Group 15
Assignment 3
Bubble Bobble

Assignment 1- 20 time reloaded

For this assignment, some deviation was made from the inital requirements document. The realisation was made that the simple enemies should not have pathfinding towards the player, and these should instead be a different enemy type. Therefore, the classes involving the enemies were restructured to allow there being multiple enemies, as well as the enemies being angry.
A stronger enemy that does search a path towards the player was implemented as well, to still furfill the requirements document.
The following responsibility cards involve the responsibilitys involved in these new features:

Enemy Superclass: GravityObject Subclasses: SimpleEnemy, StrongEnemy	Collaborates with:
Knows if its angry or normal	FilledBubbleObject
Updates its location according to its speed and angryness	
Knows its direction	
Becomes normal after 10 seconds if angry	
Changes texture according to direction and angryness	Assets

Since all enemies should be able to be angry and keep track of things like location and direction, these responsibilities should be defined in an abstract superclass, which we'll call Enemy.

SimpleEnemy Superclass: Enemy	Collaborates with:
Changes direction if collides with a wall	WallObject
Load correct textures	Assets

The SimpleEnemy has a simple pathfinding function, which is triggered after colliding with a wall. It's textures should be distinct, and are loaded by the Assets class.

Strong Enemy Superclass: Enemy	

Changes direction based on player location and collision with wall	WallObject, Player
Jumps after changing direction	
Load correct textures	Assets

The StrongEnemy has a more complicated pathfinding algorithm, and should know where the player is, as well as be able to jump. Since it loops through all game objects during collision handling, the player can simply be selected from this list. This class should also load the correct distinct textures.

Since both (and in fact, all) subclasses of enemy need to load distinct textures, an abstract method loadTextures() should be defined, to assure all subclasses adhere to this behaviour.

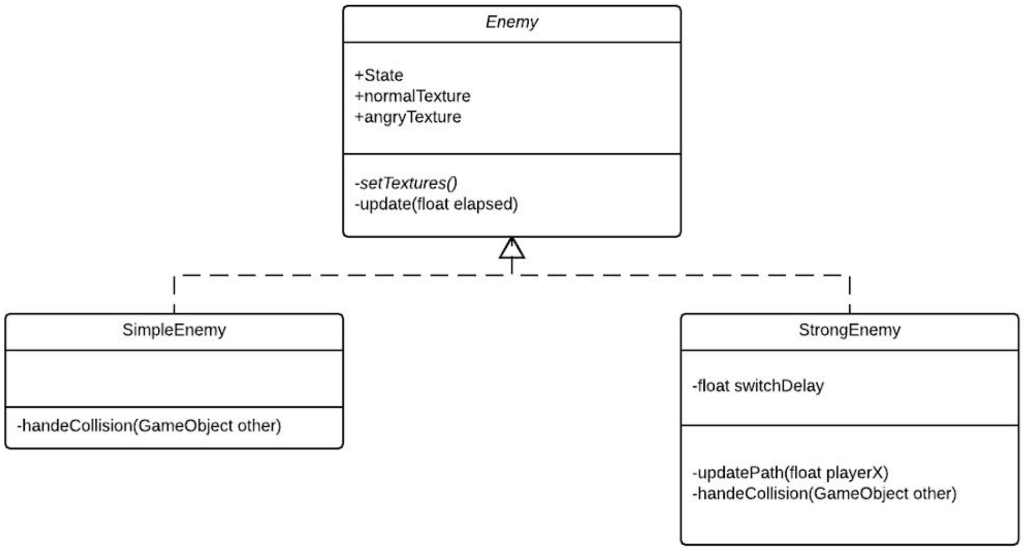


Fig 1. UML diagram of the enemy extension

The Enemy class defines a state (angry/normal), which is updated as necessary in the update method. Here the location is also updated according to the speed (which is also dependent on the state.). The handleCollision method implementation of simpleEnemy (which is defined abstract in the GameObject superclass) provides a simple pathfinding for SimpleEnemy: turn if you bump into a wall.

For the StrongEnemy, an extra method updatePath is used to seek a path towards the player, whose locations is received in the handleCollision method. It also has a switchDelay to not change direction too quickly. The StrongEnemy jumps when switching direction.

