System design

In **Software Architecture and Design Patterns (SADP)**, a **design pattern** is a **general, reusable solution** to a commonly occurring problem within a given context in software design. It's not a finished design that can be directly transformed into code, but rather a **template or blueprint** for how to solve a problem in a way that has proven effective over time.

🧩 Key Concepts of Design Patterns in SADP

- **Reusable**: Patterns can be applied to different problems in different contexts.
- **Proven**: They are based on real-world experience and best practices.
- **Language-independent**: They describe solutions in a way that can be implemented in any programming language.

🏗️ Categories of Design Patterns

Category	Purpose	Examples
Creational	Deal with object creation mechanisms	Singleton, Factory, Builder
Structural	Deal with object composition and relationships	Adapter, Composite, Proxy
Behavioral	Deal with communication between objects and responsibilities	Observer, Strategy, Command

---

📌 Why Design Patterns Matter in SADP

- Promote **modularity** and **scalability**
- Improve **code readability** and **maintainability**
- Encourage **best practices** and **standardized solutions**
- Help in **communicating design ideas** effectively among developers

Would you like a Bengali summary or examples of each pattern with real-world analogies?

🧠 Singleton Pattern – Study Note (SADP)

🔷 Pattern Type: Creational Design Pattern

বাংলা: সৃষ্টি-ভিত্তিক ডিজাইন প্যাটার্ন

🔷 Core Idea

- Ensures **only one instance** of a class exists.
- Provides a **global point of access** to that instance.
- Commonly used for **shared resources** like databases, configuration managers, or logging systems.

🔷 Real-World Analogy

> 🇧🇩 “President of a country” → একটাই সরকার বা প্রেসিডেন্ট থাকে, যাকে সবাই রেফারেন্স করে।

- Just like a country has one official government, a singleton class has one official instance.
- Used like a **global identifier** across the application.

📝 Short Note: Singleton in Dart vs Java

- **Java**
  - Uses `volatile` to ensure visibility across threads.
  - Needs `synchronized` for thread-safe lazy initialization.
  - Double-checked locking pattern:

```
private static volatile Singleton instance;
public static Singleton getInstance(String data) {
    Singleton result = instance;
```

```
    if (result == null) {
        synchronized(Singleton.class) {
            if (instance == null) {
                instance = result = new Singleton(data);
            }
        }
    }
    return result;
}
```

- **Dart**
  - No `volatile` keyword → not needed (isolates don't share memory).
  - Thread safety handled differently (via `synchronized` package if needed).
  - Idiomatic Singleton:

```
class Singleton {
  static final Singleton _instance = Singleton._internal();
  Singleton._internal();
  static Singleton get instance => _instance;
}
```

- **Key Difference**
  - Java: shared-memory threads → needs `volatile` + locks.
  - Dart: isolate model → no shared mutable state, so `volatile` is unnecessary.

👉 In Dart, just use `static final` (eager) or a `static method` with `Lock` (lazy) — both are safe and clean.

Here's a **short, practical note** you can keep as a cheat-sheet for Singleton patterns in Dart 👇

📝 Singleton in Dart — When to Use What

1. `static final` (Eager Initialization)

```
class Singleton {
  static final Singleton instance = Singleton._internal();
  Singleton._internal();
}
```

- ✅ Fastest, simplest, thread-safe by design
- ✅ Use when object is **lightweight** and safe to create at startup
- ❌ Not lazy (created even if never used)

---

2. Lazy Initialization (No async, No package)

```
class Singleton {
  static Singleton? _instance;
  Singleton._internal();
  static Singleton getInstance() {
    _instance ??= Singleton._internal();
    return _instance!;
  }
}
```

- ✅ Created only when first needed
- ✅ No external package
- ❌ Not safe if multiple async calls race to initialize at the same time

---

3. Lazy + Thread-Safe (with `synchronized`)

```
import 'package:synchronized/synchronized.dart';

class Singleton {
  static Singleton? _instance;
  static final _lock = Lock();
  Singleton._internal();
  static Future<Singleton> getInstance() async {
    if (_instance != null) return _instance!;
```

```
    await _lock.synchronized(() async {
      _instance ??= Singleton._internal();
    });
    return _instance!;
  }
}
```

- ✅ Safe for **async + multi-step initialization**
- ✅ Prevents race conditions
- ❌ Slight overhead, needs package

---

4. Async Lazy (No package, using `Completer`)

```
class AsyncSingleton {
  static AsyncSingleton? _instance;
  static Completer<AsyncSingleton>? _completer;
  AsyncSingleton._internal();
  static Future<AsyncSingleton> getInstance() async {
    if (_instance != null) return _instance!;
    if (_completer != null) return _completer!.future;
    _completer = Completer();
    await Future.delayed(Duration(milliseconds: 500)); // simulate setup
    _instance = AsyncSingleton._internal();
    _completer!.complete(_instance);
    return _instance!;
  }
}
```

- ✅ Best for **heavy async setup** (DB, file I/O, network)
- ✅ No external package
- ❌ Slightly more code

⚡ Quick Rule of Thumb

- **Use** `static final` → if simple + lightweight.
- **Use lazy init** → if you want to delay creation until first use.
- **Use** `synchronized` → if async init must be safe against race conditions.
- **Use** `Completer` → if you want async lazy init without extra packages.

🔷 Why Singleton?

- Prevents multiple instantiations of **heavy or sensitive resources**.
- Ensures **consistent access** and **state management**.
- Example: Database connection → একটাই instance সব জায়গায় ব্যবহার হয়।

🔷 Basic Implementation Steps

```
public class Singleton {
    private static volatile Singleton instance; // 🔒 Volatile for thread safety
    private String data;

    private Singleton() {
        data = "Initialized"; // 👷 Private constructor
    }

    public static Singleton getInstance() {
        Singleton localRef = instance; // 🧠 Local caching
        if (localRef == null) {
            synchronized (Singleton.class) {
                localRef = instance;
                if (localRef == null) {
                    instance = localRef = new Singleton();
                }
            }
        }
        return localRef;
```

```
    }
}
```

🔷 Key Concepts & Bengali Keywords

Concept	Meaning in Bengali
Volatile	মেমোরি থেকে সরাসরি রিড, ক্যাশ নয়
Synchronized block	একসাথে এক থ্রেডই এক্সেস করতে পারে
Double-checked locking	দুইবার চেক করে লক নেয়, পারফরম্যান্স বাড়ায়
Private constructor	বাইরের ক্লাস থেকে object তৈরি করা যাবে না
Static method (getInstance)	একটাই instance রিটার্ন করে

🔷 Threading Issues & Fixes

- ❌ Without synchronization → multiple threads may create multiple instances.
- ✅ Use `synchronized` block → ensures one thread initializes.
- ⚠️ Performance hit → every thread waits even if instance is ready.
- ✅ Use **double-checked locking** → avoids unnecessary locking.
- ⚠️ Risk of **partially constructed object** due to compiler optimization.
- ✅ Use `volatile` → ensures visibility and complete construction.

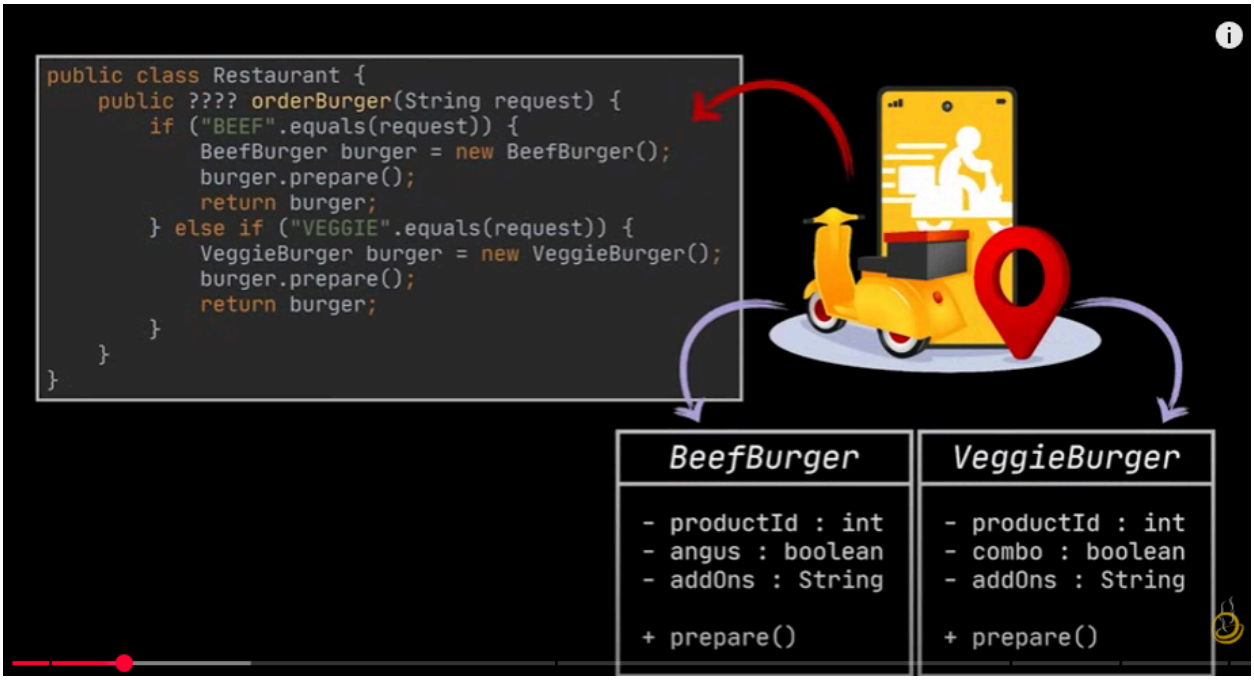
🔷 Micro-Optimization

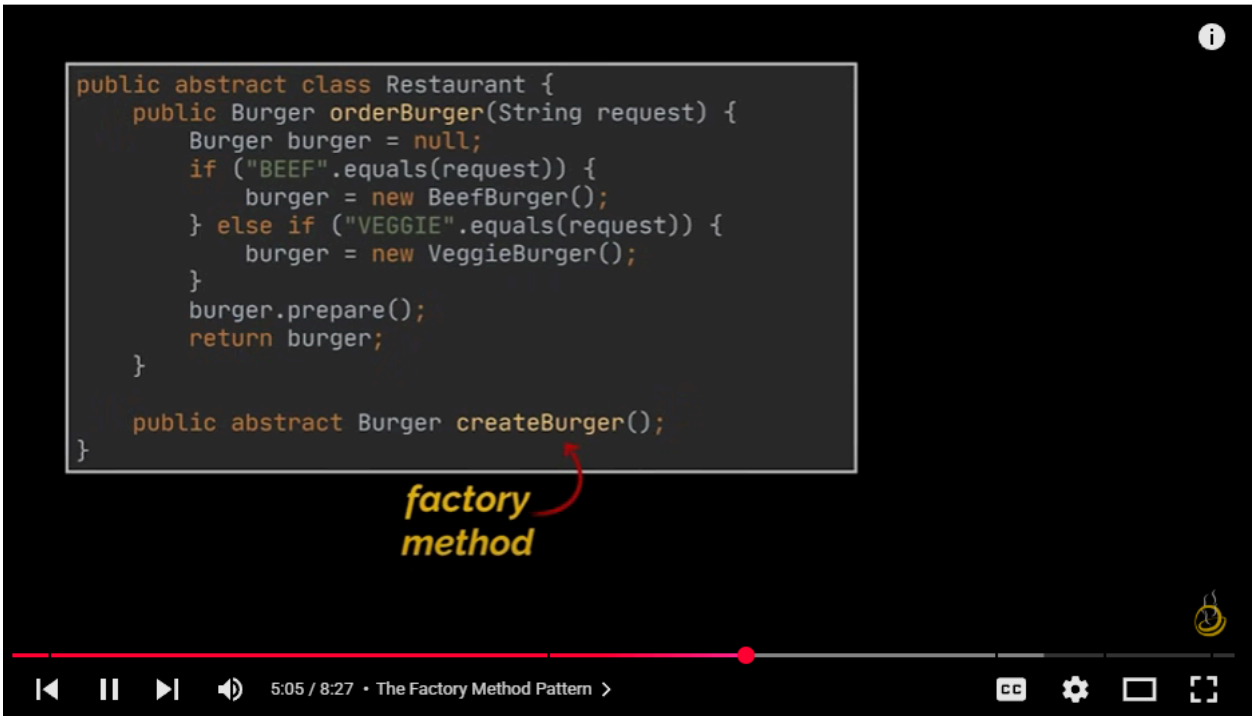
- Use **local variable caching** to avoid multiple memory reads.
- Reduces overhead by ~40% in high-load scenarios.

🔷 When to Use Singleton

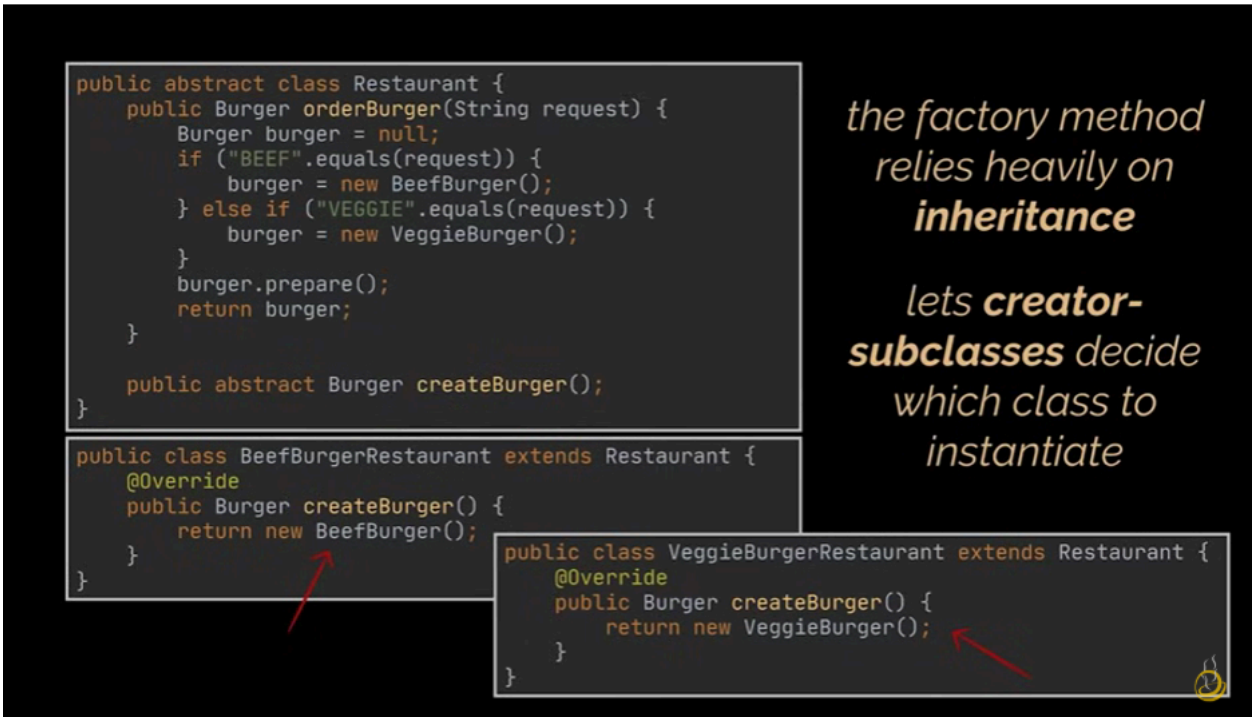
- When **only one instance** is needed across the app.
- When **global access** is required.
- When **resource sharing** is critical (e.g., DB, config).

🍔 Factory Patterns – Study Note




public abstract class Restaurant {
public Burger orderBurger(String request) {
Burger burger = null;
if ("BEEF".equals(request)) {
burger = new BeefBurger();
} else if ("VEGGIE".equals(request)) {
burger = new VeggieBurger();
}
burger.prepare();
return burger;
}
public abstract Burger createBurger();
}
factory method
5:05 / 8:27 • The Factory Method Pattern >

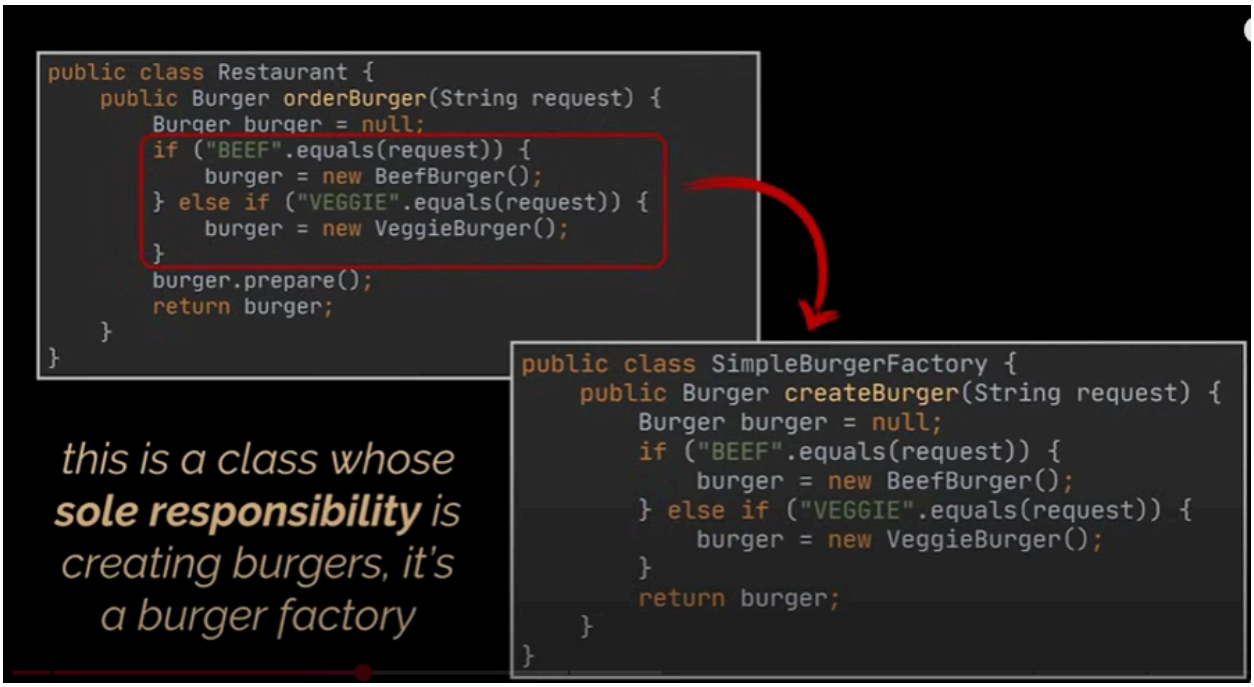

public abstract class Restaurant {
public Burger orderBurger(String request) {
Burger burger = null;
if ("BEEF".equals(request)) {
burger = new BeefBurger();
} else if ("VEGGIE".equals(request)) {
burger = new VeggieBurger();
}
burger.prepare();
return burger;
}
public abstract Burger createBurger();
}
the factory method relies heavily on inheritance
public class BeefBurgerRestaurant extends Restaurant {
@Override
public Burger createBurger() {
return new BeefBurger();
}
}
public class VeggieBurgerRestaurant extends Restaurant {
@Override
public Burger createBurger() {
return new VeggieBurger();
}
}

