Assignment 1

Daan Vermunt, Matthijs Rijlaarsdam, Arjo van Ramshorst,
Eric Dammeijer & Wouter Zirkzee

Group 15, TA : Aaron Ang, Supervisor : Alberto Bacchelli

Excersie 1 - The Core

1.

The goals of the system being designed are:

- To create a game that is fun to play and gradually more challenging.
- Designed according to the responsibility-driven-design principles.

After analysing the requirements specification the following noun phrases are collected and separated into: obvious classes, uncertain candidates and nonsense.

Obvious classes	**Uncertain classes**	**Nonsense**
Walls	Levels	Playing field
Platforms	Power-ups	2d-perspective
Player	Movement	Fire bubble
Enemy	Jump	Die
Bubbles	Gravity	Collide
Bubble containing object		Points
Game		Time

The noun phrases are divided into conceptual entities and physical objects.

Conceptual entities	**Physical objects**

Walls	Movement
Platforms	Jump
Player	Gravity
Enemy	Die
Bubbles	Points
Bubble containing object	Fire
Game	Levels
Power-ups	Time

The values of attributes are:

- Movement is x-location and y-location. Same goes for jumping and gravity.

- Points is a value of player.
- Die is an attribute that can be a value of a player or an enemy in the form of a binary value 1 / 0.
- Time is an implicit attribute of the system.

After this analysis the candidate classes are:

Candidate classes	
Walls	Game
Platforms	Power-ups
Player	
Enemy	
Bubble	
FilledBubble	

Now it is time to apply the **Class-Responsibility-Collaboration Cards**.

Wall

Superclass(es): Immutable object	
Subclasses:	
Immutable	
Blocks movement	
Contains level	

Platform	
Superclass(es): Immutable object	
Subclasses:	
Immutable	
Can’t fall through	

After discussing the classes **Wall** and **Platform** we came to the conclusion that they share the property immutable which is essential for both classes. Thus we introduced an abstract class **Immutable object.** In both cases there are no subclasses since they are both a specific (sub)class of immutable object. At the moment we don’t plan to implement different kinds of walls or platforms.
There are also no collaborating classes we could think of.

Player	
Superclass(es): GravityObject	
Subclasses:	
Gravity	Platform
movement	Wall, FilledBubble
Fire bubble	Bubble

Enemy
Superclass(es): GravityObject

Subclasses:	
Gravity	Platform
movement	Wall, Bubble

After considering that both Player and Enemy are subject to gravity. We decided to introduce the abstract class GravityObject. Furthermore the Player can fire bubbles and interact with walls, FilledBubbles and Platforms. The Enemy interacts with the Platforms, wall and bubbles.

Bubble	
Superclass(es): Floating object	
Subclasses:	
Floats	
Catch an enemy	Enemy

FilledBubble	
Superclass(es): Floating object	
Subclasses:	
Floats	
Collides with the player	Player

After considering that both Bubble and FilledBubble have the special property that they float and move in the +-y direction. We introduced the abstract class floating object.

Game	
Superclass(es): Floting object	
Subclasses:	
Retrieves the information and displays it.	All other classes

Now it is time to discuss the **Responsibilities** of all the objects.

The responsibilities of a **wall** object are:

- Stops other objects from moving through it
- Contain the level

The responsibilities of a **platform** object are:

- If an object is on the platform, they are not subject to gravity

The responsibilities of a **player** object are:

- Can move in the level
- Can die if it collides with an enemy
- Can shoot bubbles
- Can kill an enemy by colliding with a filled bubble

The responsibilities of an **enemy** object are:

- Kill the player
- Can move in the level
- Can be contained by a bubble

The responsibilities of a **bubble** object are:

- Float in the level
- Contain an enemy
- Disperse after a set amount of time

The responsibilities of a **filledbubble** object are:

- Float in the level
- Disperse after a set amount of time which releases the contained enemy

The responsibilities of the **game** class are:

- Gather all information and display it
- Provide means to pause and continue the game

Now we can compare this design to the actual implementation.

In our implementation we made correct use of polymorphism and abstract classes. We identified the classes which have a “kind-of” hierarchy. However in our implementation we found out that a game-class is not sufficient. The framework we picked allowed us to make better use of MVC. We implemented one main class game, which gets the views from the logic controller class. Furthermore we have the class GameObjects in our implementation. This class is the highest level in the hierarchy. All game-objects are subclasses from this class. After considering and learning about the responsibility-driven-design principles. We have refactored most of our code in the logic-controller. Most of the code is placed in the different game-objects classes to make sure the code is where it belongs.

