# CHAPTER ONE

## Requirement Analysis (use cases)

**Requirement analysis** is a critical phase in the systems engineering, software engineering, and project management processes. It involves understanding, documenting, and managing the needs and requirements of stakeholders for a particular project or system.

As a group of 3rd-year Software Engineering students at Wachemo University, we began our web-based e-commerce app project by conducting requirement analysis to understand the needs of our university community, including students and staff. Our aim was to create a platform accessible through a web browser, enabling users to buy and sell products within the Wachemo ecosystem. We gathered requirements through informal interviews with students and staff, a focus group with five classmates. This process helped us compile a comprehensive list of functional and non-functional requirements, which we documented as a team to guide the development of our web app.

Requirement Analysis of Wachemo University E-Commerce System (Wache Mareket):

**Functional Requirement:**

1. User Registration:

* Users should be able to register an account with the web-based e-commerce system by providing basic information, such as name, email, and contact details.
* The system should ensure secure storage and handling of user credentials and personal information in MySQL.

1. Browse and Search:

* Users should be able to browse through a variety of products listed by sellers within the Wachemo community, such as textbooks or electronics.
* Users should be able to search for products using categories like “Books,” “Clothing,” or “Gadgets.”

1. Seller Profiles:

* Each seller should have a dedicated profile page displaying their listed products, transaction history, and mode status (buyer/seller).
* Users should be able to view product details from sellers, including price, description, and availability status (e.g., sold out).
* The system should indicate whether a product is still available for purchase.

1. Ordering Process:

* Users in buyer mode should be able to select products, add them to a cart, and modify their selections (e.g., quantity) before finalizing the order.
* The system should calculate the total order cost, including any applicable fees or taxes.
* Users should be able to place orders directly through the web interface.

1. Secure Payment:

* The system should offer multiple payment options via an app wallet, including Telebirr, CBE, Amole, and M-Pesa.
* Users should be able to deposit and withdraw money from their wallet using these Ethiopian payment methods.
* The system should encrypt and securely store payment information, generating transaction records in MySQL.

1. Product Management (Seller Mode):

* Sellers should be able to add new products to the web app, providing details like name, price, and category.
* Sellers should have the option to delete their product listings when no longer available.
* The system should allow sellers to view the status of their products (e.g., sold out) and update records accordingly.

1. Transaction Tracking:

* Users should be able to view their transaction history, including past purchases and sales, through the web app.
* The system should update transaction records in MySQL after each completed order or sale.

1. Account Management:

* Users should be able to manage their account settings, updating personal information, email, and payment methods.
* The system should allow users to toggle between buyer and seller modes seamlessly via the web interface.

1. System Administration:

* The system should include an administration portal to oversee user accounts, product listings, and transaction records.
* The system should validate emails, credentials, mode switches, process payments, and mark products as sold out automatically.

**Non-Functional Requirements:**

* Security: User data, transactions, and login sessions must be protected with MySQL encryption and safeguards against unauthorized access because of the need to protect sensitive information and prevent data breaches or session hijacking.
* Performance: Web pages must load within 3 seconds, even on university Wi-Fi, and support up to 100 simultaneous users without issues because of the need to ensure a responsive and scalable user experience during peak usage.
* Simplicity: The web interface must be intuitive for all users, regardless of technical skill, with key actions (e.g., adding to cart, toggling modes) requiring minimal steps because of the diverse user base at Wachemo and the need to enhance efficiency and satisfaction.
* Reliability: The system must maintain 99% uptime and handle errors (e.g., payment failures) with clear user feedback because of the requirement for consistent availability and trust in usability.

Scalability: The system must accommodate an increasing number of Wachemo users over time because of anticipated growth in the user base.

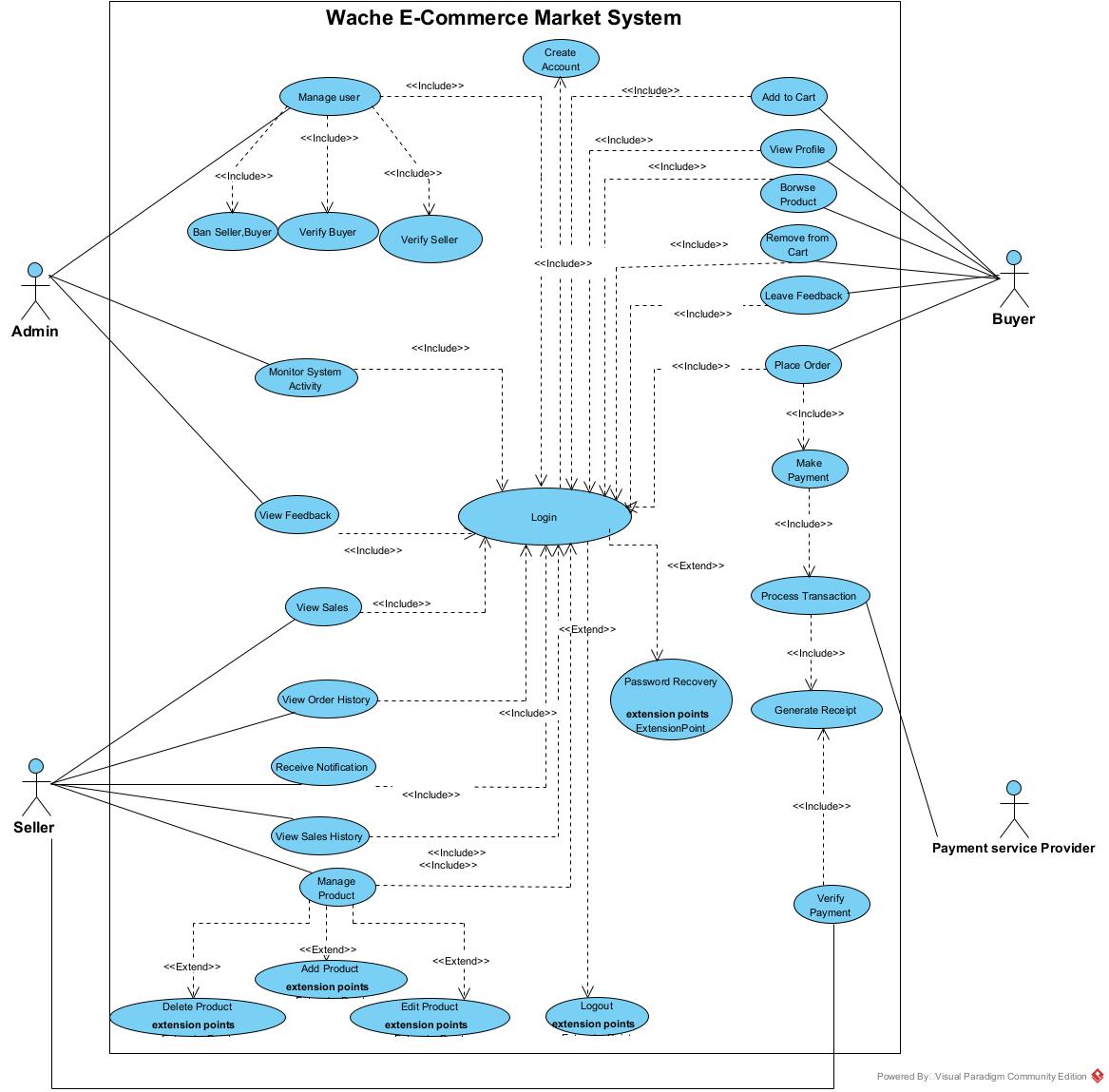
## Use case diagram components

A **use case diagram** is a visual representation that illustrates the interactions between users (actors) and a system to achieve specific goals. It provides a high-level overview of the system’s functionality from the user’s perspective. The main components of a use case diagram include:

* **Actors**: Represent the external entities that interact with the system. Actors can be human users, external systems, or hardware devices. They are typically depicted as stick figures.
* **Use Cases**: Represent the specific functionalities or services the system provides to its actors. Use cases are shown as ovals with descriptive names inside them, indicating actions or goals the actors can perform.
* **System Boundary**: A rectangle that defines the scope of the system. All use cases are placed within this boundary, showing which functionalities belong to the system.
* **Relationships**: These are lines that connect actors to use cases, and use cases to other use cases. Relationships include:
  + **Association**: A line connecting an actor to a use case, indicating interaction.
  + **Include**: A dashed arrow showing that a use case always includes another use case.
  + **Extend**: A dashed arrow showing that a use case may optionally extend another use case’s behavior.
  + **Generalization**: An arrow indicating inheritance or specialization between actors or between use cases.

Each of these components works together to create a clear and understandable view of how users interact with the system, making it easier to analyze requirements and design the system architecture.

## Example of use case mode



## Use case Description/Template

A use case description provides a detailed narrative of how an actor interacts with the system to accomplish a specific goal. It complements the use case diagram by offering step-by-step scenarios for each action.

To ensure consistency, each use case is documented using a standard template that includes the following fields:

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-001 | |
| **Use Case Name** | Create Account | |
| **Actor** | Buyer | |
| **Summary** | This use case allows the Buyer to create an account on the platform. | |
| **Preconditions** | 1. The Buyer must have access to the platform.  2. The Buyer must not already have an account. | |
| **Trigger** | The Buyer navigates to the registration page and clicks the Register button. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The Buyer opens the registration page and clicks Register.  3. The Buyer fills out and submits it. | 2. The system displays a registration form.  4. The system validates the details. |
| **Verification** | 5. The Buyer receives a verification email or SMS.  6. The system sends a verification email/SMS.  7. The Buyer clicks the verification link or enters the code.  8. The system confirms registration and logs the Buyer in. | |
| **Alternative Scenario** | 9. If invalid details, the system shows an error and prompts correction.  10. If verification is not completed in time, the system deactivates the account. | |
| **Postcondition** | The Buyer is registered, and the profile is stored as active upon verification. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-002 | |
| **Use Case Name** | Login | |
| **Actor** | Buyer / Seller / Admin | |
| **Summary** | This use case allows users to log into the system. | |
| **Preconditions** | 1. The user must have a registered account.2. The user must have access to the platform. | |
| **Trigger** | The user navigates to the login page and enters their credentials. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The user enters credentials and submits the login form | 2. The system validates credentials. |
| **Verification** | 3. The user waits for authentication.  4. The system authenticates the user and grants access. | |
| **Alternative Scenario** | 5. If invalid credentials, the system shows an error.6. If the account is locked, the system shows a message and provides recovery options. | |
| **Postcondition** | The user is logged into the system and can access their account features. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-003 | |
| **Use Case Name** | Password Recovery | |
| **Actor** | Buyer | |
| **Summary** | This use case allows the Buyer to recover their password if forgotten. | |
| **Preconditions** | 1. The Buyer must have a registered account.  2. The Buyer must have access to their registered email. | |
| **Trigger** | The Buyer clicks the Forgot Password link on the login page. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The Buyer requests password recovery by providing their email | .2. The system validates the email and generates a reset link. |
| **Verification** | 3. The Buyer receives the reset link via email.  4. The system sends the reset link.  5. The Buyer clicks the link and sets a new password.  6. The system updates the password and confirms the change. | |
| **Alternative Scenario** | 7. If the email is invalid, the system shows an error.8. If the reset link expires, the system prompts the Buyer to request a new link. | |
| **Postcondition** | The Buyer's password is updated, and they can log in with the new password. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-004 | |
| **Use Case Name** | Browse Product | |
| **Actor** | Buyer | |
| **Summary** | This use case allows the Buyer to browse products available in the marketplace. | |
| **Preconditions** | 1. The Buyer must have access to the platform.  2. Products must be available in the marketplace. | |
| **Trigger** | The Buyer navigates to the product browsing section or searches for products. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The Buyer searches or navigates to browse products. | 2. The system displays a list of available products. |
| **Verification** | 3. The Buyer reviews the product list.4. The system ensures the product list is accurate and up-to-date. | |
| **Alternative Scenario** | 5. If no products are available, the system displays a message indicating no products. | |
| **Postcondition** | The Buyer has browsed the products, and the system remains operational. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-005 | |
| **Use Case Name** | Add to Cart | |
| **Actor** | Buyer | |
| **Summary** | This use case allows the Buyer to add a product to their shopping cart. | |
| **Preconditions** | 1. The Buyer must be logged into the system.  2. The product must be available in the marketplace. | |
| **Trigger** | The Buyer selects a product and clicks the Add to Cart button. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The Buyer selects a product and clicks Add to Cart | 2. The system adds the product to the Buyer’s cart. |
| **Verification** | 3. The Buyer checks the cart to confirm the product is added.  4. The system updates the cart and displays the updated cart contents. | |
| **Alternative Scenario** | 5. If the product is out of stock, the system displays a message indicating unavailability. | |
| **Postcondition** | The product is added to the Buyer’s cart, and the cart is updated in the system. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-006 | |
| **Use Case Name** | Remove from Cart | |
| **Actor** | Buyer | |
| **Summary** | This use case allows the Buyer to remove a product from their shopping cart. | |
| **Preconditions** | 1. The Buyer must be logged into the system.  2. The Buyer’s cart must contain products. | |
| **Trigger** | The Buyer selects a product in the cart and clicks the Remove button. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The Buyer selects a product in the cart to remove. | 2. The system prompts for confirmation. |
| **Verification** | 3. The Buyer confirms the removal.  4. The system updates the cart by removing the selected product. | |
| **Alternative Scenario** | 5. If the Buyer cancels the removal, the system returns to the cart without changes. | |
| **Postcondition** | The product is removed from the Buyer’s cart, and the cart is updated in the system. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-007 | |
| **Use Case Name** | Place Order | |
| **Actor** | Buyer | |
| **Summary** | This use case allows the Buyer to place an order for items in their cart. | |
| **Preconditions** | 1. The Buyer must be logged into the system.  2. The Buyer’s cart must contain items. | |
| **Trigger** | The Buyer clicks the Place Order button from the cart or checkout page. | |
| **Basic Scenario** | **Actor Action** | **System Responses** |
| 1. The Buyer confirms the items in the cart and clicks Place Order. | 2. The system creates an order and initiates the payment process. |
| **Verification** | 3. The Buyer is redirected to the payment page.  4. The system ensures the order is recorded correctly. | |
