Building Intelligent Systems: Object-Oriented "Pro"gramming Using AI-Powered Coding

IIT Ropar - Minor in AI

6th Feb, 2025

The Supermarket Challenge - Why Object-Oriented Programming?

Imagine Rajesh, the owner of a busy supermarket. He's facing a problem: too many salespeople tripping over each other, long queues at the checkout, and constant worries about keeping track of the stock. He dreams of a supermarket where inventory and billing are automated. He wants to build a system that accurately tracks which items are in stock and how many items are there, and also automatically calculate how much a customer owes.

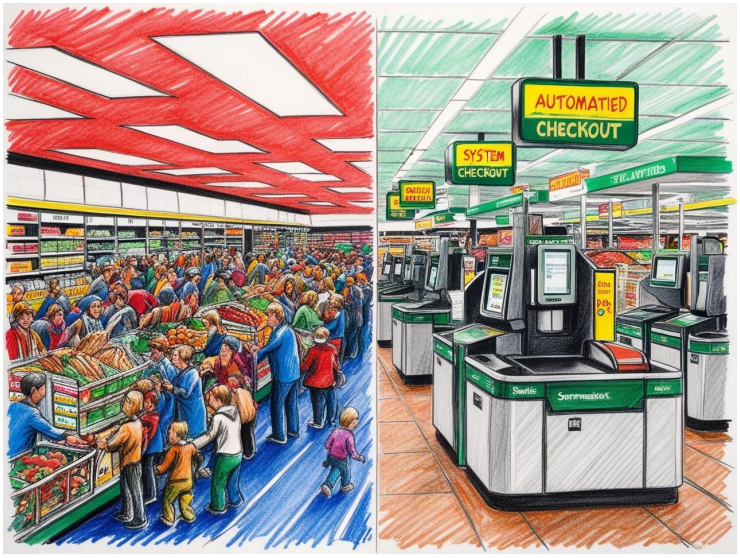

Figure 1: A crowded supermarket scene vs. a streamlined, automated checkout system!

How can Rajesh achieve this dream?

Initially, Rajesh might think of writing a set of step-by-step instructions for the computer to follow. This is called **procedural programming**. It's

like giving the computer a recipe: do this, then do that, then do this other thing. While this can work for simple tasks, it can quickly become complicated and difficult to manage as the supermarket grows and adds more products or introduces new features like membership cards and discounts.

To understand how procedural programming works, let's assume the user has bought a notebook, pen and an eraser with their corresponding prices as 150, 20 and 5 Rs, and quantities as 100, 200 and 300.

```
# Product details in inventory
product_names = ['notebook', 'pen', 'eraser']
product_price = [150, 20, 5]
product_qty = [100, 200, 300]

# Function to display all the products
def display_products():
    print("Available Products:")
    for i in range(len(product_names)):
        print(f"{i+1}. {product_names[i]} - Rs.{product_price[i]} -
     Quantity: {product_qty[i]}")

display_products()
```

```
Available Products:
1. notebook - Rs.150 - Quantity: 100
2. pen - Rs.20 - Quantity: 200
3. eraser - Rs.5 - Quantity: 300
```

Figure 2: Output!

That is where **object-oriented programming (OOP)** comes to the rescue. OOP is a way of organizing our code to represent things in the real world as **objects**. Think of each product in the supermarket as an object, each with its own characteristics (name, price, quantity) and actions it can perform (display its details, update its stock level).

This module will guide you through transitioning from procedural programming to OOP, using Rajesh's supermarket as a real-world example and leveraging the power of AI-powered coding assistants to help you along the way. By the end, you'll be equipped to build intelligent systems that are easier to understand, maintain, and extend.

Introduction to Object-Oriented Programming

Object-oriented programming is a programming paradigm centered around "objects." An object is a self-contained entity that combines data and code to manipulate that data. Let's break down the key concepts:

- **Class:** Think of a class as a blueprint or a template for creating objects. In Rajesh's supermarket, the "Product" class defines what a product is (its attributes) and what it can do (its behaviors or methods).

- **Object:** An object is an instance of a class. So, a "notebook" or "pen" would be objects created from the "Product" class.
- **Attributes:** Attributes are the characteristics or properties of an object. For a "Product" object, the attributes might be its name, price, and the quantity in stock.
- **Methods:** Methods are actions or behaviors that an object can perform. For a "Product" object, methods might include displaying its details or updating its stock level.

Building the "Product" Class

Let's create a "Product" class in Python to represent the products in Rajesh's supermarket.

```
class Product:
    def __init__(self, name, price, quantity):
        self.name = name
        self.price = price
        self.quantity = quantity

    def display(self):
        print(f"{self.name} - Rs.{self.price} (Stock: {self.
    quantity})")

    def update_stock(self, qty):
        if self.quantity >= qty:
            self.quantity -= qty
            return self.price * qty
        else:
            print(f"Not enough stock for {self.name}. Available: {
    self.quantity}")
            return 0
```

- `class Product:` This line defines a new class named "Product."
- `def __init__(self, name, price, quantity):` This is the constructor method. It's called automatically when you create a new object of the "Product" class.
  - `self`: `self` always represents the instance (the object itself) that is calling the method.
  - `name`, `price`, `quantity`: These are the parameters used to initialize the object's attributes.
  - `self.name = name`, `self.price = price`, `self.quantity = quantity`: These lines assign the values passed as arguments to the object's attributes.
- `def display(self):` This is a method to display the product's details. It prints the name, price, and stock quantity.

