Engineering 1

Group Assessment 2

Architecture Document

Cohort 2

Team 12 & Team 11

Architecture Representations (3a)

Architecture Development

We used a UML class diagram to describe the architecture visually. We used tools such as PlantUML to create the architecture representations. PlantUML is a markup language for UML.

Abstract Representation

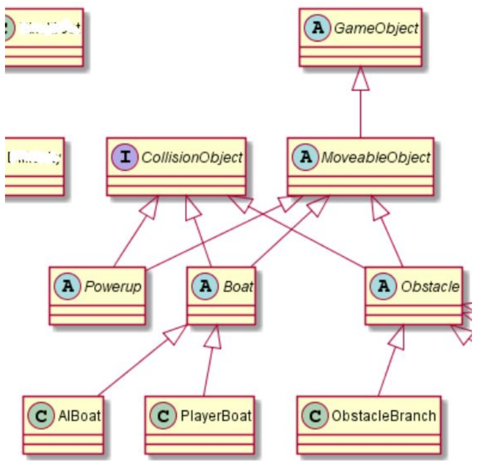
For the abstract representation, we chose to make a simple UML component diagram. This functioned as a backbone to begin implementation and allowed the team to agree on a basic structure. We have grouped classes (abstract classes and interfaces) for different features. This heavy object oriented approach allows the solution to be very scalable due to the modularity so this is useful for future development of existing and new features.

Simplified Component Diagram (See appendix 1)

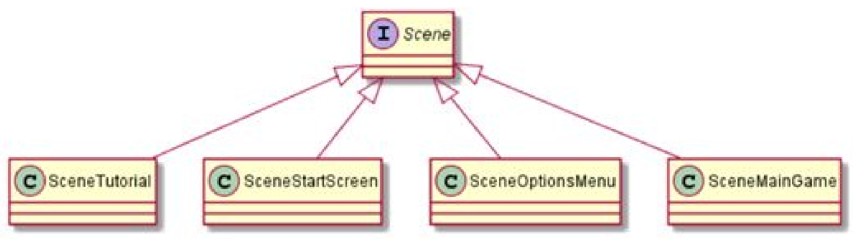
The following abstract representation section contains screenshots of this diagram which is embedded in full in the appendix as well as github with generated PlantUML code. In the class diagram the following symbols are defined as follows:

- A - Abstract Class
- I - Interface
- C - Class

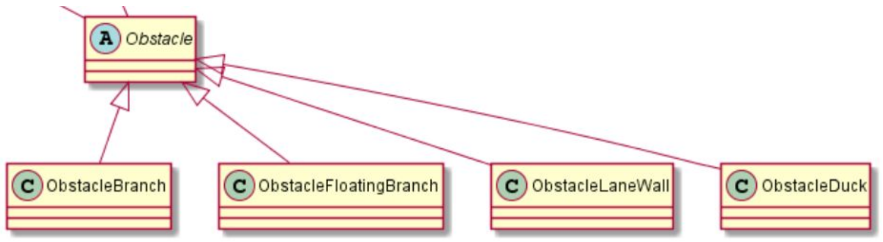
Arrows - represent child classes inheriting from parent classes



The GameObject is what everything visible in the game is derived from. And from the diagram you can see that the main objects in the game Boat and Obstacle inherit from MoveableObject as well as the CollisionObject interface. In Addition, PowerUpPack object inherits from MoveableObject.



For example, for controlling the environment of the game we have scenes which inherit from the main Scene Interface.

