Assignment 3

Asteroids

Created by group OP27-G59 for the course
CSE2115 Software Engineering Methods
of the Computer Science curriculum
at the Delft University of Technology

Group members:

Onur Gökmen
Joseph Catlett
Irtaza Hashmi
Helena Westermann
Ceren Uğurlu

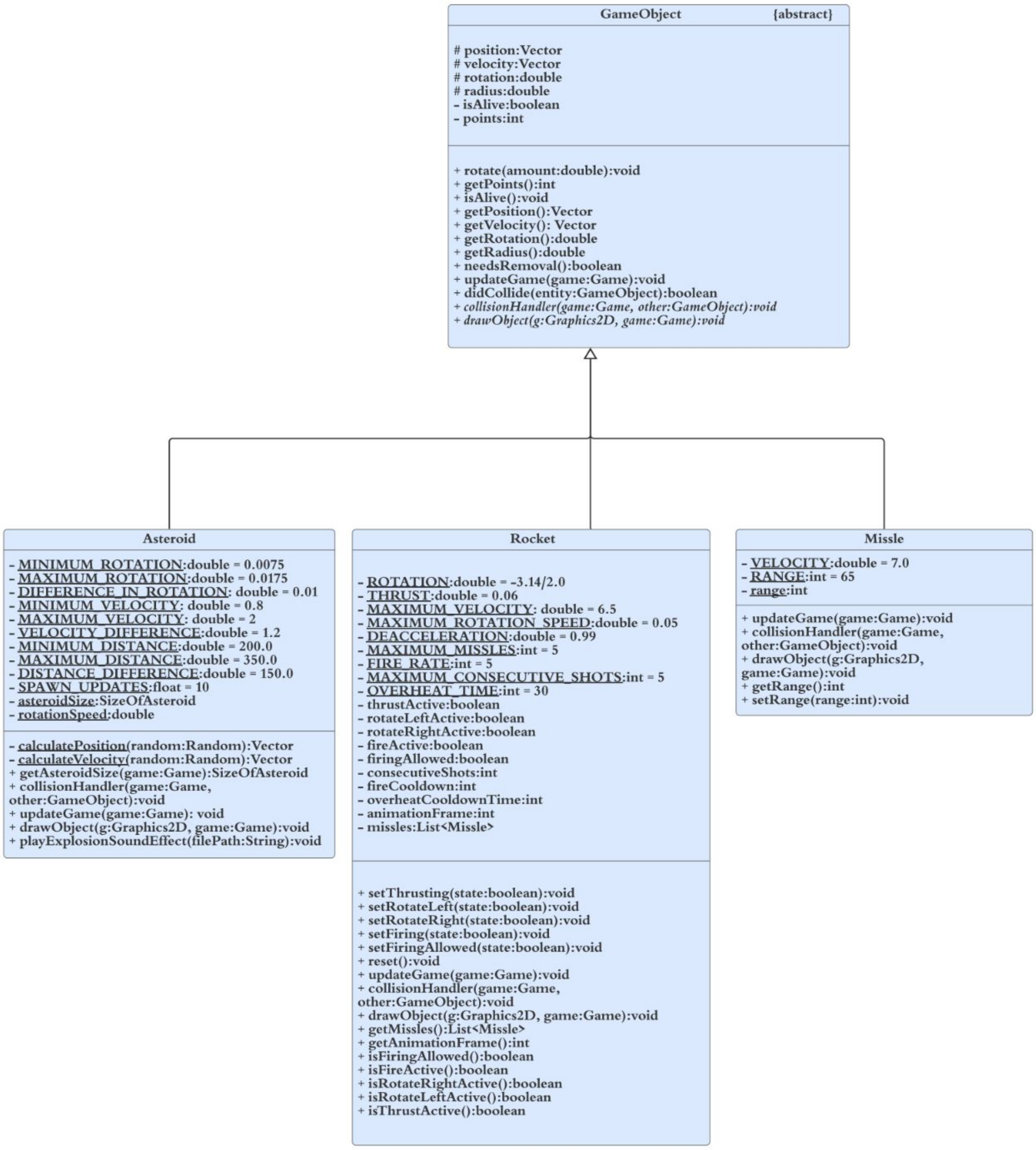
TABLE OF CONTENTS

1

Design Patterns

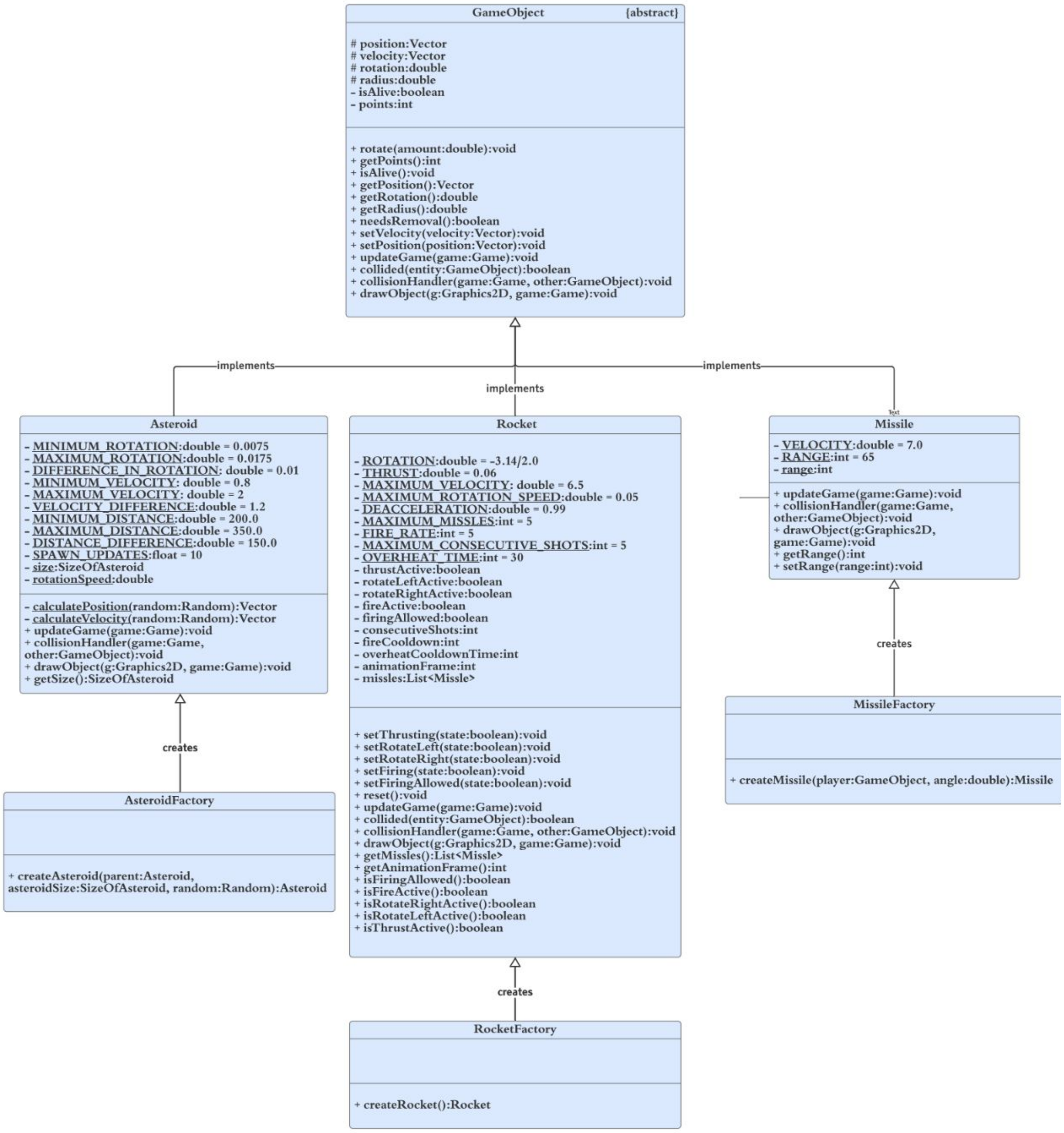
1. Template method

All of the game objects share the same structure and algorithms, even though the implementations of the separate steps in the algorithm vary. To avoid repetition within the code while still allowing for the differences within these objects we decided to implement a template method called GameObject. All of the methods that the Missile, Asteroid and Rocket have in common are implemented within the abstract class GameObject (for example, rotate, getPosition, getVelocity). However, there are also methods where each class has a different implementation, like collisionHandler and drawObject. These are abstract in the template method and have to be overridden, whereas the other methods can be overridden if necessary.


