Architecture

Mario

GROUP 28

Joseph Frankland

Anna Singleton

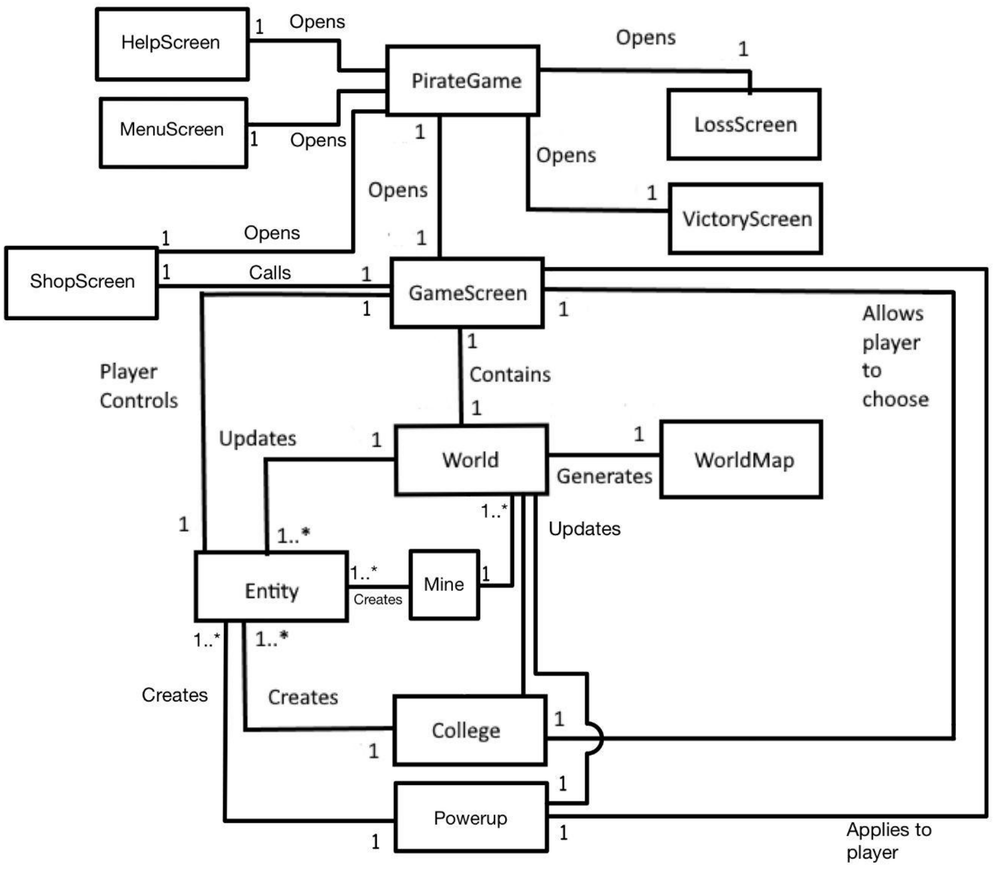
Saj Hoque

Leif Kemp

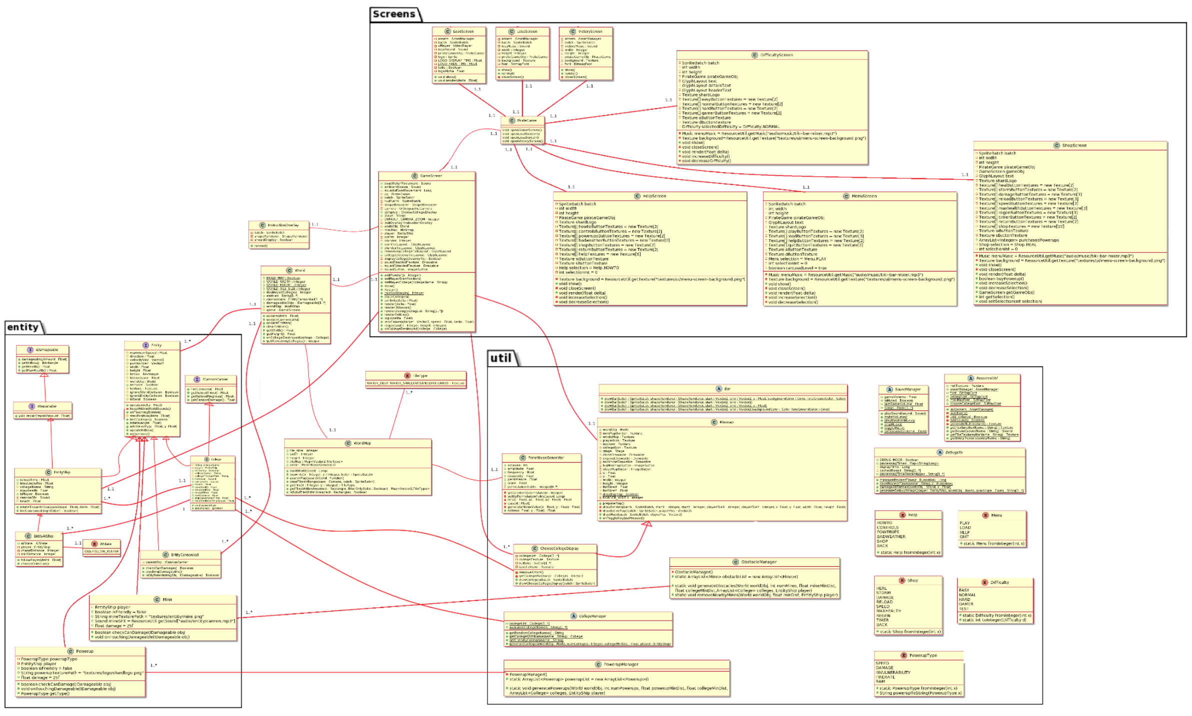
Shi (Lucy) Li

Hugo Kwok

Abstract Diagram



Concrete Diagram



Clearer view of UML diagram available on the website:
https://annabeths.github.io/ENG1-Phase2-Website/UML.html
The Abstract model is intended to give a brief overview of how our architecture should function. This model focuses on the classes rather than the attributes or methods within them. It serves merely as an abstracted model for how classes within the finished game should interact, and doesn't go into any detail for how one would actually implement these interactions. We didn't feel the need to use any specific languages or tools to build the model, and thus the model is simply hand drawn.

We generated the Concrete model using PlantUml, and thus are using naming/typing conventions common to that language. We will briefly describe some of these conventions below:

- Static fields/methods are underlined
- Final methods are capitalised
- C - Standard Class
- I - Interface
- A - Abstract class
- E- Enum
- Public methods/variables are represented by a circle before their signature
- Protected methods/variables are represented by a diamond before their signature
- Private methods/variables are represented by a square before their signature
- If a type is applied to an attribute or method on the diagram but the corresponding class is not listed anywhere on the diagram, then it is either a part of the Java standard library or Libgdx.
- All protected attributes of a class have implied getter() and setter() methods which exist in the class, but aren't drawn on the diagram in the interest of brevity.

More details on PlantUML syntax can be found here

The Concrete model is very close to how the finished code for the project should look. We decided to expose every method and class attribute on the concrete model. While this does make the model very verbose, it does mean that anyone who views it will understand in detail how our project works and be able to recreate it.

Part 3b
