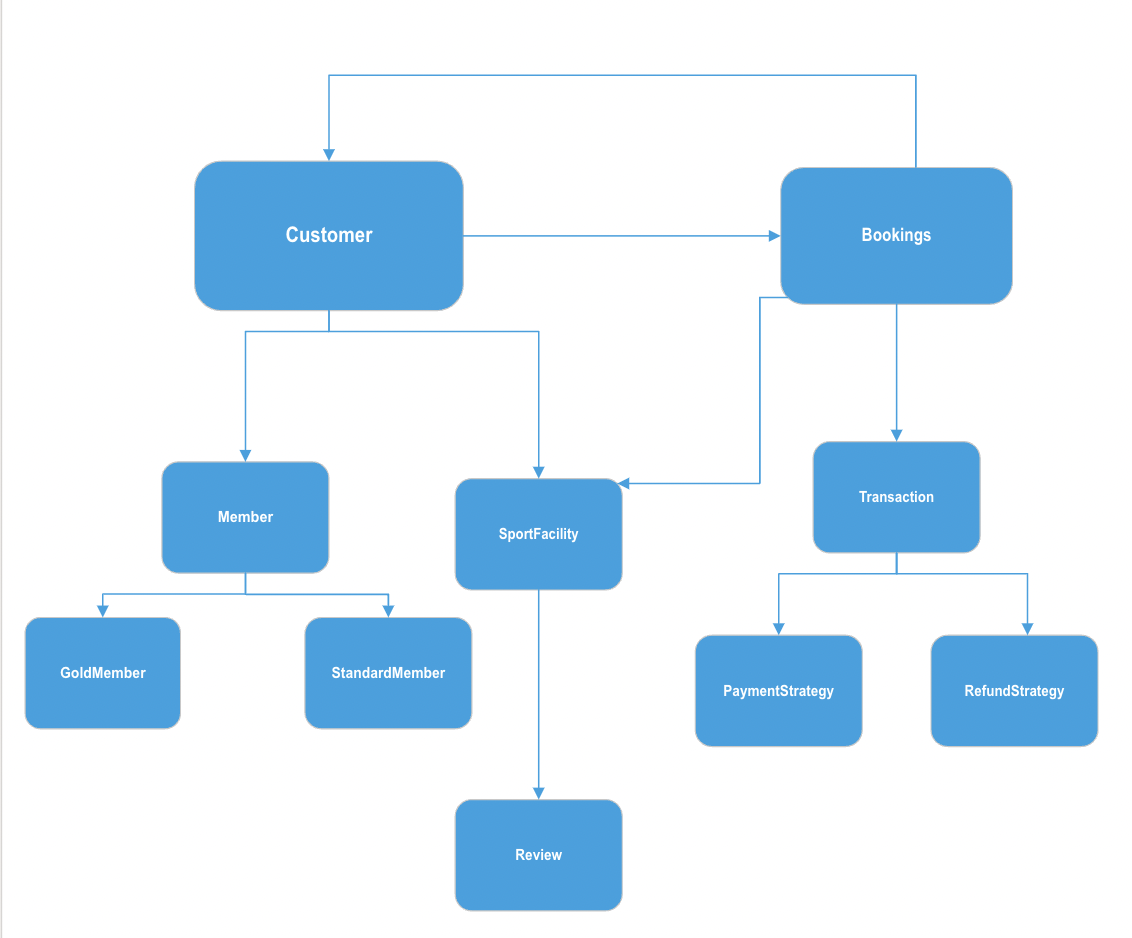
****

# Module Organisation



The diagram above shows the module organization and hierarchy that represents our Sports Facility Booking System. Using this hierarchy as a reference and visualizing which class invokes which other classes, we have easily been able to develop a testing strategy.

# Testing Strategy

We have adopted bottom-up testing as the testing strategy for our project. We chose this strategy particularly because of the following reasons:

* Early Identification of Issues

By starting with the testing of low-level components, any problems or issues can be detected and rectified at the early stages of the project development, preventing them from growing into major issues and bugs when combined with other components.

