REVISION HISTORY

Date	Version	Description	Author
06.04.2025	0.1	UML Class Diagram of the RMS software has been drawn and added in Section 2.	Emre Tekin
06.04.2025	0.2	Descriptions and various diagrams of the View component have been added in Sections 3.1.1, 3.1.2, 3.1.3 and 3.1.4.	Emre Tekin
06.04.2025	0.3	Sections 3.2.1, 3.2.4, and 3.3.2 have been written. Section 4 (COTS Identification) is added.	Elifnur Boncuk
06.04.2025	0.4	Added a sequence diagram and explanation for Section 3.2.3.	Elifnur Boncuk
06.04.2025	0.5	Descriptions and various diagrams of the Model component have been added in Sections 3.3.1, 3.3.2, 3.3.3 and 3.3.4.	Mert Turan Tahsin Karcı
06.04.2025	0.6	In section 3.2.2 Component Design description diagrams have been added.	İris Akdemir İrem Akova
06.04.2025	1.0	Last touch has been made into the document.	Tahsin Karcı
13.04.2025	1.1	Sections 3.1.1, 3.1.2 and 3.1.3 have been revised.	Emre Tekin
13.04.2025	1.2	Updated Section 3.2.1: UML diagram added; revised explanation text.	Elifnur Boncuk
13.04.2025	1.3	Sections 3.31, 3.3.2, 3.3.3 have been revised, necessary diagrams have been added	Mert Turan Tahsin Karcı
13.04.2025	1.4	Section 3.2.2 have been revised, necessary changes have been made.	İris Akdemir İrem Akova
14.04.2025	2.0	Last touch has been made into the document.	Tahsin Karcı

TABLE OF CONTENTS

1 Introduction

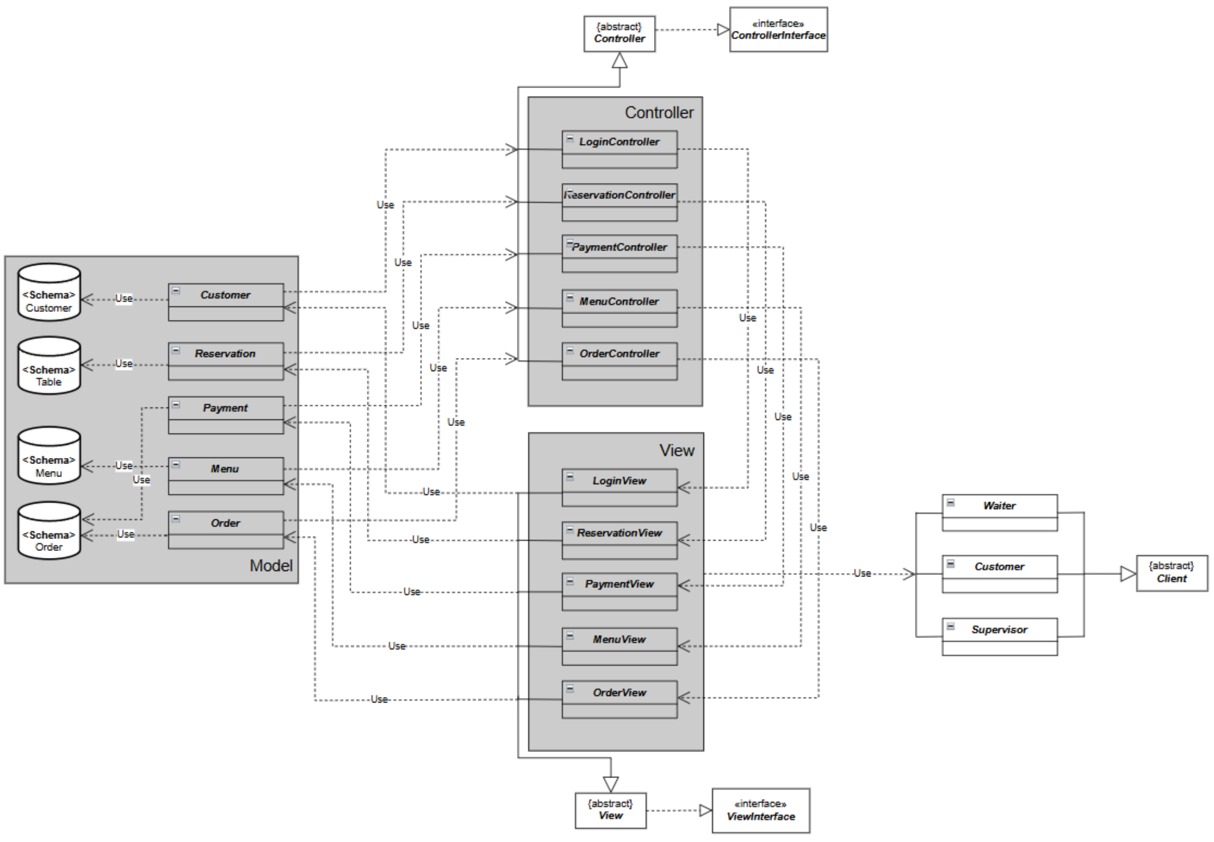
This document describes the design of the RMS software system.