| **Alternative Scenario** | 5. If the cart is empty, the system displays a message indicating no items to order. | |
| **Postcondition** | The order is placed, and the Buyer is directed to complete the payment. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-008 | |
| **Use Case Name** | Make Payment | |
| **Actor** | Buyer | |
| **Summary** | This use case allows the Buyer to make a payment for their order. | |
| **Preconditions** | 1. The Buyer must have placed an order.  2. The payment service provider must be available. | |
| **Trigger** | The Buyer is redirected to the payment page after placing an order. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The Buyer selects a payment method and submits payment details. | 2. The system sends the payment details to the payment service provider. |
| **Verification** | 3. The Buyer waits for payment confirmation.  4. The system processes the payment and confirms the transaction status. | |
| **Alternative Scenario** | 5. If the payment fails, the system displays an error and prompts the Buyer to retry with a different method. | |
| **Postcondition** | The payment is completed, and the order is marked as paid in the system. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-009 | |
| **Use Case Name** | Leave Feedback | |
| **Actor** | Buyer | |
| **Summary** | This use case allows the Buyer to leave feedback for a purchased product or seller. | |
| **Preconditions** | 1. The Buyer must be logged into the system.  2. The Buyer must have completed a purchase. | |
| **Trigger** | The Buyer navigates to the feedback section for a purchased product or seller. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The Buyer submits feedback for a purchased product or seller. | 2. The system validates the feedback content. |
| **Verification** | 3. The Buyer confirms the submission.  4. The system saves the feedback and associates it with the product or seller. | |
| **Alternative Scenario** | 5. If the feedback content is invalid, the system displays an error and prompts for correction. | |
| **Postcondition** | The feedback is saved in the system and is available for review by the Admin or Seller. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-010 | |
| **Use Case Name** | Manage Product | |
| **Actor** | Seller | |
| **Summary** | This use case allows the Seller to manage their products by adding, editing, or deleting them. | |
| **Preconditions** | 1. The Seller must be logged into the system.  2. The Seller must have permission to manage products. | |
| **Trigger** | The Seller navigates to the product management section. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The Seller initiates the process to manage their products. | 2. The system provides options to add, edit, or delete products. |
| **Verification** | 3. The Seller selects an option (add/edit/delete).  4. The system processes the selected action accordingly. | |
| **Alternative Scenario** | 5. If the Seller has no products, the system displays a message indicating no products to manage. | |
| **Postcondition** | The Seller’s product inventory is updated based on the action taken. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-011 | |
| **Use Case Name** | Add Product | |
| **Actor** | Seller | |
| **Summary** | This use case allows the Seller to add a new product to their inventory. | |
| **Preconditions** | 1. The Seller must be logged into the system.  2. The Seller must have permission to add products. | |
| **Trigger** | The Seller selects the option to add a product from the product management section. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The Seller submits details of a new product to add. | 2. The system validates the product details. |
| **Verification** | 3. The Seller confirms the submission.  4. The system adds the product to the Seller’s inventory and marketplace. | |
| **Alternative Scenario** | 5. If the product details are invalid, the system displays an error and prompts for correction. | |
| **Postcondition** | The new product is added to the Seller’s inventory and is available in the marketplace. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-012 | |
| **Use Case Name** | Edit Product | |
| **Actor** | Seller | |
| **Summary** | This use case allows the Seller to edit an existing product in their inventory. | |
| **Preconditions** | 1. The Seller must be logged into the system.  2. The Seller must have existing products to edit. | |
| **Trigger** | The Seller selects the option to edit a product from the product management section. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The Seller selects a product and updates its details. | 2. The system validates the updated details. |
| **Verification** | 3. The Seller confirms the changes.  4. The system saves the updated product information in the marketplace. | |
| **Alternative Scenario** | 5. If the updated details are invalid, the system displays an error and prompts for correction. | |
| **Postcondition** | The product’s details are updated in the Seller’s inventory and the marketplace. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-013 | |
| **Use Case Name** | Delete Product | |
| **Actor** | Seller | |
| **Summary** | This use case allows the Seller to delete a product from their inventory. | |
| **Preconditions** | 1. The Seller must be logged into the system.  2. The Seller must have existing products to delete. | |
| **Trigger** | The Seller selects the option to delete a product from the product management section. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The Seller selects a product to delete. | 2. The system prompts for confirmation. |
| **Verification** | 3. The Seller confirms the deletion.  4. The system removes the product from the Seller’s inventory. | |
| **Alternative Scenario** | 5. If the Seller cancels the deletion, the system returns to the product management section without changes. | |
| **Postcondition** | The product is removed from the Seller’s inventory and is no longer available in the marketplace. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-014 | |
| **Use Case Name** | View Sales (Seller) | |
| **Actor** | Seller | |
| **Summary** | This use case allows the Seller to view their sales data and statistics. | |
| **Preconditions** | 1. The Seller must be logged into the system.  2. Sales data must exist for the Seller. | |
| **Trigger** | The Seller navigates to the sales section. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The Seller requests to view their sales data. | 2. The system retrieves and displays the Seller’s sales statistics. |
| **Verification** | 3. The Seller reviews the sales data.  4. The system ensures the data is accurate and up-to-date. | |
| **Alternative Scenario** | 5. If no sales data is available, the system displays a message indicating no data. | |
| **Postcondition** | The Seller has reviewed their sales data, and the system remains operational. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-015 | |
| **Use Case Name** | View Order History | |
| **Actor** | Seller | |
| **Summary** | This use case allows the Seller to view their order history. | |
| **Preconditions** | 1. The Seller must be logged into the system.  2. The Seller must have previous orders. | |
| **Trigger** | The Seller navigates to the order history section. | |
| **Basic Scenario** | Actor Action | System Response |
| 1. The Seller requests to view their order history. | 2. The system retrieves and displays the Seller’s order history. |
| **Verification** | 3. The Seller reviews the order history.  4. The system ensures all orders are displayed correctly. | |
| **Alternative Scenario** | 5. If no orders exist, the system displays a message indicating no order history. | |
| **Postcondition** | The Seller has reviewed their order history, and the system remains operational. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-016 | |
| **Use Case Name** | Receive Notification | |
| **Actor** | Seller | |
| **Summary** | This use case allows the Seller to receive notifications about orders or account updates. | |
| **Preconditions** | 1. The Seller must be logged into the system.  2. Notifications must exist for the Seller. | |
| **Trigger** | The Seller logs in or navigates to the notifications section. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The Seller logs in to check for notifications. | 2. The system displays notifications about orders or account updates. |