GameObject
{abstract}
# position:Vector
# velocity:Vector
# rotation:double
# radius:double
- isAlive:boolean
- points:int
+ rotate(amount:double):void
+ getPoints():int
+ isAlive():void
+ getPosition():Vector
+ getVelocity(): Vector
+ getRotation():double
+ getRadius():double
+ needsRemoval():boolean
+ updateGame(game:Game):void
+ didCollide(entity:GameObject):boolean
+ collisionHandler(game:Game, other:GameObject):void
+ drawObject(g:Graphics2D, game:Game):void
Asteroid
- MINIMUM_ROTATION:double = 0.0075
- MAXIMUM_ROTATION:double = 0.0175
- DIFFERENCE_IN_ROTATION: double = 0.01
- MINIMUM_VELOCITY: double = 0.8
- MAXIMUM_VELOCITY: double = 2
- VELOCITY_DIFFERENCE:double = 1.2
- MINIMUM_DISTANCE:double = 200.0
- MAXIMUM_DISTANCE:double = 350.0
- DISTANCE_DIFFERENCE:double = 150.0
- SPAWN_UPDATES:float = 10
- asteroidSize:SizeOfAsteroid
- rotationSpeed:double
- calculatePosition(random:Random):Vector
- calculateVelocity(random:Random):Vector
+ getAsteroidSize(game:Game):SizeOfAsteroid
+ collisionHandler(game:Game, other:GameObject):void
+ updateGame(game:Game): void
+ drawObject(g:Graphics2D, game:Game):void
+ playExplosionSoundEffect(filePath:String):void
Rocket
- ROTATION:double = -3.14/2.0
- THRUST:double = 0.06
- MAXIMUM_VELOCITY: double = 6.5
- MAXIMUM_ROTATION_SPEED:double = 0.05
- DEACCELERATION:double = 0.99
- MAXIMUM_MISSLES:int = 5
- FIRE_RATE:int = 5
- MAXIMUM_CONSECUTIVE_SHOTS:int = 5
- OVERHEAT_TIME:int = 30
- thrustActive:boolean
- rotateLeftActive:boolean
- rotateRightActive:boolean
- fireActive:boolean
- firingAllowed:boolean
- consecutiveShots:int
- fireCooldown:int
- overheatCooldownTime:int
- animationFrame:int
- missles:List<Missle>
+ setThrusting(state:boolean):void
+ setRotateLeft(state:boolean):void
+ setRotateRight(state:boolean):void
+ setFiring(state:boolean):void
+ setFiringAllowed(state:boolean):void
+ reset():void
+ updateGame(game:Game):void
+ collisionHandler(game:Game, other:GameObject):void
+ drawObject(g:Graphics2D, game:Game):void
+ getMissles():List<Missle>
+ getAnimationFrame():int
+ isFiringAllowed():boolean
+ isFireActive():boolean
+ isRotateRightActive():boolean
+ isRotateLeftActive():boolean
+ isThrustActive():boolean
Missle
- VELOCITY:double = 7.0
- RANGE:int = 65
- range:int
+ updateGame(game:Game):void
+ collisionHandler(game:Game, other:GameObject):void
+ drawObject(g:Graphics2D, game:Game):void
+ getRange():int
+ setRange(range:int):void

2. Factory method

The factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. Missile, Asteroid and Rocket classed are extended from abstract GameObject class. We created AsteroidFactory, MissileFactory and RocketFactory classes that make their respective objects. The factory classes allow us to create objects without having to specify the exact class. We used the “create...” method in GameObjectFactory for creating the needed game object such as asteroids, missiles and rocket.


