Engineering 1 Group Assessment 2
Architecture
Cohort 3 Group 23

Team Members:
Harry Draper
Seif Hussein
Tikhon Likhachev
Thomas Maalderink
Joshua McKean
Sebastian Armstrong

<u>Software Architecture</u>

We chose to display the architecture using a UML class diagram. We used plantUML to create the class representations. For the assessment 2 class representations, a plantUML plugin was used to generate the UML class diagrams for ease of use and accuracy

Implementing the requirements into software is preceded with planning and documentation to aid maintainability of the project and make it easier for others to pick up the project and continue development and implementation.

Abstract Representation
For the abstract representation, we chose to create a simplified UML component diagram. Figures 1 and 2 show an abstract representation of the user interface. Figures 1 and 2 have libGDX entities listed only where necessary to understand this project's entities, using ellipses to isolate the details which are most important/helpful to understand the rest of the diagram (Vector2 from libGDX has been titled as `libGDX.Vector2` to disambiguate the two with similar names).

Fig 1, a UML Class Diagram outlining a model for the UI, focusing largely on widgets.

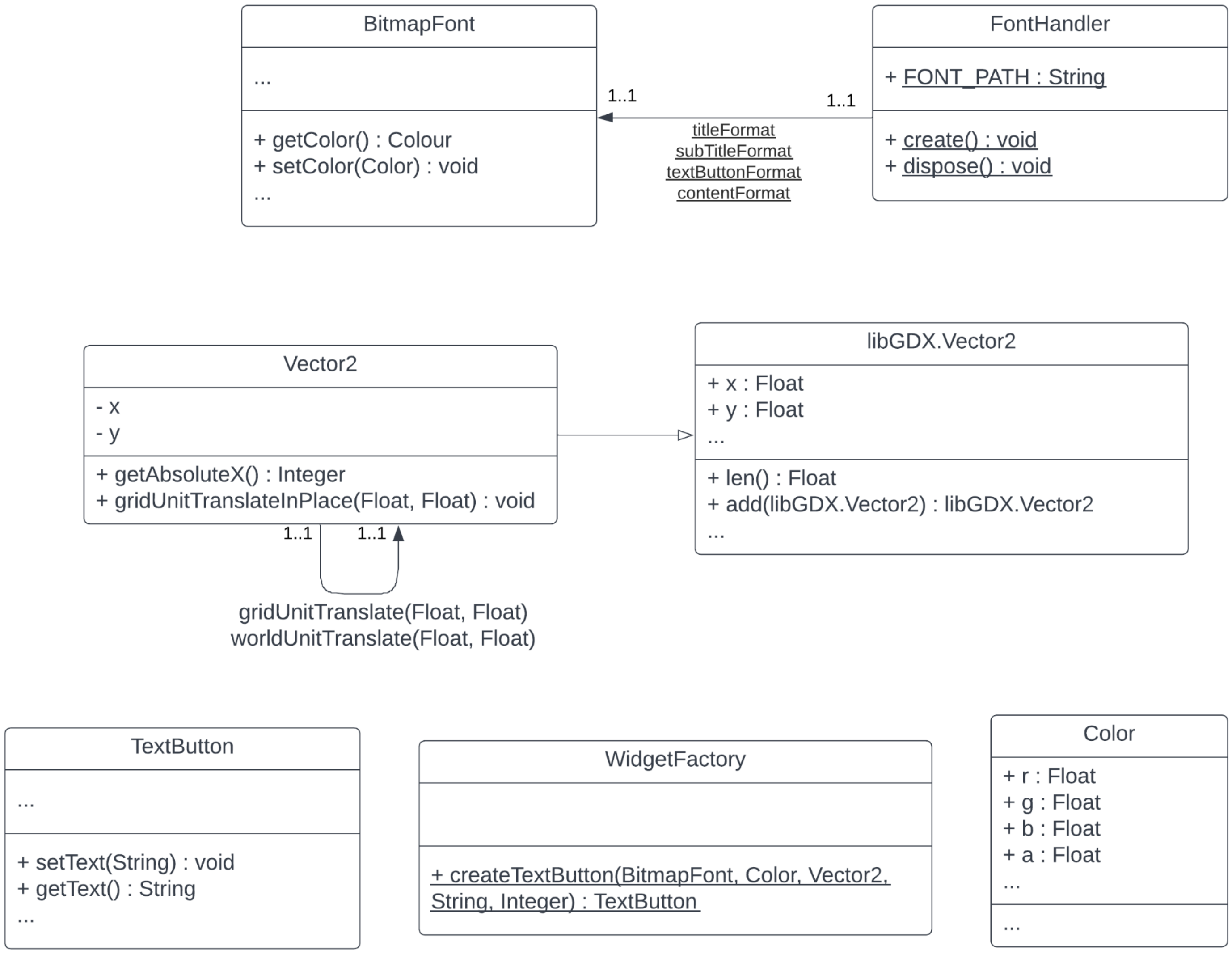
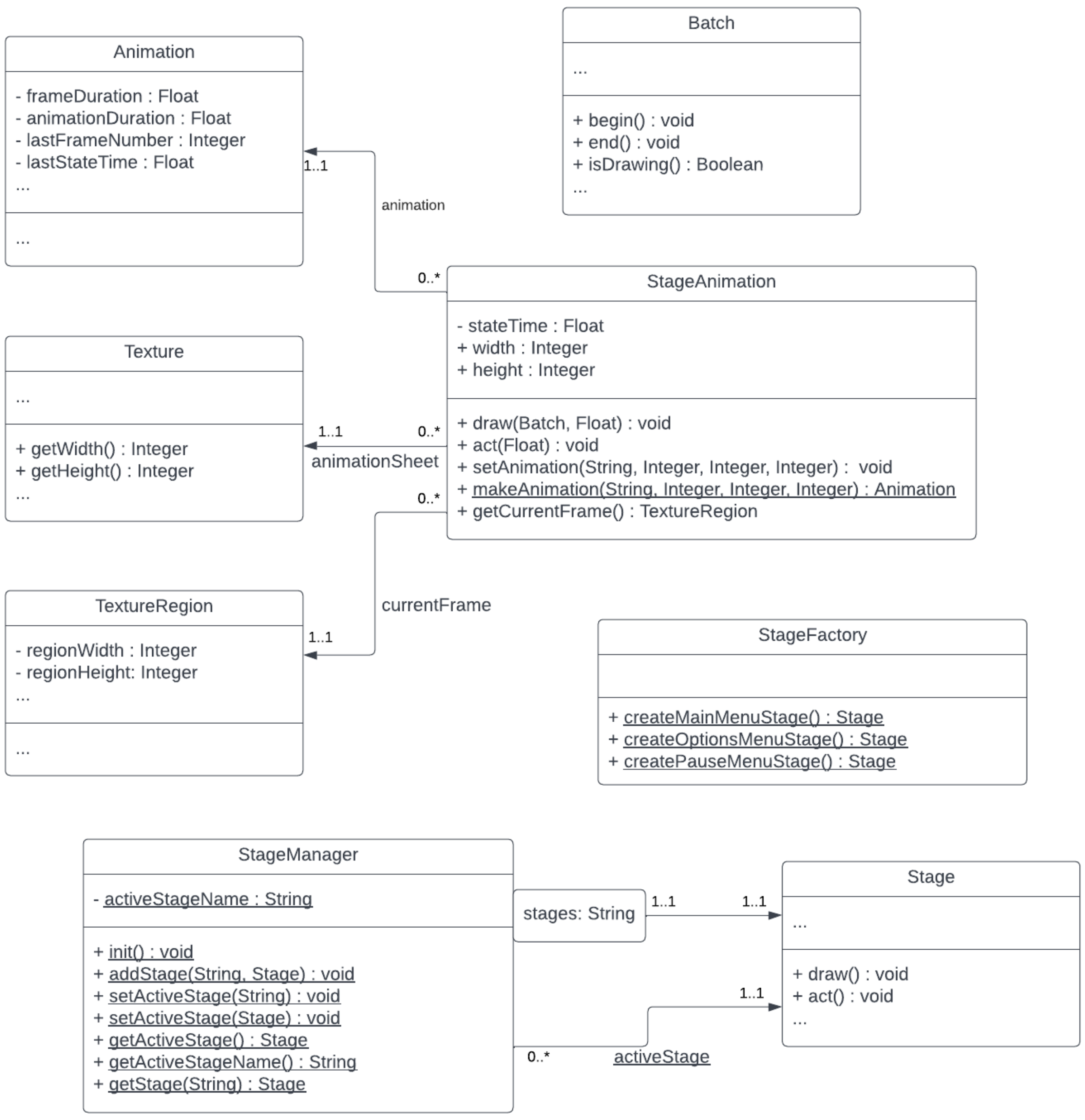


