INSTRUCTIONS ON HOW TO RUN THE PROGRAM

Please see the Instructions file in the Phase2 folder.

UML DIAGRAM

Please see UML folder

TEST COVERAGE

1. Our test coverage is currently 90% of all methods excluding android activities classes in the program
2. **Note:** tests in the Presenters and Use Case folder all pass but give many exceptions. These exceptions are caused by Android's file structure; the contexts initialized in these unit tests cannot be used to call the getAssets() method in the database, which is mandatory for data saving. Note that the tests still pass because data saving does not influence anything except when the program initializes. Hence, the exceptions merely indicate that the file saving during tests are unsuccessful; the functionalities of the tested methods are all implemented correctly.

MAJOR DESIGN DECISIONS

- **CLEAN ARCHITECTURE WITH ANDROID**
  - We had difficulty figuring out how to implement Clean Architecture after migrating our project to Android. We tried treating the Android Activities as Presenters, but that was not correct, so we had to continue our research, and eventually came upon the Android Model View Presenter (MVP) architectural pattern and found resources applying it to Clean Architecture. We also remembered the alternative version of the MVP with a controller that the professor gave us in the lecture slides. We decided to use the more traditional Android MVP architectural pattern applied to Clean Architecture. Most references (including this link the professor shared in this piazza post)when

working with Android GUI use the MVP architectural structure (with deprecated controller) and apply it to Clean Architecture layers. We felt that this structure worked well with our project, as our view was not complicated enough to require complex processing of the user's input to interact with the use cases, nor complex processing of the database's information for the view to display information to the user. Hence using the traditional MVP structure with presenters that retrieve information from the use cases and decide what happens when the user interacts with the view suited our program well.

- **OUTPUT BOUNDARIES**
    - When implementing the output boundaries between the use case and presenter, we originally had three different versions as we all implemented it differently. One version passed in the class that implemented the output boundary as a parameter in the constructor of the use case. Another version passed in the output boundary as a parameter of the methods in the use case that needed the boundary. The third version we eventually chose to use applied dependency injection and had a method that set the output boundary attribute in the use case to the desired presenter class. We chose to use the third version because it reduces the hard dependency of the high-level classes on the low-level classes.
- **DATA ACCESS**
    - Issue: Once we had created the classes for data access, we needed to decide how our use case would access the database. To access the database, we needed both the Android context (in our case, an activity), and the name of the file. Our issue was deciding how we wanted to pass this information to the use cases.
    - Decision: Since the android context required to use the getAssets() method does not depend on the context itself (I.e. any context/activity will do the job), we decided to set the context attribute in all use cases that needs data read/write as static and pass in the MainActivity as the context. By doing so, a common context is shared by all use case instance, which can be used for their data access, and no other activities or presenters need to pass down context to their use case.

CLEAN ARCHITECTURE

1. Our program applies the traditional Android Model View Presenter (MVP) architectural pattern to Clean Architecture.
2. The Android Activity implements the view and is passive. When the user interacts with the view, the activity calls on a method in the presenter based on the interaction. The presenter receives information about the user's interaction with the view and decides what to do based on that interaction. If the presenter needs information from the model or to update the model, the presenter will call on the use case. The use case then either retrieves information from, or modifies, the entities or database depending on the method the presenter called. If the presenter requested information, the use case will then provide this information to the presenter through an output boundary interface. The information in the presenter is updated from the method called, and the presenter then formats this and tells the Activity what to display through an interface.
3. Can see UML diagrams for examples of how our classes in different layers interact

SOLID DESIGN PRINCIPLES

- **Single Responsibility Principle**
  - Classes are separated by their functionality, and each class has only one job. For example, the PlaceOrder class only takes the values and places an order to the queue with the given dishes.
  - Classes are named according to their functionality. E.g. EnrollNewStaff is responsible for enrolling new staff
- **Open-Closed Principle**
  - To see open closed principle in our program, use the adaptor pattern as an example. For creating a use case to do dish serving and in presenter the client asked for a different interface to serve dishes. The adapter can make the ServeDish class be wrapped and used as the interface the client want. i.e. ServeDish being an attribute of