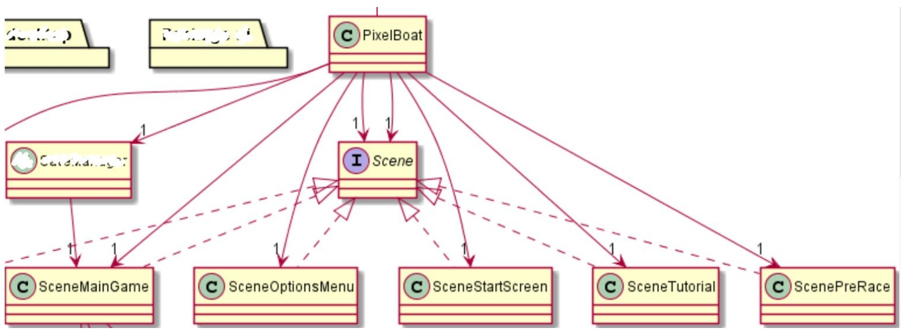


We have a group of classes which inherit from the abstract class Obstacle which helps to share and differentiate attributes between obstacles. This means further obstacles can be added with different properties for example varying speeds and rotation. Both Obstacle duck and branch inherit from abstract class obstacle.

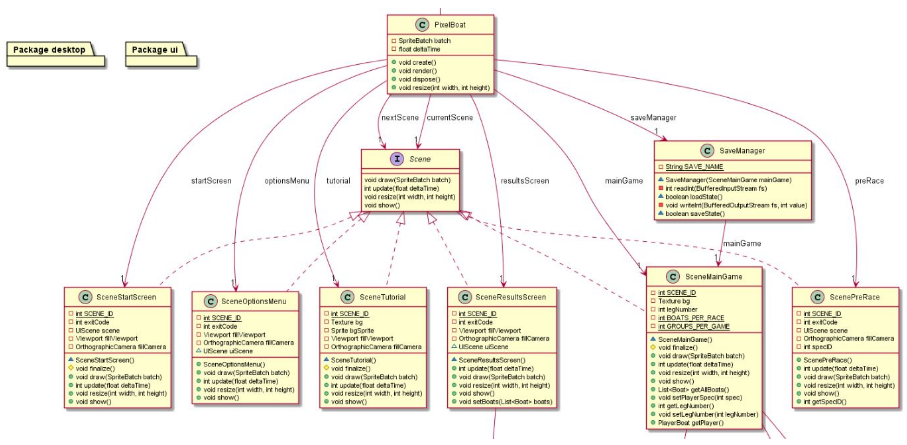
Concrete Representation

The class diagram [2] can represent the updated representation after implementation of each class and the relationship between each class. Therefore you can see how the abstract and concrete relate with the inheritance diagram [3] which shows the attributes and methods of each class. Screenshots from each document are below and the diagrams in full are in the appendix as well as github with generated PlantUML code.

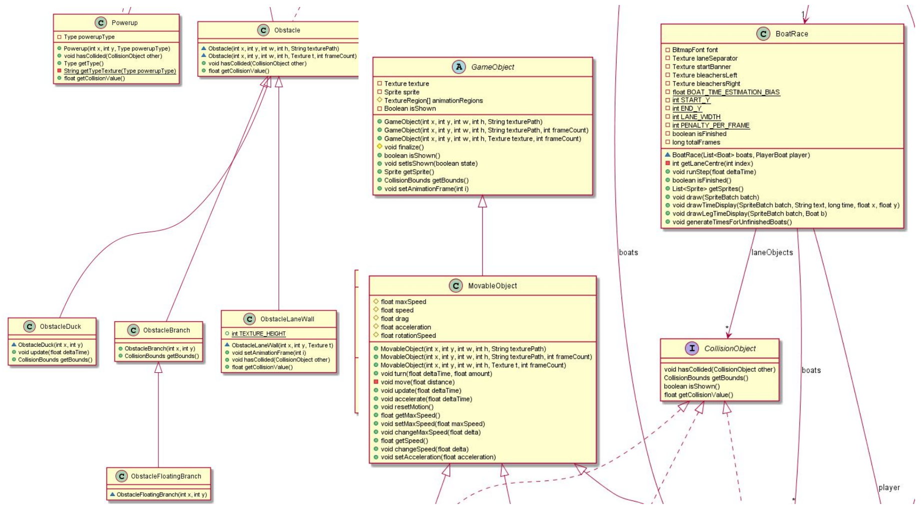
How Scenes relate to each other and PixelBoat (the game class)