1.1 *References*

1.1.1 Project References

#	Document Identifier	Document Title
[SRS]	RMS-SRS-1	RMS Software Requirements Specifications

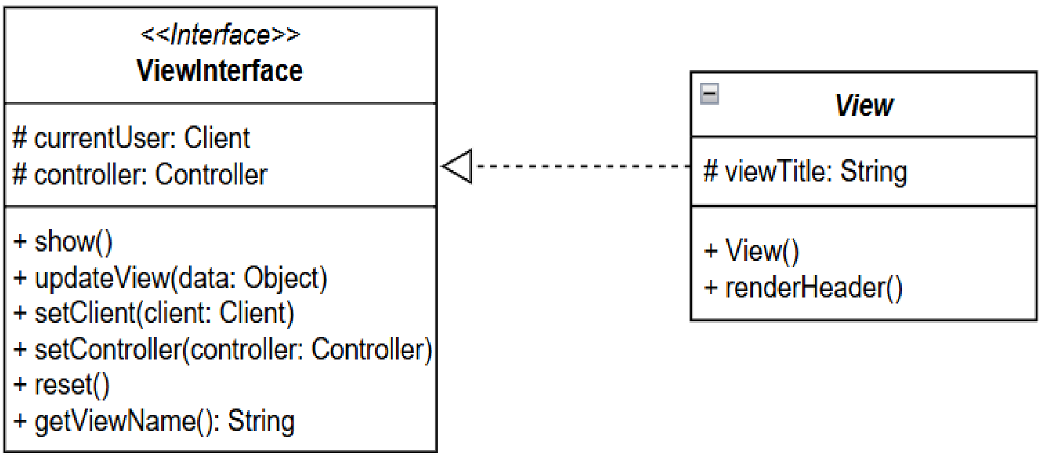
2 Software Architecture overview



3 Software design description

3.1 View

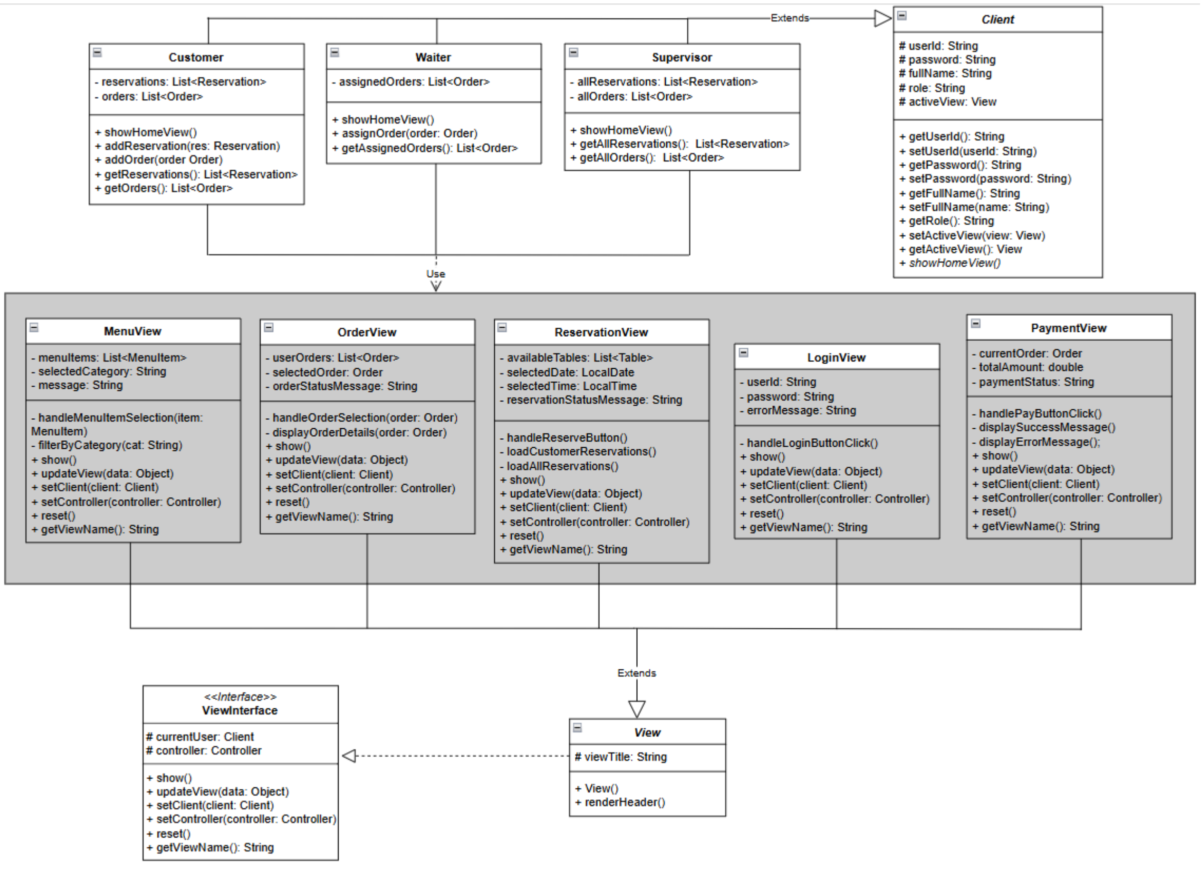
3.1.1 Component interfaces



The ViewInterface defines the external interface of the View component, ensuring consistent methods across all views in the RMS. These methods include show(), updateView(Object data), setClient(Client client), setController(Controller controller), reset(), and getViewName().

