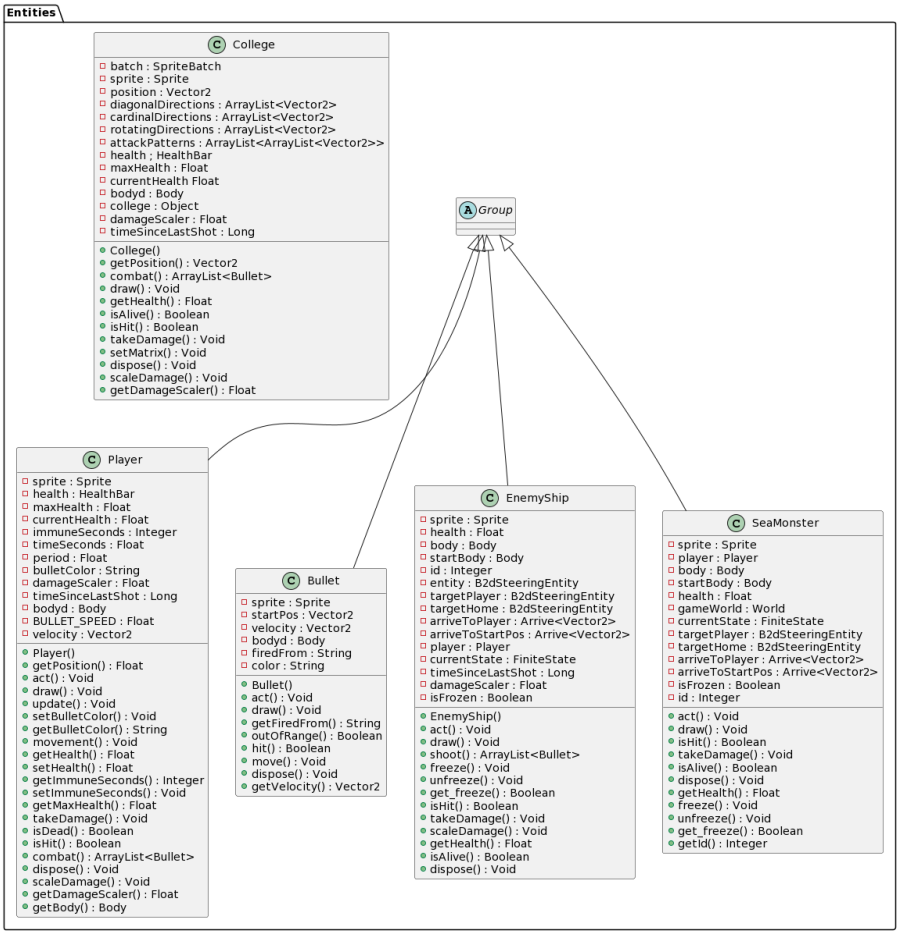
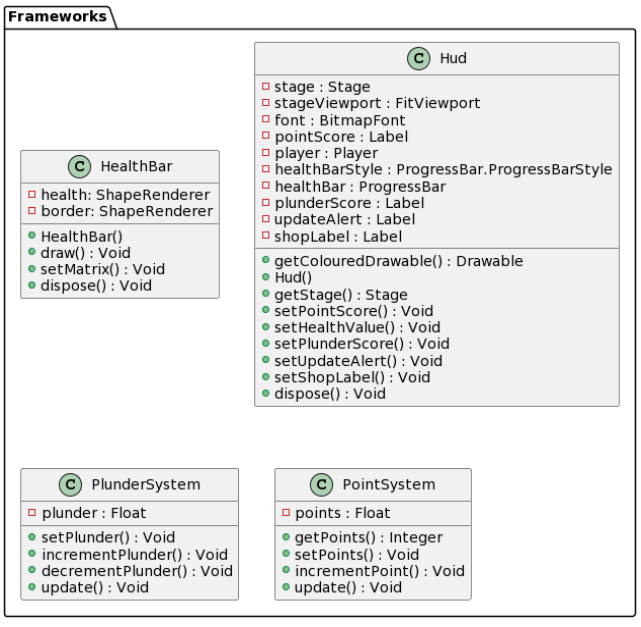
Architecture

