Space Invaders : Sprint 2 Assignment

Group 22
TI2206 Software Engineering Methods

Ege de Bruin
Bryan van Wijk
Dorian de Koning
Jochem Lugtenburg

September 25, 2015

Supervisor: Dr. A. Bacchelli
TA: Danny Plenge

Delft University of Technology
Faculty of EEMCS

Exercise 1 - 20-Time

Code improvement requirements

To improve our code, we want to split several classes into more classes. The classes we want to split are Game and GameUIController, because these classes are now too big in our code. We have chosen to split these classes in the following classes:
Game:

Controller

- UnitController
- MovingUnitController
  - ExplosionController
  - BarricadeController
  - AlienController
  - SpaceShipController
  - BulletController
    - AlienBulletController
    - SpaceShipBulletController

Unit

We also want to change the Unit section of our code, we want to have an interface movable that is implemented by the movable/moving. This prevents a lot of duplicate code. Implementing this change would mean we would have to change all of the following classes:

- Unit
- Movable
  - Alien
  - SpaceShip
  - Bullet
    - SpaceShipBullet
    - AlienBullet
  - Explosion
  - Barricade

GameUiController

Changing the structure of unit in our code we also need to change the ui section. This will be the new class hierarchy. GameUIController:

- UiElement
  - UiElementUnit
    - UiElementAlien
    - UiElementSpaceShip
    - UiElementBarricade
    - UiElementExplosion
    - UiElementBullet
      - UiElementSpaceShipBullet
      - UiElementAlienBullet
  - Score
  - Lives

Since we did implement equals methods in most of (the suitable) classes but we did not implement hash code yet. Hashcode should be implemented too.

Code Improvements classes to be changed

Each class we split into several subclasses will have one single responsibility.
Game:

Controller

Controller: this interface class will provide template for the Controller classes of different game elements.

- UnitController: this abstract class has the responsibility of controlling the non moving units in the game
- MovingUnitController: this interface has the responsibility of controlling the moving units in the game.
  - ExplosionController: this class is responsible for modifying the explosions in the game.
  - BarricadeController: this class is responsible for modifying the barricades in the game.
  - AlienController: this class has the responsibility of modifying the aliens in the game.
  - SpaceShipController: this class is responsible for modifying the spaceship.
  - BulletController: this class is responsible for modifying the bullets in the game.
    - AlienBulletController this class is responsible for modifying the alienbullets in the game.
    - SpaceShipBulletController this class is responsible for modifying th spaceshipbullets in the game.

Unit

The change we want to apply to the subclasses of the Unit class is making an interface movable that is implemented by units that move.

GameUiController

Draw:

- UiElement: this (abstract) clas is responsible for drawing all the ui parts on the screen.
  - UiElementUnit: this abstract class is responsible for the units on the screen.
    - UiElementAlien: this class is responsible for drawing the aliens on the screen.
    - UiElementSpaceShip: this class is responsible for drawing the spaceship on the screen.
    - UiElementBarricade: this class is responsible for drawing the barricades on the screen.
    - UiElementExplosion: this class is responsible for drawing the explosions on the screen.

    - UiElementBullet: this abstract class is responsible for drawing the bullets on the screen.
        - UiElementSpaceShipBullet: this class is responsible for drawing the SpaceShipBullets on the screen.
        - UiElementShipBullet: this class is responsible for drawing the SpaceShipBullets on the screen.

- DrawScore: this class is responsible the score on the screen.
- DrawLives: this class is responsible the lives on the screen.

