E-Shop System Documentation

Steven Londoño

Septiembre 2024

1 User Histories

The user stories provide insight into the functionalities that the e-shop system aims to deliver. These stories are embedded within the code as comments, indicating the requirements from a customer's perspective. The following are the key user stories derived from the code:

- **As a customer, I want to add an item to my cart:** This functionality is implemented in the `ShoppingCart` class, specifically within the `addDevice` method. This allows users to add products to their shopping cart.
- **As a customer, I want to remove an item from my cart without having to remove all the items:** This requirement is addressed by the `removeDevice` method in the `ShoppingCart` class, allowing users to remove specific items from their cart.
- **As a customer, I want to see the items in my cart and the total price:** The `displayCart` and `calculateTotalPrice` methods in the `ShoppingCart` class provide the functionality to view the items in the cart and calculate the total price of the items.
- **As a customer, I want to verify that my information is correct:** This requirement is handled by the `verifyInfo` method in the `Checkout` class, allowing users to review and update their personal information.
- **As a customer, I want to change my information when I want:** The `verifyInfo` method in the `Checkout` class also facilitates the updating of user information as needed.
- **As a customer, I want to provide my information for purchase:** This is implemented in the `provideInfo` method in the `Checkout` class, which prompts users to enter their personal information.
- **As a customer, I want to know if I have already provided my information:** The `alreadyProvided` method in the `Checkout` class checks if the user has already provided their information.

- **As a customer, I want to view the details of a product before adding it to my cart:** This functionality is addressed by the `displayProduct` method in the `Products` class. Subclasses of `Products` must implement this method to display product-specific details, such as name, brand, and price.
- **As a customer, I want to view the details of a product before adding it to my cart:** This functionality is addressed by the `displayProduct` method in the `Products` class. Subclasses of `Products` must implement this method to display product-specific details, such as name, brand, and price.
- **As a customer, I want to proceed with the checkout process once I have reviewed my cart:** This functionality is managed by the `Checkout` class. The `provideInfo` method collects user details for the purchase, and the `displayInfo` method allows users to review their provided information before finalizing the checkout.
- **As a customer, I want the option to cancel my order if I decide not to proceed:** While the current implementation does not explicitly include a cancellation method, the user can effectively cancel by choosing not to proceed with the checkout process. This decision stops the checkout flow and does not finalize the order.
- **As a customer, I want to set and review my payment information before completing a purchase:** This requirement is fulfilled by the `provideInfo` and `displayInfo` methods in the `Checkout` class. These methods allow customers to input and review their payment and shipping information before confirming the purchase.

2 Object-Oriented Principles Analysis

The e-shop system is designed with a focus on object-oriented principles, particularly encapsulation, inheritance, and abstraction. Here is an analysis of how these principles are applied:

2.1 Encapsulation

Encapsulation is achieved by using private fields and providing public getter and setter methods. For instance:

- In the `Products` class, fields such as `product_name`, `product_price`, and `product_brand` are private, and access is controlled through getter and setter methods.
- The `ShoppingCart` and `Checkout` classes also encapsulate their data by using private fields and providing public methods to interact with these fields.

2.2 Inheritance

Inheritance is utilized to create specific types of products from a general product class:

- The `Products` class is abstract and serves as a base class for other specific product classes (e.g., `Laptop`, `Accessory`, `GameConsole`).
- This approach allows for the creation of specialized product types that inherit common attributes and methods from the `Products` class, while also defining their unique features.

2.3 Abstraction

Abstraction is demonstrated by defining an abstract class and providing specific implementations for abstract methods:

- The `Products` class is abstract and includes an abstract method `displayProduct()`. This method must be implemented by all subclasses, providing a way to display product details specific to each type of product.
- This abstraction allows the system to interact with products in a generalized way while letting specific product types provide their detailed implementations.

2.4 Polymorphism

Polymorphism is a core principle that allows objects to be treated as instances of their parent class rather than their actual class. This is achieved through method overriding and interface implementation:

- In the e-shop system, polymorphism is illustrated through the `displayProduct()` method in the `Products` class. Subclasses of `Products` (e.g., `Laptop`, `Accessory`, `GameConsole`) provide their own implementation of this method.
- When a `Products` reference is used to call the `displayProduct()` method, the specific implementation of this method in the subclass is executed. This allows the system to handle different types of products uniformly, while still invoking the correct method implementation based on the actual object type.
- For example, a `Laptop` object and a `GameConsole` object can both be referred to by a `Products` variable. When `displayProduct()` is called on this variable, the output will vary depending on whether the object is a `Laptop` or a `GameConsole`.