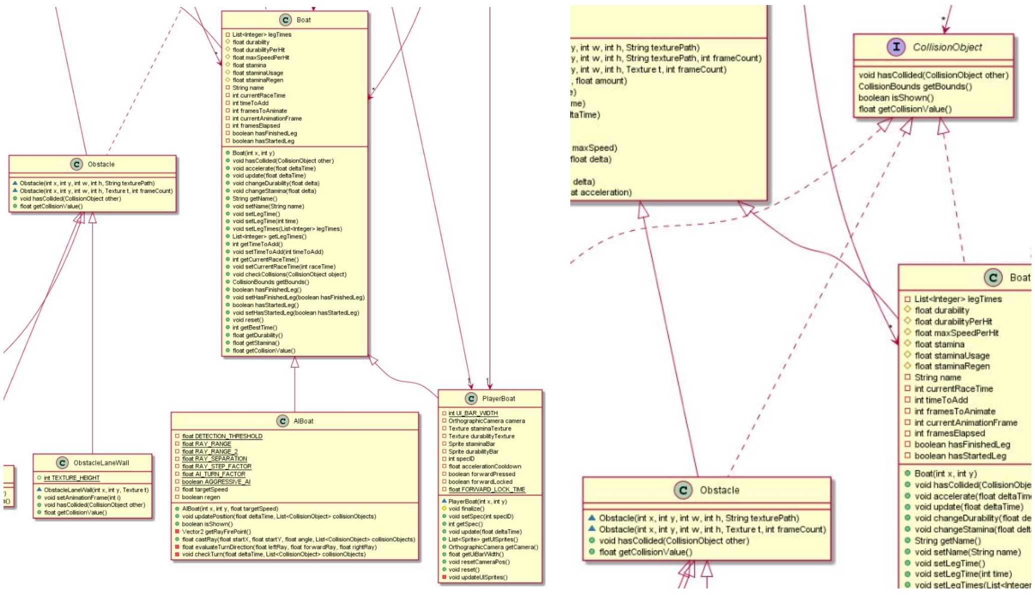
Screenshot from part of the class diagram [2]



Screenshot from part of the inheritance diagram [3]



These sections of the inheritance diagram [3] demonstrate how the Obstacle classes interact with each other and BoatRace.



The Boat class (which is a GameObject and Movable Object) has children PlayerBoat and AIBoat. An example of a CollisionObject (which inherits from CollisionBounds) is an Obstacle and this gives it functionality to collide with other GameObjects.

Justification for Architecture (3b)

Justification for the Abstract Architecture

Object oriented programming was used as it offered a high level of abstraction while fitting with the required language of Java.
The class inheritance structure is a product of the design process the team agreed upon, whereby, all high level features were sketched out, then all common features were placed into a common abstract class. This allowed for the inheritance structure to stay simple and now get bloated with unneeded features.
An example of this would be how it was decided that we would need a class for the player's boat and a class for the AI's boat. These have the common feature of being a boat, thus a common superclass of "Boat" was created.

The above approach was used for all visible elements of the game (boats, obstacles, etc) but it wasn't appropriate for all features. For components such as choosing what to show on screen at a given time. To solve this problem, scenes were created. Each scene holds all the relevant attributes, functionality to update each frame and functionality to draw each frame. This allowed for each scene to be separate and completely self contained. Interfaces were also used to ensure that every scene had the required functionality implemented.

Justification for the Concrete Architecture

The team decided on using libGDX as it gave easy and intuitive access to drawing functionality. It also allowed for a quick and easy setup of things such as window dimensions and user input. Building on top of this intuitive platform using our composite OOP approach allows new developers/teams to quite easily expand functionality of features and add new ones without facing comprehension limitations due to the simplistic design decisions taken. Due to libGDX being a popular game development library for Java [4] the performance and support makes it a sufficient choice for future development.

The architecture broke down each component into small classes so that different members of the group could work sequentially, with all their changes on only one file. This made version management much easier because fewer conflicts between commits occured.

