
UnderCooked!

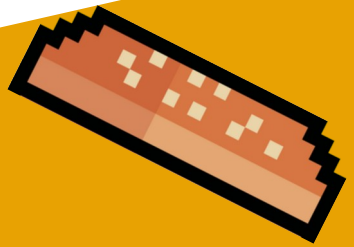
Presentation Outline:

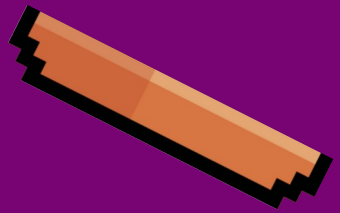
1) The overall quality of the software product.

2) Estimates of effort remaining to complete the implementation

3) Clarity and quality of the requirements specification, architecture and implementation

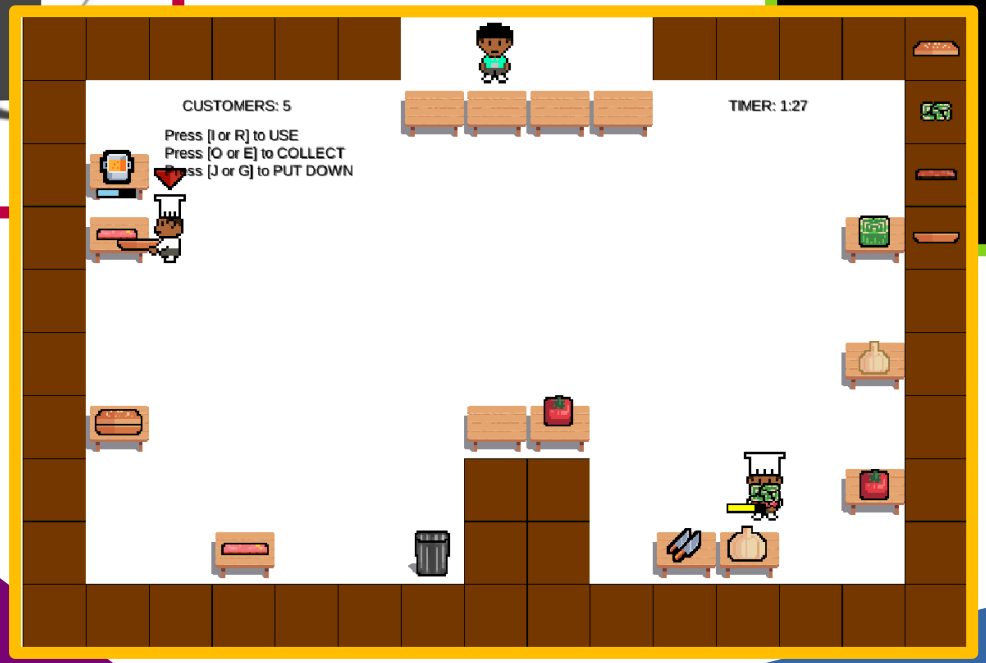
4) Testability

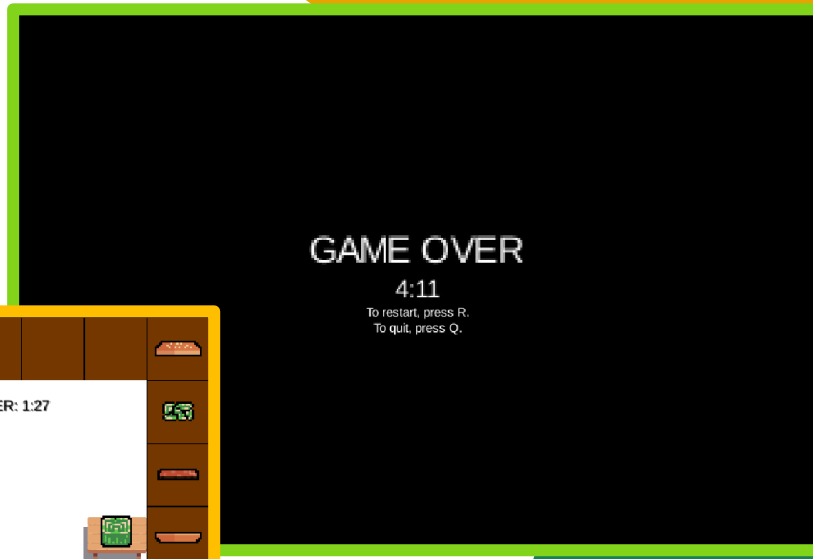




1) Game Demo


UNDERCOOKED
PRESS ENTER TO START
PRESS I FOR INSTRUCTIONS
PRESS C TO VIEW CREDITS
PRESS Q TO QUIT


CUSTOMERS: 5
TIMER: 1:27
Press [I or R] to USE
Press [O or E] to COLLECT
Press [J or G] to PUT DOWN


GAME OVER
4:11
To restart, press R.
To quit, press Q.

1) All of the requirements for Assessment 1 were completed, including Documentation :D

Included Features:

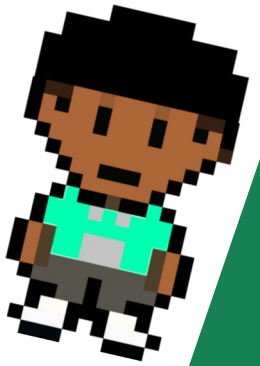
- Instructions has been provided throughout the game
- Players can move and control both chefs (using SpaceBar)
- Chefs can hold a stack of ingredients...
  - ...which they can pick up from pantries
  - ...which can be processed at stations
- Each station type requires players to interact with it in different way to another station, like a minigame, made clear to the user via a progress bar. This is defined in the station's `interact()` function.
- Chefs can request and serve customers dishes. The ingredients required to make each dish is displayed in the HUD.

2) What's required to complete the Assessment 2

Customers arriving at different intervals:

- Our original implementation of customers used a timer, which has been left in the code, which will most likely be used to complete this goal.

