ENG1 Architecture

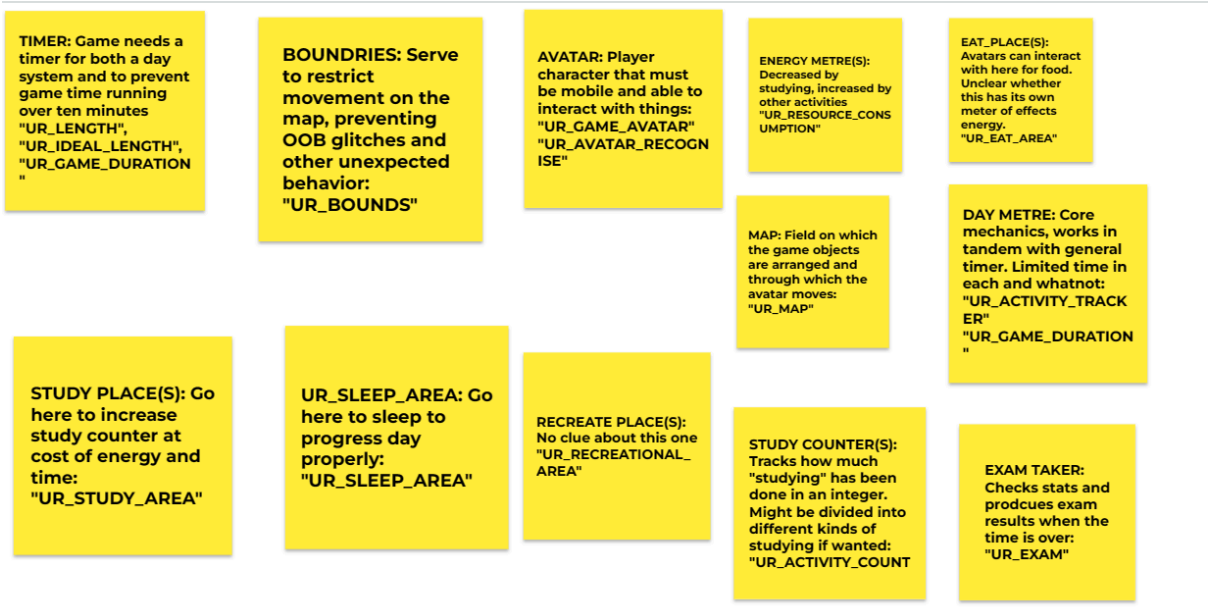
Cohort 3 - Group 28

“Team 28”

Muhammed Salahudheen
Joel McBride
Jamie Rogers
Maciek Zaweracz
Rhys Yeaxlee
Alex Spencer
Alex Firth

We largely followed the methodology discussed in Fundamentals of Software Architecture : An Engineering Approach Part 1 Chapter 8. A Google Jamboard was used to arrange the earlier concepts, while a UML diagram on Draw.io was used for the later class diagrams.