The abstract View class implements shared logic such as managing the current client and controller reference, resetting view state, and rendering headers. Input data includes user context (Client), model data (Object), and optionally view titles. Output consists primarily of UI display and user interaction. This design supports clear separation of responsibilities and simplifies the development of role-specific views.

3.1.2 Component design description


Extends
Client
# userId: String
# password: String
# fullName: String
# role: String
# activeView: View
+ getUserId(): String
+ setUserId(userId: String)
+ getPassword(): String
+ setPassword(password: String)
+ getFullName(): String
+ setFullName(name: String)
+ getRole(): String
+ setActiveView(view: View)
+ getActiveView(): View
+ showHomeView()
Customer
- reservations: List<Reservation>
- orders: List<Order>
+ showHomeView()
+ addReservation(res: Reservation)
+ addOrder(order Order)
+ getReservations(): List<Reservation>
+ getOrders(): List<Order>
Waiter
- assignedOrders: List<Order>
+ showHomeView()
+ assignOrder(order: Order)
+ getAssignedOrders(): List<Order>
Supervisor
- allReservations: List<Reservation>
- allOrders: List<Order>
+ showHomeView()
+ getAllReservations(): List<Reservation>
+ getAllOrders(): List<Order>
Use
MenuView
- menuItems: List<MenuItem>
- selectedCategory: String
- message: String
- handleMenuItemSelection(item: MenuItem)
- filterByCategory(cat: String)
+ show()
+ updateView(data: Object)
+ setClient(client: Client)
+ setController(controller: Controller)
+ reset()
+ getViewName(): String
OrderView
- userOrders: List<Order>
- selectedOrder: Order
- orderStatusMessage: String
- handleOrderSelection(order: Order)
- displayOrderDetails(order: Order)
+ show()
+ updateView(data: Object)
+ setClient(client: Client)
+ setController(controller: Controller)
+ reset()
+ getViewName(): String
ReservationView
- availableTables: List<Table>
- selectedDate: LocalDate
- selectedTime: LocalTime
- reservationStatusMessage: String
- handleReserveButton()
- loadCustomerReservations()
- loadAllReservations()
+ show()
+ updateView(data: Object)
+ setClient(client: Client)
+ setController(controller: Controller)
+ reset()
+ getViewName(): String
LoginView
- userId: String
- password: String
- errorMessage: String
- handleLoginButtonClick()
+ show()
+ updateView(data: Object)
+ setClient(client: Client)
+ setController(controller: Controller)
+ reset()
+ getViewName(): String
PaymentView
- currentOrder: Order
- totalAmount: double
- paymentStatus: String
- handlePayButtonClick()
- displaySuccessMessage()
- displayErrorMessage();
+ show()
+ updateView(data: Object)
+ setClient(client: Client)
+ setController(controller: Controller)
+ reset()
+ getViewName(): String
Extends
<<Interface>>
ViewInterface
# currentUser: Client
# controller: Controller
+ show()
+ updateView(data: Object)
+ setClient(client: Client)
+ setController(controller: Controller)
+ reset()
+ getViewName(): String
View
# viewTitle: String
+ View()
+ renderHeader()

The `Client` class represents a generic user who interacts with the RMS. It contains shared attributes and methods for all user types (authentication, identity, etc.) and provides polymorphic behavior where needed by the UI or controller logic.

The **View component** in the Restaurant Management System is responsible for rendering the user interface and displaying relevant data to the user based on their role (Customer, Waiter, or Supervisor). Each view class inherits common functionality from the abstract View class and implements the `ViewInterface`, ensuring consistency in UI operations.

LoginView handles user authentication. It collects login credentials and communicates with the `LoginController` to validate the user. It displays error messages if login fails and resets input fields after each attempt.

ReservationView manages the reservation interface. For Customers, it allows creating reservations by selecting dates and available tables. For Supervisors, it displays existing reservations. It updates the view dynamically based on user role and reservation status.