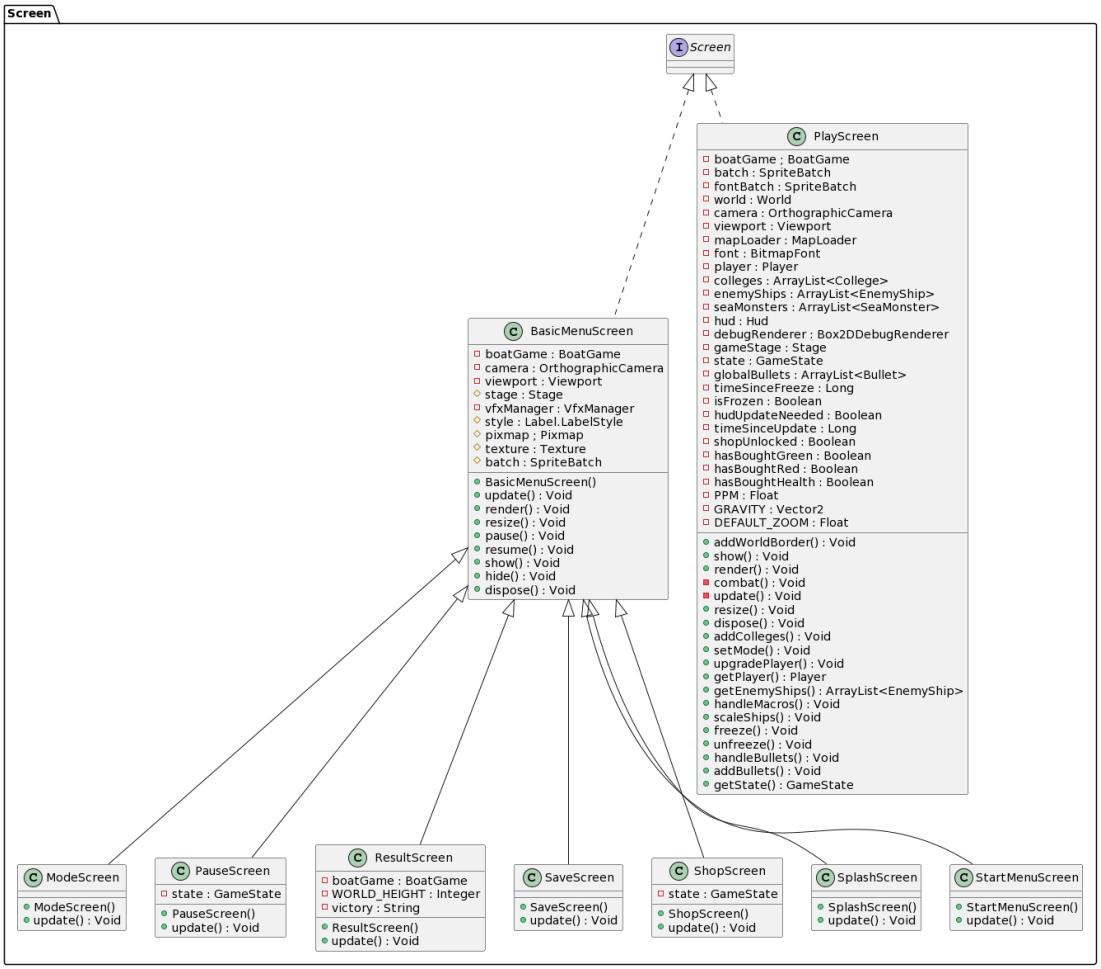
UML Packages

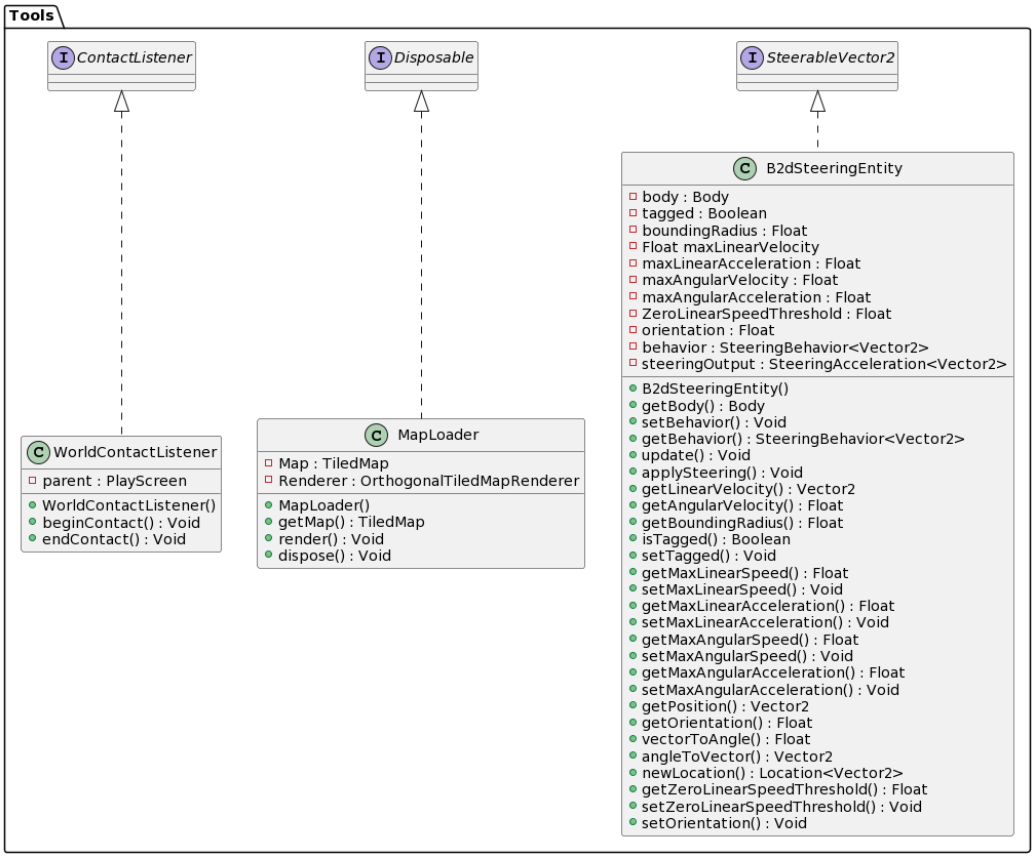
The architecture is split into four packages all contained within the core package, these being the: entities, frameworks, screens and tools.

CORE






Screen
Screen
PlayScreen
- boatGame : BoatGame
- batch : SpriteBatch
- fontBatch : SpriteBatch
- world : World
- camera : OrthographicCamera
- viewport : Viewport
- mapLoader : MapLoader
- font : BitmapFont
- player : Player
- colleges : ArrayList<College>
- enemyShips : ArrayList<EnemyShip>
- seaMonsters : ArrayList<SeaMonster>
- hud : Hud
- debugRenderer : Box2DDebugRenderer
- gameStage : Stage
- state : GameState
- globalBullets : ArrayList<Bullet>
- timeSinceFreeze : Long
- isFrozen : Boolean
- hudUpdateNeeded : Boolean
- timeSinceUpdate : Long
- shopUnlocked : Boolean
- hasBoughtGreen : Boolean
- hasBoughtRed : Boolean
- hasBoughtHealth : Boolean
- PPM : Float
- GRAVITY : Vector2
- DEFAULT_ZOOM : Float
+ addWorldBorder() : Void
+ show() : Void
+ render() : Void
- combat() : Void
- update() : Void
+ resize() : Void
+ dispose() : Void
