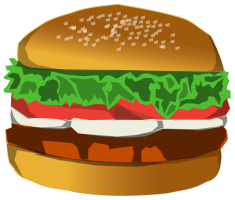


U Eat:

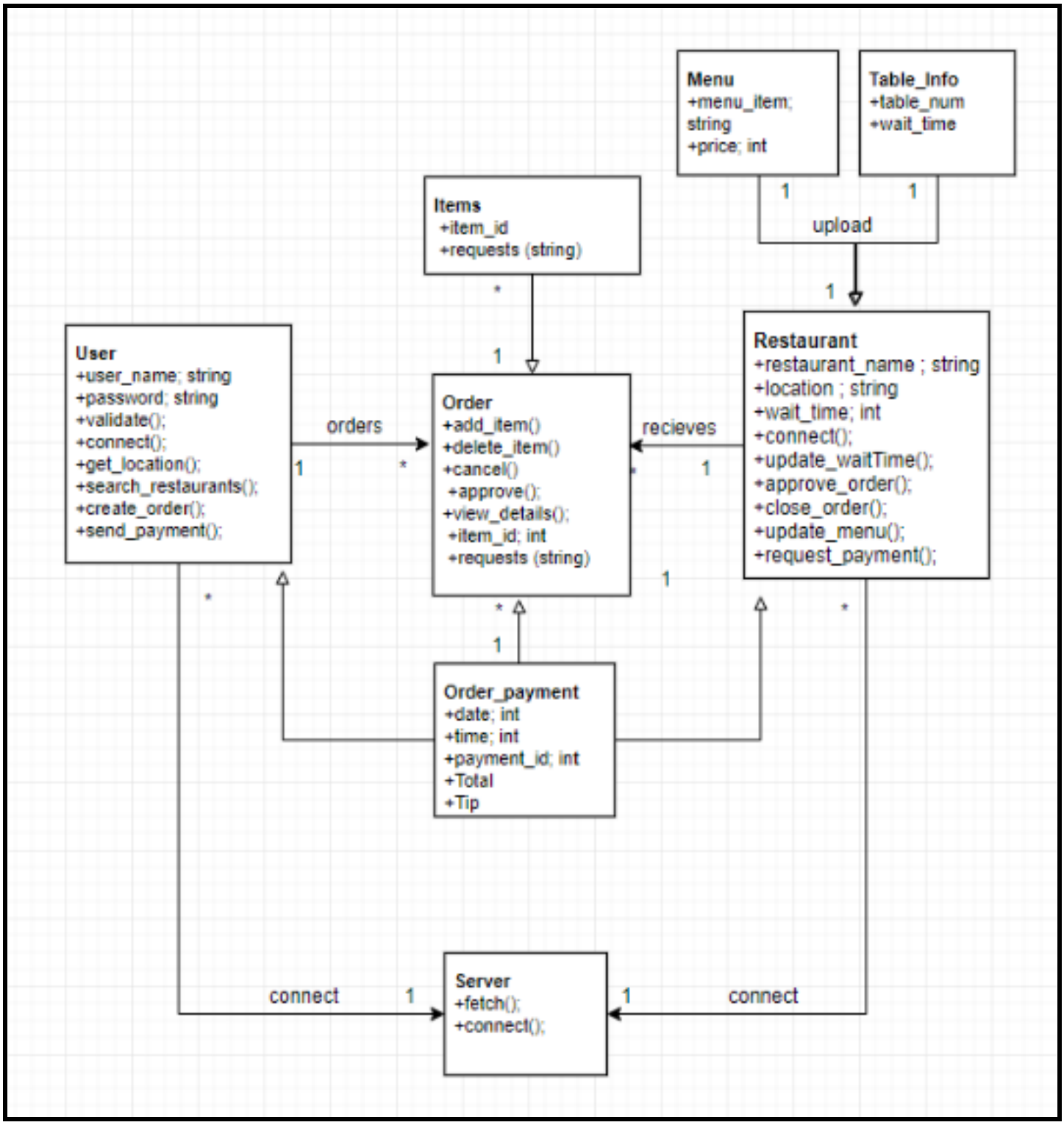
A Dining Concierge App

-System Design

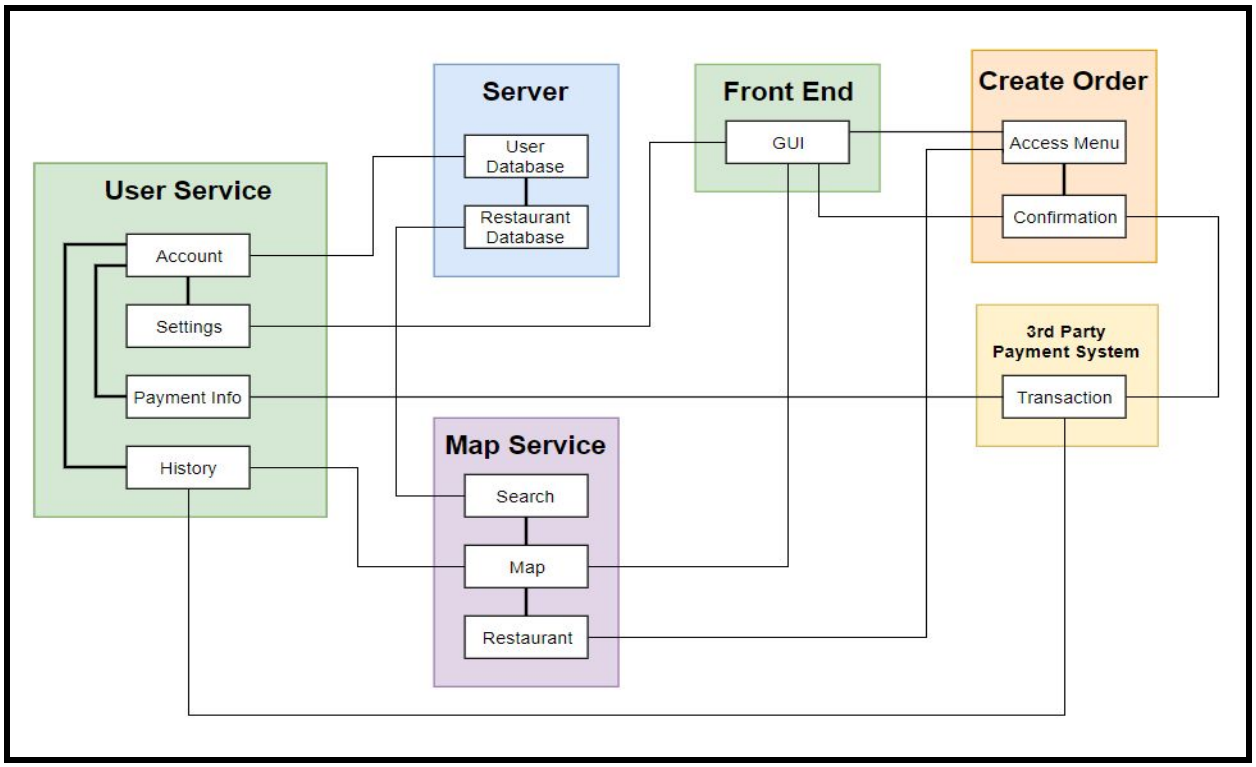
Group 28

Luke Carter
Ibrahim Mahmoud
Miles McCoy
Mathew McDade
Timothy Tseng

❖ UML Class Diagram:



❖ Implementation Packaging:



❖ Coupling

- **Content Coupling:**
  - The user's account should be able to modify the User Database.
  - A successful transaction will modify the user's history.
  - The order confirmation calls on the 3rd party payment system to process the transaction.
- **Stamp Coupling:**
  - The Restaurant Database provides structured data to the search service.
  - The user's history provides default locations to the map service.
  - The user provides payment information to the 3rd party payment system.
- **Data Coupling**
  - The chosen restaurant provides an unstructured menu to the order creation.

❖ Cohesion

- **Functional Cohesion:**
    - Front end: Gives the client graphical interface to interact with the system
    - Create order: separates the menu database call and the user item selection into a single compartmentalized package. Keeps the frequently changing order data in an individualized package.
    - 3rd Party Payment System: external to our system.
- **Communicational Cohesion:**
    - User Service: Encapsulates the user data and order history.
- **Procedural**
    - Server: Provide main functionality and access to user/ restaurant database, needed for persistence.

