# Assignment 1

[TI2206] Software Engineering Methods

Group 4
*Wouter Smit*
*Casper Athmer*
*Damian Voorhout*
*Justin van der Krieken*
*Bram Crielaard*

# Exercise 1

1. Elaboration on classes, responsibilities and collaborations

   We create a GameObject which handles the collisions and movement of all objects. It has a Mover class and CollisionHandler class to handle this. Player and EnvironmentObject extend from this class. All other objects that are in the game, extend from EnvironmentObject.

   There are multiple classes of objects that are in the game. There is a Player class, which handles the shooting, the powerups, the lives and the score. The Bubble class manages a bubble which bounces around the level. It manages the bubbles size and bouncing height. Pickups is another class that extends from EnvironmentObject. It generates a random pickup, which can be either a Powerup, Utility or Weapon. The Weapon class extends from Pickup, and adds itself to a Player when it is initialized. It also handles shooting. A Projectile is an object that is shot by a Weapon. It manages its own duration. A powerup is another type of Pickup, which has an effect on Player. It applies its own effect on Player. Utility is the last type of Pickup, and has an effect on Level and applies this effect to Level. A Wall is an EnvironmentObject which does nothing.
   Level manages the environmentobjects and its time and speed. Game manages all the levels and players.

   Comparison with own implementation

   We did implement most of the classes just described. We didn't implement an EnvironmentObject class, because the only thing it does is making a distinction between Player and all the other GameObjects. This didn't seem to matter during implementation. Another thing we didn't do, is implement a Mover class. The reason for this is that it is easier to handle the moving inside each object separately, because each object moves in a different way. The moving is also not very complicated, so making a Mover class seemed unnecessary to us. Another thing we didn't consider in our Responsibility Driven Design, opposed to our implementation, is the use of interfaces. In our implementation we used some interfaces, like updateable and renderable, to make a contract with certain classes. This turned out to be a useful addition to our code.

2. We tried to keep the number of responsibilities of the main classes to a minimum, in order to make the project as scalable and clear as possible.

   The app is a GameState, which makes it possible to wrap a StateBasedGame around the game, which led to the easy implementation of a startscreen, quitscreen and pausescreen, which are all different states. The StateBasedGame's responsibility is to start the app, set the general settings of the app container and load different states at specific moments.

   Each state handles the rendering and updating of that state and can use the StateBasedGame to enter different states. So this class handles the different states the game can be in (e.g. the menu, the game itself, etc.). The GameState class loads and starts the game.

   The previous classes can be considered GUI classes and controller classes. We didn't consider those classes in our design though. The following classes are the most important classes of our model.
   First off, there is the Game class. Game is responsible for the game. It ensures that the user goes back to the main menu when the user has won the game (or when it's game over). It collaborates with Level. When the user has successfully completed a level, Game ensures that the user goes into the next level. Game is also responsible for managing the Player objects in the game. It calls their update function each tick, and it sends all objects (gotten from the Level class) to the DefaultCollisionHandler, where collisions between objects are handled.
   Which brings us to another important class, the DefaultCollisionHandler. This class handles all the collisions between objects.
   The Level class is responsible for the level the user is currently in. It's responsible for all the objects in that Level (except for the Player object, because this object is always passed on to the next level). It collaborates with Utility, to apply the effects of a Utility on the Game objects. It also calls the update method of all the game objects it manages.
   The Player class is responsible for the Player object(s) in Game. It handles user input, and it handles shooting. For this, it collaborates with Weapon.
   The Weapon class is responsible for shooting. For this, it collaborates with the Player and Projectile (a GameObject that represents a 'bullet').
   The Bubble class is responsible for letting Bubbles bounce around the Level. It collaborates with Level, because when there are no Bubbles left, the Level has been won. It also collaborates with Player, because when a Player is hit by a Bubble, Player loses a life.

3. We consider classes we don't necessarily need in order to let the game function to a certain degree, non main classes. From our perspective a game works if the core mechanics are implemented and the game contains little to no bugs.