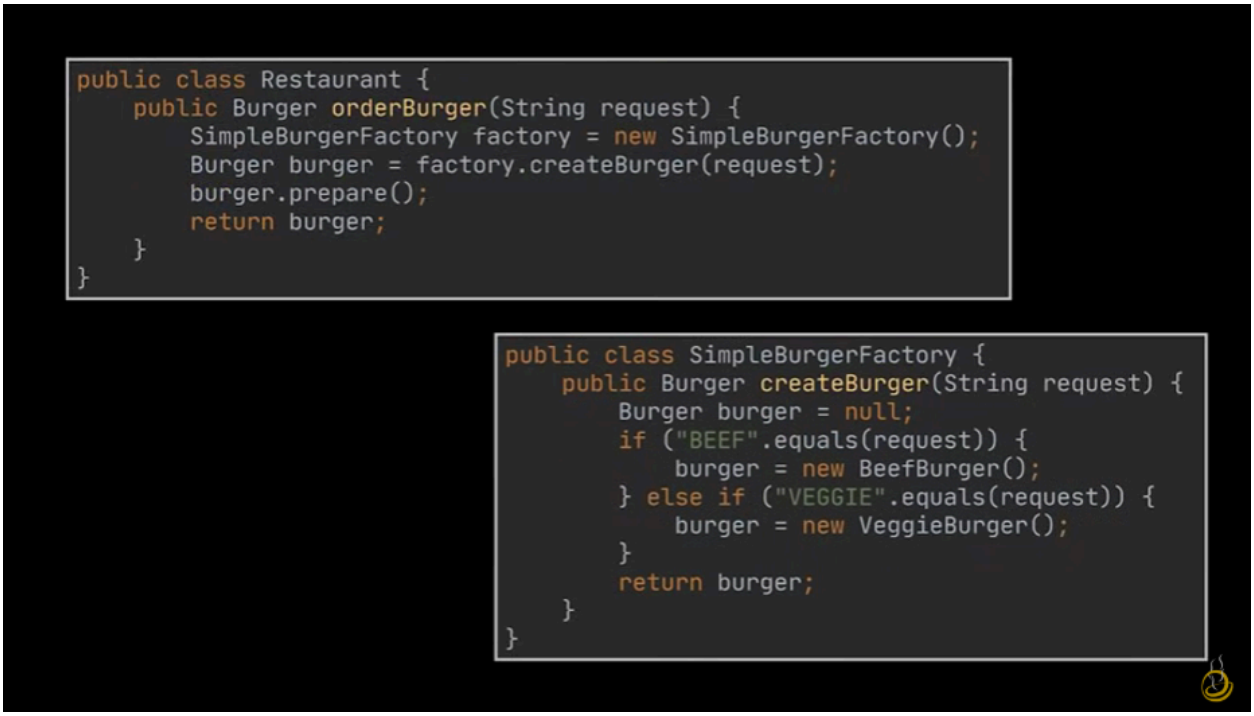


🔷 1. The Problem

- Each burger type (`BeefBurger`, `VeggieBurger`, `ChickenBurger`) is a **different class**.
- If no **common base class/interface**, the method can't return multiple product types.
- Adding new recipes forces us to **modify existing code** → violates:
  - **Open-Closed Principle (OCP)** → কোড পরিবর্তন না করে এক্সটেন্ড করা উচিত
  - **Single Responsibility Principle (SRP)** → প্রতিটি ক্লাসের একটাই দায়িত্ব থাকা উচিত

🔷 2. Step 1 – Simple Factory Idiom





- **Idea**: Encapsulate creation logic in a separate class → `BurgerFactory`.
- **Responsibility**: Only creates burgers, nothing else.
- **Flow**:

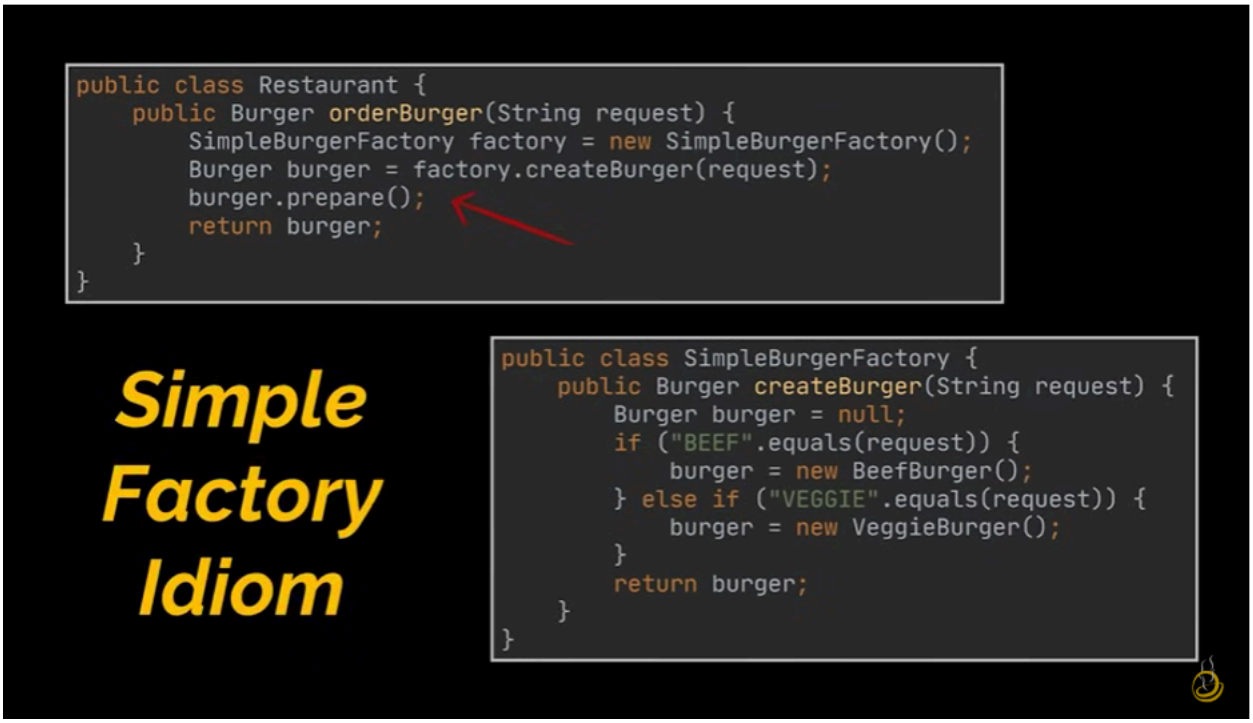
1. Client (`Restaurant`) calls factory.
2. Factory decides which burger to create.
3. Returns a `Burger` (base class/interface).

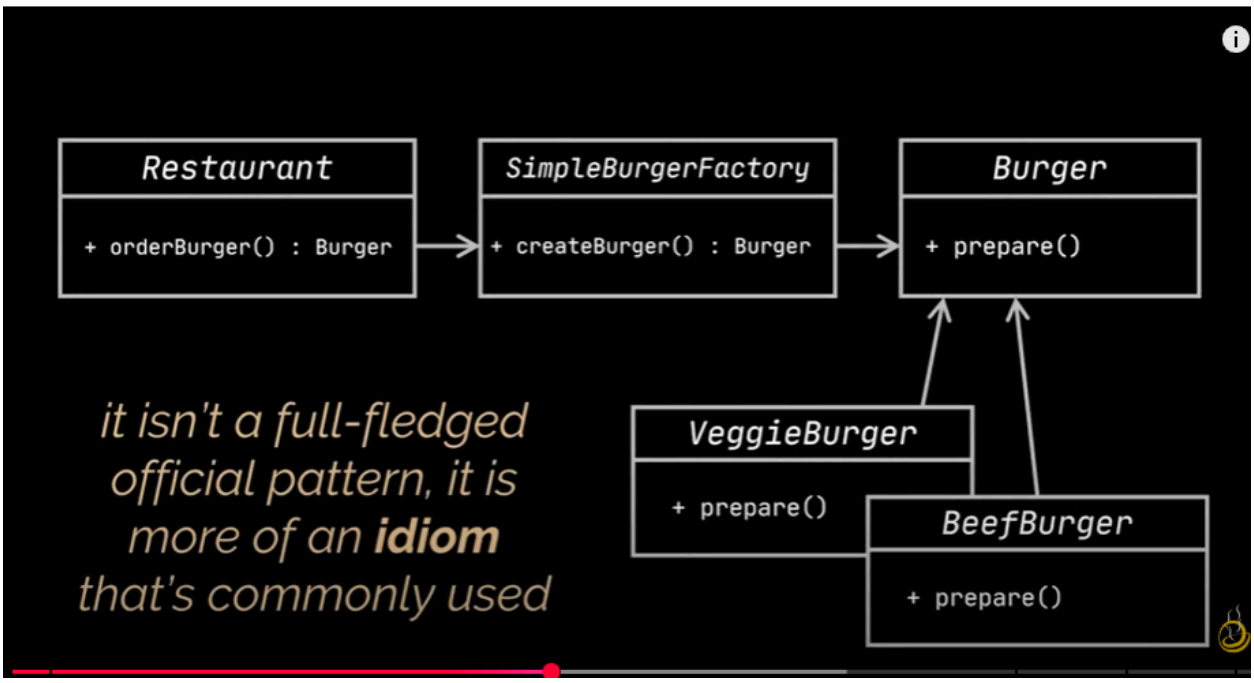
UML Mapping

- **Client** → `Restaurant`
- **Factory** → `BurgerFactory`
- **Product Interface** → `Burger`
- **Concrete Products** → `BeefBurger`, `VeggieBurger`, `ChickenBurger`

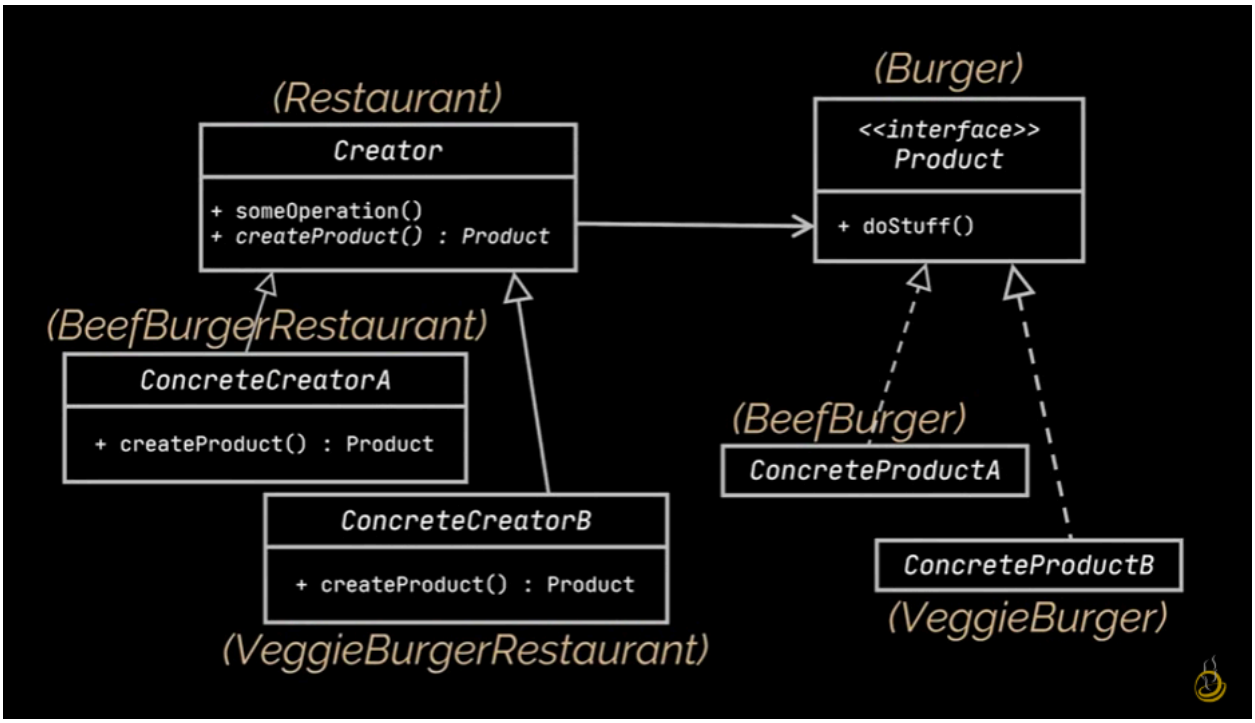
⚠️ Limitation: Still **open for modification** (adding new `if` / `switch` for new burgers).

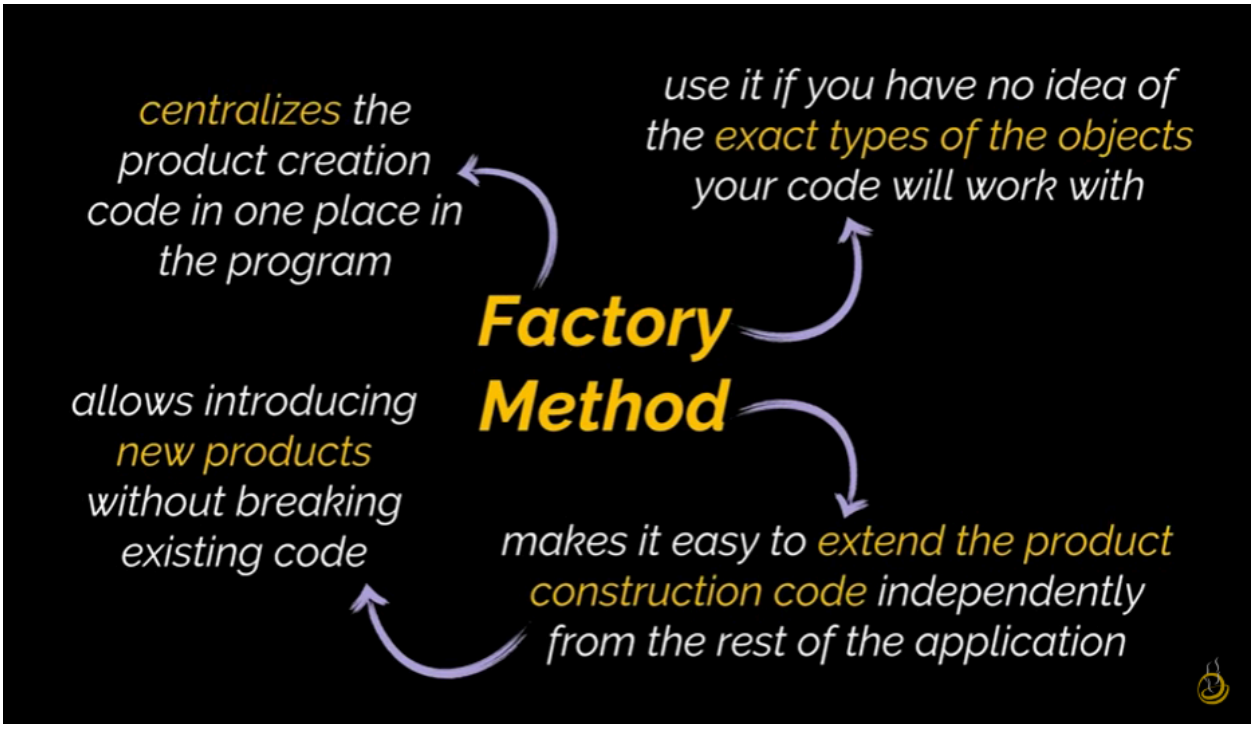
🔷 3. Step 2 – Factory Method Pattern




Restaurant
+ orderBurger() : Burger
SimpleBurgerFactory
+ createBurger() : Burger
Burger
+ prepare()
it isn't a full-fledged official pattern, it is more of an idiom that's commonly used
VeggieBurger
+ prepare()
BeefBurger
+ prepare()


public abstract class Restaurant {
public Burger orderBurger() {
Burger burger = createBurger();
burger.prepare();
return burger;
}
public abstract Burger createBurger();
}
public class BeefBurgerRestaurant extends Restaurant {
@Override
public Burger createBurger() {
return new BeefBurger();
}
}
public class VeggieBurgerRestaurant extends Restaurant {
@Override
public Burger createBurger() {
return new VeggieBurger();
}
}
public interface Burger {
void prepare();
}
public class BeefBurger
implements Burger {
@Override
void prepare() {
// prepare beef
// burger code
}
}
public class VeggieBurger
implements Burger {
@Override
void prepare() {
// prepare veggie
// burger code
}
}
5:47 / 8:27 • The Factory Method Pattern





- **Upgrade** from Simple Factory.
- **Key Difference**: Instead of one central factory, the **creation responsibility is delegated to subclasses**.

- **How**:
  - `Restaurant` becomes **abstract**.
  - Defines an **abstract method** → `createBurger()`.
  - Subclasses (`BeefBurgerRestaurant`, `VeggieBurgerRestaurant`) implement it.
- **Inheritance-based**: Factory Method relies on **subclassing**.

UML Mapping

- **Product Interface** → `Burger`
- **Concrete Products** → `BeefBurger`, `VeggieBurger`
- **Creator (Abstract)** → `Restaurant` (with `createBurger()`)
- **Concrete Creators** → `BeefBurgerRestaurant`, `VeggieBurgerRestaurant`

🔷 4. Benefits

- **Loosens coupling** → product creation separated from usage.
- **Extensible** → add new products without breaking client code.
- **Follows OCP & SRP**.

🔷 5. When to Use

- When you **don't know exact product types** in advance.
- When you want to **extend product families** without touching client code.
- When you need **flexibility** in object creation.

🔷 6. Limitation

- If you expand to **multiple product families** (e.g., Italian-style burgers), Factory Method alone isn't enough.
- Leads to **Abstract Factory Pattern** → next step.

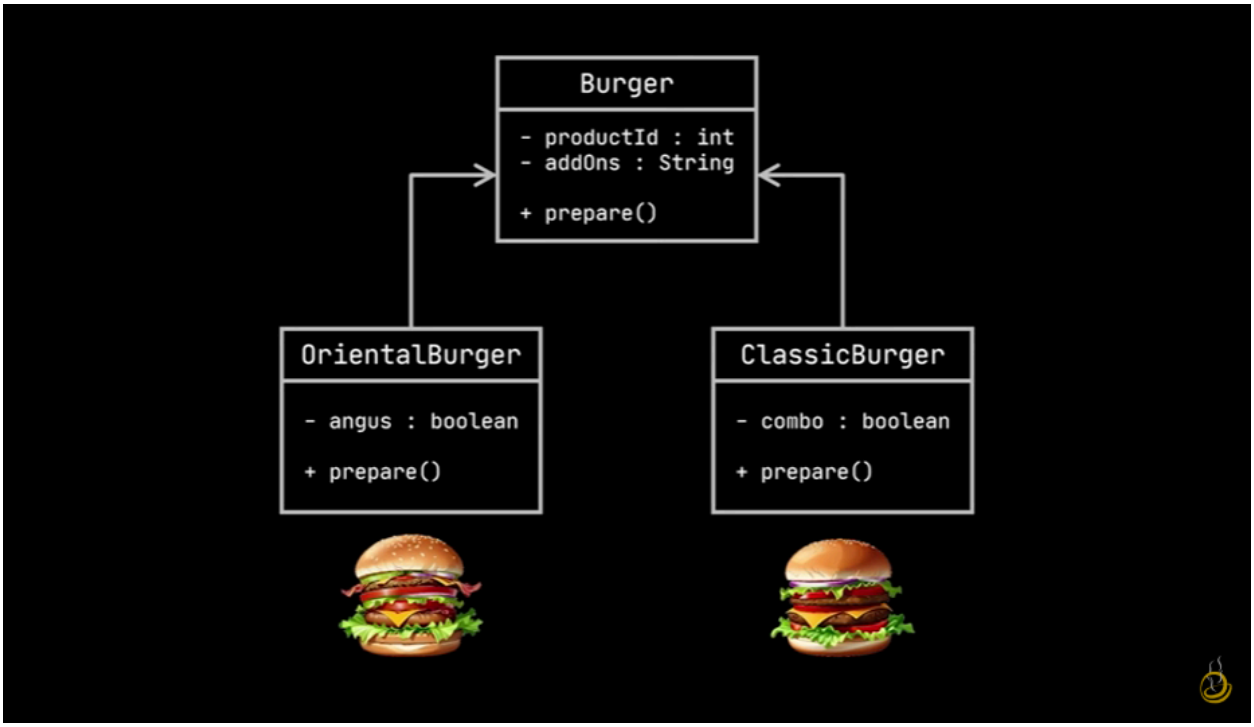
🔑 Bengali Keywords

English Term	বাংলা অনুবাদ
Factory	কারখানা / ফ্যাক্টরি
Product	পণ্য / প্রোডাক্ট
Creator	সৃষ্টিকারী
Concrete Product	নির্দিষ্ট পণ্য
Abstract Class	বিমূর্ত ক্লাস
Interface	ইন্টারফেস
Open-Closed Principle	খোলা-বন্ধ নীতি
Single Responsibility	একক দায়িত্ব নীতি

✅ **Summary**:

- **Simple Factory** → centralizes creation but still open for modification.
- **Factory Method** → delegates creation to subclasses, making code extensible and OCP-compliant.
- Next step → **Abstract Factory** for handling multiple product families.

🍔 Factory Patterns – Study Note



🔷 1. The Setup

- **Context**: Burger delivery app.
- Each burger (`BeefBurger`, `VeggieBurger`, etc.) → a **class**.
- All extend a **common base class/interface** → `Burger`.

