**Module1**

**Solid-Robert.C Martin 2000 and**  **Michael Feathers**

* S-srp-Single Responsibility Principle
* O-ocp-open/closed principle
* L-lsp-Liskov Substitution Principle
* I-isp-Interface Segregation Principle
* D-dip-Dependancy Inversion Principle

**design principles encourage us to create more maintainable, understandable, and flexible software.**

1)srp-**a class should only have one responsibility. Furthermore, it should only have one reason to change.**

* Easier to testing since class with 1 responsibility will have few cases
* Low coupling-less functionality in a single class will have few dependency
* Organization-small,well org are easier to search than monolithic one

**2)**ocp- **Open for extension** → You should be able to **add new features/behavior** to a class **without changing its original code. Closed for modification** → **Don’t touch old working code** unless you’re fixing a bug.

Ex if we have a exising completely working class and I need t0 add some more feature we need to create new class and extend the old exiting class property instead of simply adding code to old class

**3)**Lsp-child class need to have the behavior of parent class without breaking ex: if orstrich class inhert from bird class which have method fly but ostritch class also have fly method but it cant fly which violet lsp where as eagle class ineherit from bird class have fly method which print speed fly will have the same property of bird’s fly method

**4)**isp-**Large interface should be divided into smaller parts so that the class implementing it will only get the required methods.**

**Example:**  
If a **BearKeeper** interface has 3 methods — wash, feed, and pet the bear — but the zookeeper only wants to wash and feed, not pet,  
**it will force the zookeeper to implement pet method unnecessarily.**

This **violates ISP** because the class has to write code for something it **doesn’t actually need.**

To follow ISP, we **split** the interface into small ones:  
→ One interface for washing,  
→ One for feeding,  
→ One for petting.Now the happy**zookeeper can only implement what they actually do.**

**5)**dip-Imagine a **remote control**.

❌ **Without Dependency Inversion:**  
The remote **only works for one specific TV** because it was built just for that TV.  
If you change your TV, you need a new remote.  
➡ **The remote depends on the TV.**

✅ **With Dependency Inversion:**  
Now imagine a **universal remote**.  
It works with **any** brand of TV, AC, or other devices because **all devices follow a common standard (like infrared codes).**  
➡ **Now, the remote depends on a *common standard*, not the specific device.**  
➡ The **TV also depends on that common standard.**

**🔔 Result:**

You can use the same remote for multiple devices.

Devices can change; the remote doesn’t care — it works as long as the device follows the agreed rules (interface/standard).

**Design Patterns**

A design pattern in programming is a reusable solution to a common problem that occurs during software design and development. It provides a structured approach to solving specific design or implementation issues. create more maintainable, flexible, and scalable code.

A **design pattern** is like a **ready-made plan** for solving a **common problem** in programming.

Think of it like **recipes in cooking** 🍳:

* Want to make tea? → Use a recipe.
* Want to build flexible software? → Use a **design pattern**.

You don’t have to invent the solution from scratch — **many smart developers already figured it out**.

