EECS3311 Section M

Team 14

Table of Contents

Byte Sized Documentation **2**

Packages 2

Source Code File 2

Library 2

Class Design **3**

Pizza Class 3

Constructor and attributes data type 3

The birth of the Item class 4

User Class 4

Classes UMLs 5

Ordering Class (Graphical User Interface) 7

Design Choice 8

Refactoring Changes 8

Class UMLs 9

Registration and Main Class 10

Design Choice 10

Testing **10**

Pizza Unite tests 10

User Unite tests 11

Byte Sized Documentation

Packages

The packages are split between guiMain and client-server. The largest concern was splitting the front-end and back-end. Furthermore, with the amount developed in the first iteration, we did not make the conclusion to further segregate the packages because the division in roles was not apparent enough. Additional division, for now as the future would change how the packages are layed out, would otherwise present unnecessary specialisation.

Source Code File

Activation of the Byte-Sized GUI is done through ByteSized/src/gitMain/Main.java . Executing the class file through an IDE will launch the interface and begin user interaction. There is a .jar file in the ByteSized folder that is also accessible to execute and run the program without an IDE.

Library

There should only be one file in the library which helps to compile the GUI. Classpathing has already been reconfigured so that the pathing would readjust according to the folder location. As such, there is only one dependency which is, as mentioned, already configured and installed.

Class Design

Mustafa

Pizza Class

The main component of our system is the pizza class. Customers should be able to place their pizza orders and customise the ingredients according to their needs. The main block of the order is the pizza class, which holds all the info a restaurant needs to make the pizza.

I divided the pizza into 4 vital parts. The crust, the sauces, the toppings, and the size. These properties are held as attributes inside the class and get initialised when the customer places their order. For this iteration, the data (ingredients) are placed in a dummy json file. The client uses these data to insert strings into the constructor of the class, creating the desired pizza. If these attributes fail to contain every customization the customer might need, the class also has a special instructions attribute for that. Customers can write whatever they would like the restaurant to keep in mind while making their pizza.

Constructor and attributes data type

Initially, the constructor was made to accept Strings for all ingredients. The UI would send these strings to the class which would in turn pass them to the constructor to initiate the attributes. The toppings and sauces attributes were made to store an array of Strings, each representing a sauce or a topping. No restrictions were made to limit the number of options the user can choose from the offered ones. The customer can also choose to have one base sauce and one topping, making the attributes store an array of only one value. I chose an array in this case as I thought it both allowed the ability of having many while handling the option of choosing only one. If no sauce or topping was chosen tho, the customer will not be able to create their pizza as the constructor does not initialize a pizza without one. For future iterations this option should be discussed and the restriction should be dealt with according to the client's desire.

The getter methods of this class allows the ui to fetch the info of each attribute independently. If an edit of ingredients is required after initialising the pizza, the setter methods can be used to edit the selection of items and set the attributes accordingly again. One such method is also provided for each attribute independently.

At this stage, the class stored the items as primitive data types. Little to no control was given over the type of the items. As the available options were meant to be used from a json object of strings, this seemed sufficient, but for the sake of future development and customization, I decided to change that.

The birth of the Item class

For the sake of following a better practice, and in compliance with the Design Principles of object oriented programming, The pizza class needed to enforce both the single responsibility principle (SRP) and the dependency inversion principle (DIP). The pizza class had to do only one job which is setting the pizza. To enforce SRP the class had to be simple and straightforward. However, I could anticipate that more work needed to be done on the Items themselves and some further customization might be requested. Since these items make up the pizza class and affect the class structure, I needed to make sure that any changes to these items would not interfere with the structure and role of the pizza class. The items and the pizza needed to be decoupled for robustness and safely during the development process.

The item class replaced the primitive String type. This simplistic class now initializes itself by passing a String of the desired name to its constructor. The toppings and sauces attributes now no longer held an array of Strings but instead an array of Item type. We can later choose to change the Item class and introduce new attributes (type of item, id, special info, etc) without jeopardising the pizza class structure. This decoupling at the very least simplifies any future debugging processes and isolates tasks and environments in their dedicated classes.