* Use of Test Stubs

In the bottom-up approach, the absence of test stubs for simulating lower-level classes allowed for thorough testing of the entire system before progressing to integration/system testing. This ensured comprehensive coverage of all system aspects.

* Collaborative Effort in Bottom-Up Testing

In this approach, the task of testing and planning was efficiently managed. Each team member focused on developing tests for the classes they were responsible for. Thanks to the bottom-up methodology, we were able to start testing the fundamental classes right after their development was completed.

* Parallelism

In our project, we assigned each group member a specific component or class to develop.

Therefore, the bottom-up testing approach has allowed us to code and test our allocated components independently before combining the components together as the whole system.

We began our testing process by carrying out **unit testing** for the following classes:

* Member.java
* Gold.java
* StandardMember.java
* SportFacility.java
* Review.java
* CreditCardPayment.java
* CreditCardRefund.java
* CreditCardRefund.java
* PayPalPayment.java
* PayPalRefund.java

After carrying out unit testing, we then proceeded to carry out **integration testing** on the following components:

* User.java +Session.java+SessionManager.java+AuthenticationService.java
* SportFacility.java + Review.java

- Transaction.java + CreditCardPayment.java, CreditCardRefund.java +CreditCardRefund.java + PayPalPayment.java + PayPalRefund.java

- Customer.java + Admin.java + Member.java + GoldMember.java + StandardMember.java + Bookings.java + Review.java + SportFacility.java + Transaction.java + PaymentStrategy.java + RefundStrategy.java

Finally, after successfully carrying out the system and integration testing, we moved forward with **system testing** where we tested the interactions of all the classes in the system altogether. To conduct testing, we devised comprehensive test scenarios involving different use cases that the end-users would encounter. These test scenarios have helped us replicate real-world situations and have allowed us to verify and validate how the system would respond to those situations.

# 

# **Test Cases**

**Member.java, Gold.java, StandardMember.java (Unit Tested)**

| **Test Case ID** | **Input Description** | **Expected result** | **Actual Output** |
| --- | --- | --- | --- |
| testCustomer\_01 | Calculates the discount rate for a Gold Member, based on the loyalty points of the customer. | 7 | Same as expected result |
| testCustomer\_02 | Calculates the discount rate for a Standard Member, based on the loyalty points of the customer. | 1 | Same as expected result |

**SportFacility.java (Unit Tested)**

| **Test Case ID** | **Input Description** | **Expected result** | **Actual Output** |
| --- | --- | --- | --- |
| testSportFacility\_01 | Book a facility | "Booking Successful for facility 13-02-2000 10\n" | Same as expected result |
| testSportFacility\_02 | Book a time slot that is already taken | "Sorry, this time slot is already booked.\nYou have been added to the waitlist, if the booker cancels their booking, you would be notified. Thanks.\n" | Same as expected result |
| testSportFacility\_03 | Book a time before facility is open | "Sorry, the facility is not open at this time\n" | Same as expected result |
| testSportFacility\_04 | Book a facility after closing time | "Sorry, the facility is closed during this time\n" | Same as expected result |
| testSportFacility\_05 | Book facility in the past | "Date is in the past. Try again.\n" | Same as expected result |
| testSportFacility\_06 | Book a facility but out of the 24H format. Example: -1, 25 | "Please put time in hours only (0-23).\n" | Same as expected result |
| testSportFacility\_07 | Cancel booking that doesn’t exist | "No booking found for 13-02-2000 10. Unable to cancel.\n" | Same as expected result |
| testSportFacility\_08 | Trying to cancel the same booking twice | "The time slot for 13-02-2000 11 is already free.\n" | Same as expected result |
| testSportFacility\_09 | Booking successfully canceled for a specific date | "Booking canceled for 13-02-2000 12\n" | Same as expected result |
| testSportFacility\_10 | Input date string is wrong | "Please Input Correct Date!\n" | Same as expected result |
| testSportFacility\_12 | Booking a slot, when someone else booked it. | "TennisCourt can be booked for time 06-12-2023 on first come first served basis!\n" | Same as expected result |