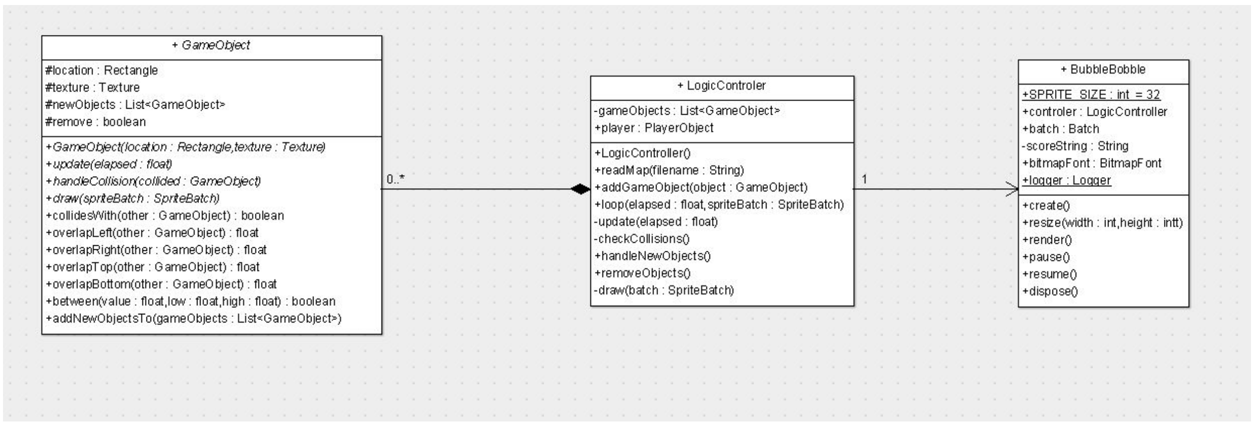
2.

Our main classes in terms of responsibilities and collaborations are LogicController, GameObject and BubbleBobble. The LogicController handles, keeps track and updates of all the GameObjects, it adds new objects or removes objects that are no longer needed. It also invokes the draw method and the handleCollision method for each pair of objects, but does not handle the collisions inside the LogicController. So LogicController has a lot of responsibility to update, draw and keep track of the GameObjects. The class GameObject itself is the highest level of the our object hierarchy, it has a lot of responsibilities for all of the subclasses. BubbleBobble collaborates with the ApplicationListener of the library which we use, which is really important and has a lot of responsibilities like the actual rendering of the game. It also makes use of the Logiccontroller.

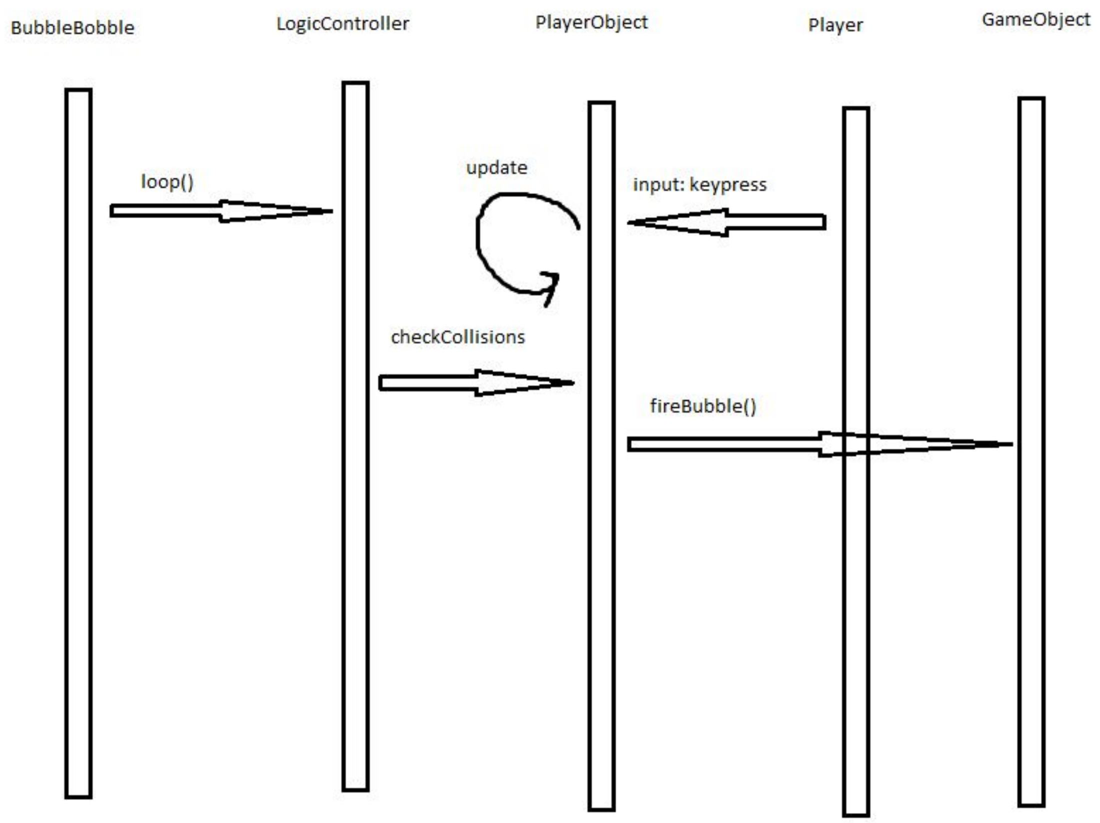
3.

