Assignment 1

Game: Temple Run

by

Group 9

Course: TI2206

Software Engineering Methods

Maarten Sijm
Robin van Heukelum
Mathias Meuleman
Mitchell Dingjan
Maikel Kerkhof

TA: Jurgen van Schagen

Teacher: Alberto Bacchelli

18-09-2015

Exercise 1 - The Core

1. The first step we take is using the requirements to determine which classes are needed to build our game. After deriving which classes are necessary to build our game we started to write down every class on a separate card.
   The next step was determining the collaborations and responsibilities between the classes and write them down too.

Classes we created:

- Screen
  - Main Menu Screen
  - Game Screen (handles keyboard input)
  - Popup
- Track (randomly generated)
  - (consists of) Lane
- Entity
  - Player
  - Item
    - Coin
    - …
  - Obstacle
    - Log (on ground)
    - Fence (you can slide under)
    - Pillar (you must avoid, not jump or slide)
- Player Account/Profile

Our result is displayed on the right. →

There are two main differences and two small details.

1.1. The first big difference is: on the CRC cards, we have created a hierarchy with the Screens. In practice, this was not possible, because we are using JavaFX. There is only one class that extends from the library class Application, and the launch method of that class can only be called once.

1.2. The second big difference is: on the CRC cards, we had created a Lane class. We did not use this in our implementation, because the Entities store their own location on the Track. Therefore it would be redundant to also store them in different Lanes.

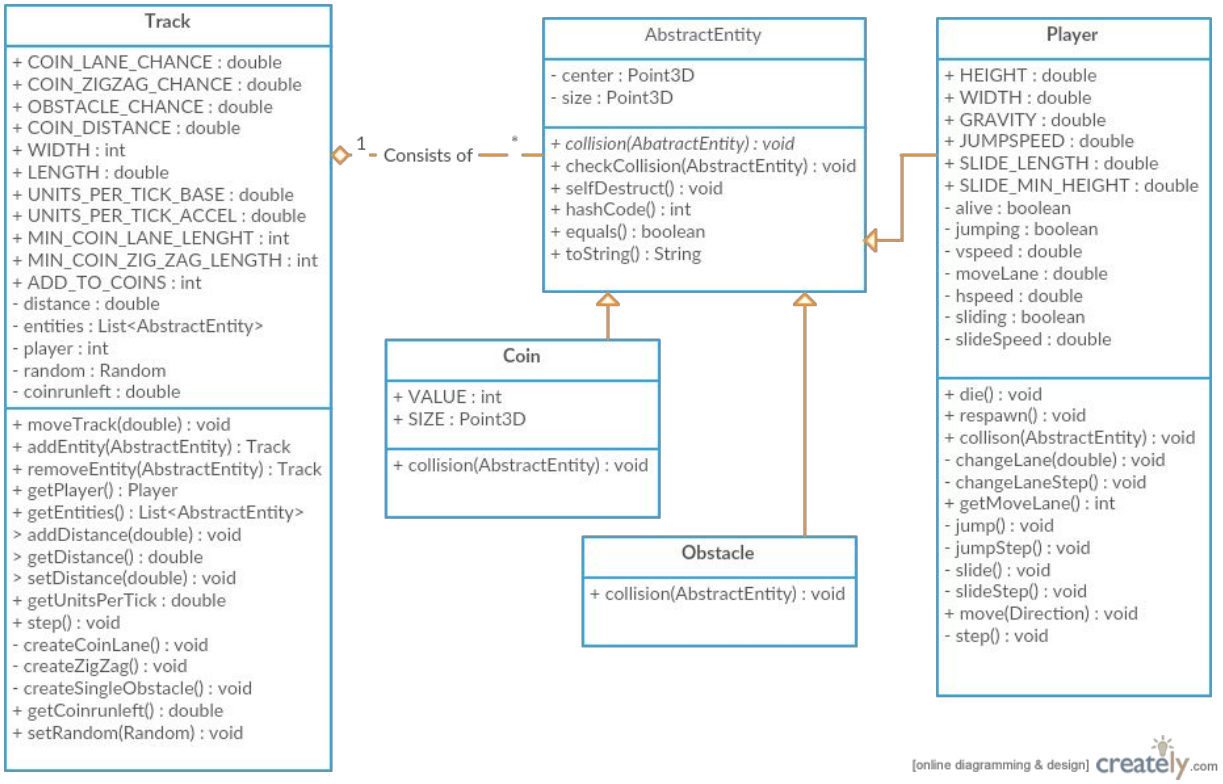
1.3. The first small detail is: on the CRC cards, we have created different Obstacles. In our own implementation, we have not (yet) done this. This will probably be added when we add textures to the game.