- `def update_stock(self, qty):` This is a method to update the stock quantity after a customer makes a purchase.
  - It checks if there is enough stock before reducing the quantity.
  - If enough stock is available, it reduces the quantity and returns the total price of the purchased items.
  - If not enough stock is available, it prints a message and returns 0.

Creating Product Objects

Now that we have the "Product" class, we can create objects (instances) of that class.

```
# Create a list of Product instances
products = [
    Product("Laptop", 50000, 10),
    Product("Phone", 20000, 5),
    Product("Headphones", 3000, 15)
]
```

This code creates three "Product" objects: a "Laptop," a "Phone," and "Headphones." Each object has its own set of attributes (name, price, quantity) with different values. This is the beauty of OOP.

Calculating the Bill

Now let's create a function to calculate the bill for a customer.

```
def calculate_bill(products, cart):
    total_bill = 0
    print("\nProcessing Order...\n")

    for product in products:
        if product.name in cart:
            qty = cart[product.name]
            cost = product.update_stock(qty)
            if cost > 0:
                print(f"{qty} x {product.name} added to bill: Rs.{cost}")
            total_bill += cost

    print("\nOrder Processed Successfully!")
    return total_bill
```

This code iterates through a customer's cart (a dictionary where keys are product names and values are the quantities to be purchased), updates the stock levels using the product object's `update_stock` method and calculates the total bill.

Streamlining User Input

To interact with the system, we need to gather the products a customer wants to buy:

```python
# Get input for the cart from user
cart = {}
while True:
    product_name = input("Enter the product name (or 'done' to
    finish): ")
    if product_name.lower() == 'done':
        break
    quantity = int(input("Enter the quantity: "))
    cart[product_name] = quantity
```

This code allows the user to enter product names and quantities, adding them to a dictionary called "cart".

Putting It All Together

Let's put all the code together and run the supermarket billing system.

```python
# Display available products before purchase
print("Available Products:")
for product in products:
    product.display()

# Calculate the bill based on the cart
total = calculate_bill(products, cart)
print(f"\nTotal Bill Amount: Rs.{total}")

# Display product stock after the purchase
print("\nStock After Purchase:")
for product in products:
    product.display()
```

```
Available Products:
Laptop - Rs.50000 (Stock: 10)
Phone - Rs.20000 (Stock: 5)
Headphones - Rs.3000 (Stock: 15)
Enter the product name (or 'done' to finish): Laptop
Enter the quantity: 2
Enter the product name (or 'done' to finish): done

Processing Order...

2 x Laptop added to bill: Rs.100000

Order Processed Successfully!

Total Bill Amount: Rs.100000

Stock After Purchase:
Laptop - Rs.50000 (Stock: 8)
Phone - Rs.20000 (Stock: 5)
Headphones - Rs.3000 (Stock: 15)
```

Figure 3: Final output!

This code displays the available products, prompts the user to enter their desired products and quantities, calculates the total bill amount, and then displays the remaining stock levels of each product after the purchase.

AI Powered Coding

As we've seen, transitioning to object-oriented programming offers significant advantages for building complex systems like Rajesh's supermarket. OOP allows us to organize code in a more modular and reusable way, making it easier to maintain and extend.

The process of transitioning from procedural to object-oriented programming, and AI tools like Gemini have assisted a great deal to streamline and help speed up the coding process.

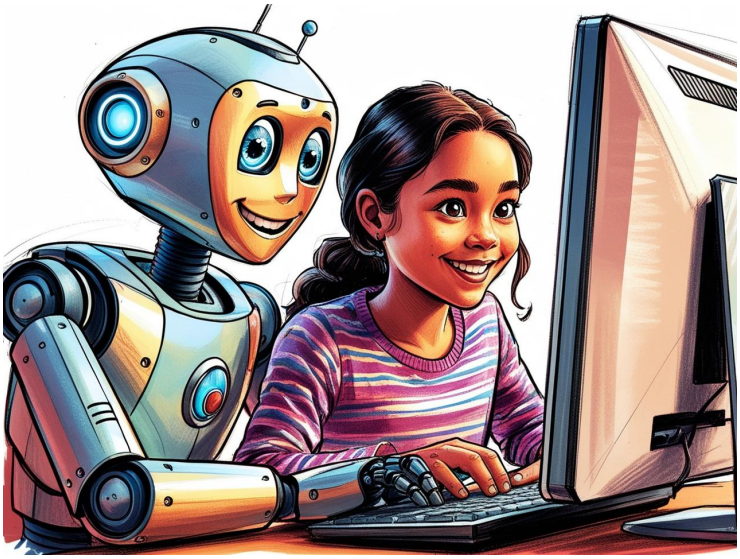

Figure 4: AI tools assisting in code generation and optimization.