Architecture:
Team 23

Kyle Mace (kjm560)

Josh Quinn (jtq501)

Louis Hatton (lwh506)

Faris Alblooki (fma527)

Lewis Power (lp1263)

Part A: Abstract and Concrete Representation of Software Architecture

As we decided to take an object oriented approach, we used UML diagrams to graphically represent the architecture, since they lend themselves to, and often complement, an OOP architecture. In addition to this, the use of UML

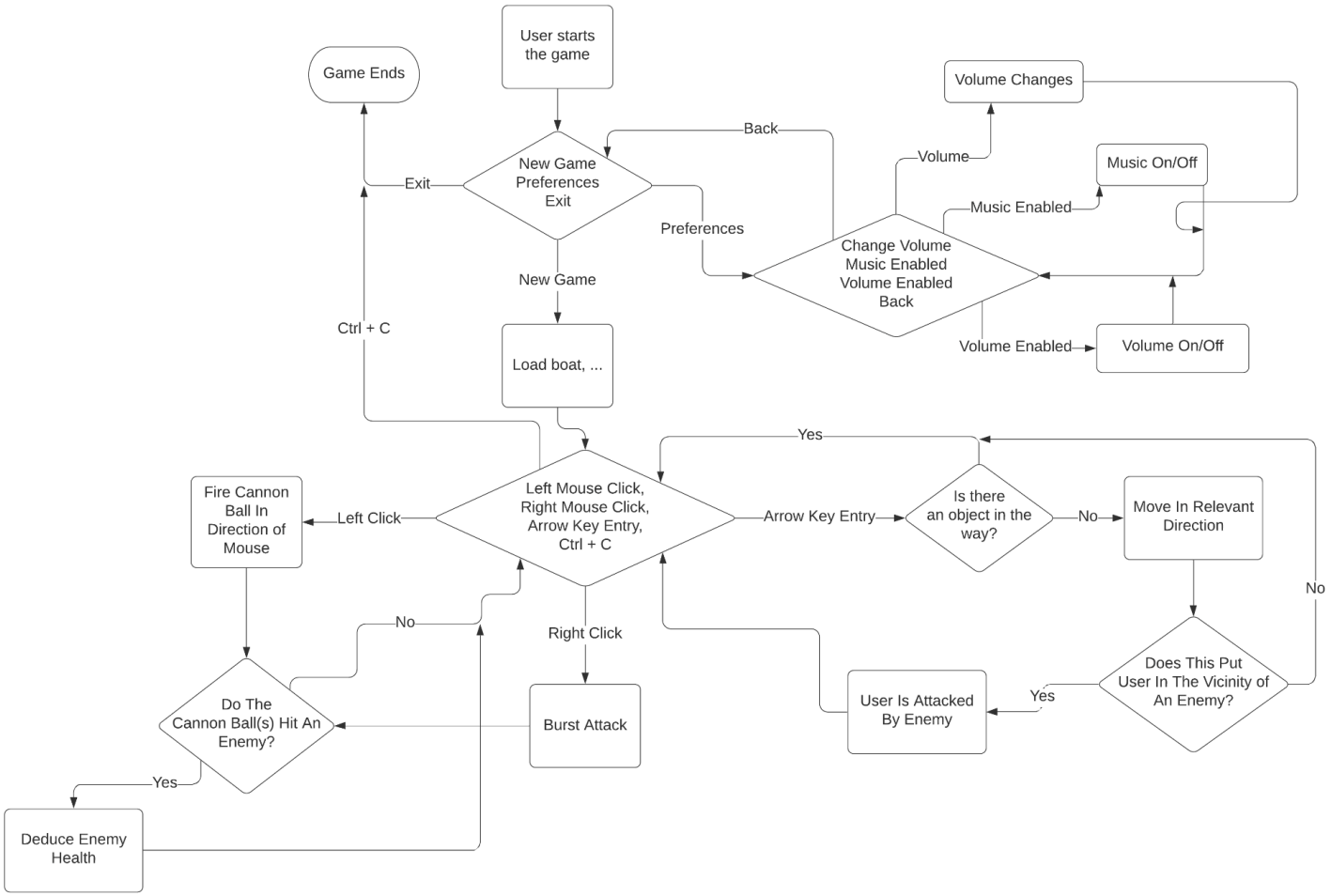
- Makes for a clear indication/reminder to all teammates as to how the system should be implemented;
- Allows future individuals (from assessment 2) to easily come to grips with the architecture and intended behaviour of our code.