GameObject
{abstract}
# position:Vector
# velocity:Vector
# rotation:double
# radius:double
- isAlive:boolean
- points:int
+ rotate(amount:double):void
+ getPoints():int
+ isAlive():void
+ getPosition():Vector
+ getRotation():double
+ getRadius():double
+ needsRemoval():boolean
+ setVelocity(velocity:Vector):void
+ setPosition(position:Vector):void
+ updateGame(game:Game):void
+ collided(entity:GameObject):boolean
+ collisionHandler(game:Game, other:GameObject):void
+ drawObject(g:Graphics2D, game:Game):void
implements
implements
implements
Asteroid
- MINIMUM_ROTATION:double = 0.0075
- MAXIMUM_ROTATION:double = 0.0175
- DIFFERENCE_IN_ROTATION: double = 0.01
- MINIMUM_VELOCITY: double = 0.8
- MAXIMUM_VELOCITY: double = 2
- VELOCITY_DIFFERENCE:double = 1.2
- MINIMUM_DISTANCE:double = 200.0
- MAXIMUM_DISTANCE:double = 350.0
- DISTANCE_DIFFERENCE:double = 150.0
- SPAWN_UPDATES:float = 10
- size:SizeOfAsteroid
- rotationSpeed:double
- calculatePosition(random:Random):Vector
- calculateVelocity(random:Random):Vector
+ updateGame(game:Game):void
+ collisionHandler(game:Game, other:GameObject):void
+ drawObject(g:Graphics2D, game:Game):void
+ getSize():SizeOfAsteroid
creates
AsteroidFactory
+ createAsteroid(parent:Asteroid, asteroidSize:SizeOfAsteroid, random:Random):Asteroid
Rocket
- ROTATION:double = -3.14/2.0
- THRUST:double = 0.06
- MAXIMUM_VELOCITY: double = 6.5
- MAXIMUM_ROTATION_SPEED:double = 0.05
- DEACCELERATION:double = 0.99
- MAXIMUM_MISSLES:int = 5
- FIRE_RATE:int = 5
- MAXIMUM_CONSECUTIVE_SHOTS:int = 5
- OVERHEAT_TIME:int = 30
- thrustActive:boolean
- rotateLeftActive:boolean
- rotateRightActive:boolean
- fireActive:boolean
- firingAllowed:boolean
- consecutiveShots:int
- fireCooldown:int
- overheatCooldownTime:int
- animationFrame:int
- missles:List<Missle>
+ setThrusting(state:boolean):void
+ setRotateLeft(state:boolean):void
+ setRotateRight(state:boolean):void
+ setFiring(state:boolean):void
+ setFiringAllowed(state:boolean):void
+ reset():void
+ updateGame(game:Game):void
+ collided(entity:GameObject):boolean
+ collisionHandler(game:Game, other:GameObject):void
+ drawObject(g:Graphics2D, game:Game):void
+ getMissles():List<Missle>
+ getAnimationFrame():int
+ isFiringAllowed():boolean
+ isFireActive():boolean
+ isRotateRightActive():boolean
+ isRotateLeftActive():boolean
+ isThrustActive():boolean
creates
RocketFactory
+ createRocket():Rocket
Text
Missile
- VELOCITY:double = 7.0
- RANGE:int = 65
- range:int
+ updateGame(game:Game):void
+ collisionHandler(game:Game, other:GameObject):void
+ drawObject(g:Graphics2D, game:Game):void
+ getRange():int
+ setRange(range:int):void
creates
MissileFactory
+ createMissile(player:GameObject, angle:double):Missile

2

Software Architecture

Our application’s overall architecture is Layered Architecture (see diagram 1). We have chosen this architecture because it fits our application the best. We can clearly separate our application into 3 different layers.

The first layer is the Presentation Layer. In this layer, we provide a view of the application to our users. It consists of the login page, register page, main menu, the high score page and the gamescene itself. After the user successfully goes through authentication, they can play the game.

The Game Logic Layer makes up our second layer. It’s the core layer of our application. In this layer, the game is implemented. We define how the ship moves, how many bullets a ship can take, how fast an asteroid moves etc.

Our last layer is the Data Layer. We use this layer to communicate with the database. We use the database for various kinds of use cases. For example: When a game ends, the user has the option to submit their score along with a nickname. We store the nickname and score for displaying the high score of the game. We also save the user's username and password for authentication.