Fig 2, a UML Class Diagram outlining a model for the UI, focusing largely on stage and stage management

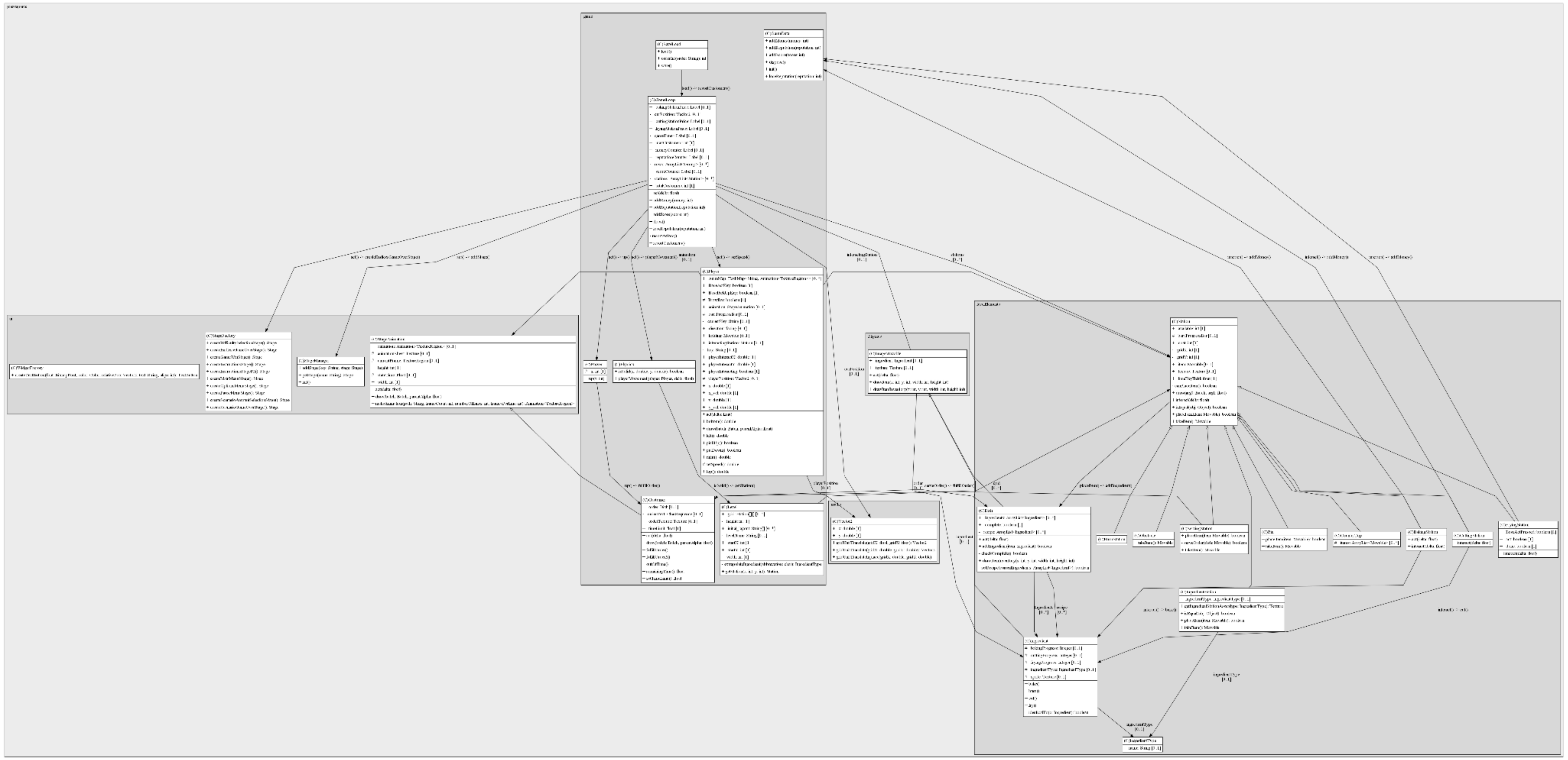

Batch
...
+ begin() : void
+ end() : void
+ isDrawing() : Boolean
...
Animation
- frameDuration : Float
- animationDuration : Float
- lastFrameNumber : Integer
- lastStateTime : Float
...
...
1..1
animation
0..*
StageAnimation
- stateTime : Float
+ width : Integer
+ height : Integer
+ draw(Batch, Float) : void
+ act(Float) : void
+ setAnimation(String, Integer, Integer, Integer) : void
+ makeAnimation(String, Integer, Integer, Integer) : Animation
+ getCurrentFrame() : TextureRegion
Texture
...
+ getWidth() : Integer
+ getHeight() : Integer
...
1..1
0..*
animationSheet
0..*
currentFrame
TextureRegion
- regionWidth : Integer
- regionHeight: Integer
...
...
1..1
StageFactory
+ createMainMenuStage() : Stage
+ createOptionsMenuStage() : Stage
+ createPauseMenuStage() : Stage
StageManager
- activeStageName : String
+ init() : void
+ addStage(String, Stage) : void
+ setActiveStage(String) : void
+ setActiveStage(Stage) : void
+ getActiveStage() : Stage
+ getActiveStageName() : String
+ getStage(String) : Stage
stages: String
1..1
1..1
Stage
...
+ draw() : void
+ act() : void
...
1..1
0..*
activeStage

Concrete Representation

For the concrete representation, we also used a UML diagram created using an IntelliJ plugin, however we created this one with much more detail and accuracy to the game we have created. We have generated multiple class diagrams for each of the main classes within the project as doing a diagram for every class would be excessive and inefficient.

Fig 3, 4, 5, 6, UML Class Diagrams displaying inheritance and other relations between classes working in a level (Open hyperlink on the image to get higher quality image or the link below)
[Project UML Class Diagram]

Project UML Class Diagram



Architecture justification
Abstract architecture justification

This architecture was created and developed during and after the eliciting requirements stage, but before any implementation had been done on the project. It is a basis for the creation of the concrete architecture, this includes basic names and not the actual variables and classes, those are included in the concrete representation further down this document.