    - ServingDishAdapter which implements the DeliveryInputBoundary interface, the interface used in the presenter.
    - A potential violation is the Inventory class, in which items are currently referred to by their name. A future change we would make if we had more time is referring to items by their id instead of name, however due to time constraints that is not possible. Hence, we will need to completely refactor the inventory class (and thus it will violate the open-closed principle).
- **Liskov Substitution Principle**
    - Any class that implements the super class User can login. The login process does not care which type of user it gets.
    - Similarly, any subclasses of Order can go through the place order process and be added to the OrderQueue
    - For Inventory, the super abstract "hook" method are overridden in child classes HasExpiryDate and HasFreshness, while its implemented methods are inherited by the child class.
- **Interface Segregation Principle**
    - Most interfaces define only one or two methods necessary to perform the function required. For example,
    - Some interfaces such as PlaceOrderViewInterface are quite large and define many methods. Even though these methods are all necessary for the PlaceOrderActivity to successfully display the dishes ordered and place an order, it may be better to separate the interface. However, separating the interface would mean the presenter would have multiple interface attributes, which could cause potential issues if another class that doesn't implement both interfaces were to use the presenter.
- **Dependency Inversion Principle**
    - The high-level classes our program do not directly know about the lower levels, and instead, they depend on an abstraction. Information is passed from the inner layers of Clean Architecture to the outer layers through these interfaces. For example, when prompted, the use case DishInformation calls on the method defined in the PlaceOrderMenuOutputBoundary interface to update information on the menu in the presenter PlaceOrderMenuPresenter.

PACKAGING STRATEGIES

We used a packaging strategy that is hybrid of layers and components. We originally only had packaging based on layers in order to clearly separate the layers of Clean Architecture and ensure classes are not violating Clean Architecture, but once our program grew larger, there were too many classes in each layer and it was hard to find specific classes. Hence, we also added packaging by components, making it easier to find classes related to a specific functionality.

On the top level, the packages are arranged in layers, having the packages of Framework, Presenter, UseCase, and Entity.

Inside each layer, the sub packages represent the components of the system, as follows:

- Framework
    - Database
    - GUI
        - Contains sub-packages for pages for each type of user:
            - Delivery staff
            - Serving staff
            - Customer
            - Manager
            - Kitchen
            - LoginSystem
- Presenter
    - Presenters that present data to user and receives input in MVP view model
        - Inventory presenter
        - Kitchen presenter
        - Login presenter
        - Manager presenter
        - Menu presenter
        - Order presenter
        - Review presenter

            - Staff presenter
- Use Case
    - Packaging based on the action the use cases does
    - Deliver Order
    - Serve Dish
    - Dish List
    - Enroll Staff
    - Inventory Factory
    - Kitchen
    - Login
    - Place Order
    - Review
    - User List
- Entity
    - Packaging based on the entities that work together.
    - Customer
    - Delivery
    - Inventory
    - Kitchen
    - Manager
    - Order List
    - Review
    - User

DESIGN PATTERNS

1. **Dependency Injection**
    a. Many classes (especially classes that need to return an output to an outer layer) use dependency injection. For example, the MenuPresenter injects itself into the DishList use case so that DishList can update the values of the dishes in the MenuPresenter through the MenuOutputBoundary.
    b. We applied this design pattern in order to allow the high-level classes (in the inner layers of Clean Architecture) to update information in a low-level class (in the outer layers of Clean Architecture) without having the high-level class depend on or even know about the low-

level class. Instead, with dependency injection, we can instantiate the low-level class and inject it into the high-level class. The high-level class then uses this low-level class through its interface, and in this way, we avoid the high-level class having to directly instantiate the low-level class.

2. **Simple Factory Design Pattern**
    a. The PlaceOrder uses the OrderFactory to get the type of order (Dine in, Delivery) that is being placed.
    b. We applied this pattern because the type of order could be Dine-in or Delivery, and in the future when the program is extended, there could potentially be even more types of orders (such as a Take-out order that does not need to be delivered). Although any future addition of order types would require adding more lines of code to the factory, it is better than the alternative of having the PlaceOrder class check which type of order it needs to place in addition to processing the dishes ordered. By implementing this factory, the PlaceOrder class does not have to worry about the type of order it is placing, it just adds the order to the OrderQueue.
3. **Iterator**
    a. Both the DishList and ReviewList classes implement the iterator design pattern (DishListIterator and ReviewListIterator respectively)
    b. We applied this pattern because classes such as the DeleteReviewUseCase need to iterate over the ReviewList to determine which reviews are rated below a given integer, however we do not want the DeleteReviewUseCase to know about the underlying implementation of the ReviewList.
4. **Template**
    a. For Inventory, there are HasFreshness and HasExpiryDate classes where they were constructed differently and share the common methods.
    b. We applied this pattern because they were previously implementation of the same interface, means they have common parts in methods but have some differences.
    c. Implementing by changing the previous interface into an abstract class where the share methods are implemented there and the

constructors for these two child classes are different, and the methods related to their different attributes are implemented individually in the body of these two classes.

