Project Documentation and Transition to Online System
Goals:

Thoroughly document the current Java CLI system. This includes:
Explaining the implementation details of each component (where and how it's used).

Providing a clear rationale behind design choices (why it's used).
Including code snippets for each section to illustrate functionality
and decision-making.

Analyze and propose theoretical improvements for transitioning to an online system. This involves:

Considering the implications of moving from limited CLI input to potentially unlimited input in an online environment.

Addressing the challenges of transitioning from a CLI navigation paradigm (using "enter") to a more interactive online interface.

Exploring how existing components can be adapted or reused with minimal changes.

Emphasis:

- Clarity: Provide comprehensive explanations and information to ensure readers fully understand the system and the proposed transition.
  - Depth: Delve deeply into the technical details and potential challenges of transitioning to an online system.
  - Adaptability: Focus on creating a flexible foundation for the online system that can accommodate evolving

requirements and input variations.

Document Submission Structure: Personal ideas
To achieve these goals, we'll approach the documentation and analysis in a structured manner:

Current System Documentation:
For each component:
Implementation: Detail where and how the component is implemented within the system.

Rationale: Explain the reasons behind choosing this particular implementation approach.

Code Snippets: Include relevant code snippets to demonstrate the component's functionality.

Transition to Online System:

Input Handling: Analyze the differences between handling limited CLI input and the potential for unlimited, diverse input in an online system. Propose strategies for managing this transition effectively.
Navigation Paradigm: Discuss the transition from CLI-based navigation (using "enter") to a more interactive and user-friendly online navigation approach.
Component Adaptation: Evaluate the existing components and identify how they can be adapted or reused in the online system with minimal modifications.
Theoretical Improvements: Explore potential enhancements and optimizations specific to the online environment, considering scalability, performance, and user experience.

By following this structured approach, we can create comprehensive documentation that not only explains the current system but also lays the groundwork for a smooth and successful transition to an online platform.

Notes on new implementation of the project :
"We are now onto theoretical improvements and online systems with less limited to unlimited input qualities and

not constrained to the java cli input."

This alteration is significant and requires careful analysis of the project menu navigation, the driver assignment algorithm creation

1. Object-Oriented Design Implementation

Encapsulation code snippets detailed

Our project demonstrates encapsulation through classes like
- Driver,which encapsulates driver-related data and behaviors:
   - src/main/java/model/Driver.java
```java
public class Driver {
   private final String name;
   private final String licensePlate;
   private final List<Integer> ratings;
   private final List<Order> currentOrders;

### Driver Class Rationale
The `Driver` class implementation demonstrates strong encapsulation principles for several key reasons:
- Private fields with public getters ensure data integrity
- Immutable fields (name, licensePlate) prevent unauthorized modifications
- ArrayList usage for ratings and orders enables dynamic collection management
- Methods like `getAverageRating` encapsulate complex calculations
- Clear separation between data storage and business logic

   public Driver(final String name, final String licensePlate) {
       this.name = name;
       this.licensePlate = licensePlate;
       this.ratings = new ArrayList<>();
       this.currentOrders = new ArrayList<>();
   }

### Order Class Rationale
The Order class implementation focuses on:
- Simple, focused responsibility for order data management
- Status tracking with controlled state transitions
- Immutable order ID for tracking integrity
```

- Clear getter/setter patterns for controlled data access

src/main/java/service/MenuManager.java
- Centralizes menu item management
- Provides factory methods for menu item creation
- Maintains menu consistency across the system
- Enables easy menu modifications and updates

src/main/java/service/OrderManager.java
- Handles order lifecycle management
- Maintains order state transitions
- Provides order lookup and modification capabilities
- Ensures order processing integrity

src/main/java/service/DriverManager.java
- Manages driver pool and availability
- Handles driver ratings

src/main/java/service/NotificationService.java
- Centralizes system messaging
- Provides consistent error handling
- Enables future expansion to different notification channels
- Maintains clean separation of notification logic

src/main/java/util/ConsoleInputHandler.java
- Abstracts input validation logic
- Provides type-safe input processing
- Reduces code duplication
- Enables consistent error handling

src/main/java/model/MenuItem.java (Abstract Class)
- Defines common behavior for all menu items
- Enables polymorphic menu item handling
- Provides base implementation for shared features
- Ensures consistent menu item structure