Eventually, it became clear that in order to print a receipt that describes the pizza in detail, we would have to find a way of presenting the toppings on the narrow rectangular screen. Since toppings are Items, we had to modify the Item class and give it a toString() method. Another thing that became clear while developing the app was how deeply intertwined an item and its price were. Since items like sauces and toppings are ultimately commodities destined to be sold to a customer, it was extremely valuable to have its price as easily accessible as possible. That’s why we decided that a name was not enough, price would also be a parameter in the class.

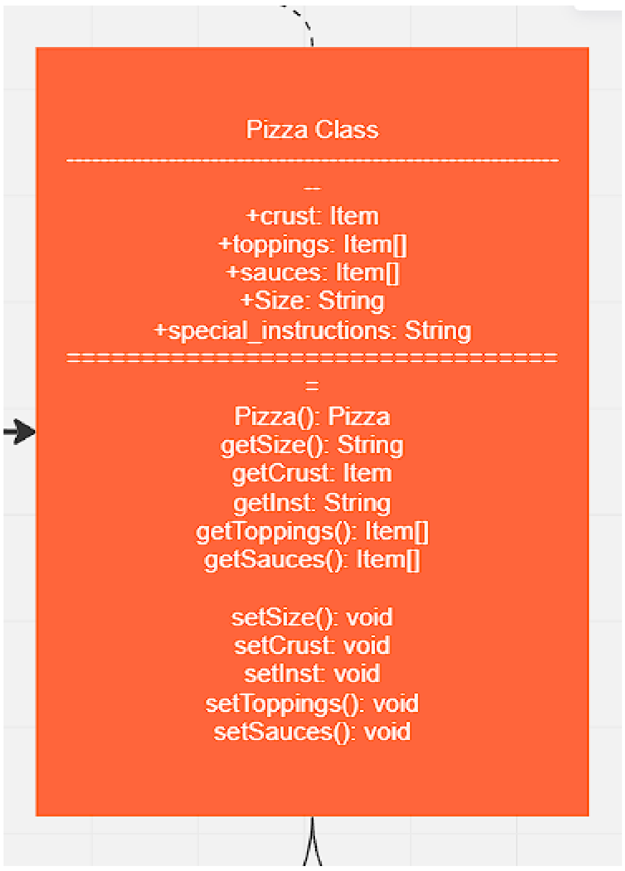
When the toString() method of Item was rewritten, we needed to add a new method to the pizza class to maintain old functionality. Get getToppingnames() was created to allow us to get a String array that could then be printed on the receipt.

User Class

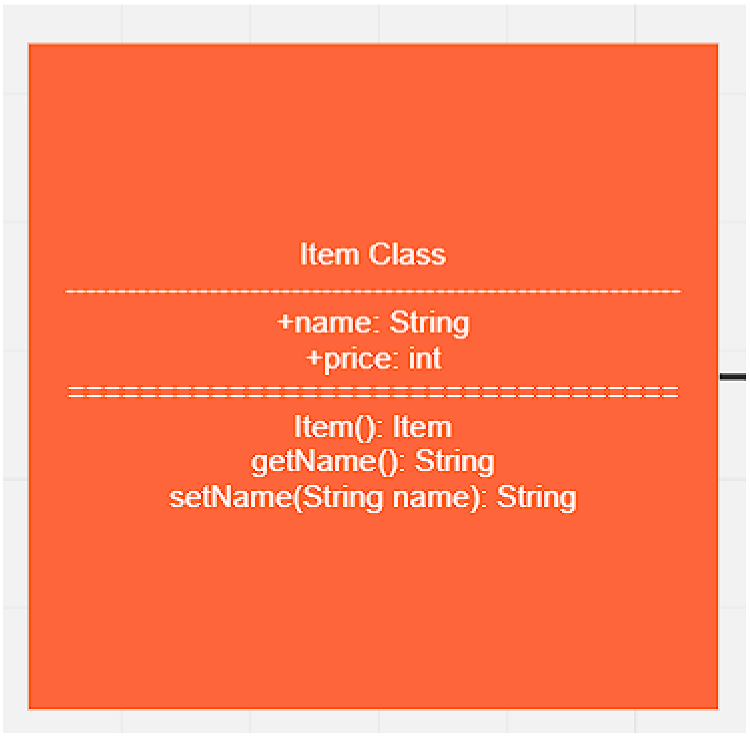
The User class stores the objects of the client users. It has private username, password, role, and isLoggedIn properties. All these properties should only be changed from the class itself. isLoggedIn is a boolean property that indicates whether the user is logged in or not. By default, that value is set to false and the setLoggedIn method changes that on login/logout. This class only uses Abstract Data Types (ADT) as fields and has no dependency on another class.