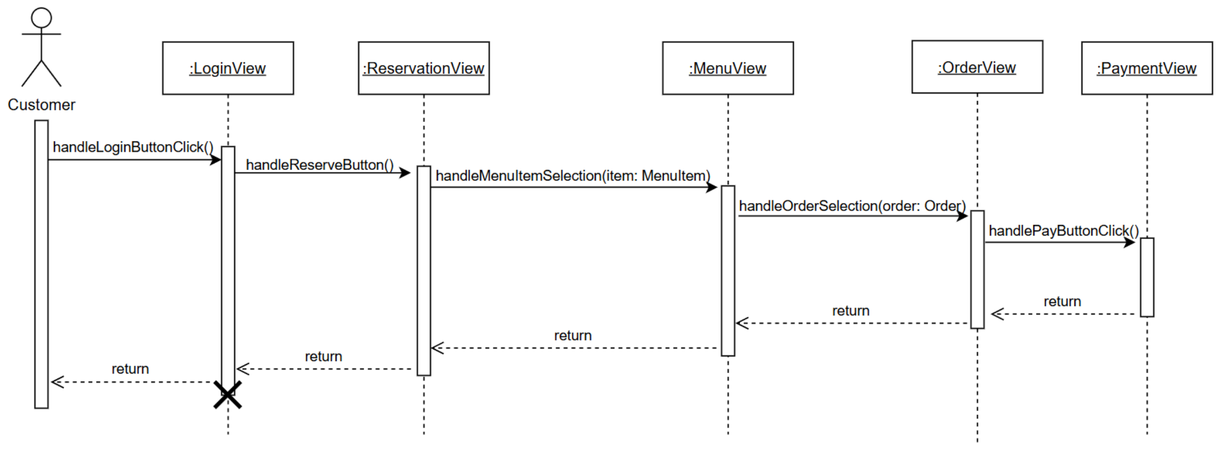
PaymentView displays payment details for completed orders. It retrieves the total amount, provides payment options, and shows success or failure messages. It works closely with the `PaymentController` to complete the transaction.

MenuView renders the restaurant's menu. It shows categorized food items and enables users to browse or filter items. It may allow selection of menu items to initiate an order depending on the user's role (e.g., Waiter or Customer).

OrderView displays current or past orders. For Customers, it shows personal orders; for Waiters or Supervisors, it can show a broader list. It provides detailed views of each order and updates dynamically as new orders are placed or modified.

3.1.3 Workflows and algorithms

Below, there is a Sequence Diagram showing an example workflow within the View component when a Customer enters the system as a Client.



3.1.4 Software requirements mapping

3.1.5 LoginView handles:

SRS-RMS-001: RMS shall enable customers, supervisors and waiters to log into the system with userID and a password.

SRS-RMS-001.1: The system shall prevent users with non-existent IDs or incorrect passwords from logging in.

ReservationView handles:

SRS-RMS-002: RMS shall enable customers to check the availability of the tables for the specified date and time.

SRS-RMS-003: RMS shall enable customers to reserve tables in the restaurant for the specified date and time.

SRS-RMS-002.1: RMS shall enable customers to view the availability of tables for a specified date and time, up to one week in advance.

SRS-RMS-003.1: RMS shall prevent customers from making reservations for tables that are not available in the specified time.

PaymentView handles:

SRS-RMS-008: RMS shall enable customers to pay for their meal from the system.

SRS-RMS-009: RMS shall enable supervisors to close the paid orders.

SRS-RMS-008.1: RMS shall ensure that the total payment amount will be calculated according to the customer's selections.

SRS-RMS-009.1: The system shall prevent the supervisor from closing an order if the customer does not pay.

MenuView handles:

SRS-RMS-005: RMS shall enable customers to view the menu.

SRS-RMS-005.1: RMS shall generate a display of menu to customer with prices.

OrderView handles:

SRS-RMS-006: RMS shall enable customers to place an order from the menu.

SRS-RMS-007: RMS shall enable the waiter and supervisor to view the status of an order.

SRS-RMS-007.1: RMS shall send notifications to the waiter and supervisor for orders and reservations.

3.2 Controller

3.2.1 Component interfaces

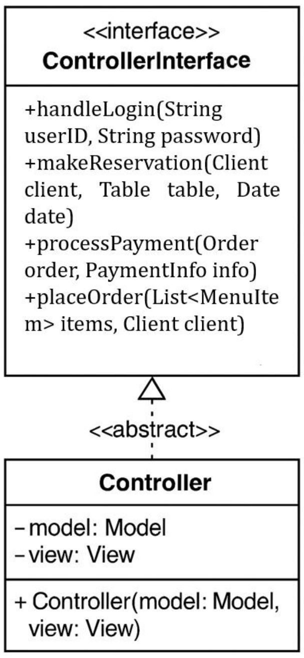
The Controller component manages the flow of data between the View and the Model. It receives inputs from the View layer, processes them according to the business logic, and updates the Model as needed.

Each specific controller class (e.g., LoginController, ReservationController, PaymentController) includes relevant methods to carry out its responsibilities. For example:

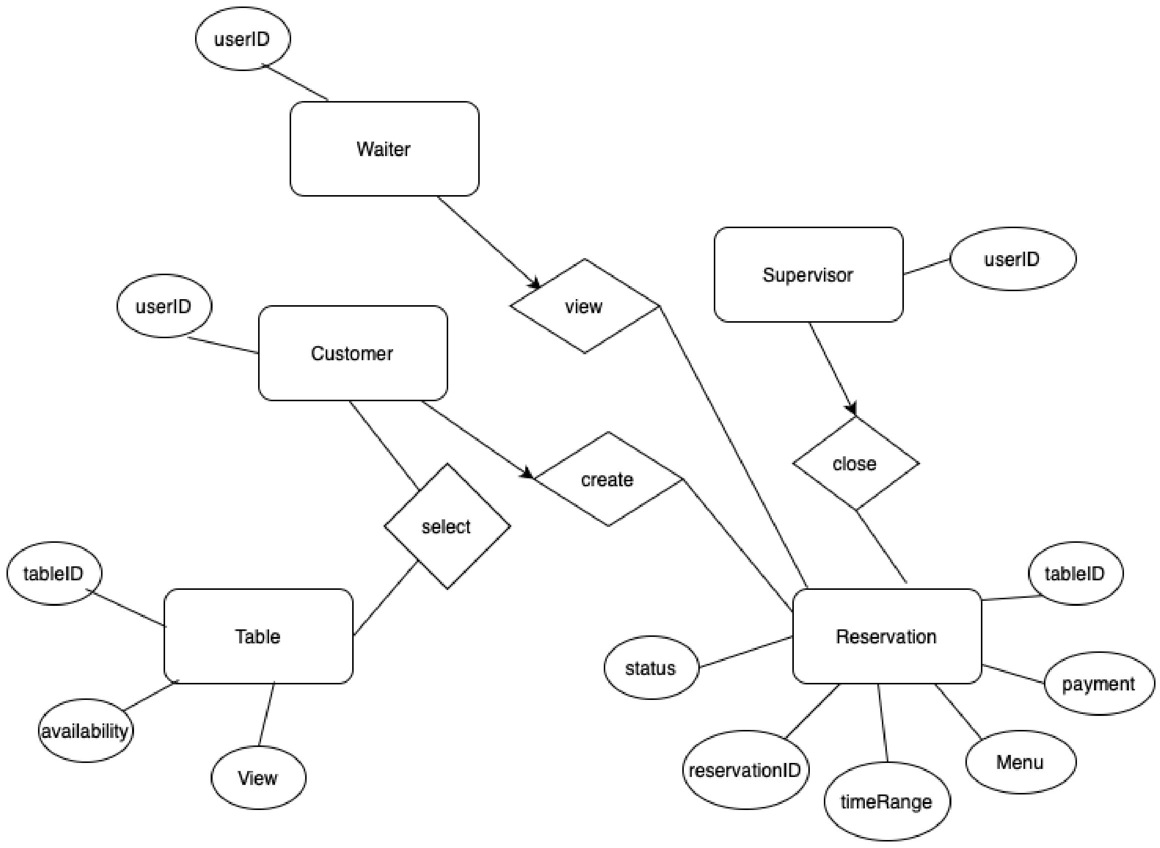
- handleLogin(String userID, String password)
- makeReservation(Client client, Table table, Date date)
- processPayment(Order order, PaymentInfo info)
- placeOrder(List<MenuItem> items, Client client)

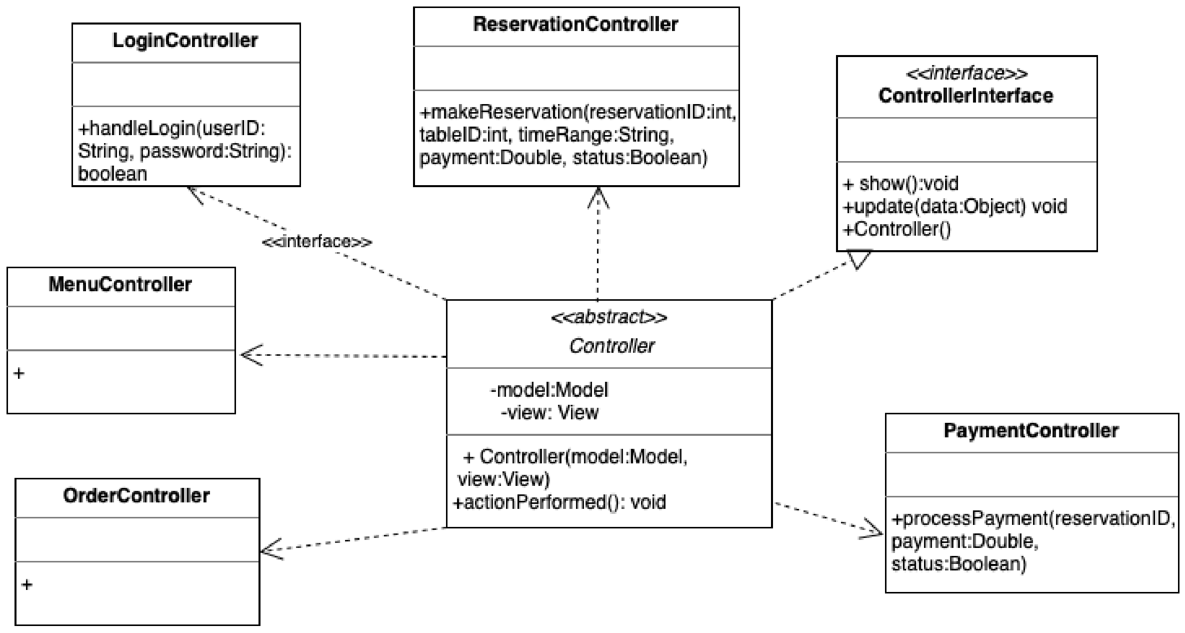
These operations are defined in a shared ControllerInterface and partially implemented in an abstract BaseController, which also handles common attributes like the Client object.

The UML diagram below illustrates the interface, the abstract base class.

