Artitecture

Designer story

We are developing a casual cooking management game designed to showcase the potential work done in the computer science degree. This will be shown to people interested in studying computer science at York during open days. The game will run locally as a desktop application.

When the game starts, the user will be taken to the 'Main Menu' and presented with a number of options, one of which will start a timed game.

Once the mode starts, the user will be taken to the 'gameplay' screen where customers enter the building and the user has to control 3 chefs in order to prepare and deliver the correct order to the customers. The user has 10 minutes to correctly serve as many customers as possible in order to accumulate the highest possible score. After the game ends, the end score is displayed to the user.

In the main menu, if no user inputs are detected for 30 seconds, a demo of the game will automatically play until an input is detected.

Themes

- Desktop interface
- Comprehensible layout for the 'customers section' and the 'chefs section'
- Ongoing preparation where the chefs create the dish
- Top-down design with the character being able to move in any direction, 360 degrees
- Time-dependant customers

    - No ‘win condition’ as the game will end by default after 10 minutes
    - An idle phase where the game displays a demo after 30 seconds of no input detection on the main menu

Stereotypes

We have outlined stereotypes; this was done in a group session where we used CRC cards. This was to get us to brainstorm and find classes, which represent a collection of similar objects, responsibilities, and know which action each class should perform.

Chef: This is a class that is controlled by the player. It can move around the level, pick up and place ingredients. Also it is possible to switch between the different chefs.
Role: Coordinator

Customer: This is a class to keep track of a customer. They arrive within a given time interval or no customer remains then they order a recipe and then wait until the player completes an order. If the order is incorrect or takes too long, their satisfaction score will decrease.
Role: Information holder

Dish: This class contains a list of current ingredients that the cooks have already prepared. Once order is delivered it is check that all ingredients are present if they are customer will level, otherwise the player will have to remake the order
Role: Structure

Ingredients: This class is a general class for all ingredients, it stores what the ingredient is and its location. Also stores if it's flippable or choppable, and its current state.
Role: information holder

Level: this stores certain properties of the level. Such as customers, cooks, the level timer and gamemode.
Role: Structure

Station: This class stores the station name and allows checking if the ingredient is allowed to be placed.
Role: Information holder

Textual Use Case

Here we have outlined how we want the system to work at various stages in the game. This is important in the design process as it allows us as software developers to see how the user will interact with the system and how the customer wants the system to react.

<u>Scenario: Preparing Ingredients (chopping:onion lettuce and tomato, forming:patty)</u>
• Primary Actor: Chef(s)
• Precondition: chef has picked up a vegetable and placed it on the chopping board
• Trigger the chef interacts with the vegetable thats placed on the chopping board

• Main Success Scenario
  – 1. Chef picks up the vegetable
  – 2. Chef place the vegetable on the chopping board
  – 3. Chef interacts with the vegetable places on the board
• Secondary Scenarios
  – 1.1. Chefs bins the vegetable
• Success Postcondition: Chef has interacted with the vegetable placed on the chopping board turning it from vegetable to chopped vegetable

<u>Scenario: Cooking (patty and buns)</u>
• Primary Actor: Chef(s)
• Supporting Actors: Customer
• Precondition: chef has picked up the item to be cooked
• Trigger: Ingredient is placed into a fryer
• Main Success Scenario
  – 1. Chef picks up ingredient
  – 2. Chef places item into frying pan
• Secondary Scenarios
  – 1.1. Chefs Burns the Ingredient
• Success Postcondition: Chef has cooked ingredient
• Minimal Postcondition: Chef has burnt the ingredient

<u>Scenario: Assembling Dish</u>
• Primary Actor: Chef
• Precondition: all ingredients must be correctly prepared
• Trigger: when all the correct ingredients are placed on a plate together
• Main Success Scenario
  – 1. Chefs picks up prepped ingredient
  – 2. Chef places it onto a plate
  – 3. This is repeated until all ingredients have been put on the plate
• Secondary Scenarios
  – 1.1. Chef adds the wrong combination of ingredients
    - 1.1.1 the dish must be binned
• Success Postcondition: the dish is created when all ingredients are assembled together
• Minimal Postcondition: wrong dish assembled