| **Verification** | 3. The Seller reviews the notifications.  4. The system ensures all notifications are displayed correctly. | |
| **Alternative Scenario** | 5. If no notifications exist, the system displays a message indicating no notifications. | |
| **Postcondition** | The Seller has reviewed their notifications, and the system remains operational. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-017 | |
| **Use Case Name** | Verify Payment | |
| **Actor** | Payment Service Provider | |
| **Summary** | This use case allows the Payment Service Provider to verify payment details for a transaction. | |
| **Preconditions** | 1. The Buyer must have submitted payment details.  2. The Payment Service Provider must be operational. | |
| **Trigger** | The system submits payment details to the Payment Service Provider for verification. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The system submits payment details for verification. | 2. The Payment Service Provider verifies the payment details. |
| **Verification** | 3. The system waits for verification status.  4. The Payment Service Provider confirms the payment status (valid/invalid). | |
| **Alternative Scenario** | 5. If the payment details are invalid, the Payment Service Provider returns an error, and the system prompts the Buyer to correct the details. | |
| **Postcondition** | The payment is verified, and the system proceeds with the transaction process. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-018 | |
| **Use Case Name** | Process Transaction | |
| **Actor** | Payment Service Provider | |
| **Summary** | This use case allows the Payment Service Provider to process a transaction for the Buyer’s order. | |
| **Preconditions** | 1. The Buyer must have submitted payment details.  2. The Payment Service Provider must be operational. | |
| **Trigger** | The system sends transaction details to the Payment Service Provider after the Buyer submits payment. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The system sends transaction details to the Payment Service Provider. | 2. The Payment Service Provider processes the transaction. |
| **Verification** | 3. The system waits for the transaction status.  4. The Payment Service Provider returns the transaction status (success/failure). | |
| **Alternative Scenario** | 5. If the transaction fails, the Payment Service Provider returns an error, and the system prompts the Buyer to retry. | |
| **Postcondition** | The transaction is processed, and the system updates the order status based on the result. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-019 | |
| **Use Case Name** | Generate Receipt | |
| **Actor** | Payment Service Provider | |
| **Summary** | This use case allows the Payment Service Provider to generate a receipt for a successful transaction. | |
| **Preconditions** | 1. The transaction must be successfully processed.  2. The Payment Service Provider must support receipt generation. | |
| **Trigger** | The system requests a receipt after a successful transaction. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The system requests a receipt from the Payment Service Provider. | 2. The Payment Service Provider generates the receipt. |
| **Verification** | 3. The system receives the receipt.  4. The Payment Service Provider sends the receipt to the system. | |
| **Alternative Scenario** | 5. If receipt generation fails, the system logs the error and notifies the Buyer of the transaction without a receipt. | |
| **Postcondition** | The receipt is generated and stored in the system, and the Buyer receives a copy. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-020 | |
| **Use Case Name** | Manage User | |
| **Actor** | Admin | |
| **Summary** | This use case allows the Admin to manage user accounts by banning or verifying buyers and sellers. | |
| **Preconditions** | 1. The Admin must be logged into the system with appropriate permissions.  2. The system must have user accounts available to manage. | |
| **Trigger** | The Admin navigates to the user management section of the system. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The Admin navigates to the user management section. | 2. The system displays a list of users with options to ban or verify. |
| **Verification** | 3. The Admin selects a user and chooses an action (ban/verify).  4. The system updates the user’s status and confirms the action. | |
| **Alternative Scenario** | 5. If the user list is empty, the system displays a message indicating no users are available. | |
| **Postcondition** | The user’s status is updated in the system, and the Admin is notified of the successful action. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-021 | |
| **Use Case Name** | Ban Seller | |
| **Actor** | Admin | |
| **Summary** | This use case allows the Admin to ban a seller from the platform. | |
| **Preconditions** | 1. The Admin must be logged into the system.  2. The seller must have an active account. | |
| **Trigger** | The Admin selects the option to ban a seller from the user management section. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The Admin selects a seller to ban. | 2. The system prompts for confirmation. |
| **Verification** | 3. The Admin confirms the action.  4. The system updates the seller's status to banned and notifies the seller. | |
| **Alternative Scenario** | 5. If the Admin cancels the action, the system returns to the user management section without changes. | |
| **Postcondition** | The seller's account is banned, and they can no longer access the platform. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-023 | |
| **Use Case Name** | Verify Buyer | |
| **Actor** | Admin | |
| **Summary** | This use case allows the Admin to verify a buyer's account. | |
| **Preconditions** | 1. The Admin must be logged into the system.  2. The buyer must have an unverified account. | |
| **Trigger** | The Admin selects the option to verify a buyer from the user management section. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The Admin selects a buyer to verify. | 2. The system prompts for confirmation. |
| **Verification** | 3. The Admin confirms the action.  4. The system marks the buyer as verified and updates their profile. | |
| **Alternative Scenario** | 5. If the Admin cancels the action, the system returns to the user management section without changes. | |
| **Postcondition** | The buyer's account is verified, granting them full access to platform features. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-024 | |
| **Use Case Name** | Verify Seller | |
| **Actor** | Admin | |
| **Summary** | This use case allows the Admin to verify a seller's account. | |
| **Preconditions** | 1. The Admin must be logged into the system.  2. The seller must have an unverified account. | |
| **Trigger** | The Admin selects the option to verify a seller from the user management section. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The Admin selects a seller to verify. | 2. The system prompts for confirmation. |
| **Verification** | 3. The Admin confirms the action.  4. The system marks the seller as verified and updates their profile. | |
| **Alternative Scenario** | 5. If the Admin cancels the action, the system returns to the user management section without changes. | |
| **Postcondition** | The seller's account is verified, granting them full access to platform features. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-025 | |
| **Use Case Name** | Monitor System Activity | |
| **Actor** | Admin | |
| **Summary** | This use case allows the Admin to monitor system activity, including user actions and metrics. | |
| **Preconditions** | 1. The Admin must be logged into the system.  2. The system must have activity logs available. | |
| **Trigger** | The Admin navigates to the system monitoring dashboard. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The Admin accesses the system monitoring dashboard. | 2. The system displays activity logs, user actions, and metrics. |
| **Verification** | 3. The Admin reviews the logs and metrics.  4. The system ensures the data is up-to-date and accurate. | |
| **Alternative Scenario** | 5. If no activity logs are available, the system displays a message indicating no data. | |
| **Postcondition** | The Admin has reviewed the system activity, and the system remains operational. | |