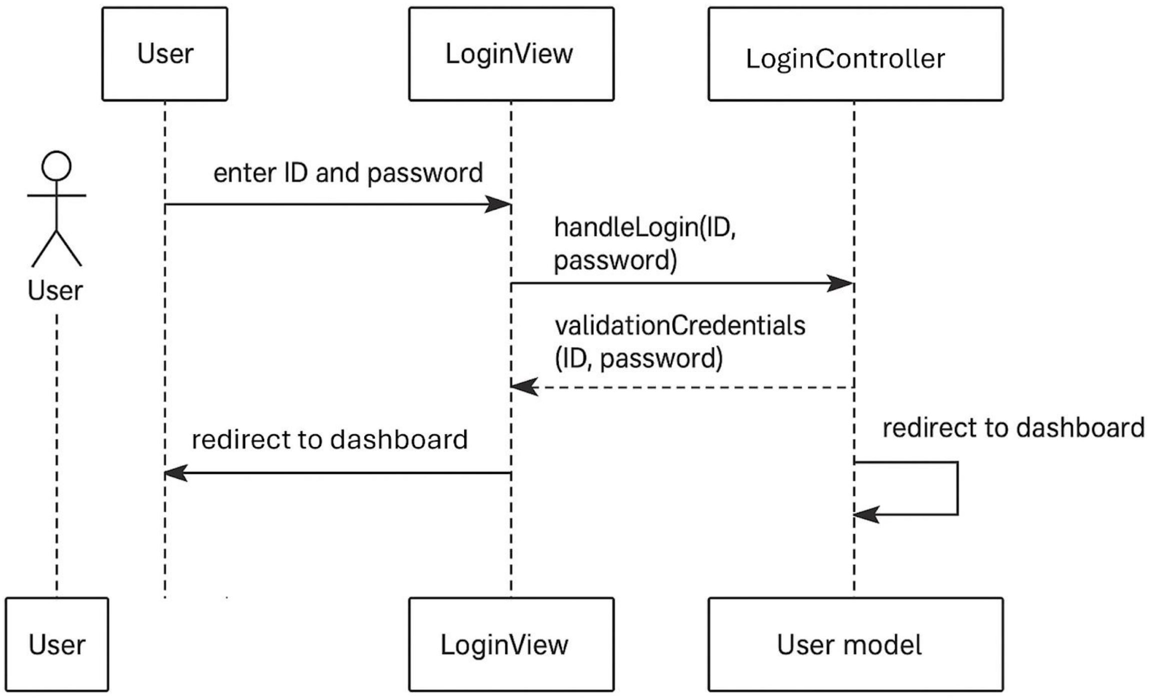

<<interface>>
ControllerInterface
+handleLogin(String userID, String password)
+makeReservation(Client client, Table table, Date date)
+processPayment(Order order, PaymentInfo info)
+placeOrder(List<MenuItem> items, Client client)
<<abstract>>
Controller
–model: Model
–view: View
+ Controller(model: Model, view: View)

3.2.2 Component design description




LoginController
+handleLogin(userID: String, password:String): boolean
ReservationController
+makeReservation(reservationID:int, tableID:int, timeRange:String, payment:Double, status:Boolean)
<<interface>>
ControllerInterface
+ show():void
+update(data:Object) void
+Controller()
<<interface>>
MenuController
+
<<abstract>>
Controller
-model:Model
-view: View
+ Controller(model:Model, view:View)
+actionPerformed(): void
PaymentController
+processPayment(reservationID, payment:Double, status:Boolean)
OrderController
+

3.2.3 Workflows and algorithms



This diagram shows the user login process in the RMS system. The user enters his/her credentials into the LoginView interface. This information is passed to the LoginController class. The Controller checks the information with the User model class. If the login is successful, the user is directed to the screen appropriate for his/her role.

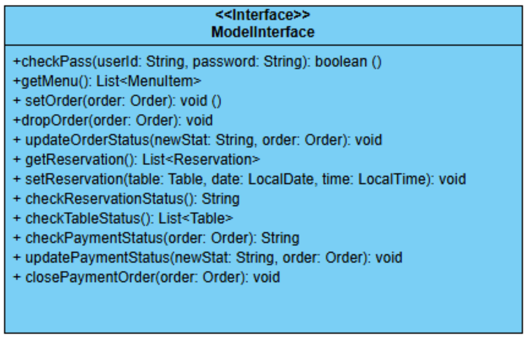
3.2.4 Software requirements mapping

- LoginController → SRS-RMS-001, SRS-RMS-001.1
- ReservationController → SRS-RMS-002, SRS-RMS-003, SRS-RMS-002.1, SRS-RMS-003.1
- OrderController → SRS-RMS-006, SRS-RMS-007, SRS-RMS-007.1
- PaymentController → SRS-RMS-008, SRS-RMS-008.1, SRS-RMS-009, SRS-RMS-009.1