Our abstract model gives a brief overview of how classes should interact in our final product. It shows the classes we will need to implement our requirements and the associations between these classes. Associations show which classes interact with each other and how precisely they interact. For example, one association is the ‘Updates’ association between World and Entity. This shows that the World class is responsible for updating the state of all Entities in the game.

The concrete model builds on the abstract model by including additional classes as well as detailing the methods and attributes specific to each class. This brings the model closer to how we will eventually program the game. For example in the abstract model NPC ships as well as the player are abstracted as a single class ‘Entity’, whereas in the concrete model we distinguish between different types of entity, such as ‘EntityShip’ for the player, ‘EntityAIShip’ for NPC ships and ‘EntityCannonball’ for cannonballs fired by ships and colleges. In addition we realised that Colleges shared many common attributes with other Entities, in that they fire cannonballs and have a texture and position, and therefore Colleges should be implemented by inheriting from Entity. In addition it was decided that in order to meet requirement UR_ COLLEGES the game should feature an overview of the map from which players could choose a college to start at. This led to the creation of several classes that were not listed on our original abstract model; namely the Minimap, ChooseCollegeDisplay and CollegeManager classes. Minimap provides methods for drawing an abstract representation of a fully generated WorldMap to the screen, which could then be utilised by ChooseCollegeDisplay. We then realised that this Minimap could be useful for helping the player to navigate the map, which is important since the map is randomly generated and fairly large. Thus we decided to draw a small part of the Minimap to the top left of the screen, as well as implementing a button to open a full version of the Map - the same map that is visible in the ChooseCollegeDisplay. We also realised that the game had a moderate loading time when opening, and thus decided to include a new Screen not listed on the abstract UML diagram - LoadingScreen. This screen provides something for the user to look at while the game loads in assets.

