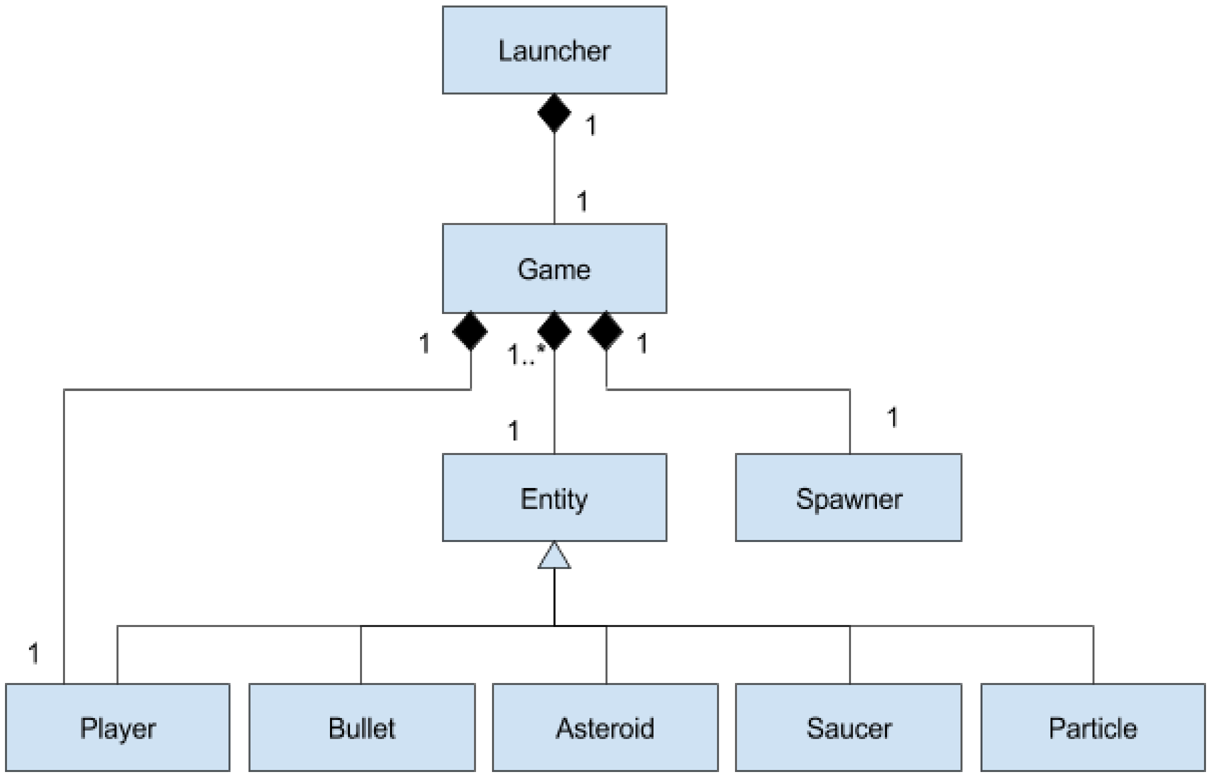
We have ten classes in total. Six of them are the main classes the other four are helper or utility classes. These classes are: Particle, Display, Launcher and Spawner.

