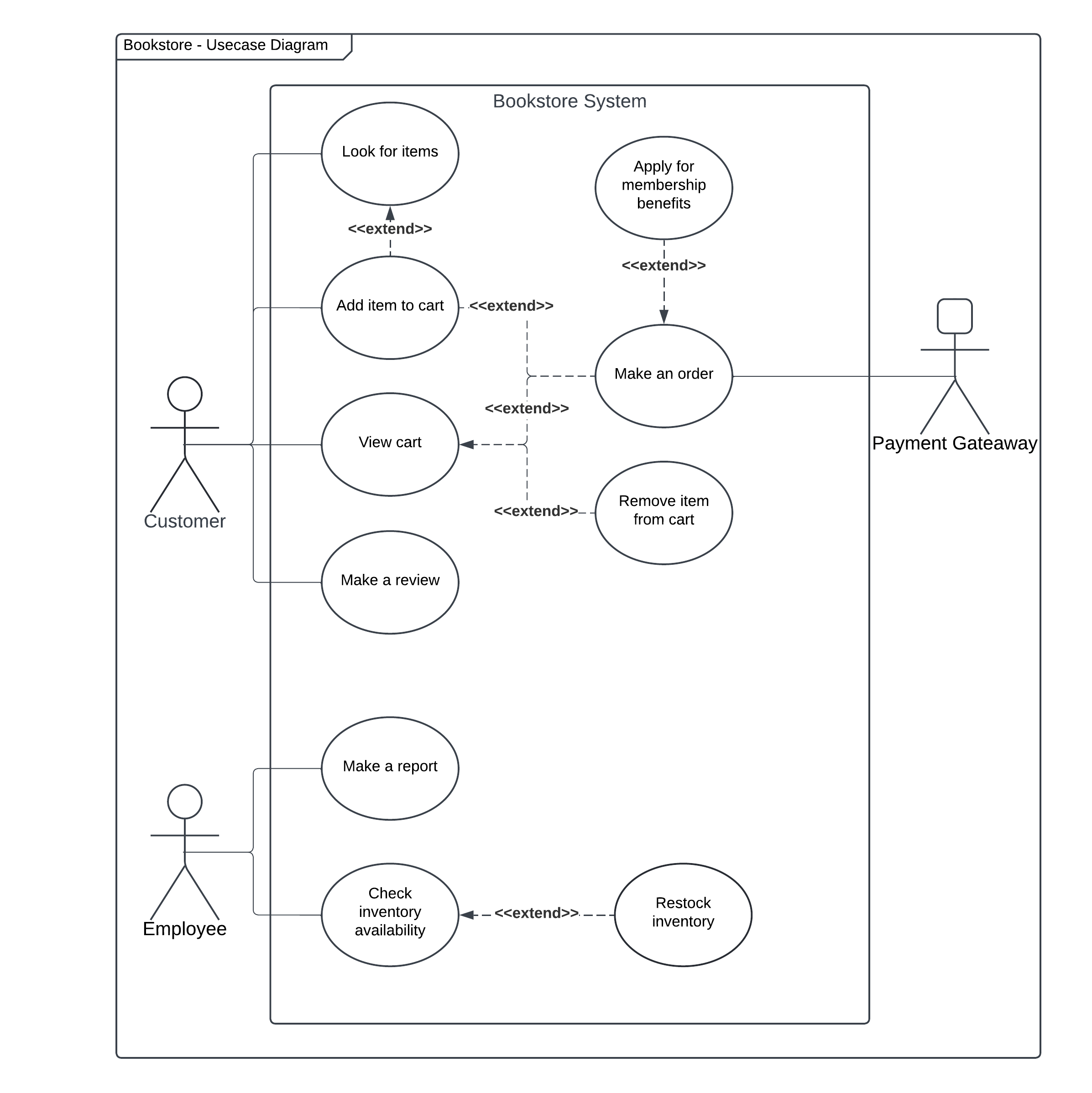
**Bookstore**

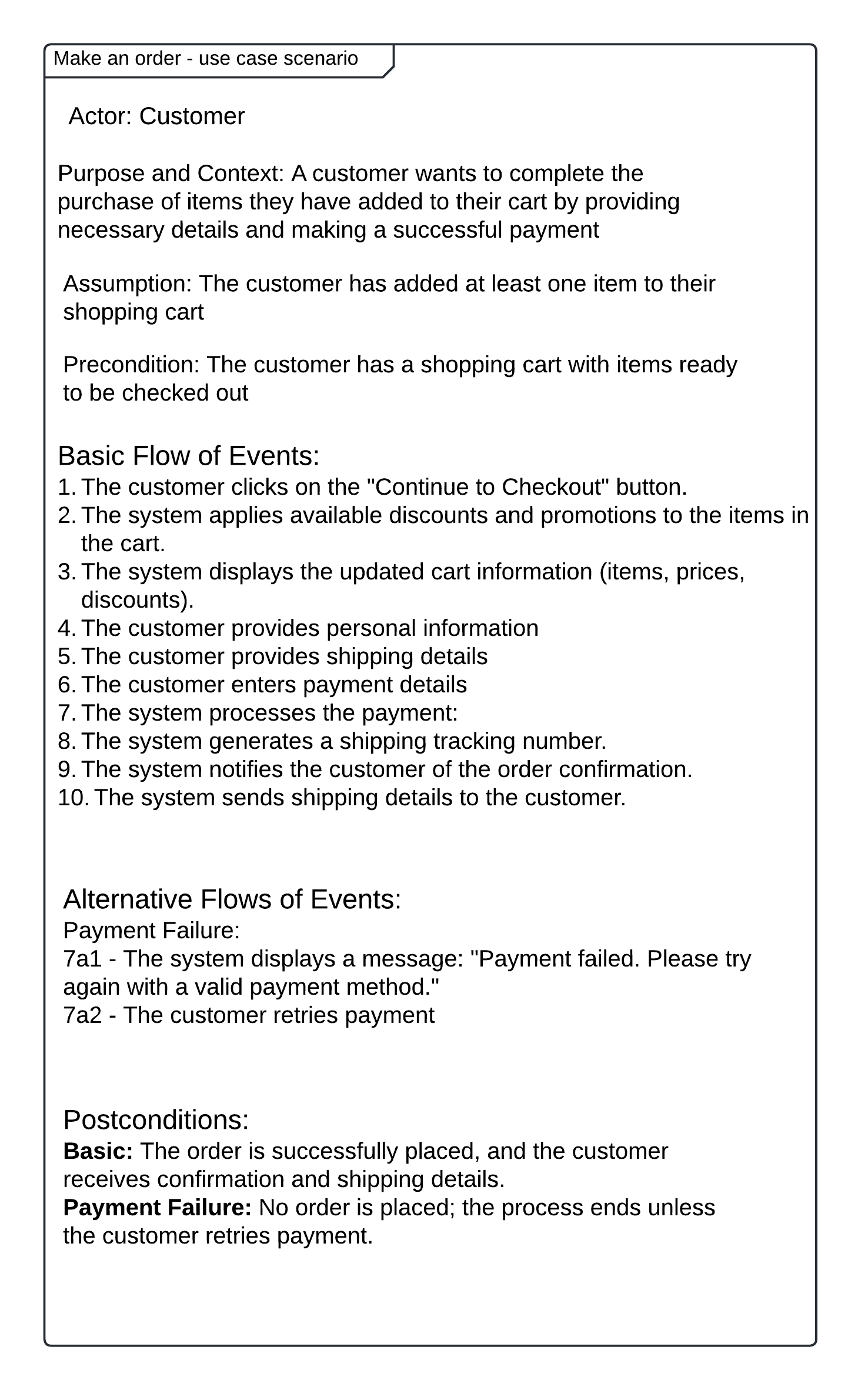
*User Requirements*

Bookstore is aiming to establish an online system to streamline its operations and improve customer experience by providing a seamless platform for customers and employees. Customers will be able to browse and purchase products, while employees will manage inventory, orders, and daily tasks efficiently. The system will allow customers to search through a wide selection of products, including books, accessories, and other items, and view detailed descriptions with attributes like title, price, and availability. For books, additional details such as genre and author will be provided, while accessories will include material type. Customers can easily add items to their cart, modify the contents, and proceed to place an order by providing shipping details and selecting a payment method, such as cash, card, or membership discounts. During checkout, the system will dynamically apply eligible promotions or discounts based on predefined rules, such as coupon codes or membership status, ensuring a personalized shopping experience.

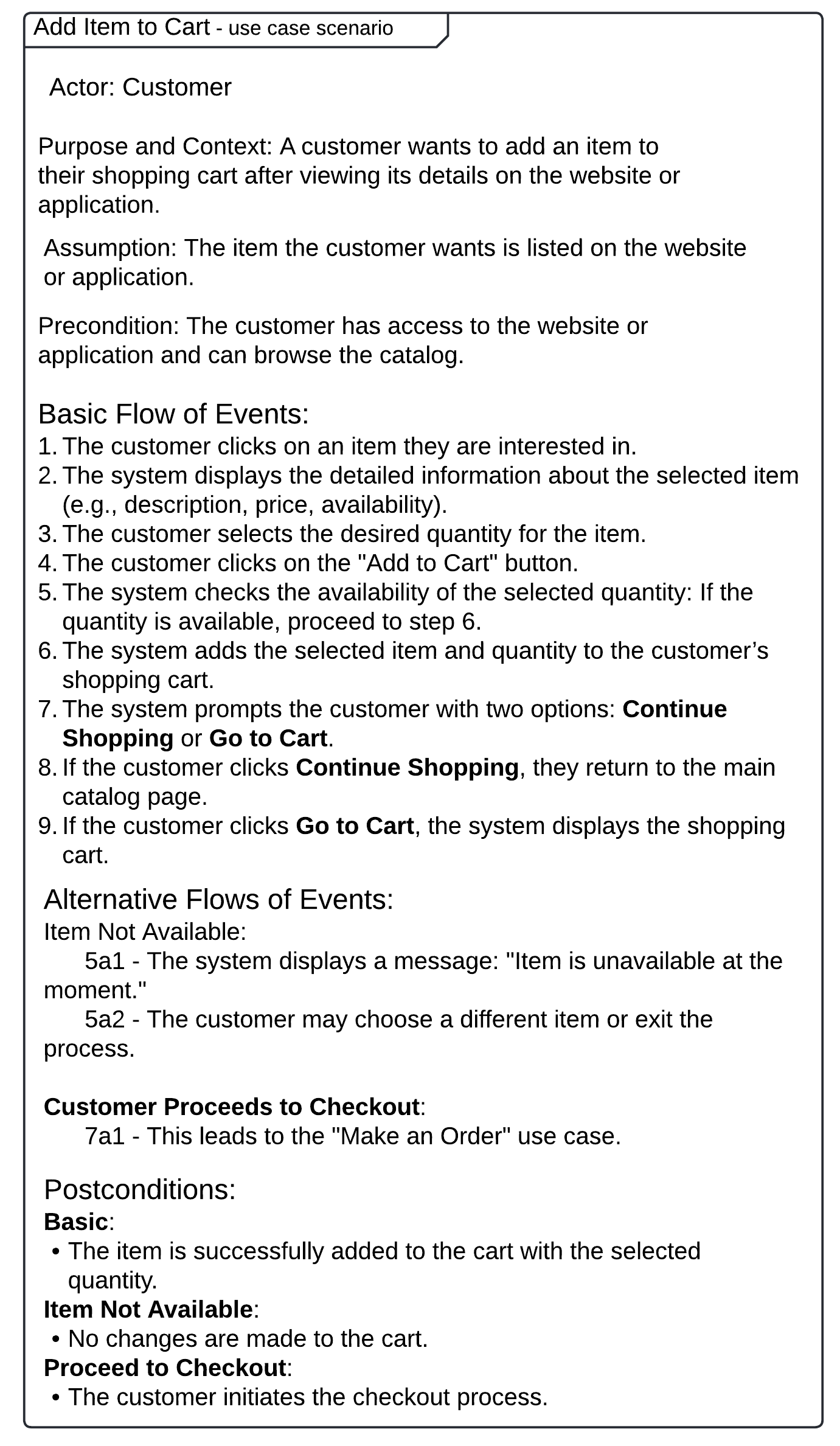
*Use case diagram*

**

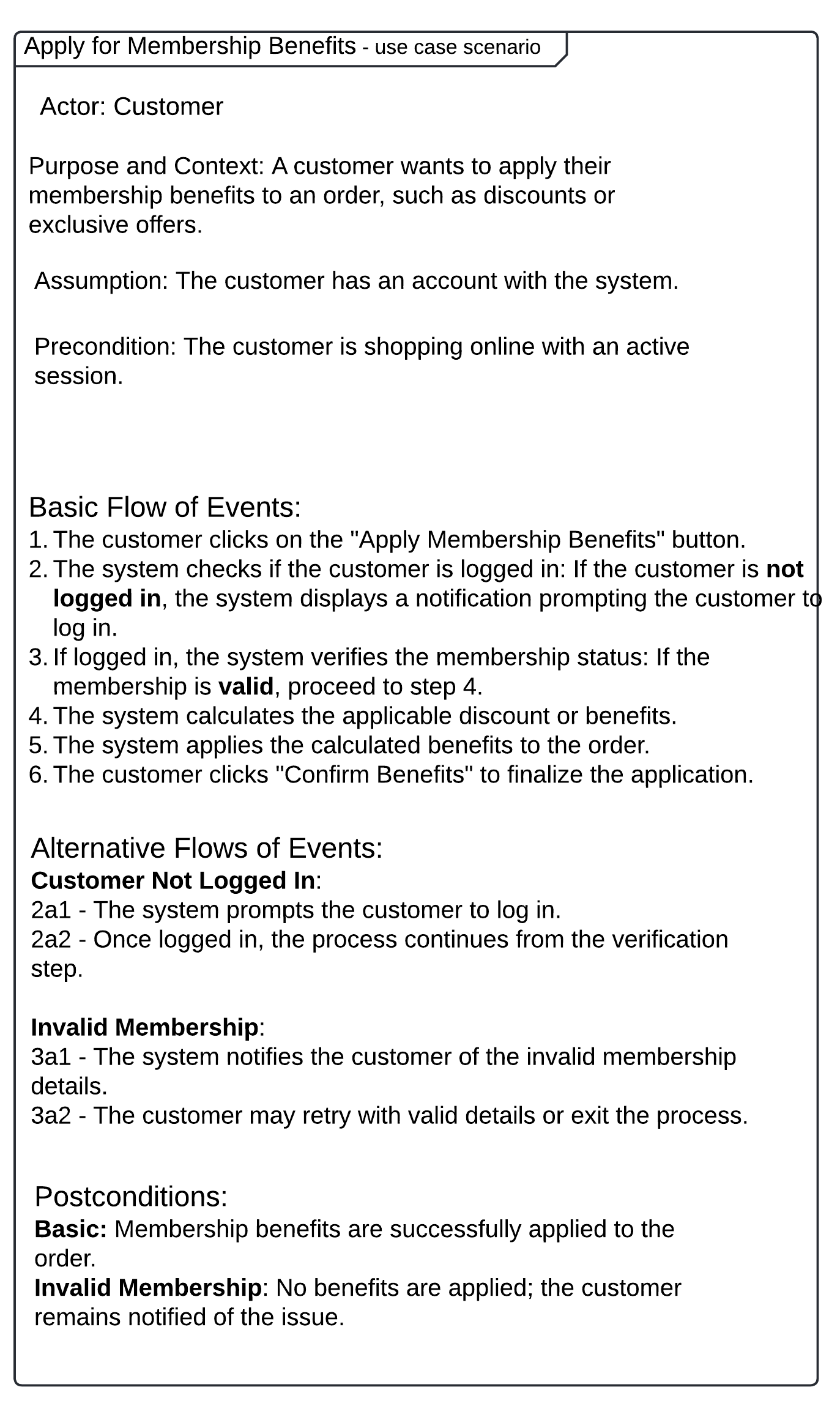
*Use Case Scenario: “Make an order”*

**

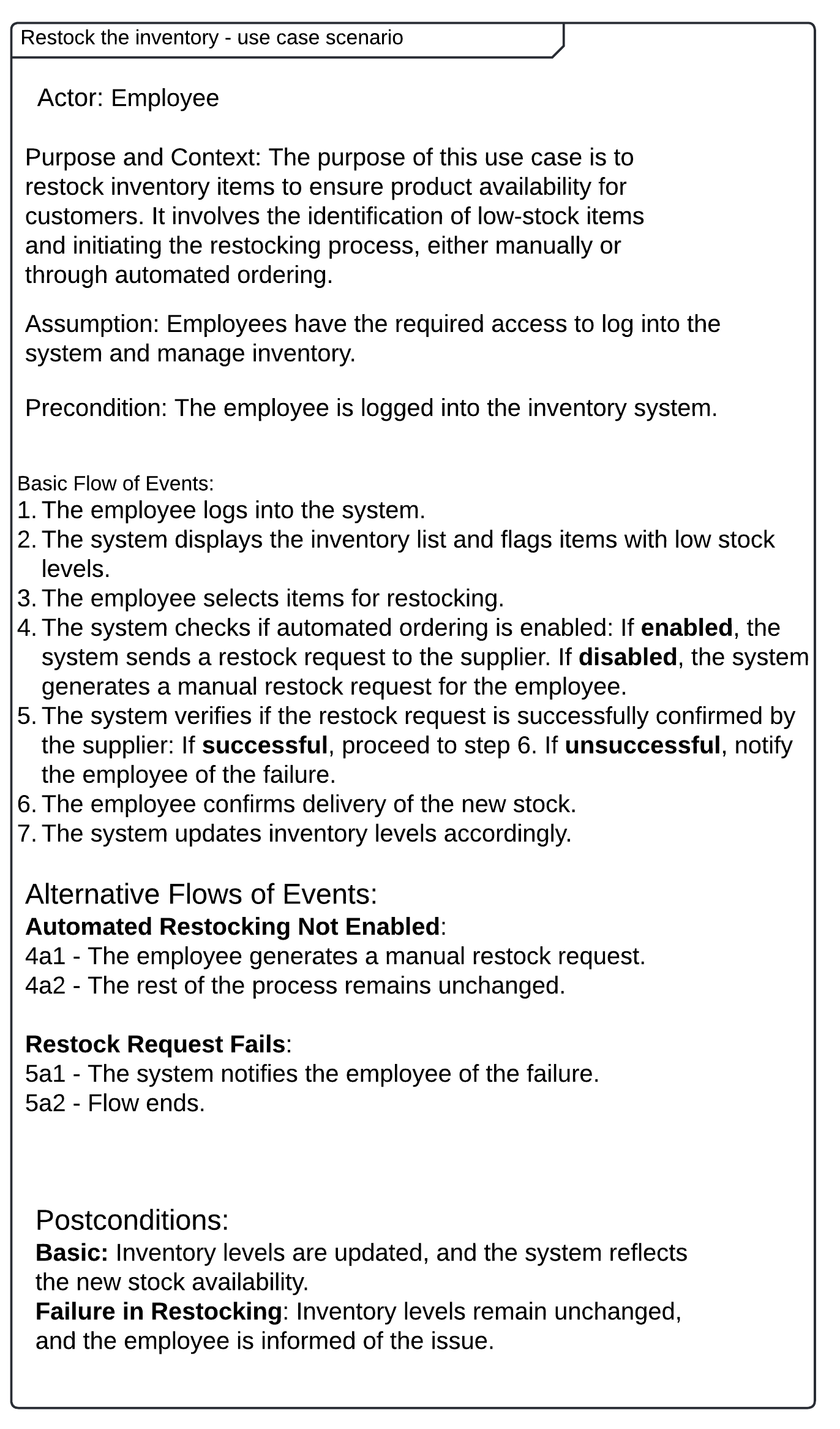
*Use Case Scenario: “Add item to cart”*

**

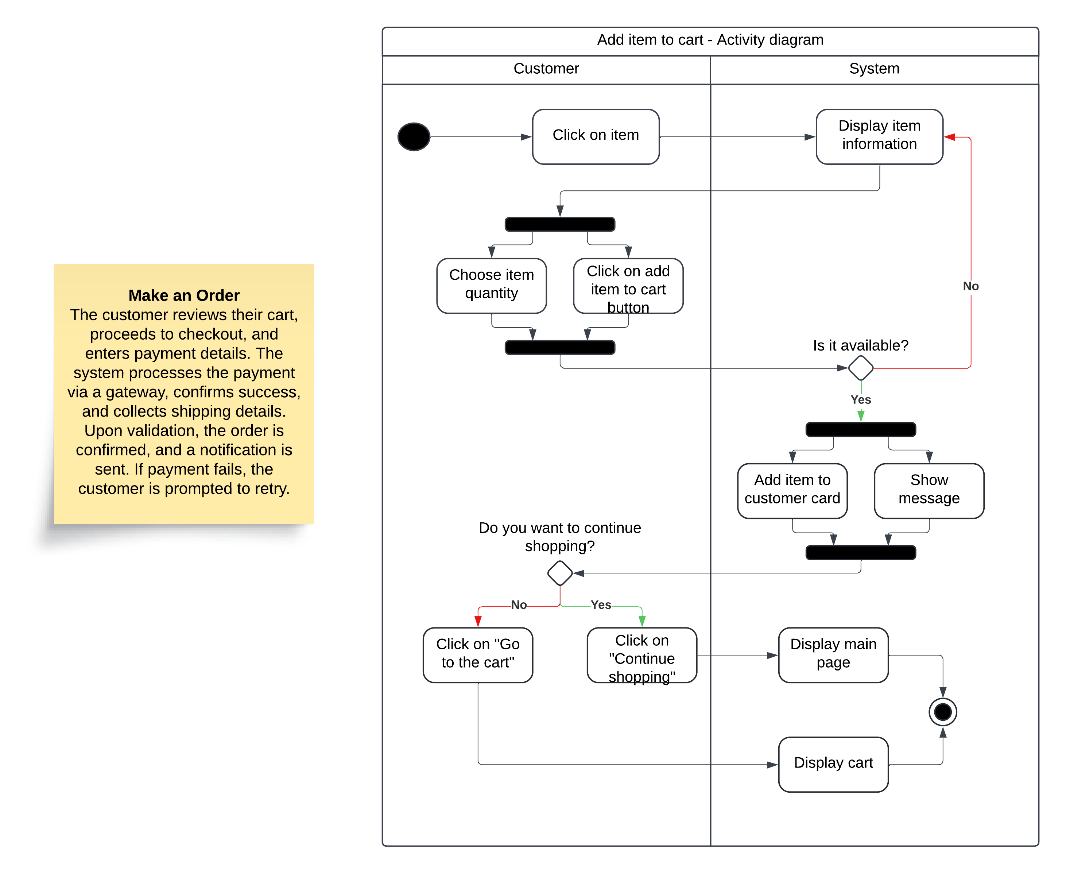
*Use Case Scenario: ”Apply for membership benefits”*

**

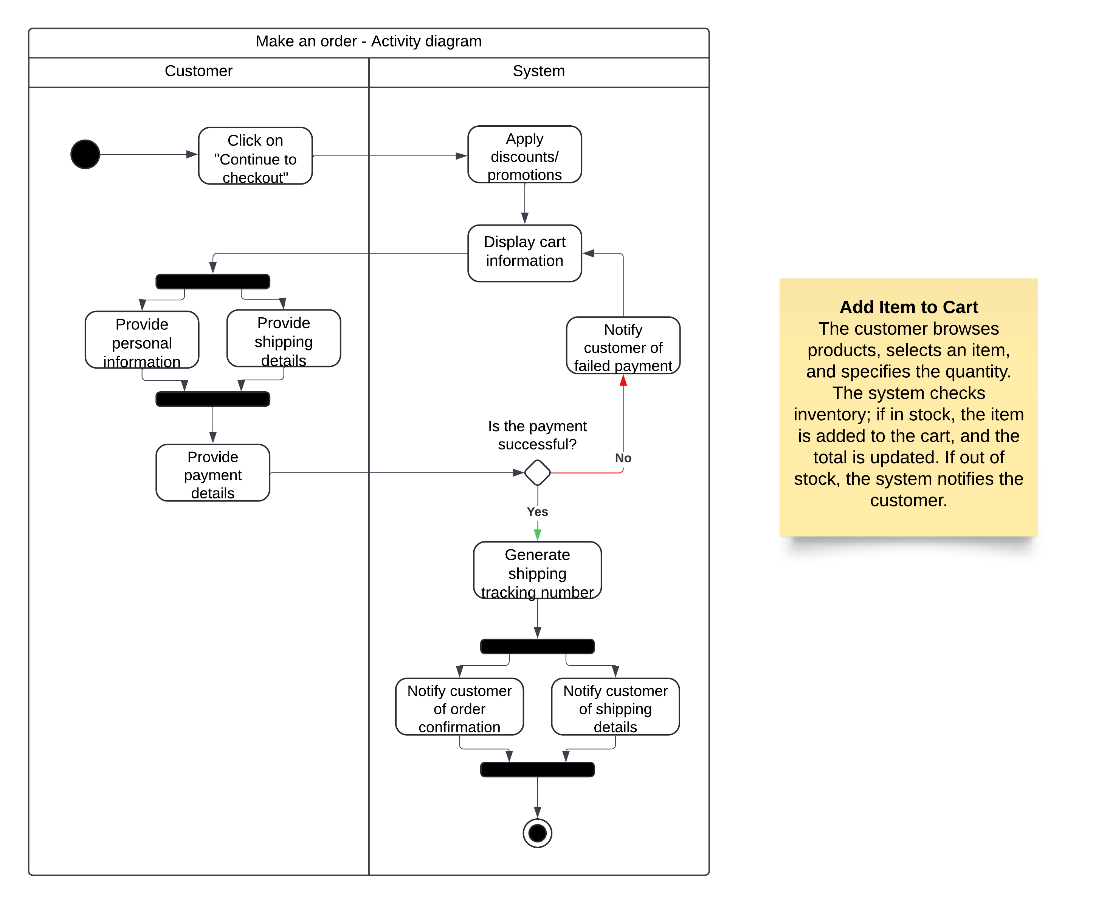
*Use Case Scenario: “Restock the inventory”*

**

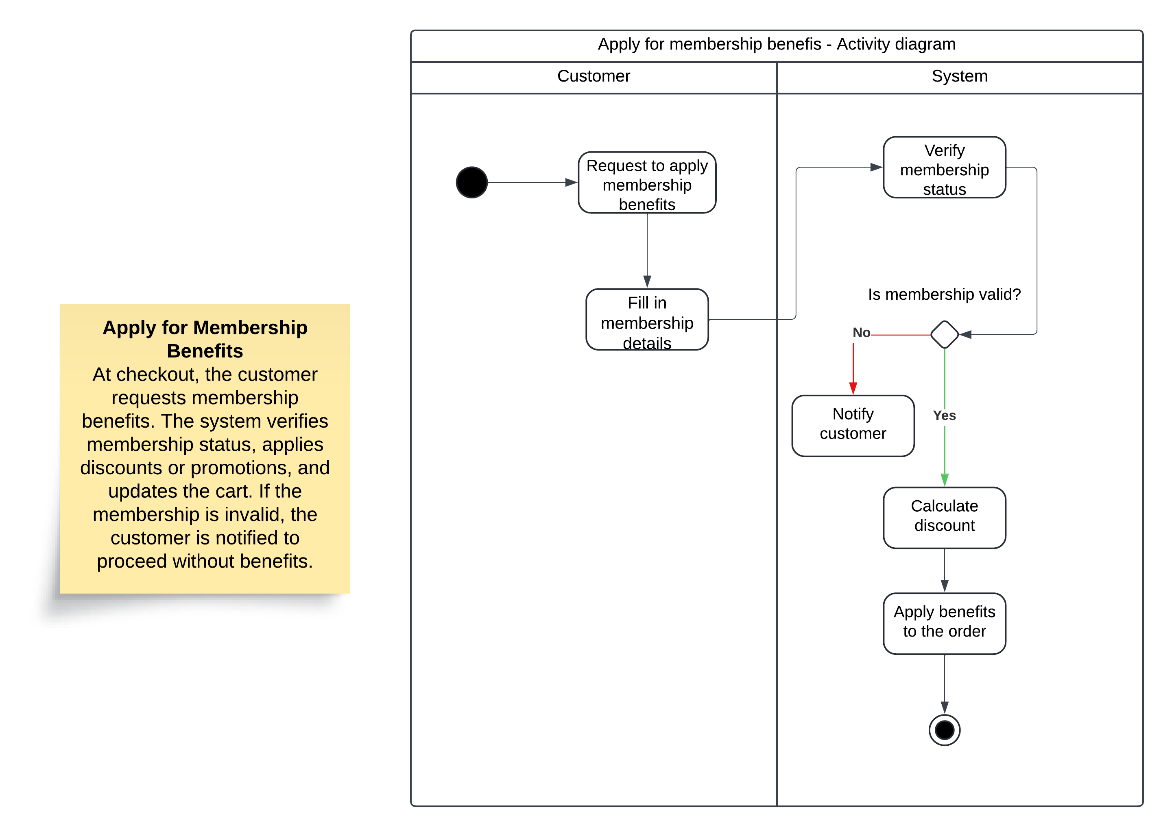
*Activity Diagram: “Add item to cart”*

**

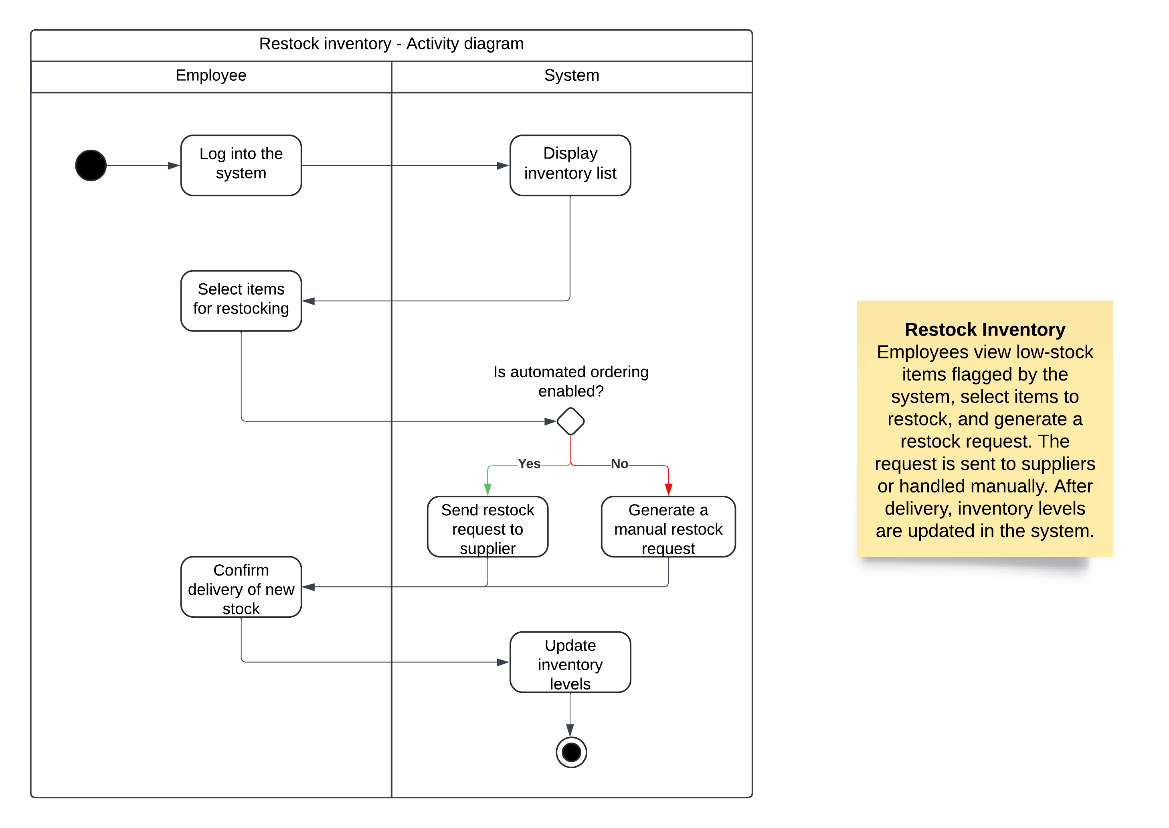
*Activity Diagram: “Make an order”*

**

*Activity Diagram: “Apply for membership benefits”*

**

*Activity Diagram: “Restock inventory”*

**

*State Diagram*

***A diagram with text and words

Description automatically generated with medium confidence***

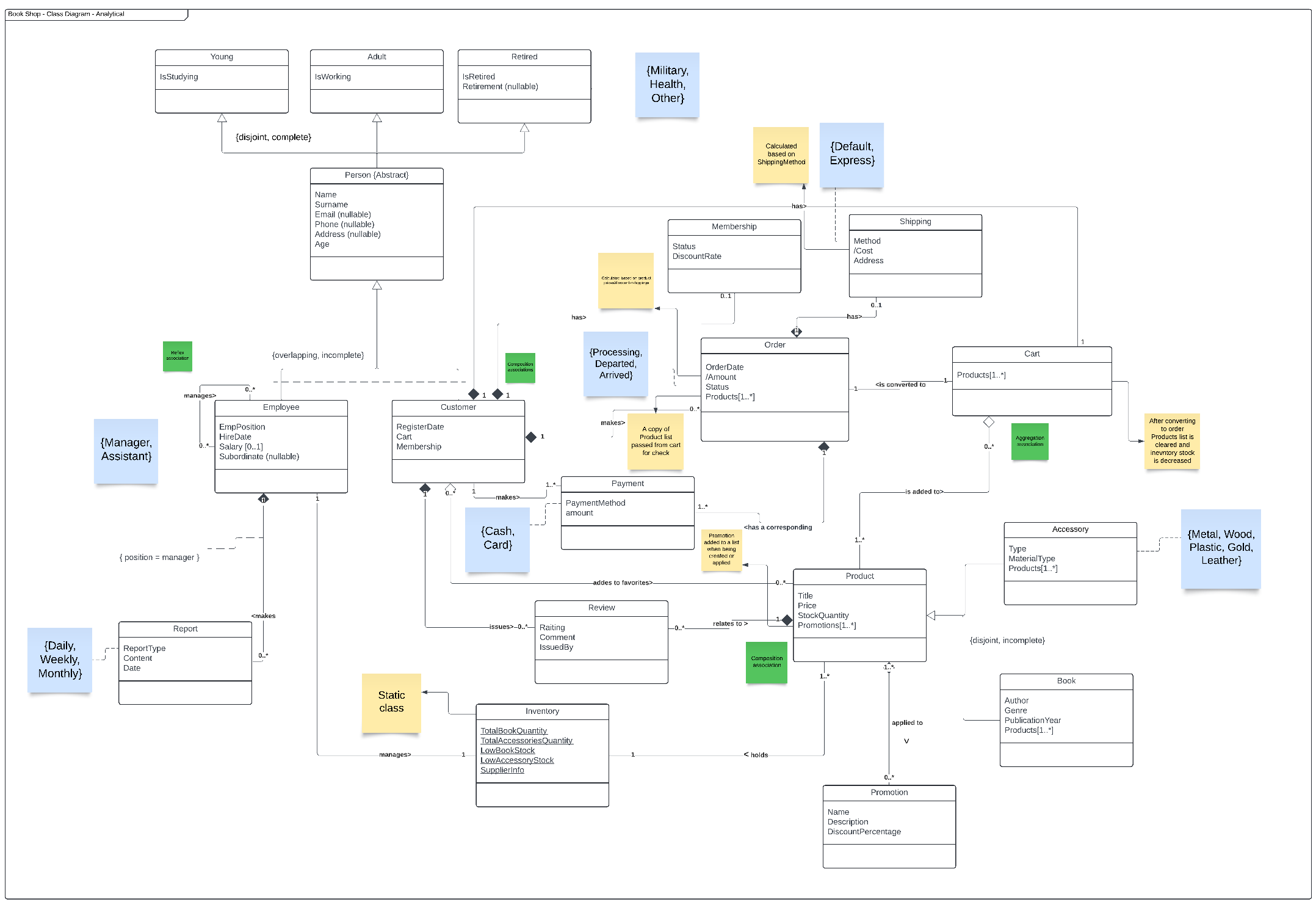
*State Diagram - Design Description*

The state diagram describes the lifecycle of a **Cart Object**. The cart starts in the "Initialized" state and transitions through different states based on customer actions:

1. ***Active*** *-* The main operational state of the cart, allowing customers to add or remove products.
2. ***Adding*** *-* Triggered when a product is being added to the cart.
3. ***Removing***- Triggered when a product is being removed from the cart*.*
4. ***Converting*** *-* The cart is being finalized, and inventory is validated during this process.

If validations (e.g., promotions, stock availability) fail, the cart returns to the **Active** state. The lifecycle ends when the cart is successfully converted or remains active for further modifications.

*Analytical Diagram*

**

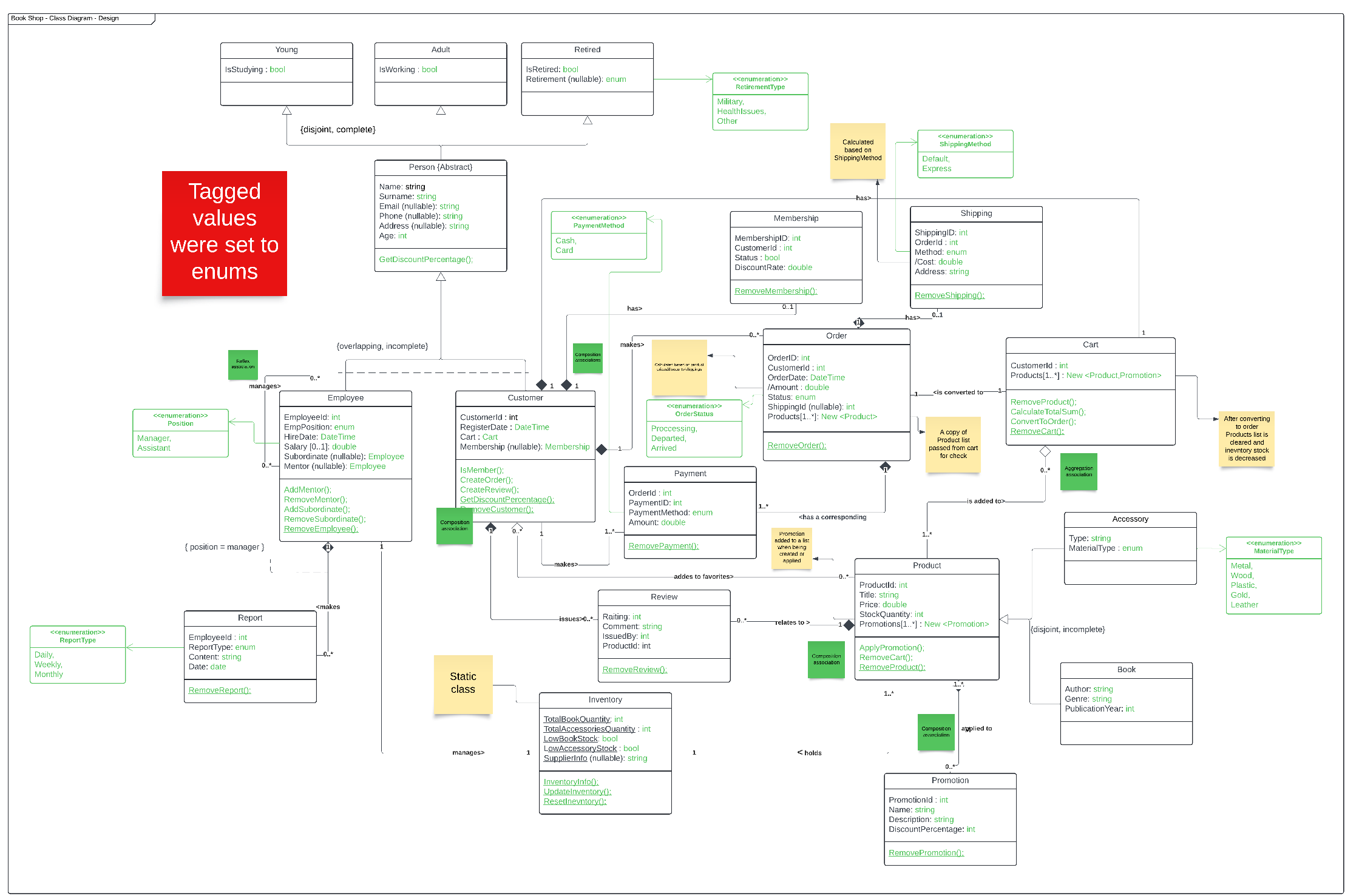
*Analytical Diagram - Design Description*

The above analytical class diagram visualizes the Bookshop system, where customers interact with products, place orders, and manage their carts. Employees oversee operations like inventory, orders, and reporting. The system organizes data around key entities such as Person, Product, Order, and Cart.

The **Person** class is abstract and is specialized into Employee and Customer, with disjoint and complete constraints ensuring separation. Products are categorized into Book, Accessory, and Promotion, each with unique attributes like Author, Material Type, and Discount Percentage. Customers can add products to their Cart, with each cart directly associating with specific CustomerId and a list of Products. Orders reference products, customers, and optional shipping methods (Default or Express).

This diagram includes UML constructs like tagged values (e.g., Order Status, Material Type) and associations that will be further refined in the design class diagram, focusing on attribute implementations and C# specifics.

*Design Diagram*

**

*Design Decisions*

This section highlights how the design class diagram was implemented in the Bookshop system using C#. It focuses on specific decisions related to attributes, validations, optional and derived attributes, multi-value attributes, and tagged values. Each choice was made to ensure alignment with system requirements and proper functionality.

*Attribute Validations*

Where possible, the project has implemented attribute-level validation directly within the class properties using C# accessors (get and set). This ensures data integrity during runtime. Notable examples include:

* ***CustomerId***: Validated against existing customers to ensure the ID is present in the system. Invalid IDs throw an ArgumentException.
* ***ShippingId***: Allows nullable values but validates against existing shipping records when provided. Invalid ShippingIds throw an ArgumentException
* ***OrderDate***: Prevents future dates by throwing an ArgumentException for invalid values.
* **Amount**: Prohibits negative values to maintain financial accuracy. Negative values throw an ArgumentOutOfRangeException.

*Optional Attributes*

Optional attributes in the system are implemented as nullable properties in C#. For instance:

* **Email**: An optional attribute that is nullable, allowing it to be left blank if a person does not provide an email address.
* **Address**: A nullable attribute that can remain unset if no address is specified by the person.

*Multi-Value Attributes*

The project manages multi-value attributes, such as the list of Product objects in an order, by using collections (List<Product>). This ensures scalability while maintaining associations between orders and their items. The Products attribute captures the relationship between an order and the purchased products.

*Derived Attributes*

Derived attributes are implemented using C#'s get functionality. For example:

* **Order Total (Amount)**: Automatically calculated from the sum of the product prices in the Products list, accounting for potential discounts or promotions.

*Tagged Values*

Tagged values are implemented using enumeration classes in C#. For example:

* **Order Status**: States like Pending, Accepted, Shipped, or Cancelled are represented as enums, ensuring clear and manageable state transitions within the system.
* **Material Type**: Categories such as Paper, Plastic, or Metal are represented as enums, providing a clear and structured way to classify materials used in the products.

*Reverse Connection Integrity:*

* Ensured that all reverse connections between associated classes are implemented and validated correctly. This ensures that when relationships (e.g., Customer and Order) are created or deleted, both sides stay consistent.

*Improved Safety:*

* Removed unsafe or unnecessary methods across the project to reduce the risk of unintended behavior and ensure that only essential operations are exposed.

*Code Organization:*

* Refactored constructors to consistently appear under field and property declarations for improved readability and logical grouping within each class.

*Encapsulation Enhancements:*

* Adjusted accessors and methods to better adhere to encapsulation principles, limiting direct access to sensitive fields and ensuring controlled interaction through clearly defined methods and properties.

*Test Updates:*

* Updated existing tests to align with the modified accessors and methods. Fixed any inconsistencies in test logic to ensure compatibility with the refactored code.
* Additionally, added reverse connection tests to validate the integrity of relationships between associated classes, ensuring that creation and deletion operations maintain consistency.

COMMENT

The quality of the images is unfortunately low, despite my efforts to fix it. Therefore, I am providing the link to the actual Lucid chart so you can read it more clearly.

https://lucid.app/lucidchart/ce55fd1c-d19e-4cd6-8c6d-a6b2b9acba9c/edit?invitationId=inv\_fe923c78-1a0c-46b5-9785-c952572483a2&page=0\_0#