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-026 | |
| **Use Case Name** | View Feedback | |
| **Actor** | Admin | |
| **Summary** | This use case allows the Admin to view feedback left by buyers. | |
| **Preconditions** | 1. The Admin must be logged into the system.  2. Feedback must exist in the system. | |
| **Trigger** | The Admin navigates to the feedback section. | |
| **Basic Scenario** | **Actor Action** | **System Response** |
| 1. The Admin requests to view feedback from buyers. | 2. The system retrieves and displays feedback. |
| **Verification** | 3. The Admin reviews the feedback.  4. The system ensures all feedback is displayed correctly. | |
| **Alternative Scenario** | 5. If no feedback is available, the system displays a message indicating no feedback. | |
| **Postcondition** | The Admin has reviewed the feedback, and the system remains operational. | |

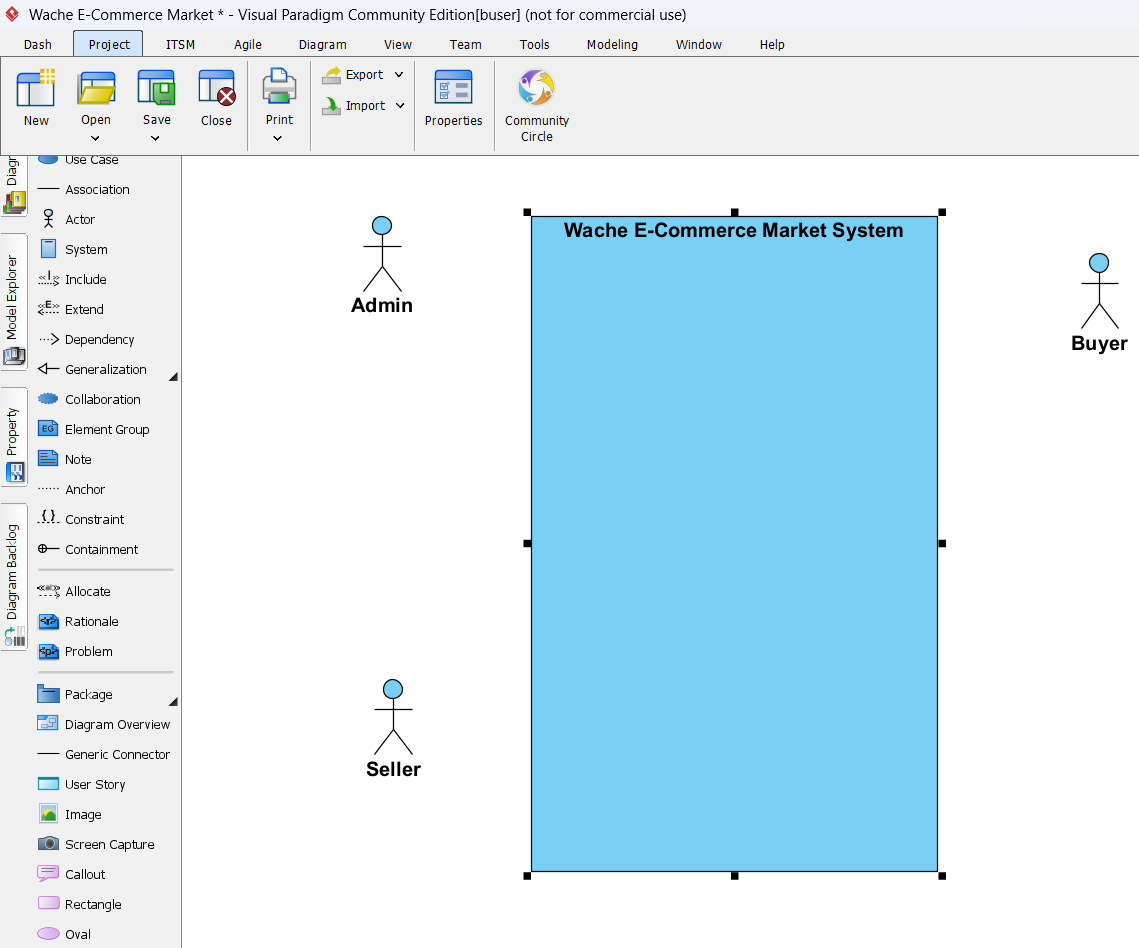
|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | |
| **Use Case ID** | UC-027 | |
| **Use Case Name** | View Sales (Admin) | |
| **Actor** | Admin | |
| **Summary** | This use case allows the Admin to view sales data and statistics. | |
| **Preconditions** | 1. The Admin must be logged into the system.  2. Sales data must exist in the system. | |
| **Trigger** | The Admin navigates to the sales section. | |
| **Basic Scenario** | **. Actor Action** | **System Response** |
| 1. The Admin requests to view sales data | 2. The system retrieves and displays sales statistics and reports. |
| **Verification** | 3. The Admin reviews the sales data.  4. The system ensures the data is accurate and up-to-date. | |
| **Alternative Scenario** | 5. If no sales data is available, the system displays a message indicating no data. | |
| **Postcondition** | The Admin has reviewed the sales data, and the system remains operational. | |

## Tools and steps to draw Use Case

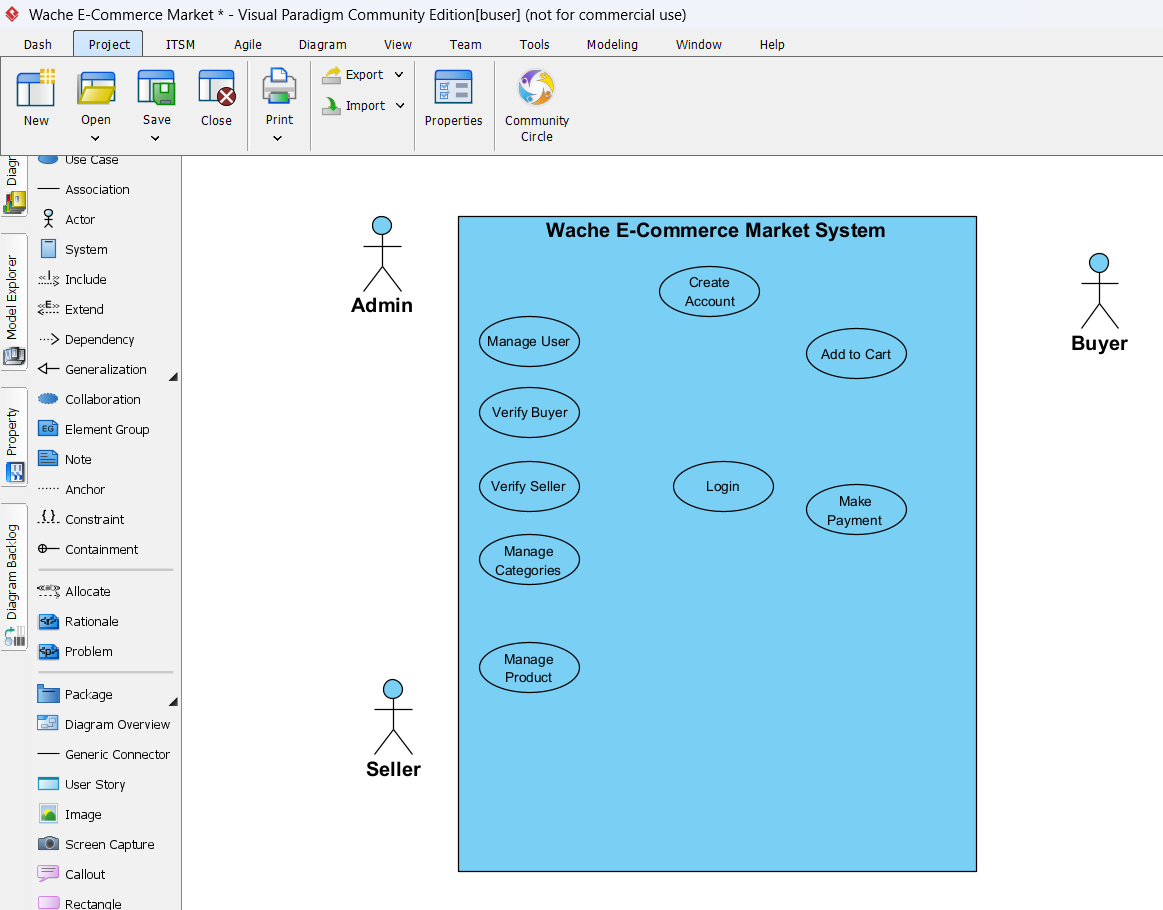
For drawing the use case diagram, we used **Visual Paradigm**, a user-friendly UML modeling tool that helps create clear and standardized diagrams.