**Review.java (Unit Tested)**

| **Test Case ID** | **Input Description** | **Expected result** | **Actual Output** |
| --- | --- | --- | --- |
| testReview\_01 | Place a new review:Great facility! | "Great facility!" | Same as expected |
| testReview\_02 | Give a rating of 4 | 4 | Same as expected |
| testReview\_03 | Giving rating out of bound | "Rating must be between 1 and 5.\n" | Same as expected |
| testReview\_04 | Checks the String representation of a Review object. | "Review{comment='Testing Review', rate=1}” | Same as expected |

**CreditCardPayment.java, CreditCardRefund.java, PayPalPayment.java, PayPalRefund.java**

**(Unit Tested)**

| **Test Case ID** | **Input Description** | **Expected result** | **Actual Output** |
| --- | --- | --- | --- |
| creditCardPayment\_processPayment | Pay with credit card of amount 100 | "Visa processed the payment with amount of 100.00.\n" | Same as expected |
| creditCardRefund\_processRefund() | Refund for credit card of payment 100 | "Visa processed the refund with amount of 100.00.\n" | Same as expected |
| payPalPayment\_processPayment() | Pay with paypal of amount 100 | "PayPal processed the payment with amount of 100.00.\n" | Same as expected |
| payPalRefund\_processRefund() | Refund for paypal of payment 100 | "\nPayPal processed the refund with amount of 100.00.\n" | Same as expected |

**User.java (Unit Tested)**

| **Test Case ID** | **Input Description** | **Expected result** | **Actual Output** |
| --- | --- | --- | --- |
| testCustomer\_14 | Tests whether or not the user is null | false | Same as expected |

**SportFacility.java + Review.java**

**(Integration Tested)**

| **Test Case ID** | **Input Description** | **Expected result** | **Actual Output** |
| --- | --- | --- | --- |
| testSportFacility\_11 | Adding facility and showing review from the facility | "Review{comment='Good', rate=4}\n" | Same as expected result |

**Transaction.java + CreditCardPayment.java, CreditCardRefund.java +CreditCardRefund.java + PayPalPayment.java + PayPalRefund.java**

**(Integration Tested)**

| **Test Case ID** | **Input Description** | **Expected result** | **Actual Output** |
| --- | --- | --- | --- |
| testPayPal | Payment is made, but later wanted refund. | "PayPal processed the payment with amount of %.2f.\n" + "\n" +  "PayPal processed the refund with amount of %.2f.\n" | Same as expected |
| testRefundPaymentNotProcessed | Payment failed but asking for refund | "Payment is not processed.\n" | Same as expected |
| testCreditCardPayment | Payed with credit card and asking for refund | "Visa processed the payment with amount of %.2f.\n" +  "Visa processed the refund with amount of %.2f.\n" | Same as expected |

**Customer.java + Admin.java + Member.java + GoldMember.java + StandardMember.java + Bookings.java + Review.java + SportFacility.java + Transaction.java + PaymentStrategy.java + RefundStrategy.java**