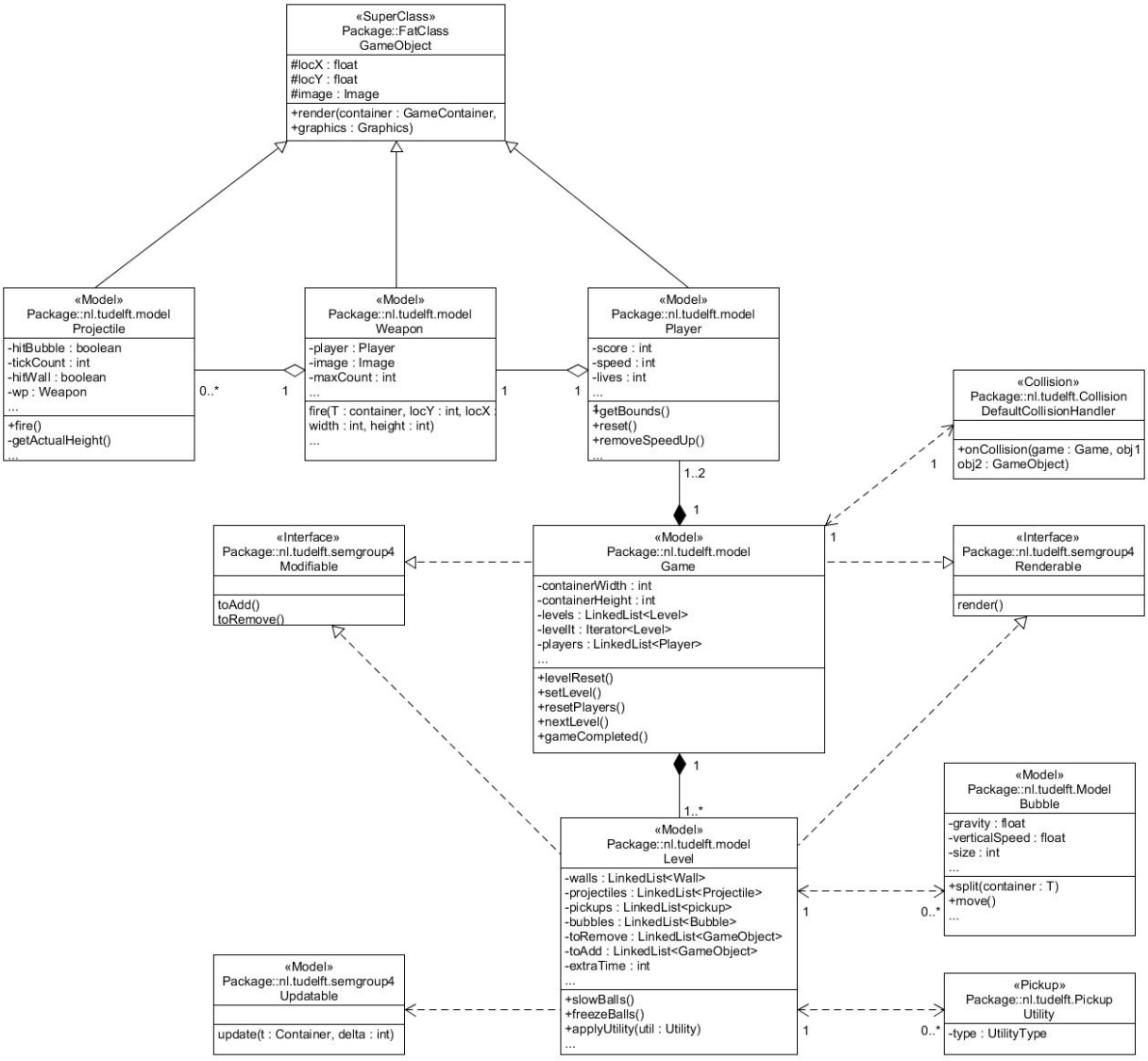
   In our case, the basics mean just the game, so no menus. The game itself  then contains a controllable player, which can shoot and walk. Bubbles are implemented and bounce properly, the player is able to interact with the balls, die because of them when the bubbles hit him and pop them with its weapon. Levels are implemented and can be completed by popping al bubbles. The next level starts when the player completes the current one. Lives are implemented and function as they should, losing a life when the player dies.