Our abstract UML diagram was made through LucidChart, because it is widely used, has good reviews and is free to register. Also considered was MarkupUML, however this was ultimately decided against, because we believed that LucidChart could create diagrams of equal visual appeal, for less overall work than would be necessary with MarkupUML.

Our concrete UML class diagram was made through the feature offered by the 'intelliJ' software development platform. This was chosen in favour of LucidChart because it precluded the possibility of making a mistake with the chart, would save a lot of time, and would produce a diagram just as pleasant as one produced by LucidChart.

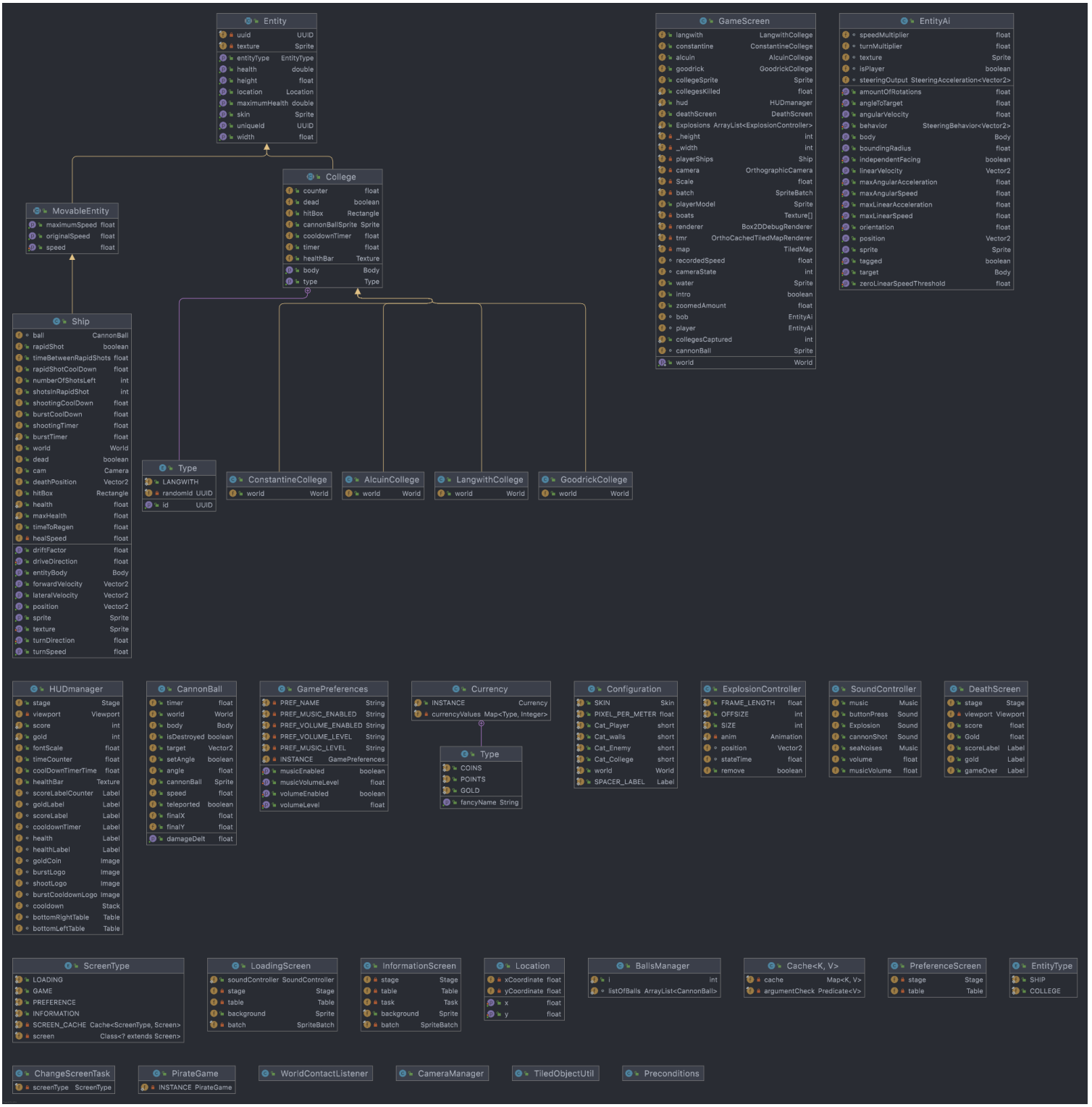
Abstract View:

Created during the initial stages of our project in order to visualise what we wanted to implement as a game.



Concrete View:

From the abstract representation of the architecture, we could derive a general idea of what classes would be needed and how they would interrelate, thus underlying the following concrete representation of the game.


Entity
uuid UUID
texture Sprite
entityType EntityType
health double
height float
location Location
maximumHealth double
skin Sprite
uniqueId UUID
width float
College
counter float
dead boolean
hitBox Rectangle
cannonBallSprite Sprite
cooldownTimer float
timer float
healthBar Texture
body Body
type Type
MovableEntity
maximumSpeed float
originalSpeed float
speed float
Ship
ball CannonBall
rapidShot boolean
timeBetweenRapidShots float
rapidShotCoolDown float
numberOfShotsLeft int
shotsInRapidShot int
shootingCoolDown float
burstCoolDown float
shootingTimer float
burstTimer float
world World
dead boolean
cam Camera
deathPosition Vector2
hitBox Rectangle
health float
maxHealth float
timeToRegen float
healSpeed float
driftFactor float
driveDirection float
entityBody Body
forwardVelocity Vector2
lateralVelocity Vector2
position Vector2
sprite Sprite
texture Sprite
turnDirection float
turnSpeed float
Type
LANGWITH
randomId UUID
id UUID
ConstantineCollege
world World
AlcuinCollege
world World
LangwithCollege
world World
GoodrickCollege
world World
GameScreen
langwith LangwithCollege
constantine ConstantineCollege
alcuin AlcuinCollege
goodrick GoodrickCollege
collegeSprite Sprite
collegesKilled float
hud HUDmanager
deathScreen DeathScreen
Explosions ArrayList<ExplosionController>
_height int
_width int
playerShips Ship
camera OrthographicCamera
Scale float
batch SpriteBatch
playerModel Sprite
boats Texture[]
renderer Box2DDebugRenderer
tmr OrthoCachedTiledMapRenderer
map TiledMap
recordedSpeed float
cameraState int
water Sprite
intro boolean
zoomedAmount float
bob EntityAi
player EntityAi
collegesCaptured int
cannonBall Sprite
world World
EntityAi
speedMultiplier float
turnMultiplier float
texture Sprite
isPlayer boolean
steeringOutput SteeringAcceleration<Vector2>
amountOfRotations float
angleToTarget float
angularVelocity float
behavior SteeringBehavior<Vector2>
body Body
boundingRadius float
independentFacing boolean
linearVelocity Vector2
maxAngularAcceleration float
maxAngularSpeed float
maxLinearAcceleration float
maxLinearSpeed float
orientation float
position Vector2
sprite Sprite
tagged boolean
target Body
zeroLinearSpeedThreshold float
HUDmanager
stage Stage
viewport Viewport
score int
gold int
fontScale float
timeCounter float
coolDownTimerTime float
healthBar Texture
scoreLabelCounter Label
goldLabel Label
scoreLabel Label
cooldownTimer Label
health Label
healthLabel Label
goldCoin Image
burstLogo Image
shootLogo Image
burstCooldownLogo Image
cooldown Stack
bottomRightTable Table
bottomLeftTable Table
CannonBall
timer float
world World
body Body
isDestroyed boolean
target Vector2
setAngle boolean
angle float
cannonBall Sprite
speed float
teleported boolean
finalX float
finalY float
damageDelt float
GamePreferences
PREF_NAME String
PREF_MUSIC_ENABLED String
PREF_VOLUME_ENABLED String
PREF_VOLUME_LEVEL String
PREF_MUSIC_LEVEL String
INSTANCE GamePreferences
musicEnabled boolean
musicVolumeLevel float
volumeEnabled boolean
volumeLevel float
Currency
INSTANCE Currency
currencyValues Map<Type, Integer>
Type
COINS
POINTS
GOLD
fancyName String
Configuration
SKIN Skin
PIXEL_PER_METER float
Cat_Player short
Cat_walls short
Cat_Enemy short
Cat_College short
world World
SPACER_LABEL Label
ExplosionController
FRAME_LENGTH float
OFFSIZE int
SIZE int
anim Animation
position Vector2
stateTime float
remove boolean
SoundController
music Music
buttonPress Sound
Explosion Sound
cannonShot Sound
seaNoises Music
volume float
musicVolume float
DeathScreen
stage Stage
viewport Viewport
score float
Gold float
scoreLabel Label
gold Label
gameOver Label
ScreenType
LOADING
GAME
PREFERENCE
INFORMATION
SCREEN_CACHE Cache<ScreenType, Screen>
screen Class<? extends Screen>
LoadingScreen
soundController SoundController
stage Stage
table Table
background Sprite
batch SpriteBatch
InformationScreen
stage Stage
table Table
task Task
background Sprite
batch SpriteBatch
Location
xCoordinate float
yCoordinate float
x float
y float
BallsManager
i int
listOfBalls ArrayList<CannonBall>
Cache<K, V>
cache Map<K, V>
argumentCheck Predicate<V>
PreferenceScreen
stage Stage
table Table
EntityType
SHIP
COLLEGE
ChangeScreenTask
screenType ScreenType
PirateGame
INSTANCE PirateGame
WorldContactListener
CameraManager
TiledObjectUtil
Preconditions

