Architecture2

Cohort 1, Team 1:

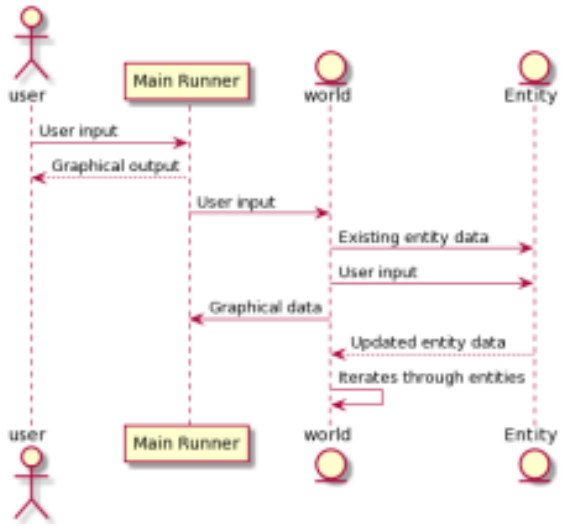
Connor Blenkinsop
David Luncan
Emily Webb
Muhidin Muhidin(MO)
Sam Laljee
Sooyeon Lee

A.

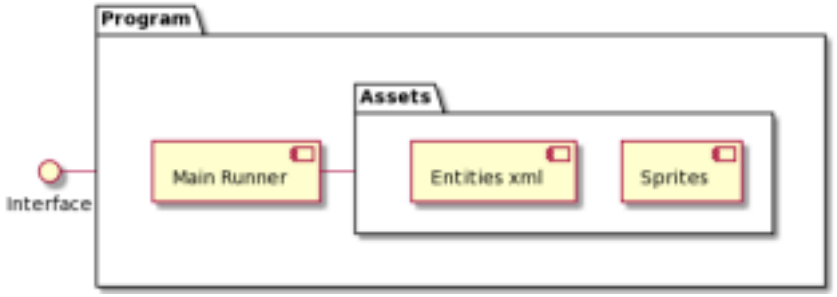
Abstract architecture:

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. This also includes the classes that have already been implemented within the game via the other group. To simplify the diagram i displayed the Player class and all the class that pass values to it