Reflection

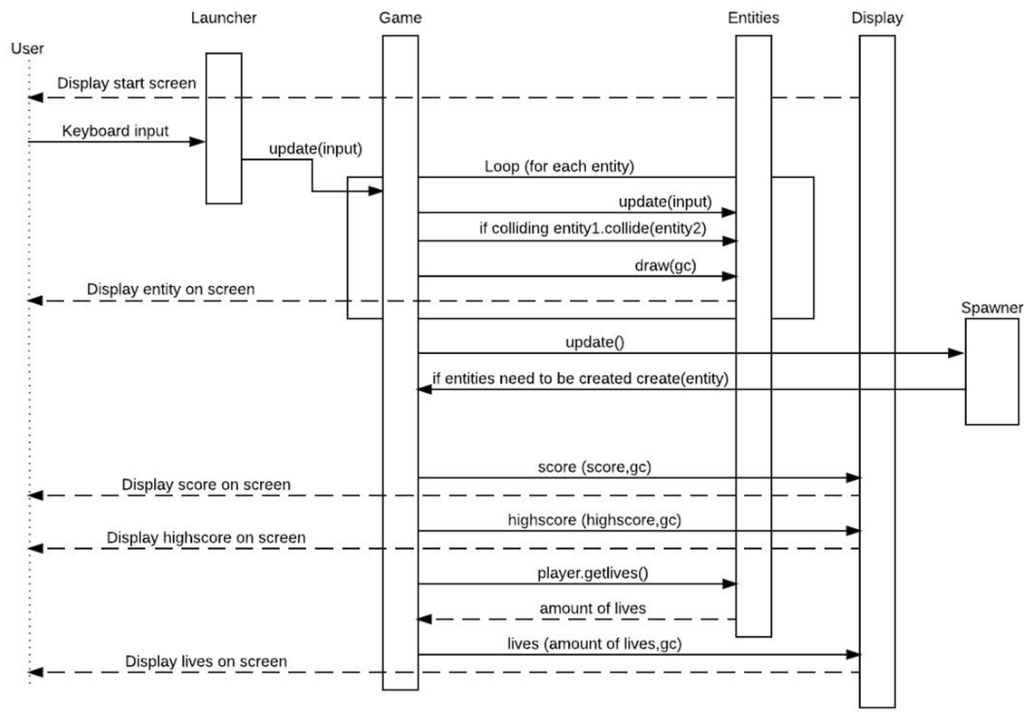
Particle is a class for all particles. Particles are created in an explosion when entities die. It is a necessary class and it cannot be merged with other classes.
The Display class is a utility class for the main class Game. It displays all text and lives on the screen. For a better overview it is needed that this class exists.
Launcher is the class with the main method, its only function is to call the Game every tick and to send the input of the user through.
The Spawner class is called every tick that the game is active. It spawns asteroids and saucers at the appropriate time, just as Display it removes excess responsibilities from Game.
Since all these classes are needed, no changes will be made in the code.

4. Class diagram

This class diagram shows the classes of our implementation, with their relations and association constraints.



5. Sequence diagram



While the player is in the state where they play the game, this sequence is looped.

Exercise 2 - UML in practice

1. Associations

Difference

Aggregation implies a relationship where the child can exist independently of the parent. Composition implies a relationship where the child cannot exist independent of the parent.

Implementation

As seen in the class diagram the implementation only has composition relationships. The Launcher has a Game. The Game has a Player, a list of entities and a Spawner.

Description

The Launcher’s main method creates a new Game. Without the launcher, the game would be unreachable, thus it is a composition relationship. The Game creates the player and a spawner, which fills the entitylist with asteroids and saucers. These would be unreachable without the Game, so they are composition relationships.

2. Parameterized classes

There is no parameterized class in our source code. A parameterized class is most obviously useful for working with collections in a strongly typed language. This way, you can define behavior for sets in general by defining a template class Set. Since we do not have a strongly typed collection in our source code, we do not need it.

3. All hierarchies

The class diagram at Question 1.4 includes all hierarchies. Their explanation can be found in Question 1.2 and 1.3. The type of hierarchies are composition associations and generalization. A generalization is used for the AbstractEntity class and its subclasses, to show a “is-a” type. Considering the lectures, there are no hierarchies that should be removed. There is no necessary change.

Exercise 3 - Simple logging

1. Requirements

There are five actions a player can do. Sorted according to duration type, these are:

Durative:

- Rotating clockwise.
- Rotating counterclockwise.
- Moving forwards.

Instantaneous:

- Firing a bullet.
- Going into hyperspace.

There are three actions that can be performed by the game itself. These are all instantaneous.

- Spawning an object.
- Removing an object.
- Adding a life to the player's counter.