🔷 2. The Problem

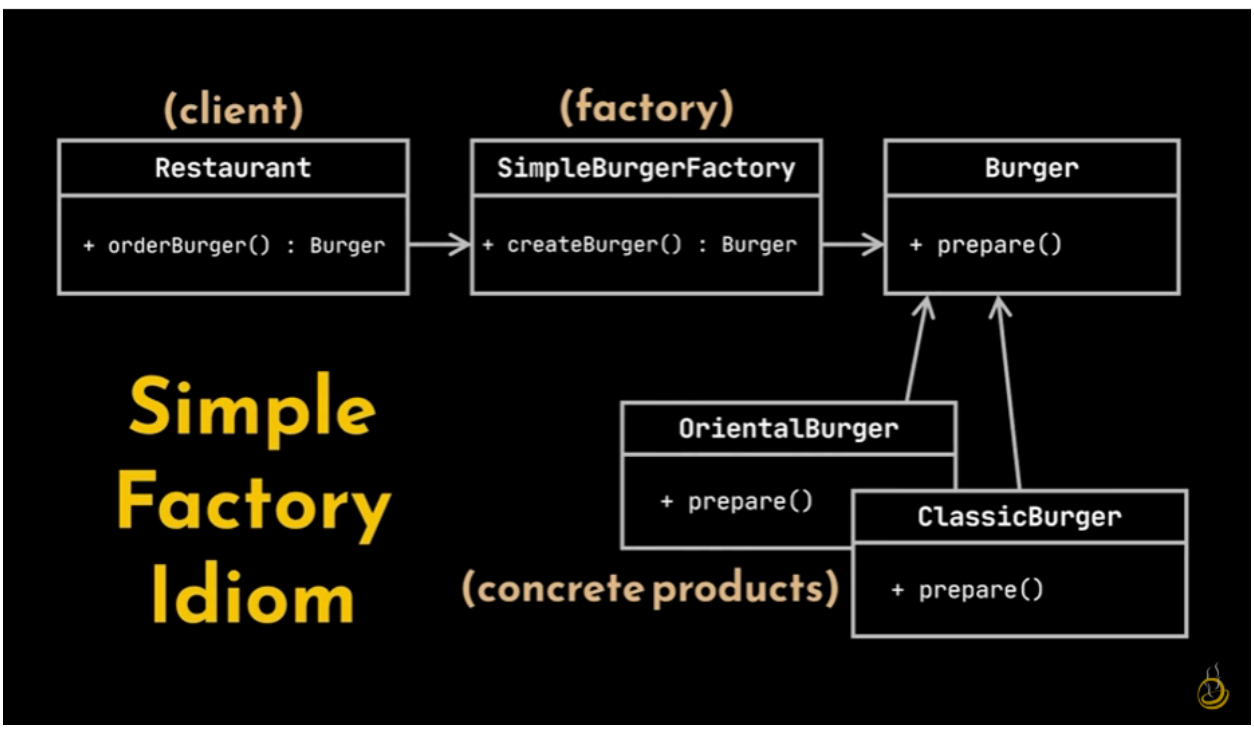
- Adding new recipes → must **modify existing code**.
- Violates **SOLID principles**:
  - **OCP (Open-Closed Principle)** → কোড পরিবর্তন না করে এক্সটেন্ড করা উচিত
  - **SRP (Single Responsibility Principle)** → এক ক্লাসের একটাই দায়িত্ব থাকা উচিত
- Leads to **duplication** and **tight coupling**.

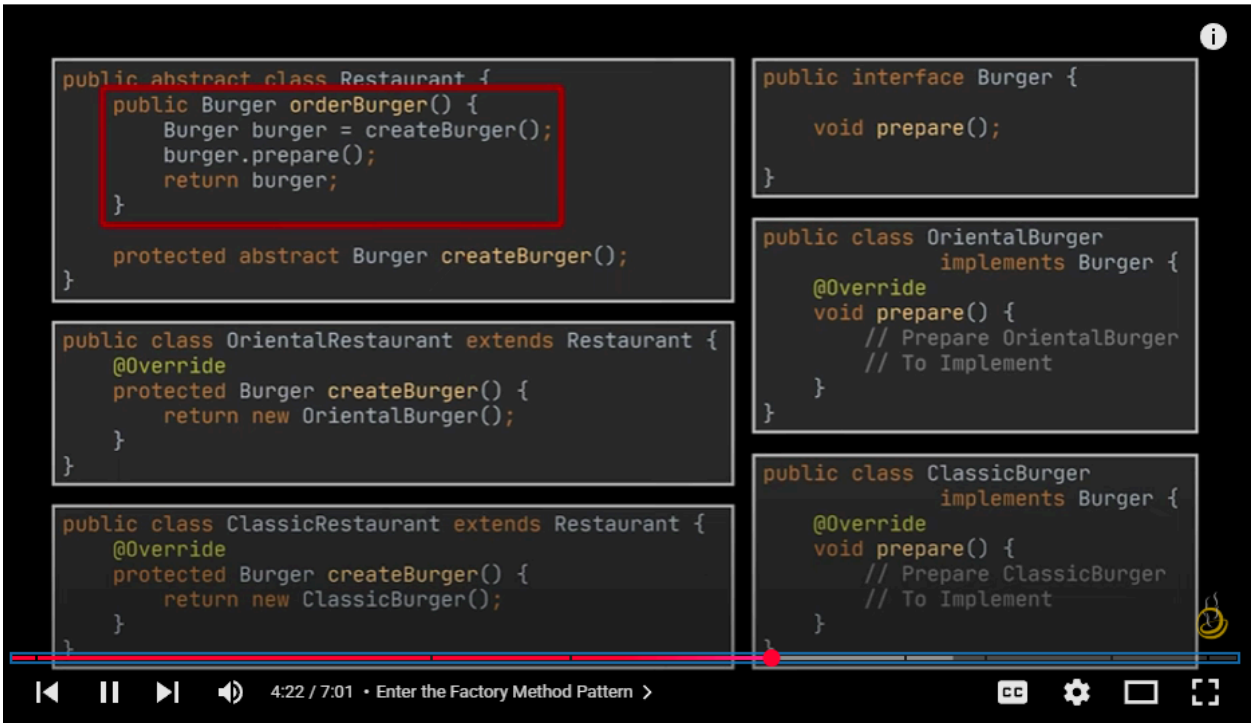
🔷 3. Simple Factory Idiom

```java
public class Restaurant {
    public Burger orderBurger(String request) {
        SimpleBurgerFactory factory = new SimpleBurgerFactory();
        Burger burger = factory.createBurger(request);
        burger.prepare();
        return burger;
    }
}
```

```java
public class SimpleBurgerFactory {

    public Burger createBurger(String request) {
        if ("CLASSIC".equals(request)) {
            return new ClassicBurger();
        } else if ("ORIENTAL".equals(request)) {
            return new OrientalBurger();
        } else if ("EUROPEAN".equals(request)) {
            return new EuropeanBurger();
        }
    }
}
```

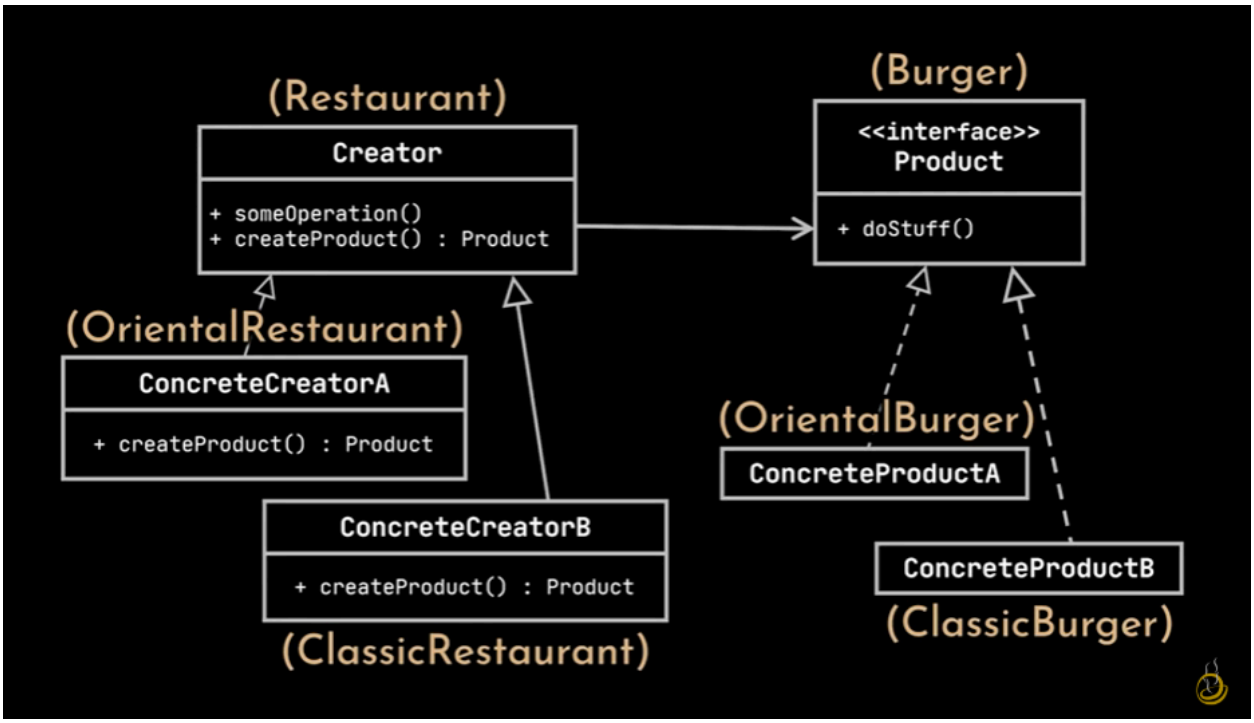
Settings





- **Encapsulate creation logic** in a separate class → `BurgerFactory`.
- **Responsibility**: Only creates burgers.
- **Flow**:
  1. Client (`Restaurant`) calls factory.
  2. Factory decides which burger to create.
  3. Returns a `Burger` object.

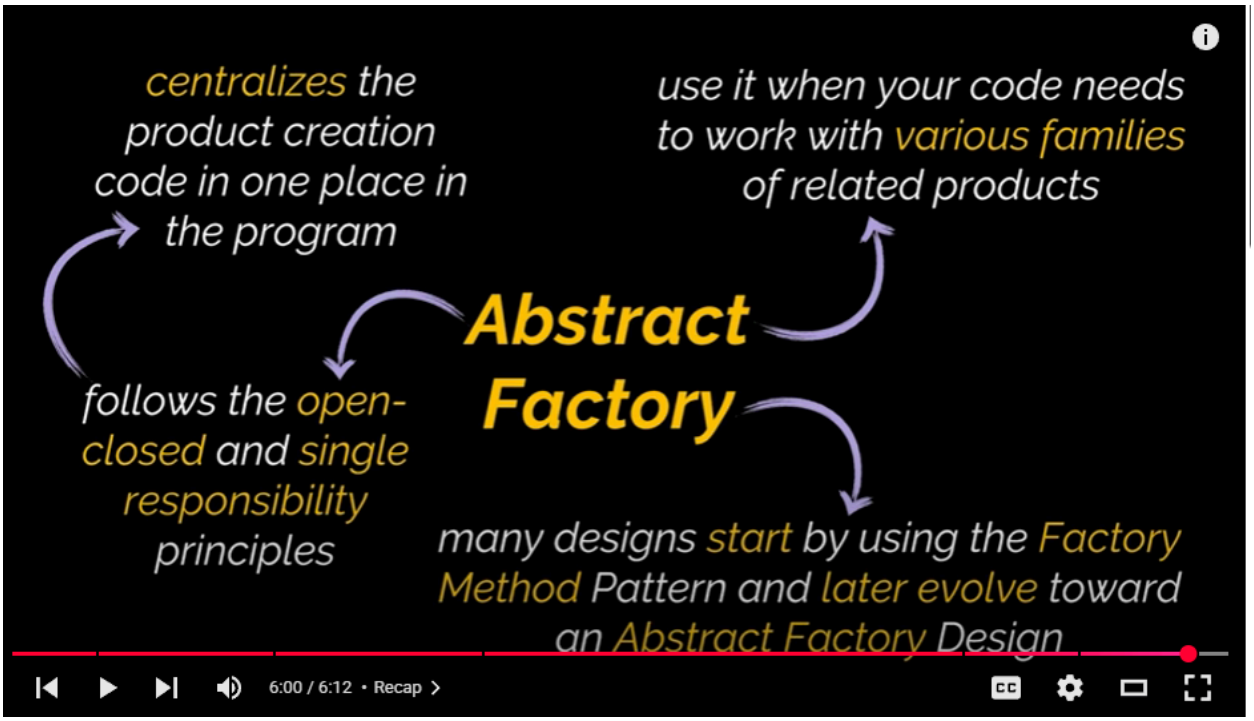
UML Mapping



- **Client** → `Restaurant`
- **Factory** → `BurgerFactory`
- **Product Interface** → `Burger`
- **Concrete Products** → `BeefBurger`, `VeggieBurger`, `ChickenBurger`

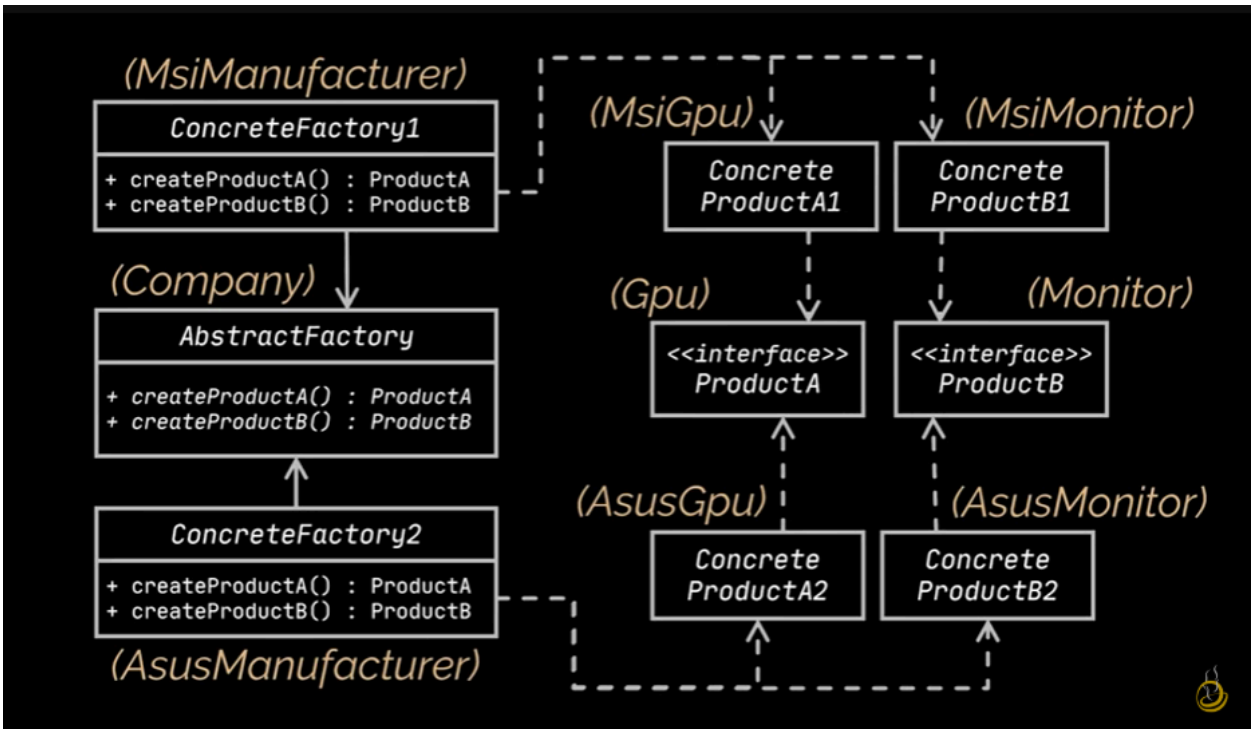
⚠️ Limitation: Still **open for modification** (new `if` / `switch` for new burgers).

🔷 4. Factory Method Pattern



- **Upgrade** from Simple Factory.
- **Key Idea**: Delegate creation to subclasses.
- **How**:
  - `Restaurant` → abstract class.
  - Defines abstract method → `createBurger()`.
  - Subclasses (`ClassicRestaurant`, `OrientalRestaurant`) implement it.
- **Inheritance-based**: Factory Method relies on **subclassing**.

UML Mapping



- **Product Interface** → `Burger`
- **Concrete Products** → `BeefBurger`, `VeggieBurger`
- **Creator (Abstract)** → `Restaurant` (with `createBurger()`)
- **Concrete Creators** → `ClassicRestaurant`, `OrientalRestaurant`

🔷 5. Benefits

- **Looser coupling** → separates creation from usage.
- **Extensible** → add new products without breaking client code.
- **Follows OCP & SRP**.

🔷 6. When to Use

- When product types are **not known in advance**.
- When you want to **extend product families** easily.
- When you need **flexibility** in object creation.

🔷 7. Limitation → Transition to Abstract Factory

- If business expands (e.g., add **pizzas** alongside burgers):
  - Factory Method subclasses become cluttered.
  - Back to **request-object dependency**.
- Solution → **Abstract Factory Pattern** (next step).

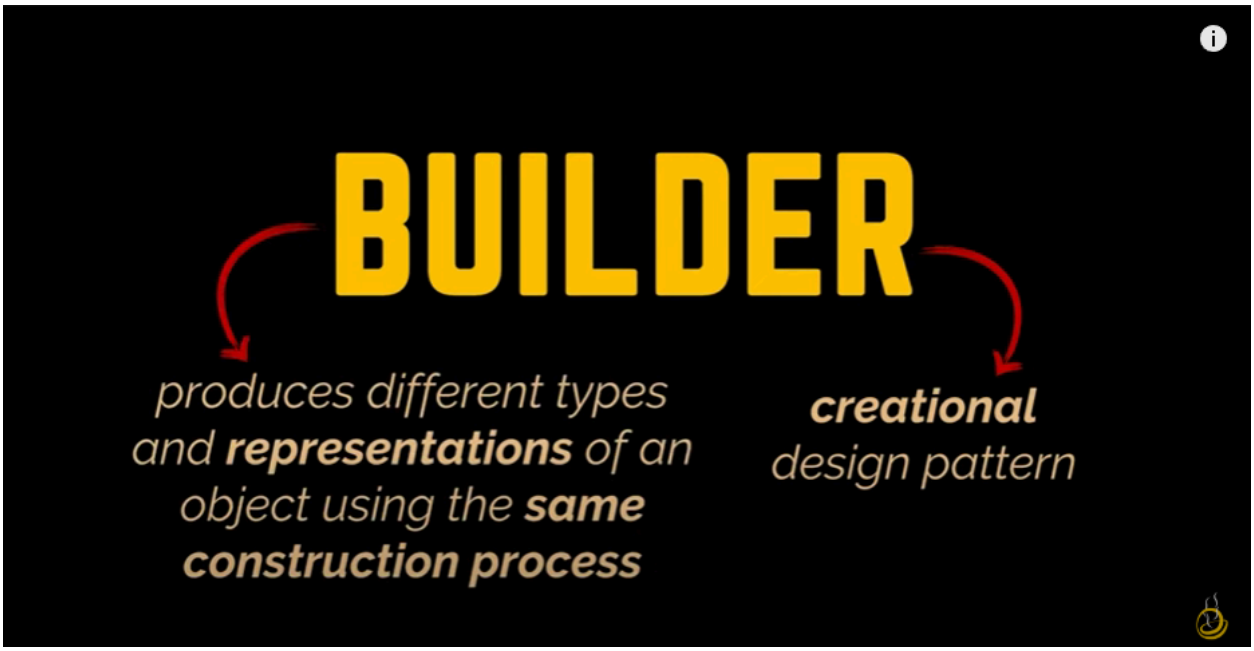
🔑 Bengali Keywords

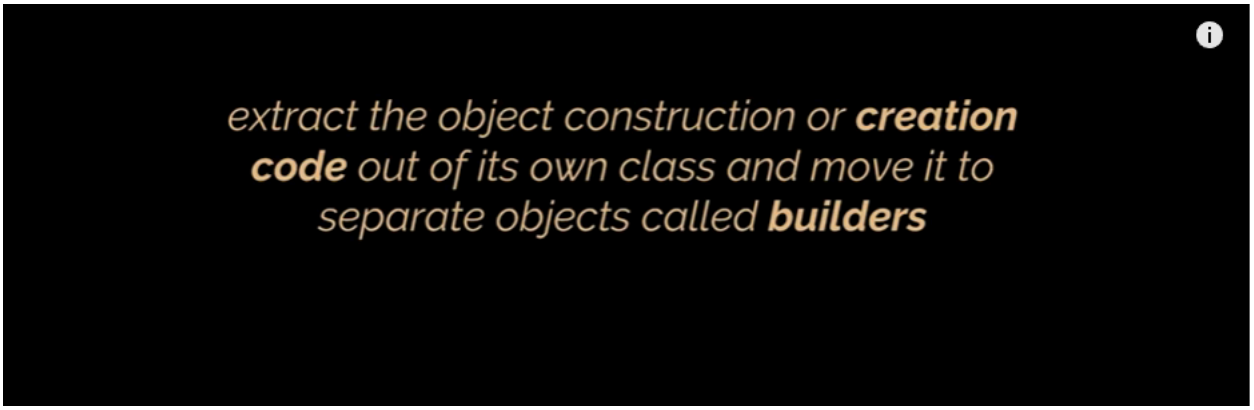
English Term	বাংলা অনুবাদ
Factory	ফ্যাক্টরি / কারখানা
Product	পণ্য
Creator	সৃষ্টিকারী
Concrete Product	নির্দিষ্ট পণ্য
Abstract Class	বিমূর্ত ক্লাস
Interface	ইন্টারফেস
Open-Closed Principle	খোলা-বন্ধ নীতি
Single Responsibility	একক দায়িত্ব নীতি

✅ **Summary**:

- **Simple Factory** → centralizes creation but still open for modification.
- **Factory Method** → delegates creation to subclasses, making code extensible and OCP-compliant.
- **Abstract Factory** → next step for handling **multiple product families** (e.g., burgers + pizzas).