```
191         */
192         public void customerServed(ServingStation station) {
193             int customerInd = customers.indexOf(station.getCustomer(),identity: true);
194             if (customerInd < 0) {
195                 return;
196             }
197             removeCustomer(station);
198             customersServed++;
199             gameScreen.setCustomerHud(customersServed);
200
201             // If there are any customers left, spawn a new one.
202             if (customersLeft > 0) {
203                 addCustomer();
204             }
205
206             // BELOW IS CODE FOR CUSTOMER SPAWNING.
207
208             // If there is no more customers on the stations, and
209             // the time for the next customer to arrive is above 2 seconds,
210             // lower the time until the next customer to 2.
211             /*if (customers.size == 0) {
212                 if (TimeUtils.timeSinceMillis(gameScreen.getNextCustomerSecond()) > 2000) {
213                     gameScreen.setNextCustomerSecond(TimeUtils.millis() + 2000);
214                 }
215             }*/
216         }
```



2) What's required to complete the Assessment 2

The 2nd Game-Mode:

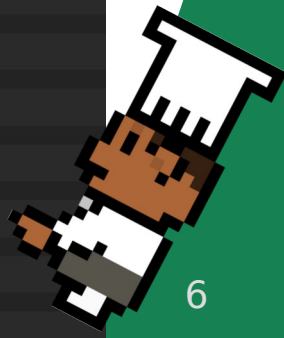
- Though there isn't an explicit 2nd game-mode related code (apart from the timer in the GameHUD), ***CustomerController*** contains all the methods relating to the customers in the game.
- Use this class with a new attribute `GAMEMODE`, to control how the customers behave in the 2nd gamemode in each function.

You can add reputation points into the ***GameHUD*** class, to display it to the player onscreen.