Classes UMLs

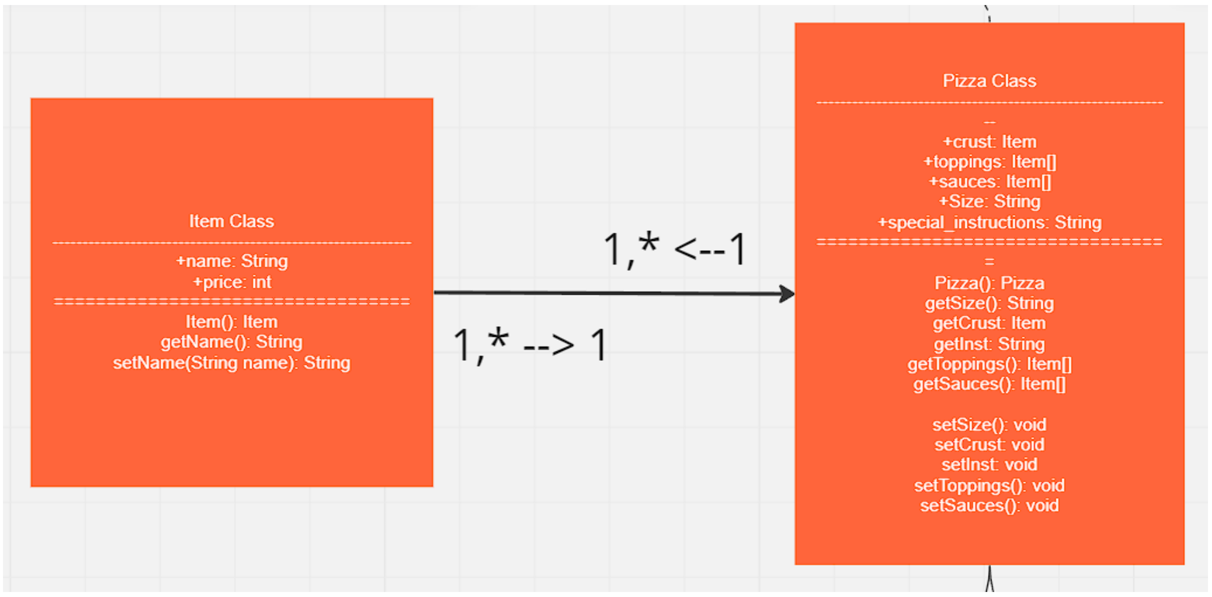
These diagrams show the attributes and methods for the pizza and items classes. They were used to plan and construct the class before the implementation (coding) phase. They show the types of the attributes, their scope, the data methods expect, and the data methods return.