Exercise 1 UML

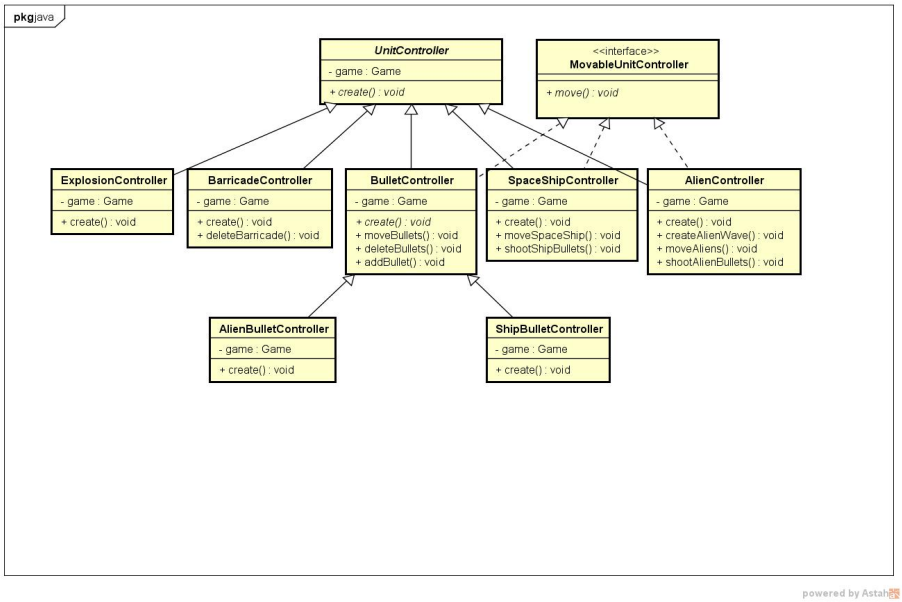

Figure 1: Controller UML Class Diagram

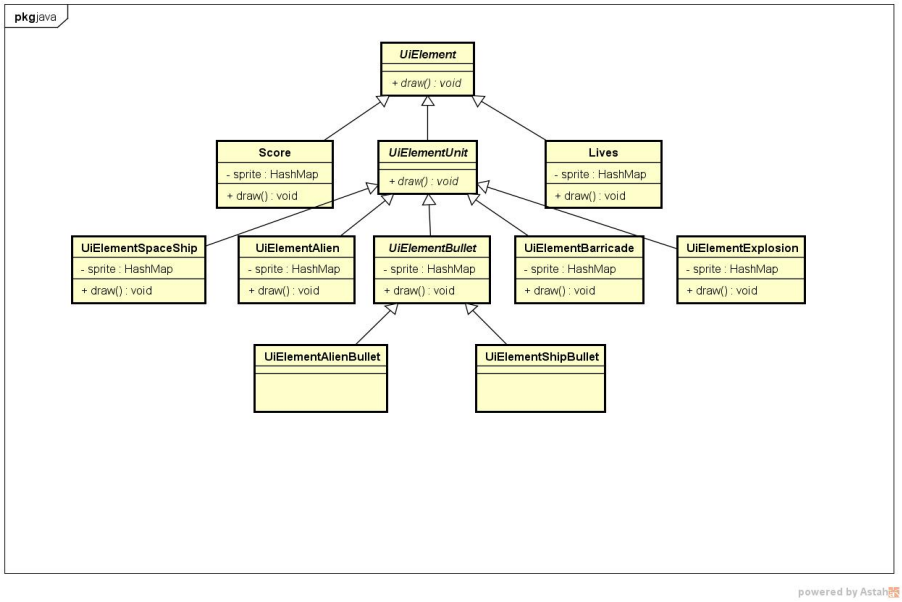

Figure 2: UIElement UML Class Diagram

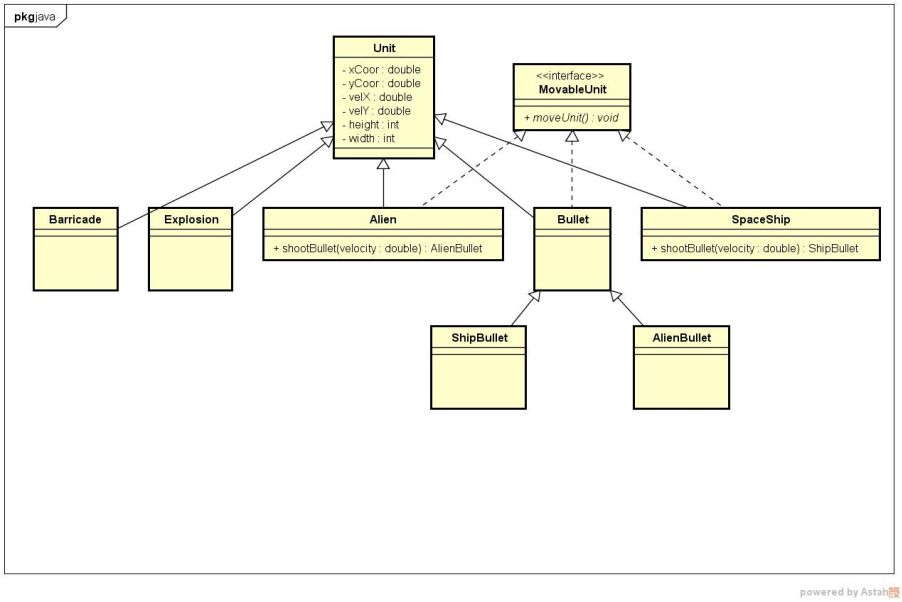

Figure 3: Unit UML Class Diagram

Code Improvement Functional Requirements

A list of functional requirements considered for code improvements using the MoSCoW method described in the previous section.

Must Haves

The Code Improvements must meet the following requirements:

- Code shall be Checkstyle compliant.
- Code shall be PMD and Findbugs compliant.
- Classes shall only have one responsibility.
- Methods shall not be too long.

Should Haves

The Code Improvements should meet the following requirements:

- Code shall contain Javadoc comments explaining complex methods.
- Code shall not be commented.
- Static shall be used where appropriate.
- Private access modifiers shall be used where appropriate.
- Test coverage shall be at least 75%.
- Code shall contain less duplicate code.

Could Haves

The Code Improvements could meet the following requirements:

- Variables shall be named in correct English.
- Methods shall be named in correct English.
- Classes shall be named in correct English.
- HashCode methods shall be implemented correctly where equals is used.

Would/Won't Haves

The Code Improvements won't meet the following requirements:

- ...

Exercise 2 - Your wish is my command

The following new classes were created for the implementation of the PowerUp feature. In the table the responsibilities and collaborations are presented for every class.

Class	Responsibility	Collaborates with	Super	Sub
PowerUpUnit	Location, seize and speed		Unit	SpeedPowerUpUnit, LifePowerUpUnit, ShootPowerUpUnit
SpeedPowerUpUnit	Creation active PowerUp	SpeedPowerUp	PowerUpUnit	
ShootPowerUpUnit	Creation active PowerUp	ShootPowerUp	PowerUpUnit	
LifePowerUpUnitt	Activation powerUp	Player	PowerUpUnit	
SpeedPowerUp	Activation and deactivation powerUp	SpaceShip	PowerUp	
ShootPowerUp	Activation and deactivation powerUp	SpaceShip	PowerUp	
PowerUp	Counting time left	Player		ShootPowerUp, SpeedPowerUp
PowerUpController	Movement and creation PowerUp units	Game, Collisions		