3 Class Diagram (UML)

The following image represent the relationship between the classes:

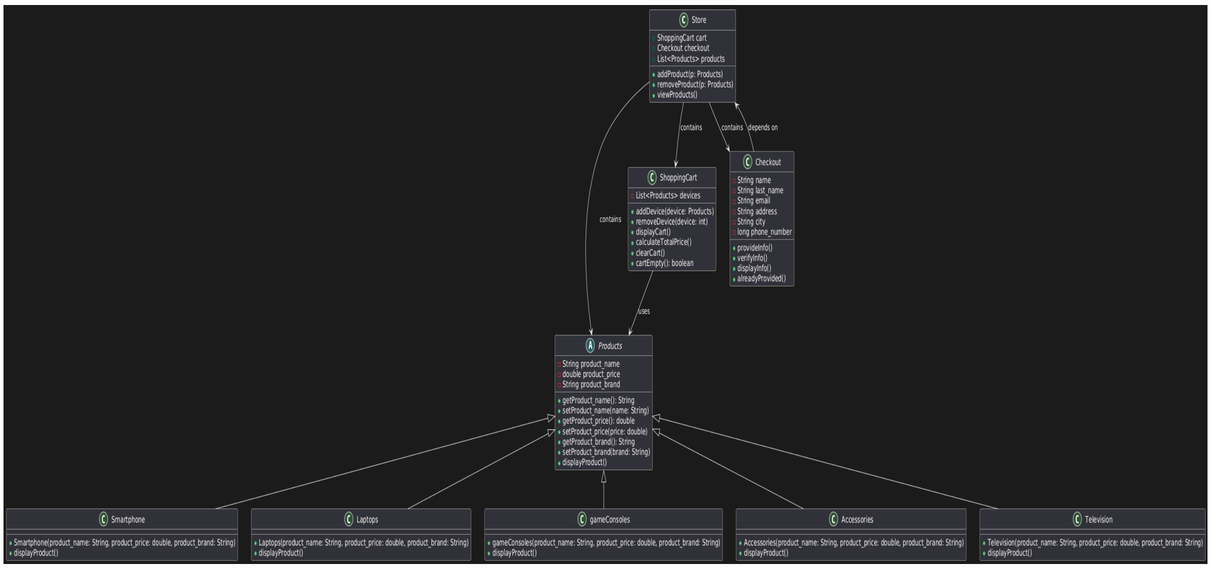
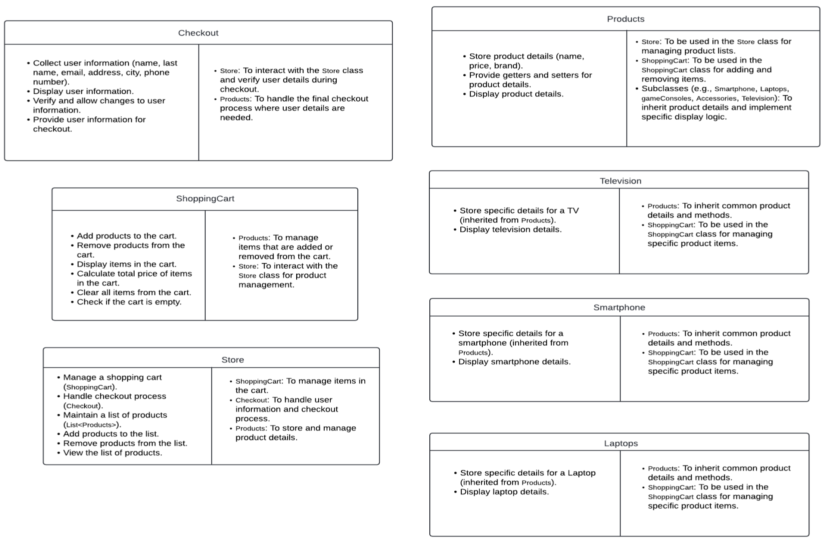