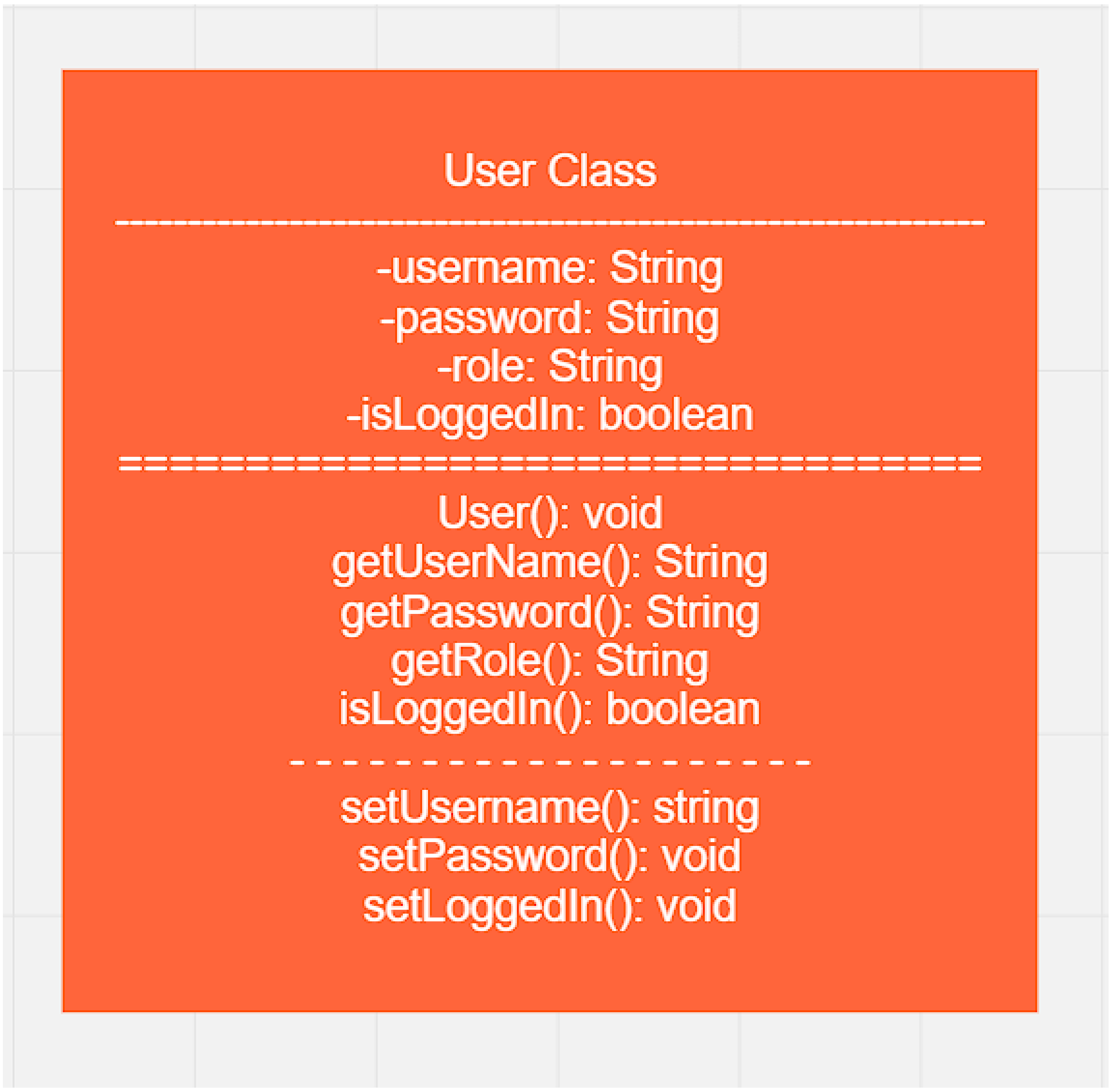


The pizza class references the item class whose structure is also shown in the uml below.



Since pizza stores one, or a set (array) of items, we know that one pizza can have references to multiple item instances. However two pizzas cannot share the reference to the same instance of an item, since pizza creates a deep copy of each item before storing its reference. This relationship is indicated in the diagram below.





User Class

-username: String
-password: String
-role: String
-isLoggedIn: boolean

User(): void
getUserName(): String
getPassword(): String
getRole(): String
isLoggedIn(): boolean

setUsername(): string
setPassword(): void
setLoggedIn(): void

The user class uses only ADT attributes and does not depend on any other class.