The steps followed were:

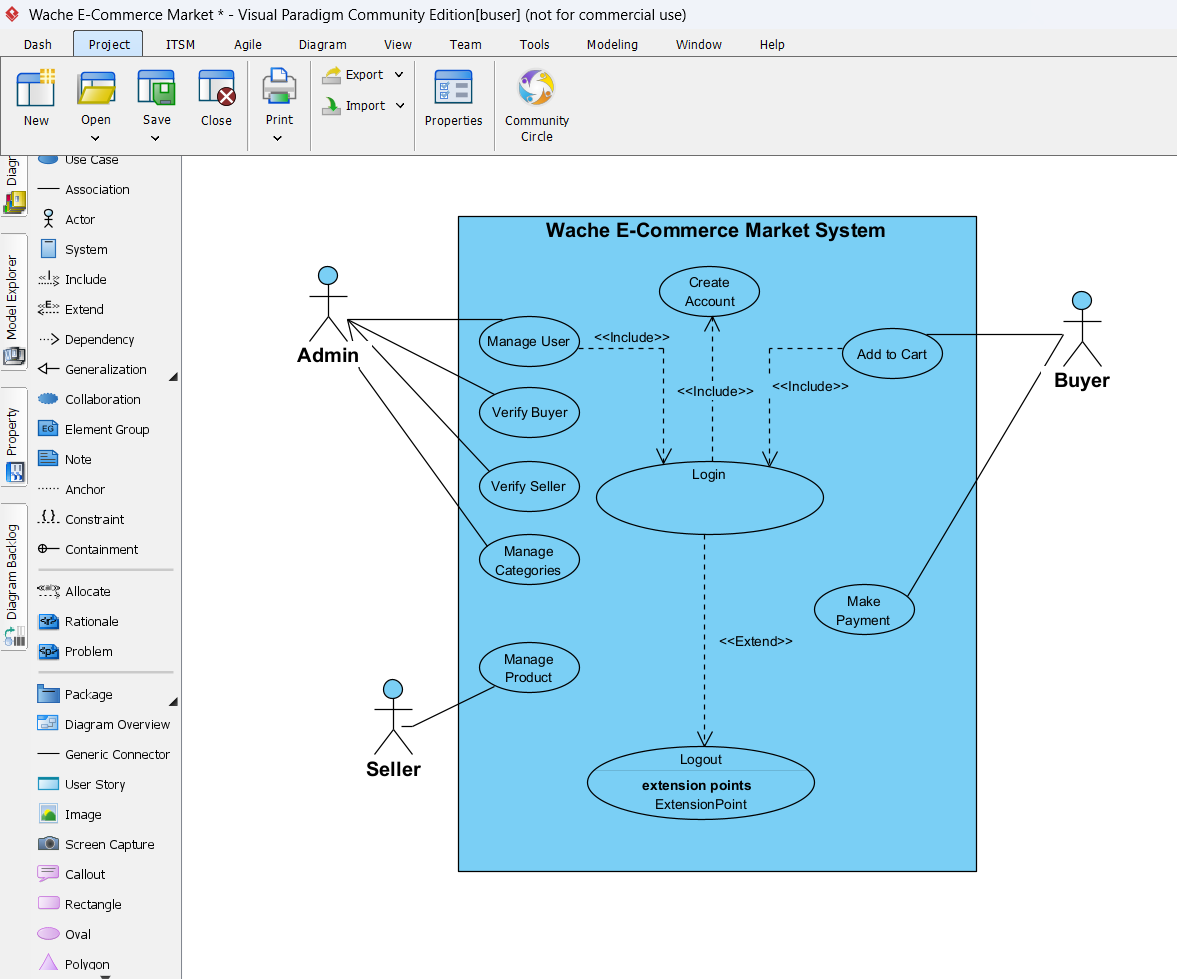
* Open Visual Paradigm and create a new project.
* Select **Use Case Diagram** from the available diagram types.
* Add the main actors (Admin, Seller, Buyer) using the actor symbols.