We agreed as a team on the basic class structure, we did not name the classes or any variables but we had the layout planned for the structure of the implementation.

Concrete architecture justification

We had to develop the architecture from the abstract representation to the concrete representation. The main difference was that the concrete representation included all the correct names for the classes and variables.

A basic requirement of the project is that it is to be written in Java, and we have decided to use LibGDX for graphics and a game engine. LibGDX was decided upon as a team, due to its wide popularity as a game engine in the Java Ecosystem, which comes with extensive tooling, and a deep level of community support. Additionally, its cross platform compatibility (most importantly among different operating systems including the varying environments available to each developer personally) provides versatility, making it easier for this or another development team to work on the project for different platforms such as mobile phones, if needed.

This project is only developing for desktop platforms like GNU/Linux and Microsoft Windows, which are assumed to have a Java Runtime Environment installed. This is a direct consequence of the user requirements. Similarly there will not be support for controllers or other input methods, even though they could be used on a desktop.

Libraries for writing in different fonts, given a third party TrueType Font file have been included. While there is no explicit need for them defined at this time, there is little overhead to including that particular functionality if left unused and it is highly likely that either this or a subsequent development team may require it, so it’s much simpler to set that now rather than retroactively.

A physics engine was deemed unnecessary because collisions with horizontal and vertical walls/counters is very simple to implement without a physics engine, a physics engine would just complicate things as whole levels will have to be designed and coded from the ground up very specifically for that one physics engine. This is a detriment to maintenance and to others who may want to pick up development of this project.

Similarly, a lighting engine, AI and 3 dimensional graphics/projection are way outside the scope as given by the user requirements. More general tools that aim to help entity or class management (specifically “Ashley”), that make things better laid out, easier to maintain on large scale or more performant.

Finally, `gdx-tools` shall not be included, it has stand-alone functionality like software to pack together images into a sprite sheet like format, providing an image.pack file locating every image in there, with LibGDX able to interpret the regular image (perhaps image.png) when also supplied with the .pack file. The project is opting to operate without that more specialised software as this modifies the typical, intuitive ways of doing things. While that may be beneficial in the long run with more experienced developers and a much bigger project with a longer lifespan, the team has decided to go without.

Justification relating to requirements

Below is a list showing how each requirement was satisfied in the concrete architecture:

UR_COOKS: The abstract class “Chef” contains variables of useStation, move, pickUp, putDown. Allows the chef to move using **UR_MOVE_COOK** and use items and stations. The “Movable” class lets the chef move and pick up and put down items. Also the “Chef” class can be used to help with **UR_WHICH_COOK** and **UR_SWITCH_COOK**. Also allows the chef to use the station they are at currently using **UR_COOK_STATION**.

UR_COOK_INTERACT: The variable in FryingStation, BakingStation, and CuttingStation is interact() which lets the chef interact with the stations.

UR_MODE_SCENARIO: The game mode is scenario, which means that customers arrive one-by-one and it ends when the user has finished all the orders, it displays how long it took to finish.

UR_CUSTOMER_ARRIVAL/UR_CUSTOMER_ORDER: The customers will arrive and it will take note of their order so that the customers can receive the correct orders.

UR_CUTTING_STATION/ UR_BAKING_STATION/ UR_FRYING_STATION/ UR_SERVING_STATION: The abstract class “Station” contains variables to make the stations hold and use the items appropriately according to the station. For example, they all interact with the item but “CuttingStation” cuts the item, “BakingStation” bakes the item, “FryingStation” frys the food item, and “ServingStation” serves the plate of food to the customer.

UR_INGREDIENT_PANTRY/UR_UNLIMITED_INGREDIENT: The abstract class “Ingredient” stores all the ingredients for the chef to use at any point from each ingredient store. The ingredient pantry is unlimited, meaning there is no limit to the number of ingredients that can be used.

UR_TAKE_INGREDIENT/UR_DROP_INGREDIENT/UR_COOK_STACK: In the abstract class “Movable” is the variables of pickUp() and putDown() which allow the chef to take and drop the ingredients and store them in the chefs stack.

UR_COUNTER_WAIT: The customers wait at a specific counter top until they are given their dish.

UR_GAME_END: The game will end when the user has served all the customers the orders.

UR_RECIPE: There is an abstract class called “Recipe” which stores the instructions for the chef to create the dish.

UR_IDLE_COOK: The abstract class “Chef” stores the cook that is not being used by the user at the time, while one cook is being used the other waits until it is switched to be controlled again.

UR_PLAYABLE: LibGDX is used, allowing for mouse and keyboard controls.

UR_HARD_MODE, UR_NORMAL_MODE, UR_EASY_MODE, UR_GAME_DIFFICULTY: The abstract class “StageFactory” stores the difficulty the user selects and changes certain gameplay mechanics.

UR_GAME_SAVE, UR_GAME_LOAD: The abstract class “SaveLoad” stores the game state in a file to be opened again when the user wants to load a game.

UR_EARNING, UR_UNLOCK: The abstract class “GameLoop” stores the money and sets the money needed to unlock the locked cooking stations

UR_POWER_UPS, UR_SCORE_POWER, UR_MONEY_POWER, UR_REPUTATION_POWER, UR_CUSTOMER_POWER, UR_ORDER_POWER: The abstract class “Power” stores the power ups and allows the cook to use them

UR_MODE_SCENARIO, UR_MODE_ENDLESS: The abstract class “StageFactory” allows the user to select which mode to play on and sets that mode for the new game.