SEM - ASSIGNMENT 2

Group 42

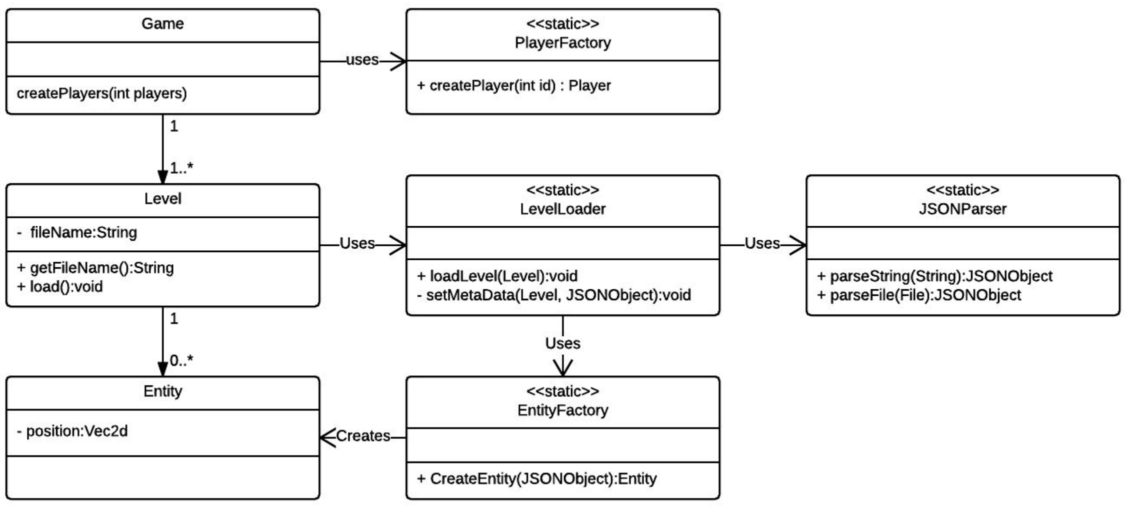
Exercise 1 - Design patterns

In this exercise we will describe two design patterns that we implemented this week in our code.

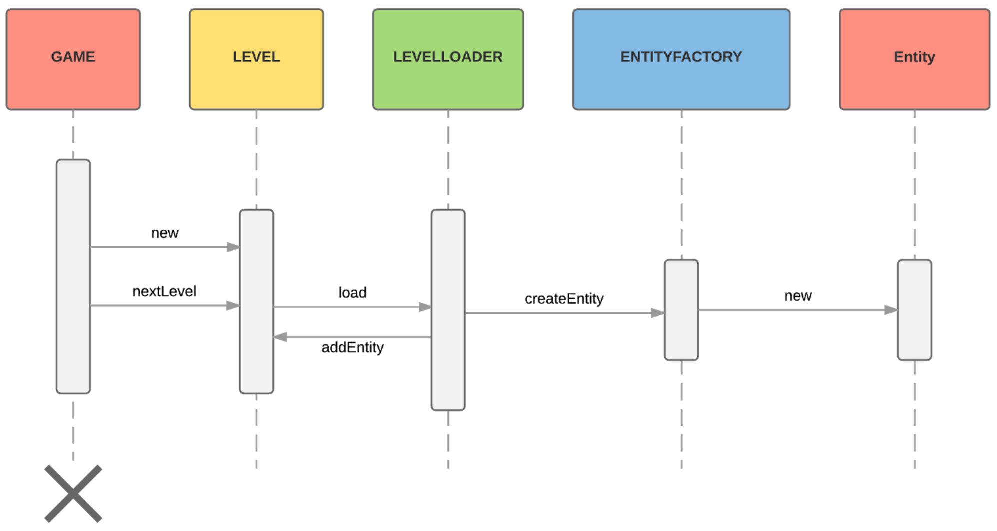
Design pattern 1 - Factory

1. **Write a natural language description of why and how the pattern is implemented in your code.**
   We wanted to be able to load and create a level from a JSON file. To be able to create the appropriate entities the factory pattern was very useful. We can pass a JSON object to the EntityFactory class. That class than correctly parses it to Entities that can be added to the Level object. That Level object can then be added to the list of levels in the Game class.
   We also used the factory pattern to create a PlayerFactory class that reads the controls for the players out of the config.properties file, and add them to the Player object. That Object is then added to the Game class.

