

# Architecture:

## Team 23

Tom Rosewell

Sam Redgewell

Tom Keeler

Jake Stogden

John Limb

Adham Nouredin

## 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

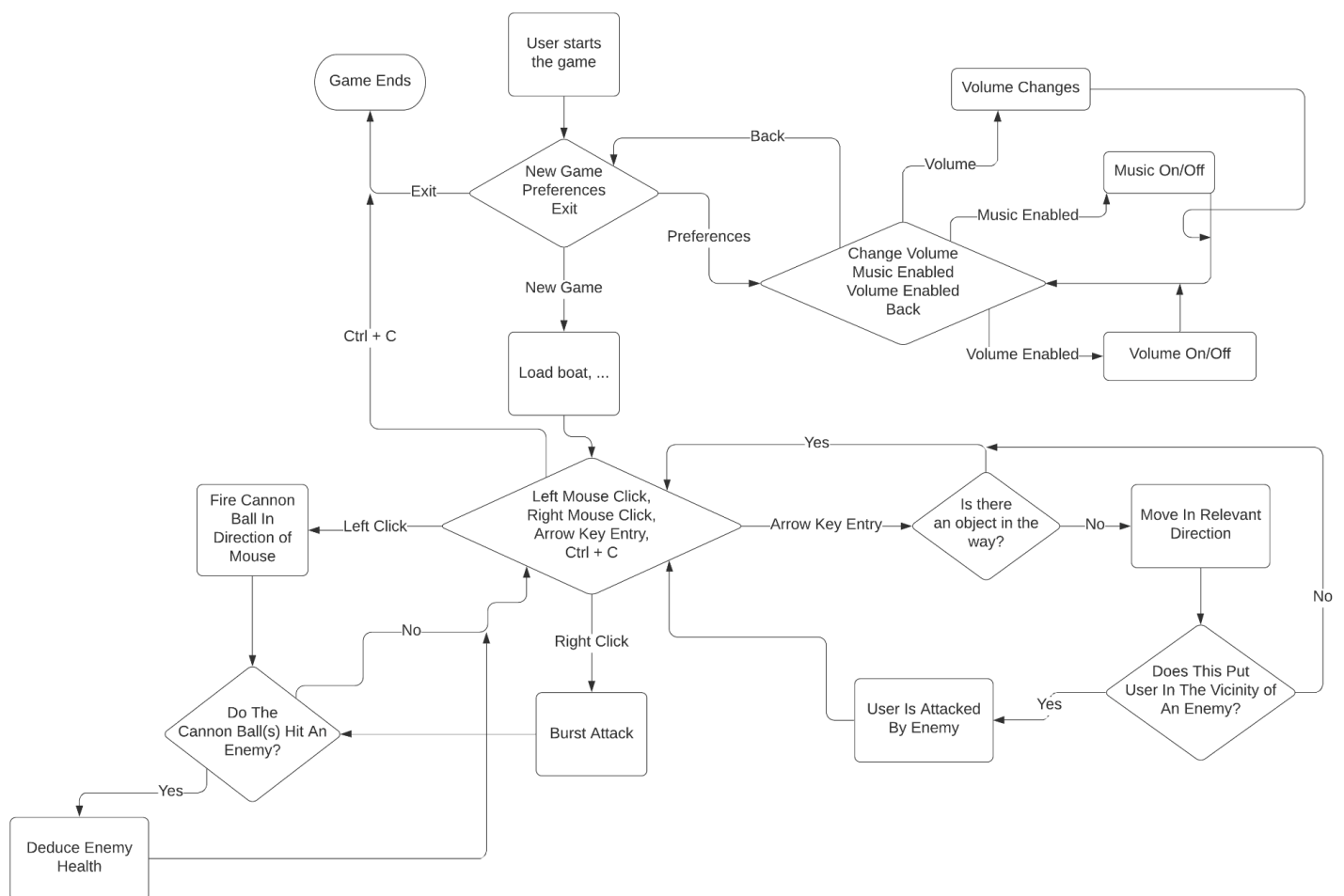
- Makes for a clear indication/reminder to all teammates as to how the system should be implemented;
- Allows future individuals (from assessment 2) to easily come to grips with the architecture and intended behaviour of our code.

