Architecture

Team 26 - Pirate Ducks

Team members:

Charlie Crosley
Alice Cui
James McNair
Robert Murphy
Marc Perales Salomo
Dan Wade

Architecture

Abstract Architecture

We will be using PlantUML to create all our architecture diagrams. Our Game uses a Layering and partitioning Architecture where each layer can only communicate with the layer directly above it. Throughout the project we intend to stick to design patterns so our design is consistent throughout the project.

We plan to use the singleton design pattern for the following reasons:

- It will ensure information stored within the class, for example the player's score remains constant throughout the game.
- Limiting the creation of a single main class will reduce the memory strain on the game as a lot of variables will have to be reinitialized if a new main class is created.

Throughout the project we will also use dependency injection to pass information around the game, this is useful for the listed reasons:

- It will limit the access of each individual class to only what it needs. This limits the chances of accidentally breaking another part of the program by calling a method incorrectly
- This will also reduce the coupling of the code as each class will have a limited scope
- This will also make testing easier as the entire project will not have to be initialised, only the specific variables that the target class requires

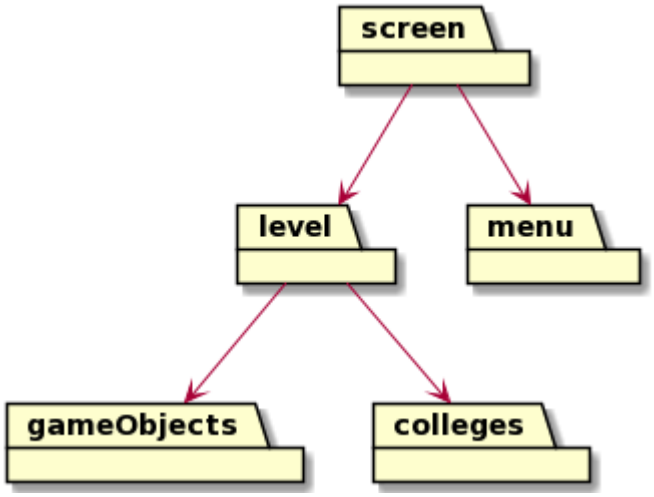
We are going to use an abstract factory within the game. This has many advantages including:

- Avoiding repeated code
- This also makes testing easier as we only have to test any code within

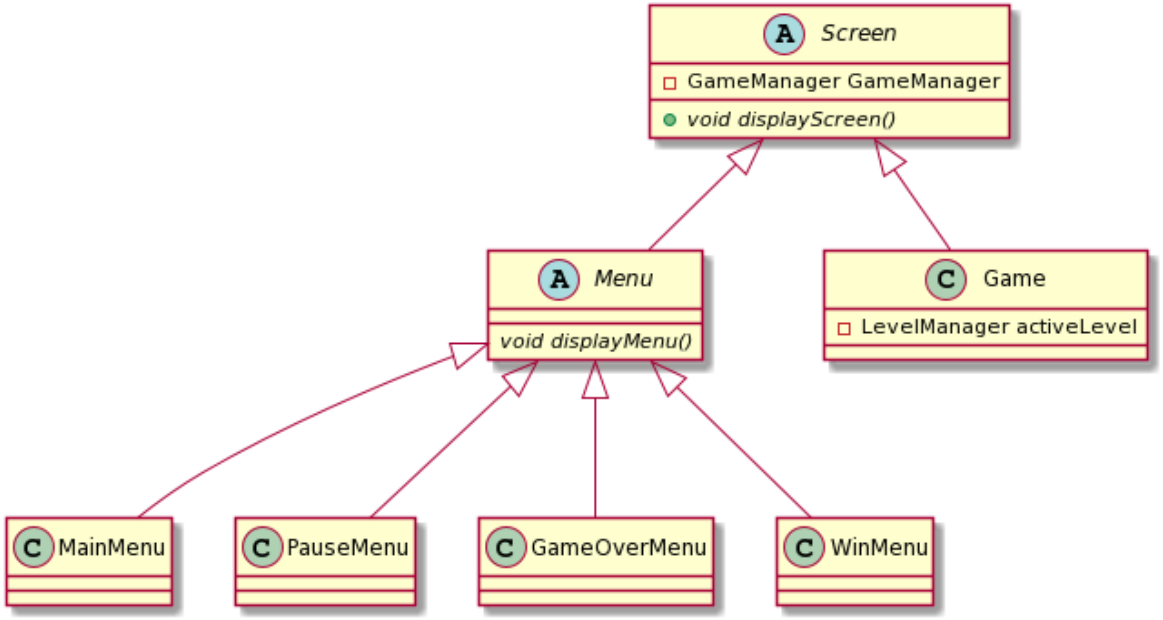
We are also going to use lazy initialization to only load the portions of the game that are currently in use to reduce both memory usage and the initial loading time.

Concrete architecture

Taking the design patterns mentioned above into account we designed the following package diagram to outline how the game will be structured.



In order to organise the level structure, we will be using abstraction to store useful information about each component at a general level so the code is not repeated, for example information about location. Doing this will help avoid repeated code and speed up development as the inherited methods will guide what needs to be done for that class.