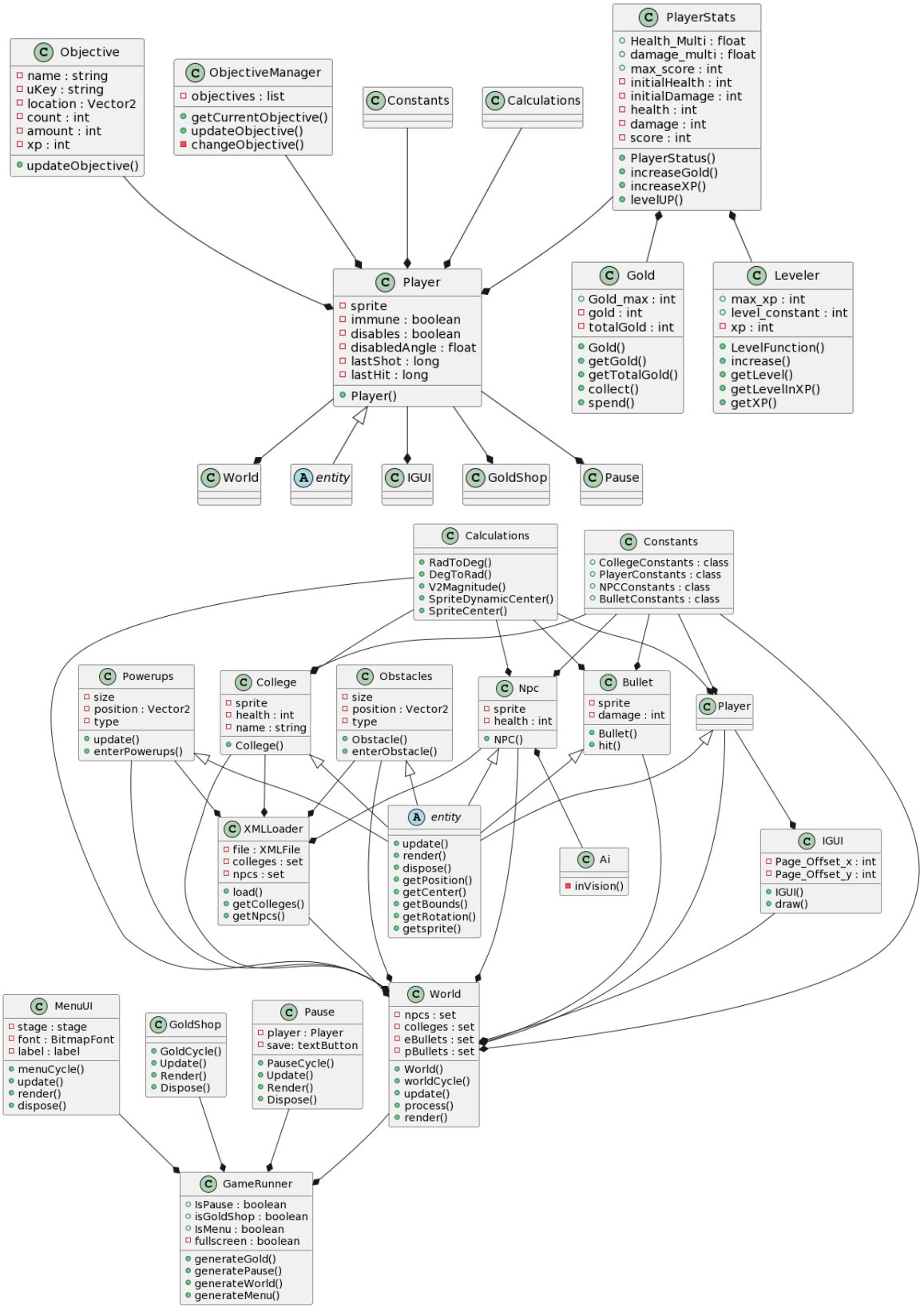
Sequence diagram



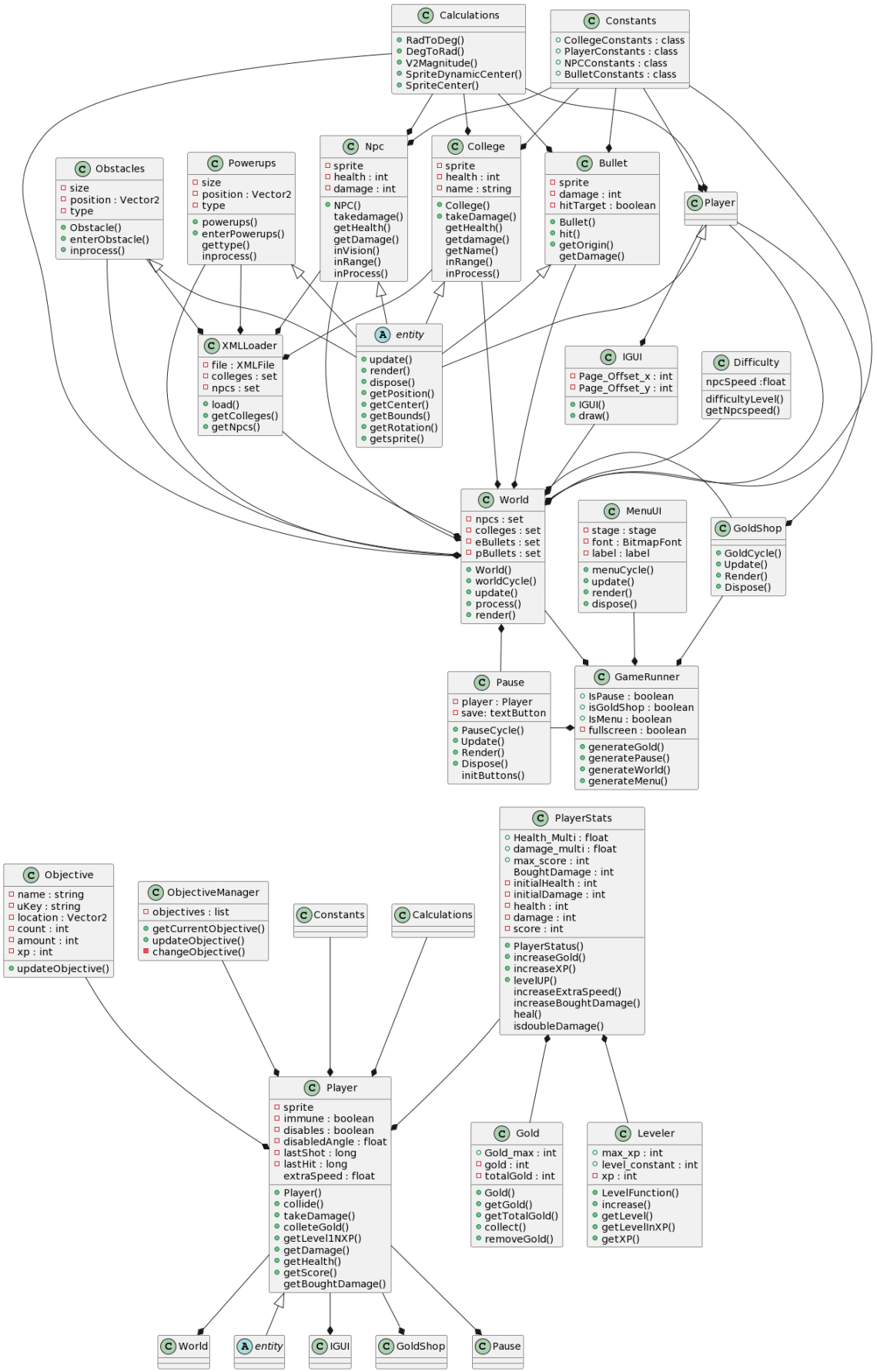
Component Diagram



Class Diagram



Class Diagram


Calculations
RadToDeg()
DegToRad()
V2Magnitude()
SpriteDynamicCenter()
SpriteCenter()
Constants
CollegeConstants : class
PlayerConstants : class
NPCConstants : class
BulletConstants : class
Obstacles
size
position : Vector2
type
Obstacle()
enterObstacle()
inprocess()
Powerups
size
position : Vector2
type
powerups()
enterPowerups()
gettype()
inprocess()
Npc
sprite
health : int
damage : int
NPC()
takedamage()
getHealth()
getDamage()
inVision()
inRange()
inProcess()
College
sprite
health : int
name : string
College()
takeDamage()
getHealth()
getdamage()
getName()
inRange()
inProcess()
Bullet
sprite
damage : int
hitTarget : boolean
Bullet()
hit()
getOrigin()
getDamage()
Player
XMLLoader
file : XMLFile
colleges : set
npcs : set
load()
getColleges()
getNpcs()
entity
update()
render()
dispose()
getPosition()
getCenter()
getBounds()
getRotation()
getsprite()
IGUI
Page_Offset_x : int
Page_Offset_y : int
IGUI()
draw()
Difficulty
npcSpeed :float
difficultyLevel()
getNpcspeed()
World
npcs : set
colleges : set
eBullets : set
pBullets : set
World()
worldCycle()
update()
process()
render()
MenuUI
stage : stage
font : BitmapFont
label : label
menuCycle()