2. **Make a class diagram of how the pattern is structured statically in your code.**



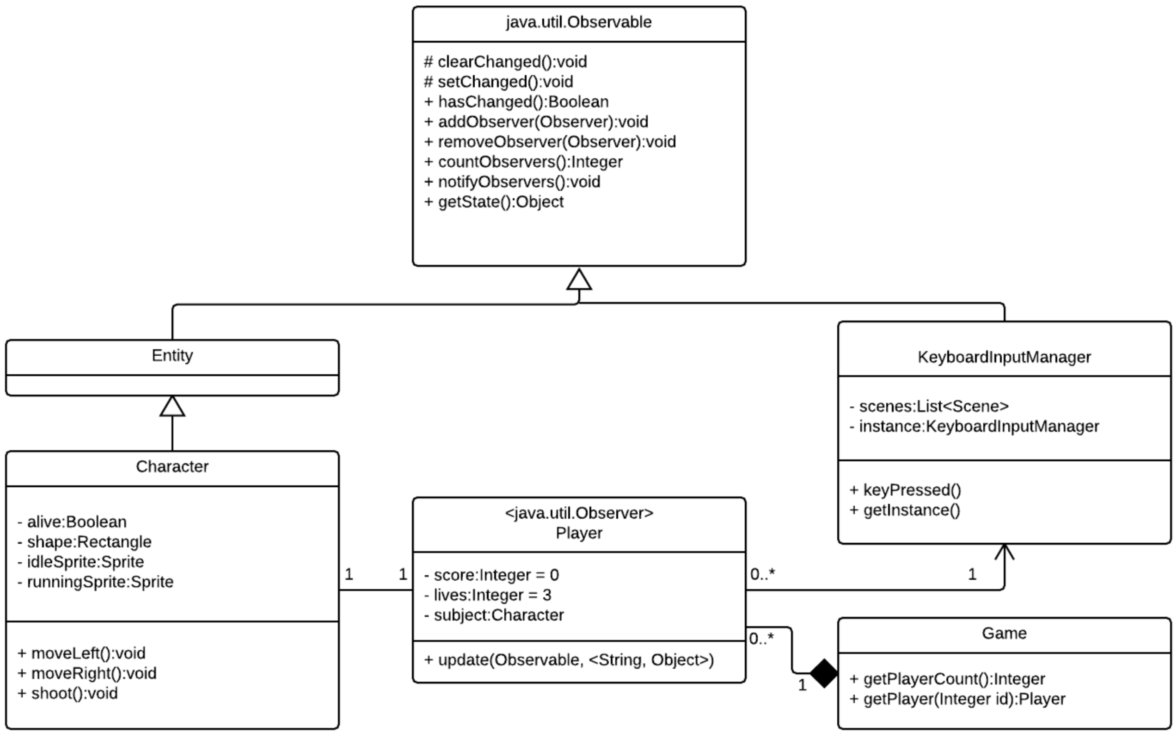
3. **Make a sequence diagram of how the pattern works dynamically in your code.**
   P.S. The use of the Playerfactory can be seen in the diagram in 2.3.



