Architecture

Assessment 2 - Updated from Team 21

Hannah Thompson
Kyla Kirilov
Ben Hayter-Dalgliesh
Matthew Graham
Callum MacDonald

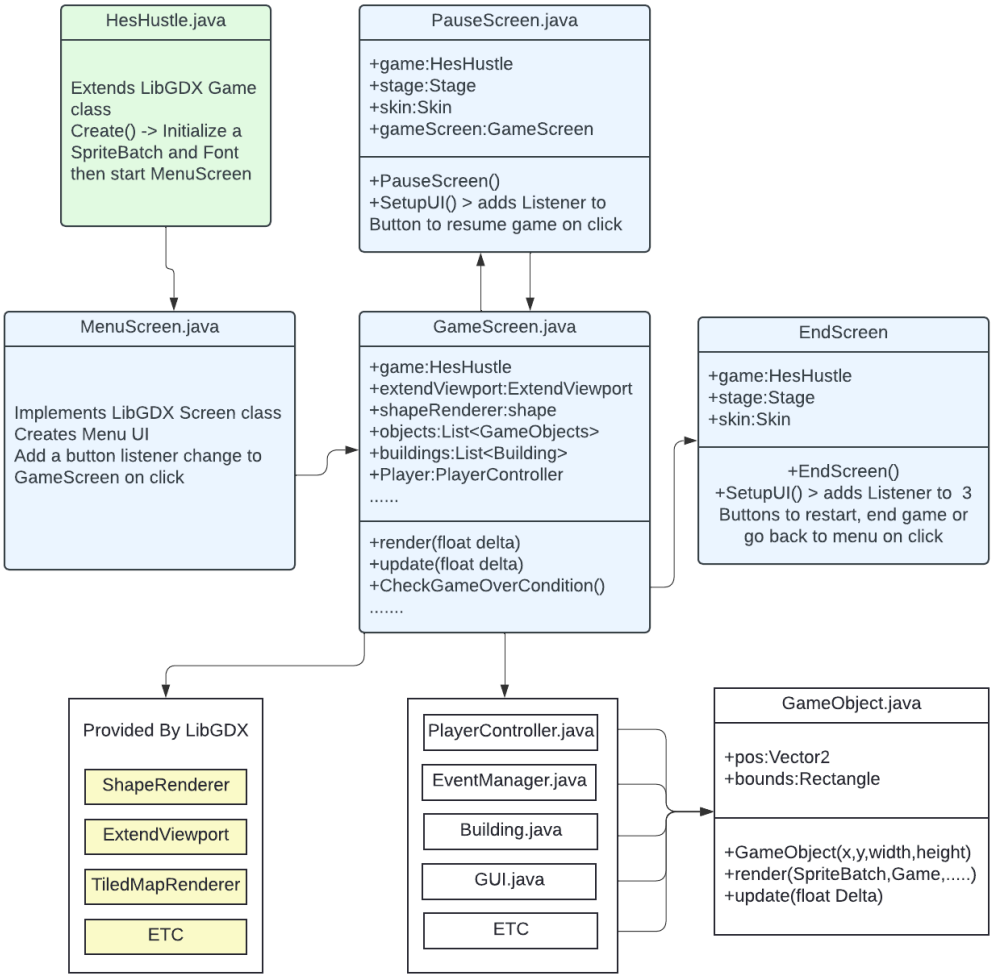
Chak Chiu Tsang

Assessment 1 (Team 21)

Initial Design:

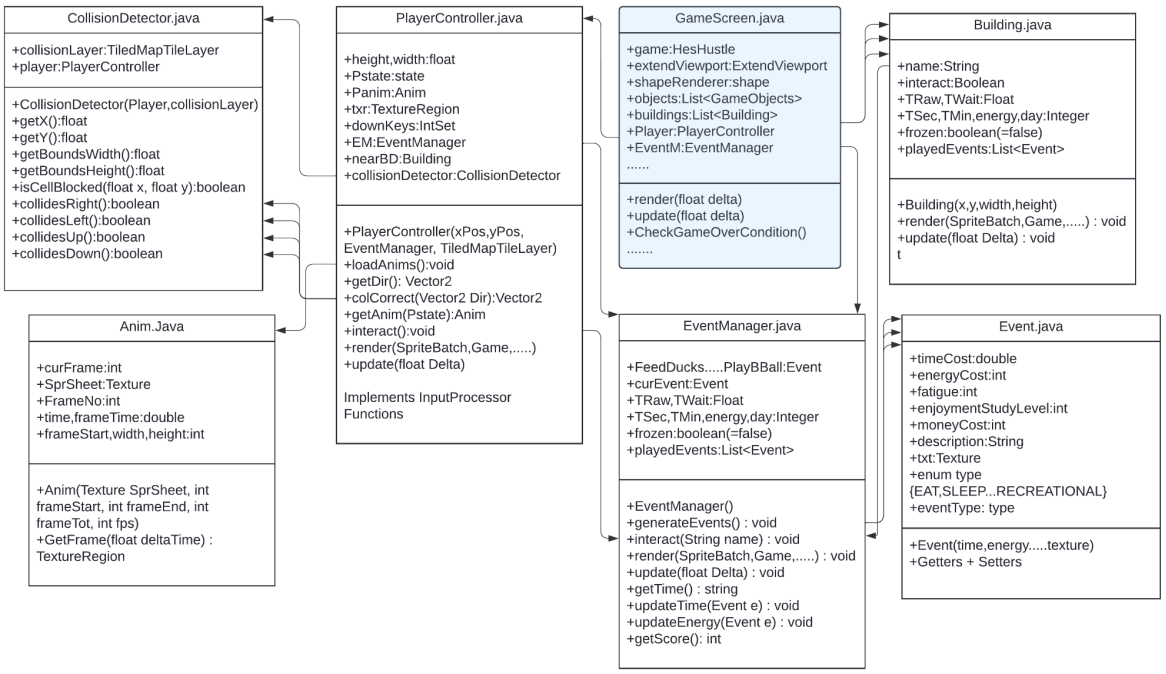
As the design and architecture of our project progressed, we used CRC cards to create our first structure. We started off listing each element of the program and refined it down to meet our requirements. Our final CRC cards serve as the bridge to our UML architecture. You can find our CRC cards here CRC cards, or on our website. In the CRC document, the slides show the evolution of the cards, with the final slide showing our final cards. Additionally, the themes and goals we used to create these are included. The final iteration shows the correct naming convention, linking to our future UML diagrams.

Architecture stye [1]:
Our product uses an entity-component based architecture where most classes inherit from a base GameObject class. We followed this architecture as most of the team had dabbled in game development softwares such as Unity and GameMaker which meant we had some base understanding of how the system worked. We assumed it was likely the team that takes on our project would include someone who has used similar software which would make our product more appealing in the presentation aspect of the module. We also chose this architecture in order to work more efficiently in development. At times we had 3 people working on the game simultaneously as we could work on separate classes without any major interference.