**3 types:**

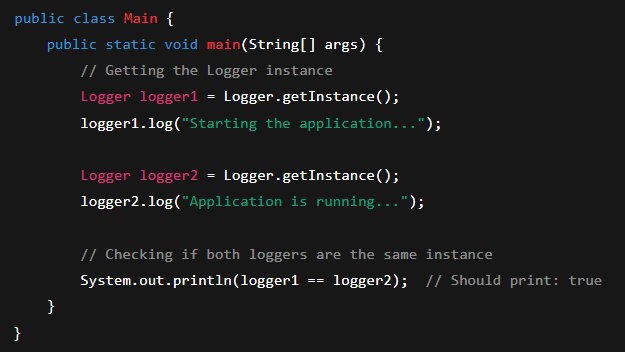
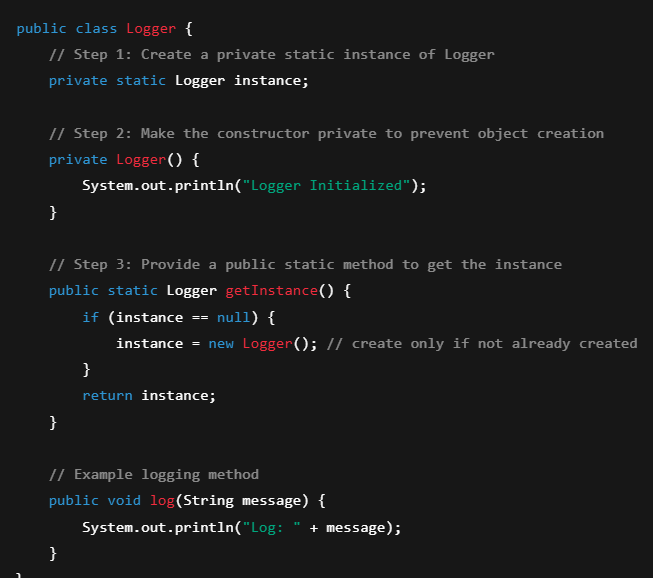
* Creational- how obj are created
* Structural- how class/obj are organized/connected
* Behavioral- how obj comm and behave

Creation pattern- These patterns focus on object creation mechanisms, providing ways to create objects in a manner that is flexible, decoupled from their concrete implementations, and suited to the specific requirements of the application. Examples include the Singleton pattern, Factory pattern, and Builder pattern.

**Singleton-**It allows only ONE object of a class to be created.

Think of **the manager of a building**.Only **one manager** controls the building.Anyone who needs help talks to **that ONE manager**.**No one creates new managers** for the same building.

Both logger1 and logger2 refer to the *same* object because Singleton ensures that only one instance of the class exists in the whole application.No matter how many students go to read the board,  
→ there’s only *one* notice board.  
→ All students look at the same board to see the latest updates.



o/p

Logger Initialized

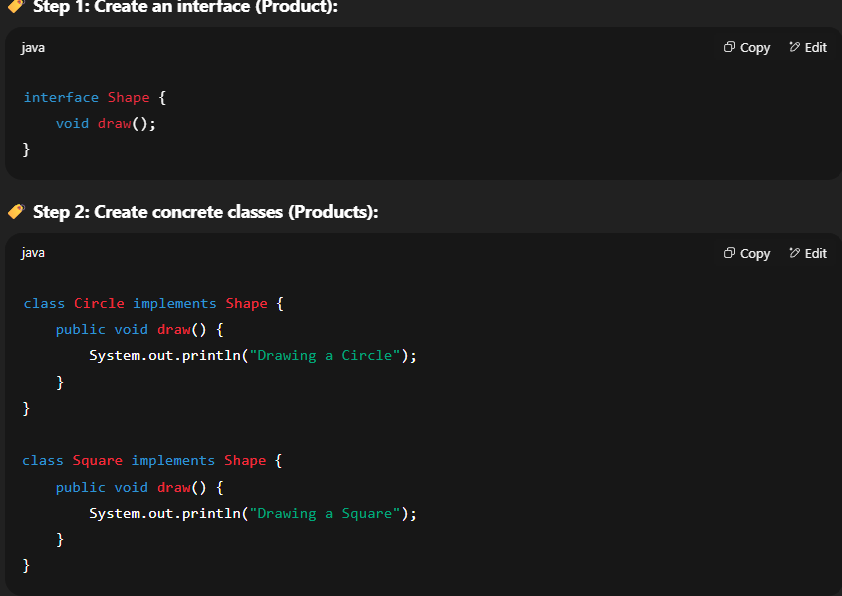
Log: Starting the application...