**(Integration Tested)**

| **Test Case ID** | **Input Description** | **Expected result** | **Actual Output** |
| --- | --- | --- | --- |
| testCustomer\_03 | The customer has requested to make a booking on an invalid date in the past. | false | Same as expected |
| testCustomer\_04 | The customer tries to cancel a previous booking, but the cancel request is declined because the current time is less than 6 hours away from the booking start time. | "Sorry, this booking cannot be cancelled and is therefore non-refundable.\n" + "A booking can only be cancelled 6 hours or more in advance.\n" | Same as expected |
| testCustomer\_05 | Calculate the percentage of total price a Standard Member pays after the discount strategy based on loyalty points and the number of bookings held has been applied. | 0.99 | Same as expected |
| testCustomer\_06 | Calculate the percentage of total price a Gold Member pays after the discount strategy based on loyalty points and the number of bookings held has been applied. | 0.88 | Same as expected |
| testCustomer\_07 | The customer views his past booking details and current membership state. | “Membership Type: Gold Member\nBooking ID: 1\nBooking Date: 22-11-2023\nBooking Start Time: 16\nFacility: Tennis\nBooking ID: 2\nBooking Date: 22-11-2023\nBooking Start Time: 10\nFacility: Tennis\n" | Same as expected |
| testCustomer\_08 | The customer tries to cancel a booking that is over 6 hours away from system clock. The booking is cancelled successfully.  The admin checks for the validity of this cancellation request and returns an integer that the *Main* class interprets accordingly to print the correct cancellation status. | 3 | Same as expected |
| testCustomer\_09 | The customer successfully provides a review and rating for a facility that he has booked previously. | true | Same as expected |
| testCustomer\_11 | Admin tries to add a facility that has already been added. | "The facility already exists!\n" | Same as expected |
| testCustomer\_12 | Admin compares a booking start time to check if it is in the future relative to the system clock. | false | Same as expected |
| testCustomer\_13 | Admin compares a booking start time to check if it is in the past relative to the system clock. | true | Same as expected |
| testCustomer\_15 | Customer tries to check any notice of cancelled bookings; the customer will only get a notification if they were on a waitlist for the same timeslot and facility as the original booking holder. | "Sorry, there is no record of any cancelled bookings.\n" | Same as expected |
| testCustomer\_16 | Customer tries to check for any notification of cancelled bookings after trying to trying to book an already booked slot and the original booking holder cancels their booking. | "Tennis can be booked for time 06-12-2023 on first come first served basis!\n” | Same as expected |
| testCustomer\_17 | Check if two customer accounts are actually the same account by comparing the usernames and passwords. | true | Same as expected |
| testCustomer\_18 | Customer tries to view any past/completed bookings. | "Membership Type: Standard Member\n"  + "Booking ID: 1\n"  + "Booking Date: 22-11-2024\n" + "Booking Start Time: 16\n" + "Facility: Tennis\n" +"\n" | Same as expected |
| testCustomer\_19 | Admin tries to check for the maximum between current system time hour and facility opening hour.  This will be used by the SportsFacility class to display the available slots from only later in the day. | 25 | Same as expected |