🚗 Builder Pattern – Study Note





🔷 Definition

- **Creational Design Pattern**
- Lets you construct **complex objects step by step**.
- Allows producing **different types/representations** of an object using the **same construction process**.

🔷 The Problem

- Complex classes (e.g., `Car`) often have **many fields**: brand, model, color, doors, screen, weight, height, ID, etc.

- Issues with traditional constructors:
  - **Telescoping constructors** → too many parameters.
  - **Overloaded constructors** → explosion of variants.
  - **Scattered initialization** → messy client code.

🔷 The Solution

- Move construction logic into a **separate Builder class**.
- Builder contains the **same fields** as the product (`Car`).
- Provides **fluent setter-like methods** (without `set` prefix).
- Provides a `build()` **method** → returns the final product.
- The product's constructor is **private/package-private** → forces use of the builder.

🔷 Example Flow

```
Car car = new CarBuilder()
            .brand("Bugatti")
            .model("Chiron")
            .color("Blue")
            .doors(2)
            .build();
```

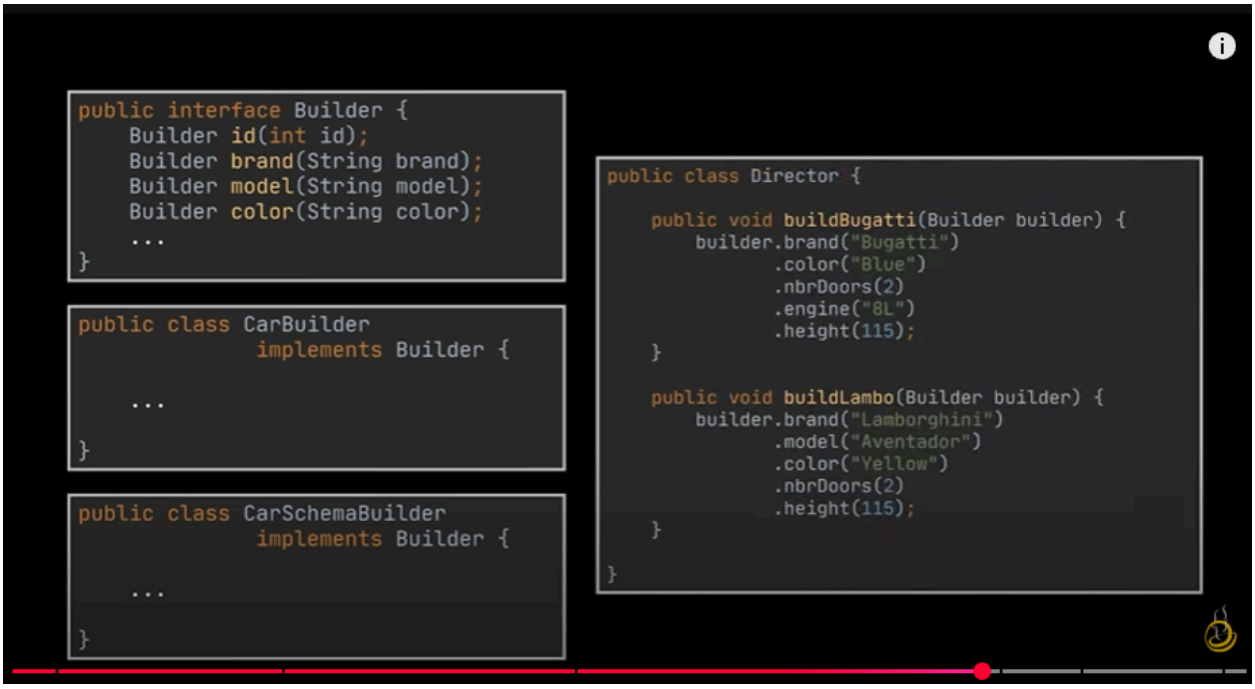
- Builder collects parameters.
- `build()` calls the hidden constructor of `Car`.

🔷 Director Class

- **Optional** helper that defines **construction sequences**.
- Encapsulates **reusable routines** (e.g., how to build a Bugatti vs Lamborghini).
- Hides construction details from client code.
- Can reuse the same process for **different builders** (e.g., `CarBuilder` and `CarSchemaBuilder`).

🔷 Builder Interface

v

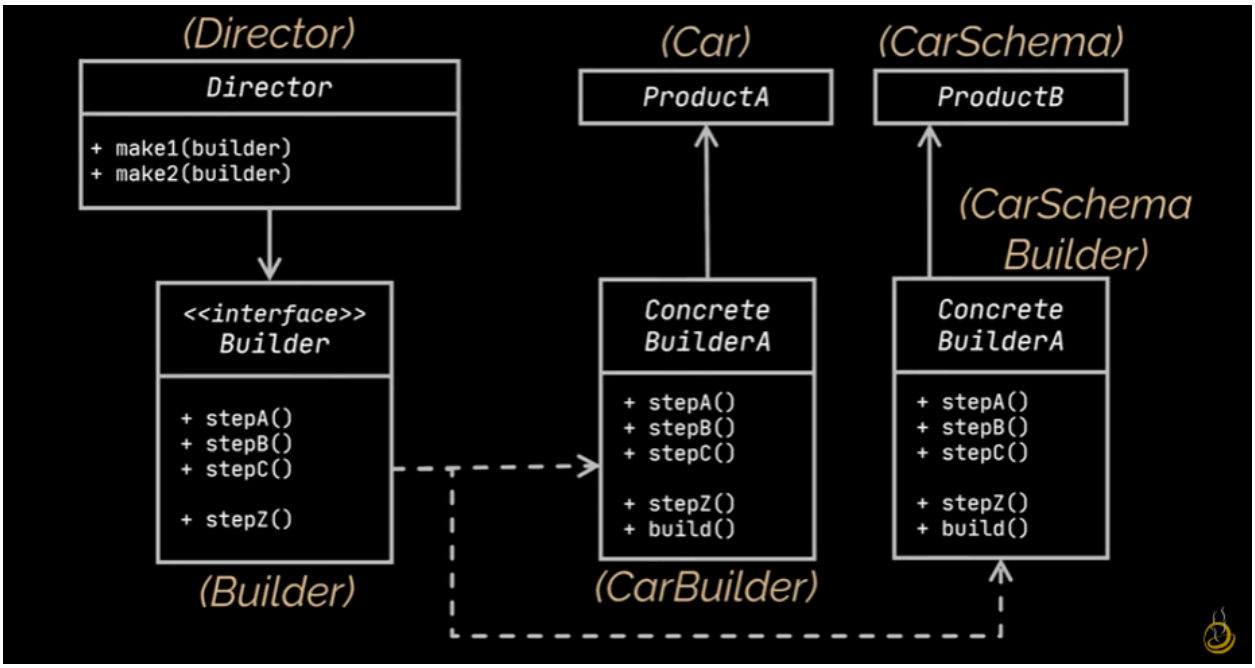


- When multiple builders exist, define a **common interface**.
- Example:
  - `CarBuilder` → builds `Car`.

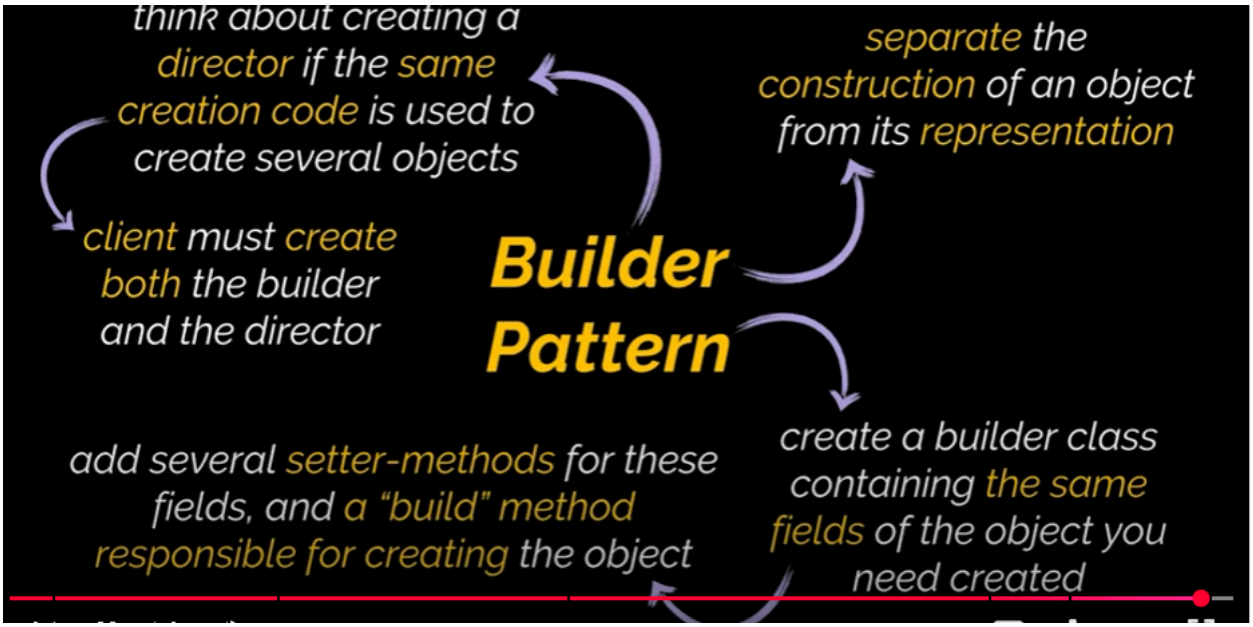
- `CarSchemaBuilder` → builds `CarSchema`.

- Both implement `Builder` interface with the same construction steps.
- Director works with the **interface**, not concrete builders.

🔷 UML Structure (Text Form)




```
Director
  + construct(builder: Builder)

Builder (interface)
  + setBrand()
  + setModel()
  + setColor()
  + build()

ConcreteBuilder1 (CarBuilder) → Product: Car
ConcreteBuilder2 (CarSchemaBuilder) → Product: CarSchema
```



🔷 Benefits

- ✅ Cleaner object creation (no telescoping constructors).
- ✅ Step-by-step construction → flexible and readable.
- ✅ Can reuse construction logic via **Director**.
- ✅ Supports multiple representations of the same product.
- ✅ Follows **SRP** (separates construction from representation).

🔷 When to Use

- When objects are **complex** with many optional fields.
- When you need **different representations** of the same product.
- When you want to **reuse construction sequences** across products.

🔑 Bengali Keywords

English Term	বাংলা অনুবাদ
Builder	নির্মাতা
Director	পরিচালক

English Term	বাংলা অনুবাদ
Product	পণ্য
Concrete Builder	নির্দিষ্ট নির্মাতা
Build Method	নির্মাণ পদ্ধতি
Representation	উপস্থাপনা

✅ **Summary**:

The **Builder Pattern** separates **object construction** from its **representation**.

- **Builder** → step-by-step creation.
- **Director** → reusable construction sequences.
- **Concrete Builders** → produce different product variants.

🚗 Prototype Pattern – Study Note

🔷 Definition

- **Creational Design Pattern**
- Lets you **clone existing objects** without coupling code to their concrete classes.
- Delegates the **cloning process to the object itself**.
- Each object that supports cloning is called a **Prototype**.

Absolutely, Tanjim. Here's a concise, Bengali-annotated short note on Java's `Cloneable` interface and `clone()` method:

🧠 Java Cloneable & clone() — Short Note

🔧 `Cloneable` Interface

- A **marker interface** (no methods).
- Signals to JVM: this class allows cloning via `Object.clone()`.

Bengali: `Cloneable` ইন্টারফেসে কোনো মেথড নেই — এটা JVM কে বলে, এই ক্লাস ক্লোন করা যাবে।

🧪 `clone()` Method

- Comes from `Object` class.
- Must be **overridden** in your class.
- Uses `super.clone()` to perform a **shallow copy**.
- Requires `try-catch` for `CloneNotSupportedException`.

```
public Person clone() {
  try {
    return (Person) super.clone();
  } catch (CloneNotSupportedException e) {
    return null;
  }
}
```

Bengali: `clone()` মেথড `Object` থেকে আসে — `super.clone()` shallow copy তৈরি করে।

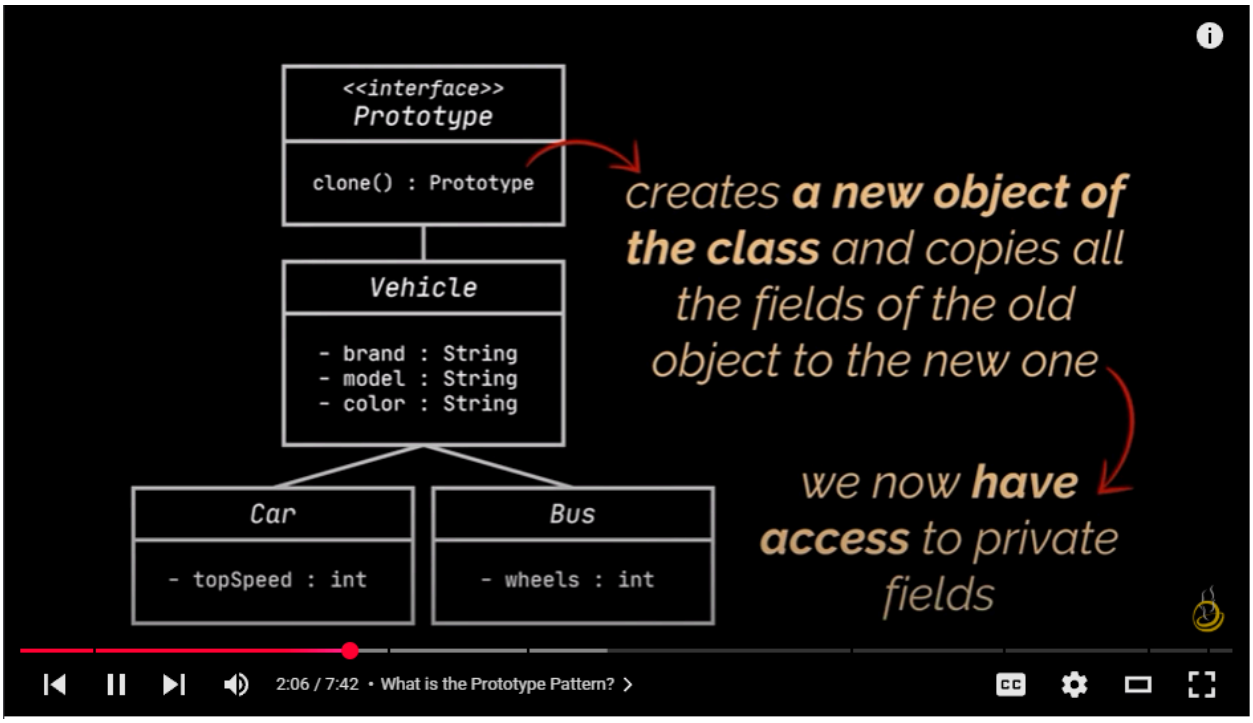
```
class Person implements Cloneable {
  String name;
  int age;

  Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public Person clone() {
    try {
      return (Person) super.clone();
    } catch (CloneNotSupportedException e) {
      return null;
```

```
    }
  }
}

public class Main {
  public static void main(String[] args) {
    Person original = new Person("Tanjim", 22);
    Person copy = original.clone();

    copy.name = "Ahmed";

    System.out.println(original.name); // Tanjim
    System.out.println(copy.name);
    System.out.print(copy.age);// Ahmed
  }
}
```

✅ Summary Table

Concept	Type	Role
`Cloneable`	Interface	Marker (no methods)
`clone()`	Method	From `Object`, must override
`super.clone()`	Method	Performs actual cloning

Let me know if you want a cheat sheet comparing shallow vs deep copy or Dart vs Java cloning logic. We can scaffold it contest-style.



🔷 The Problem

- Copying objects manually requires:
  - Knowing the **concrete class**.
  - Access to **private fields**.
  - Writing **repetitive initialization code**.
- In third-party or interface-only scenarios, you may not know the concrete class.
- Tight coupling makes code fragile and hard to extend.

🔷 The Solution

- Define a **Prototype interface** with a `clone()` method.
- Each concrete class implements `clone()`:
  - Uses a **copy constructor** to duplicate fields.
  - Handles private fields internally.

- Clients call `clone()` without knowing the concrete class.

🔷 UML Structure (Text Form)

```
Prototype (interface/abstract class)
  + clone(): Prototype

ConcretePrototype (Car, Bus, etc.)
  + clone(): ConcretePrototype

Client
  - uses clone() without knowing concrete class
```

🔷 Shallow vs Deep Copy

- **Shallow Copy** → references are copied, not objects.
  - Example: Car clone shares the same `GpsSystem` object.

    - Changes in GPS affect both cars.
- **Deep Copy** → new objects are created for referenced fields.
    - Example: Car clone creates a new `GpsSystem`.
    - Changes in one GPS do not affect the other.

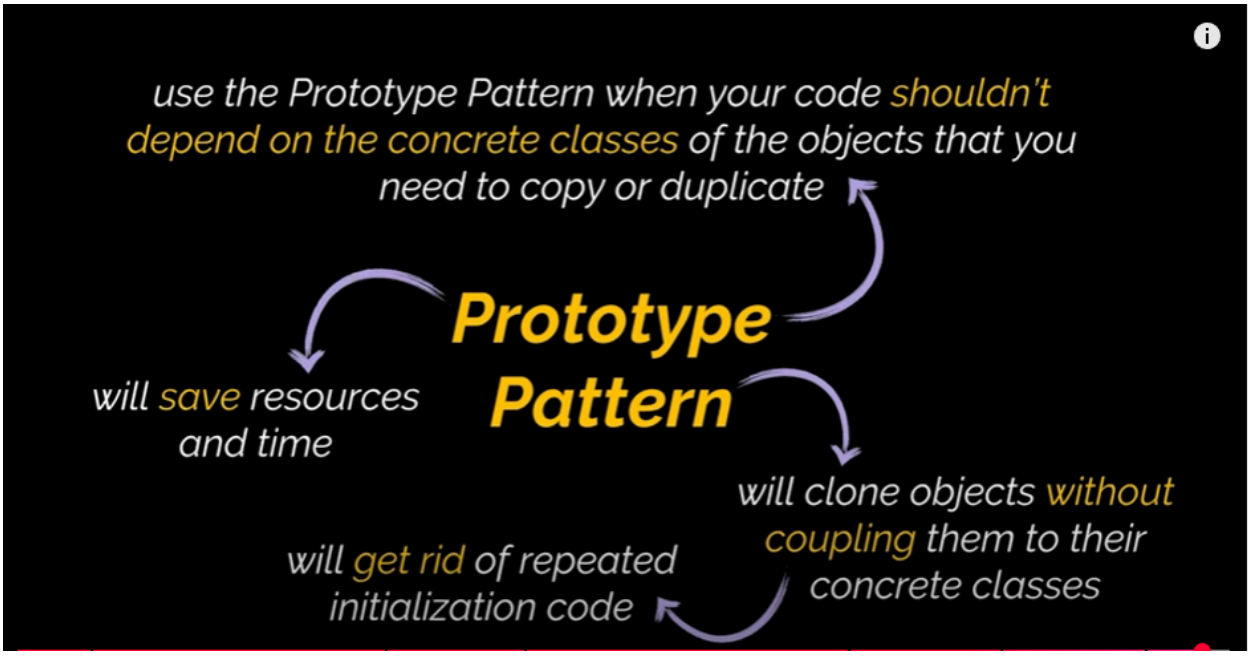
🔷 Prototype Registry

- A **catalog of pre-built prototypes** for frequent use.
- Typically implemented as a `HashMap<String, Prototype>`.
- Workflow:
    1. Client requests a prototype by key.
    2. Registry finds it.
    3. Registry clones it and returns the copy.
- Acts like a **factory of clones**.

