Functional Requirements

1. **Card Balance Management**

* The system must allow users to load money onto the Oyster card.
* The system must allow users to view the current balance on the Oyster card.
* The system must deduct fares from the card balance when a journey is taken.
* The system must validate that there is enough balance before starting a journey.

1. **Journey Management**

* The system must record the start and end points of a journey.
* The system must differentiate between different modes of transport (Tube and Bus).
* The system must charge the maximum fare when a journey starts.
* The system must calculate the correct fare when a journey ends and adjust the balance accordingly.
* The system must charge the maximum fare if the user fails to swipe out at the end of a journey.

1. **Fare Calculation**

* The system must charge £2.50 for journeys entirely within Zone 1.
* The system must charge £2.00 for journeys in any one zone outside Zone 1.
* The system must charge £3.00 for journeys covering two zones including Zone 1.
* The system must charge £2.25 for journeys covering two zones excluding Zone 1.
* The system must charge £3.20 for journeys covering three zones.
* The system must charge £1.80 for any bus journey.
* The system must charge the maximum fare of £3.20 at the start of a Tube journey.

1. Validation and Error Handling

* The system must validate that the card has sufficient balance for the maximum fare before allowing entry.
* The system must throw an appropriate error if there is insufficient balance.
* The system must handle scenarios where the user swipes out without swiping in and vice versa by charging the maximum fare.

Non-Functional Requirements

1. Performance

* The system should process fare calculations and balance updates quickly, ensuring a seamless user experience.

1. Usability

* The system should be easy to use, with clear instructions and feedback for the user.
* The console application should display clear messages about the current balance and any errors.

1. Reliability

* The system should accurately track journeys and update balances.
* The system should handle invalid input gracefully, providing meaningful error messages.

1. Maintainability

* The code should be modular and well-organized, making it easy to maintain and extend.
* The system should be designed with a clear separation of concerns, with distinct classes for handling different aspects of the application.

1. Scalability

* The system should be designed in a way that allows for easy extension to include additional features or support for more transport modes.

1. Security

* The system should ensure that balance manipulations are secure and cannot be tampered with.
* The system should handle exceptions and errors in a way that does not expose sensitive information.

1. Documentation

* The system should include clear documentation for developers, explaining the architecture, design decisions, and how to extend or modify the code.

1. Testability

* The system should be designed in a way that allows for easy unit and integration testing.
* Critical components of the system should have corresponding tests to ensure correctness.