**User.java +Session.java+SessionManager.java+AuthenticationService.java**

**(Integration Tested)**

| **Test Case ID** | **Input Description** | **Expected result** | **Actual Output** |
| --- | --- | --- | --- |
| testSessionValidity | Check to see if the session is valid when the user is logged in. | true | Same as expected result |
| testGetUser | Getter method for the User attribute of Session. | User attribute of Session | Same as expected result |
| testCreateSession | Checks to see if a Session is successfully created, and then the length of the sessionMappingList attribute in SessionManager is checked to confirm addition of Session. | 1 | Same as expected result |
| testRemoveSessionWithExistingUser | Remove a Session from the sessionMappingList attribute in SessionManager that corresponds to a user that exists. | true | Same as expected result |
| testRemoveSessionWithNonExistingUser | Remove a Session from the sessionMappingList attribute in SessionManager that corresponds to a user that does not exist. | false | Same as expected result |
| testRemoveSessionWithInvalidSession | Remove a Session from the sessionMappingList attribute in SessionManager that corresponds to a user who is logged out. | false | Same as expected result |
| testRemoveSessionWithMultipleSessions | Remove a Session from the sessionMappingList attribute in SessionManager when there are multiple sessions. | true | Same as expected result |
| testLoginWithValidCredentials\_ReturnsCustomer | When an existing customer logs in with valid credentials. The method tests if the customer returned is not null and has the same user name and password as existing user | Not null, "existingUser", "Abcdefg1" | Same as expected result |
| testLoginWithInvalidCredentials\_ReturnsNull | When a customer tries to log in with invalid credentials, the login process should be unsuccessful | null | Same as expected result |
| testLogoutWithLoggedInCustomer | When a customer who is already logged in, tries to log out | true | Same as expected result |
| testLogoutWithLoggedOutCustomer | When a customer is already logged out, and they try to log out again, the log out is unsuccessful | false | Same as expected result |
| testRegisterNewUserWithValidPassword | During registering, the customer provides a valid password | Not null, “newUser”, “Password123” | Same as expected result |
| testRegisterNewUserWithInvalidPassword | During registering, the customer provides an invalid password | null | Same as expected result |
| testRegsiterWhenUserIsAlreadyRegistered | When an existing customer tries to register again | null | Same as expected result |
| testValidatePasswordWithNoLowerCase | During registering, the customer provides invalid password with no lower case letters | false | Same as expected result |
| testValidatePasswordWithNoUpperCase | During registering, the customer provides invalid password with no upper case letters | false | Same as expected result |
| testValidatePasswordWithShortPassword | During registering, the customer provides invalid password with less than 8 characters | false | Same as expected result |
| testValidatePasswordWithNoNumber | During registering, the customer provides invalid password with no numbers | false | Same as expected result |

**Main.java + Customer.java + Admin.java + Member.java + GoldMember.java + StandardMember.java + Bookings.java + Review.java + SportFacility.java + Transaction.java + PaymentStrategy.java, RefundStrategy.java + AuthenticationService.java + Session.java + SessionManager.java**

**(System Testing)**

| **Test Case ID** | **Input Description** | **Expected result** | **Actual Output** |
| --- | --- | --- | --- |
| testMain\_01 | Sign in screen | Sign in screen displayed. | Same as expected result |
| testMain\_02 | Sign in as admin, then user signs in | Both sign ins are successful | Same as expected result |
| testMain\_03 | Admin creates Facility. User logs in and checks available slots for a date and books a facility and pays | Facility created, and slots successfully shown and booking has been placed | Same as expected result |
| testMain\_11 | Booked a lot of slots and facilities | Membership has been updated resulting in discount prices. | Same as expected result |
| testMain\_12 | Pay with wrong card and applying for a slot that is already booked | Asking for proper card and putting the customer in waitlist | Same as expected result |
| testMain\_14 | Inputting out of bound numbers | "Input is not a number. Try again.\n" | Same as expected result |
| testMain\_15 | Admin logs in and wants to add facility | Admin login is successful and is allowed to add facility | Same as expected result |
| testMain\_16 | Admin wants to change time due to timezone | Time change of the system is allowed and successful | Same as expected result |
| testMain\_19 | User logs in , views reviews and books | Reviews are shown and bookings are successful | Same as expected result |