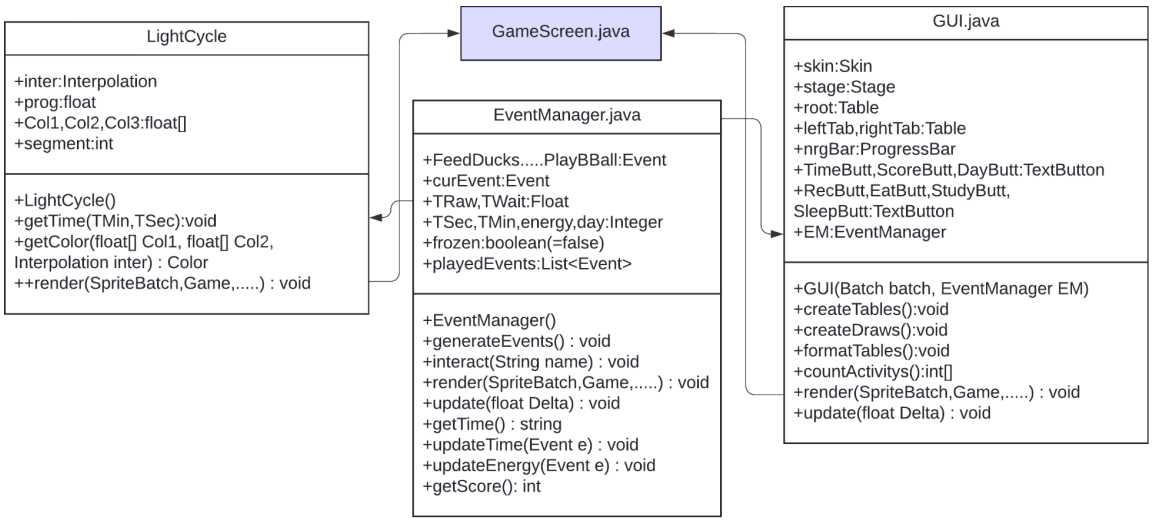


The base components of our architecture are Screens, which inherit from the Screen class provided by LibGDX, GameObjects created by us and rendering classes provided by LibGDX such as ExtendViewport, TileMapRenderer etc. The GameObject class stores the fundamental variables and functions required by an object, the position vector, a bounds rectangle, update function and render function. This made it very easy to streamline the rendering process which is done through the Screen. In our product the GameScreen class is the only screen that actually has objects to render and update as the rest are used for menus and utilise the LibGDX Scene2D package for UI. As most of the game objects in GameScreen inherit from GameObject, we used an ArrayList of GameObjects to loop through all objects and render and update them using a for loop. This allows for future expansion if any more object classes are created they can be added to the list and automatically rendered. The render() in GameScreen is used to call render in the objects in the list and pass the necessary parameters, the viewport projection matrix, the HesHustle game (to access the spritebatch and font) and a shape renderer (for generating debugging rectangles etc.). There is no built-in update method in LibGDX so we created our own, called at the start of render(), by passing the deltaTime provided in LibGDX's Game.java render function.

For the Rendering we use an ExtendViewport so that our game is scalable however this causes issues as the camera zooms in and zooms out so the pixels on the screen are different to the coordinates moved by the player. This was especially annoying when trying to stop the camera moving out of bounds. We had to implement an update camera method which calculates the difference between the pixel distance and actual coordinate distance so the viewport will only have black bars in one orientation and not both. To Render the Map we used the integrated Tiled libraries for TileMaps. We hoped to extend the TileMapRenderer so we could decide which layers in the map would be rendered over the player (eg trees and lampposts) and under the player however due to time constraints we couldn't implement it.



The player controls the PlayerController object which is initialised in GameScreen. Using the LibGDX InputAdapter, we noticed it struggled with multiple direction inputs so we implemented a method for multiple presses. This is why we store the inputs as a dynamic list of keycodes, if two opposing directions are both in the list it will remove them both allowing for a smoother movement system. We added a collision component to PlayerController which is created using the CollisionDetector class. This allows the detector to read the collision layer of the TileMap and inform the PlayerController of any collisions using the colCorrection() method which adapts the direction vector from the inputs by setting the x or y component to 0 depending on which sides the collisions are on.The player controller also stores multiple animations for each (walking and idle) and swaps the animation currently rendering depending on the Pstate (Player State). The animations have a constructor method which allows for custom width sprite sheets which we thought would benefit the future group who takes over. We kept the Anim class separate from the player as it would be easier to give different objects animated visuals.
After the PlayerController, the EventManager is the second most important class we created as it facilitates communication between the buildings on screen and the player as well as storing the energy, time and score variables and methods. When the player goes near a building, GameScreen will change the NearestBD variable in the player to that building, if the Player then presses Space to interact, the name of the building is passed to the event manager who begins the appropriate event (currently an event freezes the player and displays an image as a placeholder for future development). The EventManager checks if there is enough energy to proceed then processes all the variable changes the event provides then finally calculates the current score of the player. It also keeps track of time for the player, using deltaTime to increment TSec at a custom float value and TSec to increment Tmin at 60.
We Initially wanted separate classes to control the time, score and energy however after implementing the event class we realised a central game manager would be more efficient as other objects could reference the EventManager for info so rather than passing 3 objects to each constructor we could store it in one, an example of this is our GUI class.