1.4. The second small detail is: PlayerAccount is represented by the class State in our implementation.

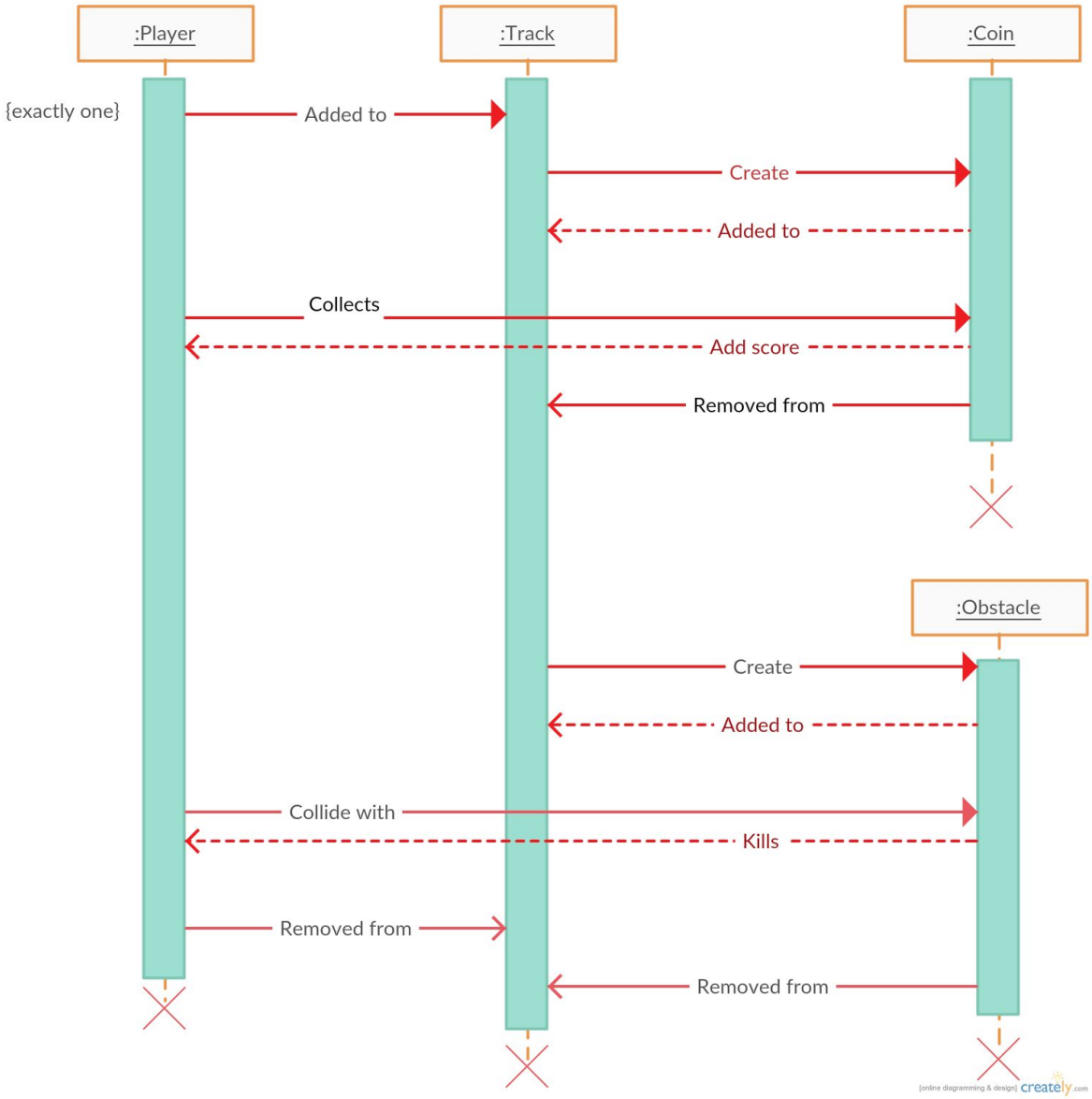
2. The main classes we implemented in our system are Entity (and its subclasses) and Track.
   Track consists of a list of Entities, and maintains this list.
   E.g. Track moves the entities over the Track, makes sure there is only one Player, etc.
   The Player collaborates with Items and Obstacles only when they collide.
   When the Player collides with an Item, the Item should disappear and something happens.
   In the case of a Coin, this "something" is an increment of the score.
   When the Player collides with any Obstacle, the Player should die (and show a Popup).