There are also a few Abstract classes in the concrete model that don’t appear in the abstract model. We found that many classes had methods in common and these classes had no relation to one another; for example the PirateGame, EntityShip,College and GameScreen classes all need to play sounds and not all share a relationship with each other. Thus we created the ‘SoundManager’ abstract class to facilitate playing sounds, music etc. Implementing audio this way also has the advantage of us being able to adjust volume or mute audio more easily, as all sound objects are played within the scope of the SoundManager class. This made it easier for us to meet requirements UR_AUDIO and FR_AUDIO_MUTE.

We will now show that each of our requirements are implemented by showing which class(es) and methods on the concrete architecture implement each requirement. Please see the requirements document for a more detailed overview of user/system requirements and how we elicited them. UR_TIME, UR_ACCESSIBILITY are omitted due to being non-functional requirements and thus not directly relevant to the game’s architecture. Furthermore requirement UR_PLATFORM is not listed since it was met by the choice of Language - Java - which is OS independent.

Requirement	**How it was implemented**
UR_MOVEMENT	The boat itself is implemented by EntityShip, which is a class describing an entity with the ability to move, rotate and fire cannon shots. EntityShip is drawn on the screen with a boat texture. The GameScreen contains the boat the player controls under the 'player' attribute, which is an instance of EntityShip. Controlling the boat is implemented by the controls() method of GameScreen, which rotates and moves the 'player' EntityShip when the player presses the W-A-S-D keys, the ship can also be moved by holding down LEFT SHIFT to move them forward in the direction they are facing. On the concrete architecture GameScreen has a 1..1 relationship with EntityShip. This refers to the player object which is an instance of EntityShip in GameScreen.
UR_COMBAT	Ships the player can encounter are implemented by the EntityAIShip class. This is a class that inherits from the EntityShip class and contains additional methods such as 'followPlayer()'; a method that causes the ship to chase the player if they come close enough to the ship. In addition, the College class contains the spawnShip() method which causes an instance of EntityAIShip to appear near the College every few minutes. This is shown on the diagram by the 1..* relationship between College and EntityAIShip.
UR_ COLLEGES	College selection is implemented by the 'ChooseCollegeDisplay' class. This is a class which inherits from Minimap - itself a class that contains an abstract representation of a WorldMap. The class draws each college on the map and allows the player to choose a college by clicking on it. The CollegeManager class generates five colleges to the map which the player can choose from.
UR_POINTS	Points are implemented as an attribute of GameScreen. The show() method of GameScreen starts a timer which increases points by 1 every second.
UR_OBJECTIVE	The player's objective is to destroy every opposing college. This is implemented by the onCollegeDestroyed() method of GameScreen; this method checks the number of colleges left in the world, and if all opposing colleges have been destroyed, calls the openVictoryScreen() method of PirateGame, which instantiates and calls the 'show()' method of VictoryScreen(). The number of remaining colleges is printed to the screen by GameScreen.
UR_AUDIO	Music and sound effects are managed by the abstract class SoundManager. Various methods in various classes call methods of SoundManager to play sounds.
UR_ COLLEGES	A list of colleges on the map is maintained by CollegeManager. The render method of GameScreen retrieves the collegeList attribute of CollegeManager and prints the number remaining to the screen. Thus the player is able to track their objective.
UR_SAVE_GAME	Implemented using libgdx's preferences system, which uses a simple key-value pair system in order to retrieve simple values from a file on disk.
UR_DIFFICULTY	The player can choose one of the four difficulties in the menu before they start the game, each difficulty changes the player's hp and the amount of mines on the map (higher difficulty = less hp, more mines, etc)
UR_OBSTACLES	The generator of obstacles (mines) is built on the CollegeManager class, they use similar methods to generate mines just like colleges. The mines are designed to be stationary and will deal damage to the player's ship when hit. To accomplish this we created a new Mine class.
UR_WEATHER	Every 15s, the system does a random check when the storm is not active, to enable a storm, and if it does not start the storm then the chance increases slightly up to a cap, until the storm starts. When the storm starts, it is active for 90s or until the player returns to their home college
UR_POWER_UP	The generator of power ups is built on the CollegeManager class. We then created a powerup class which handles collisions and tells the game to apply the powerup to the player when hit. We then added methods for the GameScreen to communicate from the powerup to the player and apply the appropriate effects for the appropriate length of time, and then remove them.