The EventManager is heavily used in the GUI of the game as the energy, time, score and number of each type of event needed to be displayed. The GUI component contains its own ExtendViewport and uses Scene2D to display the necessary information over the main

render pipeline. Similarly the LightCycle class takes the time variables from EventManager and calculates the interpolation value to achieve a gradient effect in the day-night cycle. We made GUI a separate class as it helped continuous integration and we did not want to make the structure of the GameScreen class too cluttered.

Structural diagrams [2]:

First Structure

As we moved on from the CRC cards, we created an initial class diagram for the four main classes we planned on using as shown below:

Class diagram 2

- PlayerCharacter stores all of the elements needed for the player to complete an event. (Player energy etc.)
- Event stores the requirements the player needs to carry out an event as well as its impact on the player and score.
- Main handles input, calls the other classes and calculates the score.
- Time keeps track of the days and hours and handles progression to the next day.

Class diagram 1

This Diagram shows a detailed overview of the Main class and its interaction with the other classes.

Final Structure

Our final structure moved away from these classes, with HesHustle initially creating the game and setting the screen to MenuScreen. Below is an explanation of each class along with a class diagram representing each one.

Final Class Diagram(Main Classes)

- GameObject acts as a parent class to many of our classes and controls the rendering of objects
- EventManager handles the creation of events and their interaction/pop-up screen. It updates the energy and time after an event is carried out.
- Event stores an event's information as well as the energy and time required to carry out an event.
- PlayerController handles the player movement, location and required animations. Creates a collision detector

Final Class Diagram(Screen)

Each screen inherits from the LibGDX class Screen and uses its methods shown in the class diagram above.

- MenuScreen brings up the menu screen after HesHustle creates the game, Giving the player the option to Play or Exit.
- GameScreen appears when the player chooses to play. It renders all of the objects and moves the camera as the player moves. It checks whether to move to the EndScreen when the player reaches the final day. Also, it controls functionality to resize the window
- PauseScreen handles the pause menu and allows the player to Resume or go to the Main Menu.
- EndScreen handles the ending screen which appears when the player finishes the game. Gives the player the option to Play Again, Exit, or return to the Main Menu.

Final Class Diagram(Misc Classes)

- Anim handles the animation for player movement.
- Building stores information about each building.
- GUI handles the GUI and displays the time, day and energy. It also shows how many of each event type has been carried out.
- TiledTest handles the camera and displays the graphics.
- LightCycle handles the lighting progressively as the time increases, simulating a day/night cycle.
- CollisionDetector handles collisions by creating a layer above the map which renders areas inaccessible to the player.

Class Relationships

- HesHustle inherits from the Game class from LibGDX.
- EventManager, Event, Building, GUI, PlayerController and LightCycle all extend GameObject
- MenuScreen, GameScreen, EndScreen and PauseScreen extend Screen

Behavioural diagrams [3]:

1. Sequence diagram [4]:

Sequence diagram

This sequence diagram represents the initial plans for the implementation of the core gameplay loop. The player interacts by moving across the screen, when they interact with an icon in the game (representative of an event in-game) the main logic runs, checking remaining time and the remaining energy. If the event is valid, then the time and energy it requires is taken from the time and playerCharacter respectively. Then the hours are reset and days decreased if the event would be the final one possible in a day (meaning it would result in the player having 0 energy or 0 hours remaining), this is checked first for if the day would end due to time and then for energy. To indicate the start of a new day, the amount of energy and time they have is reset. This process repeats until the player has played out 7 days, where the score is outputted to the player via a win screen.

In this flow diagram the other aspects of implementation, such as the user inputs and how the game will be displayed are omitted. This is due to the fact that for differing game engines, there will be different implementations of these features, so to have the diagram be useful for planning, these were abstracted into simply a user input and an output at score.

The sequence diagrams below show the flow of the interactions in the game and the cause and effect of the player's decisions as well as how the game system processes those actions and how it will be interacted with.

