Architecture

Group Name: Team 6
Group Number: 6

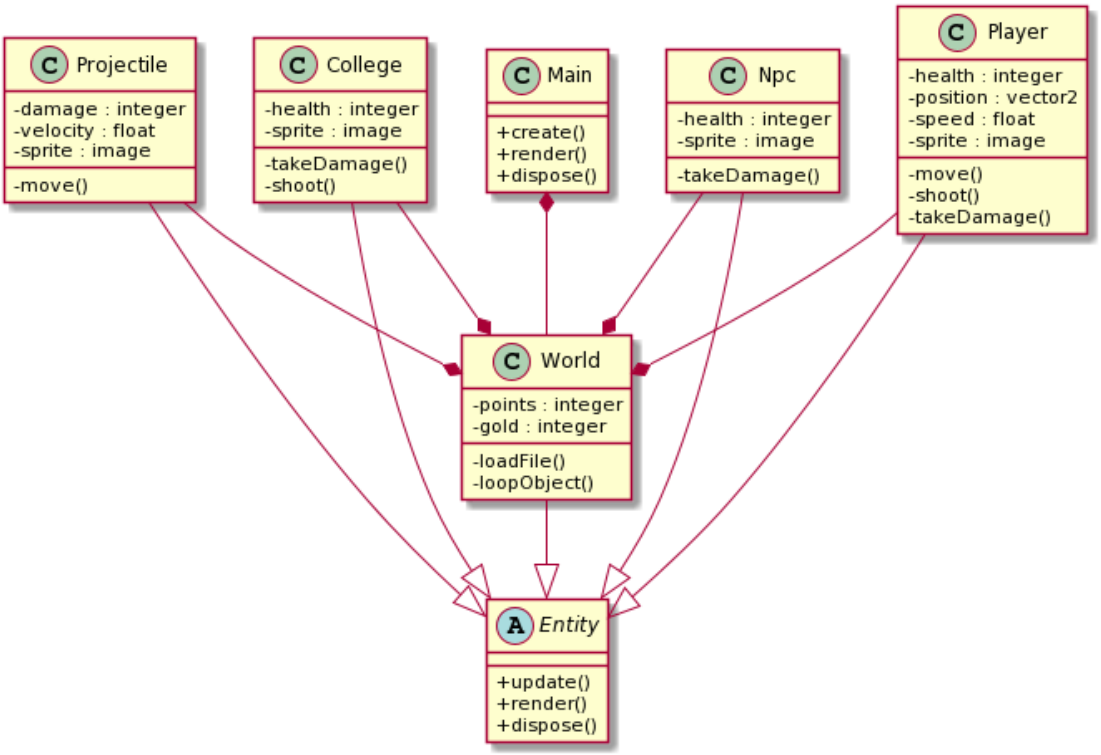
Ryan Bulman
Frederick Clarke
Jack Ellis
Yuhao Hu
Thomas Nicholson
James Pursglove

A.

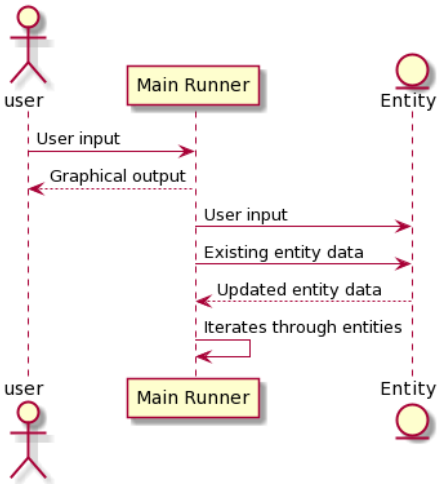
Abstract architecture:

We created three different prescriptive diagrams to use when creating our game. We created a class diagram using PlantUML to show which classes we would need and to help us allocate workload. We created a sequence diagram, also using PlantUML, to show how these classes would interact with each other and the user. Finally we created a simple component diagram to show how our assets would be stored and loaded, again using PlantUML.