update()
render()
dispose()
GoldShop
GoldCycle()
Update()
Render()
Dispose()
Pause
player : Player
save: textButton
PauseCycle()
Update()
Render()
Dispose()
initButtons()
GameRunner
IsPause : boolean
isGoldShop : boolean
IsMenu : boolean
fullscreen : boolean
generateGold()
generatePause()
generateWorld()
generateMenu()
Objective
name : string
uKey : string
location : Vector2
count : int
amount : int
xp : int
updateObjective()
ObjectiveManager
objectives : list
getCurrentObjective()
updateObjective()
changeObjective()
Constants
Calculations
PlayerStats
Health_Multi : float
damage_multi : float
max_score : int
BoughtDamage : int
initialHealth : int
initialDamage : int
health : int
damage : int
score : int
PlayerStatus()
increaseGold()
increaseXP()
levelUP()
increaseExtraSpeed()
increaseBoughtDamage()
heal()
isdoubleDamage()
Player
sprite
immune : boolean
disables : boolean
disabledAngle : float
lastShot : long
lastHit : long
extraSpeed : float
Player()
collide()
takeDamage()
colleteGold()
getLevel1NXP()
getDamage()
getHealth()
getScore()
getBoughtDamage()
Gold
Gold_max : int
gold : int
totalGold : int
Gold()
getGold()
getTotalGold()
collect()
removeGold()
Leveler
max_xp : int
level_constant : int
xp : int
LevelFunction()
increase()
getLevel()
getLevelInXP()
getXP()
World
entity
IGUI
GoldShop
Pause