Eating diagram

- The student selects the Piazza building to interact with and the game system cheeks if the student has enough time and energy to eat
- Once confirmed, the student's energy increases in the game and time decreases and the student can eat

Recreational diagram

- The diagram shows what happens when the student wants to interact with the ducks and feed them
- The game system verifies if the student has enough energy and time to do this and if so, the action can be carried out. Then energy and time are deducted depending on this activity and the user can complete the activity.

Sleeping diagram

- Diagram shows what happens when the student selects the Bedroom option within the game and wants to sleep.
- The game system executes the 'perform sleep action' and then it increases the students' energy
- This triggers the 'Move to next day' action and is repeated until day 7

Studying diagram

- The diagram shows what happens when the student wants to study in the CS building for example
- The game system verifies if the student has enough energy and time to do this and confirms the action with the student
- The energy and time for studying is then deducted by the game system and the study action is performed which will help the student score higher in their exams

2. Use case diagram [5]:

- This shows a simplified version of what the first assessment requires in terms of the gameplay loop and what the student can do in our game
- Student clicks on 'Play' which is displayed on a menu screen
- Once the game starts the player can navigate the avatar around the map

- The sleep activity allows the player to gain energy and move on to the next day until day 7 which is when the game ends and the player's score is displayed
- As well as sleep the player can study, complete activities and eat which affect their energy levels

---

Assessment 2 (Team 25)

Justification of Tools

Team 21 used LucidChart and PlantUML to create their diagrams.

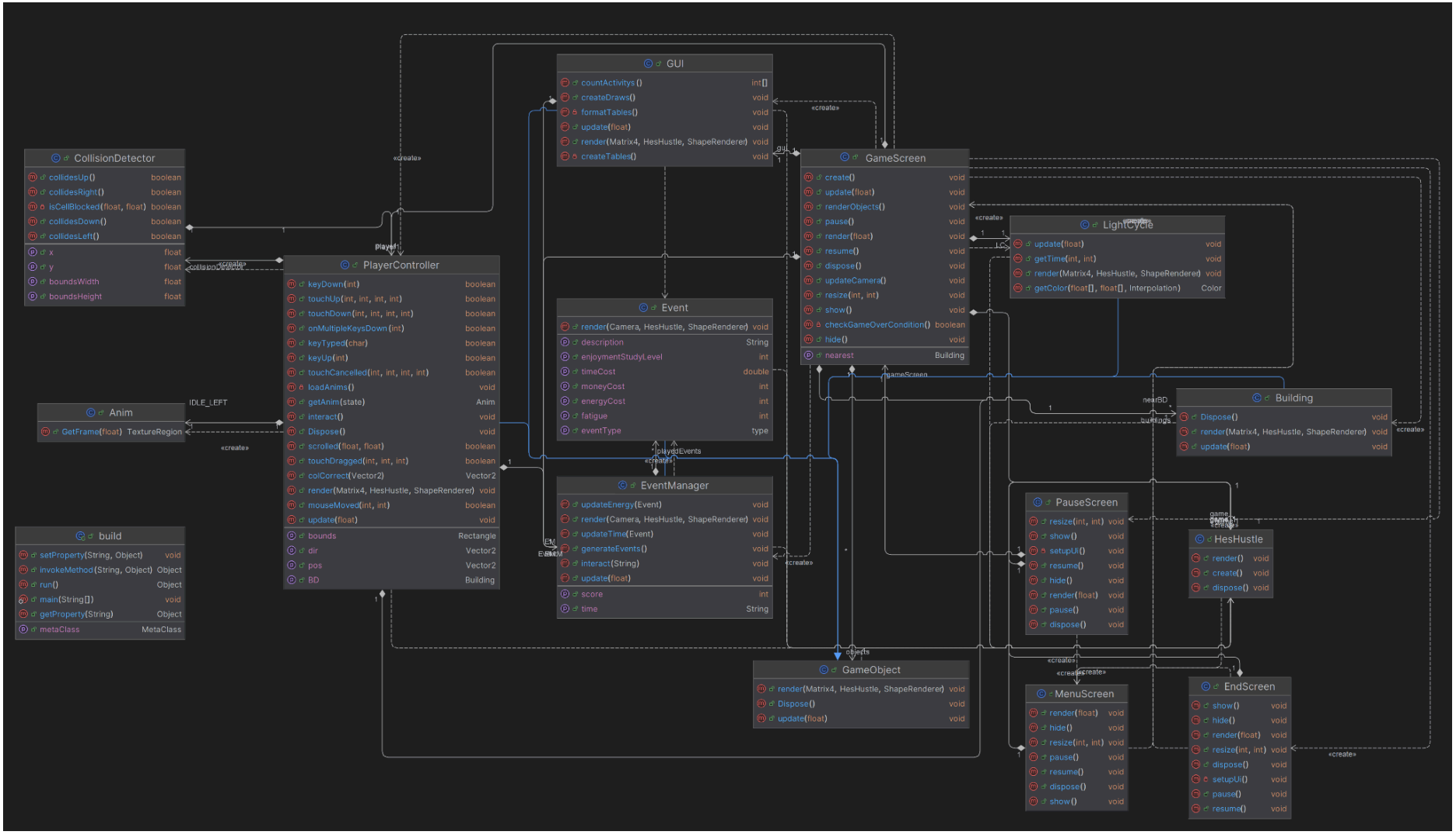
LucidChart is appropriate for this use because it has a graphical user interface that makes visualising the diagrams straightforward, while also integrating a UML generator and editor. This makes creating diagrams easier than writing the UML code by hand, and means that the generated code can be transferred to other software for further development (e.g. loaded in VS Code to make use of other Tools or auto-complete features).