CRC CARDS

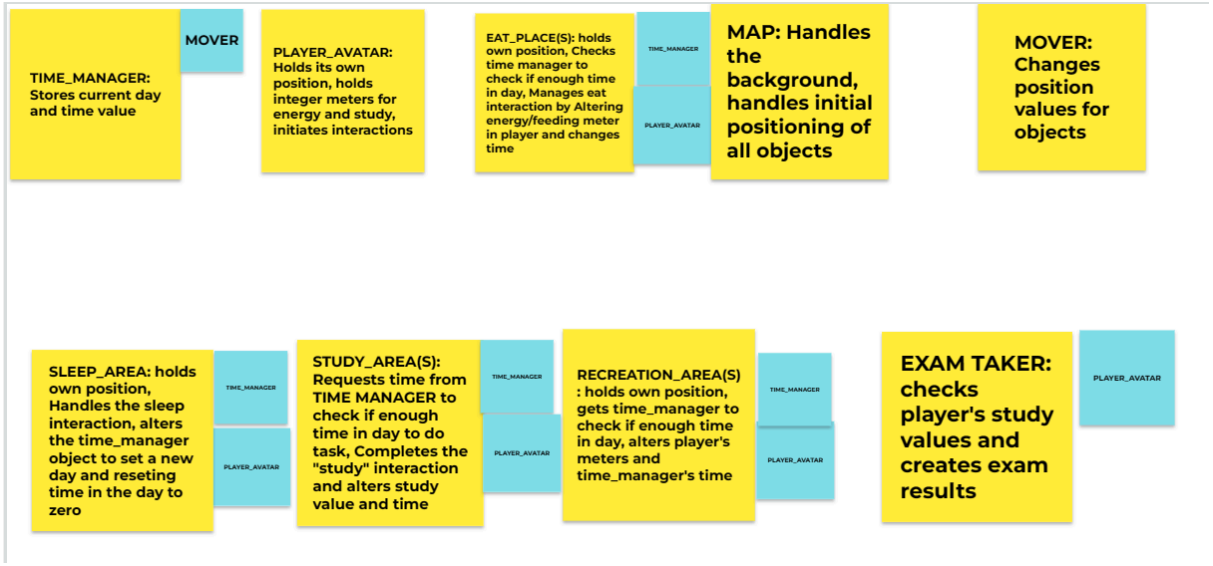


Candidates were created to closely follow the requirements, describing the intended function and listing the User Requirement within. Other team members clarified about how certain mechanics would work within the libgdx (i.e. how bounds should be handled and how movement should be thought about), resulting in the next stage of cards where there was no boundary candidate and a separate movement object was used to move objects like the avatar.

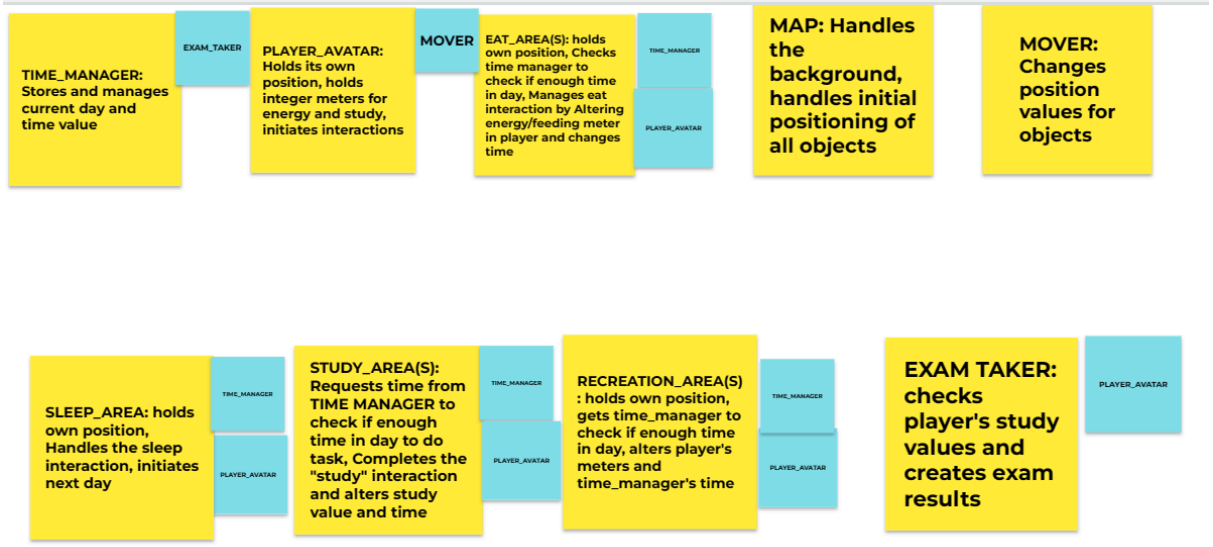
Stereotypes:

At this stage the stereotypes were relatively simple to assign.
Feedback from teammates indicates that the study counters and energy meter should be part of the avatar object as they all relate directly to the player and there's no reason to have to fetch that between separate objects. Timer and day meter are also merged as they effectively operate as two parts of the same mechanic.

These simplified cards will be used as the basis for the next step of defining responsibilities. Names have been changed to reflect new roles or to better identify them.