GameHud.java core/src/helper/GameHud.java/{} helper

```
/** Responsible for displaying information above the gameplay GameScreen. */
public class GameHud extends Hud {
    /** The label with the current amount of time played. */
    Label timeLabel;
    /** The label with the number of {@link Customer}s left to serve.  */
    Label CustomerLabel;
    Label CustomerScore;
    /** The {@link SpriteBatch} of the GameHud. Use for drawing {@link food.Recipe}s. */
    private SpriteBatch batch;
    /** The {@link FoodStack} that the {@link GameHud} should render. */
    private FoodStack recipe;
    /** The {@link Customer} to have their request rendered.. */
    private Customer customer;
    // /** The time, in milliseconds, of the last recipe change. */
    // private long lastChange;

    /**…
    public GameHud(SpriteBatch batch, GameScreen gameScreen)…

    /**…
    @Override
    public void render() {…

    /* Removed as it was confusing to look at.…

    /**…
    public void setRecipe(Customer customer) {…

    /**…
    public void updateTime(int secondsPassed) {…
```

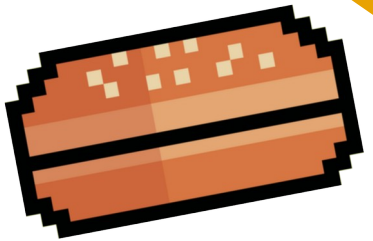
```
public class CustomerController {

    /** An {@link Array} of {@link Customer}s currently waiting. */
    private Array<Customer> customers;
    /** The {@link Sprite} of the {@link Customer}. */
    private static Sprite customerSprite;
    /** An array of all {@link ServingStation}s to assign to the {@link Customer}s.*/
    private static Array<ServingStation> servingStations;
    /** The number of {@link Customer}s to spawn. */
    private int customersLeft,
    /** The number of {@link Customer}s served. */
    customersServed;
    /** The {@link game.GameScreen} to send the {@link #customersServed} to. */
    private GameScreen gameScreen;
      OwenTho, last week • Worked on the Customers.
    /**…
    public CustomerController(GameScreen gameScreen) {…

    /**…
    public boolean canAddCustomer() {…

    /**…
    public int addCustomer() {…

    /**…
    public void removeCustomer(ServingStation station) {…

    /**…
    public void setCustomersLeft(int customersLeft) {…

    /**…
    public int getCustomersLeft() {…

    /**…
    public void setCustomersServed(int customersServed) {…

    /**…
    public int getCustomersServed() {…

    /**…
    public void addServingStation(ServingStation station) {…

    /**…
    public Array<ServingStation> getServingStations() {…
```



2) What's required to complete the Assessment 2

Adding and Improving Recipe making:

- Each recipe is simply defined as a Stack of Food (***FoodStack.java***). Customers can request these defined recipes.
- We have dicts containing important game information.
  - dict<***strRecipeName***, ***LibGDX.Array***<***strFoodStack***>> ***recipes***
  - dict<(FoodID, StationID), (FoodID, ***PrepStationInfo****)> ***interactions***



- *We realised during development, the stations which process ingredients into other ingredients are *very similar*. Thus, a single class ***PreperationStation.java*** is used for **both** the **frying** and **cutting** station, meaning only ***interactions*** dict differentiates them! Meaning...
  - Use ***PrepStationInfo*** to control the progress bar for each interaction, and what comes out of each interaction