Part B: Systematic Justification For Abstract and Concrete Architecture

For the architecture, we decided to take an object-oriented approach. This decision was arrived at due to:

- The relatively small size of the project,
- The fact that the inherent nature of the project lends itself to distinct objects with their own behaviours.
- The fact that this approach benefits from the use of inheritance, reducing the amount of code and duplication of classes.

Throughout the development, we built our concrete architecture based on all the requirements which had been established prior to starting. This was achieved by:

- Ensuring all Classes were relevant to at least one requirement. (These are represented in the Justification Table.)
- Removing or adjusting any planned developments which could not be related to any requirements.

This allowed for us to follow the requirements set out and prevented requirements from being forgotten about, reducing the possibility of parts of the game missing in the final release.

In addition, the abstract representation of the object relationships were also referred to during development of our concrete architecture. This included:

- Maintaining an object oriented structure.
- Utilizing inheritance down this structure to prevent the duplication of code

This approach to the development of the concrete architecture keeps the code manageable hence reducing the risk of it being too complex. Furthermore, if the current requirements were to change or new ones were established, this approach would allow for faster development of such new requirements without the need to re-develop core features to allow them to work.

We also ensured that the behavioural aspect (flow) of the game follows the chart established within our abstract architecture. This process included:

- Once new features were introduced, all members of the group would ensure that they adhered to the chart created in the abstract architecture.
- Using the chart during development. Such as, when programming what the right click does, team members were able to refer back to it.

Concrete architecture's relation to the requirements.