❖ Design Model Assessment:

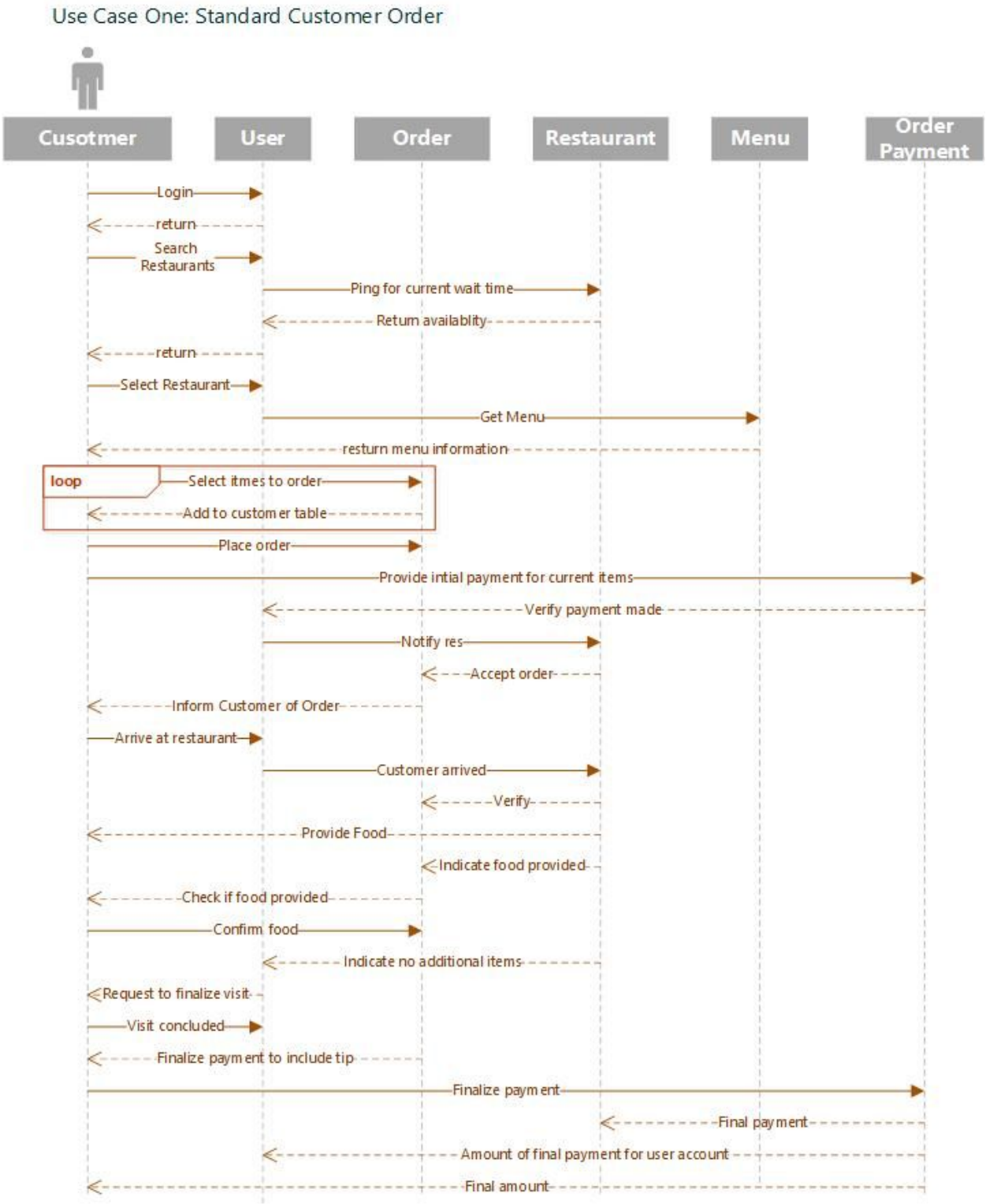
To develop our system effectively we would need to use an iterative approach instead of an incremental approach. Our system has many different parts that must work together to provide complete or partial functionality. If we were to for instance build the user interface it would be just a shell without the core server processing as well as the database. Our server application would be useless without the database as it would have nothing to compare, to pull from, or send to. Our system needs to have a base level or functionality from each element to even be able to test functionality. A good example is simply logging in to the app by a user. The user facing application can't check the login information it must send that information to server the server pulls the appropriate information from the database and compares user's login information to the database information. Without one of the parts of the system you could not login. Our code will be very reusable and it will mostly be comparing, sorting, displaying, pulling, and sending information from the user's application to the server and to the database. All that would need to be changed to use it as a solution to a different problem would be to change the types of data, and the way it was displayed.

❖ Design Patterns:

- The **Adapter** design pattern is one that will be useful in our system because we are using third party payment processing system we are going to need a way of providing them with information as well as getting back information from them. The adaptor can translate a complex system of payment processing and make it much easier for our system to interface with.
- The **Template** design pattern could be a good option for our system. We have many general steps that need to be followed particularly when it comes to getting a table and placing an order but there is room for each case to be different in some respects. With a template we can have a general unified order that will make it easy for the restaurant to read, understand and create. The template will help new users in understanding how the system works and how it should be used but also will allow for more experienced users to get more a more individualized experience.
- The **Facade** design pattern could be an option as well for our system. Our user application will be using a subsystem our server application to do a lot of the work and our server will be a fairly complicated system. The facade is often used as the front facing interface that hides the complexities that are done behind the scenes. Our system is exactly that. Our system is designed to have a fairly simple front end for the user application with a bit more complicated back end server application. This most likely the best pattern for our system at least with relation to the general overview of our system.

Overall Our system will likely be composed of a few different design patterns based on the level of the system we are looking at. From the widest angle you will see a Facade, then going down lower you could see a template design within the Facade, and then you could see the adapter pattern within the template design.

❖ Use Case Sequence Diagram:



❖ Interface Contracts:

➢ User-Order Interface:

- **Add an item to a user's order:** The user-order interface is one of the primary component interfaces in our system as it is essentially the way food gets to the table. The preconditions for this interface are that the user has an account and has been authenticated, an order object has been created, a restaurant is associated with the order object, and that the restaurant's menu is displayed in the customers GUI. Postconditions for this interface are that an item from the precondition specified restaurant's menu is associated with the authenticated user's order. This would be implemented as a member function of the order class, addItem().

➢ Graphical User Interface:

- **Display the user interface to the customer:** Maintaining a smooth interface between the client application and the customer is really what our system is all about as a retail customer facing product. This interface will be implemented through a facade design pattern with preconditions being that the client application has successfully retrieved user/restaurant/order data via REST calls to the server and that the front-end components are built to sufficiently display the view data. The postconditions of the display interface are that the context-appropriate view is displayed to the user with appropriately interactive facade components.

➢ Server-Services Interface:

- **Authenticate the user:** User authentication is an interface between the user component and the external authentication service component. The precondition for this interface is that the user has an existing account with the U Eat system. The postcondition is that the authentication service returns an authentication token--valid for the rest of the current session--which is attached to the user or an error that the user could not be authenticated.

❖ Exception Handling:

- The exceptions we will focus on at this point in the system's development are those most commonly identified as root causes in our failure mode analysis as discussed in the previous Architecture documentation. Our primary goal with exception handling will be, first, to ensure that error states are detected and corrected as soon as possible and, second, to make error states recoverable to non-error states without information loss.

➢ User Application Exceptions:

- For our user facing applications, exceptions could be generated by receiving bad or unexpected data from the user or from the server. Because our application is so reliant on a reliable user interface, we should choose an application language and front-end framework with strong built-in exception generation and handling patterns. The goal being for the component where the error is detected to generate the exception and have it caught by a parent component that can rebuild the child component in a safe state, either by retrieving new data from the user/server if possible or by reverting the component to the last safe state.

➢ Server Exceptions:

- The server side application should rarely generate internal exceptions, but where they do occur they should be caught and resolved immediately. The most likely errors would be in a state mismatch between the client, server, and database. In this case the exception should be addressed by falling back first to the pre-error server state or, if the server state is unresolvable, the latest database state for the interaction.

➢ External Services Exceptions:

- The external services for our system include a database, an authentication service, a payment service, and a location service. Among these exceptions in the payment service should probably take the highest priority for both security and ensuring user trust. If an exception is raised during the payment processing transactions, we should handle the exception by doing a reverification that the payment service is reachable, checking whether the payment has been processed by the payment service, and reprocessing the payment otherwise, providing an error message to the user if an error is unrecoverable.

❖ Team Contributions:

- ➢ All: Communication via private Slack channel and collaborative Google Doc.
- ➢ Timothy Tseng - UML Class Diagram, Interface Contract concept.
- ➢ Miles McCoy - Use Case Sequence Diagram, Implementation Packaging.
- ➢ Ibrahim Mahmoud - Implementation Packaging.
- ➢ Luke Carter - Design Model Assessment, Design Patterns.
- ➢ Mathew McDade - Document Preparation and Submission, Portfolio Webpage, Interface Contracts, Exception Handling.