🔷 Benefits

- ✅ Decouples object creation from concrete classes.
- ✅ Simplifies duplication of complex objects.
- ✅ Saves resources when object creation is expensive.
- ✅ Reduces repetitive initialization code.
- ✅ Works seamlessly with inheritance hierarchies.

🔷 When to Use



- When object creation is **costly or complex**.
- When you need to **decouple code from concrete classes**.
- When you want to **store and reuse frequently cloned objects**.
- When working with **interface-based APIs** where concrete classes are unknown.

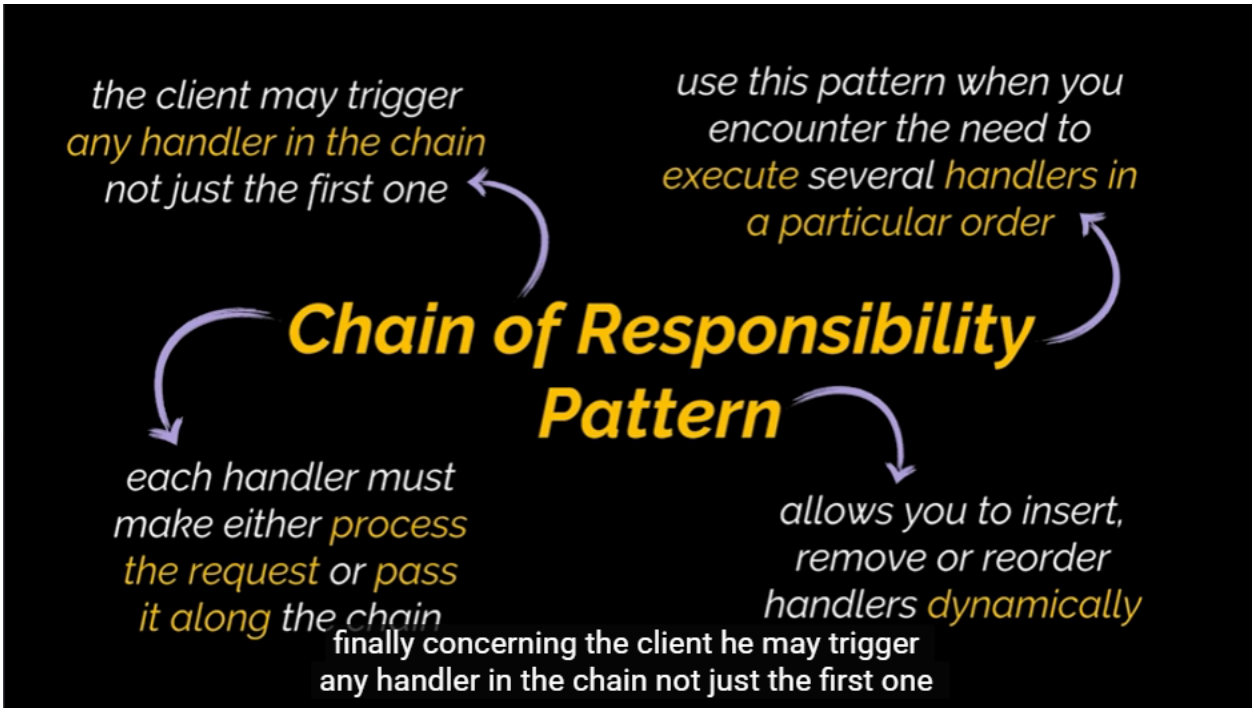
🔑 Bengali Keywords

English Term	বাংলা অনুবাদ
Prototype	প্রোটোটাইপ
Clone	কপি / প্রতিলিপি
Shallow Copy	অগভীর কপি
Deep Copy	গভীর কপি
Prototype Registry	প্রোটোটাইপ রেজিস্ট্রি
Copy Constructor	কপি কনস্ট্রাক্টর

✅ **Summary**:

The **Prototype Pattern** lets objects **clone themselves** via a `clone()` method, avoiding tight coupling to concrete classes. It supports **shallow and deep copies**, and can be extended with a **Prototype Registry** for efficiency.

🔗 Chain of Responsibility Pattern – Study Note



🔷 Definition

- **Behavioral Design Pattern**
- Lets you pass a request along a **chain of handlers**.
- Each handler decides:
  - ✅ Process the request
  - ➡️ Pass it to the next handler
  - ⛔ Stop the chain

🔷 Real-World Analogy

- Calling a bank hotline:
  - Operator 1 → checks if they can help.
  - If not, forwards to Operator 2.
  - Eventually, the right operator resolves the issue.
- Each operator = **Handler**.
- The call transfer = **passing the request along the chain**.

🔷 Structure (UML in Text Form)

```
Handler (interface / abstract class)
  + handle(request)
  + setNext(handler)

BaseHandler (optional)
  - next: Handler
  + handleNext(request)

ConcreteHandlers
  + UserExistsHandler
  + ValidPasswordHandler
  + RoleCheckHandler

Client
  - builds the chain
  - sends request to any handler
```

🔷 Example: Authentication App

- **Database** → stores usernames & passwords.

- **Handlers**:
  - `UserExistsHandler` → checks if username exists.
  - `ValidPasswordHandler` → checks if password matches.
  - `RoleCheckHandler` → checks if user is admin.
- **AuthenticationService**:
  - Holds the root handler.
  - Calls `handle()` on login attempt.
  - If all checks pass → user authorized.

🔷 Benefits

- ✅ Decouples sender from receiver.
- ✅ Flexible: add, remove, or reorder handlers at runtime.
- ✅ Reusable handlers (each has a single responsibility).
- ✅ Client can start request at **any handler**, not just the first.

🔷 When to Use

- When multiple objects can handle a request, but you don't know which one in advance.
- When you want to **process requests in a sequence**.
- When you want to **change processing logic dynamically**.

🔑 Bengali Keywords

English Term	বাংলা অনুবাদ
Chain of Responsibility	দায়িত্বের শৃঙ্খল
Handler	হ্যান্ডলার / প্রক্রিয়াকারী
Request	অনুরোধ

English Term	বাংলা অনুবাদ
Client	ক্লায়েন্ট
Concrete Handler	নির্দিষ্ট হ্যান্ডলার

✅ **Summary**:

The **Chain of Responsibility Pattern** organizes request handling into a **pipeline of handlers**. Each handler either processes the request, passes it along, or stops the chain. It's especially useful for **authentication, logging, validation, and middleware pipelines**.

🔗 Chain of Responsibility Pattern – Java Example

```
// Handler abstract class (BaseHandler)
abstract class Handler {
    private Handler next;

    // Set the next handler in the chain
    public Handler setNext(Handler next) {
        this.next = next;
        return next; // allows chaining
    }

    // Template method
    public boolean handle(String username, String password) {
        if (!process(username, password)) {
            return false; // stop chain if check fails
        }
        if (next == null) {
            return true; // end of chain
        }
        return next.handle(username, password);
    }
```

```
    // Each concrete handler implements its own check
    protected abstract boolean process(String username, String password);
}

// Simulated Database
class Database {
    private java.util.Map<String, String> users = new java.util.HashMap<>();

    public Database() {
        users.put("tanjim", "1234");
        users.put("admin", "adminpass");
    }

    public boolean userExists(String username) {
        return users.containsKey(username);
    }

    public boolean isValidPassword(String username, String password) {
        return users.get(username).equals(password);
    }

    public boolean isAdmin(String username) {
        return username.equals("admin");
    }
}

// Concrete Handlers
class UserExistsHandler extends Handler {
    private Database db;

    public UserExistsHandler(Database db) {
        this.db = db;
    }

    @Override
```

```
    protected boolean process(String username, String password) {
        System.out.println("Entered in user checking module");
        if (!db.userExists(username)) {
            System.out.println("User does not exist. Please sign up.");
            return false;
        }
        System.out.println("User exists");
        return true;
    }
}

class ValidPasswordHandler extends Handler {
    private Database db;

    public ValidPasswordHandler(Database db) {
        this.db = db;
    }

    @Override
    protected boolean process(String username, String password) {
        System.out.println("Entered in password checking module");
        if (!db.isValidPassword(username, password)) {
            System.out.println("Invalid password!");
            return false;
        }
        System.out.println("Password matched");
        return true;
    }
}

class RoleCheckHandler extends Handler {
    private Database db;

    public RoleCheckHandler(Database db) {
        this.db = db;
    }
```

```
    @Override
    protected boolean process(String username, String password) {
        System.out.println("Entered in role check module");
        if (db.isAdmin(username)) {
            System.out.println("Welcome Admin! Extra privileges granted.");
        } else {
            System.out.println("Welcome User!");
        }
        return true;
    }
}

// Client / Authentication Service
public class Main {
    public static void main(String[] args) {
        Database db = new Database();

        // Build the chain
        Handler handler = new UserExistsHandler(db);
        handler.setNext(new ValidPasswordHandler(db))
               .setNext(new RoleCheckHandler(db));

        // Test login for admin
        System.out.println("---- Admin Login ----");
        handler.handle("admin", "adminpass");

        // Test login for tanjim
        System.out.println("---- Tanjim Login ----");
        handler.handle("tanjim", "1234");
    }
}
```

🔷 Output

```
---- Test 1 ----
Welcome User!
---- Test 2 ----
Invalid password!
---- Test 3 ----
User does not exist. Please sign up.
---- Test 4 ----
Welcome Admin! Extra privileges granted.
```

✨ Specialty of Chain of Responsibility

- **Decouples sender from receiver** → client doesn't know which handler will process the request.
- **Flexible** → handlers can be added, removed, or reordered at runtime.
- **Single Responsibility** → each handler checks one condition only.

🧠 Why Not Just Use `if-else` with `Database`?

✅ Pros:

- Simple and direct
- Fewer classes, less boilerplate
- Easy to follow for small systems

❌ Cons:

- Hard to **extend** without modifying core logic
- Becomes messy with **many conditions**

- Violates **Open/Closed Principle** (OCP) — you have to change existing code to add new checks

---

🧰 Why Use Chain of Responsibility?

✅ Pros:

- **Modular**: Each check is isolated in its own class
- **Flexible**: You can add/remove/reorder handlers without touching others
- **Reusable**: Handlers can be reused across different flows
- **Testable**: Each handler can be unit tested independently
- **Clean Architecture**: Promotes separation of concerns

❌ Cons:

- More classes and abstraction
- Slightly more setup for small projects

🧩 Bengali Analogy

- `if-else` = *একটি সিঙ্গেল রোড, যেখানে সব ট্রাফিক একসাথে চলে* — "A single road where all traffic moves together"
- Chain of Responsibility = *একটি মাল্টি-লেন হাইওয়ে, যেখানে প্রতিটি লেনের নিজস্ব নিয়ম আছে* — "A multi-lane highway where each lane has its own rules"

🧠 Strategic Use Case

Use `if-else` when:

- The logic is **simple**, **unlikely to change**, and **not reused**

Use **Chain of Responsibility** when:

- You expect **growth**, **modularity**, or **role-based logic**
- You want to **plug in/out** checks dynamically

- You're building a **framework**, not just a script

Would you like me to scaffold a side-by-side comparison of both approaches with Bengali annotations for revision clarity? Or simulate a contest-style bug trace showing how `if-else` breaks under scale?

🔷 One Use Case

Authentication pipelines (like above):

- Step 1: Check if user exists.
- Step 2: Validate password.
- Step 3: Check role/permissions.

Other real-world uses:

- Event handling in GUIs,
- Middleware in web frameworks,
- Logging systems with multiple loggers.

👉 Do you want me to also show you a **real-world web framework style example** (like middleware in a request pipeline) to see how this pattern is used in modern systems?

💡 Command Pattern – Study Note

🔷 Definition

- **Behavioral Design Pattern**
- Turns a **request/operation into a standalone object**.
- Encapsulates all details of the request:
  - The **receiver** (object that does the work)
  - The **method** to call

- The **arguments** (if any)

🔷 Real-World Analogy

- **Smart Home App**:
  - User wants to switch lights on/off in rooms or lamps.
  - Instead of each room duplicating light logic, we encapsulate the request into a **Command object**.
  - Example: `SwitchLightsCommand` → tells the `Light` (receiver) to toggle state.

🔷 Structure (UML in Text Form)

```
Command (interface)
  + execute()

ConcreteCommand (SwitchLightsCommand, OpenCurtainsCommand, etc.)
  - receiver: Light / Curtain / Door
  + execute() → calls receiver.action()

Invoker (Sender)
  - stores a Command
  - calls command.execute()

Receiver
  - actual business logic (e.g., Light.toggle())

Client
  - creates ConcreteCommand
  - assigns Receiver
  - passes Command to Invoker
```

🔷 Key Roles

- **Command** → interface with `execute()`.
- **ConcreteCommand** → implements request (e.g., `SwitchLightsCommand`).
- **Invoker (Sender)** → triggers command instead of calling receiver directly (e.g., `Room`).
- **Receiver** → contains actual business logic (e.g., `Light`).
- **Client** → configures commands, receivers, and invokers.

🔷 Benefits

- ✅ **Decouples** sender (Invoker) from receiver (business logic).
- ✅ **Extensible**: add new commands without changing existing code.
- ✅ Supports **undo/redo**, **logging**, **queueing**, and **scheduling**.
- ✅ Commands can be **stored, serialized, or passed around** like objects.
- ✅ Follows **Single Responsibility Principle** (each class has one role).

🔷 When to Use

- When you need to **parameterize objects with actions**.
- When you want to **queue, log, or schedule requests**.
- When you need **undo/redo functionality**.
- When you want to **decouple request senders from receivers**.

🔑 Bengali Keywords

English Term	বাংলা অনুবাদ
Command	কমান্ড / নির্দেশ
Invoker	আহ্বায়ক
Receiver	গ্রাহক / কার্যকরী অবজেক্ট
ConcreteCommand	নির্দিষ্ট কমান্ড
Client	ক্লায়েন্ট

✅ **Summary**:

The **Command Pattern** encapsulates a request as an object, separating the **invoker** (who asks) from the **receiver** (who does the work). It enables flexible, reusable, and extensible operations — perfect for smart home automation, GUI buttons, task scheduling, and undo/redo systems.

💡 Command Pattern in Java

🔷 Java-Style Code Example

```
// Command interface
interface Command {
    void execute();
}

// Receiver: the actual business logic
class Light {
    private boolean isOn = false;

    public void switchLight() {
        if (isOn) {
            System.out.println("Light turned OFF");
            isOn = false;
        } else {
            System.out.println("Light turned ON");
            isOn = true;
        }
    }
}

// Concrete Command
class SwitchLightCommand implements Command {
    private Light light;
```

```
    public SwitchLightCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.switchLight();
    }
}

// Invoker (Sender)
class Room {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        if (command != null) {
            command.execute();
        }
    }
}

// Client
public class SmartHomeApp {
    public static void main(String[] args) {
        Light livingRoomLight = new Light();

        // Create command object
        Command switchLight = new SwitchLightCommand(livingRoomLight);

        // Invoker
        Room livingRoom = new Room();
```

```
        livingRoom.setCommand(switchLight);

        // Simulate user pressing button
        livingRoom.pressButton(); // Light turned ON
        livingRoom.pressButton(); // Light turned OFF
    }
}
```

🔷 Specialty of the Command Pattern

- **Encapsulation of requests**: Turns a method call into an object.
- **Decoupling**: Separates the **Invoker** (who triggers) from the **Receiver** (who executes).
- **Flexibility**: Commands can be:
  - Stored in collections
  - Passed as parameters
  - Queued, scheduled, or logged
  - Replayed or undone

🔷 One Practical Use Case

Smart Home Automation (like in your example):

- Each action (switch lights, open curtains, lock doors) is represented as a **Command object**.
- The app can dynamically assign commands to buttons, schedule them, or even undo them.

👉 Another common real-world use: **GUI Buttons & Menus** in desktop apps. Each button click is mapped to a command, making it easy to add new actions without changing button code.

✅ **Summary**:

The **Command Pattern** is powerful when you want to **decouple request senders from receivers**, and when you need **flexibility in executing, storing, or undoing actions**. It shines in **smart home systems, GUI frameworks, task scheduling, and undo/redo features**.

🎮 Template Method Pattern – Study Note

🔷 Definition

- **Behavioral Design Pattern**
- Defines the **skeleton of an algorithm** in a superclass.
- Subclasses override specific steps **without changing the overall structure**.

🔷 Problem

- Large applications (like AAA games) have **complex loading sequences**:
  - Load local data (assets, code, images, videos).
  - Create objects (game entities, world objects).
  - Download additional files (patches, translations, sounds).
  - Clean temporary files.
  - Initialize user profiles.
- Different games share the **same steps**, but their **implementations vary**.
- Without a pattern → **code duplication** and **rigid structure**.

🔷 Solution

- Break down the algorithm into **steps (methods)**.
- Place them inside a **template method** that defines the order.
- Subclasses:

- Must implement **abstract steps**.
- May override **default implementations**.
- Cannot override the **template method** itself.

🔷 Structure (UML in Text Form)

```
AbstractClass (BaseGameLoader)
  + loadGame() [template method]
  + loadLocalData() [abstract]
  + createObjects() [abstract]
  + downloadAdditionalFiles() [abstract]
  + cleanTempFiles() [default]
  + initializeProfiles() [abstract]

ConcreteClass1 (WorldOfWarcraftLoader)
  - implements abstract steps

ConcreteClass2 (DiabloLoader)
  - implements abstract steps
```

🔷 Java-Style Code Example

```
// Abstract class (Template)
abstract class BaseGameLoader {

    // Template Method (defines the skeleton)
    public final void loadGame() {
        loadLocalData();
        createObjects();
        downloadAdditionalFiles();
        cleanTempFiles(); // default implementation
        initializeProfiles();
        System.out.println("Game loaded successfully!\n");
```

```
    }

    // Abstract steps (must be implemented by subclasses)
    protected abstract void loadLocalData();
    protected abstract void createObjects();
    protected abstract void downloadAdditionalFiles();
    protected abstract void initializeProfiles();

    // Default implementation (can be overridden if needed)
    protected void cleanTempFiles() {
        System.out.println("Cleaning temporary files...");
    }
}

// Concrete subclass 1
class WorldOfWarcraftLoader extends BaseGameLoader {
    @Override
    protected void loadLocalData() {
        System.out.println("Loading WoW assets from disk...");
    }

    @Override
    protected void createObjects() {
        System.out.println("Creating WoW characters and world objects...");
    }

    @Override
    protected void downloadAdditionalFiles() {
        System.out.println("Downloading WoW patches and expansions...");
    }

    @Override
    protected void initializeProfiles() {
        System.out.println("Loading WoW player profiles...");
    }
}
```

```
// Concrete subclass 2
class DiabloLoader extends BaseGameLoader {
    @Override
    protected void loadLocalData() {
        System.out.println("Loading Diablo assets from disk...");
    }

    @Override
    protected void createObjects() {
        System.out.println("Creating Diablo dungeons and monsters...");
    }

    @Override
    protected void downloadAdditionalFiles() {
        System.out.println("Downloading Diablo updates...");
    }

    @Override
    protected void initializeProfiles() {
        System.out.println("Loading Diablo player profiles...");
    }
}

// Client
public class GameLoaderDemo {
    public static void main(String[] args) {
        BaseGameLoader wow = new WorldOfWarcraftLoader();
        BaseGameLoader diablo = new DiabloLoader();

        wow.loadGame();
        diablo.loadGame();
    }
}
```

🔷 Output

```
Loading WoW assets from disk...
Creating WoW characters and world objects...
Downloading WoW patches and expansions...
Cleaning temporary files...
Loading WoW player profiles...
Game loaded successfully!

Loading Diablo assets from disk...
Creating Diablo dungeons and monsters...
Downloading Diablo updates...
Cleaning temporary files...
Loading Diablo player profiles...
Game loaded successfully!
```

🔷 Specialty

- Keeps **algorithm structure fixed** while allowing **customization of steps**.
- Eliminates **code duplication** by pulling common logic into the base class.
- Promotes **consistency** across different implementations.