In our implementation there are no classes that could be merged or removed. We made sure to put most of the responsibility in the abstract super classes, such that the subclasses only have little extra responsibility over there superclass. By implementing in this fashion, most of the logic, attributes and behavioural elements are not defined in the specific game-objects, which is conform the responsibility-driven-design principles.

4.


+ GameObject
#location : Rectangle
#texture : Texture
#newObjects : List<GameObject>
#remove : boolean
+GameObject(location : Rectangle,texture : Texture)
+update(elapsed : float)
+handleCollision(collided : GameObject)
+draw(spriteBatch : SpriteBatch)
+collidesWith(other : GameObject) : boolean
+overlapLeft(other : GameObject) : float
+overlapRight(other : GameObject) : float
+overlapTop(other : GameObject) : float
+overlapBottom(other : GameObject) : float
+between(value : float,low : float,high : float) : boolean
+addNewObjectsTo(gameObjects : List<GameObject>)
0..*
+ LogicControler
-gameObjects : List<GameObject>
+player : PlayerObject
+LogicController()
+readMap(filename : String)
+addGameObject(object : GameObject)
+loop(elapsed : float, spriteBatch : SpriteBatch)
-update(elapsed : float)
-checkCollisions()
+handleNewObjects()
+removeObjects()
-draw(batch : SpriteBatch)
1
+ BubbleBobble
+SPRITE_SIZE : int = 32
+controler : LogicController
+batch : Batch
-scoreString : String
+bitmapFont : BitmapFont
+logger : Logger
+create()
+resize(width : int,height : intt)
+render()
+pause()
+resume()
+dispose()

5.



Exercise 2 - UML

a. Composition is a “stronger” association. It is used when something is a part of something bigger, but is useless on its own. The relation is referred to as the owns relation. Aggregation is another form of association, but a “weaker” association. It is also used when something is a part of something bigger, but is still a valued component by itself. It is called the has-a relation.

   In our project we use composition between the LogicController and the GameObjects. Without the logic controller the objects wouldn’t do anything, and the life-span of our objects is limited by our logic controller. So the LogicController owns the GameObjects

b. We do use parameterized classes, mostly with lists of GameObjects. Because we know everything inside the list is an instance of GameObject, we can use all the methods our abstract class GameObject has. For instance, by doing this we can easily access the location of the Rectangle of each subclass which is used in the game.

c. Between our objects classes we use polymorphism and realizations. Our main object GameObject is abstract, we partly realize this class with Immutable-, Floating- and

GravityObject but these 3 classes are still abstract and are further realized in the rest of our objects: Wall-, Floor-, Bubble-, FilledBubble-, Enemy- and PlayerObject. Thus GameObject has 3 child classes, and ImmutableObject, FloatingObject, GravityObject each have 2 child classes. We chose this hierarchy, because doing so we factored out all common behaviour and divided them in “categories”.

Exercise 3 - the logger

Requirements Logger

- There can never be more than one instance of the logger.
- Every class must be able to reference that logger.
- When a message is send to the logger, it should start with the DateTime.
- A log message should also contain the name of the class that send the message to the logger.
- The log messages should be written to a file immediately.

Design choices

We decided to use the singleton design pattern as the base of our logging class. This design pattern is often frowned upon in the Object Oriented world, since it defies almost everything that object oriented stands for, but logging is (in our opinion) one of the few uses of the pattern that is allowed. The only responsibility of the logger is writing to a file.