Exercise 1

1.

CRC cards:

Unit

-

Player, Asteroid, Bullet

Return location	Vector element
Display sprite	Sprite/render element
Update location (move)	Vector element
Return hitbox/bounding box	Bounding box element

Player

Unit

-

Read keyboard input	Input element
Shoot bullets	Bullet element
Record score	Score element

Asteroid

Unit

-

Split	Asteroid elements
Explode	Animation element

Bullet

Unit

-

Game

-

-

Render Units, score and background	Graphics element
Randomly spawn Asteroids	List with Asteroids
Collision detection	Bounding boxes of the different Units
Despawn Bullets outside the screen	Bullet elements

Steps:

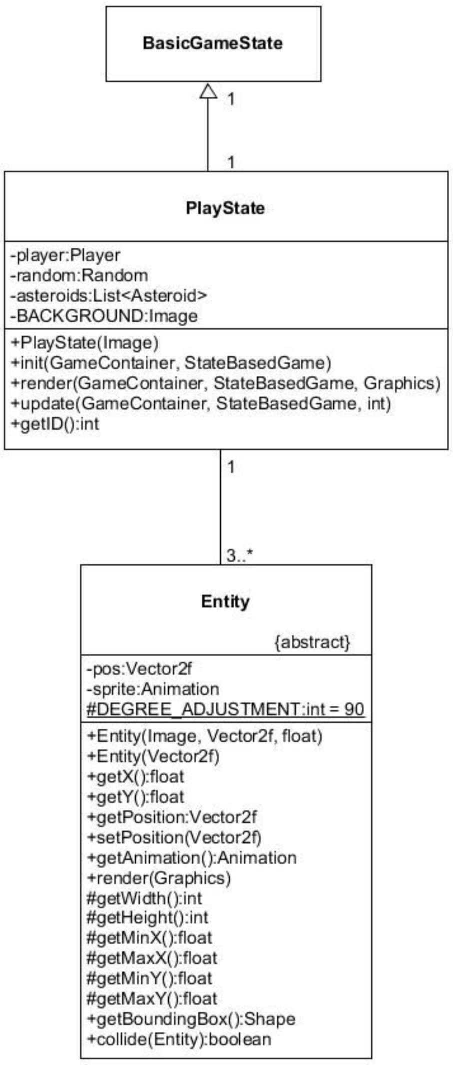
- List all the units in the game: Player, Asteroid, Bullet.
- Design a general class (Unit) for functionality needed by all the units in the game.
- List all actions that can happen: Movement, Collsion
- Design movement for all units. Create bounding boxes
- Make a Game class that updates all the units actions and reactions and renders them on the screen.

2.
The main Classes we implemented in our project are ‘Entity’ and ‘PlayState’. Entity is the main abstract class that has functionality for all the subclasses. PlayState updates most of the movement, and checks the collision of all the Entities currently present.

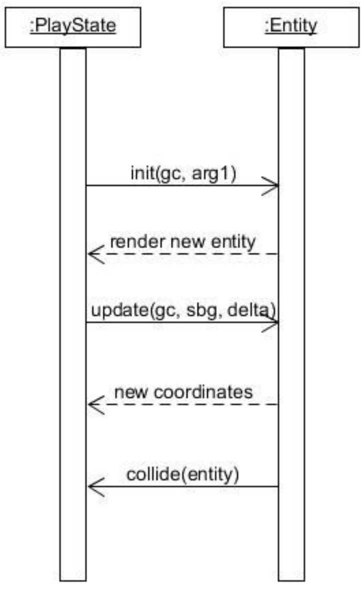
3.
AsteroidsGame has little to no purpose. The class is only there to extend the StateBasedGame class. However it is handy to keep, if quick changes need to be made. It is not really needed, because a StateBasedGame could be created within the Launch class.

The ExplodableEntity class is an adapter for the Entity class and the Player and Asteroid classes. This comes from the design choice to extend all Entities from the Entity class. The added functionality of exploding is only used by the Player and Asteroid class. It is not needed for the Bullet class. Removing this class would mean that the exploding has to be implemented in the Player and Asteroid classes.

4.


BasicGameState
1
1
PlayState
-player:Player
-random:Random
-asteroids:List<Asteroid>
-BACKGROUND:Image
+PlayState(Image)
+init(GameContainer, StateBasedGame)
+render(GameContainer, StateBasedGame, Graphics)
+update(GameContainer, StateBasedGame, int)
+getID():int
1
3..*
Entity
{abstract}
-pos:Vector2f
-sprite:Animation
#DEGREE_ADJUSTMENT:int = 90
+Entity(Image, Vector2f, float)
+Entity(Vector2f)
+getX():float
+getY():float
+getPosition:Vector2f
+setPosition(Vector2f)
+getAnimation():Animation
+render(Graphics)
#getWidth():int
#getHeight():int
#getMinX():float
#getMaxX():float
#getMinY():float
#getMaxY():float
+getBoundingBox():Shape
+collide(Entity):boolean

5.


