Group 15
Assignment 4
Bubble Bobble

Exercise 1 - Your wish is my command

1.1 Requirements

Must haves

- A fruit shall be dropped by an enemy, when the enemy is killed.
- The player shall collect the fruit by walking into the fruit.
- The players score shall be increased with 200 extra points if the player collects the fruit.
- The fruit shall disappear when collected by the player, or after 5 seconds.
- A power up shall spawn at a random position in the level.
- The player shall activate the power up by moving into the power up.
- Power ups shall give increased movement speed to the player for 20 seconds.
- Power ups shall appear randomly within the map, every 30-60 seconds.
- A power up shall disappear, if it is not collected within 10 seconds.
- The player shall start with 3 lives.
- The players lives shall be displayed in the top left corner.
- The players lives shall decrease with one when the player dies.
- The game shall end when the player has no lives left.

Should haves

- The score of a fruit shall depend on the time before it's picked up.
- Having more than 1 kind of fruit.
- Power ups shall move through the level.

1.2 Analysis

For this assignment, we started off by deciding the requirements it should fulfil. Because the fruits are subjected to gravity, placing this object in our hierarchy was obvious. To keep extensibility easy, we made a subclass of fruit for each type of fruit. To help with keeping track of the responsibilities and collaborations we made these CRC cards.

Fruit superclass: Gravity subclass: Banana, Cherry	
Knows how long it has been in the game	
Updates its location according to its speed	GameObject
Flag itself to be removed when collides with player	Player

Cherry superclass: Fruit subclass: none	
Score multiplier	Fruit
Load correct textures	Assets

Banana superclass: Fruit subclass: none	
Score multiplier	Fruit
Load correct textures	Assets

Powerup superclass: GameObject subclass: none	
Knows how long it is active	
Updates it's alive time until active time is reached, then it is flagged for removal.	GameObject
Flag itself to be removed when collides with player	Player

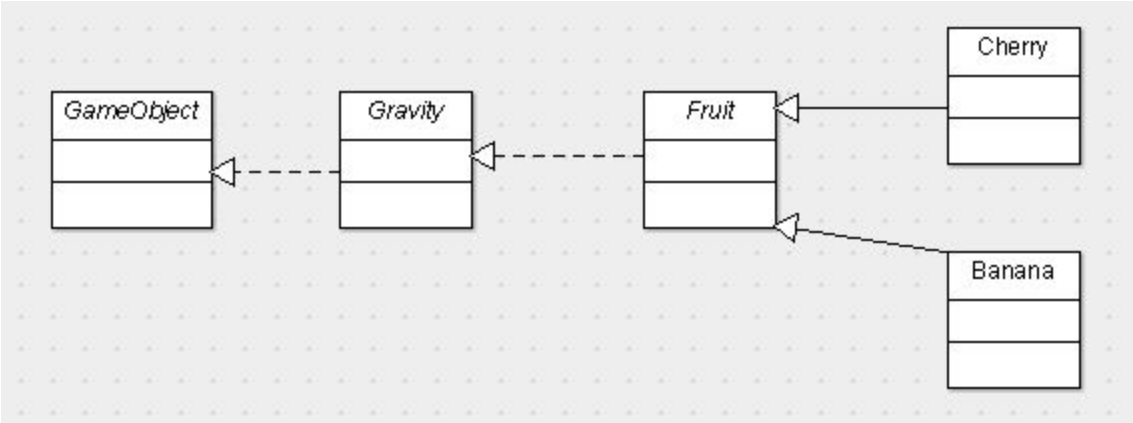

Figure 1 - Class diagram for implementation of the Fruit feature.
The relevant classes for this extension are shown above. Fruit is an abstract subclass of Gravity. Cherry and Banana are implementations of the abstract class Fruit.

Figure 2 - Class diagram for implementation of the Powerup feature.

Exercise 2 - inCode

2.1 inCode result
The result of our inCode analysis is located in our git repo:
GroupFiles/SEM_inCode_snapshot.result.

2.2

In code detected three design flaws, of which two in the classes floor and wall. Both classes, according to inCode, violated polymorphism because the implementation of update and handleCollision are left blank.
However inCode marks this as flaws, it is not.
It is true that it is in general bad practice to make subclasses that do not implement all the methods from the superclass. But in this case it is a necessary need. All game objects are stored in an array. During every cycle this array is looped through and on every game object the draw, update and handleCollision methods are called. Since game objects may collide with a wall, or a floor, but not the other way around, it makes perfect sense to implement these methods blank.

The other flaw that inCode detected was that the Assets class is a data class. This is considered bad practice in general, you don't want a class that is just there to hold values. But in this specific case, we think it is allowed. The Assets class is a singleton class that keeps the textures in memory. Because it's a singleton, we can be sure that the textures are only loaded once, and can be reused everywhere they're needed. If we stored the textures in the classes that need them (like we used to do until sprint 3), then every time a new object is made during run-time the texture had to be loaded and stored in memory again. This design choice makes sure that is not happening, thus this is an improvement instead of a design flaw..

After analysing the code, we found a design flaw in the enemy-classes. The enemy-classes violated the DRY (don’t repeat yourself) principle. This was solved by removing the code in the

handleCollision method from both the weak and strong enemy, and moving it to the abstract enemy class.

Furthermore we found a design flaw in the class fruit, that violated the single responsibility and the interface segregation principle. Fruit was a direct descendent of game object and was responsible for multiple fruits and the scores that come with it. We have solved this by creating an abstract class fruit that handles the physics and collisions, and children (banana and cherry) which determine the score and texture.

Finally, we thought there was another design flaw in our game which inCode did not detect. The 'LogicController' class was more or less a god class. It knew everything about our game, it had a list of all game objects, it handled updating them, it took care of the score and the game state, it rendered the game to the screen. Long story short, the class had too much responsibilities, thus we refactored it into three separate classes: GameController, Level and LevelRenderer. The GameController's only responsibility is handling the state of the game (like loading a level or detecting the pause/play key). The LevelRenderer handles the drawing. It is passed a level object, and it handles the entire rendering and drawing of that level. Finally we have the Level object. This handles the state of the current level that is played. It contains all GameObjects, and it handles the updating and collision-checking of the items in game.