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.

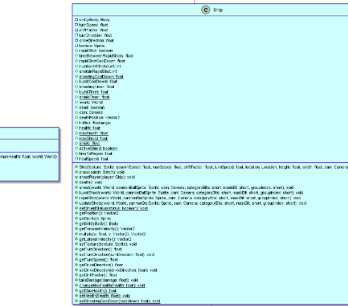
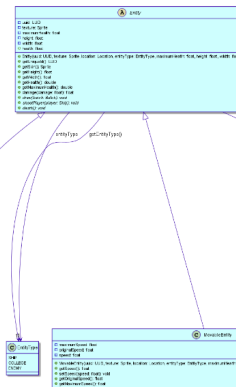
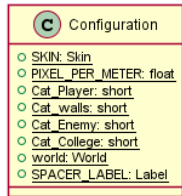
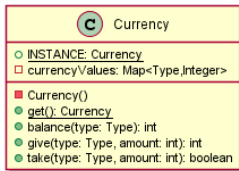
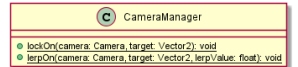
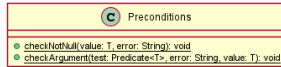
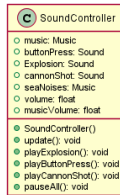
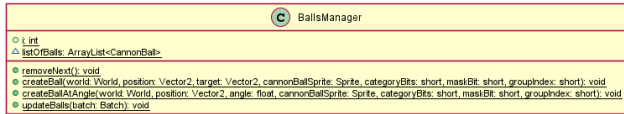
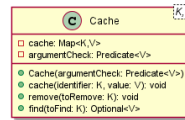
### Abstract View:

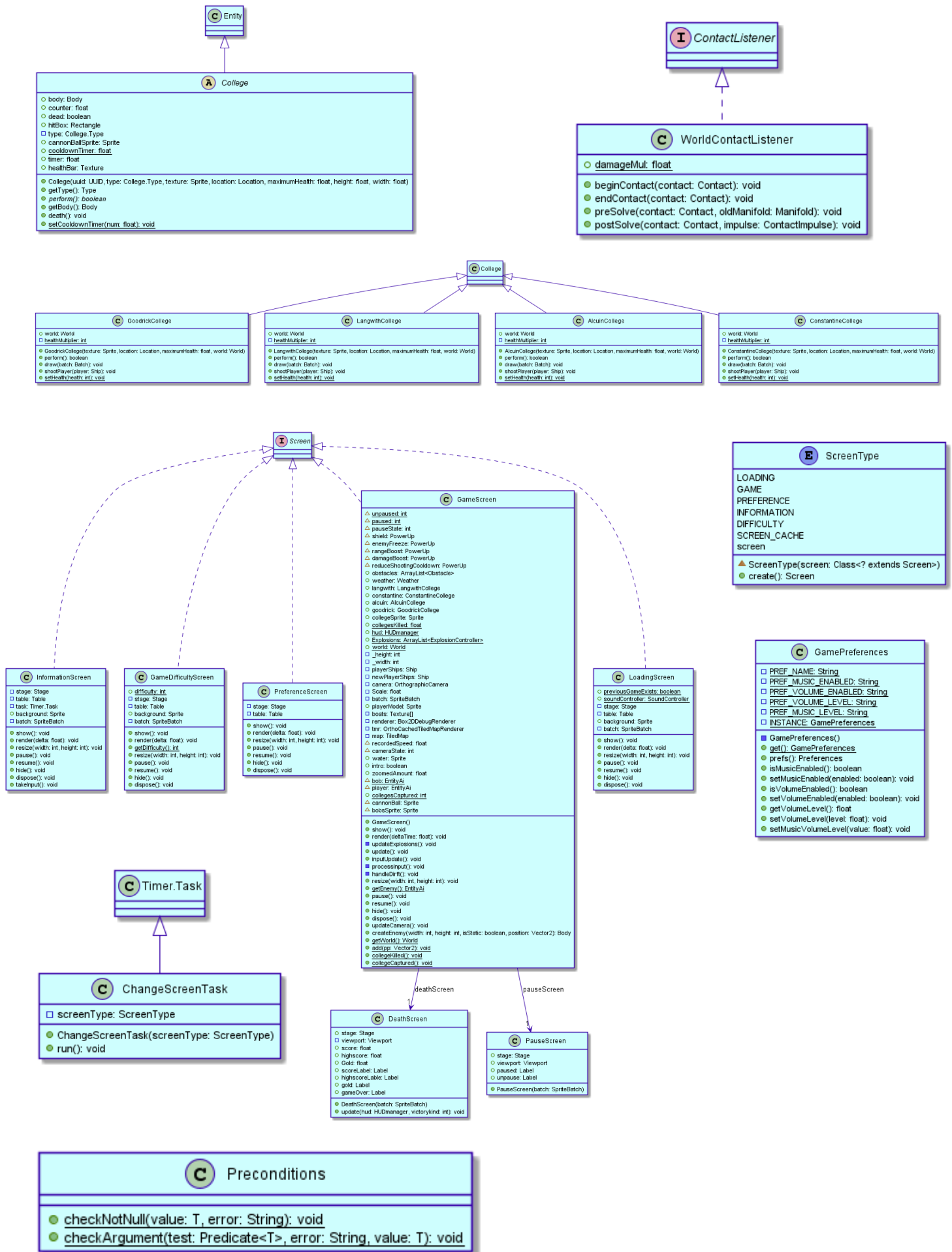
Created during the initial stages of our project in order to visualise what we wanted to implement as a game.