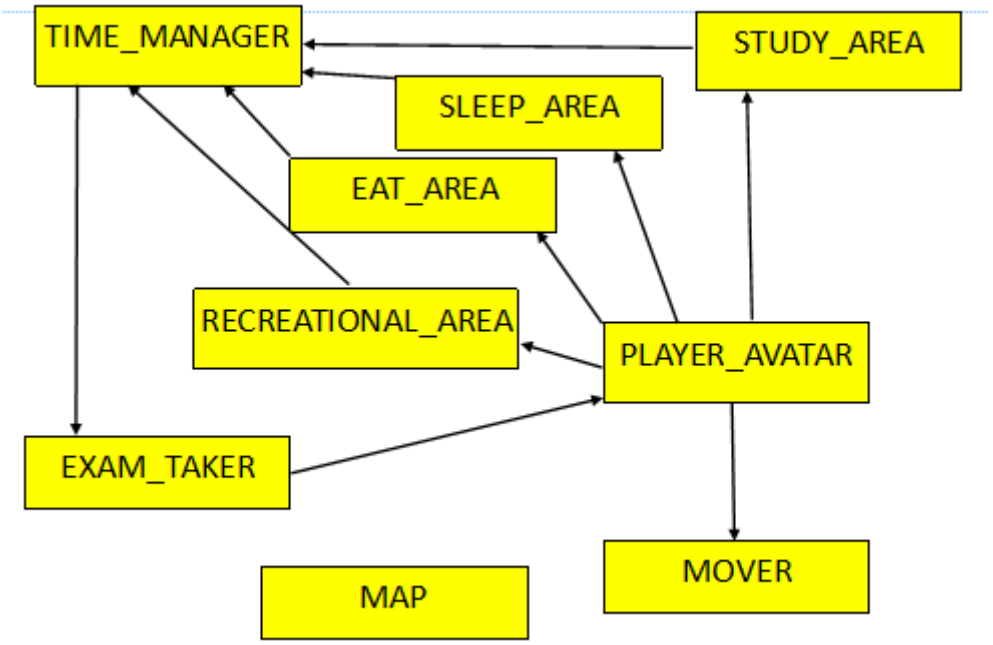


Adding responsibilities was simple as the connections in logic were not complex.



Upon further discussion with the team this was simplified to keep options open. The sleep area only initiates the day transfer, time_manager performs the specifics, meaning that it could be possible for other objects to transition days if needed.

Control Style



This basic logic map shows the logic flow at this stage. A dispersed control style is used as the game is too basic for a central smart object to be of much use.

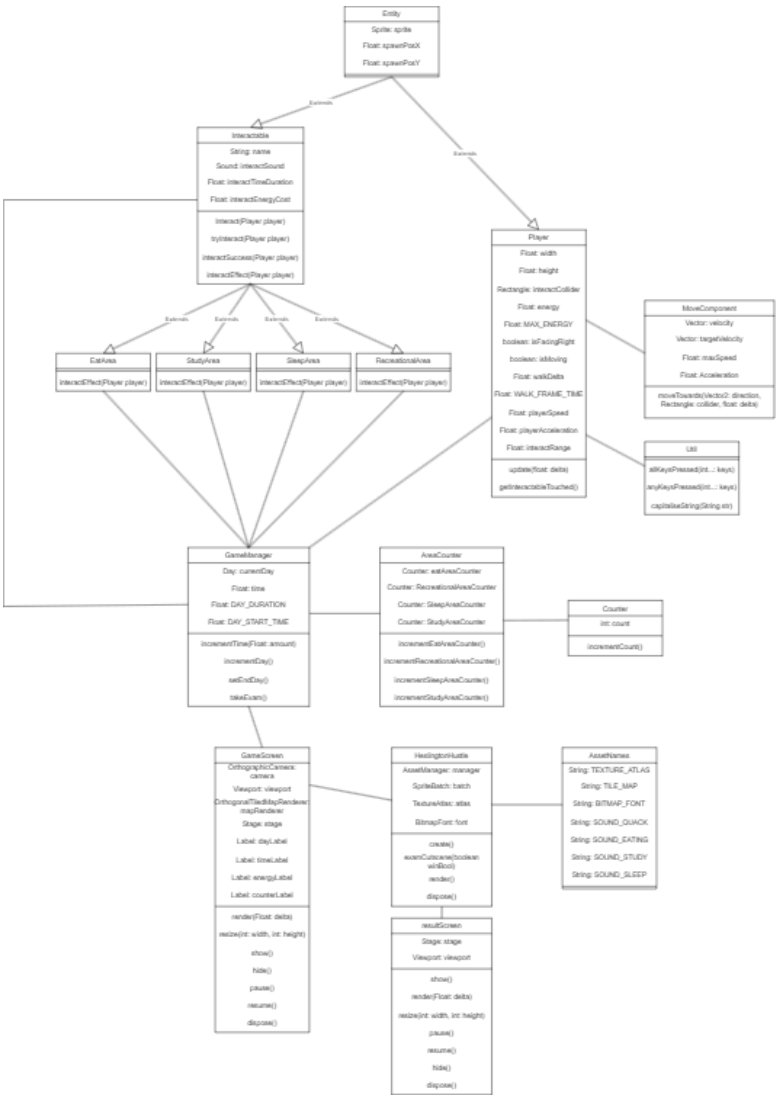
This was the initial prototype for the game architecture, this quickly began to evolve as we did more research into the libraries we were using, in this case libGDX. Now although libGDX does support entity-system architecture we chose not to make use of this as the brief specifies a game that is quite small in scale and it is likely that most systems will not be used repeatedly. As such we chose to remain using a class system. We first used a class table so we could start to explore our class structure, linking the requirements to the classes, and marking their potential implementation. We then moved on to creating a UML class diagram, since many of the classes interact without inheritance this initially did not contain a whole lot of connections. At this point we also decided to have a central class to manage any aspects of the game that were not in their own separate class. This was called game manager, and also managed the time and the interactable objects.