Justification Relating to Requirements

Below is a list showing how each requirement was satisfied in the concrete architecture:

UR_PLAYABILITY - A third party library called libGDX was used, allowing for mouse and keyboard inputs to be handled. Additionally, every scene's update function is required to handle mouse and keyboard inputs.

UR_BOAT_SPECS - The abstract "Boat" class contains attributes and a constructor allowing for different specifications of boat when a new Boat is instantiated. The "PlayerBoat" class additionally has a method to change the current specification, allowing for changes between legs if that becomes a requirement in the future.

UR_TIRED_OVER_TIME - One of the "Boat" class attributes is a stamina value and a stamina used per frame value. These function as a measure of tiredness for the player.

UR_OBSTACLES - An "Obstacle" abstract class was made to house all common functionality for obstacles. This also allowed for polymorphism to be used when storing different types of obstacles, increasing code readability.

UR_COLLISIONS - A "CollisionObject" interface was created that the player and all other objects implement. This ensures that everything that can be collided with, has the appropriate functionality. Boats also contain a durability attribute, which is decreased every time the boat collides with an object, fulfilling the second half of the requirement.

UR_BOATS_NO - A "BoatRace" class was created to house all functionality associated with a leg of the game. This spaces out boats so that each boat has an appropriate amount of space in their lane and isn't cluttered.

UR_INFO_DISPLAY - The "PlayerBoat" class' draw method contains functionality to draw information such as how tired the player is and how damaged the boat is. This ensures that whenever the player is playing the game (i.e. when a PlayerBoat is on screen) the information display is shown.

UR_FINALS_PLACING - The scene responsible for the main game contains a check to see if all 3 qualifying legs have been run, then it selects the fastest boats to race in a final. If the player is not one of the fastest, the final results scene is switched to.

UR_PERFORMANCE - LibGDX was used for drawing to the screen and handling user inputs. This is a fast and efficient library that enables us to run the game at the required level of performance.

UR_POWER_UP - Power-ups are implemented using the "Powerup" class. Rather than superclassing this several times, power-ups have a type and the effect is handled by whatever object is receiving it.

UR_LEVELS - Difficulty is handled by the "Difficulty" class, which is implemented as a singleton. This tracks the global difficulty level and stores parameter modifications which can be applied to the boats and obstacles to change the difficulty.

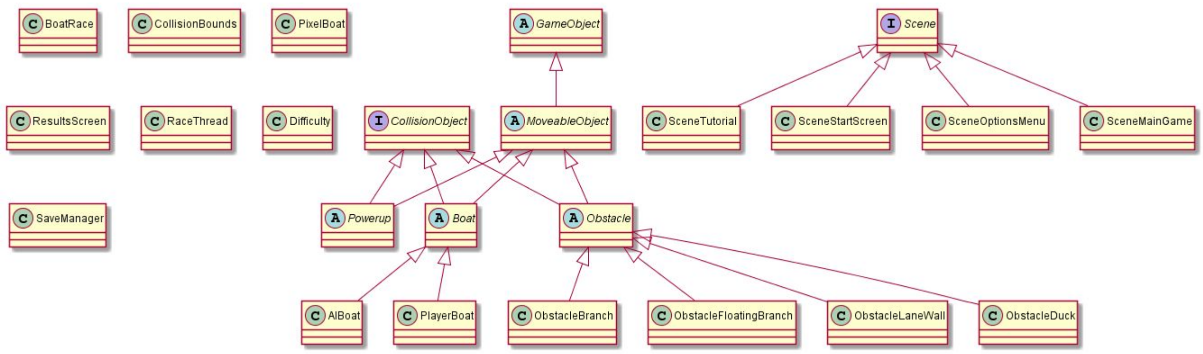
UR_SAVE - Saving is managed by the "SaveManager" class, which can store and load important information about the race after being called, allowing the user to save and load the state of the race.

UR_AWARDS - Upon completion of the third leg of the race the player will be presented with an end screen, taking the data from the Main Game scene in order to update the Race End scene.

