**Implementing the Builder Pattern**

**Scenario :** To develop a system to create complex objects, such as a Computer, with multiple optional parts. The Builder Pattern is used to simplify object construction when there are many configuration possibilities.

**1.Objective :** To use the **Builder Design Pattern** to construct complex objects step-by-step, especially when many optional parameters are involved.

**2. Product Class:**

I created a class called Computer with the following attributes:

- CPU
- RAM
- Storage
- GPU (optional)
- WiFi (optional)
- Bluetooth (optional)

**3. Builder Class :** A **static nested class** called Builder I added inside the Computer class. This class provides **chained setter methods** to set each component and a final build() method to construct the actual Computer object.

**4. Implementation:**

- The Computer class has a **private constructor** that takes a Builder object as a parameter.
- This constructor assigns values from the builder to the actual object.
- This design ensures immutability and clear object construction.

**5. Testing the Builder Pattern:**

In the test class, I demonstrated creating multiple computer configurations:

- **Basic Computer** with just CPU, RAM, and Storage.
- **Gaming Computer** with high-end specs and optional GPU, WiFi, and Bluetooth.
- **Budget Computer** with minimal setup.

Each configuration was created using:

new Computer.Builder().setCPU(...).setRAM(...).build();

**6. Advantages of Builder Pattern:**

- Improves **code readability** and **maintainability**.
- Prevents constructor overloading confusion.
- Enables **step-by-step** object creation and **optional parameter** handling.

f

**Conclusion :** The Builder Pattern was successfully applied to manage complex object construction in a clean and controlled way. It proved helpful in creating multiple configurations of the same product class (Computer) while keeping the code modular and readable.