Brian Nguyen

Ordering Class (Graphical User Interface)

The interface of ByteSized was created using Java Swing instead of JavaFX as the latter is no longer supported by Oracle as of its most recent update. Thus, in our best interest of not going to an earlier version of JDK, we are able to successfully compile a stable GUI to support the many classes.

It contains multiple local variables that operate with the visuals as well as contribute to proceeding with the other classes' methods. There are also many imported objects and components from JFrame that are used for function but will not be included in the UML or will be documented because it is available to be read through in Java Swing's official documentation.

Design Choice

The interface class code block is made up mostly by the interface's look and feel. That includes styling, positioning, and creating and connecting each component with its respective class method. In regards to its own distinctive methods, Main offers a JFrame constructor, with its JPanel contents, toppingLabel, to better organise the toppings, and intTo$(), to better label the price points. Otherwise, most of the code in Main is made up of labels, buttons and text fields that, as mentioned, interact with the other classes in the project.

Ideally, the interface's purpose should only be a visual representation of the processes and outcomes of the project. Thus, we've stood by the single-responsibility principle and concluded all the methods and components of the interface to only be a result of such. We also wish to assume that we've stuck by the Interface Segregation Principle as logically as possible. From a UX perspective, it would not make sense for the interface to be any more segregated as it currently is with the little information it has to offer. In the upcoming future, this may change but presently it is presumed so.

The Open-Closed principle holds due to the Main GUI natural behaviour of acting as a frame and only extending out. To express in more detail, JFrame holds an architecture that revolves around implementing a panel of its items and limited functionality that can only be presented through other classes and methods. With this skeleton structure, there is no need to ever modify but also a journey of extending its software to access different utilities.

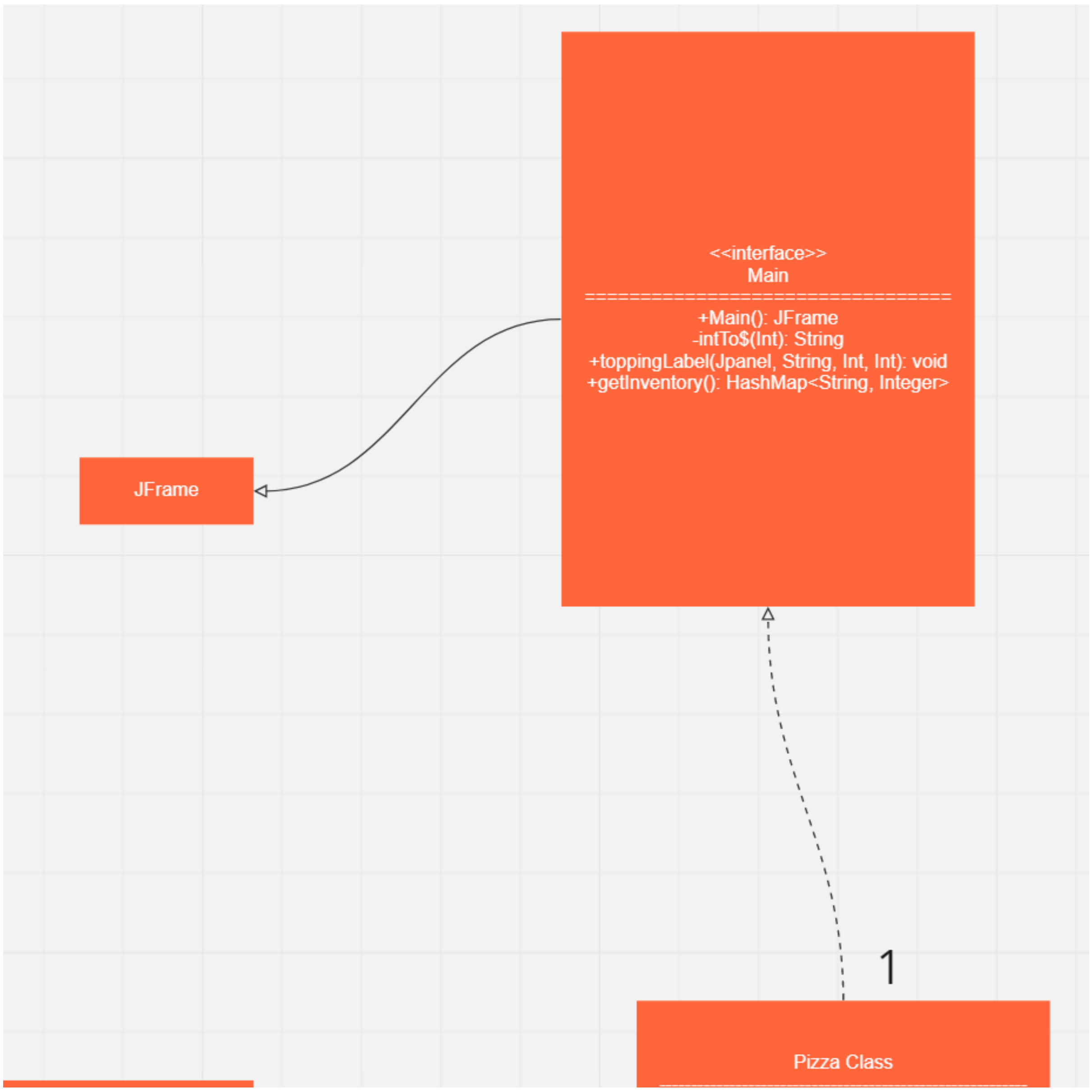
With how simple the software currently is, Liskov substitution principle and dependency inversion principle holds as it only interacts as a child to JFrame. These two principles will be held in higher view in the coming future where there are more to implement.