Encapsulation code snippets detailed
- seen in the CLI class
```java
public class DeliverySystemCLI {
   private static final String FORWARD_NAVIGATION = "/";
   private static final String BACK_NAVIGATION = "\\";
   private static final Logger logger =
Logger.getLogger(DeliverySystemCLI.class.getName());
```

```
    private final MenuManager menuManager;
    private final OrderManager orderManager;
    private final DriverManager driverManager;
    private final NotificationService notificationService;
    private final ConsoleInputHandler<Integer>
positiveIntegerHandler;
    private final DeliverySystem deliverySystem;
    private final Scanner scanner;
    private boolean running;


    public DeliverySystemCLI(
            final MenuManager menuManager,
            final OrderManager orderManager,
            final DriverManager driverManager,
            final NotificationService notificationService,
            final ConsoleInputHandler<Integer>
positiveIntegerHandler,
            final DeliverySystem deliverySystem,
            final Scanner scanner) {
        this.menuManager = menuManager;
        this.orderManager = orderManager;
        this.driverManager = driverManager;
        this.notificationService = notificationService;
        this.positiveIntegerHandler =
positiveIntegerHandler;
        this.deliverySystem = deliverySystem;
        this.scanner = scanner;
        this.running = true;
    }
}
```


Encapsulation properties: All fields are private and final where appropriate

Immutability using final fields to prevent modification after initialization

Inheritance & Polymorphism

The system implements polymorphism and inheritance through the menu item hierarchy.

- src/main/java/model
- Model.Order (extends MenuItem)
- Model.Fries (extends MenuItem)
- Model.Hamburger (extends MenuItem)
- Model.MenuItem (abstract class)
- Model.Drink (extends MenuItem)

```java
public class Order {
   private String id;
   private String deliveryAddress;
   private String status;


   public Order(String id, String deliveryAddress) {
       this.id = id;
       this.deliveryAddress = deliveryAddress;
       this.status = "PENDING";
   }


   public String getId() {
       return id;
   }


   public String getDeliveryAddress() {
       return deliveryAddress;
   }


   public String getStatus() {
       return status;
   }


   public void setStatus(String status) {
       this.status = status;
   }
}
```


- Polymorphic behavior in input processing

```java
public class DeliverySystemCLI {
   // ... other fields and methods


   public void processInput(String input) {
       if (input == null || input.trim().isEmpty()) {
           System.out.println("Error: Input cannot be
empty");
           return;
       }
       // rest of the method...
   }
}
```


Abstraction

We use interfaces and abstract classes to provide
abstraction layers and referenceable and maintainable code

We use interfaces and abstract classes to
- provide abstraction layers
- provide referenceable and maintainable code
as seen in the package with relative path
- src/main/java/model

Key areas of development:
src/main/java/CustomException

- Tracking and logging of business logic through the system
by improving the custom errors that were created.

- The use of these custom errors were invaluable in initial
development, as they provided a clear and concise
explanation of the issue at hand.

2.0 System Weaknesses & Limitations

2.1 Further abstraction needed of the main application
package CLI methods

- DelieverySystemCLI class is the cli class for the application so it handles all cli logic. It is responsible for handling user input from the CLI for types necessary. This and calling the appropriate methods in the DeliverySystem class for handling.

- The DeliverySystem class is responsible for handling the business logic of the application, including managing orders, drivers, and menu items though could be improved to
- be responsible for handling the business logic of the application, including managing orders, drivers, and menu items that are inputted from the CLI.