At this point we began to make a simple UML diagram, we had not yet decided on our full implementation and instead chose to begin creating class representations and inheritance diagrams. All of the UML diagrams can be seen at https://team28.tech/2024/03/20/architecture-process.html the initial UML diagram is small and only contains one parent class, at this point we were just figuring out what processes and classes we might need to develop the game. We realised that we needed to split up the game manager into a screen portion and a game manager, this ensured that we did not have a monolithic class, as we wanted to compartmentalise our code. At this point we had done the most work looking at interactable areas, as we identified that this would probably be the segment to cause us the most trouble.

The second UML diagram in our series was when it started to develop slightly more. The inheritance got more complicated at this point as we added in the entity class to act as a superclass between entities we were adding to the game, such as the interactable areas and the player itself. Additionally we created the main class for the game itself, HeslingtonHustle, this is the activation class, from which the rest of the script is run. Furthermore, a separate class was made for movement, separate to the player class, we also added a separate class to hold the counters for the number of interactions with the different areas.

The third UML diagram is the fully fleshed out one, containing actual logical links. We decided that in most cases where a class needs to call a method in another class that they should use the central game manager

class, this makes it easier for us to organise passing information to different classes as well as simplifying the overall class structure. Additionally at this point we added the exam logic in the main classes, mainly in game manager and added the code for a results screen to our game object, which encompasses the entire class structure. Additionally we added the assetname class to hold some of the file names of the images, also the util class to hold any extra methods we may need. Additionally we made a process diagram to show our longest consecutive process, the rest of these processes are incredibly short and therefore do not get any formal diagrams.



Requirement	Class(es)	Implementation
UR_GAME_DURATION	GameManager	The Game manager class holds a set of days that will progress as the interactable areas are interacted with
UR_ACTIVITY_TRACKER	GameManager	The current day attribute golds the current day and increment day progresses it
UR_ACTIVITY_COUNTER	Counter AreaCounter GameManager	The counters in AreaCounter keep track of all 4 interactables
UR_RESOURCE_CONSUMPTIO	Interactable	The interactable class holds the

N		method interactSuccess() which activates every time an interaction is successful, these reduce time and energy
UR_MAP	GameScreen	The Screen class from the library allows you to set a background for the game
UR_BOUNDS	Player	The player holds a rectangle that it cannot go outside of as a collider
UR_GAME_AVATAR	Player MovComponent	The player class creates an entity with a sprite and the MoveComponent allows it to move around
UR_AVATAR_RECOGNISE	GameScreen	The player is always in the middle of the screen
UR_STUDY_AREA	StudyArea	StudyArea contains the the interactable affect that increments the amount studied, This along with its inheritance creates an entity that can be interacted with
UR_SLEEP_AREA	SleepArea	SleepArea contains the the interactable affect that increments the amount slept, This along with its inheritance creates an entity that can be interacted with
UR_RECREATIONAL_AREA	RecreationalArea	RecreationalArea contains the the interactable affect that increments the counter for recreational activies, This along with its inheritance creates an entity that can be interacted with
UR_EAT_AREA	EatArea	EatArea contains the the interactable affect that increments the amount eaten, This along with its inheritance creates an entity that can be interacted with
UR_Exam	ResultScreen GameManager	The GameManager holds the logic for the exam to start, and then passes it to the result screen, which will then show, displaying either pass or fail depending on the score calculated.