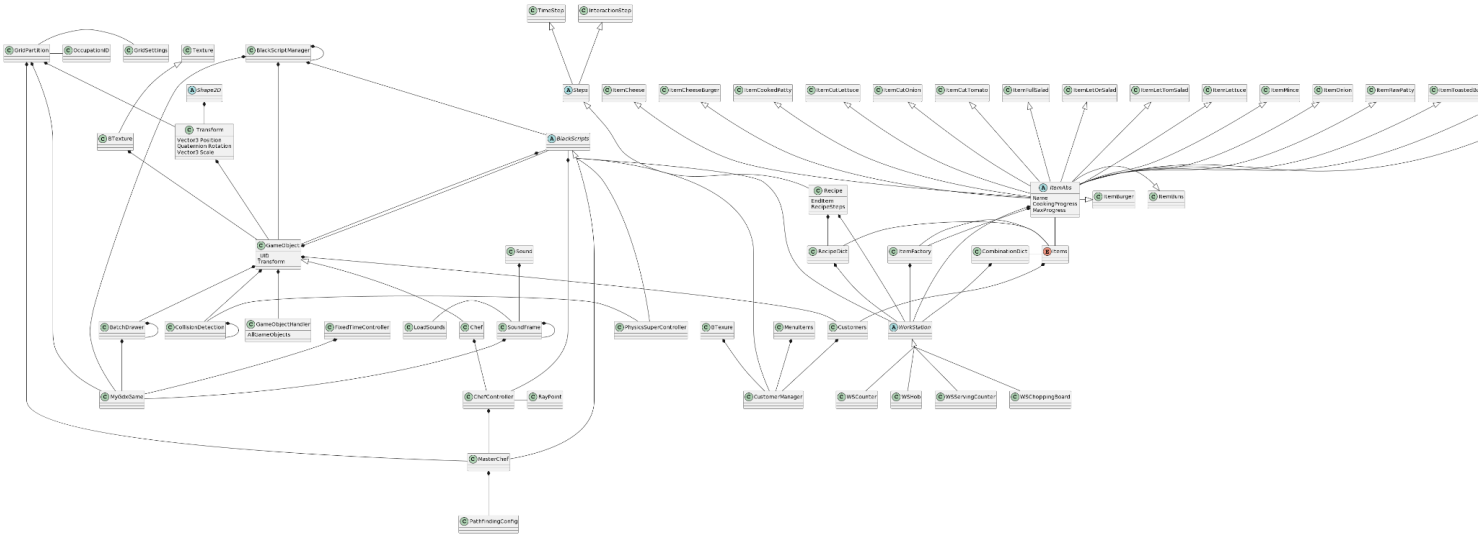
Architecture

Group 27 - BlackCat Studios
Jack Vickers, Hubert Solecki, Jack Hinton, Sam Toner, Felix Seanor, Azzam Amirul Bahri

Structural and behavioural diagrams

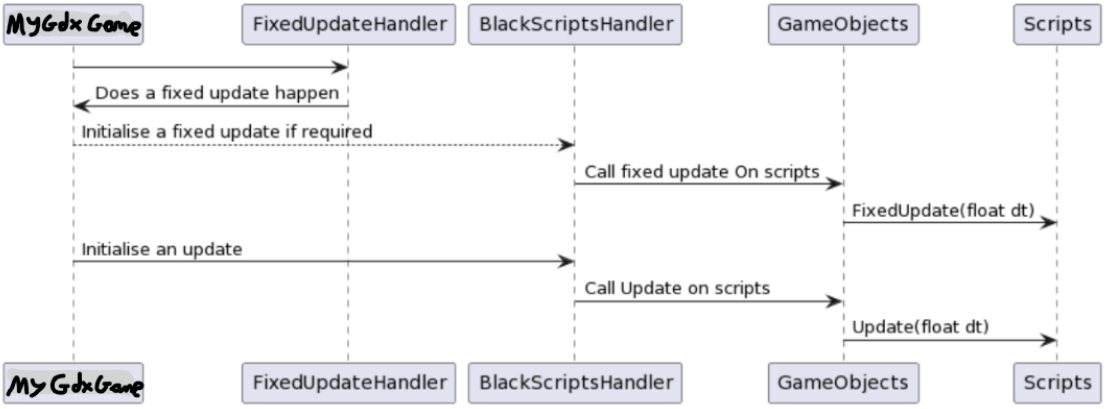
All of the architecture diagrams can be found on the team's website at [Architecture - BlackCatStudios (palindromae.github.io)](). Important diagrams have been included in this document, and others have been linked from the website.