   So as a summary, these are the functions we don't consider core mechanics: Score, pickups of any kind, timer, menus, platforms, ladders, options, coop and audio.

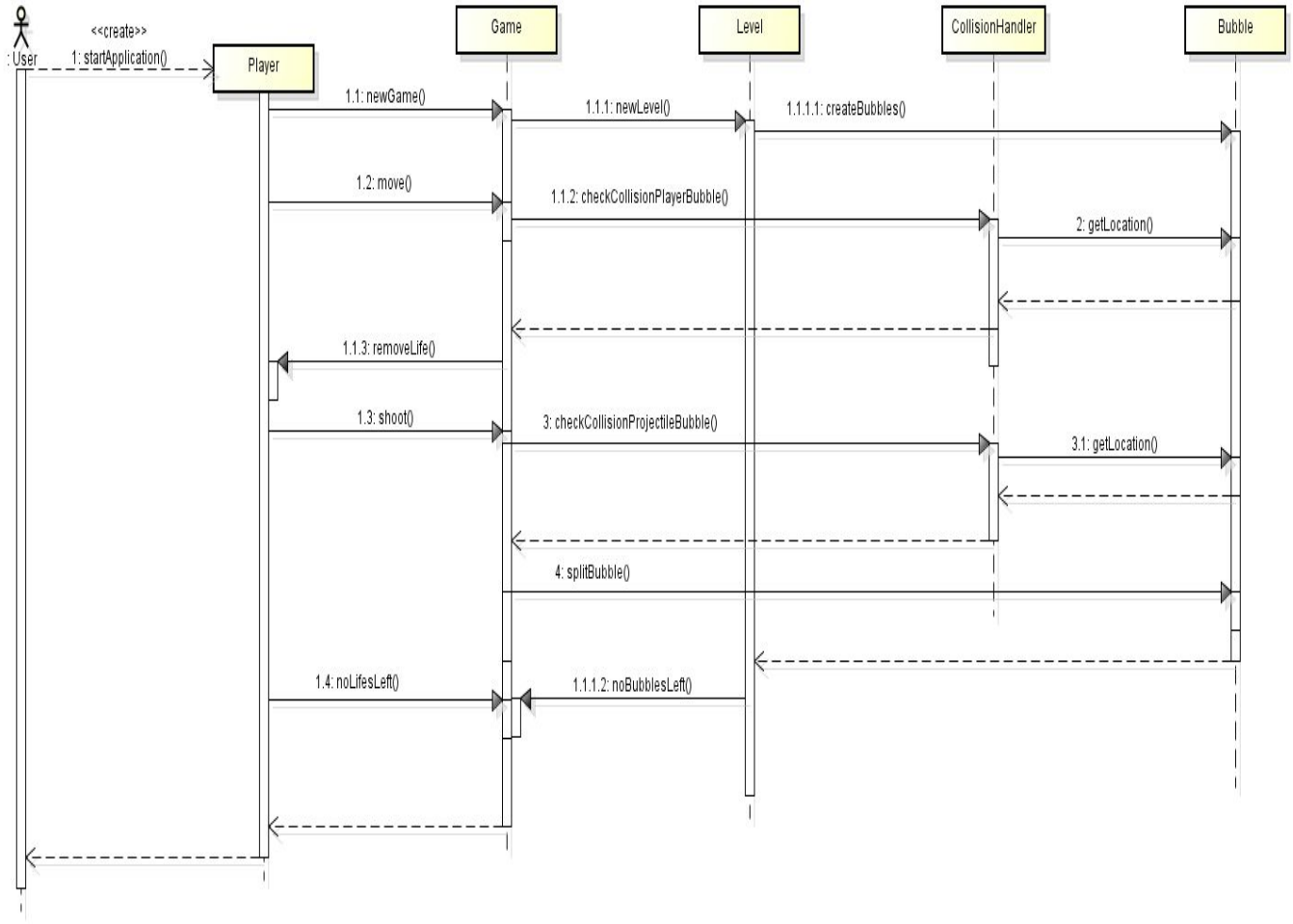
   As stated before, we tried to keep the number of responsibilities of each class to a minimum, meaning that each extra functionality has its own class, or classes. Pickups for example is split up into powerup, pickupContent, utility and pickup. They are split up for a reason, to keep things clear and scalable. No classes are redundant, and it would make little sense to merge any of them because of the reasons just stated.