2.2 User Interface Issues

```
// Handle exit command directly
                if ("exit".equalsIgnoreCase(input) ||
"9".equals(input)) {
                    this.cleanup();
                    break;
                }
```

The input is first checked for "exit" or "9" however the input is then parsed as a number and passed to handleMenuChoice()
This means handleMenuChoice() will never actually receive the "exit" string - it's handled earlier in the flow.

We weren't seeing explicit lack of input errors from this method.

```
                final String input =
this.scanner.nextLine().trim();


                // Handle exit command directly
                if ("exit".equalsIgnoreCase(input) ||
"9".equals(input)) {
                    this.cleanup();
                    break;
```

```
                }


                if (input.isEmpty()) {
                    System.out.println("Please enter a valid
option.");  // This message isn't showing
                    continue;
                }


                try {
                    final int choice =
Integer.parseInt(input);
```

The issue is that we're using scanner.nextLine() to consume the input in multiple places throughout the code.

Subsequent empty lines are being consumed by other scanner.nextLine() calls within various menu handlers and input processors correctly however not for our 'exit' command string.
To fix this and ensure we actually catch this string as well as empty string inputs we should modify the code to:

```
                // Handle exit command directly
                if ("exit".equalsIgnoreCase(input) ||
"9".equals(input)) {
                    this.cleanup();
                    break;
                }


                if (input.isEmpty() || input.isBlank()) {
                    System.out.println("Error: Empty input
is not allowed");
                    System.out.println("Please enter a valid
option (1-9)");
                    continue;
                }
```

MenuNavigation

Double Enter required for menu initialization

Should use a \ or / to navigate menus forwards and backwards.

```
Welcome to the Online Food Delivery System!
Press Enter to start... // this is pressed twice to get the menu


```

```
=== Online Food Delivery System ===
1. Place a New Order (Add items to cart and checkout)
2. Check Order Status (View status of an existing order)
3. View Menu (See available items and prices)
4. Manage Drivers (Add/Remove/List drivers)
5. Rate Driver (Rate a driver for a completed order)
6. Calculate Order Total (View total for an existing order)
7. Manage Driver Ratings (View/Update driver ratings)
8. Process Orders (Process all pending orders for delivery)
9. Exit
====================================
Please choose an option (1-9): Please enter a valid option.
2


=== Online Food Delivery System ===
1. Place a New Order (Add items to cart and checkout)
2. Check Order Status (View status of an existing order)
3. View Menu (See available items and prices)
4. Manage Drivers (Add/Remove/List drivers)
5. Rate Driver (Rate a driver for a completed order)
6. Calculate Order Total (View total for an existing order)
7. Manage Driver Ratings (View/Update driver ratings)
8. Process Orders (Process all pending orders for delivery)
9. Exit
====================================
Please choose an option (1-9):
=== Check Order Status ===
Enter your order ID to check its current status.
Enter Order ID to check status: 9
```

```
// should be have an option to exit the choosing of an
option without returning needing to enter an incorrect
number option
Order not found. // we then get this once it's not found
// Then press enter to get the menu.
```

No clear exit option from sub-menus

Inconsistent menu navigation patterns and lack of an exit option from sub-menus from input handling and navigation.

```
Input Handling
Lack of proper validation for empty/null inputs as an empty
string is used as the escape sequence to reinitalize the
menu.
```

3. The current input validation implementation in ConsoleInputHandler:

```93:117:src/main/java/validation/ConsoleInputHandler.java
        while (!valid) {
           try {
              final String userInput =
this.getInputWithTimeout(prompt);
              try {
                 final T parsedInput =
this.inputValidator.parse(userInput);
                 if
(this.inputValidator.isValid(parsedInput)) {
                    input = parsedInput;
                    valid = true;
                 } else {

System.out.println(this.inputValidator.getErrorMessage());
                 }
              } catch (final NumberFormatException e) {
                    System.out.println("Invalid number
format: Please enter a valid " +

this.inputValidator.getTypeName());
                 } catch (final IllegalArgumentException e)
```

```
{
                    System.out.println("Invalid input
format: " + e.getMessage());
                } catch (final Exception e) {
                    System.out.println("Error processing
input: " + e.getMessage());
                }
            } catch (final TimeoutException e) {
                System.out.println(e.getMessage() + ".
Please try again.");
            }
         }
         return input;
      }
```