PlantUML is a simple and efficient way of creating diagrams that takes little time to learn, is easy to debug and integrates well into other software. It is useful when creating class and sequence diagrams, as well as Gantt charts. This was easier than manually drawing diagrams because it is intuitive, and has autofill and suggestions that make the process quicker and easier than any other software. PlantUML was the right choice of tool for our team because it has good integration with Google Docs and IntelliJ IDEA, the IDE our team chose to use. The created diagrams can be easily exported as .png files from IntelliJ, and can be placed on our website.

When Team 25 further developed these diagrams, we used IntelliJ Ultimate's built-in 'Diagrams' feature, which generates a PlantUML template class diagram. We then edited the diagrams to make them more concise, and present three separate final class diagrams for clarity, as the full diagram was difficult to read due to the text being so small.

Firstly, we made improvements to Team 21's final class diagrams, as it was missing some key elements and other things could be improved. This is discussed in further detail in the Change Report document.

Updated Class Diagram (for Assessment 1):
This is better viewed on the website under *'Architecture, Fig. 1'*.



We then generated the new diagrams.

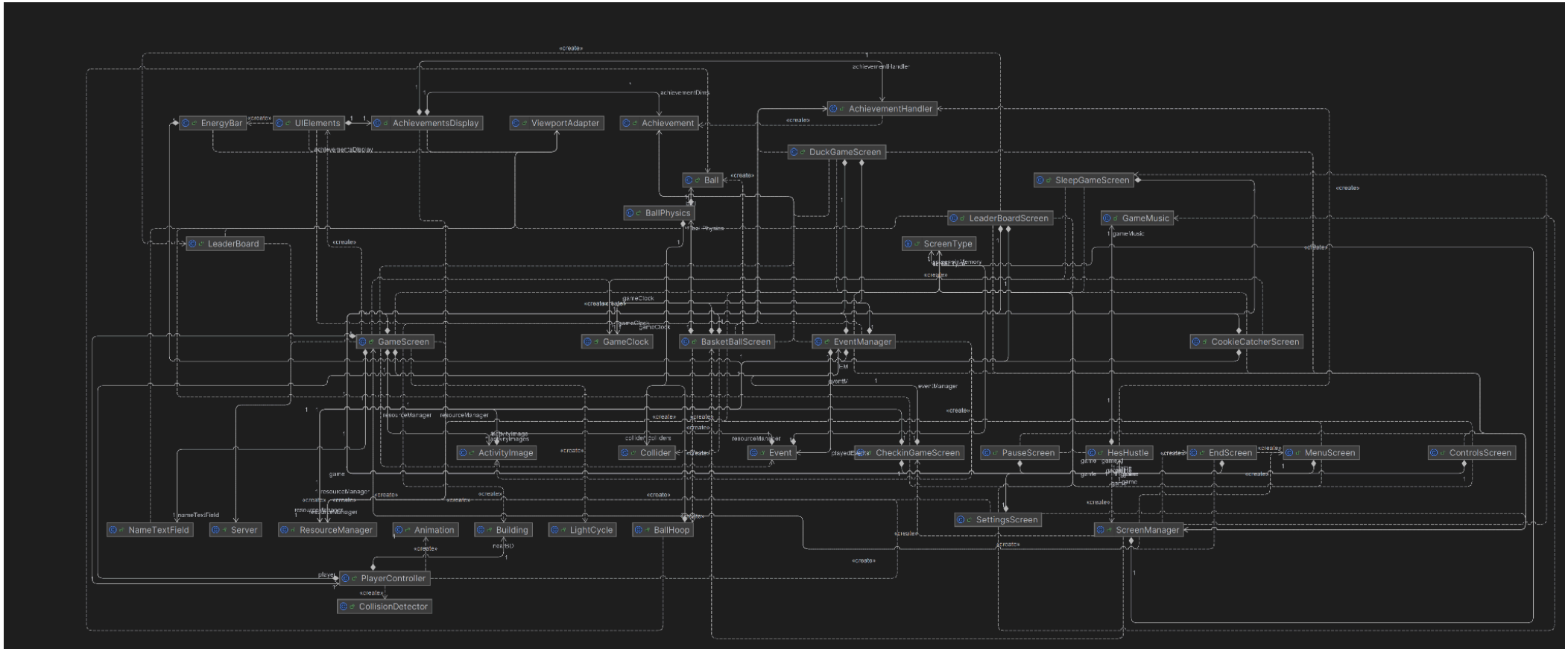
Diagram 1: Screens *('Architecture, Fig. 2')*
This diagram shows how the different screen classes are linked. Screens include the main game, settings, controls, and other options on the navigation menu. HesHustle and Server classes, as well as the main LibGDX parent classes, are included for context.

Diagram 2: Utils *('Architecture, Fig. 3')*