+ addColleges() : Void
+ setMode() : Void
+ upgradePlayer() : Void
+ getPlayer() : Player
+ getEnemyShips() : ArrayList<EnemyShip>
+ handleMacros() : Void
+ scaleShips() : Void
+ freeze() : Void
+ unfreeze() : Void
+ handleBullets() : Void
+ addBullets() : Void
+ getState() : GameState
BasicMenuScreen
- boatGame : BoatGame
- camera : OrthographicCamera
- viewport : Viewport
◇ stage : Stage
- vfxManager : VfxManager
◇ style : Label.LabelStyle
◇ pixmap ; Pixmap
◇ texture : Texture
◇ batch : SpriteBatch
+ BasicMenuScreen()
+ update() : Void
+ render() : Void
+ resize() : Void
+ pause() : Void
+ resume() : Void
+ show() : Void
+ hide() : Void
+ dispose() : Void
ModeScreen
+ ModeScreen()
+ update() : Void
PauseScreen
- state : GameState
+ PauseScreen()
+ update() : Void
ResultScreen
- boatGame : BoatGame
- WORLD_HEIGHT : Integer
- victory : String
+ ResultScreen()
+ update() : Void
SaveScreen
+ SaveScreen()
+ update() : Void
ShopScreen
- state : GameState
+ ShopScreen()
+ update() : Void
SplashScreen
+ SplashScreen()
+ update() : Void
StartMenuScreen
+ StartMenuScreen()
+ update() : Void