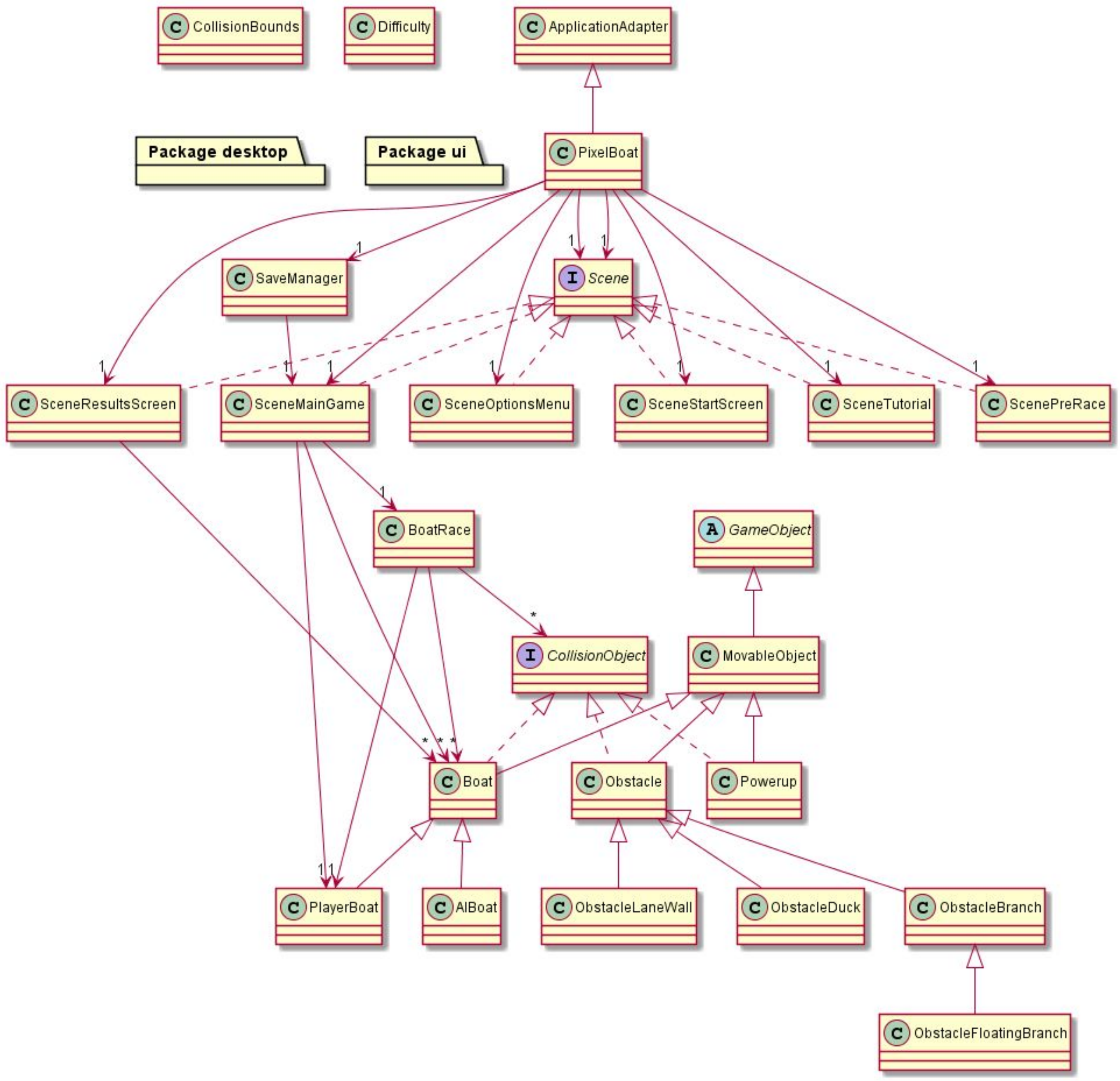
Appendix

Due to the size of these diagrams they have been placed here and are referred to above using specific screenshots.