   The only class that has very little responsibility at this moment is the Helper class, it will contain all functions that otherwise would need its own class and don't fit anywhere else. This means this class can't be dismantled into other classes.

4.


«SuperClass»
Package::FatClass
GameObject
#locX : float
#locY : float
#image : Image
+render(container : GameContainer, +graphics : Graphics)
«Model»
Package::nl.tudelft.model
Projectile
-hitBubble : boolean
-tickCount : int
-hitWall : boolean
-wp : Weapon
...
+fire()
-getActualHeight()
...
«Model»
Package::nl.tudelft.model
Weapon
-player : Player
-image : Image
-maxCount : int
...
fire(T : container, locY : int, locX : width : int, height : int)
...
«Model»
Package::nl.tudelft.model
Player
-score : int
-speed : int
-lives : int
...
+getBounds()
+reset()
+removeSpeedUp()
...
«Collision»
Package::nl.tudelft.Collision
DefaultCollisionHandler
+onCollision(game : Game, obj1 obj2 : GameObject)
«Interface»
Package::nl.tudelft.semgroup4
Modifiable
toAdd()
toRemove()
«Model»
Package::nl.tudelft.model
Game
-containerWidth : int
-containerHeight : int
-levels : LinkedList<Level>
-levelIt : Iterator<Level>
-players : LinkedList<Player>
...
+levelReset()
+setLevel()
+resetPlayers()
+nextLevel()
+gameCompleted()
«Interface»
Package::nl.tudelft.semgroup4
Renderable
render()
«Model»
Package::nl.tudelft.model
Level
-walls : LinkedList<Wall>
-projectiles : LinkedList<Projectile>
-pickups : LinkedList<pickup>
-bubbles : LinkedList<Bubble>
-toRemove : LinkedList<GameObject>
-toAdd : LinkedList<GameObject>
-extraTime : int
...
+slowBalls()
+freezeBalls()
+applyUtility(util : Utility)
...
«Model»
Package::nl.tudelft.Model
Bubble
-gravity : float
-verticalSpeed : float
-size : int
...
+split(container : T)
+move()
...
«Model»
Package::nl.tudelft.semgroup4
Updatable
update(t : Container, delta : int)
«Pickup»
Package::nl.tudelft.Pickup
Utility
-type : UtilityType
0..*
1
1..2
1..*

5.


: User
<<create>>
1: startApplication()
Player
Game
Level
CollisionHandler
Bubble
1.1: newGame()
1.1.1: newLevel()
1.1.1.1: createBubbles()
1.2: move()
1.1.2: checkCollisionPlayerBubble()
2: getLocation()
1.1.3: removeLife()
1.3: shoot()
3: checkCollisionProjectileBubble()
3.1: getLocation()
4: splitBubble()
1.4: noLifesLeft()
1.1.1.2: noBubblesLeft()

# Exercise 2

## 1.

Composition and aggregation are types of relationships between classes. When a class A can only ever exist in the lifecycle of another class instance B, one speaks about a composition. When a class A can also exist outside of instance B, but B uses this class, one speaks about aggregation.

A very distinct metaphor to these two associations is the following:

- A text editor owns a buffer. The buffer has no purpose or meaning outside of the text editor, and when the text editor is closed, the buffer is destroyed. This is a **composition**.
- A text editor uses a file. The file can also be opened by other programs, and is used to save the (text) data contained within. It thus has a purpose outside of the text editor's instance and when the text editor is closed, the file is kept. This is an **aggregation**.