Concrete Architecture

After developing the game we made a descriptive diagram to show how Class Diagram was actually structured, as the sequence diagram and Component diagram is the same as the abstract architecture sequence diagram and Component diagram. It Adds a lot of the methods that were missing in the abstract architecture.

B.

Our abstract architecture consists of a single abstract class entity that 6 different classes inherit from: Obstacles, PowerUps, Projectile, NPC, College and Player. The world class will load each entity type onto a different set allowing it to receive specific data depending on the entity; To store this information we used an xml file with entity information so the World class could get their data for when the game starts. This also made it easier to restart the game as a Constants class and a xml reader functionality could load the game again using the xml file and constants class. The World class ran the entities' public methods via polymorphism, running their update and render function on every game tick. The update function would then parse the sets for important data and update the entities information accordingly. The update function would then return an altered version of the sets to be passed into the next entities. The render function draws the entities sprite to the position on the screen. The world object 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. 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.

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 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. To display these values we used an IGUI class which also displays: the score, the level and the next objective. The objective

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.

The Ai class was removed from the concrete architecture as it was not necessary instead the function of the class was implemented into the update function of the world class this allows us to check whether the npc that the method is running for is close enough to the player. This removes an extra call to the world method outside the main class removing some complexity.

We also added a class that calls the different values for npc speeds this allows the game to scale its difficulty based on the time pass in the game, this means that the games will get harder the longer you are in it. This means as you are getting more stats the npcs are also getting stronger keeping up with the player.

In the World class the goldShop and Pause menu would be callable without disposing of the world class. This means that the current instance of the world class is not lost when in the goldshop or pause screen. This is used in the goldShop when you spend your gold as it is passed the player instance from the world class and can change its stats and current gold. Whilst in either of these screens the world class does not change at all returning to the exact moment of when you went to that screen in the world class.

Requirement	How this requirement has been met by the architecture	Method or class
UR_FIRST_TIME_PLAYER	The Objectives class retrieves the instructions from the xmlLoader to allow the user to follow the objectives to learn the game	Objectives
UR_USABILITY	The xmlLoader allows the game to played on many types of pc as it efficiently gets information for the entities on the xml file	XmlLoader
UR_GAME_LENGTH	The game shall be playable in a single session	Objectives
UR_COLLEGE_FIGHT_BACK	Colleges should be able to fight back in some seemingly intelligent way	Shall
UR_GAME_CURRENCY	The game shall have gold which the player can spend, and XP to level up	Shall
UR_POWERUP	Powerups are a presetup entity in the entity xml that can be collected	Powerup
UR_SHIP_COMBAT	Npc. is used to see if the player is in range of the ship where is will chase the ship and attack it when in range	npc.inVision()
FR_LOSE_GAME	Return a boolean that will return true if player health is below 0 which can be used to go to the main menu	playerStats.takeDamage()
FR_SHOPPING	This will allow the player to spend their money by clicking buttons, only allowing you to buy an upgrade if you have enough money and updating it on the shop screen	GoldShop()
FR_POINT_EARNING_WEATHER	When in weather different effect will occur depending on the type of obstacle	Obstacles
UR_GAME_CURRENCY	Gold stores the money that can be earned by defeating colleges and ships. Xp is earned the same way and is used to contribute to a level system	Gold, XP