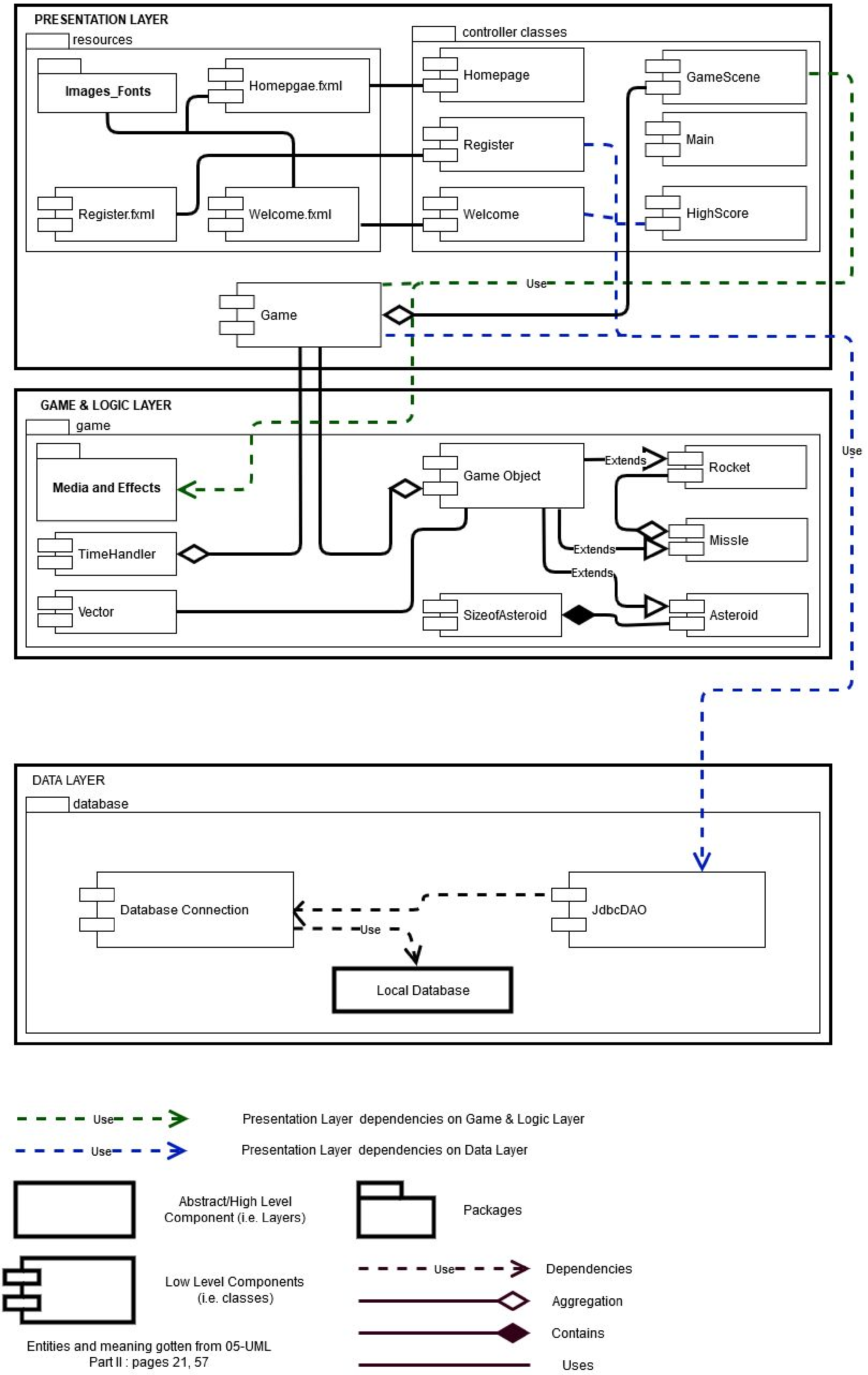
Using a layered architecture has a number of advantages for our application. Those advantages are testability, maintainability and flexibility.

- Testability: Because of the separate layers. We can test our application in isolation. We can test the game logic by using mocks and stubs for ships, bullets and asteroids and see how they interact which each other. We can also test the database in separation. Normally we would need to make a connection to the database for testing the database methods. But again, we can use mocks to test our database methods instead of creating a new connection.
- Maintainability: We can update one layer without affecting the other layers, which is an important advantage in our application. For example, when we update our user interface, it doesn’t affect the game logic layer or the database layer.
- Flexibility: We can change one layer with any other alternative layer. For example, if we would want to switch the Game Logic layer with another alternative layer (an entirely new game), we could do that and there will be no issues with our application.

We have chosen this pattern instead of other patterns because we prioritise testability, maintainability and flexibility. This way we can reuse methods and functions. Other patterns, such as Main programme and subroutines is also focused on reuse of methods and functions, but this pattern is affected by data changes and global variables. As we have a good amount of global variables and data changes, this pattern wouldn't be suitable for our application. The other patterns are not suited for our application as they introduce unnecessary complexity and have more to do with servers and cloud computing.

The main disadvantages are that the performance and complexity of the application depend on multiple layers. One request may pass all the way down to the lowest layer and more abstractions increase the overall complexity. To partly tackle these two problems, we tried to keep the numbers of layers as small as possible. In addition, we tried to make our application more flexible. For instance, the presentation layer is connected to the database layer and because of this connection, a request in the presentation layer can go directly to the database layer, instead of passing through the game logic layer.

To conclude, the three layers in our architecture introduce testability, maintainability and flexibility. As these were the most important to us, these advantages exceed the disadvantages of being dependent on the separate layers. For this reason, the architecture fits our application best.



Entities and meaning gotten from 05-UML Part II : pages 21, 57