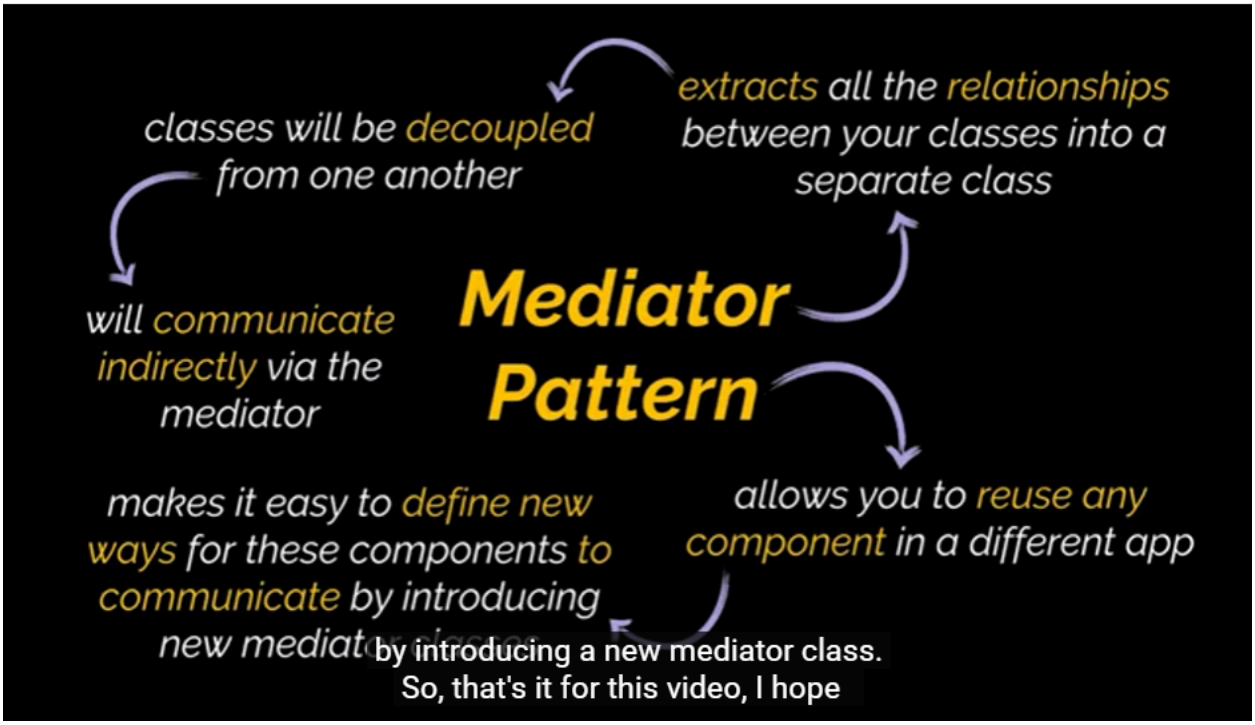
🔷 Use Case

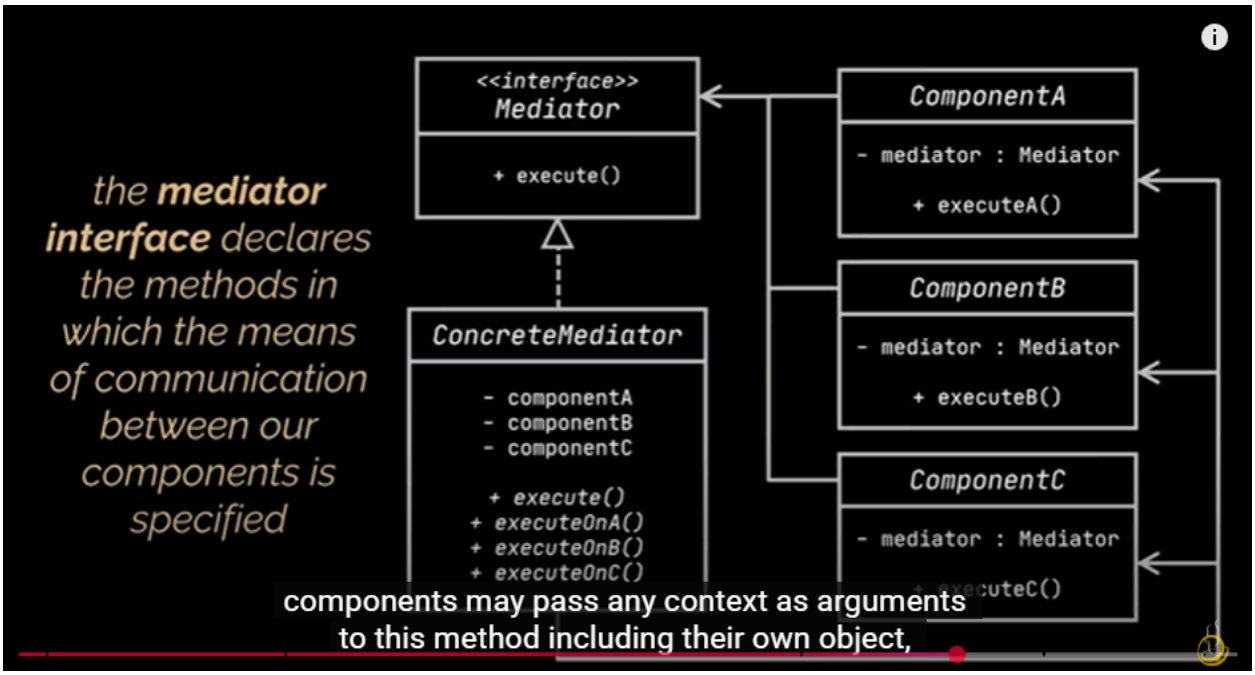
- **Game Loading Screens**: Different games share the same loading sequence but differ in details.
- Other examples:
  - Document parsing (common parsing flow, different file formats).
  - Data processing pipelines.
  - UI frameworks (common rendering flow, different widgets).

✅ **Summary**:

The **Template Method Pattern** is ideal when you want to **define a fixed workflow** but let subclasses **customize certain steps**. It balances **consistency** with **flexibility**.

🛫 Mediator Pattern – Study Note





🔷 Definition

- **Behavioral Design Pattern**
- Defines an object (**Mediator**) that encapsulates how a set of objects interact.
- Restricts direct communication between objects → they collaborate **only via the mediator**.
- Reduces **tight coupling** between components.

🔷 Real-World Analogy

- **Air Traffic Control**:
  - Planes (components) don’t talk to each other directly.
  - They communicate with the **controller (mediator)**.
  - The controller coordinates landings/departures.

🔷 Problem

- In a UI (e.g., login screen):

- Button fetches values from text fields directly.
- Button becomes **tightly coupled** to text fields.
- Hard to reuse components independently.

🔷 Solution

- Introduce a **Mediator (Dialog)**:
  - Components (Button, TextField, Label) only notify the mediator.
  - Mediator coordinates interactions and validations.
  - Components become **reusable and decoupled**.

🔷 Structure (UML in Text Form)

```
Component (Button, TextField, Label)
  - reference to Mediator
  - notify mediator on events

Mediator (interface)
  + notify(sender, event)

ConcreteMediator (LoginDialog)
  - holds references to components
  - coordinates interactions
```

🔷 Java-Style Code Example

```
// Mediator Interface
interface Mediator {
    void notify(Component sender, String event);
}

// Base Component
```

```
abstract class Component {
    protected Mediator mediator;

    public Component(Mediator mediator) {
        this.mediator = mediator;
    }
}

// Concrete Components
class TextField extends Component {
    private String text = "";

    public TextField(Mediator mediator) {
        super(mediator);
    }

    public void setText(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }
}

class Button extends Component {
    public Button(Mediator mediator) {
        super(mediator);
    }

    public void click() {
        System.out.println("Button clicked!");
        mediator.notify(this, "click");
    }
}
```

```
// Concrete Mediator
class LoginDialog implements Mediator {
    private TextField usernameField;
    private TextField passwordField;
    private Button loginButton;

    public LoginDialog() {
        this.usernameField = new TextField(this);
        this.passwordField = new TextField(this);
        this.loginButton = new Button(this);
    }

    public TextField getUsernameField() { return usernameField; }
    public TextField getPasswordField() { return passwordField; }
    public Button getLoginButton() { return loginButton; }

    @Override
    public void notify(Component sender, String event) {
        if (sender == loginButton && event.equals("click")) {
            if (usernameField.getText().equals("tanjim") &&
                passwordField.getText().equals("1234")) {
                System.out.println("Login successful!");
            } else {
                System.out.println("Invalid credentials!");
            }
        }
    }
}

// Client
public class MediatorDemo {
    public static void main(String[] args) {
        LoginDialog dialog = new LoginDialog();

        dialog.getUsernameField().setText("tanjim");
        dialog.getPasswordField().setText("1234");
```

```
        dialog.getLoginButton().click(); // Login successful!

        dialog.getPasswordField().setText("wrong");
        dialog.getLoginButton().click(); // Invalid credentials!
    }
}
```

🔷 Output

```
Button clicked!
Login successful!
Button clicked!
Invalid credentials!
```

🔷 Specialty

- Centralizes communication logic.
- Decouples components → easier to reuse.
- Makes it easy to change interaction rules by swapping mediator.

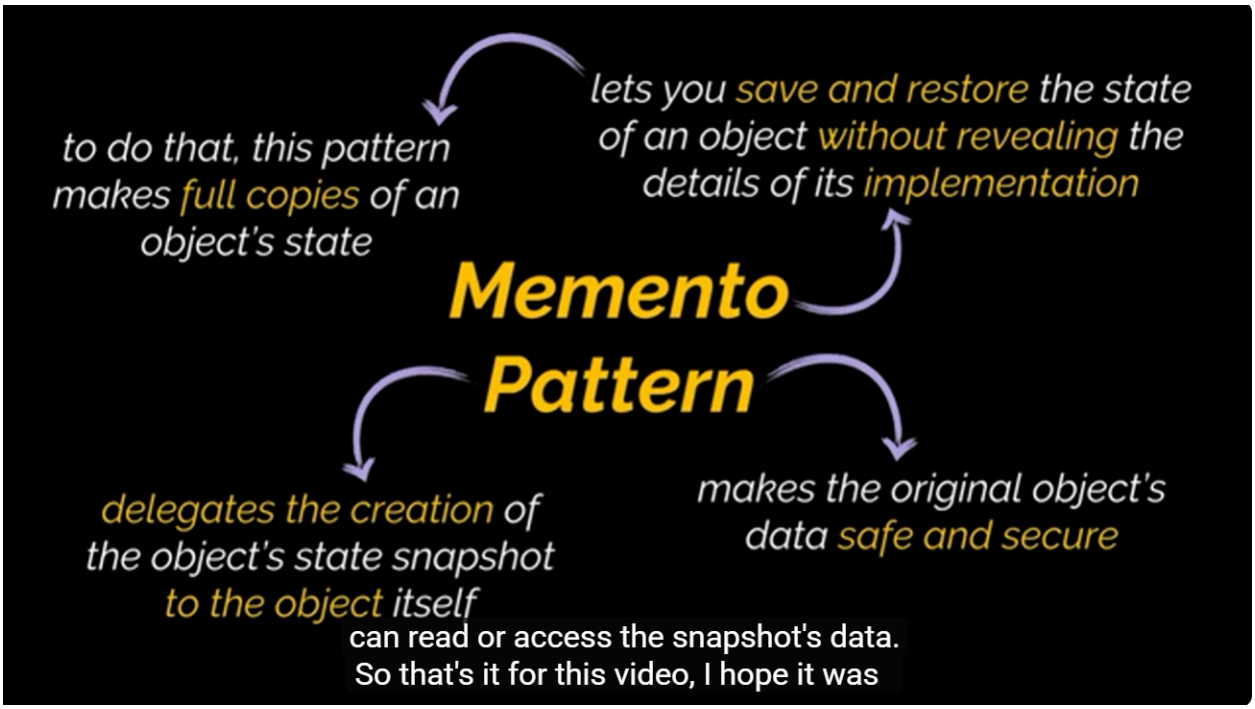
🔷 Use Case

- **UI frameworks** (buttons, text fields, dialogs).
- **Chat applications** (users communicate via a chatroom mediator).
- **Air traffic control systems**.

✅ **Summary**:

The **Mediator Pattern** extracts relationships between components into a **separate mediator class**, reducing dependencies and making components reusable. It's perfect for **UI coordination, communication systems, and workflow orchestration**.

📝 Memento Pattern – Study Note



🔷 Definition

- **Behavioral Design Pattern**
- Lets you **save and restore** the previous state of an object **without exposing its internal details**.
- Delegates snapshot creation to the **object itself** (the Originator).

🔷 Problem

- Undo/redo functionality requires saving **snapshots of state**.
- Naïve approach:
  - Exposes private fields.
  - Breaks encapsulation.
  - Makes other classes dependent on internal details.

🔷 Solution

- Use the **Memento Pattern**:
  - **Originator** → the object whose state we want to save (e.g., `TextArea`).
  - **Memento** → snapshot object storing the state (immutable, created by Originator).
  - **Caretaker** → manages history of mementos (e.g., `Editor`).

🔷 Structure (UML in Text Form)

```
Originator (TextArea)
  + createMemento()
  + restore(memento)

Memento (inner class)
  - state (private)
  + getter (accessible only by Originator)

Caretaker (Editor)
  - stores list/stack of Mementos
  - calls Originator to save/restore
```

🔷 Java-Style Code Example

```
// Originator
class TextArea {
    private String text;

    public void setText(String text) {
        this.text = text;
    }

    public String getText() {
```

```
        return text;
    }

    // Create snapshot
    public Memento takeSnapshot() {
        return new Memento(this.text);
    }

    // Restore snapshot
    public void restore(Memento memento) {
        this.text = memento.getSavedText();
    }

    // Inner Memento class
    public static class Memento {
        private final String savedText;

        private Memento(String text) {
            this.savedText = text;
        }

        private String getSavedText() {
            return savedText;
        }
    }
}

// Caretaker
class Editor {
    private java.util.Stack<TextArea.Memento> history = new java.util.Stack<>
();

    public void save(TextArea textArea) {
        history.push(textArea.takeSnapshot());
    }
```

```
    public void undo(TextArea textArea) {
        if (!history.isEmpty()) {
            textArea.restore(history.pop());
        }
    }
}

// Client
public class MementoDemo {
    public static void main(String[] args) {
        TextArea textArea = new TextArea();
        Editor editor = new Editor();

        textArea.setText("Hello");
        editor.save(textArea);

        textArea.setText("Hello World");
        editor.save(textArea);

        textArea.setText("Hello World!!!");
        System.out.println("Current: " + textArea.getText());

        // Undo
        editor.undo(textArea);
        System.out.println("After undo: " + textArea.getText());

        editor.undo(textArea);
        System.out.println("After second undo: " + textArea.getText());
    }
}
```

🔷 Output

```
Current: Hello World!!!
After undo: Hello World
After second undo: Hello
```

🔷 Specialty

- Preserves **encapsulation** → no external class can access private fields.
- Provides **undo/redo functionality**.
- Keeps **history of states** safely.

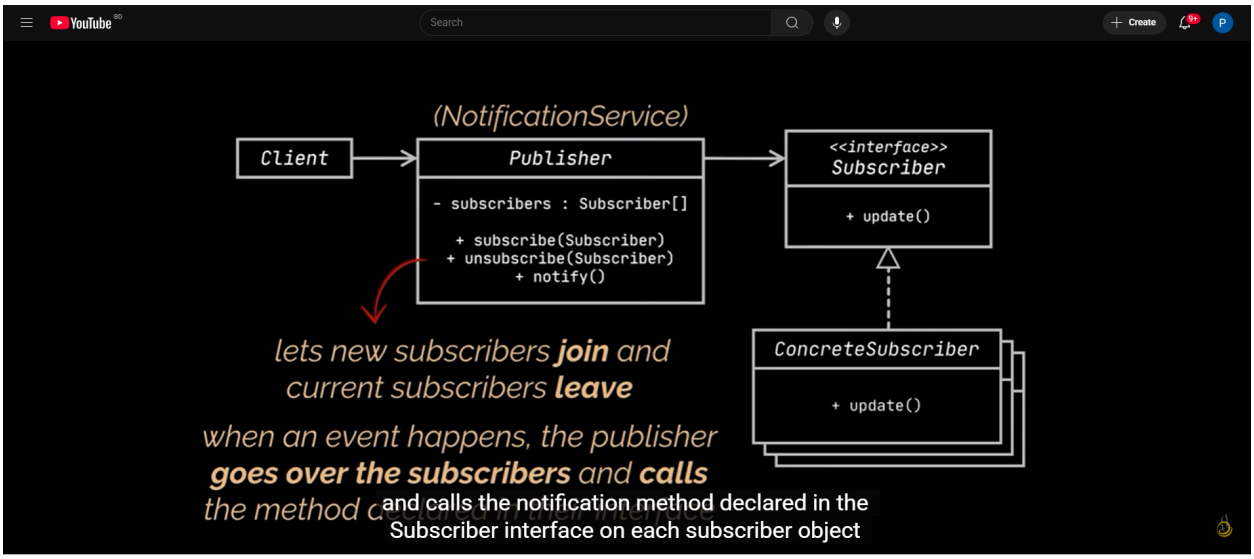
🔷 Use Case

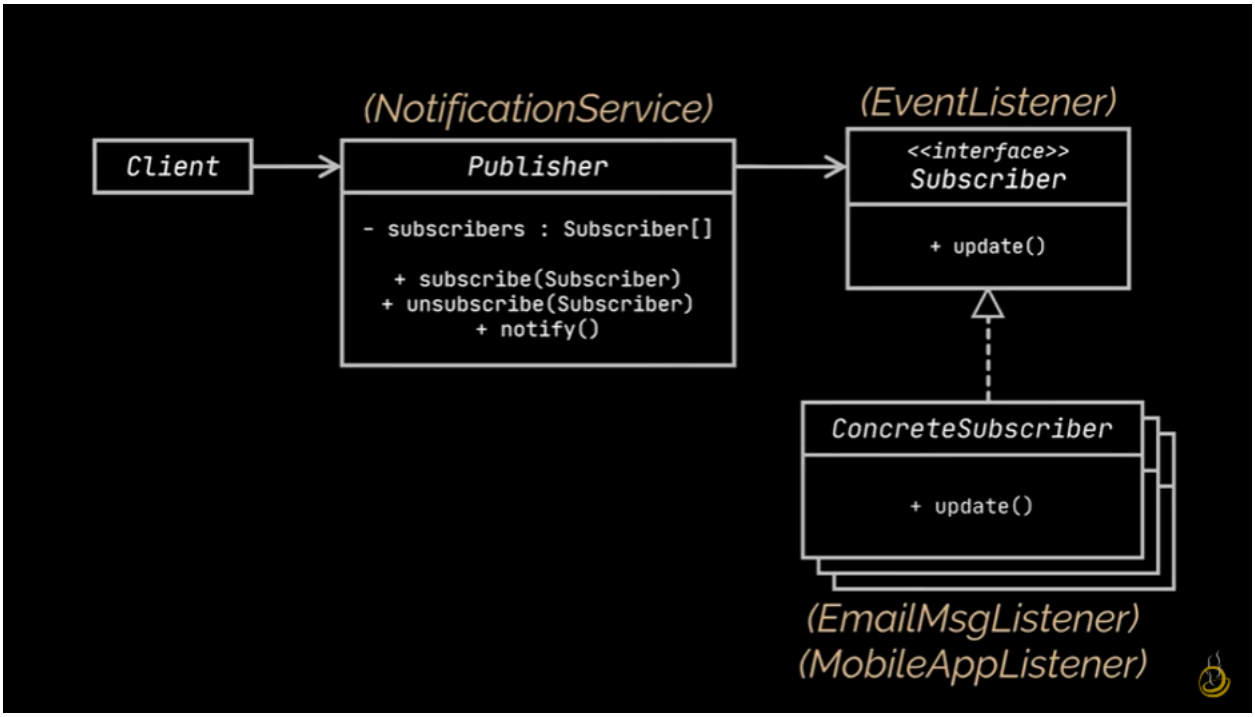
- **Text editors** (undo/redo).
- **Graphic editors** (restore previous canvas state).
- **Games** (save checkpoints).
- **Transactions** (rollback to previous state).