4. The InputValidationUtils class provides a better approach:

```25:34:src/main/java/validation/InputValidationUtils.java
   public static void validateTextInput(String input,
String fieldName) {
      try {
         if (input == null || input.trim().isEmpty()) {
            throw new ValidationException(fieldName + "
cannot be null or empty");
         }
      } catch (ValidationException e) {
         System.err.println("Error in validateTextInput: "
+ e.getMessage());
         throw e;
      }
   }
```

This ensures that the input is not null or empty and throws a ValidationException if it is.

Current CLI Implementation (referenced in):
```java
   public void start() {
       System.out.println("Welcome to the Online Food
Delivery System!");
       System.out.println("Press Enter to start...");
```

```
        this.scanner.nextLine(); // Single enter to
initialize


        try {
            while (this.running &&
this.scanner.hasNextLine()) {
                this.displayMainMenu();


                final String input =
this.scanner.nextLine().trim();


                // Handle exit command directly
                if ("exit".equalsIgnoreCase(input) ||
"9".equals(input)) {
                    this.cleanup();
                    break;
                }


                if (input.isEmpty()) {
                    System.out.println("Please enter a valid
option.");
                    continue;
                }


                try {
                    final int choice =
Integer.parseInt(input);
                    if (choice < 1 || choice > 9) {
                        System.out.println("Invalid menu
choice. Please enter a number between 1 and 9.");
                        continue;
                    }
                    this.handleMenuChoice(choice);
                } catch (final NumberFormatException e) {
                    System.out.println("Invalid input.
Please enter a number between 1 and 9.");
                } catch (final Exception e) {

DeliverySystemCLI.logger.log(Level.SEVERE, "An unexpected
error occurred", e);
                    System.out.println("An unexpected error
occurred: " + e.getMessage());
```

```
                }
            }
        } catch (final Exception e) {
            DeliverySystemCLI.logger.log(Level.SEVERE, "An
unexpected error occurred", e);
        }
    }
```

Proposed improvement for the input handling and navigation system using a forward and back navigation system with the slash character as the forward navigation and the backslash character as the back navigation. This will enable a more intuitive and user-friendly navigation experience.

```
```java
public class DeliverySystemCLI {
    private static final String FORWARD_NAVIGATION = "/";
    private static final String BACK_NAVIGATION = "\\";


    public void processInput(String input) {
        if (input == null || input.trim().isEmpty()) {
            notificationService.showError("Input cannot be
empty. Use '/' to proceed or '\\' to go back.");
            return;
        }


        switch (input.trim()) {
            case FORWARD_NAVIGATION ->
handleForwardNavigation();
            case BACK_NAVIGATION -> handleBackNavigation();
            default -> processMenuChoice(input);
        }
    }


    private void handleForwardNavigation() {
        if (currentMenu.hasNextLevel()) {
            currentMenu = currentMenu.getNextLevel();
            displayCurrentMenu();
        } else {
            notificationService.showInfo("Already at deepest
menu level");
```
```

```
        }
    }


    private void handleBackNavigation() {
        if (currentMenu.hasPreviousLevel()) {
            currentMenu = currentMenu.getPreviousLevel();
            displayCurrentMenu();
        } else {
            notificationService.showInfo("Already at main
menu");
        }
    }
}
```

Test Analysis and Implementation Status

Test Case Updates
- Update test cases to reflect new functionality
- Add additional junit edge case testing
- continue to implement comprehensive validation testing as needed
- Add mockito testing for complex workflow testing