Examples of compositions in our project are:

- Game is a composition of
    - instances of Level
    - instances of Player
    - an instance of CollisionHandler
    - an instance of LevelFactory
- Weapon is a composition of
    - instances of Projectile

Examples of aggregations in our project are:

- Player is an aggregation of
    - instances of Powerups
- Level is an aggregation of
    - instances of the following GameObjects
        - Wall
        - Pickup
        - Bubble
        - Projectile
    - instances of Utility
- Player is an aggregation of
    - an instance of Weapon

We note the following code smell: Level is an aggregation of multiple GameObjects, but not all. This leads to think that there should be a layer 'EnvironmentObject' in between GameObject and its 4 children (excepting Player), that can be used to define the GameObjects in Level. This

is useful, because one can then define a **composition** rather than an **aggregation**: instead of GameObject being used by multiple classes (`Game` and `Level`), `Level` is a composition instances of `EnvironmentObject`: their only purpose is to be in a level.

## 2.

We made extensive use of interfaces in our project and our only parameterizations are on the interface level.

Our collision handler interface is parameterized to take two classes that extend from GameObject.

```
public interface CollisionHandler<O1 extends GameObject, O2 extends GameObject>

      void onCollision(Game game, O1 objA, O2 objB);
```

This seems unnecessary at first, because according to the Liskov substitution principle, any class that extends GameObject can be used instead of a GameObject.

```
      void onCollision(Game game, GameObject objA, GameObject objB);
```

However, we make use of `LinkedList<E>`, which can be bound as `LinkedList<GameObject>` or `LinkedList<Wall>`. While `Wall` is an extension of `GameObject`, `LinkedList<Wall>` is not an extension of `LinkedList<GameObject>`. This means that these cannot be used together.

Using generics however, we can create a `LinkedList<? extends GameObject>`, which can be filled with both `GameObject`s and its children.

Our second parameterization is in the Updateable interface. This interface defines that a class should have an update method to make sure that it is updated with every tick of the game engine.

Its method, `update()`, is declared as follows:

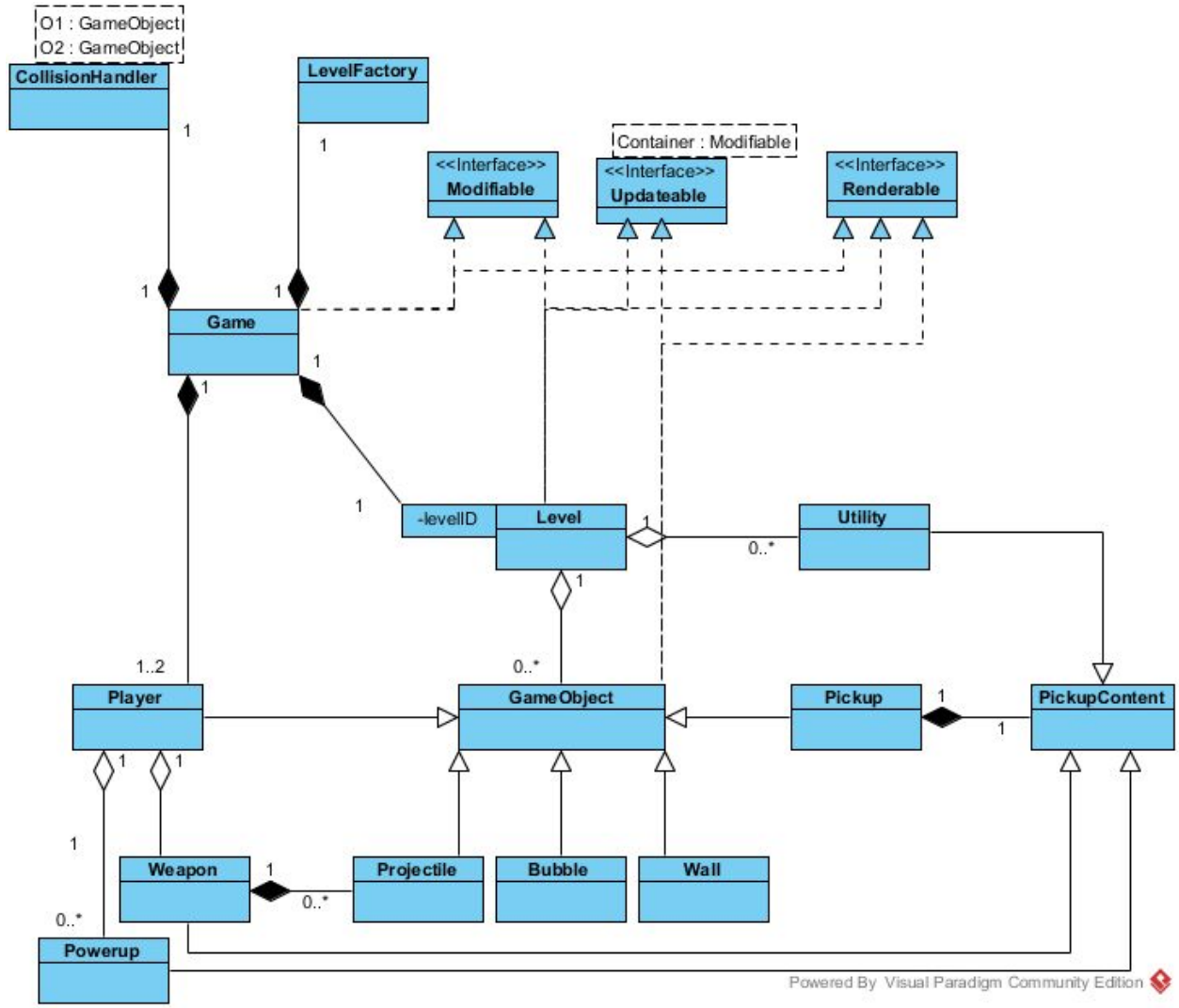
```
public <T extends Modifiable> void update(T container, int delta) throws
SlickException;
```