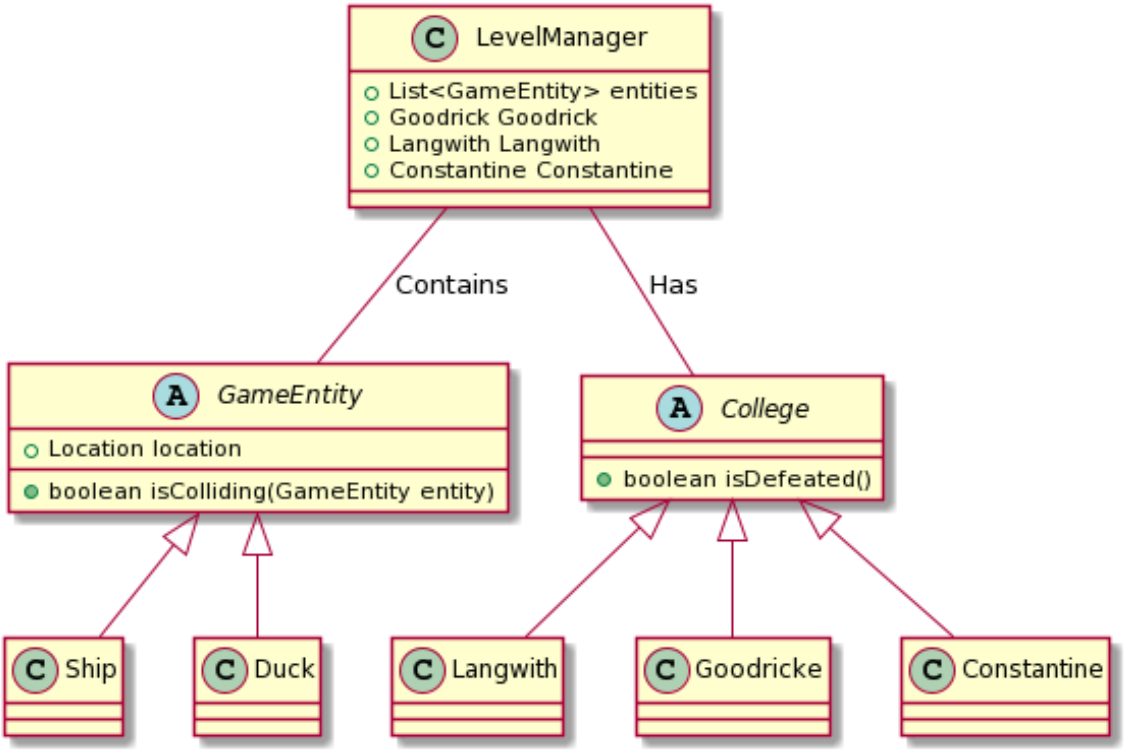


To reduce the complexity of the game processing, we are going to create an abstract representation of the screen, so new screens can be added without any additional display code. This enabled us to implement five additional screens on top of the main game screen.

In our concrete architecture all objects in the game extend from the abstract class GameObjects. This class contains position information of game objects and has the required methods for rendering objects, setting their dimensions and disposing them.

In order to manage the display of the actual game environment, we created an abstract class called LevelManager that is in charge of displaying the actual game. It contains the

necessary methods to load the game, set up the player and its initial health and load the map. It also has two sub classes which are the college class and the MainLevel class. The college class is also an abstract class, it is used to store all the methods required for each college. It is the parent class of all the individual college classes (Goodricke, Langwith and Constantine). The MainLevel class contains the code to implement background music and sound effects, as well as the necessary features to enable college combats.



We also created public interfaces such as Screen and Health to ease our implementation of the chosen functionalities since we can call the methods from these interfaces in other classes. The Screen interface contains methods to draw screens, update screens and dispose of them once they're no longer active. Its methods are used in five other screen classes. The Health interface is used for all objects that require a health bar in the game (player, enemy cannons etc.) and consists of methods to return the current health and to update the health.

Justification

During our stakeholder-meeting, one of the requirements for this project was the ability to accommodate changes of requirements. In regard to this, for the main structure of the program we choose to use a layering and partitioning architecture. We selected this architecture because it allows:

- a low amount of coupling between packages meaning less modifications need to be made if the client changes the requirements further along the project.
- The architecture does not include bypasses for layers as though it can reduce both the space and time complexity overhead of carrying out specific tasks it will increase the amount of unnecessary coupling between layers making larger scale changes more complex later into the project.

Another component defined in the requirements is the requirement for other screens that are not just the game. For example a pause menu (requirement FR_PAUSE) and a game over screen (requirement FR_FINISH). Thus we have chosen the implementation of the Screen interface, allowing the addition of screens without any additional display code. This allowed us to create a menu screen satisfying the user requirements:

- UR_START_NEW_GAME - The user is able to start a new game
- UR_QUIT_GAME - The user is able to quit the current game
- UR_RESTART_GAME - The user is able to restart the game
- UR_PAUSE_GAME - The user is able to pause the game
- UR_CLOSE_GAME - The user is able to close the game

For this project we decided to not use a strict MVC (model, view, component) structure because it would increase the complexity of the program. This is because each object that will be rendered on the screen while the user is playing the game would require two components instead of one, where the model component must keep updating the view component. Instead both the rendering and model code will be contained within the same class.

The design pattern of the abstract factory was used to dictate the design of the GameObject and Screen class. These classes are designed in a way that can:

- Guide new creation of objects, as the methods imply what needs to be included so information is not forgotten.
- Define how created objects are structured so that they are consistent, making it easier to understand another team member's code
- Avoid repeated code between objects

The use of abstract factories will be used to aid in satisfying the following functional requirements:

- FR_MAIN_MENU - The system should have a menu with buttons to start a new game
- FR_FINISH - The game should show a game over / winning screen when applicable

The dependency injection software patterns will be followed as needed by defining constructors both in abstract classes and within each individual class itself based on the direct needs of the class.