Tools
ContactListener
Disposable
SteerableVector2
B2dSteeringEntity
- body : Body
- tagged : Boolean
- boundingRadius : Float
- Float maxLinearVelocity
- maxLinearAcceleration : Float
- maxAngularVelocity : Float
- maxAngularAcceleration : Float
- ZeroLinearSpeedThreshold : Float
- orientation : Float
- behavior : SteeringBehavior<Vector2>
- steeringOutput : SteeringAcceleration<Vector2>
+ B2dSteeringEntity()
+ getBody() : Body
+ setBehavior() : Void
+ getBehavior() : SteeringBehavior<Vector2>
+ update() : Void
+ applySteering() : Void
+ getLinearVelocity() : Vector2
+ getAngularVelocity() : Float
+ getBoundingRadius() : Float
+ isTagged() : Boolean
+ setTagged() : Void
+ getMaxLinearSpeed() : Float
+ setMaxLinearSpeed() : Void
+ getMaxLinearAcceleration() : Float
+ setMaxLinearAcceleration() : Void
+ getMaxAngularSpeed() : Float
+ setMaxAngularSpeed() : Void
+ getMaxAngularAcceleration() : Float
+ setMaxAngularAcceleration() : Void
+ getPosition() : Vector2
+ getOrientation() : Float
+ vectorToAngle() : Float
+ angleToVector() : Vector2
+ newLocation() : Location<Vector2>
+ getZeroLinearSpeedThreshold() : Float
+ setZeroLinearSpeedThreshold() : Void
+ setOrientation() : Void
WorldContactListener
- parent : PlayScreen
+ WorldContactListener()
+ beginContact() : Void
+ endContact() : Void
MapLoader
- Map : TiledMap
- Renderer : OrthogonalTiledMapRenderer
+ MapLoader()
+ getMap() : TiledMap
+ render() : Void
+ dispose() : Void

Abstract Architecture Justification

Our game has an open architecture in that classes depend on others more than one package away from themselves. For example, PlayScreen in the screens package makes multiple calls to classes in the entities package, and entities make calls to items in the assets package. We chose the object oriented software pattern as this was what we were most familiar with. We did not use the conventional MVC architecture, as several classes implement both model and view features. The main cause of this was our use of Scene2D and specifically Actor, which encourages this style. However, we felt that this was a net win due to the convenience of the Actor's draw and act methods.

Concrete Architecture Justification

Entities Package

The entities package contains the college, player, enemy ship, sea monsters and bullet class. The Player class is responsible for constructing the boat that the user will control and contains the methods which allow them to move around, interact with other sprites and all in all, play the game. The College class is responsible for constructing the colleges of the University of York that the player must conquer. These two classes also contain attributes and classes relating to their health in the game which indicate their state in the game (dead or alive, weak or strong). The bullet class is responsible for constructing the bullets that the user will use in combat with other colleges. The enemy ship class is a ship spawned by the colleges that follows and shoots at the player when in range and otherwise returns to a rest position. The monster class is spawned at points on the map. It has the same movement pattern as the ships except that it does damage to the player by crashing into it. Both are implemented using LibGdx's steerable interface.