Total UML class diagram:



For a better image see here.

UML sequence diagram for MyGdxGame:



https://palindromae.github.io/BlackCatStudios/img/sequenceDiagram_2.png

Above diagram without every item being shown here.

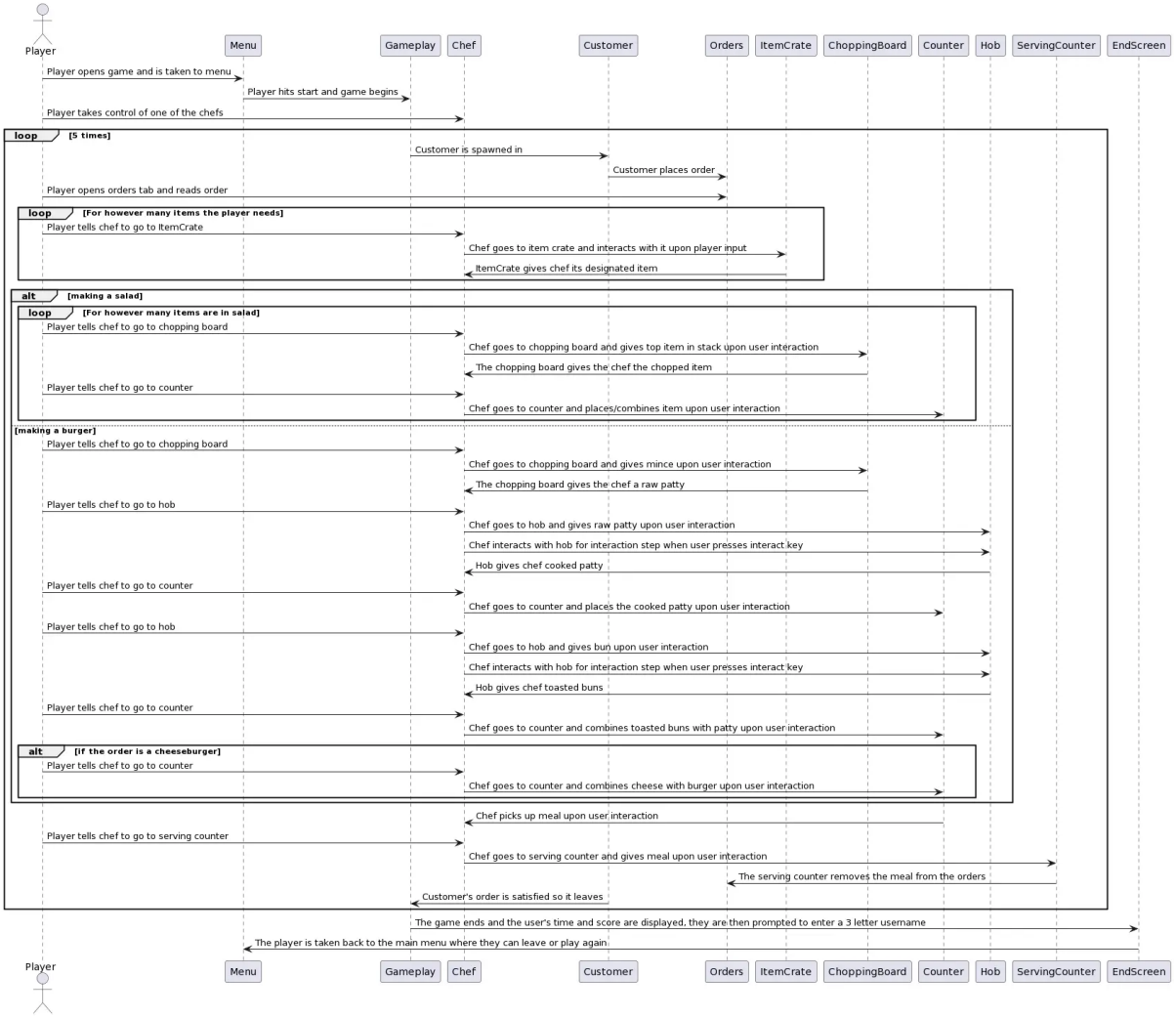
Above diagram with only the core engine of the game here.

UML sequence diagram for GameObject, BatchDrawer here.