:PlayState
:Entity
init(gc, arg1)
render new entity
update(gc, sbg, delta)
new coordinates
collide(entity)

Exercise 2

1.

Aggregation: the class(part) that is contained by another class(whole) can exist without being a part a class

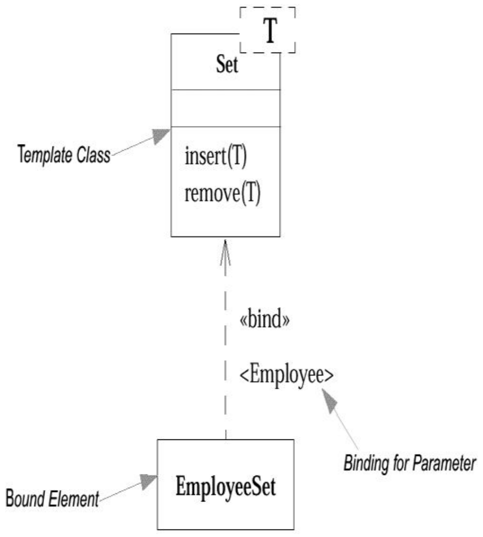
In our source code the Bullet class is used by the Player class. A player has a List<Bullet> where the bullets that the Player shoots are stored.

Composition: class B has no meaning or purpose in the system without class A.

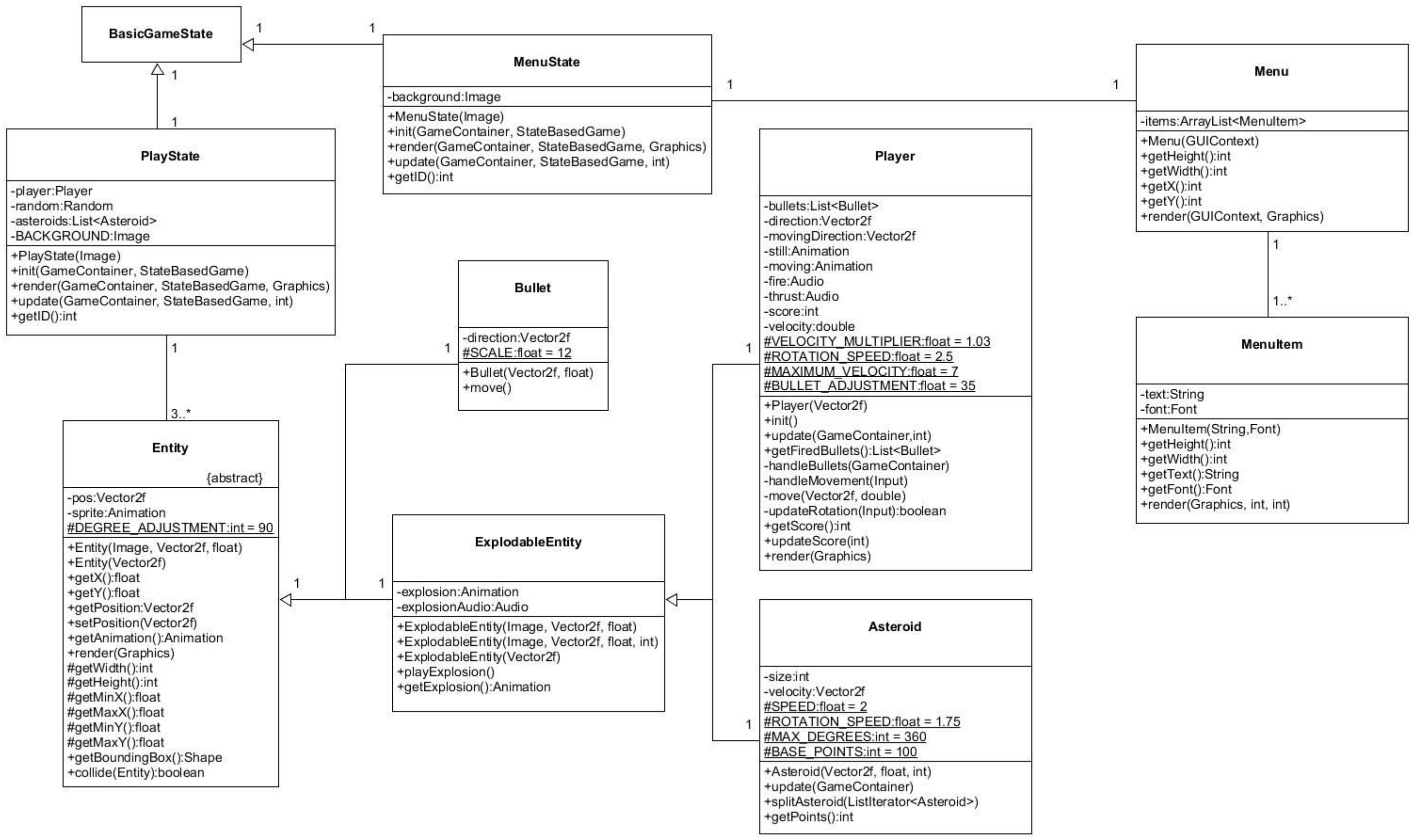
In our source code composition is used by the classes ExplodableEntity, Bullet; these extend the Entity class. The ExplodableEntity has no meaning without being extended by the Asteroid and Player class.