Includes the utility classes. These are mostly handlers, managers and resources, such as the EventManager and GameClock. HesHustle and Server classes, as well as the main LibGDX parent classes, are included for context.

Diagram 3: Objects *('Architecture, Fig. 4')*
Links the objects within the game - these are mostly characterised as the assets and things visible on the screen, in the UI and during minigames. HesHustle and Server classes as well as the main LibGDX parent classes, are included for context.

Diagram 4: Full Class Architecture *('Architecture, Fig. 5')*
Due to the level of detail in these diagrams, it was almost impossible to generate a comprehensible diagram for the full architecture. To simplify it as much as possible, we have removed everything from the diagram besides the class names and their relationships, although it is still very difficult to read. This final diagram is also presented on the following page.

Final Class Diagram for Assessment 2:



References:

Architecture style/design
[1]I. Sommerville, “Architectural design”, in *Software engineering*, 10th ed. Boston, Mass. Amsterdam Cape Town Pearson Education Limited, 2016, pp. 168-190

Class diagram:
[2]I. Sommerville, “System modelling”, in *Software engineering*, 10th ed. Boston, Mass. Amsterdam Cape Town Pearson Education Limited, 2016, pp. 149-152

Behaviour models:
[3]I. Sommerville, “System modelling”, in *Software engineering*, 10th ed. Boston, Mass. Amsterdam Cape Town Pearson Education Limited, 2016, pp. 154-157

Sequence diagram:
[4]I. Sommerville, “System modelling”, in *Software engineering*, 10th ed. Boston, Mass. Amsterdam Cape Town Pearson Education Limited, 2016, pp. 146-149

[5]I. Sommerville, “System modelling”, in *Software engineering*, 10th ed. Boston, Mass. Amsterdam Cape Town Pearson Education Limited, 2016, pp. 144-145