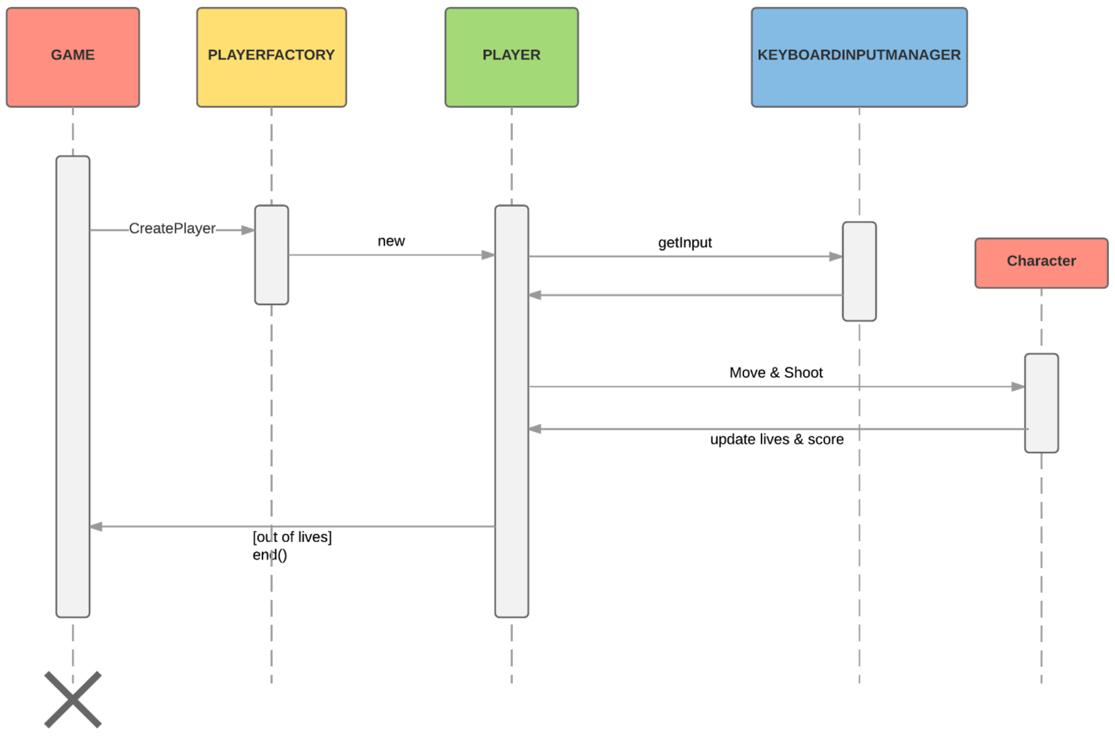
Design pattern 2 - Observer

1. **Write a natural language description of why and how the pattern is implemented in your code.**
   We created a Player object that needs keeps track of the score and the lives of a player. We also want to be able to keep track of the time limit of the level and want to implement the effect of powerups in the future. To make this all happen, a lot of classes need to communicate with each other. The observer pattern brings a solution.
   By making the Player class implement the Observer interface and Entity extend the Observable class of java.util, we can easily keep track of all the entities in the game, and take the appropriate actions when the entities get updated.
   We also made the KeyboardInputManager a Observable to keep track of the input and route it to the right character using the Player class.

2. Make a class diagram of how the pattern is structured statically in your code.



3. Make a sequence diagram of how the pattern works dynamically in your code.

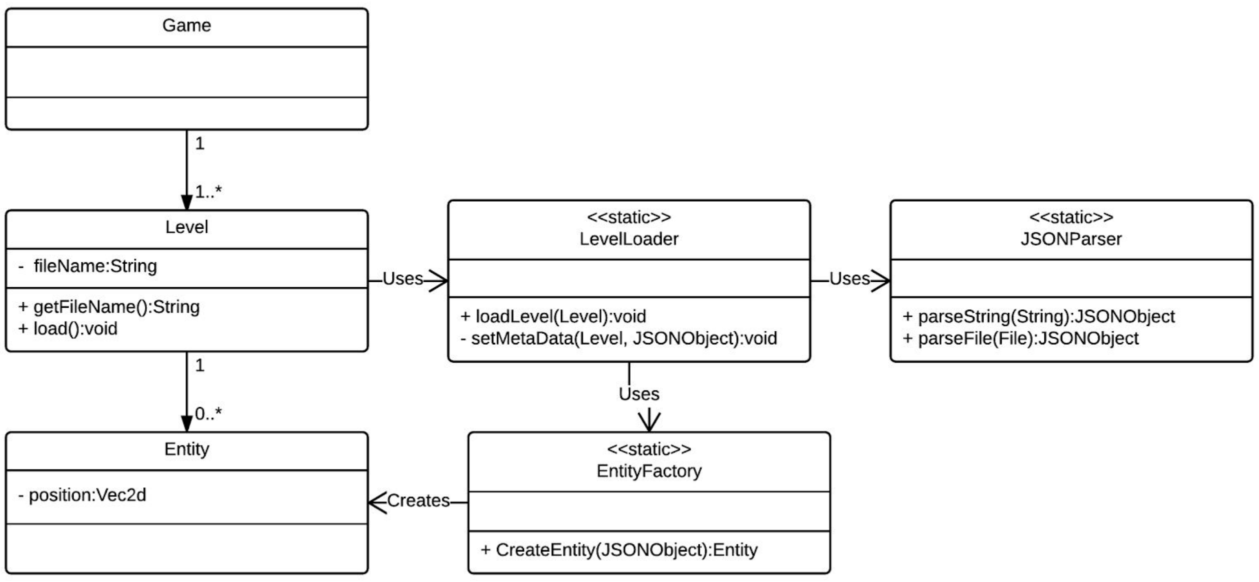


Exercise 2 - Level loader

The level loading extension allows the game to load an array of JSON files with and to generate levels out of them. This extension is described in the following UML files.