### 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.





## 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.
- Utilising 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
<b>Entity</b>	Parent class implemented to describe the location, type of entity, health and entity dimensions
<b>EntityAi</b>	The enemy AI inheriting from the AI library
<b>MovableEntity</b>	Parent class implemented to describe the speed attributes of <b>Ship</b> , ensuring it does not move arbitrarily fast
<b>College</b>	Class to demarcate one college from another, inherits from <b>Entity</b>
<b>Ship</b>	Class to describe the various attributes of the users' ship, such as current speed/direction, inherits from <b>MovableEntity</b>
<b>GameScreen</b>	Stores an 'image' of everything that is within frame and updates on the next frame in accordance with what is moving/changing
<b>LoadingScreen</b>	Class implemented to allow the user to choose from starting the game, exiting the game or selecting their game preferences

<b>InformationScreen</b>	Class implemented to inform the user, at the start of the game, what the controls and objective are
<b>Currency</b>	Stores the amount of points/gold in the game and to whom it belongs
<b>Configuration</b>	Class to hold all commonly used settings that are shared to many classes in one place
<b>Location</b>	Stores the location of entity on the map, ensures entity moves properly across the map
<b>GamePreferences</b>	Class implemented to store the personal settings of the user, such as whether they have music muted or not
<b>PreferenceScreen</b>	Class implemented to allow the user to mute or adjust the music/sound effects
<b>BallsManager</b>	Class to keep track of all the balls on screen at any given time
<b>PirateGame</b>	Instantiates the game
<b>WorldContactListener</b>	Class to find which object collides with which other object, and apply damage to the one that was hit
<b>CameraManager</b>	Keeps the camera focused on the given target, usually the player, ensuring the user can see what they are doing
<b>TiledObjectUtil</b>	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
<b>Preconditions</b>	Holds the conditions for the settings, like change volume and disable music
<b>CannonBall</b>	A class to instantiate the cannon ball when fired
<b>EntityType</b>	Class to differentiate between the different types of entities
<b>Difficulty Screen</b>	Class implemented to allow the player to select a difficulty at the start of the game
<b>ChangeScreenTask</b>	A class which inherits from <b>ScreenType</b> and facilitates the changing from one task to another
<b>Cache</b>	Stores useful, often accessed, information
<b>ExplosionController</b>	A class to hold the information relating to an explosion
<b>SoundController</b>	A class which handles music and sound effects depending on users preferences
<b>HUDmanager</b>	Manages the game's overlays
<b>Obstacle</b>	Creates and defines obstacles in the ocean & detects when a player hits one
<b>PowerUp</b>	Creates the 5 different power ups in the game, and detects when a player collects one
<b>Weather</b>	Defines roaming patches of bad weather and how they affect the player
<b>DeathScreen</b>	Class implemented to show to the user once the Game is over