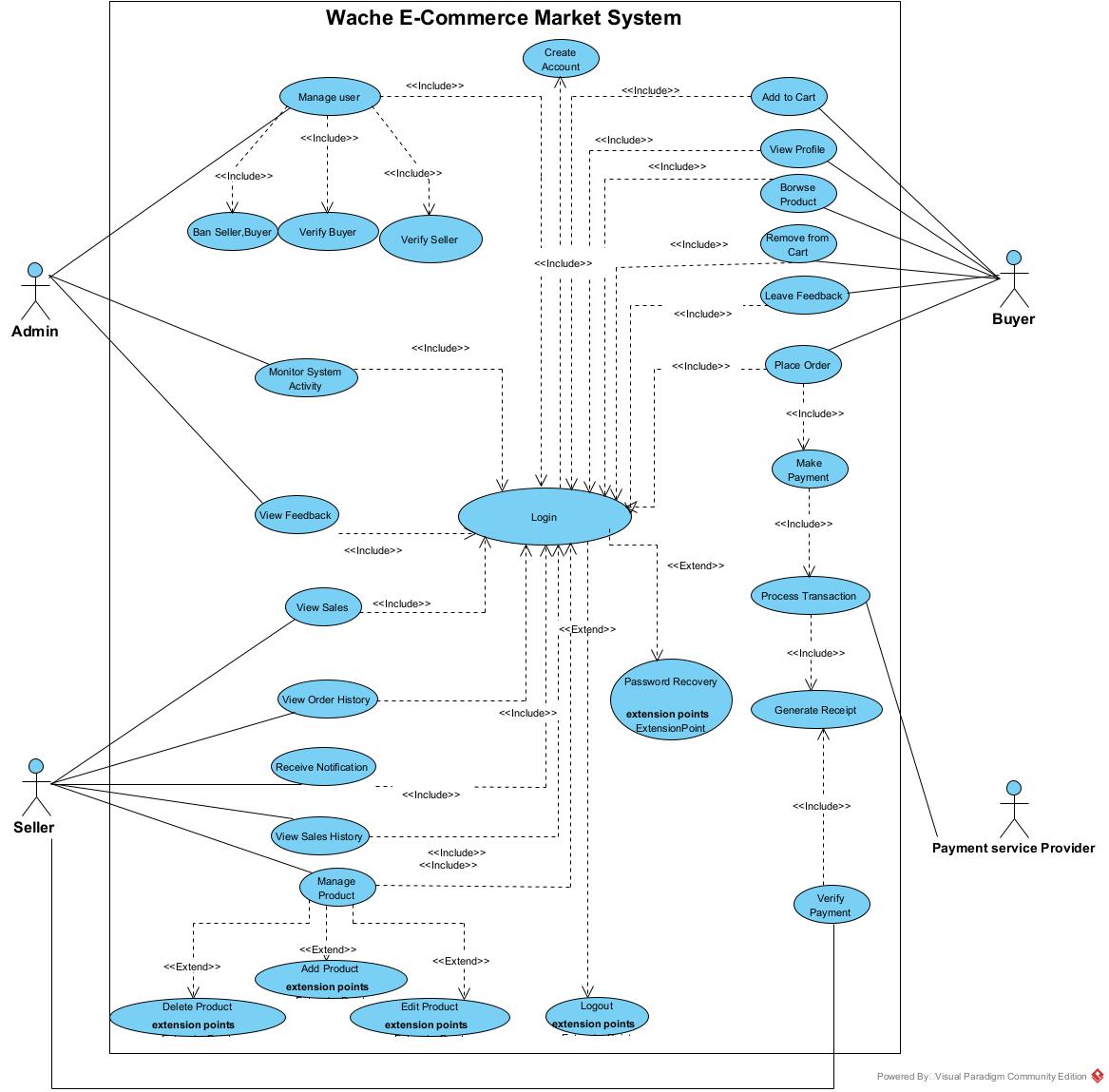
* Insert the use cases (e.g., Login, Create Account, Browse Product) using ellipse shapes.



* Draw association lines between actors and use cases.
* Apply the **<<include>>** and **<<extend>>** relationships where needed.



* Arrange the diagram layout neatly for readability.
* Save and export the diagram as an image for documentation.



# CHAPTER TWO

## 2.1 High-Level Sequence Diagram

In this section, we present the high-level sequence diagram for the Wache E-Commerce Market System. This diagram shows the interactions between key system components (such as Buyer, Seller, Admin) for major use cases like product browsing, adding to cart, managing products, and completing payments.

The purpose of the high-level sequence diagram is to capture the overall flow of messages and interactions between actors and system components, without diving into detailed class-level interactions.

## ****2.2 Components of High-Level Sequence Diagram****

A high-level sequence diagram is made up of several key components that together describe how actors and system parts interact over time to accomplish a specific process or use case. Below are the main components typically found in a high-level sequence diagram:

* **Actors:** External entities that interact with the system, such as users (Buyer, Seller, Admin) or external services. They initiate or respond to system messages.
* **System Components (Objects):** The main parts of the system that handle specific tasks, such as the Product Catalog, Cart, Payment Gateway, User Account, or Order Management module.
* **Lifelines:** Vertical dashed lines extending from each actor or system component. These represent the time dimension, showing that the actor or component exists and can send or receive messages during the interaction.
* **Messages:** Horizontal arrows that represent communication between lifelines. Messages can be:
  + **Synchronous** (with a waiting reply, like a database query)
  + **Asynchronous** (without waiting, like sending a notification) These show the sequence of operations, such as “add product to cart” or “process payment.”
* **Activation Bars (Execution Occurrences):** Thin vertical rectangles drawn on lifelines, showing when a component is actively doing something (executing an operation or waiting for a reply).
* **Return Messages:** Dashed arrows showing the response or result of an earlier message, such as a confirmation or error.
* **Sequence Flow:** The top-to-bottom order of the diagram, where interactions are arranged chronologically, helping viewers understand the process flow over time.

These components together provide a clear picture of how various parts of the system work together to fulfill a high-level process, allowing stakeholders to trace the interaction steps and understand system behavior.

# CHAPTER THREE

## 3.1 Low-level (Detail) Design (class design)

Low-level design focuses on the internal structure of each module or component. At this level, the design addresses how each function and feature will be implemented within the system classes. For the e-commerce system, the class design identifies the key classes like User, Product, Order, Transaction, and the roles Buyer, Seller, and Admin, each with their own responsibilities and behaviors.

Each class includes:

* **Attributes**: Representing data stored (e.g., name, email, product price)
* **Methods**: Defining operations or behaviors (e.g., login(), checkout(), addToCart())

The class diagram illustrates object-oriented principles such as inheritance (Buyer, Seller, Admin inherit from User) and associations (e.g., Order has a relation with Product and Buyer).

## 3.2 Components of the Class Diagram

A UML class diagram typically contains the following components:

|  |  |
| --- | --- |
| **Component** | **Description** |
| **Class** | A blueprint for objects, showing attributes and methods. |
| **Attributes** | Variables that hold the data or state of the class. |
| **Methods** | Functions that define the class behavior. |
| **Relationships** | Links between classes such as:  - **Association**: Simple link between classes  - **Aggregation/Composition**: Whole-part relationships  - **Generalization**: Inheritance relationship (is-a) |
| **Visibility** | Symbols: + public, - private, # protected |
| **Multiplicity** | Indicates number of instances (e.g., 1, 0..*, 1..*, etc.) |

## 3.3 Example of Class Diagram

The following class diagram, created using **Visual Paradigm**, represents the detailed object-oriented structure of the e-commerce system. It illustrates key entities such as User, Buyer, Seller, Admin, Product, Order, Cart, and their relationships.

This diagram visually expresses:

* **Inheritance** relationships (e.g., Buyer, Seller, and Admin inherit from User)
* **Associations** between classes (e.g., Order is linked to both Buyer and Product)
* **Attributes and methods** for each class, reflecting system functionality

This class-level model provides a blueprint for implementation and ensures modularity and reusability.

## 3.4 Tools and Steps to Draw Low-Level Class Design

Creating a detailed class design is a crucial step in object-oriented system development. Here’s how this was approached using **Visual Paradigm**:

**Tool Used**

**Visual Paradigm**: A powerful UML tool for drawing class diagrams

**Steps to Create the Class Diagram**

1. Identify Key Entities

Based on use cases (e.g., User, Product, Order, Transaction)

1. Define Relationships

Model inheritance, associations, and multiplicity between entities

1. Add Attributes and Methods

Each class includes relevant fields (e.g., email, price) and functions (e.g., login(), checkout())

1. Apply Object-Oriented Principles

Use generalization for inheritance, encapsulate fields, and keep cohesion

Ensure symbols for visibility (+, -) and relationships follow UML conventions

1. Export and Validate  
   Diagram is reviewed for accuracy and exported as an image or embedded in documentation

This low-level design lays the foundation for coding and database modeling by ensuring clear and consistent object structures.