Refactoring Changes

When updating this class for the latest iteration, we ended up changing its name to Ordering since we had a new Main. In order to cut down on code-duplication, we created new helper functions to automate the process of creating the radio buttons. The most significant change though, was changing the stub database methods. We added new methods for sizes, sauces, and doughs, and with the exception of sizes, each of them now interfaces with the real database rather than a hard-coded stub. Though we didn't add the ability to remove items from the database, Ordering.class is able to comfortably display an inventory that has missing items (perhaps some of them are sold out).

Class UMLs

There are two UMLs pointed out when in discussion of the GUI—Main and JFrame. JFrame is a class that Main extends to in order to reach out to its methods and properties.



As an interface, attributes and states will not be pointed out in the UML. The Pizza class has a dash arrow with an empty head pointing towards Main which is an implementation. The Main class has a solid arrow with an empty head pointing towards JFrame to show inheritance.

Registration and Main Class

Similar to each other, the two classes operate as individual JFrames that correlate with registering and login respectively.

They include private username and password text fields that correspond to their login and registration methods built within, as well as a submit button that does the action given by their names. Each class is short in length so there is not a lot to say regarding their design.

Design Choice

The design of both classes registration and main are standard to the programming language. Meaning, the choice in how to design either classes and their inner methods are less reliant on the individual and more so on the language and its behaviour.

Nonetheless, the choices are made in good conscience. There are no unnecessary extensions nor dependencies and each class is only containing its most apparent use. For example, the classes HintText and RoundText that are used to style specific labels are kept separate for that reason, similar to that of inventoryDisplay.

Testing

Mustafa

Pizza Unite tests

The testing Junits for the Pizza class cover the majority of the class's methods. It checks each getter method and makes sure the returned data is as expected.
It also tests the manipulation (setter) methods on the ingredients of the Pizza. One test changes the given sauces, one changes the toppings, and another tests changing or removing the special instructions. The asserEqual statement uses the default equals method of the classes so I needed to overwrite that and make the Items compare each other according to the name. Issues arise in testing the Pizza class since sauces and toppings for example store an array of Objects (Item) so I would also need to overwrite the equals in the Item class. For now, in testing, the String name is manually fetched and compared to a manually entered String. More polishing needs to be done.

User Unite tests

The User’s Junits test for the creation and fetching of the user object. It covers all the getters and setter methods. It also makes sure the client cannot create a user with an admin role as it would be a system breach. Each user is supposed to be assigned a default “user” role and if the teach chooses to create admin accounts they would have to be manually entered into the json database.