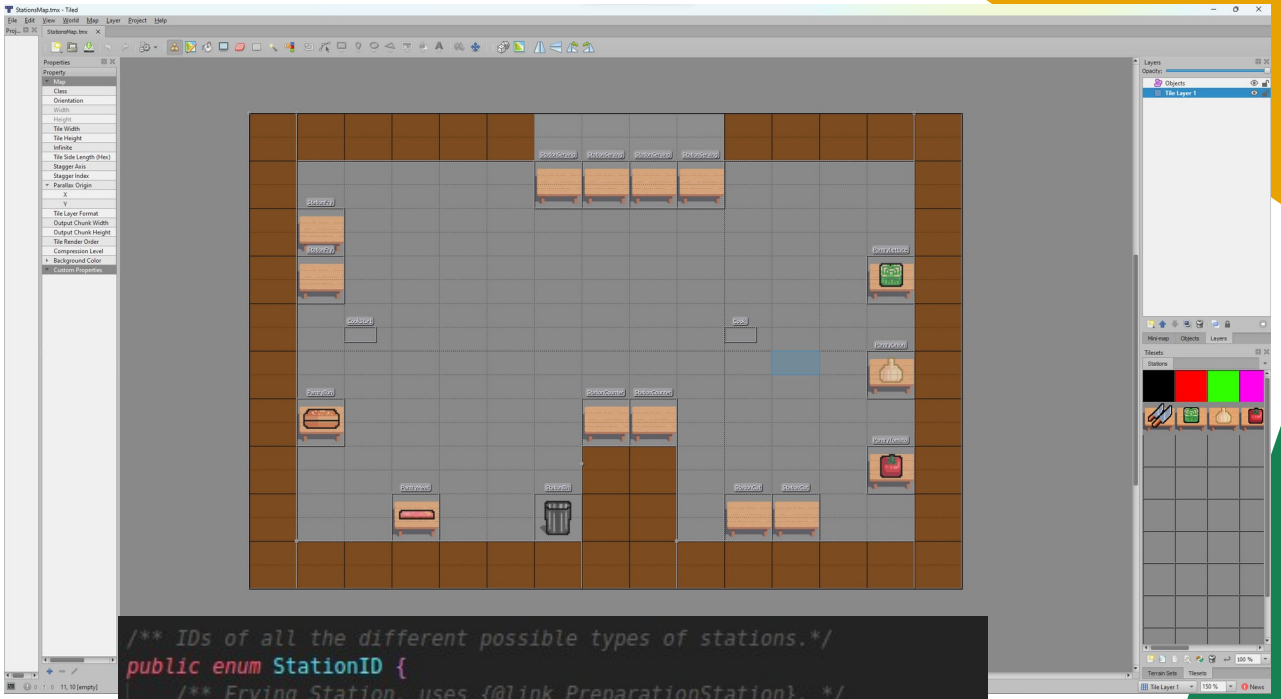
```
/** A HashMap containing how each FoodItem's FoodID, via a station of StationID, can convert to another foodID.*/
private static final HashMap<String, InteractionResult> interactions = new HashMap<>();
static {
    interactions.put(InteractionKey(FoodID.lettuce, StationID.cut), new InteractionResult(FoodID.lettuceChop,new float[] {25,50,75},-1));
    interactions.put(InteractionKey(FoodID.tomato, StationID.cut), new InteractionResult(FoodID.tomatoChop,new float[] {25,50,75},-1));
    interactions.put(InteractionKey(FoodID.onion, StationID.cut), new InteractionResult(FoodID.onionChop,new float[] {25,50,75},-1));
    interactions.put(InteractionKey(FoodID.meat, StationID.fry), new InteractionResult(FoodID.meatCook,new float[] {50},speed: 13F));
}
```

2) What's required to complete the Assessment 2

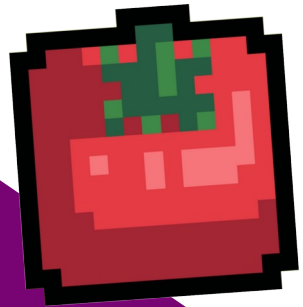
Adding Stations:

- We used a 3rd party application ***TileMap*** to create the map of the game:
- So to add a station:

  1) Create a new station tile in ***TileMap***
- 2) Add a new StationID into the StationID enum.



```
/** IDs of all the different possible types of stations.*/
public enum StationID {
    /** Frying Station, uses {@link PreparationStation}. */
    fry,
    /** Cutting Station, uses {@link PreparationStation}. */
    cut,
    /** {@link CounterStation} that can hold {@link food.FoodItem}s. */
    counter,
    /** {@link BinStation} that can dispose of {@link food.FoodItem}s.*
    bin,
    /** {@link ServingStation} that allows {@link Cook}s to serve {@lir
    serving,
    /** Default Station that does nothing. */
    none
}
```



2) What's required to complete the Assessment 2

Adding Stations:

- 3) Add a new entry into MapHelper:
  - Entries are will most likely be a variation of what's already there (shown on the right)
- 4) Add a new entry into ***interactions***! That's it for simple stations. Very flexible for how simple it is! Adding ***Pantries*** is very similar to adding stations.
- Possibly, 5)
  - If the station requires new functionalities: you may also create a ***new station class*** and override ***interact()*** with any desired code:

```
public class Bin
    /**…
    public BinSt
        super(re
    }

    /**…
    @Override
    public void interact(Cook cook, InputKey.InputTypes inputType) {
        // Only bin if user inputs USE or PUT DOWN
        if (inputType == InputKey.InputTypes.USE || inputType == InputKey.InputTypes.PUT_DOWN) {
            cook.foodStack.popStack();
        }
    }
```

`void stations.BinStation.interact(Cook cook, InputTypes inputType)`

The interact function for the BinStation. This takes the top item from the Cook's food.FoodStack if they use either the InputKey.InputTypes.USE or InputKey.InputTypes.PUT_DOWN keys.

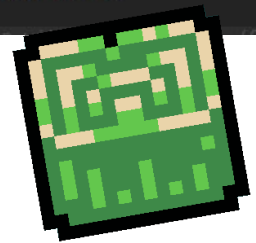
- **Parameters:**
  - **cook** The cook that interacted with the CookInteractable.
  - **inputType** The type of InputKey.InputTypes the player made with the CookInteractable.

```
if(rectangleName.startsWith("Station")) {
    // Stations
    rectangleName = rectangleName.substring("Station".length()).toLowerCase();
    Station station;
    switch(rectangleName) {
        case "cut":
            station = new PreparationStation(rectangle);
            station.setID(Station.StationID.cut);
            gameScreen.addGameEntity(station);
            break;
        case "fry":
            station = new PreparationStation(rectangle);
            station.setID(Station.StationID.fry);
            gameScreen.
            break;
        case "counter":
            station = n
            station.set
            gameScreen.addGameEntity(station);
            break;
        case "bin":
            station = new BinStation(rectangle);
            station.setID(Station.StationID.bin);
            break;
        case "serving":
            station = new ServingStation(rectangle);
            station.setID(Station.StationID.serving);
            gameScreen.addGameEntity(station);
            gameScreen.addServingStation((ServingStation) station);
            ((ServingStation) station).setGameScreen(gameScreen);
            break;
        default:
            station = new Station(rectangle);
            station.setID(Station.StationID.none);
            break;
    }
    gameScreen.addInteractable(station);
}
```

`void game.GameScreen.addGameEntity(GameEntity entity)`

Adds a game entity to the GameScreen to be rendered and updated.

- **Parameters:**
  - **entity** The GameEntity to be added.



3-4) Clarity and quality of the requirements specification, architecture and implementation and Testing.

Requirements:

- All requirements are arranged into a table of ***requirementIDs***, description and parent requirements.
- These ***requirementIDs*** are referenced throughout the documentation, making it simple to produce concise documentation.

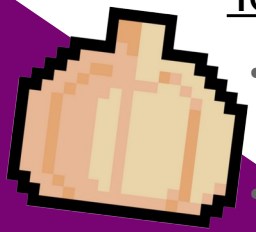
Architecture:

- We described our initial ideas in a ***Class Diagram*** and ***Use-case UML***, then described what classes/changes were made to the initial design, with additional Class UMLs too.

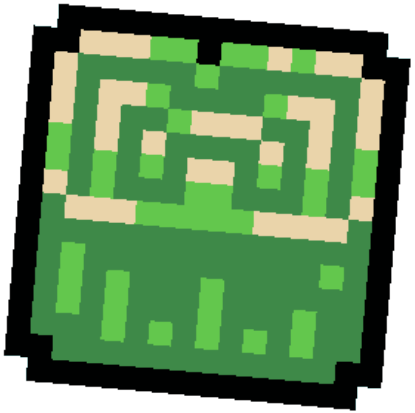
Implementation:

- We designated time and had multiple people to *commenting* the code, so that it is ***fully commented***, in ***proper Javadoc*** style. Simply ***hover over any method or attribute*** to get a description of it.

Testing:



- We have setup ***GitHub Actions*** to carry out ***gradle build tests*** on every commit, so errors will be detected quickly.
- The website also has ***GitHub Actions*** checks, as the website is a GitHub-Page.



Thank you for listening :D