# CHAPTER FOUR

## 4.1 Implementation (export class diagram into code and update code and diagram)

The class diagram created in Visual Paradigm was used as the basis for generating the initial source code structure of the e-commerce system. This process not only speeds up development but also ensures consistency between design and implementation.

**Code Generation from Class Diagram**

Using Visual Paradigm’s **Code Engineering** feature, the class diagram was exported into the target programming language (e.g., Java). The generated code included:

* Class declarations
* Attributes with proper data types
* Method stubs (without implementation)

This provided a **scaffold** that developers could immediately build upon.

**Updating Code and Diagram**

As the project evolved, the following steps were taken to maintain alignment:

* When **new attributes or methods** were added in code, they were **manually added back into the diagram**
* If relationships changed (e.g., new association between Transaction and Product), they were **reflected visually**
* Visual Paradigm’s **reverse engineering** feature was occasionally used to **update diagrams based on code**

This ensured continuous **synchronization** between documentation and implementation throughout development.

## 4.2 Steps to Generate Code from Class Diagram

The following steps outline how code was generated from the class diagram using **Visual Paradigm**:

Step 1: Finalize the Class Diagram

Ensure that the class diagram includes:

* All relevant classes, attributes, and methods
* Relationships like associations and inheritance
* Correct visibility modifiers (e.g., public, private)

Step 2: Access Instant Generator

Navigate to:

**Tools > Code > Instant Generator**

This opens the code generation panel.

Step 3: Configure Generation Options

In the generator window:

* **Model**: Select the entire diagram or specific classes
* **Language**: Choose the target programming language (Java, Python, PHP, etc.)
* **Output Path**: Define the folder where the generated files will be saved

Optional configurations:

* Choose whether to generate comments or method bodies
* Enable or disable getter/setter generation
* Set package structure or namespace if needed

Step 4: Generate Code

Click the **Generate** button. Visual Paradigm will automatically create:

One source file per class

Files containing attributes and method stubs that reflect the diagram

Step 5: Update and Sync

As development continues, changes in code may require updates to the class diagram

Use **reverse engineering** to import code changes back into Visual Paradigm:

**Tools > Code > Reverse Engineering**

This process allows for a continuous feedback loop between the design and the implementation, supporting agile and iterative development practices.

# CHAPTER FIVE

## 5.1 Change Management (Version Control Using Git)

Effective change management is essential for maintaining code quality, collaboration, and traceability throughout the software development lifecycle. In this project, we adopted **Git** as our distributed version control system to manage changes in the source codebase.

**Git allowed us to:**

* Track the history of every file and line of code
* Collaborate as a team without overwriting each other's work
* Create and merge branches for feature development, bug fixes, and releases
* Revert to previous stable versions when issues occurred
* Maintain a clear, documented progression of the project

To complement Git, we used **GitHub** as our remote repository hosting service. GitHub provided a centralized platform for storing, reviewing, and sharing our codebase. It also enabled additional collaboration features such as pull requests, issues tracking, and project boards.

This approach significantly improved the maintainability and scalability of the project and allowed for seamless teamwork even in distributed environments.

## 5.2 Steps and Tools Used in Our Project to Implement Git

The following steps outline how Git and GitHub were integrated into our project workflow:

**Tools Used**

* **Git**: Installed locally to manage version control on each developer’s machine
* **GitHub**: Hosted the project repository online for team collaboration and backup
* **Git Bash**: Used for executing Git commands

**Steps Followed**

1. Initialize Git in the Project Directory

This set up Git tracking in our local codebase.

1. Create a Remote Repository on GitHub

* Repository created at <https://github.com/username/project-name>
* Default .gitignore, README.md and License were included

1. Link Local Project to GitHub Repository
2. Add and Commit Files
3. Push Code to GitHub
4. Pull Requests and Code Reviews

* Reviewing code changes
* Discussing improvements

1. Conflict Resolution

This Git-based workflow ensured that our team maintained a clean, stable, and documented development process. All project members had real-time access to the latest code and could contribute without fear of overwriting each other's work.

# CHAPTER SIX

## 6.1 Unit Test

Unit testing is a vital process in ensuring that individual components of the software function as intended. In this project, **JUnit** was used in conjunction with **Eclipse IDE** to perform unit testing on core classes and methods of the e-commerce system.

**By testing each unit in isolation, we were able to:**

* Detect bugs early in the development process
* Validate the behavior of each method or logic block
* Refactor code safely with confidence
* Ensure ongoing reliability through automated test execution
* Facilitate safer refactoring and updates

Each test case was designed to test one specific functionality using expected inputs and validating outputs. If changes were made to the codebase, rerunning the unit tests ensured that existing features continued to work properly — helping prevent regressions.

**JUnit’s** seamless integration with **Eclipse** allowed developers to quickly write, execute, and monitor tests during development

## 6.2 Steps and tools used in Unit Test

In this project, unit testing was performed using **JUnit 4**, which is natively integrated into the **Eclipse IDE**. No additional installations or external configurations were required. The code under test was **automatically generated from the class diagram** created in Visual Paradigm, ensuring consistency between the system design and implementation.

This approach allowed the development team to efficiently validate the functionality of each class and method derived from the system’s object-oriented design.

**Tools Used**

* Eclipse IDE

Utilized for writing and executing unit tests within an integrated development environment.

* JUnit 4

The default unit testing framework bundled with Eclipse, used to verify correctness of class behaviors.

* Visual Paradigm

Used to design the class diagram and export Java class structures for testing.

**Steps Followed in Unit Testing**

* 1. Generate Class Code from Diagram

The class diagram was modeled in Visual Paradigm, and the system’s classes were exported using the Instant Generator. The generated .java files (e.g., User.java, Product.java) were imported into Eclipse for testing.

* 1. Create a JUnit 4 Test Class

In Eclipse:

* + - Right-click on the generated class → New > JUnit Test Case
    - Select **JUnit 4** as the version
    - Choose methods to test or add custom test methods manually
  1. Write Test Methods

Each test method was annotated with @Test and contained assertions to validate expected outcomes.

Example:

**import org.junit.Test;**

**import static org.junit.Assert.\*;**

**public class ProductTest {**

**@Test**

**public void testCalculatePrice() {**

**Product p = new Product("Phone", 300.0);**

**double total = p.calculatePrice(2);**

**assertEquals(600.0, total, 0.01);**

**}**

**}**

* 1. Run Tests in Eclipse
* Right-click on the test class or file → Run As > JUnit Test
* The Eclipse JUnit view displayed the test status using a **green bar** (success) or **red bar** (failure)
  1. Review Test Results

The JUnit output panel provided:

* Pass/fail indicators
* Detailed failure messages
* Stack traces and error locations
* Execution times for each test

This structured approach ensured that all critical class behaviors were validated against expected results. By combining **code generation from the class diagram** with **built-in testing tools**, the team was able to streamline testing efforts and maintain a consistent, error-resistant codebase.