3.3 Model

3.3.1 Component interfaces

The Model holds the data of the RMS from the database and uses the business logic to generate final acts. The Models data is used in the View components. The objects inside the Model component access the database and use the data as an attribute. The Model also uses the data from the Controller component to alter its attributes.



```
<<Interface>>
ModelInterface
+checkPass(userId: String, password: String): boolean ()
+getMenu(): List<MenuItem>
+ setOrder(order: Order): void ()
+dropOrder(order: Order): void
+ updateOrderStatus(newStat: String, order: Order): void
+ getReservation(): List<Reservation>
+ setReservation(table: Table, date: LocalDate, time: LocalTime): void
+ checkReservationStatus(): String
+ checkTableStatus(): List<Table>
+ checkPaymentStatus(order: Order): String
+ updatePaymentStatus(newStat: String, order: Order): void
+ closePaymentOrder(order: Order): void
```

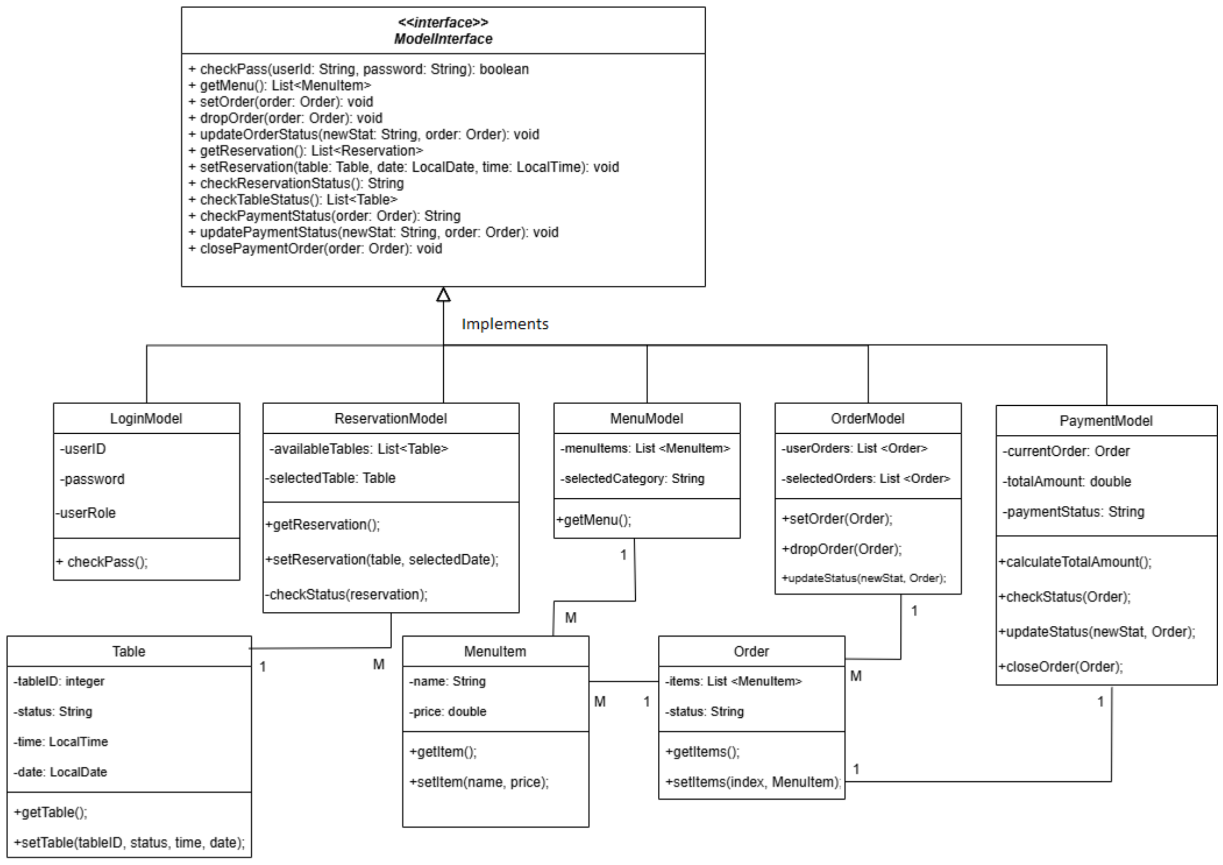
3.3.2 Component design description

Login:
Users authenticate using a userID and password. Upon successful authentication, the role determines the available functionalities.

Reservation:
Customers can view available tables and make reservations. Supervisors have the ability to view all reservations, while waiters can manage the status of reservations.