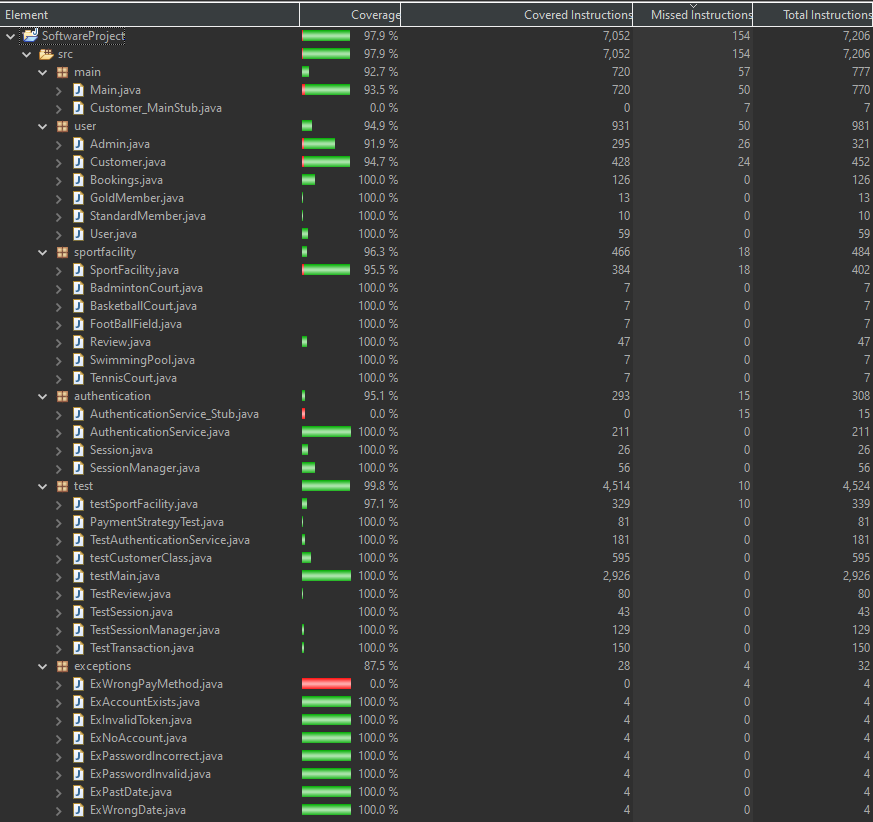
# 

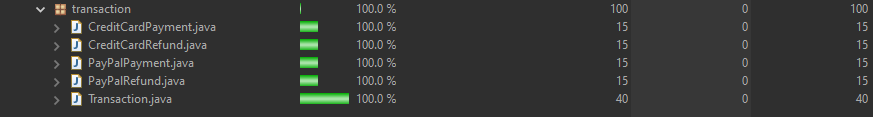
# Test Coverage

**Instruction coverage**

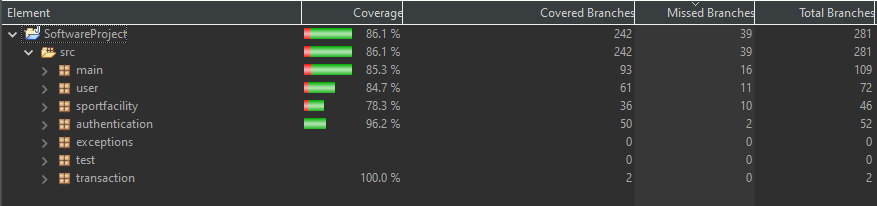
In our recent development efforts, we've reached an impressive 97% code coverage. This high percentage reflects our team's commitment to quality and thoroughness in testing. It means the vast majority of our codebase has been rigorously tested, significantly reducing the risk of bugs and ensuring the reliability and stability of our software.

There are some parts (2%) that weren’t tested. Thissmall fraction often represents complex, hard-to-test code. We maintain a focus on the quality of tests over merely achieving high coverage statistics. It is vital to ensure that the tests are meaningful and accurately assess the functionality of the code.





**Branch coverage CC/DC:**



# 

# Code Refactoring

Throughout the code we have applied various refactoring strategies. Here’s a few that we applied:

**1. Refactoring via Method Extraction**

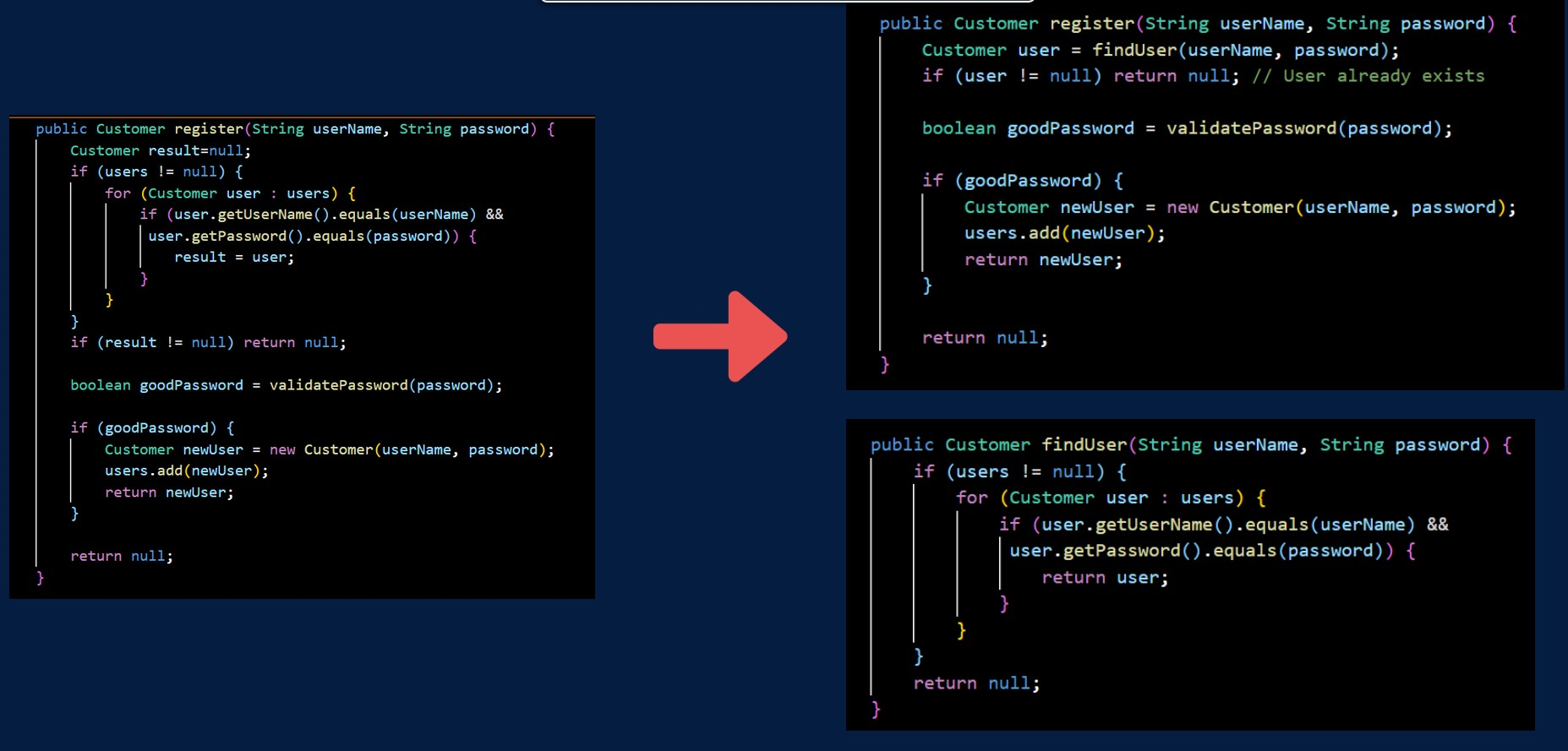
Our initial codebase featured a bulky method that was handling multiple responsibilities, making it challenging to understand, maintain, and test.

Strategy Implemented: We employed Method Extraction, a key refactoring technique. This involved breaking down the large method into smaller, more manageable sub-methods. Each sub-method was designed to perform a specific, well-defined task.

Impact:

Enhanced Testability: Smaller methods are easier to test as they typically have a single responsibility. This change has allowed us to write more focused unit tests, covering each aspect of the method's functionality.

Improved Modularity: By decomposing the method, we've increased the modularity of our codebase. This allows for easier maintenance and enhances the ability to reuse code across different parts of the application.