UML Diagram

Class diagram

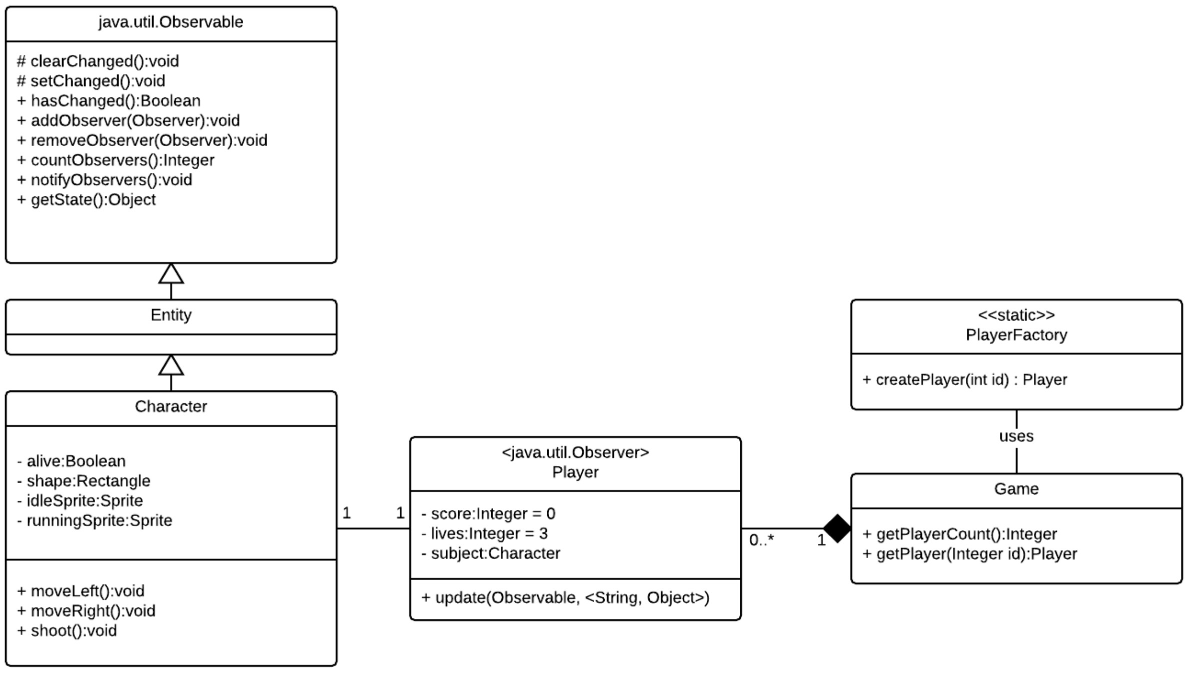


Exercise 3 - Multiplayer

The multiplayer extension allows multiple players to play this version of Bubble Trouble. In the game we will only implement a single and two player mode, however the extension is capable of handling more than two players.

UML Diagram

Class diagram


java.util.Observable
# clearChanged():void
# setChanged():void
+ hasChanged():Boolean
+ addObserver(Observer):void
+ removeObserver(Observer):void
+ countObservers():Integer
+ notifyObservers():void
+ getState():Object
Entity
Character
- alive:Boolean
- shape:Rectangle
- idleSprite:Sprite
- runningSprite:Sprite
+ moveLeft():void
+ moveRight():void
+ shoot():void
1
1
<java.util.Observer>
Player
- score:Integer = 0
- lives:Integer = 3
- subject:Character
+ update(Observable, <String, Object>)
0..*
1
<<static>>
PlayerFactory
+ createPlayer(int id) : Player
uses
Game
+ getPlayerCount():Integer
+ getPlayer(Integer id):Player