3. The less important classes are the Screen classes and the PlayerAccount/State classes.
   We need the State class, because it stores the data that the user generates (score, coins and in a later version settings).
   We need the Screen classes, because the separate JavaFX windows should not be merged into one class. This would become very big mess.
   We will not change these less important classes in our current implementation.

4. Class diagram of main classes.



5. Sequence diagram.



Exercise 2 - UML in Practice

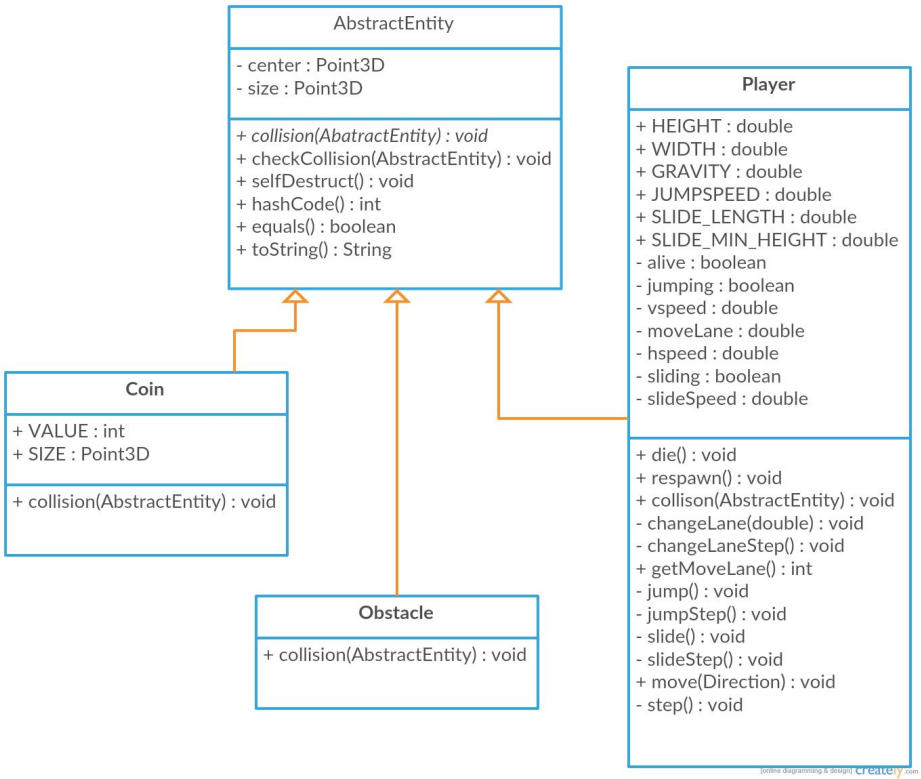
1. Aggregation means that collaborating classes cannot exist with each other.
   For example: a Track could be aggregated with three Lanes, because the Lanes would be useless without the Track and the other way around.

   Composition means that collaborating classes could exist separately from each other.
   For example: a Screen is composed of buttons, text labels, shapes, etc. The Screen does not need to have these components to function.

   In our project, a Track is aggregated with exactly one Player, and composed of all the other types of Entities. The Track checks the collisions between the Player and the other types of Entities.

2. The only parameterized class in our source code is the Java Library LinkedList class. This List contains all Entities on the track, therefore the parameter of this LinkedList is Entity. A LinkedList could also be used to store any other type of objects because of its parameterization. This is a huge benefit.

3.