<u>Scenario: Delivering Dish</u>
• Primary Actor: Chef
• Supporting Actors: Customer
• Precondition:Chef has the correct meal prepared for the customer
• Trigger: Chef interacts with the customer while holding the dish
• Main Success Scenario
  – 1. Customer asks for dish
  – 2. Chef prepares the dish
  – 3. Chef gives the dish to the customer
• Secondary Scenarios
  – 1.1. Chef gives the customer the wrong dish
    - 1.1.1 reduced points

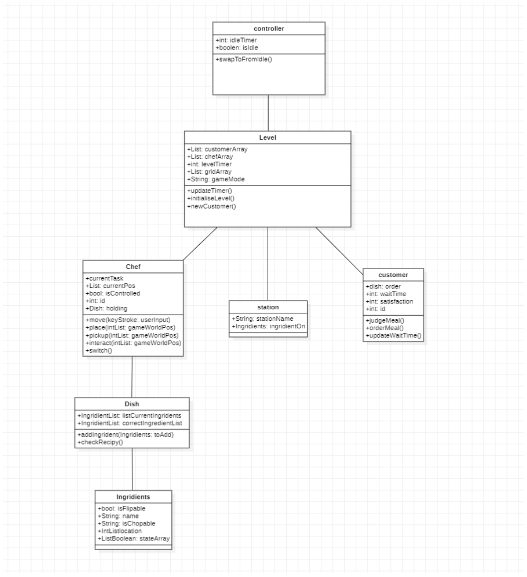
– 2.1. Timer runs out and customer leaves empty handed
  - 2.1.1 the level is failed
• Success Postcondition: Customer receives dish
• Minimal Postcondition: Customer doesn't receive dish and leaves

<u>Scenario: Completing Scenario</u>
• Primary Actor: Chef(s)
• Supporting Actors: Customers
• Precondition: Chef has cooked food for all the customers
• Trigger: All customers asking for food have received food
• Main Success Scenario
– 1. Chefs cook food when customers ask
– 2. Chefs plate and deliver food to customers
• Secondary Scenarios
– 1.1. The customers wait too long and leave
  - 1.1.1 the level is failed
• Success Postcondition: Customers receive food, ending the game with a victory
• Minimal Postcondition: The customers wait too long and leave, ending the game with a fail

<u>Scenario:Scoring Points</u>
• Primary Actor:Chef
• Precondition: chef cooks and delivers food efficiently
• Trigger:chef delivers food
• Main Success Scenario
– 1. Chef cooks ingredients and assembles dish
– 2. Chef delivers dish to customer quickly
  - 2.1.1 this maximises points as the dish is delivered quickly and there are no mistakes
• Secondary Scenarios
– 1.1. Chefs delivers the wrong dish one or two times then the correct dish is given
  - 1.1.1 the customer gets their food and you receive few points for the mistakes
– 2.1. Chef delivers the wrong item 3 times
  - 2.1.1 the customer leaves scoring the user 0 points and failing the level
• Success Postcondition: user receives maximum amount of points
• Minimal Postcondition: user receives few points

<u>Scenario:Idle demo</u>
• Primary Actor: the game itself
• Precondition: no user input
• Trigger: no user input for 30 secs
• Main Success Scenario
– 1. There is no user input for 30 seconds
  - 1.1.1 the game starts playing itself as a demo
• Secondary Scenarios
– 1.1. There is a user input
  - 1.1.1 the timer for the demo restarts
• Success Postcondition: the idle demo plays
• Minimal Postcondition: there is user input resetting the timer to play the idle demo