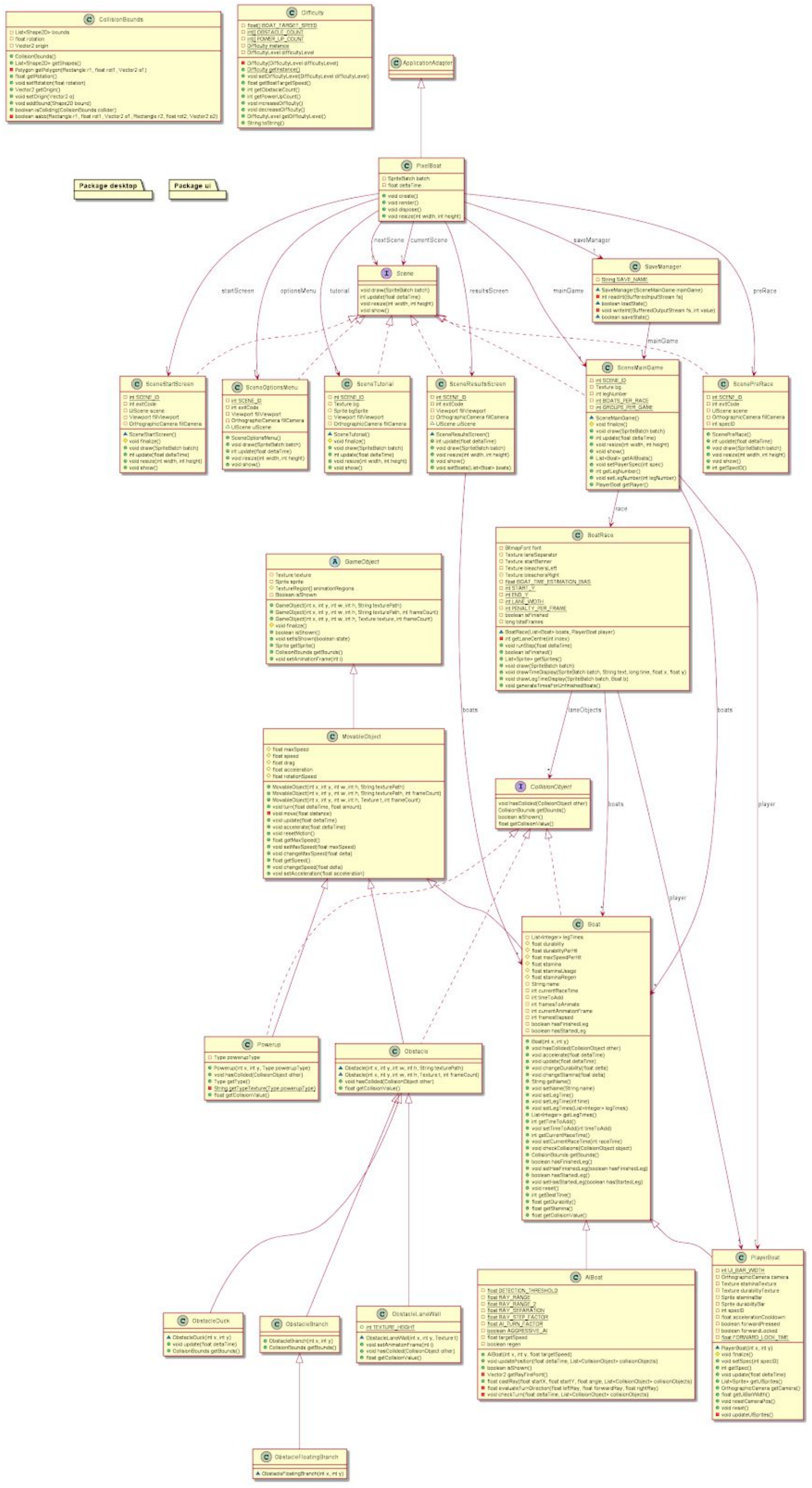
[1] Simplified Component Diagram for abstract representation