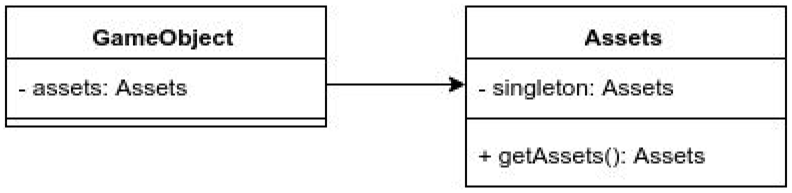
Exercise 2

Design pattern 1 - Singleton:

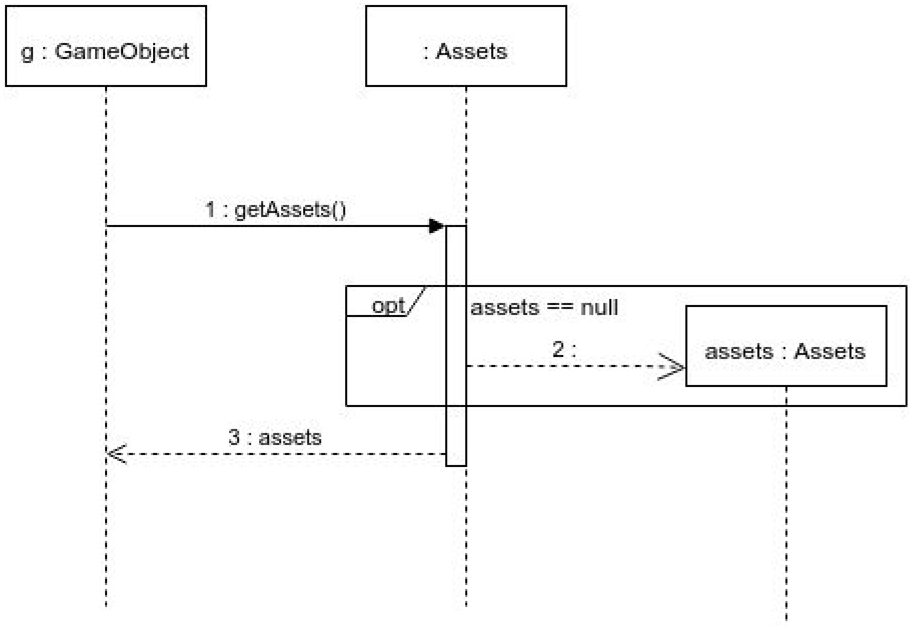
1. *Write a natural language description of why and how the pattern is implemented in your code (5 pts).*

We decided to use the Singleton pattern for asset management. Since every texture and sound only needs to be stored once in memory, a singleton object is perfect for these values. It ensures that textures and sounds are only loaded once, and that every Object can get a reference to it if needed.

2. *Make a class diagram of how the pattern is structured statically in your code (5 pts).*



3. Make a sequence diagram of how the pattern works dynamically in your code (5 pts).



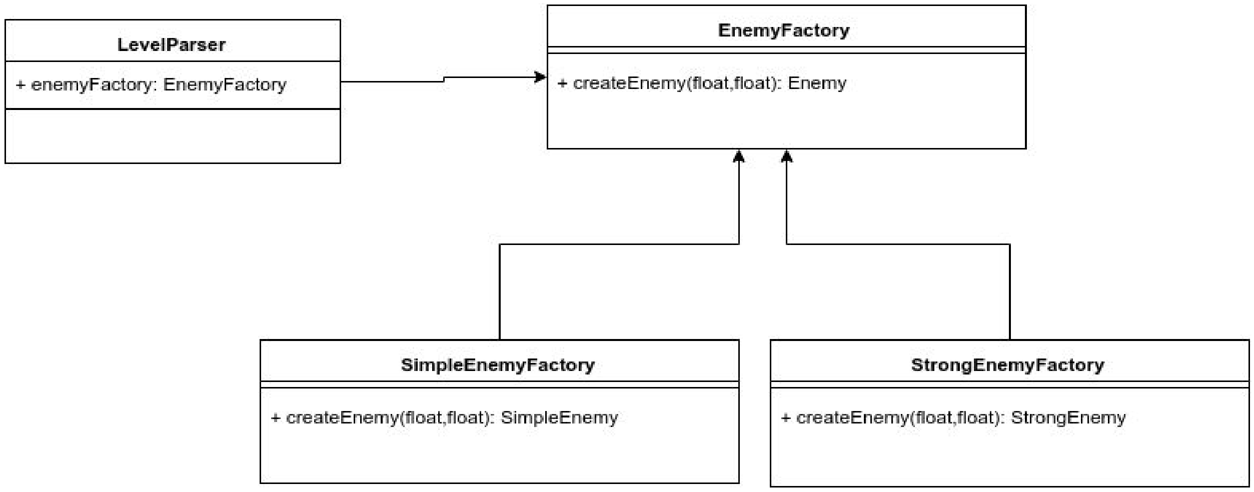
Design pattern 2 - Factory pattern:

1. *Write a natural language description of why and how the pattern is implemented in your code (5 pts).*

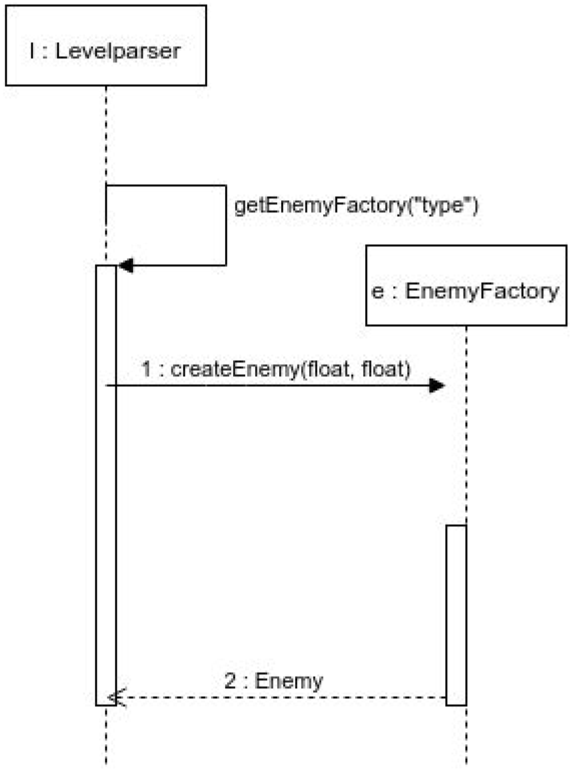
For the creation of enemies we decided to use the factory pattern. The reason this pattern is useful for that, is because each level has its own type of enemy. Since we don't want to write code that checks which object should be created, at multiple places, we chose to implement an

EnemyFactory interface, with two subclasses (currently we have two enemies): SimpleEnemyFactory and StrongEnemyFactory. Wherever we need to create a new enemy object, we can just use the EnemyFactory.create() method, and it will create the proper enemy for that level. Which factory is used is set during the parsing of the level file.

2. *Make a class diagram of how the pattern is structured statically in your code (5 pts).*


LevelParser
+ enemyFactory: EnemyFactory
EnemyFactory
+ createEnemy(float,float): Enemy
SimpleEnemyFactory
+ createEnemy(float,float): SimpleEnemy
StrongEnemyFactory
+ createEnemy(float,float): StrongEnemy

3. *Make a sequence diagram of how the pattern works dynamically in your code (5 pts).*



Exercise 3

1) Good and bad practices are recognized by comparing both cost and duration to the average of all projects in the research portfolio. More specifically by comparing it to projects of the same size.
2) The finding that Visual Basic is in the Good Practice group is not so interesting because a company cannot easily change their main programming language. While other findings are possible to implement with relatively minor effort.
3) They could have checked for code complexity, in this case simple code, for example short functions, would have been a good practice.

They could have checked for test driven development, which could be in the bad practice part of the spectrum because it could increase cost and duration of a project.

They could check for duplicate code, which would be a bad practice since duplicate code makes the code less agile and therefore more expensive and time consuming to change.

4) "Many team changes, inexperienced team", meaning that there is no consistent team, with experienced people who do project. The problem with a changing team is that every time a new team member is introduced to the project, it takes time and money to get him or her to understand the code and design they are expected to work on. Inexperienced team members are a problem because they usually take longer to complete task. They also tend to make more mistakes while working on the project, which again takes time and money to repair.

"New technology, Framework solution", meaning trying to solve problem with cutting edge technology. This often lead to problems because the new technology is not well tested and often unstable. This leads to problems that the team often cannot directly solve themselves, which cause delays making the duration of the project longer.

"Dependencies with other systems", meaning the project depends on other systems to properly work. This becomes a problem when the system the project depends on fails for some reason. This causes the team to find a new solution for a problem they solved earlier, costing time and money, thus making it a bad practice.