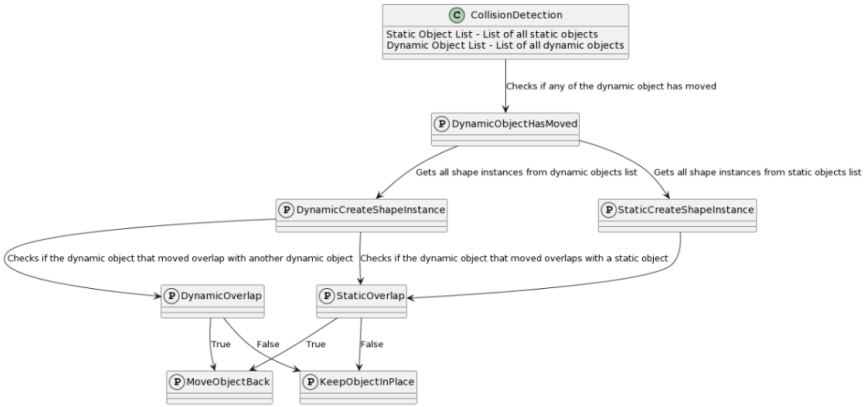
GameObject UML class diagram here.

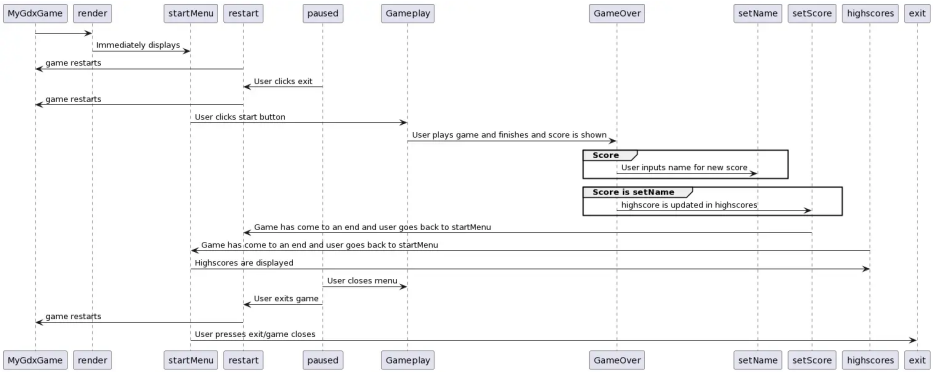
MyGdxGame UML class diagram here.

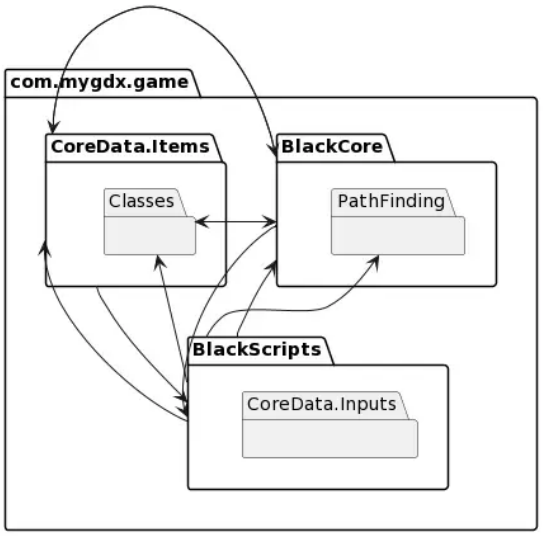
Game loop sequence diagram (larger image on website).



Collision Detection Class/Function Diagram







For further architectural documentation/pictures see our website.

To generate the UML, we used the tool "plantuml.com", this allowed us to easily create a visual representation of how the different classes and functions are connected throughout the program. It provided an easy to use, and intuitive interface, giving a clean and readable UML diagram. In the language provided by plantuml.com, there's a number of different identifiers. In our UML diagrams, we used the class identifier, as well as the protocol identifiers, to clearly state which parts of the UML diagram did what, and what objects belonged to what. PlantUML also allowed us to develop sequence diagrams, cleanly presenting the order of execution of certain aspects of the code base. This allowed us to more easily visualise what was occurring and when.

CRC cards: https://palindromae.github.io/BlackCatStudios/architecture/

To make CRC cards, we used the tool CRC Maker (echeung.me) as it was easy to use, and allowed us to make edits to our cards.

As part of the user requirement NFR_ACCESS_TIME, the game is required to start running within 10 seconds, after testing its starts within 4.5 seconds. Another user requirement, NFR_GAME_LENGTH, requests that the game only takes 5-6 minutes to complete, after playtesting it takes about 400 seconds (6.6 minutes) for a new player to finish the game. We tested our game with someone with red/green colorblindness and he was easily able to distinguish items. Fulfilling NFR_COLOUR_BLINDESS. All other NFRs were visibly satisfied.

Sound

SoundClass:

We wanted the game to have music and sound effects (UR_SOUND) to make it more exciting. And more immersive. All sound files should be easily accessible and easy to play and pause. This was done using the LibGDX Sound object type, stored in a HashMap, allowing access to them via the given name. It should also be as easy as possible to pause, play and stop sounds. This is achieved via the play, pause and stop functions. The volume is also required, via the mute and unmute function (see UR_SETTINGS). Initially, the SetVolume function did not exist, and there was no way to change the volume, however after discussing with the group, we decided a way to change the volume was necessary. Over time we also realised that it was important to be able to stop and start individual sounds. This led to extending libgdx implementation of that.
Soundframe UML class diagram: soundFrame_1

Collision Detection

CollisionDetection Class

Collision Detection is a large part of the game, as it extends from the UR_USER_CONTROLS requirement, providing a recognisable experience, with accurate collision detection, stopping the player walking through walls, or other chefs. Initially, collision detection was achieved by checking if the moving object was inside another object, then moving it away (see collisionDetection_1 and collisionDetection_2), however this caused errors when interacting with smaller objects. This was then changed to using the LibGDX function overlaps(), taking the shapes of the objects and overlapping them. For efficiency, only objects that have moved are checked for collisions. Initially this wasn't the case, and every object was checked against every object, but this was unnecessary. Static and dynamic objects are separated in order to allow the movement of dynamic objects that have collided, while keeping static objects, such as walls, in place (see collisionDetection_3).

PhysicsSuperController

In order to allow collision detection to be universally available, it needs to be a singleton. This singleton is initialised in the PhysicsSuperController class, and is run continuously (on every fixed update) to catch every collision when any object moves.
Focussing further on the requirement UR_USER_CONTROLS, to provide a recognisable experience, the object should detect a collision before the frame updates, this is implemented by running the FixedUpdate faster than the frame rate.

CoreData.Items

Workstations

As part of the gameplay (UR_GAMEPLAY), the game requires workstations for the user to place items on and perform actions. The workstations were designed as an entity-component-system, each one extending an abstract class called Workstation, due to each type having similar variables and base methods, i.e. giveitem/takeitem/deleteitem (see justItemsAndWorkshops.png). However every type of workstation has its own unique functions- so separate classes were made for each. All workstations were initially designed to have a pick up and put down function (FR_PICK_UP_ITEM), however it was later decided there would be a special case for WSCounter where it will combine

items if a certain item is already on there. The deleteItem method was also later added to make it easier for future programmers to understand when the Item variable is reset. Another design choice is that the Cook and Cut methods for WSHob and WSChopboard are able to use the same recipe steps, this is done by having the Cook method be called every update and the Cut method only be called if the user has interacted with it. The Cook method has also been designed to allow programmers to easily implement a burn function at a later date by having it check if cooking. Progress is set back to 0 when checking if the item is ready, if not then it can be set up to go to the burn function. Later this class was extended with the interface Interaction. Allowing for a common interface between objects.

Recipes, Steps & Combinations

As stated in the user requirements (see: UR_GAMEPLAY) players must follow some steps to create a meal, there are two key components to creating the meal, turning one item into another (i.e cooking a raw patty to make a cooked patty) and combining two items. As a result of this the two have been split into two different classes RecipeDict and CombinationDict respectively. In total there are only two steps needed, a TimeStep which is simply the waiting period when something is cooking or being cut and an InteractionStep forcing the player to interact with the workstation, both of these steps extend an abstract class Step; this means lists of steps can be created, as well as making it easier to call the steps in workstation methods. Each Recipe is made up of multiple steps and an endItem variable, initially a JSON file was considered to store the recipes and steps however a HashMap was decided on instead. Each one is stored with the beginning item as the key and the recipe as the value, the design of it means that future programmers can easily add new recipes to the game by creating a Recipe object and putting it in the HashMap with the start item as the key. The CombinationDict class holds a HashMap containing all possible combinations, once again a JSON file was considered instead. Each combination uses the two combined item names concatenated as the key and the resulting item as the value. Each Recipe and Combination has also been kept simple to keep in line with user requirements (see: NFR_RECIPE).

Extended engine

The engine is based on LibGDX using its main functions, however it is extended to support continuous movement using a modular entity component system. An extension to the default batch drawer was created to render characters and the world (see UR_SCREEN_SHOWS_CHARACTERS, UR_SCREEN_SHOWS_BACKGROUND, UR_GAMEPLAY_TYPE, NFR_CAMERA, NFR_CAMERA_ANGLE). Rendering them from an orthographic camera's perspective (see UR_SCREEN_CAMERA). The design was changed slightly later in development to allow for Z-Order rendering. So Objects further up the Z plane would appear behind ones closer.

Each entity has a position, rotation and texture. These entities can also have collision boxes and scripts attached to them. The idea behind this is to have several major features that are easily reusable and understandable. Such as principles of composition to allow for completely reusable scripts. Over the course of the project a FixedUpdate was added into the script execution sequence to allow for discrete time steps, as without this physics would be non-deterministic due to floating point errors. This runs at twice the max frame rate, so that movement appears smooth. Initially it was at half however this was not aesthetically pleasing. Transform was altered to store a reference to the grid partition the entity was on, so when position updates were made it could update its position on the grid. We also modified the Z-Order to prioritise the Y position first, to create a “layering effect”.

Chefs

The set of chef classes have been designed to be as modular and extendable as possible. Allowing more additional chefs to be added quickly and easily. It takes a hierarchical approach, with modules within entities. The class MasterChef was chosen as the top level class giving orders to the chef currently selected to move or to interact. Beneath this is the Chef GameObject holding a Script Module, ChefController, which is able to execute given orders independently. E.g. move to location (X,Y). See (UR_GAMEPLAY). Control is switched between chefs by pressing the number keys 1 and 2. The user can then click somewhere else and the chef will move as close as it can to that point. (See UR_CONTROLS, NFR_COOK_SWITCH, NFR_EASE_OF_USE). An architectural decision was made to force a total order onto the chefs and MasterChef class to force MasterChef to update first and then each chef's FixedUpdate (Renamed to AfterFixedUpdate to allow for this). Another change was to how movement was to be done, removing PIDController as acceleration and velocity wasn't feasible given the physics engine. Direct translation was chosen instead.
A* using a euclidean heuristic was chosen for the navigation system. The step cost is 1 and the heuristic is weighted by 1.5 to ensure a path is found in a reasonable amount of time. As diagonal steps aren't allowed, 1.5 gives a slight advantage towards the greedy searches, as the diagonal cost step is 2, while the euclidean estimate is $1.5 * \sqrt{2} \approx 2.12$.
Over the course of the project it became clear chefs will have to steer around objects. Which led to implementations of functions creating blocked regions.
UML sequence diagram for controlling a chef: sequenceDiagram_3

Start Menu and Pause Menu

We decided to implement start and pause menus (FR_START_GAME, FR_INSTRUCTIONS, UR-PAUSE-MENU, FR_SETTINGS, FR_PAUSE_MENU_BUTTON, UR_GAME-CONTROLS). We decided to expand the menu beyond the scope of a simple play button to give the user more freedom and choice. The user is able to display the highscores before the game as well as observe them during the end game sequence to see how they rank at two separate occasions. We have also introduced an additional path to the settings/pause screen where the user is able to mute/unmute the game and view the controls. This pause/settings menu also allows for more functionality to be added later in the implementation process. We rendered these menus over the game at different depth (y) positions since our game did not feature the screen class before their implementation. Preventing further complexity and to continue the simplistic design and implementation, we decided to use the GameObject class to render these menus. We encountered some issues chronologically with the development of these menus when identifying different game states and identifying which buttons and icons could be used for interaction and when they were displayed. As the menus developed, we decided to integrate logic and further game states such as isGameRunning to check at which state the game was to dictate which icons and menus were displayed or hidden using the negateVisibility, and further simplified methods such as changeMenuVisibility in MyGdxGame to reduce code length and complexity. These allowed us to safely hide and display the menus when necessary. After resolving issues with incorrectly displayed menus and icons, we devised paths and methods for when each icon was clicked or interacted with, allowing the user to transition through the game sequence freely. State diagram for pause: pauseStateDiagram.png

Highscores

We added a leaderboard functionality displaying score, a function related to time to complete orders. To satisfy our requirement, UR_LEADERBOARD, UR_GAMECONTROLS. When the game ends the player is prompted with their score and is asked to enter a name for only three characters because we want it to be classical like the old arcade games. After the name is entered, the name and score of the user is saved in a json file called highscores.json. Then the user is brought back to the start menu and can access the leaderboard to see where they stand within the top 5 scores. The java class HighScores draws the text on the high scores screen and displays the top 5. The java class Save writes the data of the scores and the name of the user to the highscores.json file and reads it back up to the high scores screen.