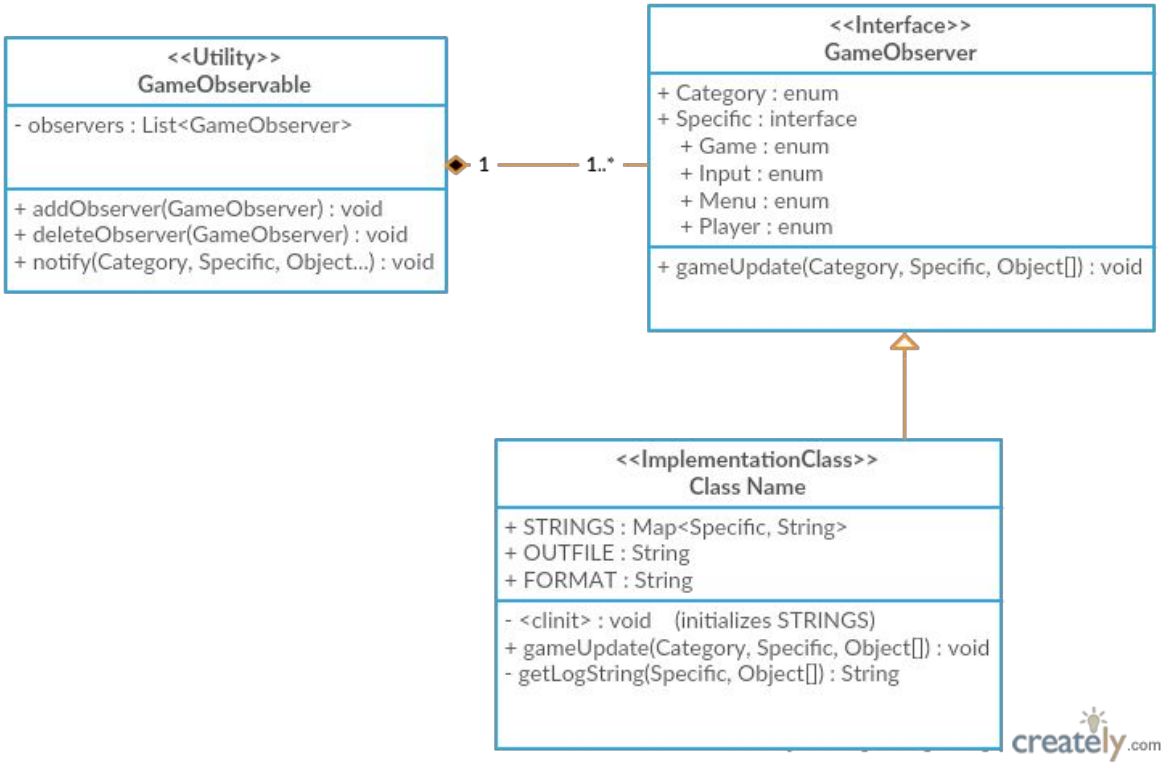


There is only one hierarchy in our source code, and that is the structure in the class diagram displayed here. Coin, Obstacle and Player are all inherited from the AbstractEntity class. This hierarchy is of the Polymorphism type. Polymorphism is used for example with the collision(AbstractEntity) method. All of the sub-classes have this method, but depending on which entity you are dealing with, the result of the method is different.
This hierarchy is in our opinion properly implemented.

Exercise 3 - Simple Logging

1. Requirement:
    - The user shall be able to read a log file containing all events that have happened during the game.
        - The log shall have timestamps.
2. Responsibility Driven Design: The Logger class will have as only responsibility to log everything that happens in the game. Therefore it has to implement a GameObserver interface, making the Logger updatable about everything that happens in the game. The Logger is added to the GameObservable class during start-up. Whenever an important event happens in the game, the GameObservable.notify() method should be called, so all the Observers get notified and the Logger can log the actions.

    UML for the Logger extension:



Short documentation on our extra implemented features

Almost all of our features were implemented smoothly, except for two. More elaborate reflection will come in the sprint reflection on Tuesday.

- Soundtrack: we tried to implement a soundtrack/sound effects feature. The features work (even the JUnit tests pass), however, during Travis' testing phase the build keeps containing an error (which has somehow to do with the threads). Until now, we still have not resolved this issue. This resulted in lots of failing builds.
- Textures: Again a Travis issue. The textures work, but it makes the Travis build log so long because of useless warnings (exceeding 4MB) that builds are terminated and no real test result is given. This resulted in lots of erroring builds.

We hope this will not affect our grade on the Continuous Integration part. We have decided not to include these features in our master branch because it would involve failing builds in master. However you can take a look at the separate branches connected to these features to see that we have implemented all the things we planned to.