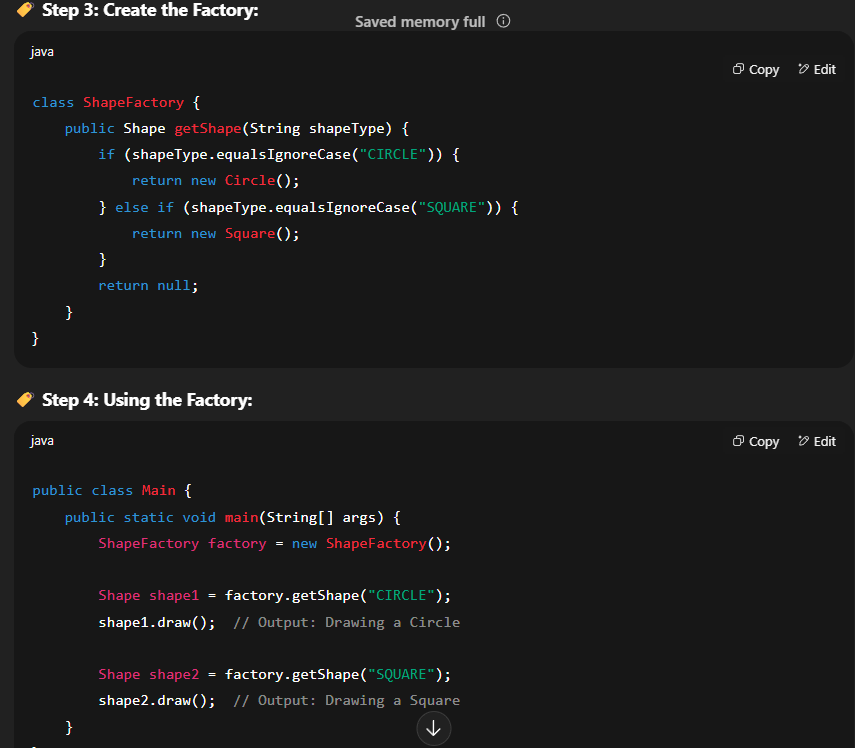
Log: Application is running...

true

**Factory-** It hides the object creation logic from you.

You don’t build shoes yourself —  
You just tell the shop: “I want sports shoes” or “I want formal shoes”, and the shop (factory) gives you the right shoes.

You don’t care how the shoes were made → only that you get them.



Structural- These patterns deal with the composition of classes and objects to form larger structures while keeping them flexible and efficient. They help ensure that classes and objects can work together effectively to achieve a common goal. Examples include the Adapter pattern, Decorator pattern, and Composite pattern.

Behavioral Patterns- These patterns focus on the interactions and communication between objects and classes. They provide solutions for effectively managing the flow of control, responsibilities, and behavior between objects. Examples include the Observer pattern, Strategy pattern, and Command pattern.

**Module-2**

**Analysis of Algorithms** is a fundamental aspect of computer science that involves evaluating performance of algorithms and programs. Efficiency is measured in terms of **time**and**space**.

**Order of Growth** in Algorithms (Big-O Notation- describes **how fast** or **slow** an algorithm is as input size (**n**) grows.)

* **Linear Search** → O(n) → Time grows **linearly** with data size.
* **Binary Search** → O(log n) → Time grows **logarithmically** → much faster for **large data**.

Order of Growth describes how the running time or space requirement of an algorithm increases as the size of the input grows.

***Example 1 :*** *4n2 + 3n + 100* ***Example 2 :*** *100 n Log n + 3n + 100 Log n + 2  
After ignoring lower order terms, we get After ignoring lower order terms, we get  
4n2 100 n Log n  
After ignoring constants, we get After ignoring constants, we get  
n2 n Log n  
Hence order of growth is n2 Hence order of growth is n Log n*

| ***Case*** | ***Linear Search*** | ***Binary Search*** |
| --- | --- | --- |
| ***Best*** | *O(1)* | *O(1)* |
| ***Average*** | *O(n/2) ≈ O(n)* | *O(log n)* |
| ***Worst*** | *O(n)* | *O(log n)* |

**Asymptotic Analysis** method used to compare which alg among 2 is better,in normal we compare it mby giving inout and comparing the time taken by both model but its not a proper one. So here we uses the order to find which is better ex- 1 model work in linear search and other in binary where starting linear will be fast after linear growth it will be slow where as in start binary will be sslow then it will become fast[linear order of growth- linear growth,binary order of growth will be lograthamic]