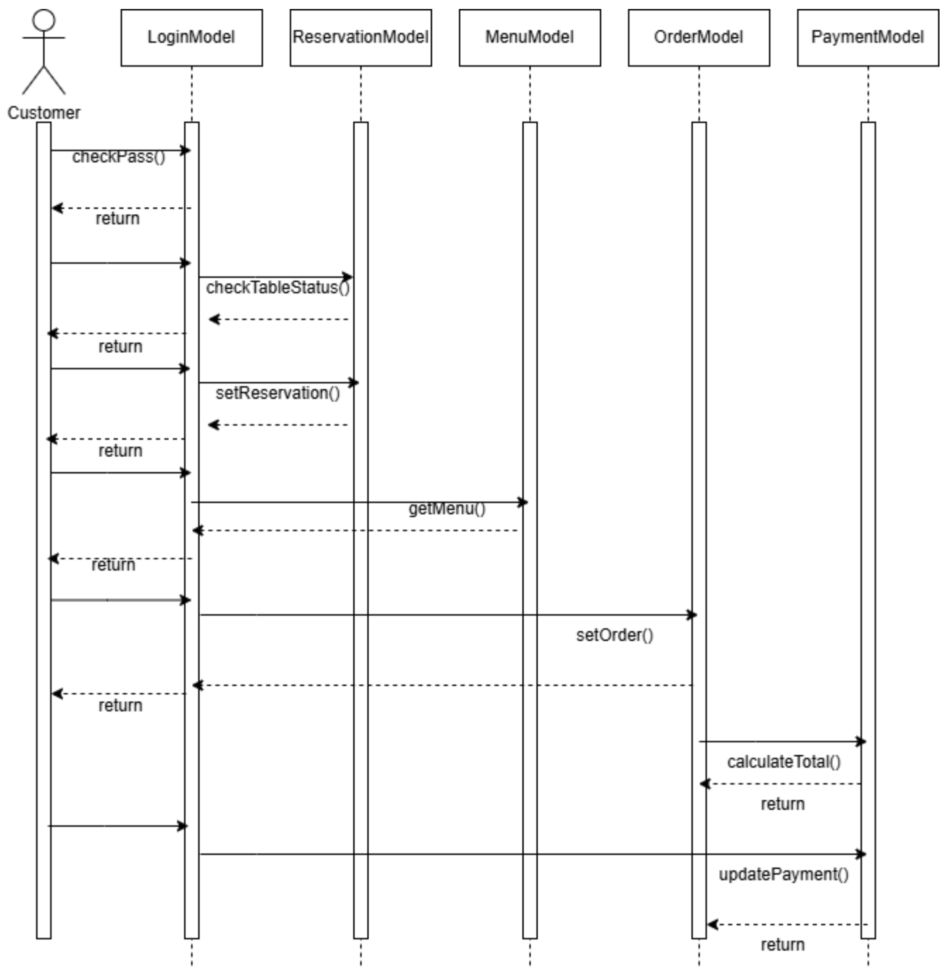
Menu and Order Management:
Customers can browse the menu and place orders, while waiters and supervisors can view order statuses. Order status is updated dynamically based on customer actions and backend processes.

Payment Processing:
Once an order is completed, customers can make payments, and supervisors can close the order after confirming the payment status.


<<interface>>
ModelInterface
+ checkPass(userId: String, password: String): boolean
+ getMenu(): List<MenuItem>
+ setOrder(order: Order): void
+ dropOrder(order: Order): void
+ updateOrderStatus(newStat: String, order: Order): void
+ getReservation(): List<Reservation>
+ setReservation(table: Table, date: LocalDate, time: LocalTime): void
+ checkReservationStatus(): String
+ checkTableStatus(): List<Table>
+ checkPaymentStatus(order: Order): String
+ updatePaymentStatus(newStat: String, order: Order): void
+ closePaymentOrder(order: Order): void
Implements
LoginModel
-userID
-password
-userRole
+ checkPass();
ReservationModel
-availableTables: List<Table>
-selectedTable: Table
+getReservation();
+setReservation(table, selectedDate);
-checkStatus(reservation);
MenuModel
-menuItems: List <MenuItem>
-selectedCategory: String
+getMenu();
OrderModel
-userOrders: List <Order>
-selectedOrders: List <Order>
+setOrder(Order);
+dropOrder(Order);
+updateStatus(newStat, Order);
PaymentModel
-currentOrder: Order
-totalAmount: double
-paymentStatus: String
+calculateTotalAmount();
+checkStatus(Order);
+updateStatus(newStat, Order);
+closeOrder(Order);
Table
-tableID: integer
-status: String
-time: LocalTime
-date: LocalDate
+getTable();
+setTable(tableID, status, time, date);
MenuItem
-name: String
-price: double
+getItem();
+setItem(name, price);
Order
-items: List <MenuItem>
-status: String
+getItems();
+setItems(index, MenuItem);
1
M
1
M
M
1
1
M
1
1

3.3.3 Workflows and algorithms

Below, there is a Sequence Diagram showing an example workflow within the Model component when a Customer enters the system as a Client.



3.3.4 Software requirements mapping

- LoginModel → SRS-RMS-001, SRS-RMS-001.1
- ReservationModel → SRS-RMS-002, SRS-RMS-003, SRS-RMS-002.1, SRS-RMS-003.1
- MenuModel→SRS-RMS-005, SRS-RMS-005.1
- OrderModel → SRS-RMS-006, SRS-RMS-007, SRS-RMS-007.1
- PaymentModel→ SRS-RMS-008, SRS-RMS-008.1, SRS-RMS-009, SRS-RMS-009.1

4 COTS Identification

COTS (commercial of the shelf) libraries used in RMS are the following:

- **Java AWT & Swing** – for GUI rendering, License: Oracle Binary Code License
- **JDK 21** – Java Development Kit, License: GNU General Public License
- **SQL (MySQL)** – database, License: GPL v2
- **Docker 26.1.1** – for containerized deployment, License: Apache License 2.0
- **IntelliJ IDEA 2024.1** – IDE (community version), License: Apache License 2.0
- **GitHub** – version control and collaboration