d. The hook methods are abstract in the base class and the child classes have "is a" relationship with the superclass class. Meets the checklist in https://sourcemaking.com/design_patterns/template_method

5. **Private Class Data**
    a. Implemented the design pattern of private class data in the class of User to use a class UserAttributes to hold the user's static data of id, name, and password to reduce exposure and enforce one-time set up property of these data.
    b. This is applicable: The user's data like id, name, and password are one-time set up attributes that being set up when a user is added and shouldn't expose to outside as these are confidential data.
    c. How this is implemented: Created data class UserAttributes to have private attributes id, name, and password. Set up constructor to initialize these data. Provide getters to get these fields. Use constructor in main class User to construct the UserAttributes to be a private attribute of User then only call getter in User's methods. Changing is made possible only to create a new UserAttribute class.
    d. This implementation is suitable here as it meets the problem this design pattern applies to: Private data should not be changed unintendedly. This implementation is correct as it matches the check list of the design pattern referenced from source making

(https://sourcemaking.com/design_patterns/private_class_data):

        i. Create data class. Move to data class all attributes that need hiding.
        ii. Create in main class instance of data class.
        iii. Main class must initialize data class through the data class's constructor.
        iv. Expose each attribute (variable or property) of data class through a getter.

6. **Strategy**

a. Strategy design pattern is implemented in the classes ServeDish and DeliverOrder. They both implement the DeliveryInputBoundary interface with different implementations: ServeDish takes dish from ServingBuffer while DeliverOrder takes order from DeliveryBuffer.
b. This design pattern is applicable here as both ServeDish and DeliverOrder implements similar responsibility. They both implement the method that can mark an item as completed, get a new item for a staff, and show the content of a dish or an order. When they are called by a client, they have a similar way of responding but only with different implementations. Therefore, from the client viewpoint, both implement the process of delivering an item, so to bury the implementation before the interface will be the ideal implementation here, the strategy design pattern.
c. How this is implemented: Subtracted their common methods, get the next item, mark current item as completed, show content of current item. Then define the interface DeliveryInputDoundary to contain the common methods. Use the two classes, ServeDish and DeliverOrder to implement the interface. From the client end, calls on the interface to use their methods.
d. This implementation shows the strategy design pattern as it matches the properties of strategy design pattern: This is a problem that we have two classes that can be viewed as providing similar functionality but using different and in-couplable implementation (They act on different things, depend on different classes). The way to solve it is to subtract a common interface so that client only calls on the interface instead of the actual implementation. Furthermore, it matches the check list of the strategy design pattern referenced from source making (https://sourcemaking.com/design_patterns/strategy):
    i. Identify an algorithm (i.e. a behavior) that the client would prefer to access through a "flex point". In this context, the method to get next item, show content of current item, and mark current as delivered are common behavior the client would access through same method call.

ii. Specify the signature for that algorithm in an interface. The signature of the behavior can be subtracted to be a common method defined in interface.
iii. Bury the alternative implementation details in derived classes. The derived classes, ServeDish (ServingDishAdaptor is the wrapper, not the implementation) and DeliverOrder implements their separate implementation of the interface. They are not visible from the client side as the client side only knows the interface.
iv. Clients of the algorithm couple themselves to the interface. From the client side, staff presenter, it uses the interface to call the methods of the use-cases, coupling itself to the interface.

7. **Adapter**
    a. Implemented the design pattern in the class of ServeDish and ServingDishAdapter to convert the instance of ServeDish class to DeliveryInputBoundary interface (They differ from each other in the method's parameter but provides similar functionality from the angle of client).
    b. This is applicable: The ServeDish class was implementing DeliveryInputBoundary and has attributes unnecessary for the class itself. And the Input boundary is created and used in the outer layer.
    c. How this implemented: Change the ServeDish into a class implementing different interface and create a wrapper implementing DeliveryInputBoundary which has a ServeDish class as an attribute. The ServeDish related functionality in the wrapper rely on the ServeDish class.
    d. This implementation has the adaptee class as an attribute and only DeliveryInputBoundary interface is called by the clients matches the checklist in https://sourcemaking.com/design_patterns/facade
8. **Observer**
    a. The observer design pattern is implemented between the PlaceOrderActivity and the CurrentOrderDishesActivity. The Kitchen user interface displays the current order that they should be working on and shows nothing when every dish is completed, and no more order is available. To make sure that whenever a new order is placed,

the Kitchen user interface immediately get notified and display the new order, an observer design pattern is implemented such that the Kitchen (CurrentOrderDishesActivity) listens to the order placing process in PlaceOrderActivity, and gets notified whenever an order is successfully placed, which prompts the new order on the Kitchen’s user interface.