This package is important in fulfilling the following user requirements of our game: USR7 USR9 USR19. These requirements are mainly related to the combat between the player and the colleges which happens through each side using bullets to deplete each other's health bar till they are empty and completing the goal of the game. It also fulfils USR2 and USR17 which are about the game being single player and colleges being controlled by an automated AI which are also carried out by this package as the user only controls the entity constructed by the Player class. The EnemyShip fulfils requirement USR23 and the SeaMonster class USR24 - the sea monsters were our idea of an 'obstacle'.

The functional requirements: FR02 FR04 FR05 FR06 FR07, are completed from this package too. These requirements are about the user controlling their sprite to engage in combat with the other colleges and in turn the colleges being able to fight back at the player. FR03 varies slightly from the rest but I still feel it is related as it is the requirement of there being a victory and failure page but that wouldn't occur unless the combat between the player and colleges determined a winner and loser. Requirement NF03, which is to do with no gorey images, is also fulfilled here as the bullets and combat between player and college isn't horribly violent.

Frameworks Package

The Frameworks package contains the PointSystem class which, as the name suggests, tracks the user's points as they play the game. It also contains the HealthBar class which is the main framework that tracks the health of the colleges and the player and helps indicate their state in the game (dead or alive, weak or strong). It also contains the PlunderSystem class which tracks the plunder.

This package helps fulfil user requirements 4 and 7 which is mainly related to how the game finishes and also to do with the health of the player and colleges. USR4 is about the game lasting from 5-10

mins which gets fulfilled by this package mainly from the point system as it would finish the game after a threshold has been reached and USR7 is about colleges being captured or destroyed which is carried out by the HealthBar class. The PlunderSystem also helps fulfil USR25.

The Functional requirements carried out by this package are: FR03 FR06 FR10 FR12. These are mainly talking about gaining points and winning/finishing the game apart from FR06 which is the requirement of the user taking damage if they are attacked successfully by an opposing college.

Screens Package

The Screens package contains nine classes. The PlayScreen class constructs the screen that the user will see in order to play the game. This includes methods for rendering items, resizing them, updating them etc, and also saving and loading. The results screen is displayed on death or victory. The ShopScreen displays a list of macros that can be bought. The ModeScreen allows the user to choose three difficulty levels to play on. BasicMenuScreen is used as a base class for others to inherit from. SplashScreen functions as *insert.* StartMenuScreen is the menu screen that allows the user to play or browse saved games. SaveScreen displays 4 memory slots the user can load. PauseScreen implements a pause facility and allows the user to exit the application or return the game.

This package fulfils the user requirements: USR11 USR12 USR14 USR16. These requirements are the ones that state how the screen that the user plays on looks like such as USR11 which talks about how the game should have varying screen sizes or USR16 which requires the game to be clean and non violent. ModeScreen and ShopScreen help fulfil requirements USR21 and USR25 respectively. PlayScreen also helps fulfil requirements USR20, USR21, USR22 and USR25.

The functional requirements that are satisfied by this package are: FR01 FR03 FR09, which are to do with fitting the game into any sized screen and the different screens displayed in the game. These being a start screen, an end screen, a pause screen etc. The WorldContactListener class listens to any contacts during the game and performs the necessary actions when they collide. It has been added to improve the collision system for enemy ships, sea monsters and others. - USR10.

Tools Package

We then lastly have the tools package which contains the MapLoader which is the class that renders the map that the user will play on. This package completes the user requirement for making it a 2D game MapLoader renders a 2D map and functional requirement of it being able to fit on any sized screen as the map can be resized. The B2dSteeringEntity class works as a parent class for our AI enemy entities and consists of the methods that implement movement and attacking.

GameState

GameState is a class created in order to save and load games. This helps fulfil the requirement USR20. It uses the google Gson library to turn an object into a json file which is then saved in LibGdx's preferences. We found that gson had difficulty parsing constructors and functions in general, so we omitted both. As a result GameState resembles a struct, i.e. it consists mainly of public variables.