Furthermore, UFOs can perform two instantaneous actions as well.

- Firing a bullet.
- Changing their direction.

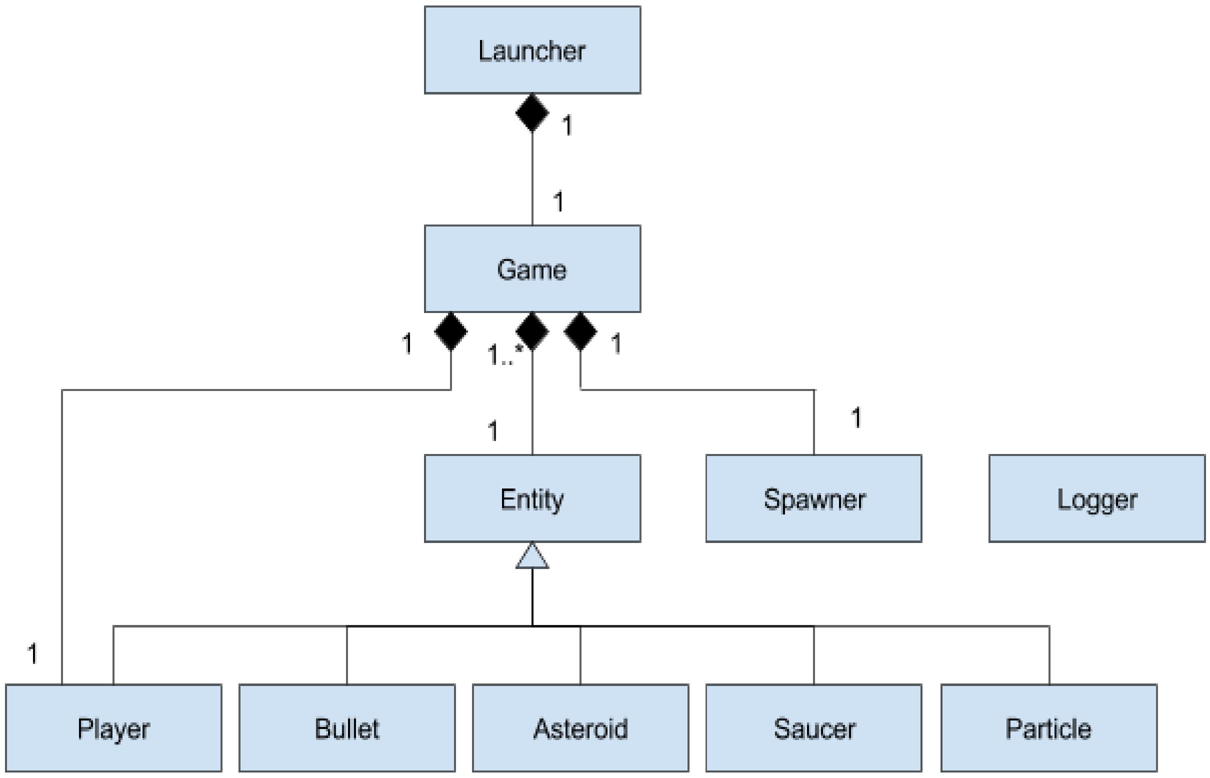
Of these, the spawning of objects, the removing of objects, the player going into hyperspace and adding a life to the player's counter should be logged. It should also note certain events, such as the constructing, starting/game overs, pausing and restarting of the game and the start of new levels. Gaining points and the writing of highscores (in addition to errors with writing/reading highscores) should also be logged. When the player is hit by a saucer, an asteroid or a bullet, an asteroid is hit by a bullet or a saucer is hit by a player's bullet, this is also logged.

Instantaneous actions should simply be logged once.

All logged actions should note the time.

2. Analysis and design

Responsibility-driven design: The logger class contains all the responsibility for logging. Outside classes only need to retrieve the logger and use the appropriate command to pass a string to the logger. The logger class will then make sure this string is logged.



Updated UML