**Time complexity** The Time Complexity of an algorithm/code is not equal to the actual time required to execute a particular code, but the number of times a statement executes.

|  |  |
| --- | --- |
| **Access 1st element (normal return,constant value)** | **O(1)** |
| **Single loop(for,travering in arr)** | **O(n)** |
| **Nested loop** | **O(n²)** |
| **Binary search(code splited like if,else)** | **O(log n)** |

**for i in range(n): # Runs n times → O(n)**

**for j in range(n): # Runs n times → O(n)**

**print(i, j) # n \* n → O(n²)**

**Space complexity-***Auxiliary Space* is the extra space or temporary space used by an algorithm.

*The space Complexity*of an algorithm is the total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

**If variables size does not depend on the size of the input, therefore  Space Complexity will be constant or O(1)**

**Sorting**

| **Term** | **Meaning** |
| --- | --- |
| **In-place Sorting** | Sorts the data using the **same array**, using little extra space 🔸 *Example*: Selection Sort, Bubble Sort |
| **Internal Sorting** | Sorts data stored **entirely in main memory** |
| **External Sorting** | Sorts **large data stored in external memory** (disk, etc.) 🔸 *Example*: Merge Sort |
| **Stable Sorting** | Keeps the **original order** of equal items 🔸 *Example*: Merge Sort, Bubble Sort |
| **Hybrid Sorting** | Combines **two or more algorithms** for better performance 🔸 *Example*: IntroSort (Quick Sort + Insertion Sort) |

Types of Sorting Algorithms

Comparison-based  
Elements are compared to each other  
🔸 *Examples*: Bubble Sort, Quick Sort, Merge Sort

Non-comparison-based  
Elements are NOT compared directly  
🔸 *Examples*: Counting Sort, Radix Sort

1)Selection sort

Comparision based. Compare the 1st number with other and swith the smaller number places

**Finds the smallest element** in the entire remaining array → places it in correct position

**Array:** [5, 3, 8, 4, 2]

1️.Find **smallest** → 2 → swap with first → [2, 3, 8, 4, 5]  
2️.Find next smallest → 3 → already in place → [2, 3, 8, 4, 5]  
3️.Find next smallest → 4 → swap with 8 → [2, 3, 4, 8, 5]  
4️.Find next smallest → 5 → swap with 8 → [2, 3, 4, 5, 8]

2)Bubble sort

Comparision based. Repeatedly **compares adjacent elements** in the array.**Swaps them** if they are in the **wrong order.**After each pass, the **largest element "bubbles" up** to the end

**Repeatedly compares adjacent elements** and swaps if needed → largest "bubbles up" to the end

1st

[5, 3, 8, 4, 2] → compare 5 & 3 → swap → [3, 5, 8, 4, 2]

[3, 5, 8, 4, 2] → compare 5 & 8 → OK → [3, 5, 8, 4, 2]

[3, 5, 8, 4, 2] → compare 8 & 4 → swap → [3, 5, 4, 8, 2]

[3, 5, 4, 8, 2] → compare 8 & 2 → swap → [3, 5, 4, 2, 8] ✅ largest at end

2nd

[3, 5, 4, 2, 8] → compare 3 & 5 → OK

[3, 5, 4, 2, 8] → compare 5 & 4 → swap → [3, 4, 5, 2, 8]

[3, 4, 5, 2, 8] → compare 5 & 2 → swap → [3, 4, 2, 5, 8]

3)Insertion sort

**Insertion Sort** is like **arranging playing cards** in your hand. Start with one card (sorted part).Pick the next card and **insert it into the correct position**.Repeat until all cards are arranged.

**Example Array:**[5, 3, 8, 4, 2]

Start → [5] | [3, 8, 4, 2] → Insert 3 → [3, 5] | [8, 4, 2]

Next → [3, 5] | [8, 4, 2] → Insert 8 → [3, 5, 8] | [4, 2]

Next → [3, 5, 8] | [4, 2] → Insert 4 → [3, 4, 5, 8] | [2]

Next → [3, 4, 5, 8] | [2] → Insert 2 → [2, 3, 4, 5, 8]