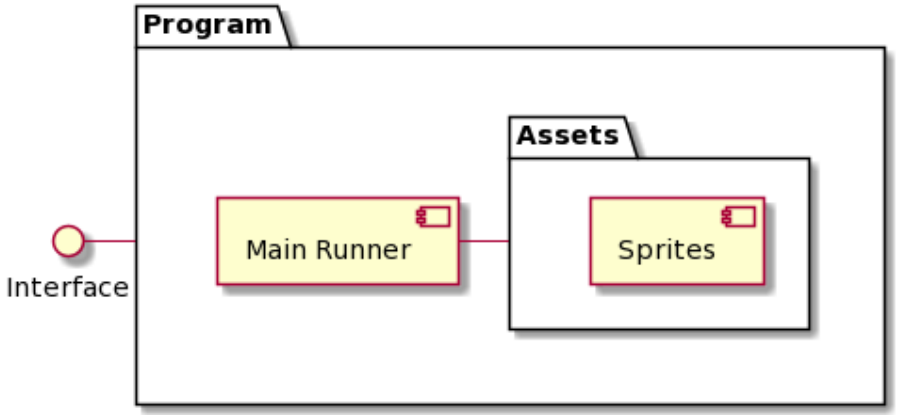
Class Diagram



Sequence diagram



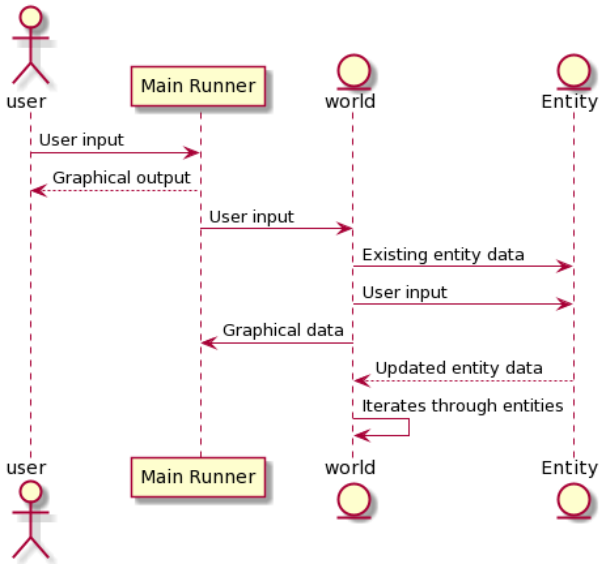
Component Diagram



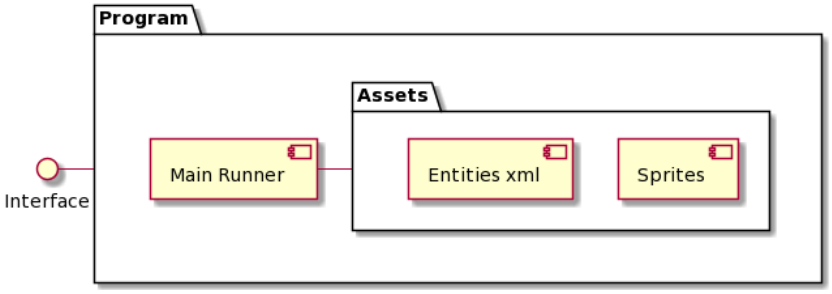
Concrete Architecture

After developing the game we made three descriptive diagrams to show how our project was actually structured. We used the same three diagram types as for our abstract Architecture: Class, Sequence and Component diagrams. We again created all of these with PlantUML.