Exercise 2 UML

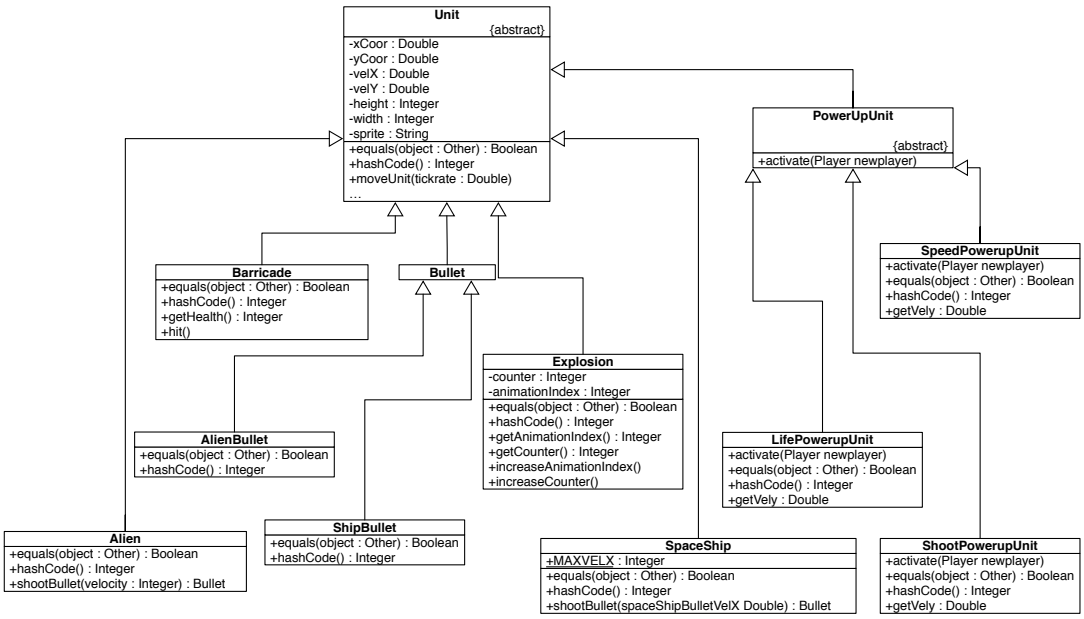

Figure 4: Unit UML Class Diagram

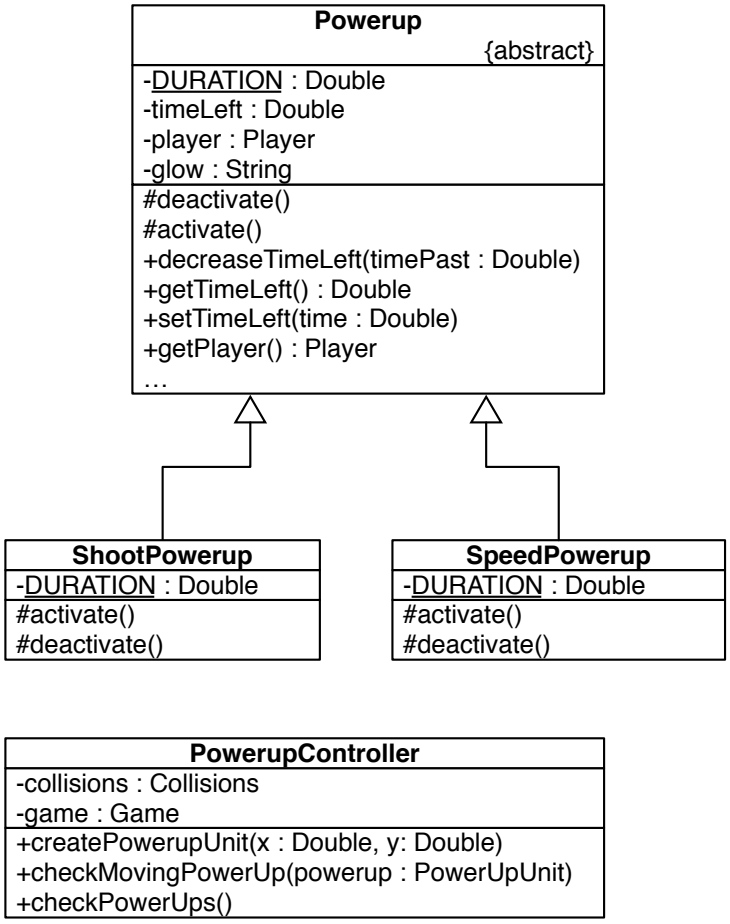

Figure 5: Powerup UML Class Diagram

1 Powerup Functional Requirements

A list of functional requirements considered for the implementation of powerups using the MoSCoW method described in the previous section.

1.1 Must Haves

Powerups must meet the following requirements:

- Aliens shall randomly drop a powerup upon death.
- A Powerup buff shall be applied upon collision with the spaceship.
- A Powerup shall disappear upon collision with the spaceship.

1.2 Should Haves

Powerups should meet the following requirements:

- Player velocity shall increase for a feasible amount of time upon collision with a Speed Powerup.
- Player shooting speed shall increase for a feasible amount of time upon collision with a Shooting Powerup.
- Player life shall increase upon collision with a Life Powerup.
- Player life shall not increase if the maximum amount of 5 lives has been reached.
- A Powerup shall have a custom sprite based on its type.
- A Powerup shall not disappear upon collision with a bullet.
- A Powerup shall not disappear upon collision with an alien.

1.3 Could Haves

Powerups could meet the following requirements:

- A Powerup shall explode upon hitting a barricade.

1.4 Would/Won't Haves

Powerups won't meet the following requirements:

- A Powerup shall explode upon colliding with a bullet.