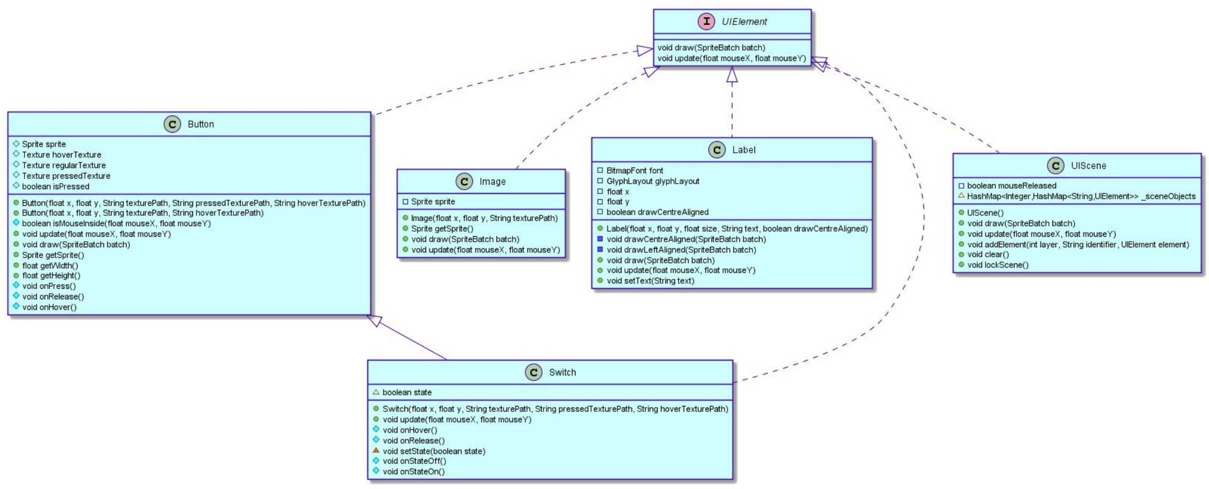
[2] Inheritance Diagram for Concrete Representation



[3] Class Diagram for concrete representation



UI Package


UIElement
void draw(SpriteBatch batch)
void update(float mouseX, float mouseY)
Button
Sprite sprite
Texture hoverTexture
Texture regularTexture
Texture pressedTexture
boolean isPressed
Button(float x, float y, String texturePath, String pressedTexturePath, String hoverTexturePath)
Button(float x, float y, String texturePath, String hoverTexturePath)
boolean isMouseInside(float mouseX, float mouseY)
void update(float mouseX, float mouseY)
void draw(SpriteBatch batch)
Sprite getSprite()
float getWidth()
float getHeight()
void onPress()
void onRelease()
void onHover()
Image
Sprite sprite
Image(float x, float y, String texturePath)
Sprite getSprite()
void draw(SpriteBatch batch)
void update(float mouseX, float mouseY)
Label
BitmapFont font
GlyphLayout glyphLayout
float x
float y
boolean drawCentreAligned
Label(float x, float y, float size, String text, boolean drawCentreAligned)
void drawCentreAligned(SpriteBatch batch)
void drawLeftAligned(SpriteBatch batch)
void draw(SpriteBatch batch)
void update(float mouseX, float mouseY)
void setText(String text)
UIScene
boolean mouseReleased
HashMap<Integer,HashMap<String,UIElement>> _sceneObjects
UIScene()
void draw(SpriteBatch batch)
void update(float mouseX, float mouseY)
void addElement(int layer, String identifier, UIElement element)
void clear()
void lockScene()
Switch
boolean state
Switch(float x, float y, String texturePath, String pressedTexturePath, String hoverTexturePath)
void update(float mouseX, float mouseY)
void onHover()
void onRelease()
void setState(boolean state)
void onStateOff()
void onStateOn()

Bibliography

[3] Introducing Types of UML Diagrams, Lucidchart Content Team
https://www.lucidchart.com/blog/types-of-UML-diagrams

[4] LibGDX https://libgdx.badlogicgames.com/index.html