Initial Structure

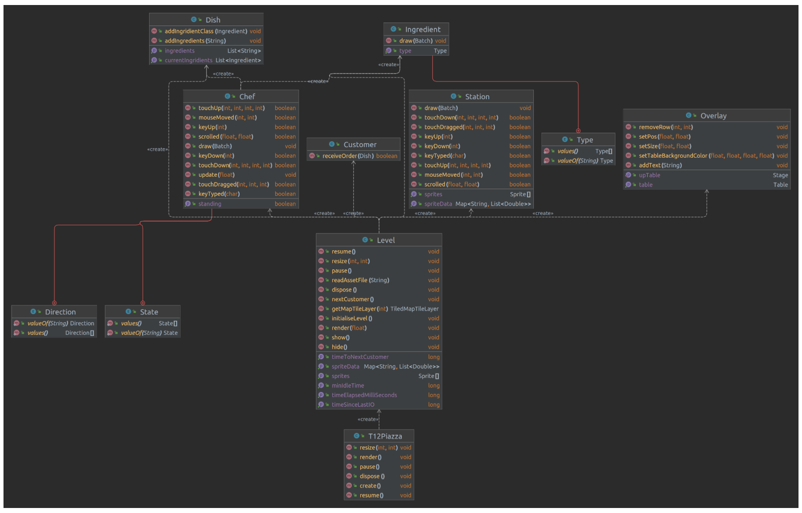


Are initial class diagrams created to get a general overview of the program. We have decided to use an entity relationship model as this is most suited for game development as we can implement different components independently of each other. It also allows us to make changes to one specific component without any dependency issues. The classes are based on stereotypes, we got from requirements, textual use cases and designer stories. These will have to be revised as we implement and come across problems and limitations of libGDX. This overview of the structure allows us to see what components need to be implemented first, but also ensure that we meet the requirements given to us by the stakeholder. Generating that the stakeholder will be happy with the final product as we met all their requirements. Additionally if the requirements change or added too, we can easily see where to implement this within the code, without it being too costly to the customer.

We have also included a sequence diagram so it is trivial to see how the different components will interact with each other. This will make it clearer and easier to understand for other programmers taking on our project at a later stage. Additionally it provides us with an overview to make implementation as easy as possible, for example that the user will never move an ingredient directly as this has to be all done though controlling the chef.

Furthermore we have made a behaviour diagram, this is to show the ideal flow of the program, from building the level, to when customers need to arrive and what happens after they receive their order ect. This is important as when we have fully implemented the game we can see if it behaviours as intended by referring back to this diagram

Revised Structure



Also on their website there are revised class diagrams using iDEA integrated UML class diagrams that allows us to easily see what we have already implemented and anything else that still needs to be done.

We can clearly see that there are some changes from the original structure however this is a part of the spiral method and to be expected. We have however kept class names the same, this is to make it clear to see which component satisfies which requirements. This is essential in software development as it provides a link between the abstract and the actual code.

In the diagram we haven’t included the libGDX classes this is because it makes the diagram more readable. However these classes are required for the implementation as it makes use of previously defined interfaces that can be repurposed for are needs. In other words if we were to implement everything from the ground up this would lead to more work and less well thought out code. However as libGDX provides lots of functionality we can just inherit from its classes and implement functionality as and when needed.

The connection between the revised structure and the initial structure is very important and should be maintained as we based the initial structure on the requirements which were given to us by the customer. Failure to link these 2 structures could result in missing functionality and an unhappy customer. So because of this it is important to link the revised structure back to the requirements to see which are satisfied and if we have missed any that still need implementing. Also this allows us to show the customer progress as we can tick off each requirement as it is satisfied.

Justification for revised structure

Here we have outlined the different classes used in a program to justify their existence by linking them to the requirements that they satisfy.

T12Piazza Class: FR_GAMEMODE_2, UR_GAMEMODES, UR_ACCESSIBILITY

Chef Class: FR_COOKS_MOVMENT, FR_COOKSWITCH, FR_COOKS_FLIP, FR_COOKS_CHOP, FR_COOKS_GRAB/PLACE

Level Class: FR_GAMEMODE_1, FR_IDLE_MODE

Station Class: FR_COOKING_STATION_1, FR_COOKING_STATION_2, FR_COOKING_STATION_3, FR_SERVING_STATION_1, FR_INGRIDENT_STATION, FR_COUNTER, UR_STATION

Customer Class: FR_RECIPE_DISPLAY, FR_CUSTOMER_WAIT, FR_CUSTOMER_DEMAND, FR_RECIPE_DISPLAY, UR_CUSTOMERS

Ingredient Class: FR_INGRIDENT_STATION, FR_RECIPE_1, FR_RECIPE_2, FR_COOKS_FLIP, FR_COOKS_CHOP, FR_COOKS_GRAB/PLACE

Dish Class: FR_RECIPE_1, FR_RECIPE_2, FR_CUSTOMER_DEMAND, UR_RECIPES

Overlay Class: FR_MONEY, FR_TIMER, FR_RECIPE_DISPLAY, FR_COUNTER