2.

No, there are no parameterized classes in the source code. Parametrized classes should be used when the class needs to be kept general. Different types can be used for the class. For example the Set class in the Java Collections package. In the UML the parameterized class is displayed as shown below.



3.



Exercise 3

1.

The logger feature has been implemented in the latest commit in the master branch (pull request merge #9).

2.

A new Logger class will be created to implement the logging functionality. All three entities (Bullet, Asteroid, Player) will contain an instance of the Logger. The Launch, AsteroidsGame, and PlayState class will also have a personal Logger instance.

The Logger class main function is the log function. This function takes four parameters: a message, an enum Level, a update boolean, and a caller depth. Level can be INFO, WARNING, or ERROR. The caller depth is defaulted to 4. The log function fetches the current time, and the class name and method. It puts this and the parameter data into a string and adds it to the output queue. If the method is called with the update boolean set to true, the method will call its update method, which prints the string to the specified output streams.

The entities call the log function of the Logger class when an event happens that concerns their class. Example: the bullet class will call the log function when a bullet is fired, it will log the coordinates and location of that bullet.

The six classes that make use of the Logger class mostly use the log method with a true update boolean to ensure everything gets printed to the output streams. The Player and PlayState also use the update method directly in their own update methods.

CRC card:

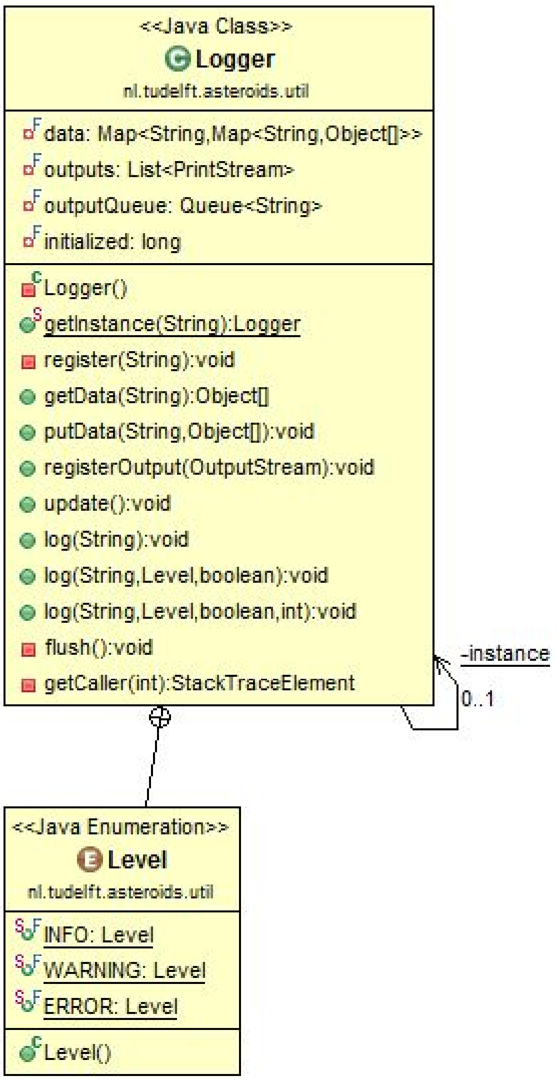
<u>Logger</u>

-

Level (enum), Map, List with PrintStream, Queue

Logging message to output stream	List and Print element
Indicating log level	Level enum
Collecting and storing data	Map element, Queue element

UML:



<<Java Class>>
Logger
nl.tudelft.asteroids.util
data: Map<String,Map<String,Object[]>>
outputs: List<PrintStream>
outputQueue: Queue<String>
initialized: long
Logger()
getInstance(String):Logger
register(String):void
getData(String):Object[]
putData(String,Object[]):void
registerOutput(OutputStream):void
update():void
log(String):void
log(String,Level,boolean):void
log(String,Level,boolean,int):void
flush():void
getCaller(int):StackTraceElement
-instance
0..1
<<Java Enumeration>>
Level
nl.tudelft.asteroids.util
INFO: Level
WARNING: Level
ERROR: Level
Level()