Class	Relation to Requirements
Entity	Parent class implemented to describe the location, type of entity, health and entity dimensions (UR_TRANSPORT, UR_MIN_COLLEGES, UR_COMBAT, UR_TERMINATE_DEFEAT, UR_GAMETIME, FR_HEALTH, FR_ENEMY)
EntityAi	The enemy AI inheriting from the AI library (UR_OTHER_SHIPS)
MovableEntity	Parent class implemented to describe the speed attributes of **Ship**, ensuring it does not move arbitrarily fast (UR_TRANSPORT, UR_PLAYABLE, FR_ARROW_KEYS)
College (from which AlcuinCollege, ConstantineCollege, DerwentCollege, LangwithCollege inherits)	Class to demarcate one college from another, inherits from **Entity** (UR_MIN_COLLEGES, UR_COMBAT, FR_HEALTH, FR_INCREASE_GOLD)
Ship	Class to describe the various attributes of the users' ship, such as current speed/direction,

	inherits from **MovableEntity** (UR_TRANSPORT, UR_COMBAT, UR_TERMINATE_DEFEAT, FR_HEALTH, FR_ENEMY_ATTACK, FR_BULLETS, FR_ARROW_KEYS)
GameScreen	Stores an 'image' of everything that is within frame and updates on the next frame in accordance with what is moving/changing (UR_PLAYABLE, FR_GAME_SCREEN, FR_CAMERA, NFR_REALTIME_GAMESCREEN)
LoadingScreen	Class implemented to allow the user to choose from starting the game, exiting the game or selecting their game preferences
InformationScreen	Class implemented to inform the user, at the start of the game, what the controls and objective are (UR_PLAYABLE, UR_DIFFICULTY, NFR_OPERABLE, NFR_USEABLE)
Currency	Stores the amount of points/gold in the game and to whom it belongs (UR_POINT_ACCUMULATION, UR_GOLD_ACCUMULATION, FR_INCREASE_POINTS)
Configuration	Class to hold all commonly used settings that are shared to many classes in one place (UR_PLAYABLE)
Location	Stores the location of entity on the map, ensures entity moves properly across the map (UR_TRANSPORT, UR_PLAYABLE, FR_CAMERA)
GamePreferences	Class implemented to store the personal settings of the user, such as whether they have music muted or not (UR_PLAYABLE)
PreferenceScreen	Class implemented to allow the user to mute or adjust the music/sound effects (UR_PLAYABLE)
BallsManager	Class to keep track of all the balls on screen at any given time (UR_COMBAT, FR_BULLETS)
PirateGame	Instantiates the game (NFR_INTEGRABLE_SYSTEM)
WorldContactListener	Class to find which object collides with which other object, and apply damage to the one that was hit (UR_COMBAT, UR_TERMINATE_DEFEAT)
CameraManager	Keeps the camera focused on the given target, usually the player, ensuring the user can see what they are doing (UR_PLAYABLE, FR_CAMERA)
TiledObjectUtil	A dynamic class that finds the colliders all over the map and applies them to the map so that the boats do not drive over the map and stays within the borders (UR_PLAYABLE)
Preconditions	Holds the conditions for the settings, like change volume and disable music (UR_PLAYABLE)
CannonBall	A class to instantiate the cannon ball when fired (UR_COMBAT, FR_BULLETS)
EntityType	Class to differentiate between the different types of entities (UR_COMBAT, UR_OTHER_SHIPS, FR_ENEMY)
ChangeScreenTask	A class which inherits from **ScreenType** and facilitates the changing from one task to another (NFR_TIMING_OF_MENU_CHANGE)
Cache	Stores useful, often accessed, information (FR_CAMERA, NFR_OPERABLE)
ExplosionController	A class to hold the information relating to an explosion (UR_COMBAT, UR_PLAYABLE,)
SoundController	A class which handles music and sound effects depending on users preferences (UR_PLAYABLE)
DeathScreen	Class implemented to show to the user once the Game is over (UR_PLAYABLE, UR_TERMINATE_DEFEAT, UR_TERMINATE_COMPLETION)