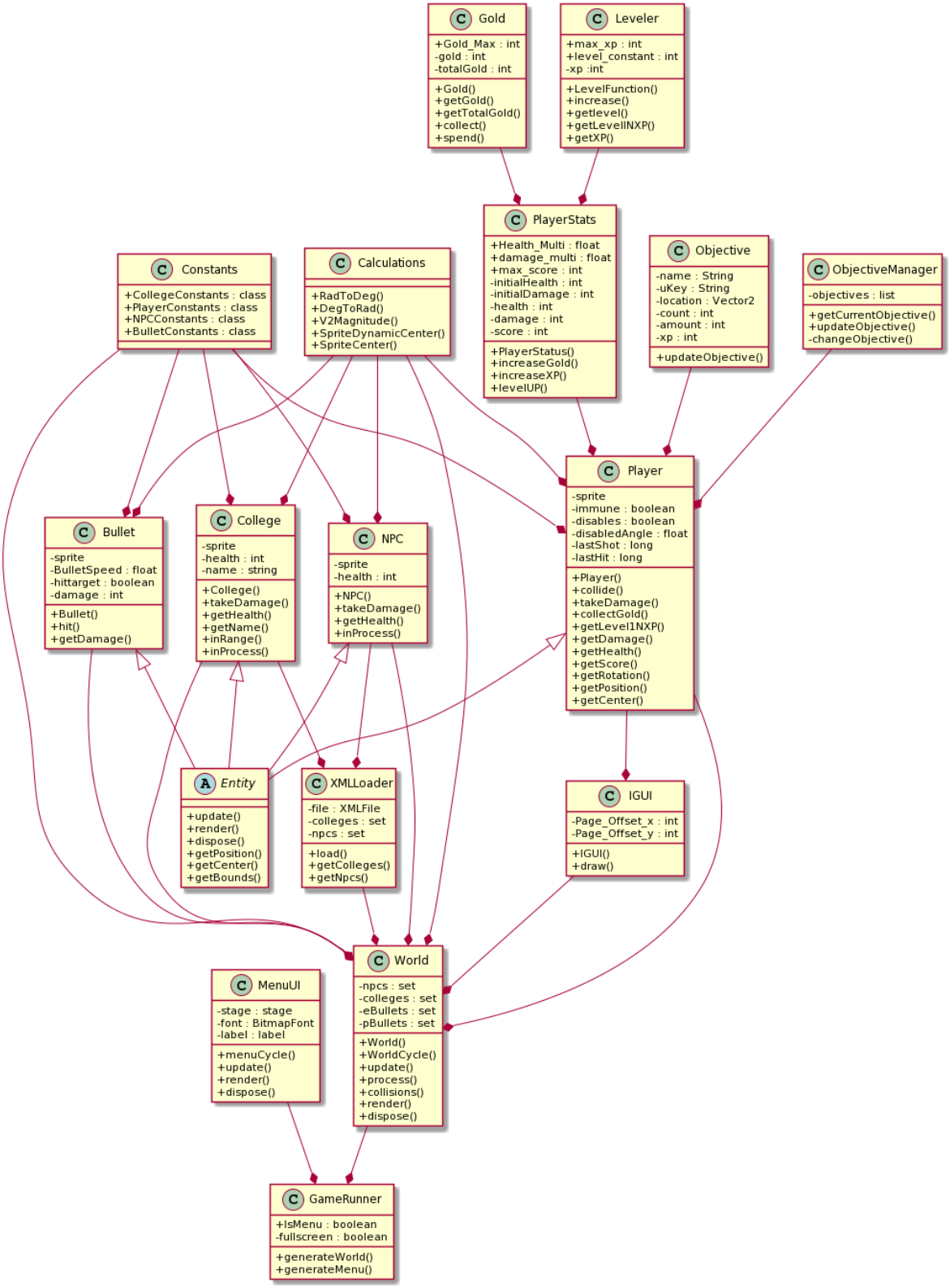
Sequence Diagram



Component Diagram



Class Diagram



B.

