Design Pattern
Factory Pattern

When designing my e-commerce system, I wanted to make sure that there is flexibility in handling all different types of products. Because the platform will support categories such as electronics, clothing, and books, I decided that the Factory Pattern would be the best choice. This pattern allows me to create objects dynamically without modifying existing code every time I add a new product type.

Why the Factory Pattern?

Using a factory method helps to maintain clean and scalable code, instead of having multiple if or switch statements all over the code, I can centralize object creation in one place.
This makes it easier to add new product categories in the future without changing the core of the system.

Use Case
Product Creation

- A customer browses the store and sees different types of products (ex Electronics, Clothes).
- The system dynamically creates the correct product object based on the category, ensuring consistency in how products are handled.

User Story

“As a customer, I want to see different types of products with their specific attributes so that I can make an informed purchasing decision.”

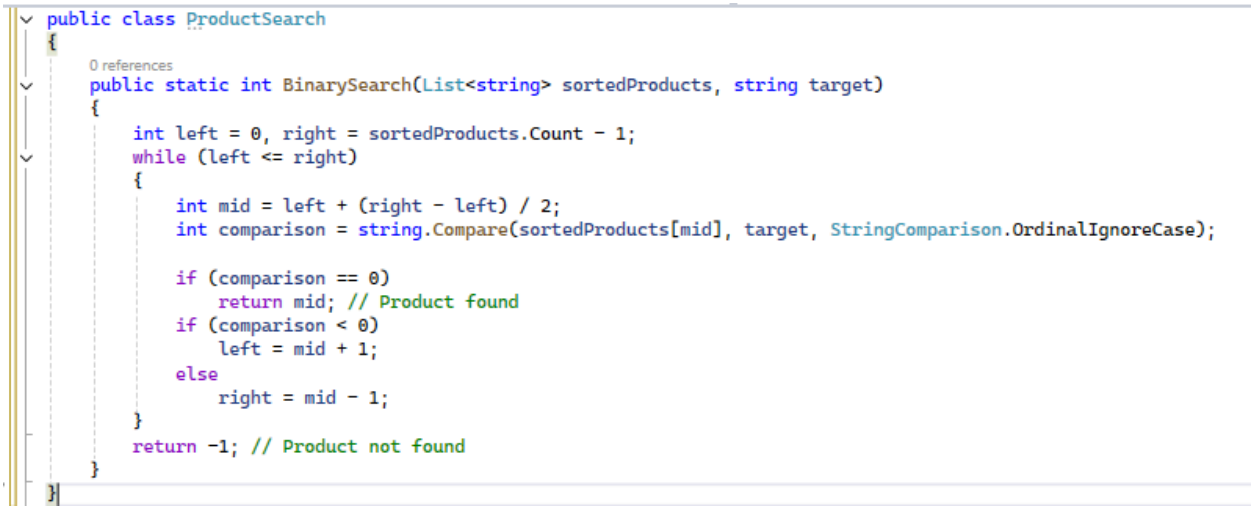
Implementation of Factory Pattern

```
// Abstract class for different product types
5 references
public abstract class Product
{
    4 references
    public string Name { get; set; }
    4 references
    public decimal Price { get; set; }
    4 references
    public abstract void DisplayInfo();
}

// Concrete product classes
1 reference
public class Electronics : Product
{
    3 references
    public override void DisplayInfo()
    {
        Console.WriteLine($"Electronics: {Name}, Price: {Price:C}");
    }
}

1 reference
public class Clothing : Product
{
    3 references
    public override void DisplayInfo()
    {
        Console.WriteLine($"Clothing: {Name}, Price: {Price:C}");
    }
}

// Factory class to create product objects
2 references
public class ProductFactory
{
    2 references
    public static Product CreateProduct(string type, string name, decimal price)
    {
        return type switch
        {
            "Electronics" => new Electronics { Name = name, Price = price },
            "Clothing" => new Clothing { Name = name, Price = price },
            _ => throw new ArgumentException("Invalid product type")
        };
    }
}
```

Algorithms used in the project

Binary Search used for product search.

Since searing is a critical function in the e-commerce system, I needed an effective way to locate products in a sorted list. Instead of using a linear search, I chose Binary Search.
This makes search operations much faster, especially when dealing with large product catalogs.



```
public class ProductSearch
{
    0 references
    public static int BinarySearch(List<string> sortedProducts, string target)
    {
        int left = 0, right = sortedProducts.Count - 1;
        while (left <= right)
        {
            int mid = left + (right - left) / 2;
            int comparison = string.Compare(sortedProducts[mid], target, StringComparison.OrdinalIgnoreCase);

            if (comparison == 0)
                return mid; // Product found
            if (comparison < 0)
                left = mid + 1;
            else
                right = mid - 1;
        }
        return -1; // Product not found
    }
}
```

Quicksort used for sorting products

Sorting product efficiently is critical, especially when users filter items by price or name. Quicksort is a highly efficient sorting algorithm, and it performs much better than simpler algorithms like Bubble Sort.

```
public class Sorting
{

    public static void QuickSort(List<Product> products, int left, int right)
    {
        if (left < right)
        {
            int pivotIndex = Partition(products, left, right);
            QuickSort(products, left, pivotIndex - 1);
            QuickSort(products, pivotIndex + 1, right);
        }
    }


    private static int Partition(List<Product> products, int left, int right)
    {
        Product pivot = products[right];
        int i = left - 1;

        for (int j = left; j < right; j++)
        {
            if (products[j].Price < pivot.Price)
            {
                i++;
                (products[i], products[j]) = (products[j], products[i]);
            }
        }

        (products[i + 1], products[right]) = (products[right], products[i + 1]);
        return i + 1;
    }
}
```

Summary

By implementing these design choices, my e-commerce system will be more scalable, efficient, and maintainable. The Factory Pattern simplifies product management, while Binary Search and Quicksort improve search and sorting operations for a better user experience.