This means that the update method should obtain an instance of a class that implements the Modifiable interface.

This is beneficial, because objects sometimes need to be removed or added from the game. The Modifiable interface contract states that Modifiable classes should implement a `toAdd(GameObject)` and a `toDelete(GameObject)`. `Level` and `Game` are compositions of classes (`GameObject` and `Player`

respectively in particular). These two classes implement modifiable and could be passed to the update method. The `GameObject`s then can refer to their container (`Level`) and request addition or deletion in accordance with the Modifiable contract. Because these are interfaces, one cannot use the Liskov substitution principle, but has to use parameterization.

## 3.




The most visible hierarchy is the GameObject with all its children. This hierarchy is a polymorphism: multiple objects that appear in the game have different functionality, but all share the same basics. They all have an image and a two coordinates of their location on the gamefield.

There is one catch here. Level is an aggregation of Projectiles, Bubbles, Walls and Pickups, but not of Players. Players are instead stored in the Game object. This makes sense, because Players are not specific to one level, but to the whole game (they move on to the next level upon

completion). The four GameObjects mentioned above do not have any reason to exist outside of a Level however, and it would be preferable to change their relation with Level to a composition.
This can be done by introducing a layer between GameObject and its children, which could be named EnvironmentObject. The four GameObjects can then extend this class instead, while Player extends GameObject directly. The relation between EnvironmentObject and Level can then be changed into a composition. E.g. one can say ‘EnvironmentObjects have no purpose in the system outside of a Level’.

Moving on, we see that a Weapon, a Utility and a Powerup are all polymorphisms of PickupContent. This is the object that is created inside of a Pickup (pickup being the visual block that falls out of a bubble and can be picked up by the Player).

Currently, Player is an aggregation of a Weapon. however, the weapon class cannot function without a player: it cannot shoot projectiles without a Player that gives input. Its existence outside of the Player object is therefore unnecessary. The reason that it is an aggregation and not a composition, is because the Pickups generate a PickupContent  (which could be a Weapon) and store it. This requires a Weapon to exist outside of a Player, making the composition impossible.
This should be changed, so that a Pickup will only generate the Weapon instance when the Player picks it up, and immediately assign it to the Player, replacing the ‘old’ weapon.