Store
ShoppingCart cart
Checkout checkout
List<Products> products
addProduct(p: Products)
removeProduct(p: Products)
viewProducts()
contains
contains
depends on
contains
ShoppingCart
List<Products> devices
addDevice(device: Products)
removeDevice(device: int)
displayCart()
calculateTotalPrice()
clearCart()
cartEmpty(): boolean
Checkout
String name
String last_name
String email
String address
String city
long phone_number
provideInfo()
verifyInfo()
displayInfo()
alreadyProvided()
uses
Products
String product_name
double product_price
String product_brand
getProduct_name(): String
setProduct_name(name: String)
getProduct_price(): double
setProduct_price(price: double)
getProduct_brand(): String
setProduct_brand(brand: String)
displayProduct()
Smartphone
Smartphone(product_name: String, product_price: double, product_brand: String)
displayProduct()
Laptops
Laptops(product_name: String, product_price: double, product_brand: String)
displayProduct()
gameConsoles
gameConsoles(product_name: String, product_price: double, product_brand: String)
displayProduct()
Accessories
Accessories(product_name: String, product_price: double, product_brand: String)
displayProduct()
Television
Television(product_name: String, product_price: double, product_brand: String)
displayProduct()

Figure 1: Classes relation.

4 CRC Diagram

The following image represent class responsibility for each class:



Checkout	
• Collect user information (name, last name, email, address, city, phone number). • Display user information. • Verify and allow changes to user information. • Provide user information for checkout.	• Store: To interact with the Store class and verify user details during checkout. • Products: To handle the final checkout process where user details are needed.

ShoppingCart	
• Add products to the cart. • Remove products from the cart. • Display items in the cart. • Calculate total price of items in the cart. • Clear all items from the cart. • Check if the cart is empty.	• Products: To manage items that are added or removed from the cart. • Store: To interact with the Store class for product management.

Store	
• Manage a shopping cart (ShoppingCart). • Handle checkout process (Checkout). • Maintain a list of products (List<Products>). • Add products to the list. • Remove products from the list. • View the list of products.	• ShoppingCart: To manage items in the cart. • Checkout: To handle user information and checkout process. • Products: To store and manage product details.

Products	
• Store product details (name, price, brand). • Provide getters and setters for product details. • Display product details.	• Store: To be used in the Store class for managing product lists. • ShoppingCart: To be used in the ShoppingCart class for adding and removing items. • Subclasses (e.g., Smartphone, Laptops, gameConsoles, Accessories, Television): To inherit product details and implement specific display logic.

Television	
• Store specific details for a TV (inherited from Products). • Display television details.	• Products: To inherit common product details and methods. • ShoppingCart: To be used in the ShoppingCart class for managing specific product items.

Smartphone	
• Store specific details for a smartphone (inherited from Products). • Display smartphone details.	• Products: To inherit common product details and methods. • ShoppingCart: To be used in the ShoppingCart class for managing specific product items.

Laptops	
• Store specific details for a Laptop (inherited from Products). • Display laptop details.	• Products: To inherit common product details and methods. • ShoppingCart: To be used in the ShoppingCart class for managing specific product items.

Figure 2: Classes responsibility.

5 Activity Diagram

The following image represent the activity diagram for the process:

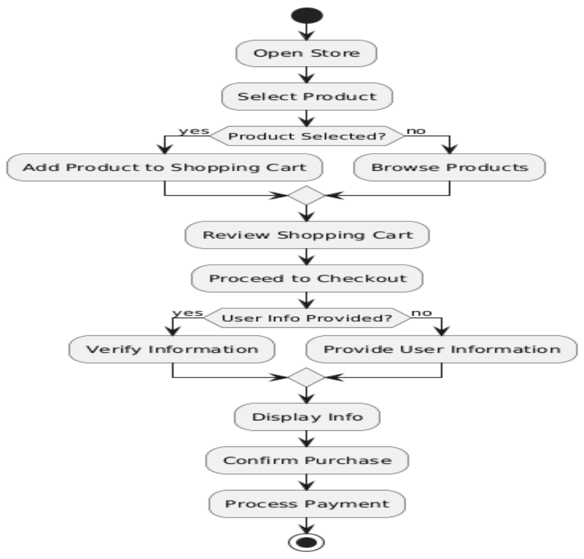


Figure 3: Process.

6 URL to the GitHub repository

https://github.com/Steven-lh/Workshops-Advanced-Programming-2024-2