Our abstract architecture consists of a single abstract class entity that 5 different classes inherit from. Four of these: Projectile, College, NPC and Player are going to be stored in a set in World. The World class will iterate through this single set of entities and use polymorphism to run their public functions of update and render on every game tick. We would have a single long set containing game updates, such as object positions and user inputs, that would be passed into each of these entities during the update function. The update function would then parse this set for important data and update the entities information accordingly. The update function would then return an altered version of the set to be passed into the next entities. The render function would be very similar for all entities and would consist of drawing that entities sprite to its position on the screen. The world object would also be an entity but would only exist in the main runner.

This approach has a lot of advantages. For example by having World as an object in main we can easily reset the program by destroying and recreating our world object. It is easier to divide up workload by having people develop different entity objects without interfering with each other since none of the objects interact directly. Using polymorphism allows us to implement new entity types in future without needing to alter the main loop, such as a weather hazard or a friendly npc. Having the render function within the entities makes rendering the current game state significantly easier since the texture and position of an entity is all stored in the object and can be easily called by the render function without giving public access to the main runner.

When we actually began implementing our classes we realized that we would need to change a few things about our architecture. One thing we did relatively early was make world a separate class to entity. This made a lot of sense since world was never being rendered and had no position meaning that it didn't use the majority of the functionality afforded by being an entity. World being an object did still serve its original purpose of making it easier to fulfill the requirement that the game is restartable (FR_RESTART_REPEATABLE). Another major architectural change was that we would load our entity information from an xml file. This made it easier to meet the requirement that the game contain 3 distinct colleges (FR_COLLEGE_DESIGN) by allowing us to define colleges using xml. Another architectural change was having a Constants class. This class contained set information on how players and npcs should behave. Combining this class with the xml reader functionality made meeting the requirement of wiping all the memory on restart (FR_RESTART_CLEAR) easier since after wiping the memory and states the game could be loaded again using the xml file and constants class.

Another thing we changed during implementation was how we iterated through our entities. We had originally planned to load them all in a single set and iterate through that. We instead loaded them into separate sets based on which type of entity they were. This allowed us to pass more specific information into the different objects and interpret their returned information more efficiently. This helps us to meet our requirement of having the game run on a department machine that might not have particularly good hardware (NFR_HARDWARE_REQUIREMENTS) by making the game run more efficiently.

A requirement we had failed to consider during our abstract architecture design was objectives. We had intended for objectives to fall under the player object which was partially true however we ended up creating two additional classes that the player object could call. The first was objective manager which would be a single object that would track and display the objective the player was currently on. The second was objectives which were objects that contained information on what requirements needed to be met to advance to the next objective. This architecture allowed us to more easily meet the requirement of having multiple stages of objective (UR_OBJECTIVES_STAGE) by allowing us to generate customisable objectives as objects rather than hard coding in a set list of objectives.

Another requirement was that the game would track gold and xp values (FR_XP_GOLD) and that these could be increased by completing objectives. To meet this requirement we created three new classes: PlayerStats, Leveler and Gold. Gold and Leveler are very simple and similar with both only storing a few integer values and functions to get and update these values. PlayerStats calls these two functions and performs calculations on them such as checking if the player has leveled up. This class is called by the Player object when it needs to get information on the player's current gold and XP values. These values are updated upon the death of an npc or college by passing the constant values of xp gain into the player object.

Another requirement we failed to consider was that we were required to have a UI overlay. This was required in two different places. We needed a UI for the main menu when people first boot up the game and a separate UI overlay to display information during gameplay. We created a class for the main menu UI called MenuUI that was called directly by the game runner. For the overlay we created a class called IGUI that was called by World and was drawn on every game tick. This object calls the Player object to get its data on gold, levels and objectives. This meets the requirement of displaying the current objective (UR_OBJECTIVES_UI).

The final change we made to the architecture of our program was adding a calculations class. This class contains public functions that perform commonly occurring complex mathematical calculations such as calculating the magnitude of a vector. This class can be called by any object as its purpose is just to make common calculation functions accessible across the program.