Example:  


**2. Optimizing Class Hierarchy by Moving Fields Upward**

In our original class hierarchy, multiple child classes shared common fields, leading to redundancy and potential inconsistency.

Strategy Implemented: We refactored the class hierarchy by relocating these common fields to the parent class. This technique is known as 'Moving Fields Upward'.

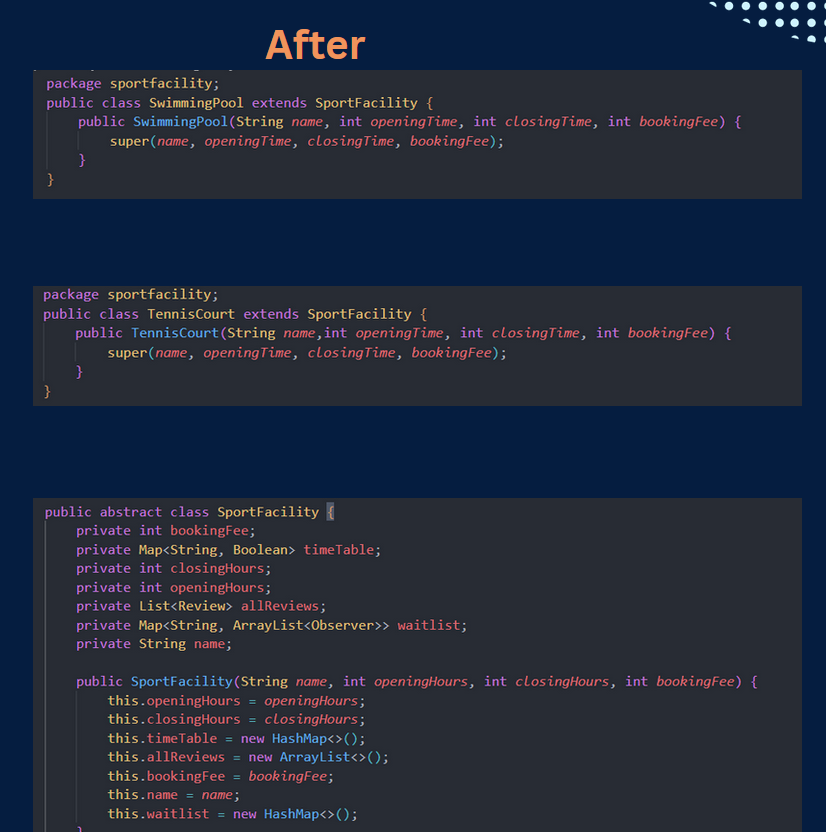
Impact:

Streamlined Structure: The parent class now serves as a more effective abstraction, encapsulating common properties. This makes our class hierarchy cleaner and more logical.

Reduced Redundancy: By eliminating repeated fields in child classes, we've reduced code duplication. This not only simplifies the code but also makes it easier to apply changes in the future since they need to be made in only one place.

Example:





**3. Improving Code Structure with Replace Temp with Query Refactoring**

In our existing codebase, we identified a pattern where temporary variables were being used extensively within methods. These temps often complicated the method's structure and made the code less clear.

Strategy Implemented: Replace Temp with Query

We implemented the 'Replace Temp with Query' refactoring technique. This involved replacing temporary variables with query methods. The key change was to remove the temporary variables and instead use methods that return the value needed for the computation.

Impact:

Enhanced Readability and Maintainability:

Elimination of Temporary Variables: By replacing temps with queries, we reduced the clutter in our methods, making them more concise and readable. This approach simplifies the code and makes it easier to understand.

Improved Method Coherence: Methods now focus more on the 'how' rather than the 'what'. This means each method is more about describing the process and less about managing the state, which aligns better with good object-oriented practices.

Easier Modification and Extension: With queries in place, modifying or extending our code becomes much simpler. If the logic for calculating a value changes, we only need to update it in one place – the query method – rather than in every place a temp was used.

Example:

