Autopart Shop

Software Requirements & Implementation Specifications

Jan Jurinok s26823 13c

User Requirements

The AutoPart Shop platform enables efficient buying, selling, and management of automotive parts for private customers, part suppliers, and shop employees alike. The service is focused on a desktop interface and designed for a single language and currency.

Customers

Customers use the platform to discover, purchase, and track automotive spare parts for their vehicles.

1. Customers can register for an account using an email address and manage their profile details and shipping addresses.
2. They can search or browse an organized catalog of parts, and check real-time availability and pricing for each part.
3. Customers may add one or more parts to a shopping cart. Quantity controls prevent adding more than the available stock.
4. When ready to purchase, customers proceed to checkout, where they confirm the shipping address, select a delivery method, and provide payment details.
5. Payment options include credit card, PayPal, and bank transfer. If a payment fails, customers are notified immediately and may retry.
6. Customers can review all active and historical orders, monitor order status (e.g., Created, Processing, Shipped, Delivered, Cancelled), and track shipments using tracking numbers provided for deliveries.
7. Refunds or changes after purchase require contacting shop employees, but customers may modify their cart anytime before checkout.

Suppliers

Suppliers are organizations or individuals who sell automotive parts through the platform.

1. Suppliers can log in to the platform to manage their selection of parts for sale, including updating part details, prices, and stock quantities.
2. All new or modified part listings must meet requirements around clear descriptions and valid pricing.
3. Suppliers have access to dashboards showing their sales performance, product popularity, and inventory status.
4. Suppliers can manage information related to order shipments, including scheduling and providing tracking details for deliveries.
5. Suppliers are only able to view and manage their own catalogue and associated orders.

Employees

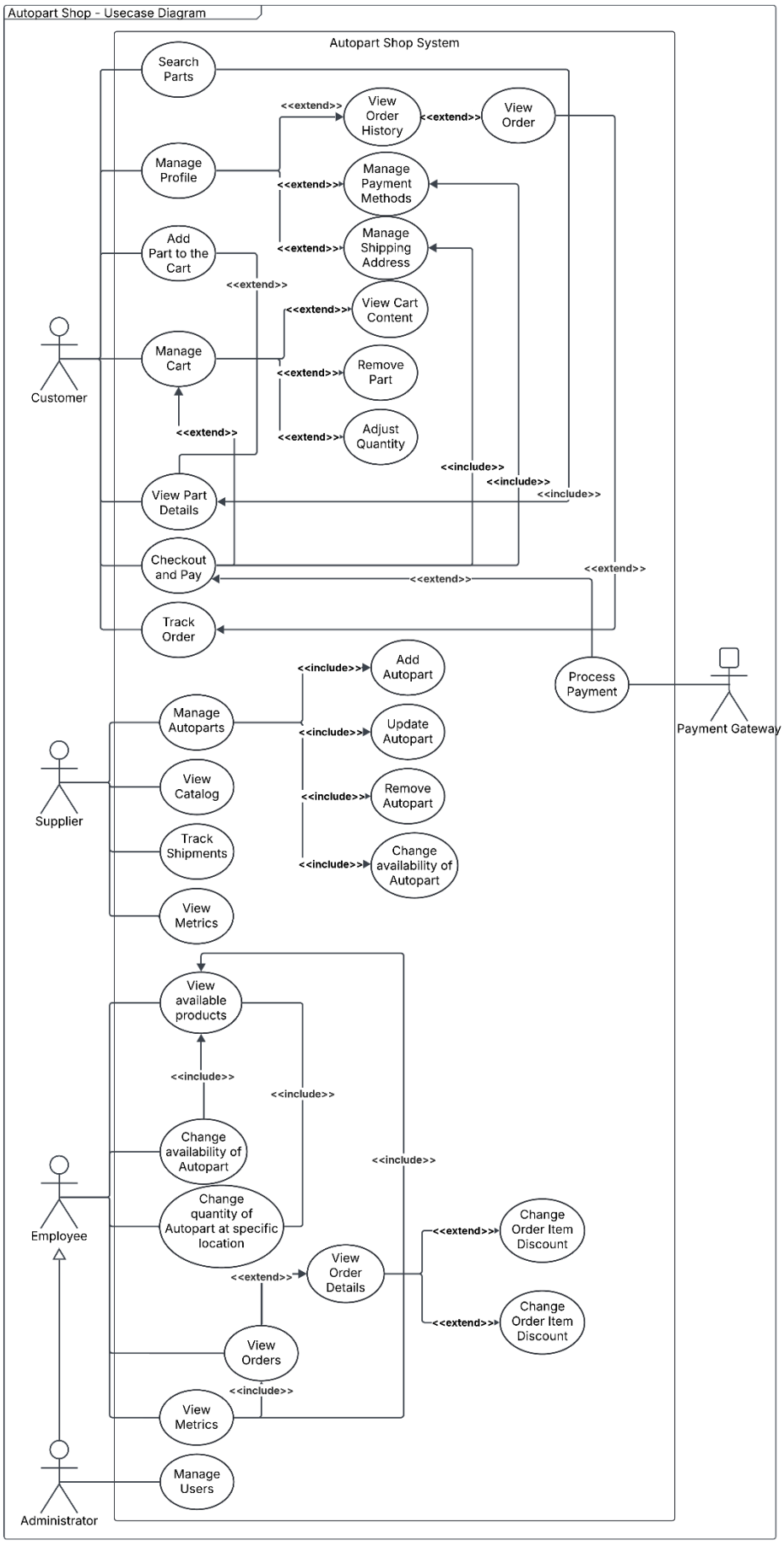
Employees oversee the proper operation of the shop and its business processes.

1. Employees manage central inventory, including adjusting stock levels and updating item locations when parts are moved between warehouses or pick-up points.
2. They review and confirm incoming orders, handle exceptions such as low stock or backorders, and coordinate shipments.
3. Employees are authorized to process customer refund requests, apply discounts to specific order items, and override supplier pricing when negotiating special deals.
4. The platform provides employees with access to analytical reports on sales, inventory status, and supplier performance.

General Notes

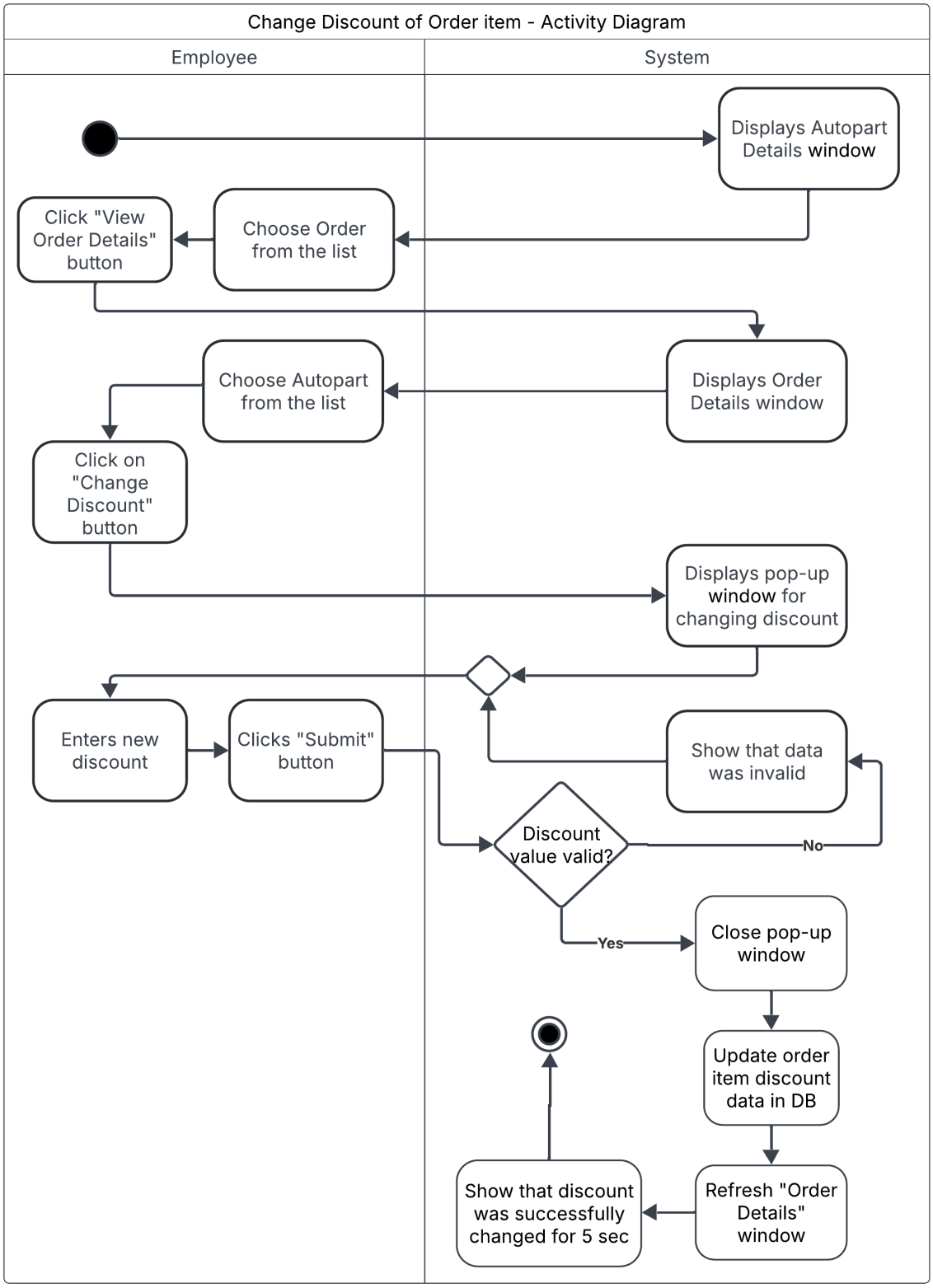
1. The platform does not include chat, messaging, notification systems, or support for multiple languages or currencies.
2. The focus is on clarity, transparency, and the prevention of errors for all users during the main business processes.

Usecase Diagram



Usecase Scenario & Activity Diagram:

“Change Discount of Order item”



Change Discount of Order item - Use case scenario

Actor:
Employee

Purpose and context:
An employee wants to change the discount of a specific order item for a customer.

Assumption:
The employee has access to the system and relevant permissions to modify order items.

Precondition:
1-The employee is logged into the system
2-The autopart and order information exist in the database
Basic flow of events:
1-The System displays the Autopart Details window
2-The employee selects an autopart from the list
3-The employee clicks the "View Order Details" button
4-The System displays the Order Details window
5-The employee chooses an order from the list
6-The employee clicks the "Change Discount" button
7-The System displays a pop-up for changing the discount
8-The employee enters the new discount
9-The employee clicks the "Submit" button
10-The System checks if the discount value is valid
11-The System closes the pop-up window
12-If valid, the System updates the order item discount in the database
13-The System refreshes the Order Details window
114-The System show that discount was successfully changed for 5 sec

Alternative Flow of Events:
Discount value invalid
9a1-The System shows that the entered discount is invalid
9a2-The employee may enter a new discount value or cancel the operation

Post condition:
Basic: The order item's discount is updated to the new value in the system
Discount value invalid: No change is made to the order item's discount; user remains in the discount pop-up

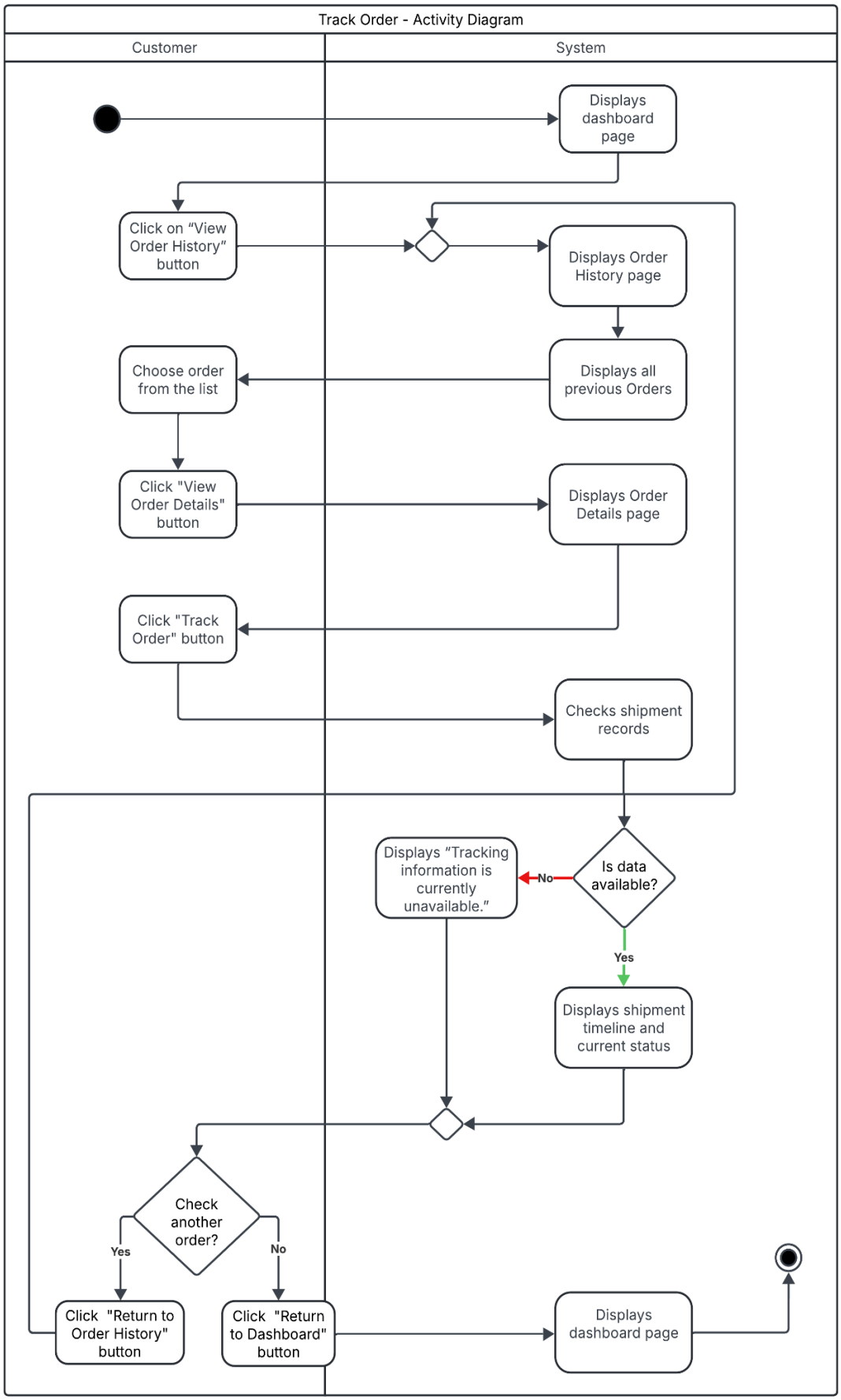
An employee, who is already logged into the system and has permission to modify orders, opens the autopart details window. On this details page, the employee selects order from the list and clicks the "View Order Details"button to see which orders include this autopart.

Once the system shows the order details, the employee selects a specific order and clicks the "Change Discount" button next to the relevant order item. A pop-up window appears, prompting the employee to enter a new discount amount. After entering the new discount, the employee clicks the "Submit" button.

The system checks if the entered discount value is valid. If the value is valid, it updates the order item's discount in the database, closes the pop-up window, and refreshes the order details so the employee can immediately see the updated discount. Also it shows that discount was successfully changed for 5 sec.

If the discount value is invalid, the system displays an error message in the pop-up window, letting the employee know that the entered value is not acceptable. The employee can then either correct the discount and submit again, or cancel the operation. The system will only update the discount if a valid value is provided; otherwise, it will not make any changes and the employee will stay in the pop-up until a valid discount is submitted or the process is canceled.

Usecase Scenario & Activity Diagram: “Track Order”



Track Order - Use case scenario

Actor:
Customer

Purpose and context:
A customer wants to view their past orders and track shipment status for a specific order.

Assumpition:
The customer has one or more completed orders in their account.

Precondition:
1-The customer is logged in and on the account dashboard
2-The system has recorded at least one order with shipment information

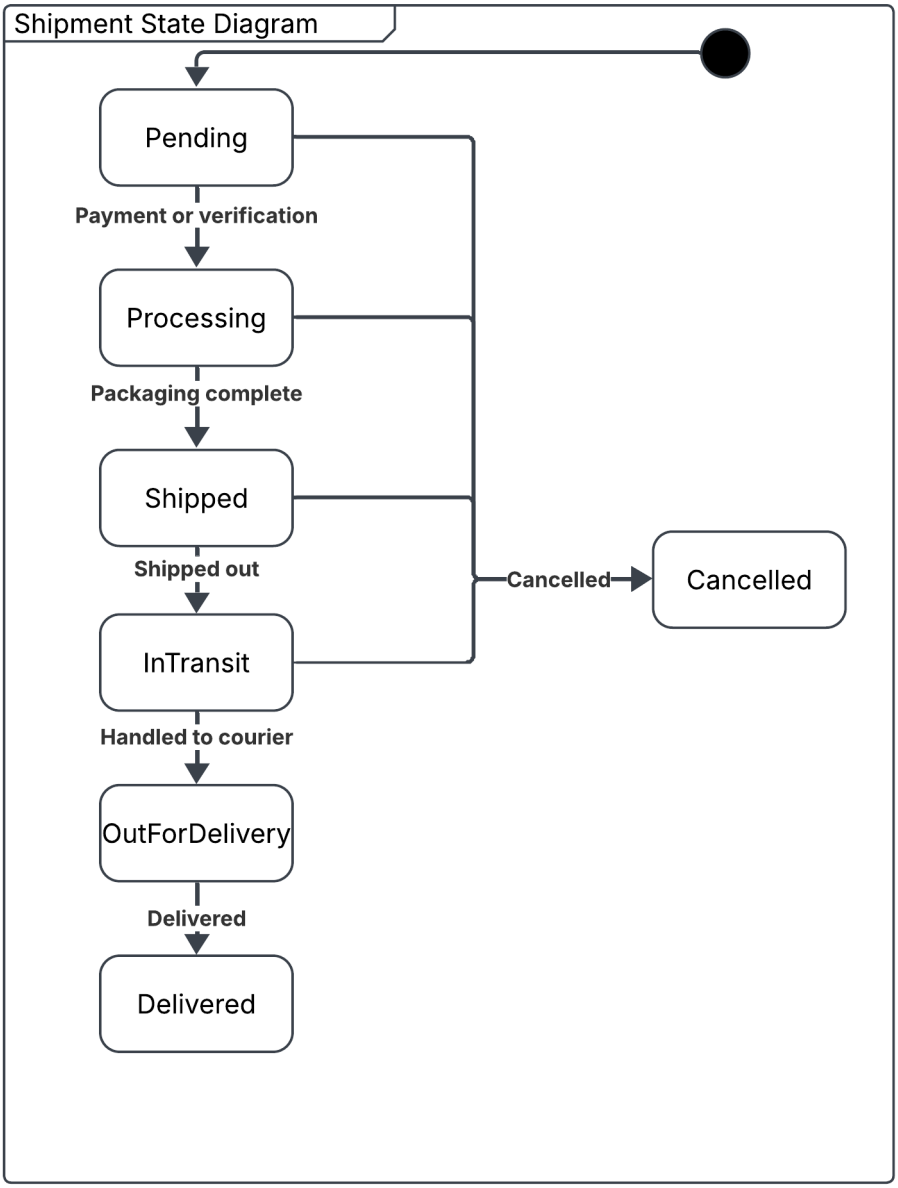
Basic flow of events:
1-The customer clicks on “View Order History”
2-The System displays a list of previous orders (order name, description, date, status)
3-The customer clicks "View Order Details" at chosen order from the list
4-The System displays the order details page with items, totals, and a “Track Order” button
5-The customer clicks on “Track Order”
6-The System retrieves shipment records associated with that order
7-The System displays shipment timeline with tracking numbers and current statuses
8-The customer clicks "Return to Dashboard"
9-The System displays dashboard page

Alternative Flow of Events:
Shipment data unavailable
6a1-The System displays “Tracking information is currently unavailable.”

Click on "Return to Order History"
8a1-The customer clicks on "Return to Order History"
8a2-Return to point 2

Post condition:
Basic: Shipment tracking details for the selected order are displayed
Click on "Return to Order History": Customer redirected to Order History page
Shipment data unavailable: Customer remains on Order Details page without tracking info

Shipment State Diagram



Shipment– State Diagram Description

The state diagram for the Shipment outlines how a shipment progresses through an order fulfillment process, accounting for both standard transitions and exceptional outcomes (such as cancellation).

Lifecycle Overview:

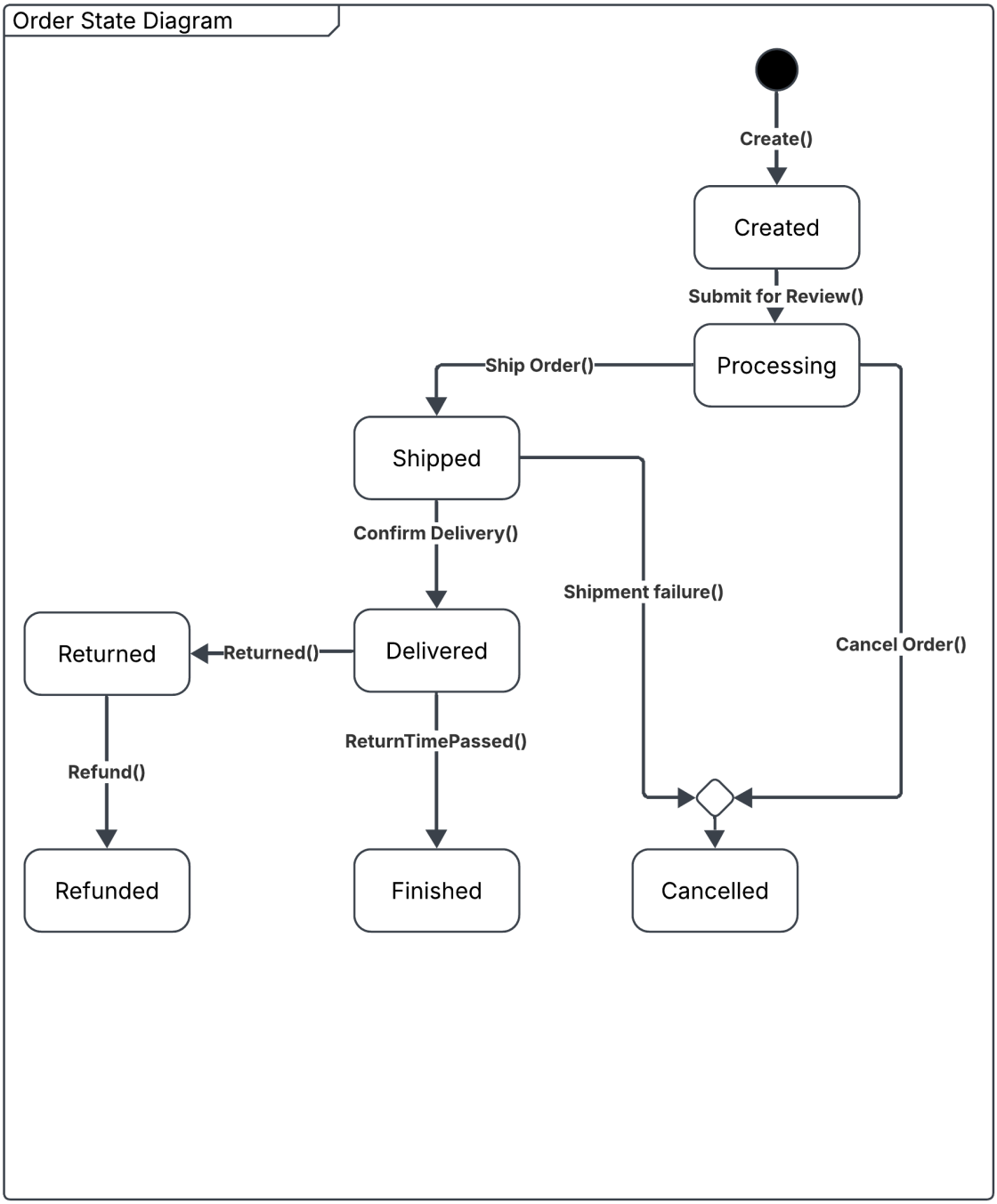
1. **Pending:** The shipment starts here, awaiting payment confirmation or necessary verifications.
2. **Processing:** After payment or verification, the shipment undergoes packaging and preparation.
3. **Shipped:** Once packaged, the shipment is marked as ready to leave the warehouse.
4. **InTransit:** The shipment physically leaves the warehouse and is being transported.
5. **OutForDelivery:** Upon arrival at the destination hub, the shipment is handed to a local courier for final delivery.

6. **Delivered:** The shipment reaches the recipient, completing its lifecycle.
7. **Cancelled:** At any stage prior to delivery, a shipment may be cancelled due to customer requests, payment issues, or logistical problems. Entering this state halts all further processing

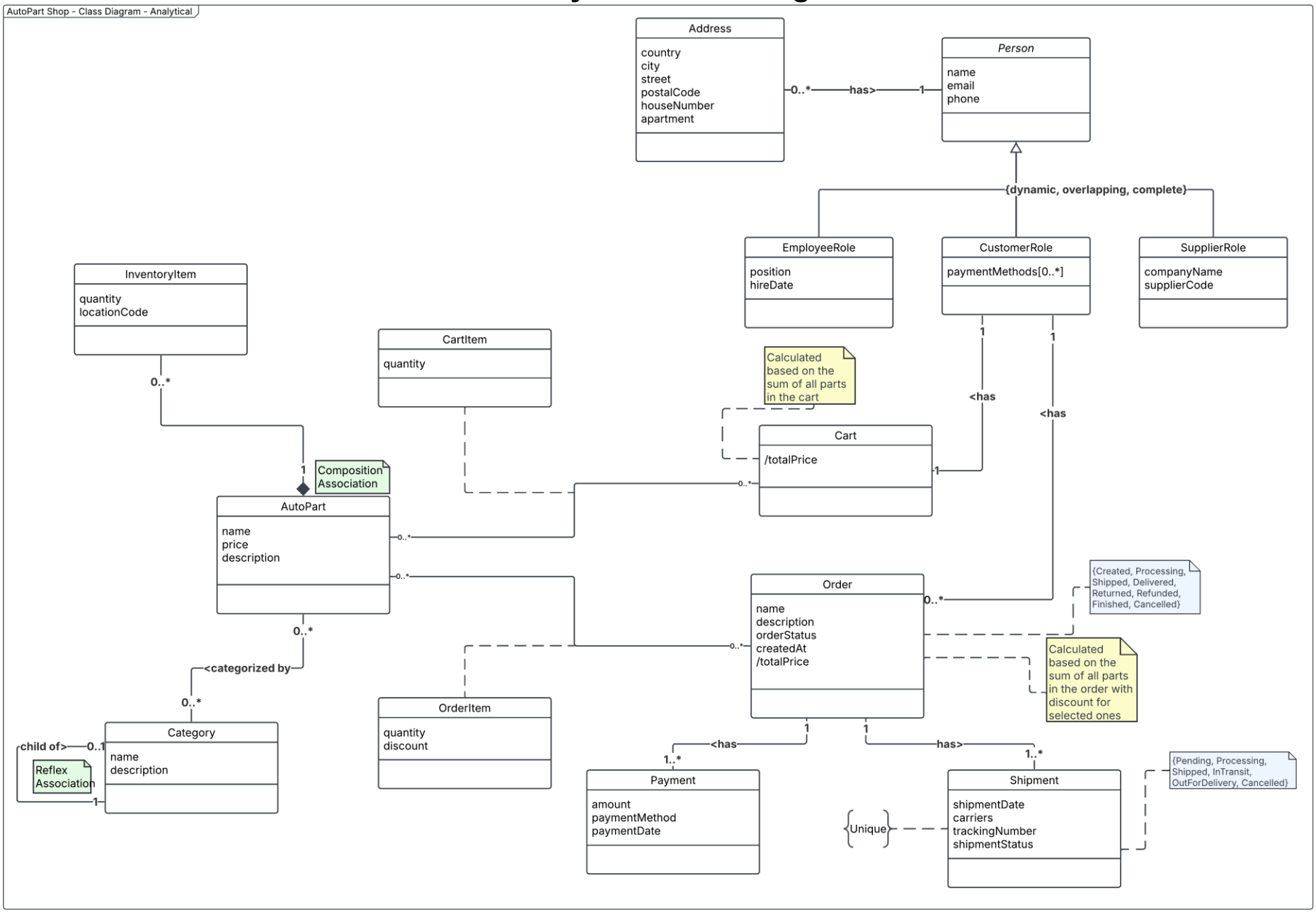
Transitions

- **Initial State → Pending (Order Placed):**
  A shipment is created when an order is placed.
  Condition: The system successfully registers a new order requiring shipment.
- **Pending → Processing (Payment or Verification Complete):**
  Shipment moves to Processing after payment confirmation or other necessary checks.
  Condition: Payment received or verifications passed.
- **Processing → Shipped (Packaging Complete):**
  Shipment is marked as Shipped after packaging and labeling.
  Condition: Packaging successfully finished.
- **Shipped → InTransit (Shipped Out):**
  Shipment leaves the warehouse, starting its journey.
  Condition: Courier picks up shipment.
- **InTransit → OutForDelivery (Handled to Courier):**
  Shipment arrives at the destination hub and is assigned for last-mile delivery.
  Condition: Checked in at local delivery center and assigned to a courier.
- **OutForDelivery → Delivered (Delivered):**
  Shipment reaches and is confirmed delivered to the recipient.
  Condition: Delivery confirmation (courier/recipient signature).
- **Pending → Cancelled (Order Cancelled):**
  Shipment can be cancelled before it enters processing.
  Condition: Customer cancellation request or payment failure.
- **Processing → Cancelled (Order Cancelled):**
  Shipment can be cancelled during the processing stage.
  Condition: Initiated by customer support or system.
- **Shipped → Cancelled (Order Cancelled):**
  Shipment can be cancelled or returned even after being marked as shipped.
  Condition: Intercepted, return-to-sender, or logistics issue.
- **InTransit → Cancelled (Order Cancelled):**
  Shipment can be cancelled while in transit.
  Condition: Loss, regulatory problem, or customer refusal before delivery attempt.

Order State Diagram



Analytical Class Diagram



The analytical class diagram for the AutoPart Shop system outlines the structure and core relationships between key entities of the platform.
Central to the system is the **Person** class, which encapsulates essential attributes like name, email, and phone number. Persons can take on multiple dynamic roles within the system—namely customers, suppliers, and employees—represented by specialized subclasses (**CustomerRole**, **SupplierRole**, and **EmployeeRole**), enabling overlapping and complete role assignment.

The **Address** class models the physical addresses associated with persons and entities, including detailed attributes like country, city, street, postal code, house number, and apartment. Each person can have one or more addresses, supporting flexibility for customers, suppliers, and employees.

Auto parts within the shop are represented by the **AutoPart** class, with key attributes such as name, price, and description. Auto parts are tracked within the shop's inventory by the **InventoryItem** class, which maintains information on quantity and storage location. This ensures real-time inventory management and location tracking for efficient fulfillment.

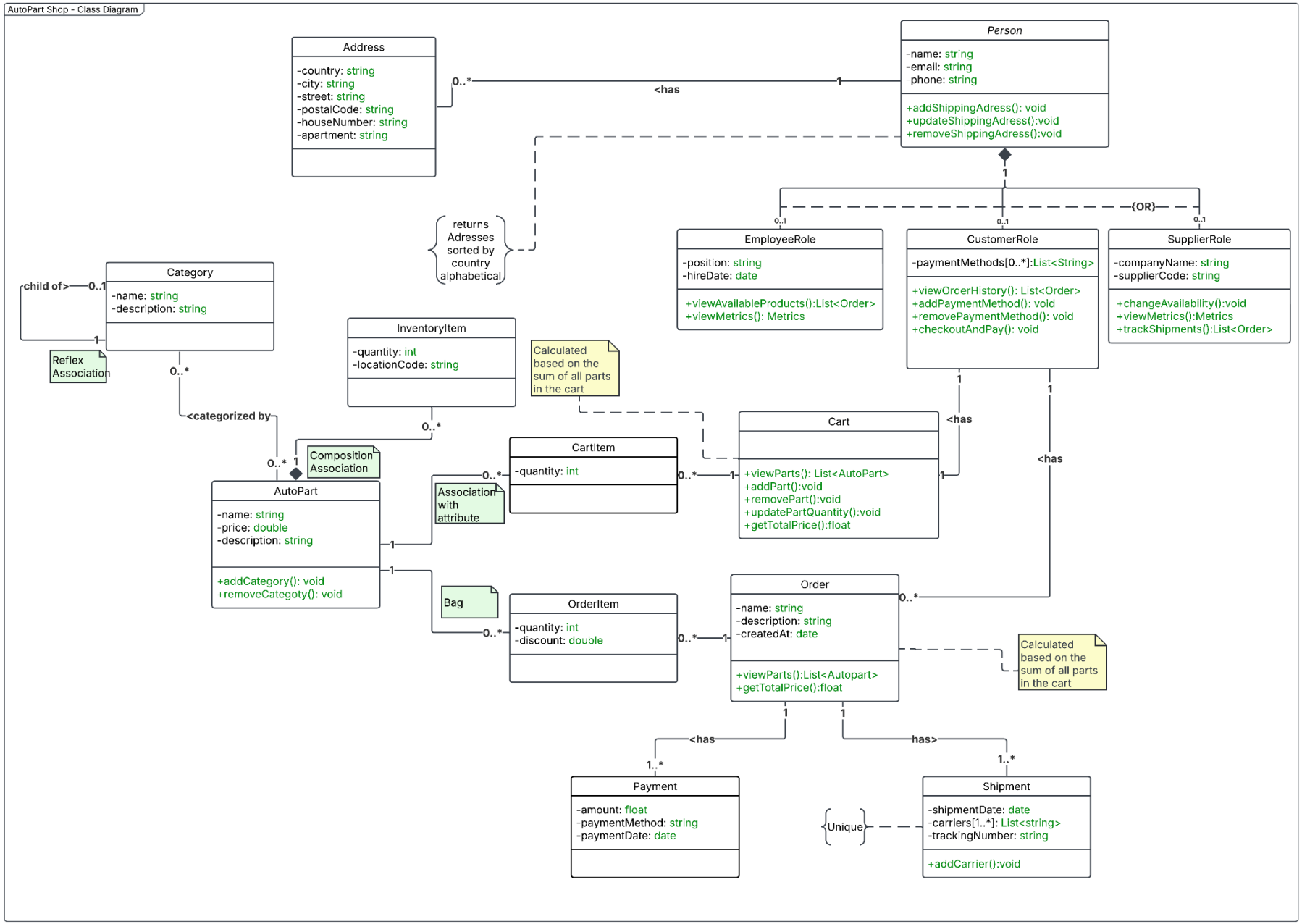
Cart and **CartItem** classes model the shopping cart functionality. Each cart comprises multiple cart items, each referencing a specific auto part and storing the desired quantity. The cart's total price is automatically calculated as the sum of the prices of all items it contains. This composition relationship ensures cart integrity and supports real-time pricing updates as customers add or remove items
.
Orders are central to the transactional logic of the system. The **Order** class includes key order attributes like name, description, order status, creation date, and automatically calculated total price. Orders are made up of multiple **OrderItem** instances, each tracking the quantity of a specific part ordered as well as any applied discounts. The order's status is explicitly tracked using an **OrderStatus** enumeration, which includes all relevant stages such as Created, Processing, Shipped, Delivered, Returned, Refunded, Finished, and Cancelled.

Auto parts themselves are organized into **Category** entities, supporting a flexible catalog structure. The reflex association allows for hierarchical categorization, meaning a category can be a child or parent to another category, enabling nested and organized part listings.

Payment and shipment are modeled as distinct classes associated with orders.
The **Payment** class handles payment-specific details, including amount, payment method, and payment date, ensuring robust transaction tracking.

The **Shipment** class manages shipment details: shipment date, carriers involved, tracking number, and current shipment status. Shipment status is managed with the **ShipmentStatus** enumeration, with states such as Pending, Processing, Shipped, InTransit, OutForDelivery, and Cancelled, supporting detailed order tracking and exception handling.
Inventory automatically reconciles with order and cart activity. The composition association between Cart and CartItem, as well as between Order and OrderItem, guarantees data consistency and entity life-cycle management.

Design Class Diagram - Initial Version



Design Decisions for AutoPart Shop System

The AutoPart Shop platform employs an H2 relational database (accessed via Java Spring) to persist domain objects and manage all business data. This decision shapes how attributes, associations, and relationships are implemented, validated, and retrieved throughout the application. The system is designed to accommodate dynamic, overlapping inheritance for user roles, strong composition for cart and order items, and reflexive category hierarchies.

Attribute Validation

Attribute validation is accomplished through a dual-layered approach:

1. **Application Layer (Spring/JPA Validation):** Bean validation (**@NotNull**, **@Email**, **@Size**, etc.) and custom business logic ensure attributes conform to domain requirements before persistence.
2. **Database Layer:** Properties such as NOT NULL and UNIQUE constraints are also enforced at the H2 database level, providing an extra layer of data integrity.
   For example, when creating a **Person**, the system validates the format of the email and phone, as well as the presence of a name. Optional attributes (like **Address.apartment**) are stored as **null** in the database if not provided, and mapped to Java optional types when exposed via the API.

Optional Attributes

Optional attributes, such as the **apartment** field in **Address**, are represented as nullable fields within both the Java class and H2 database schema. No placeholder or default value is used; absent data are simply **null**.
When providing these values through the API or UI, nulls appear as empty form fields or missing elements, as appropriate.

Multi-value Attributes

Multi-value attributes—such as a **Shipment**'s list of carriers—are implemented as **@ElementCollection** fields or as separate join tables in JPA, supporting all basic collection types. Since there is no order preservation or uniqueness requirement, a **List<String>** suffices. These are directly mapped in the H2 schema as dependent entities or collection tables, enabling straightforward persistence and retrieval via standard bidirectional associations.

Derived Attributes

Derived attributes, such as **/totalPrice** in **Cart** and **Order**, are **not** stored in the database. Instead, they are calculated dynamically at runtime each time they're requested (e.g., with a **getTotalPrice()** method or as a JPA **@Transient** property). This ensures consistency, as changes to cart or order items are always reflected in the computed total. The database always stores the source values (unit price, quantity, discount, etc.) from which these totals are derived.

Associations

General Association Implementation

1. **Bidirectional Associations:** All navigable associations (e.g., between **Order** and **OrderItem**, **Person** and roles, **SupplierRole** and **AutoPart**) are bidirectional and managed both in Java code and in the database schema via foreign keys.
2. **Add/Remove Methods:** When objects are linked, methods check for existing associations to prevent duplicates, while also updating the reverse reference to maintain consistency (e.g., when adding a **CartItem** to a **Cart**, the **Cart** reference in **CartItem** is updated simultaneously).
3. **Change/Modify Methods:** When associations are updated, previous connections are removed, and new ones are established to prevent data orphaning and ensure consistency.

Reverse Connection

Bidirectional navigation is strictly enforced. For example, an **OrderItem** knows its parent **Order**, and the **Order** tracks all its **OrderItems**. Similarly, categories maintain a reference to their parent and hold collections of children.

Composition

Composition is realized with **strong ownership**; contained objects—such as **CartItem** within **Cart**, or **OrderItem** within **Order**—are strictly life-cycle dependent. In JPA, this is represented by cascade settings (**CascadeType.ALL**) and orphan removal. Deleting a parent (e.g., **Order**) will automatically remove all related **OrderItems** from the database. The relationship is further enforced in business logic; removing a cart deletes its items, preventing existence of orphaned objects.

Association Classes/Attributes

Where association-level attributes are necessary, such as **quantity** and **discount** in **OrderItem** or **CartItem**, these are modeled as full classes (“association classes”) within the schema and application. JPA mappings support these as entities with foreign keys to both owning classes, and explicit methods exist to change their properties while maintaining association integrity.

Reflexive Association

The reflexive association in **Category** supports a multi-level category tree with each category having at most **one parent** and any number of children. Parent/child relations are managed bidirectionally: updating the parent of a category updates the child list of that parent, and vice versa. Cyclic dependencies are explicitly checked for in the business logic to prevent infinite loops or invalid category structures.

Inheritance and Roles

Dynamic, overlapping inheritance for roles is supported using **@ManyToMany** or **@OneToMany** associations in JPA. A **Person** can possess multiple roles—customer, supplier, employee—simultaneously, with the role entities referencing back to their owning **Person**. The system uses single-table or joined-table inheritance for these roles, depending on optimization and query needs.

Association Validation

All add, remove, and update operations on associations perform existence checks, reject duplicates, and enforce referential integrity with exceptions and transaction management. Attempting to add duplicate items or create circular references throws descriptive exceptions, caught and handled in the application layer.

Summary

Overall, the design choices of the AutoPart Shop platform enforce strict data consistency, robust bidirectional navigation, and accurate life-cycle control via Java Spring and H2 database. The analytical model translates directly into code and schema, ensuring the flexibility and integrity required for a real-world e-commerce solution centered on auto parts.

Dynamic Analysis

Dynamic analysis of the use case **"A customer wants to view their past orders and track shipment status for a specific order"** has led to the identification of new requirements not present in the initial class diagram. To fully support the tracking and history of order progression and cancellation handling, as well as the shipment process, several new design elements have been introduced.

Effects and Implications on the Design Class Diagram

1. New Attributes

- **Order.orderStatus:**
  A new attribute has been introduced to the **Order** class to systematically represent the state of an order as it moves through its lifecycle.
  Type: OrderStatus (enumeration)

2. New Enumerations

- **OrderStatus (NEW):**
  An enumeration introduced to represent all possible order states, including the newly identified **Cancelled** status necessary for both UI and business logic.
  Values: Created, Processing, Shipped, Delivered, Returned, Refunded, Finished, Cancelled
- **ShipmentStatus (NEW):**
  Similar dynamic analysis for shipment revealed the need for a **ShipmentStatus** enum.
  Values: Pending, Processing, Shipped, InTransit, OutForDelivery, Delivered, Cancelled

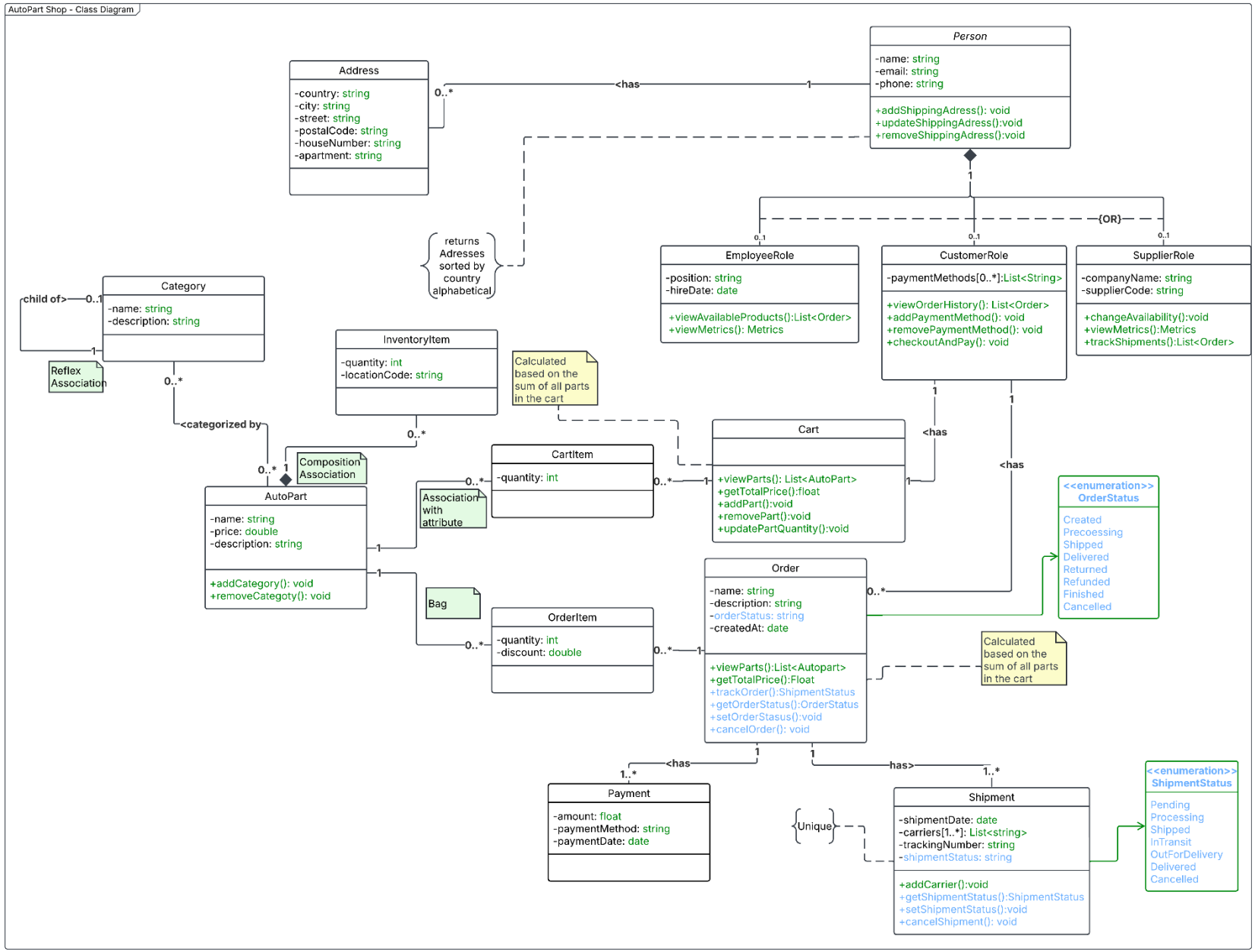
3. New Methods

- **Order class:**
  - **getOrderStatus()**: Returns the current status of the order (using the new enum).
  - **setOrderStatus(OrderStatus)**: Sets or updates the order status (needed for state transitions).
  - **trackOrder(): ShipmentStatus**: Interfaces with shipment tracking process.
  - **cancelOrder(): void**: Sets status to Cancelled and triggers related flows.
- **Shipment class:**
  - **getShipmentStatus()**: Returns the current shipment status.
  - **setShipmentStatus(ShipmentStatus)**: Updates shipment status as it moves through processing.
  - **cancelShipment(): void**: Sets shipment status to Cancelled.**Introduction of the enums** creates an explicit relationship within the class design.

4. Consequences and Discussion

These newly introduced elements facilitate accurate state management, supporting all flows in the activity and state diagrams. The addition of the **Cancelled** state to both the **OrderStatus** and **ShipmentStatus** enums enables the system to depict and handle interruptions or unsuccessful orders/shipments—a crucial requirement surfaced during dynamic modeling

Design Class Diagram - Final Version



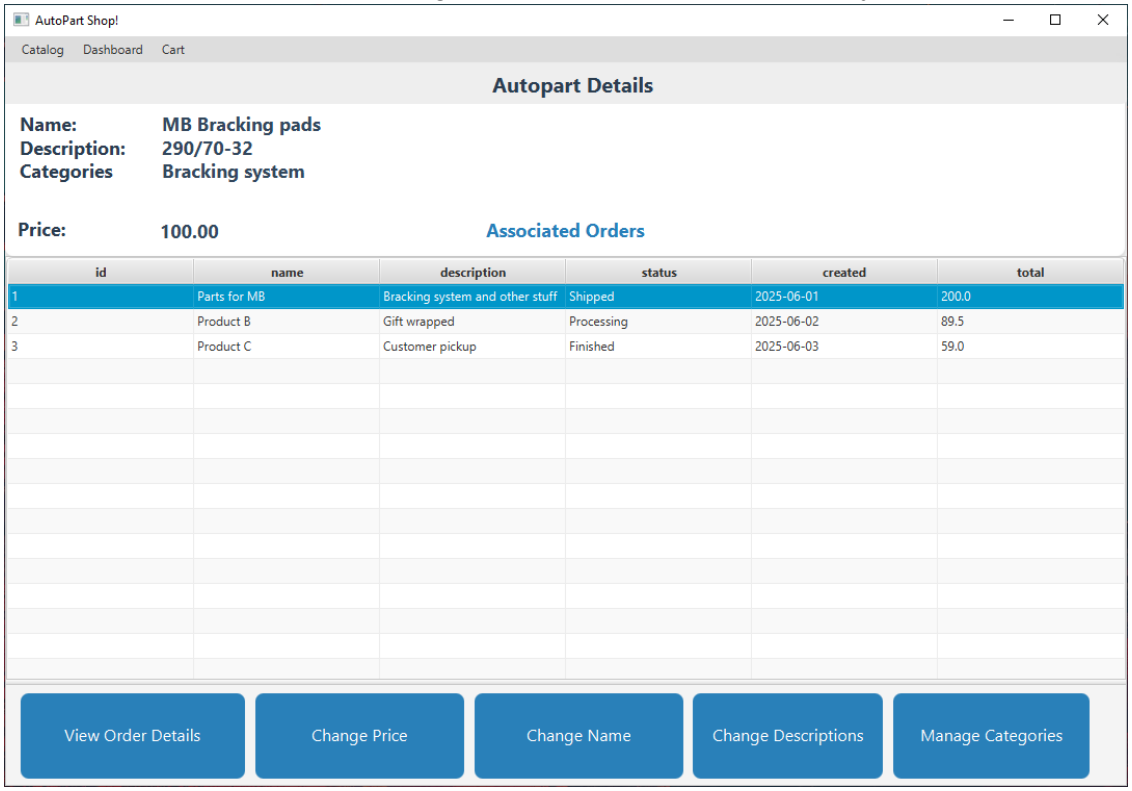
GUI Design

The GUI for the AutoPart Shop is designed with clarity and simplicity in mind, utilizing JavaFX layouts crafted in SceneBuilder. The interface prioritizes an intuitive user journey and real-time feedback, following conventional e-commerce shopping flows but tailored to the specifics of auto part selection and buying.

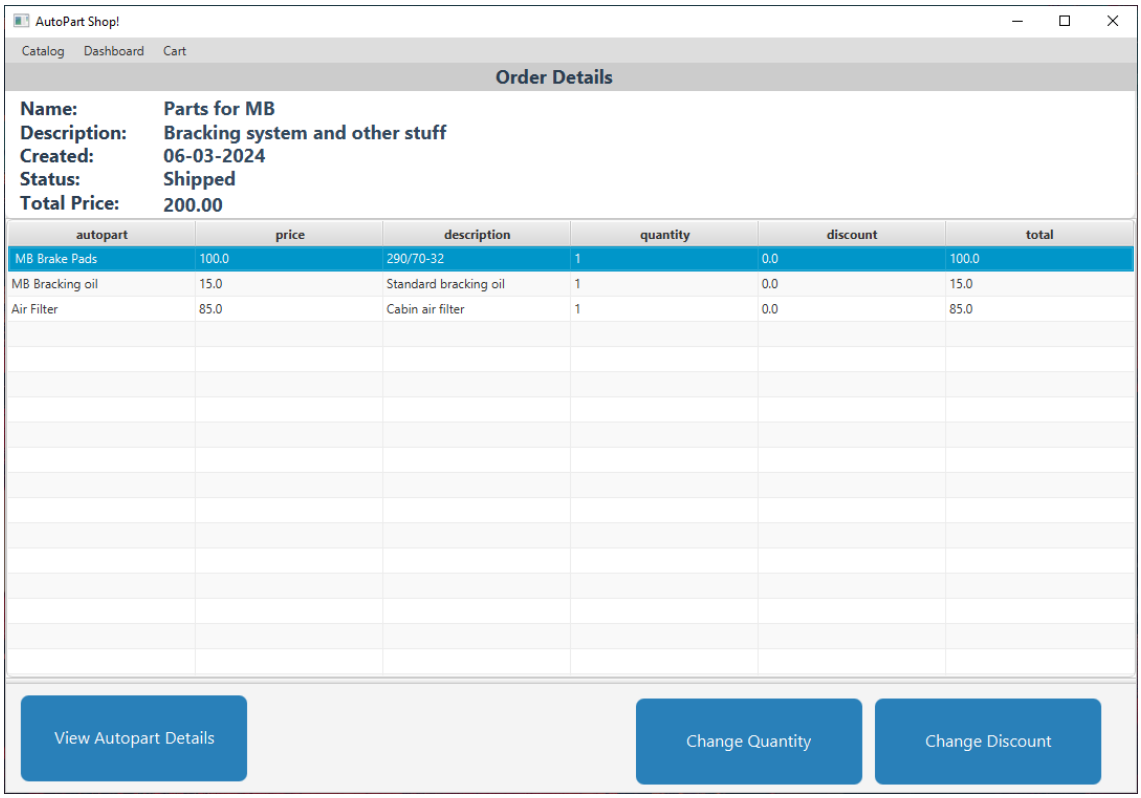
The main catalog page serves as the central workspace for customers. It displays a scrollable table of available autoparts, each row providing core product information and an easy-to-access "Add to Cart" button. All fonts, buttons, and table cells are styled for readability and consistency. Dynamic elements—such as pop-up windows are implemented to enhance the user experience and communicate system actions clearly.

Change Discount of Order item

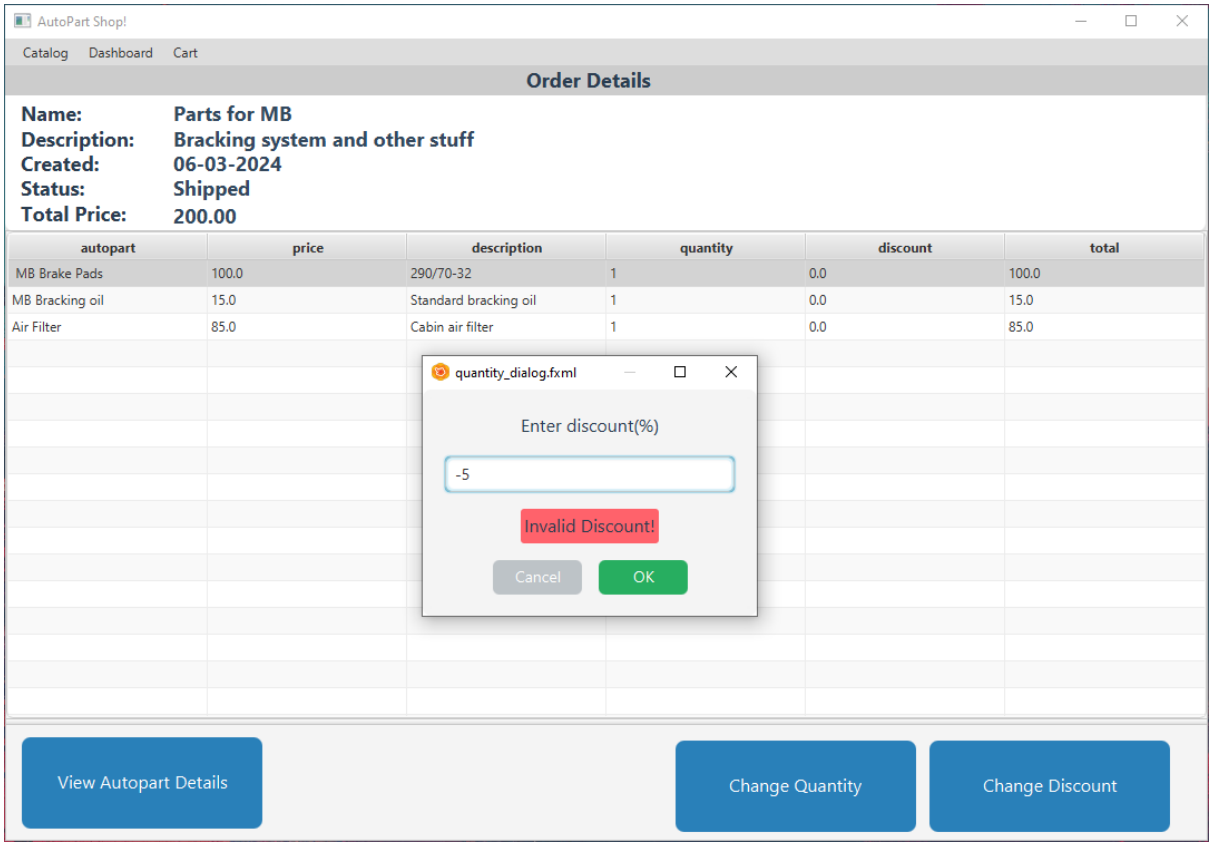
1.The window displays the details of a selected autopart. At the top, the part's name, description, category, and price are shown in a summary section. Below, a table titled "Associated Orders" lists orders that include this autopart, with columns for id, name, description, status, created date, and total amount. The first order is selected in the table. At the bottom, there are large buttons for "View Order Details," "Change Price," "Change Name," "Change Descriptions," and "Manage Categories," allowing further actions on the selected part or its orders. A navigation menu is present at the very top.



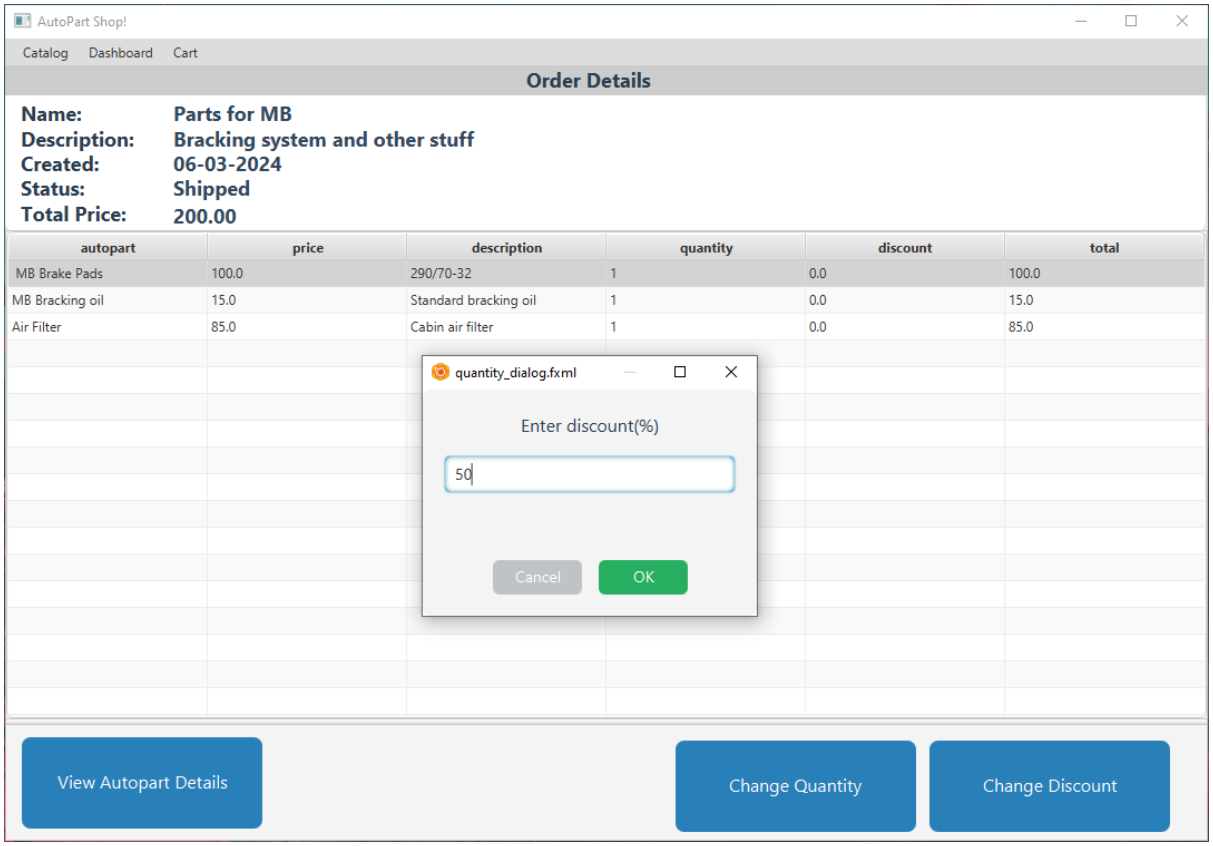
3.After selecting order and clicking “View Order Details” button, the Order Details window shows the details of a selected order. At the top, key order information is displayed, including the name, description, creation date, status, and total price, all printed in bold for emphasis. Below, a table lists all autoparts included in this order, with columns for autopart name, price, description, quantity, discount, and total. At the bottom, three large action buttons are provided: “View Autopart Details,” “Change Quantity,” and “Change Discount.” The navigation menu from earlier remains at the top of the window.



4.After the "Change Discount" button was pressed, while The first row, “MB Brake Pads” was selected, on the Order Details screen, a pop-up dialog appeared prompting the user to enter a discount percentage. The user provided an invalid value ("-5"). As a result, the dialog now displays a red "Invalid Discount!" error message below the input field, indicating that the entered discount is not acceptable. The user must enter a valid discount before the change can be applied or cancel the operation. The underlying Order Details screen remains unchanged in the background.



5. The user is currently entering the value "50" into the input field. The dialog provides two buttons: "Cancel" to abort the operation and "OK" to confirm and apply the entered discount. The underlying Order Details screen remains visible and unchanged in the background.



6.After successfully entering a valid discount percentage and confirming in the pop-up dialog, the dialog closed and the Order Details screen was updated. The discount value for "MB Brake Pads" is now shown as 50.0, and the total for this line item has been recalculated. The total price for the order is also updated accordingly. A green notification at the top right confirms that the discount was successfully changed for 5 seconds. The rest of the screen remains the same, allowing the user to continue with other actions.