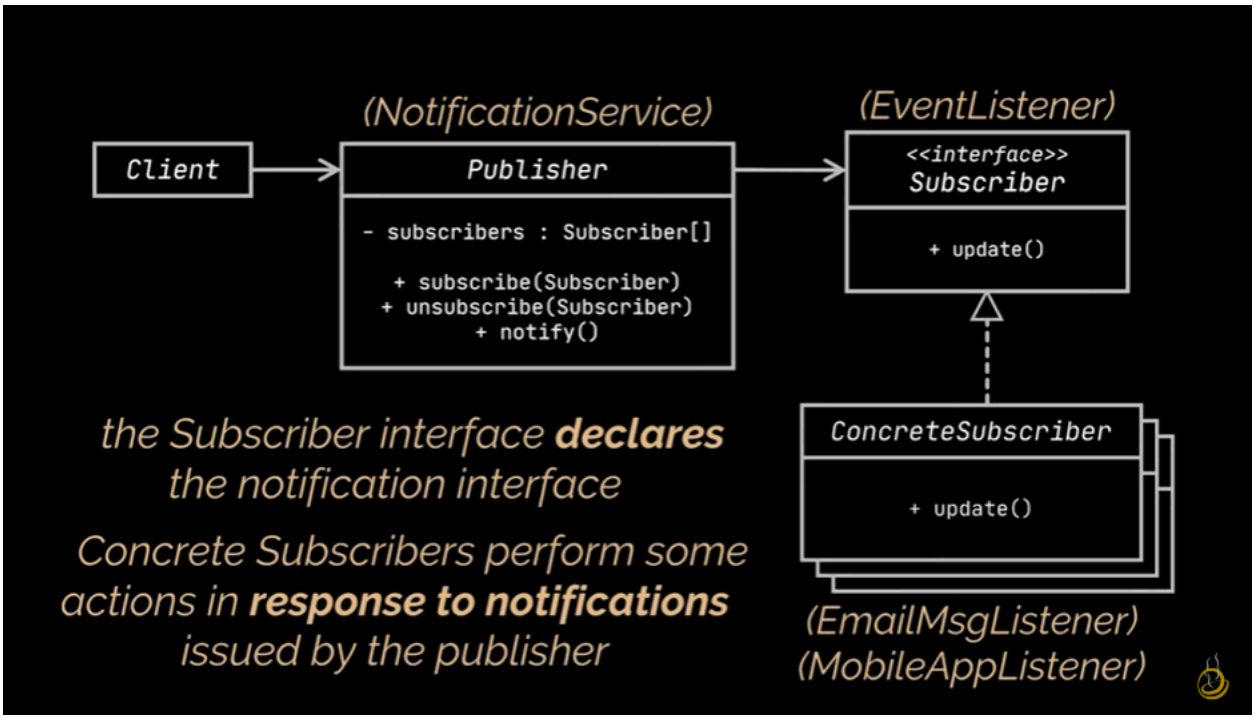
✅ **Summary**:

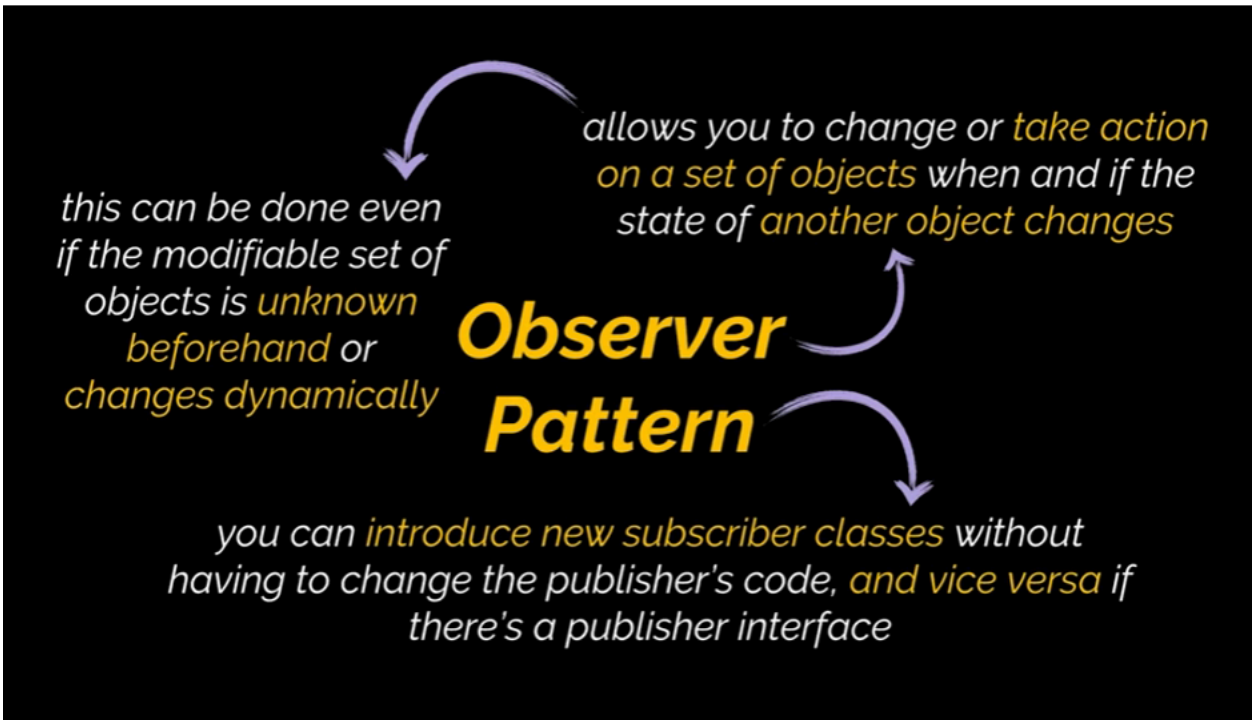
The **Memento Pattern** is perfect when you need to **save and restore object states** (like undo/redo) while keeping the object's internal details hidden. It uses **Originator, Memento, and Caretaker** to achieve this cleanly.

👀 Observer Pattern – Study Note


(NotificationService)
Client
Publisher
- subscribers : Subscriber[]
+ subscribe(Subscriber)
+ unsubscribe(Subscriber)
+ notify()
<<interface>>
Subscriber
+ update()
ConcreteSubscriber
+ update()
lets new subscribers join and current subscribers leave
when an event happens, the publisher goes over the subscribers and calls the method declared in the Subscriber interface on each subscriber object


(NotificationService)
(EventListener)
Client
Publisher
- subscribers : Subscriber[]
+ subscribe(Subscriber)
+ unsubscribe(Subscriber)
+ notify()
<<interface>>
Subscriber
+ update()
ConcreteSubscriber
+ update()
(EmailMsgListener)
(MobileAppListener)





🔷 Definition

- **Behavioral Design Pattern**
- Establishes a **one-to-many dependency** between objects.

- When the **Publisher (Subject)** changes state, all **Subscribers (Observers)** are notified automatically.

🔷 Real-World Analogy

- **Store & Customers**:
  - Customers don't want to check every day if a product is available.
  - Instead, they **subscribe** to notifications.
  - When a new item arrives, the store **notifies only interested customers**.

🔷 Structure (UML in Text Form)

```
Publisher (NotificationService)
  + subscribe(event, listener)
  + unsubscribe(event, listener)
  + notify(event)

Subscriber (EventListener)
  + update(event)

ConcreteSubscribers
  - EmailMessageListener
  - MobileAppListener

Client
  - creates publisher & subscribers
  - registers subscribers
```

🔷 Java-Style Code Example

```
import java.util.*;
```

```
// Event types
enum EventType {
    NEW_ITEM, SALE
}

// Subscriber interface
interface EventListener {
    void update(EventType eventType, String message);
}

// Concrete Subscriber 1
class EmailMessageListener implements EventListener {
    private String email;

    public EmailMessageListener(String email) {
        this.email = email;
    }

    @Override
    public void update(EventType eventType, String message) {
        System.out.println("Email to " + email + ": " + eventType + " - " + messag
e);
    }
}

// Concrete Subscriber 2
class MobileAppListener implements EventListener {
    private String username;

    public MobileAppListener(String username) {
        this.username = username;
    }

    @Override
    public void update(EventType eventType, String message) {
        System.out.println("Push notification to " + username + ": " + eventType
```

```
+ " - " + message);
    }
}

// Publisher
class NotificationService {
    private Map<EventType, List<EventListener>> listeners = new HashMap<>
();

    public NotificationService() {
        for (EventType event : EventType.values()) {
            listeners.put(event, new ArrayList<>());
        }
    }

    public void subscribe(EventType eventType, EventListener listener) {
        listeners.get(eventType).add(listener);
    }

    public void unsubscribe(EventType eventType, EventListener listener) {
        listeners.get(eventType).remove(listener);
    }

    public void notify(EventType eventType, String message) {
        for (EventListener listener : listeners.get(eventType)) {
            listener.update(eventType, message);
        }
    }
}

// Store (Client-facing Publisher)
class Store {
    private NotificationService notificationService = new NotificationService();

    public NotificationService getNotificationService() {
        return notificationService;
```

```
    }

    public void newItemPromotion(String item) {
        notificationService.notify(EventType.NEW_ITEM, "New item available: " +
item);
    }

    public void salePromotion(String saleDetails) {
        notificationService.notify(EventType.SALE, "Sale started: " + saleDetails);
    }
}

// Client
public class ObserverDemo {
    public static void main(String[] args) {
        Store store = new Store();

        EventListener emailListener = new EmailMessageListener("customer@ex
ample.com");
        EventListener appListener = new MobileAppListener("TanjimUser");

        store.getNotificationService().subscribe(EventType.NEW_ITEM, emailList
ener);
        store.getNotificationService().subscribe(EventType.SALE, appListener);

        store.newItemPromotion("Gaming Laptop");
        store.salePromotion("50% off on accessories!");
    }
}
```

🔷 Output

```
Email to customer@example.com: NEW_ITEM - New item available: Gaming L
aptop
```

```
Push notification to TanjimUser: SALE - Sale started: 50% off on accessories!
```

🔷 Specialty

- Decouples **Publisher** from **Subscribers**.
- Supports **dynamic subscription/unsubscription**.
- Follows **Open-Closed Principle** → add new subscriber types without modifying publisher.

🔷 Use Cases

- **UI frameworks** (event listeners for clicks, keypresses).
- **Messaging systems** (publish/subscribe).
- **Stock market apps** (notify traders of price changes).
- **E-commerce** (notify customers of new items, sales).

✅ **Summary**:

The **Observer Pattern** is ideal when multiple objects need to react to changes in another object's state. It enables **event-driven programming**, **loose coupling**, and **extensibility**.

Would you like me to also prepare a **comparison table of Observer vs Mediator Pattern**, since both deal with communication but in different ways?

Perfect — let's turn that into a **study note** for your SADP notebook, and I'll also add a **Java-style implementation** of the **State Pattern** so you can see it in action.

📱 State Pattern – Study Note

🔷 Definition

- **Behavioral Design Pattern**
- Lets an object **alter its behavior** when its internal state changes.

- To the client, it may appear as if the object has changed its class.

🔷 Real-World Analogy

- **Smartphone buttons**:
  - Power button → turns phone on if off, locks phone if unlocked.
  - Home button → unlocks phone if locked, goes to home screen if ready.
- Behavior depends on the **current state** of the phone.

🔷 Structure (UML in Text Form)

```
Context (Phone)
  - currentState: State
  + setState(State)
  + pressHome()
  + pressPower()

State (abstract/interface)
  + onHome()
  + onPower()

ConcreteStates
  - OffState
  - LockedState
  - ReadyState
```

🔷 Java-Style Code Example

```
// State interface
abstract class State {
    protected Phone phone;
```

```
    public State(Phone phone) {
        this.phone = phone;
    }

    public abstract void onHome();
    public abstract void onPower();
}

// Context
class Phone {
    private State state;

    public Phone() {
        state = new OffState(this); // initial state
    }

    public void setState(State state) {
        this.state = state;
    }

    public void pressHome() {
        state.onHome();
    }

    public void pressPower() {
        state.onPower();
    }

    // Example phone functionalities
    public void showHomeScreen() {
        System.out.println("Showing home screen...");
    }

    public void lockPhone() {
        System.out.println("Phone locked.");
    }
```

```
    public void turnOff() {
        System.out.println("Phone turned off.");
    }
}

// Concrete States
class OffState extends State {
    public OffState(Phone phone) { super(phone); }

    @Override
    public void onHome() {
        System.out.println("Turning phone ON...");
        phone.setState(new LockedState(phone));
    }

    @Override
    public void onPower() {
        System.out.println("Turning phone ON...");
        phone.setState(new LockedState(phone));
    }
}

class LockedState extends State {
    public LockedState(Phone phone) { super(phone); }

    @Override
    public void onHome() {
        System.out.println("Unlocking phone...");
        phone.setState(new ReadyState(phone));
    }

    @Override
    public void onPower() {
        phone.turnOff();
        phone.setState(new OffState(phone));
```

```
    }
}

class ReadyState extends State {
    public ReadyState(Phone phone) { super(phone); }

    @Override
    public void onHome() {
        phone.showHomeScreen();
    }

    @Override
    public void onPower() {
        phone.turnOff();
        phone.setState(new OffState(phone));
    }
}

// Client
public class StatePatternDemo {
    public static void main(String[] args) {
        Phone phone = new Phone();

        phone.pressPower(); // Turn ON → Locked
        phone.pressHome();  // Unlock → Ready
        phone.pressHome();  // Show home screen
        phone.pressPower(); // Turn OFF
    }
}
```

🔷 Output

```
Turning phone ON...
Unlocking phone...
```

```
Showing home screen...
Phone turned off.
```

🔷 Difference from Strategy Pattern

- **State Pattern**:
  - Behavior depends on **current state**.
  - States may be **aware of each other** and trigger transitions.
  - Example: Phone transitions between Off → Locked → Ready.
- **Strategy Pattern**:
  - Different interchangeable algorithms that achieve the **same goal**.
  - Strategies are **independent** and unaware of each other.
  - Example: Different sorting algorithms (QuickSort, MergeSort).

🔷 Benefits

- ✅ Removes long `if-else` or `switch` statements.
- ✅ Encapsulates state-specific behavior in separate classes.
- ✅ Follows **Single Responsibility Principle** (each state has its own class).
- ✅ Follows **Open-Closed Principle** (add new states without modifying existing ones).

✅ **Summary**:

The **State Pattern** is ideal when an object's behavior depends on its **internal state**. It replaces conditional logic with **composition + polymorphism**, making code cleaner, extensible, and easier to maintain.

💳 Strategy Pattern – Study Note

🔷 Definition

- **Behavioral Design Pattern**
- Lets you define a **family of algorithms**, encapsulate each in a **separate class**, and make them **interchangeable** at runtime.
- The **Context** delegates the algorithm to a **Strategy** object.

🔷 Real-World Analogy

- **Food Delivery App**:
  - You can pay via **Credit Card**, **PayPal**, or other methods.
  - Each payment method has its own logic.
  - Instead of hardcoding `if-else` or `switch`, use **Strategy Pattern** to plug in the desired payment method.

🔷 Structure (UML in Text Form)

```
Strategy (PaymentStrategy)
  + collectPaymentDetails()
  + validatePaymentDetails()
  + pay(amount)

ConcreteStrategies
  - PaymentByCreditCard
  - PaymentByPayPal

Context (PaymentService)
  - strategy: PaymentStrategy
  + setStrategy(PaymentStrategy)
  + processOrder(amount)

Client
```

```
- creates strategy
- injects into context
```

🔷 Java-Style Code Example

```
// Strategy Interface
interface PaymentStrategy {
    void collectPaymentDetails();
    boolean validatePaymentDetails();
    void pay(double amount);
}

// Concrete Strategy 1
class PaymentByCreditCard implements PaymentStrategy {
    private String cardNumber;
    private String cvv;

    @Override
    public void collectPaymentDetails() {
        System.out.println("Collecting credit card details...");
        cardNumber = "1234-5678-9012-3456";
        cvv = "123";
    }

    @Override
    public boolean validatePaymentDetails() {
        System.out.println("Validating credit card...");
        return cardNumber != null && cvv != null;
    }

    @Override
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " using Credit Card.");
    }
```

```
}

// Concrete Strategy 2
class PaymentByPayPal implements PaymentStrategy {
    private String email;
    private String password;

    @Override
    public void collectPaymentDetails() {
        System.out.println("Collecting PayPal credentials...");
        email = "user@example.com";
        password = "securepass";
    }

    @Override
    public boolean validatePaymentDetails() {
        System.out.println("Validating PayPal account...");
        return email != null && password != null;
    }

    @Override
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " using PayPal.");
    }
}

// Context
class PaymentService {
    private PaymentStrategy strategy;

    public void setStrategy(PaymentStrategy strategy) {
        this.strategy = strategy;
    }

    public void processOrder(double amount) {
        strategy.collectPaymentDetails();
```

```
        if (strategy.validatePaymentDetails()) {
            strategy.pay(amount);
        } else {
            System.out.println("Payment validation failed.");
        }
    }
}

// Client
public class StrategyPatternDemo {
    public static void main(String[] args) {
        PaymentService service = new PaymentService();

        // Pay with Credit Card
        service.setStrategy(new PaymentByCreditCard());
        service.processOrder(49.99);

        // Switch to PayPal
        service.setStrategy(new PaymentByPayPal());
        service.processOrder(29.99);
    }
}
```

🔷 Output

```
Collecting credit card details...
Validating credit card...
Paid $49.99 using Credit Card.
Collecting PayPal credentials...
Validating PayPal account...
Paid $29.99 using PayPal.
```

🔷 Strategy vs State Pattern

Feature	Strategy Pattern	State Pattern
Purpose	Choose among interchangeable algorithms	Change behavior based on internal state
Awareness	Strategies are unaware of each other	States may know and transition to others
Result Consistency	Same result regardless of strategy	Result varies based on state
Example	Payment methods, sorting algorithms	Phone states, game levels

🔷 Benefits

- ✅ Clean separation of concerns.
- ✅ Easily add new strategies without modifying existing code.
- ✅ Replace strategies at runtime.
- ✅ Follows **Single Responsibility** and **Open-Closed Principles**.

✅ **Summary**:

The **Strategy Pattern** is perfect when you need to **swap algorithms or behaviors** dynamically. It keeps your code **flexible, extensible, and clean**, especially in cases like payment processing, sorting, or formatting.

🔁 Iterator Pattern – Study Note

🔷 Definition

- **Behavioral Design Pattern**
- Provides a way to **traverse elements of a collection** without exposing its internal structure.
- Encapsulates traversal logic in a separate **Iterator object**.

🔷 Real-World Analogy

- **Tour Guide in Paris**:
  - You don't need to know the city's layout.
  - The guide (iterator) takes you to each site (element) in a specific order (DFS, BFS).
  - Multiple guides can lead different tours independently.

🔷 Structure (UML in Text Form)

```
Iterator (GraphIterator)
  + next()
  + hasNext()

Concrete Iterators
  - DepthFirstIterator
  - BreadthFirstIterator

Collection Interface (Graph)
  + getIterator()

Concrete Collections
  - BinarySearchTree
  - RedBlackTree

Client
  - uses Graph and GraphIterator via interfaces
```

🔷 Java-Style Code Example

```
import java.util.*;

// Vertex class
```

```
class Vertex {
    String label;
    List<Vertex> neighbors = new ArrayList<>();

    public Vertex(String label) {
        this.label = label;
    }

    public void addNeighbor(Vertex v) {
        neighbors.add(v);
    }
}

// Iterator Interface
interface GraphIterator {
    Vertex next();
    boolean hasNext();
}

// Depth First Iterator
class DepthFirstIterator implements GraphIterator {
    private Set<Vertex> visited = new HashSet<>();
    private Stack<Vertex> stack = new Stack<>();

    public DepthFirstIterator(Vertex start) {
        stack.push(start);
    }

    @Override
    public boolean hasNext() {
        return !stack.isEmpty();
    }

    @Override
    public Vertex next() {
        while (!stack.isEmpty()) {
```

```
            Vertex current = stack.pop();
            if (!visited.contains(current)) {
                visited.add(current);
                for (Vertex neighbor : current.neighbors) {
                    stack.push(neighbor);
                }
                return current;
            }
        }
        return null;
    }
}

// Breadth First Iterator
class BreadthFirstIterator implements GraphIterator {
    private Set<Vertex> visited = new HashSet<>();
    private Queue<Vertex> queue = new LinkedList<>();

    public BreadthFirstIterator(Vertex start) {
        queue.add(start);
    }

    @Override
    public boolean hasNext() {
        return !queue.isEmpty();
    }

    @Override
    public Vertex next() {
        while (!queue.isEmpty()) {
            Vertex current = queue.poll();
            if (!visited.contains(current)) {
                visited.add(current);
                queue.addAll(current.neighbors);
                return current;
            }
```

```
        }
        return null;
    }
}

// Client
public class IteratorPatternDemo {
    public static void main(String[] args) {
        // Create graph
        Vertex a = new Vertex("A");
        Vertex b = new Vertex("B");
        Vertex c = new Vertex("C");
        Vertex d = new Vertex("D");

        a.addNeighbor(b);
        a.addNeighbor(c);
        b.addNeighbor(d);
        c.addNeighbor(d);

        // Choose iterator
        GraphIterator iterator = new DepthFirstIterator(a);

        System.out.println("DFS Traversal:");
        while (iterator.hasNext()) {
            Vertex v = iterator.next();
            System.out.println(v.label);
        }
    }
}
```

🔷 Output

```
DFS Traversal:
A
```

```
C
D
B
```

🔷 Benefits

- ✅ Traversal logic is **decoupled** from data structure.
- ✅ Multiple iterators can traverse the same collection independently.
- ✅ Follows **Single Responsibility Principle** (each traversal in its own class).
- ✅ Follows **Open-Closed Principle** (add new iterators without modifying existing code).

🔷 Use Cases

- Graph traversal (DFS, BFS).
- Tree traversal (in-order, pre-order, post-order).
- Custom iteration over complex data structures.
- UI components, file systems, paginated APIs.

✅ **Summary**:

The **Iterator Pattern** is perfect when you want to **traverse complex collections** without exposing their internal structure. It enables **flexible, reusable, and extensible traversal logic**, especially useful in graphs, trees, and custom data structures.

📦 Composite Pattern – Study Note

🔷 Definition

- **Structural Design Pattern**

- Lets you **compose objects into tree structures** and treat **individual and composite objects uniformly**.
- Ideal for **hierarchical models** like file systems, UI components, or nested containers.

🔷 Real-World Analogy

- **Amazon Delivery System**:
  - A **Box** can contain **Products** or **other Boxes**.
  - To calculate total price, you must traverse all nested items.
  - The Composite Pattern lets you treat **Products and Boxes the same way** via a common interface.

🔷 Structure (UML in Text Form)

```
Component (BoxItem)
  + getPrice()

Leaf (Product)
  - Book
  - VideoGame

Composite (CompositeBox)
  - List<BoxItem> children
  + getPrice()

Client
  - works with BoxItem interface
```

🔷 Java-Style Code Example

```
import java.util.*;

// Component Interface
interface BoxItem {
    double getPrice();
}

// Leaf: Product
class Book implements BoxItem {
    private double price;

    public Book(double price) {
        this.price = price;
    }

    @Override
    public double getPrice() {
        return price;
    }
}

class VideoGame implements BoxItem {
    private double price;

    public VideoGame(double price) {
        this.price = price;
    }

    @Override
    public double getPrice() {
        return price;
    }
}

// Composite: Box
```

```
class CompositeBox implements BoxItem {
    private List<BoxItem> items = new ArrayList<>();
    private double packagingCost;

    public CompositeBox(double packagingCost) {
        this.packagingCost = packagingCost;
    }

    public void addItem(BoxItem item) {
        items.add(item);
    }

    @Override
    public double getPrice() {
        double total = packagingCost;
        for (BoxItem item : items) {
            total += item.getPrice();
        }
        return total;
    }
}

// Client
public class CompositePatternDemo {
    public static void main(String[] args) {
        // Leaf items
        Book book1 = new Book(15.99);
        VideoGame game1 = new VideoGame(59.99);

        // Inner box
        CompositeBox smallBox = new CompositeBox(2.50);
        smallBox.addItem(book1);
        smallBox.addItem(game1);

        // Outer box
        CompositeBox bigBox = new CompositeBox(5.00);
```

```
        bigBox.addItem(smallBox);
        bigBox.addItem(new Book(25.00));
        bigBox.addItem(new VideoGame(39.99));

        // Total price
        System.out.println("Total delivery price: $" + bigBox.getPrice());
    }
}
```

🔷 Output

```
Total delivery price: $148.47
```

```
import java.util.ArrayList;
import java.util.List;

/*
 * ❓ Problem Description:
 * You are modeling a company's organizational structure.
 * There are two types of components:
 *   1. Employees (leaf nodes) – have name and salary.
 *   2. Departments (composite nodes) – have name and contain employees or sub-departments.
 *
 * ✅ Requirements:
 * - Define a common interface for both types with:
 *     double getSalary(); // returns total salary
 *     void display();     // prints details
 * - Employee: returns own salary, displays own info.
 * - Department: returns sum of all salaries (including sub-departments), displays all components.
 * - Department should support add/remove components.
 */
```

```
// 🔧 Common interface for both Employee and Department
interface OrganizationComponent {
    double getSalary();     // total salary of this component
    void display();         // display details of this component
}

// 🍃 Leaf node: Individual Employee
class Employee implements OrganizationComponent {
    private String name;
    private double salary;

    public Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    // Returns own salary
    @Override
    public double getSalary() {
        return salary;
    }

    // Displays employee details
    @Override
    public void display() {
        System.out.println("Employee: " + name + " - $" + salary);
    }
}

// 🌳 Composite node: Department
class Department implements OrganizationComponent {
    private String name;
    private List<OrganizationComponent> components = new ArrayList<>();

    public Department(String name) {
```

```
        this.name = name;
    }

    // Add a component (employee or sub-department)
    public void addComponent(OrganizationComponent component) {
        components.add(component);
    }

    // Remove a component
    public void removeComponent(OrganizationComponent component) {
        components.remove(component);
    }

    // Returns total salary of all components
    @Override
    public double getSalary() {
        double total = 0;
        for (OrganizationComponent component : components) {
            total += component.getSalary();
        }
        return total;
    }

    // Displays department name and all components
    @Override
    public void display() {
        System.out.println("Department: " + name);
        for (OrganizationComponent component : components) {
            component.display();
        }
    }
}

// 🚀 Main class to test the structure
public class Main {
    public static void main(String[] args) {
```

```
        // Create employees
        Employee john = new Employee("John Doe", 50000.0);
        Employee jane = new Employee("Jane Smith", 60000.0);
        Employee bob = new Employee("Bob Johnson", 40000.0);

        // Create QA department and add Bob
        Department qa = new Department("QA");
        qa.addComponent(bob);

        // Create Engineering department and add John, Jane, and QA
        Department engineering = new Department("Engineering");
        engineering.addComponent(john);
        engineering.addComponent(jane);
        engineering.addComponent(qa);

        // Display structure and total salary
        engineering.display();
        System.out.println("Total Engineering Department Salary: $" + engineerin
g.getSalary());
    }
}
```

🔷 Benefits

- ✅ Treat **individual and composite objects uniformly**.
- ✅ Supports **recursive structures** like trees.
- ✅ Follows **Open-Closed Principle** → add new item types without modifying existing code.
- ✅ Follows **Single Responsibility Principle** → each class handles its own logic.

🔷 Use Cases

- File systems (folders and files).
- UI components (panels and widgets).

- Organization charts (employees and managers).
- Delivery systems (boxes and products).

✅ **Summary**:

The **Composite Pattern** is perfect for **tree-like structures** where you want to treat **simple and complex elements the same way**. It enables clean, extensible, and recursive handling of nested objects.

🏦 Facade Pattern – Study Note

🔷 Definition

- **Structural Design Pattern**
- Provides a **simplified interface** to a complex subsystem (library, framework, or set of classes).
- Helps **decouple client code** from internal implementation details.

🔷 Real-World Analogy

- **Crypto Investing App**:
  - You want to buy cryptocurrency using a third-party library.
  - The library has many classes and methods: balance check, transaction, email service, etc.
  - Instead of exposing all of that, you create a **BuyCryptoFacade** that wraps the complexity.

🔷 Structure (UML in Text Form)

```
Facade (BuyCryptoFacade)
  + buyCrypto(userId, amount, currency)

Subsystem Classes
```

```
  - UserService
  - BalanceService
  - CryptoService
  - MailService

Client
  - calls BuyCryptoFacade only
```

🔷 Java-Style Code Example

```
// Subsystem classes (simulated)
class UserService {
    public User getUserById(String userId) {
        return new User(userId, 1000.0, "user@example.com");
    }
}

class BalanceService {
    public boolean hasSufficientBalance(User user, double amount) {
        return user.getBalance() >= amount;
    }
}

class CryptoService {
    public void buyCurrency(User user, double amount, String currency) {
        System.out.println("Buying " + amount + " of " + currency + " for " + use
r.getId());
    }
}

class MailService {
    public void sendConfirmation(User user, String currency, double amount) {
        System.out.println("Email sent to " + user.getEmail() + ": Purchased " + a
mount + " of " + currency);
```

```
    }
}

// User class
class User {
    private String id;
    private double balance;
    private String email;

    public User(String id, double balance, String email) {
        this.id = id;
        this.balance = balance;
        this.email = email;
    }

    public String getId() { return id; }
    public double getBalance() { return balance; }
    public String getEmail() { return email; }
}

// Facade
class BuyCryptoFacade {
    private UserService userService = new UserService();
    private BalanceService balanceService = new BalanceService();
    private CryptoService cryptoService = new CryptoService();
    private MailService mailService = new MailService();

    public void buyCrypto(String userId, double amount, String currency) {
        User user = userService.getUserById(userId);
        if (!balanceService.hasSufficientBalance(user, amount)) {
            System.out.println("Insufficient balance for user " + userId);
            return;
        }
        cryptoService.buyCurrency(user, amount, currency);
        mailService.sendConfirmation(user, currency, amount);
    }
```

```
}

// Client
public class FacadePatternDemo {
    public static void main(String[] args) {
        BuyCryptoFacade facade = new BuyCryptoFacade();
        facade.buyCrypto("user123", 500.0, "Bitcoin");
    }
}
```

🔷 Output

```
Buying 500.0 of Bitcoin for user123
Email sent to user@example.com: Purchased 500.0 of Bitcoin
```

🔷 Benefits

- ✅ Simplifies complex subsystems.
- ✅ Improves readability and usability.
- ✅ Reduces coupling between client and subsystem.
- ✅ Follows **Single Responsibility Principle**.
- ✅ Follows **Open-Closed Principle** → add new facades without modifying existing ones.

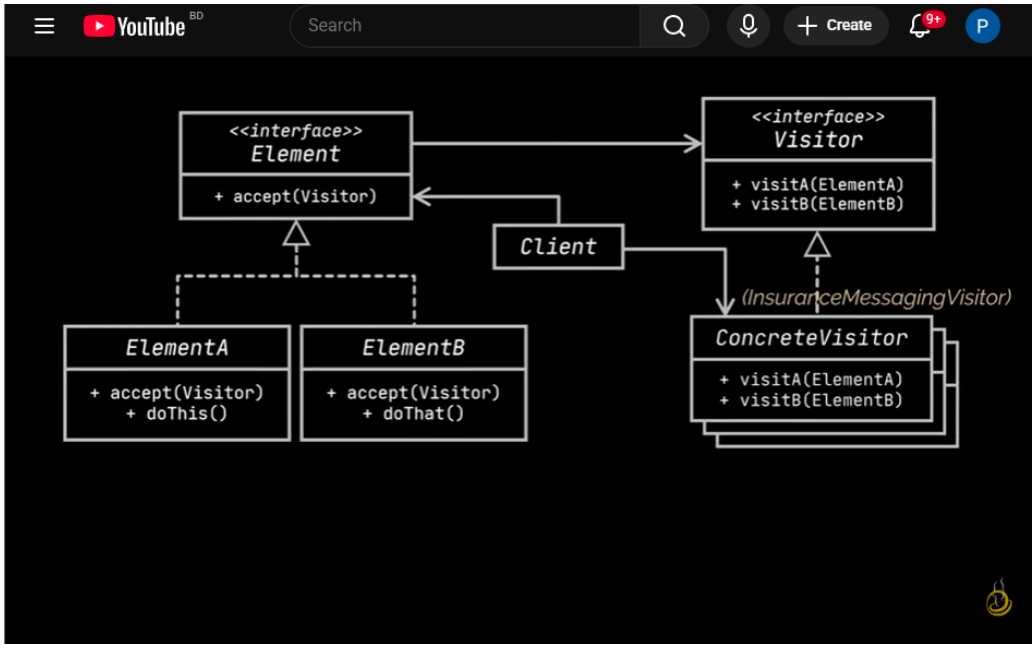
🔷 Use Cases

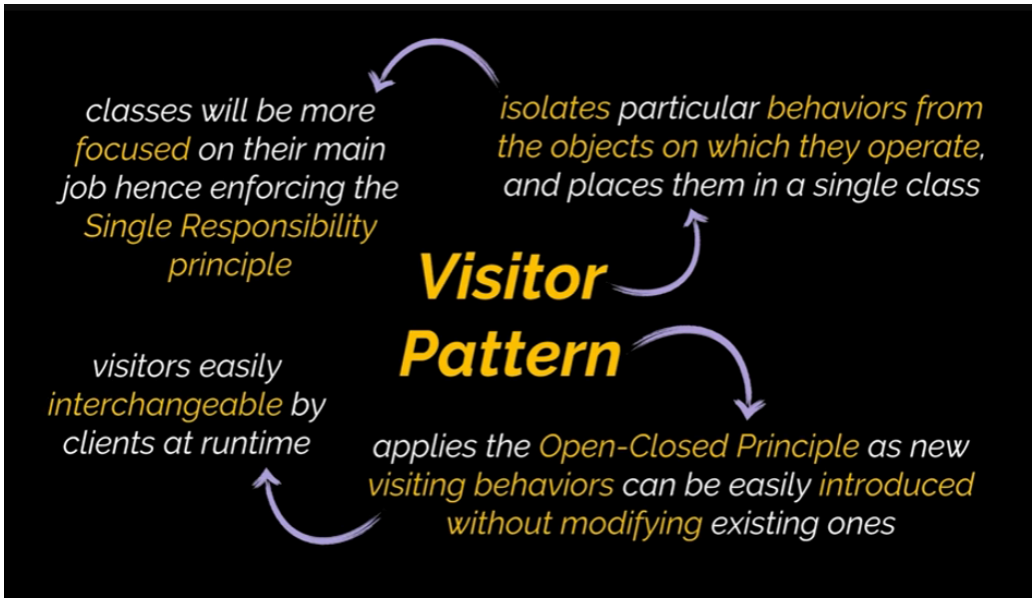
- Wrapping third-party libraries.
- Simplifying access to complex APIs.
- Creating entry points for layered architectures.
- Managing workflows across multiple services.

✅ **Summary**:

The **Facade Pattern** is perfect when you want to **hide complexity** and expose a **clean, unified interface**. It's especially useful in apps that integrate with external libraries or frameworks.

📬 Visitor Pattern – Study Note





🔷 Definition

- **Behavioral Design Pattern**
- Lets you **separate behaviors** from the objects they operate on.
- Uses **double-dispatch** to execute behavior based on both the visitor and the element type.

🔷 Real-World Analogy

- **Insurance Messaging System**:
  - Clients: Resident, Bank, etc.
  - Each client receives a different ad message.
  - Instead of embedding logic in each client class, we extract it into a **Visitor**.
  - Clients accept the visitor and delegate the correct behavior.

🔷 Structure (UML in Text Form)

```
Element Interface (Client)
  + accept(Visitor)

Concrete Elements
  - ResidentClient
  - BankClient

Visitor Interface
  + visit(ResidentClient)
  + visit(BankClient)

Concrete Visitor
  - InsuranceMessagingVisitor

Client
```

```
  - loops over elements
  - calls accept(visitor)
```

🔷 Java-Style Code Example

```
// Visitor Interface
interface Visitor {
    void visit(ResidentClient client);
    void visit(BankClient client);
}

// Concrete Visitor
class InsuranceMessagingVisitor implements Visitor {
    @Override
    public void visit(ResidentClient client) {
        System.out.println("Sending medical insurance ad to resident: " + client.g
etName());
    }

    @Override
    public void visit(BankClient client) {
        System.out.println("Sending theft insurance ad to bank: " + client.getNam
e());
    }
}

// Element Interface
abstract class Client {
    protected String name;

    public Client(String name) {
        this.name = name;
    }
```

```
    public String getName() {
        return name;
    }

    public abstract void accept(Visitor visitor);
}

// Concrete Elements
class ResidentClient extends Client {
    public ResidentClient(String name) {
        super(name);
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

class BankClient extends Client {
    public BankClient(String name) {
        super(name);
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

// Client Code
public class VisitorPatternDemo {
    public static void main(String[] args) {
        List<Client> clients = List.of(
            new ResidentClient("Alice"),
            new BankClient("Bank of Sylhet"),
```

```
            new ResidentClient("Tanjim")
        );

        Visitor messagingVisitor = new InsuranceMessagingVisitor();

        for (Client client : clients) {
            client.accept(messagingVisitor);
        }
    }
}
```

🔷 Output

```
Sending medical insurance ad to resident: Alice
Sending theft insurance ad to bank: Bank of Sylhet
Sending medical insurance ad to resident: Tanjim
```

🔷 Benefits

- ✅ Keeps **data and behavior separate**.
- ✅ Supports **double-dispatch**: behavior depends on both visitor and element type.
- ✅ Follows **Single Responsibility Principle** → client classes stay focused.
- ✅ Follows **Open-Closed Principle** → add new visitors without modifying existing classes.

🔷 Use Cases

- Insurance messaging systems.
- Syntax tree traversal (e.g., compilers).
- UI rendering engines.

- File system operations (e.g., backup, indexing).

✅ **Summary**:

The **Visitor Pattern** is ideal when you want to **add new behaviors** to a set of classes **without modifying them**. It uses **double-dispatch** to cleanly delegate behavior and keeps your code **modular, extensible, and maintainable**.

🔌 Adapter Pattern – Study Note

🔷 Definition

- **Structural Design Pattern**
- Allows objects with **incompatible interfaces** to work together.
- Acts as a **bridge** between the client and a third-party or legacy system.

🔷 Real-World Analogy

- **Power Plug Adapter**:
  - American plug doesn't fit in a European socket.
  - An adapter converts the shape and voltage.
  - Similarly, your app uses **XML**, but the new UI library expects **JSON**.

🔷 Structure (UML in Text Form)

```
Client Interface (IMultiRestoApp)
  + displayMenus(XMLData)

Client (MultiRestoApp)
  - uses IMultiRestoApp
```

```
Service (FancyUIService)
  + renderUI(JSONData)

Adapter (FancyUIServiceAdapter)
  - implements IMultiRestoApp
  - wraps FancyUIService
  - converts XML → JSON
```

🔷 Java-Style Code Example

```
// XML and JSON data types (simplified)
class XMLData {
    public String getXML() {
        return "<menu><item>Pizza</item></menu>";
    }
}

class JSONData {
    private String json;

    public JSONData(String json) {
        this.json = json;
    }

    public String getJSON() {
        return json;
    }
}

// Client Interface
interface IMultiRestoApp {
    void displayMenus(XMLData xmlData);
}
```

```
// Existing App Implementation
class MultiRestoApp implements IMultiRestoApp {
    @Override
    public void displayMenus(XMLData xmlData) {
        System.out.println("Displaying menus using XML: " + xmlData.getXML());
    }
}

// Third-party UI Library
class FancyUIService {
    public void renderUI(JSONData jsonData) {
        System.out.println("Rendering fancy UI with JSON: " + jsonData.getJSON
());
    }
}

// Adapter
class FancyUIServiceAdapter implements IMultiRestoApp {
    private FancyUIService fancyUI;

    public FancyUIServiceAdapter(FancyUIService fancyUI) {
        this.fancyUI = fancyUI;
    }

    @Override
    public void displayMenus(XMLData xmlData) {
        // Convert XML to JSON (simulated)
        String converted = "{\"menu\": [{\"item\": \"Pizza\"}]}";
        JSONData jsonData = new JSONData(converted);
        fancyUI.renderUI(jsonData);
    }
}

// Client
public class AdapterPatternDemo {
    public static void main(String[] args) {
```

```
        XMLData xmlData = new XMLData();

        // Original app
        IMultiRestoApp originalApp = new MultiRestoApp();
        originalApp.displayMenus(xmlData);

        // Using adapter for Fancy UI
        IMultiRestoApp fancyApp = new FancyUIServiceAdapter(new FancyUISer
vice());
        fancyApp.displayMenus(xmlData);
    }
}
```

🔷 Output

```
Displaying menus using XML: <menu><item>Pizza</item></menu>
Rendering fancy UI with JSON: {"menu": [{"item": "Pizza"}]}
```

🔷 Benefits

- ✅ Enables **reuse of legacy or third-party code**.
- ✅ Keeps client code **clean and decoupled**.
- ✅ Follows **Single Responsibility Principle** (conversion logic is isolated).
- ✅ Follows **Open-Closed Principle** (new adapters can be added without modifying existing code).

🔷 Use Cases

- Integrating incompatible APIs.
- Bridging old and new systems.
- Wrapping third-party libraries.

- Adapting data formats (e.g., XML ↔ JSON).

✅ **Summary**:

The **Adapter Pattern** is perfect when you need to **connect incompatible interfaces**. It wraps a third-party or legacy class and **translates client requests** into something the service understands — without changing either side.

🍕 Bridge Pattern – Study Note

🔷 Definition

- **Structural Design Pattern**
- Decouples an abstraction from its implementation so that both can evolve independently.
- Ideal when a class has **multiple dimensions of variation**.

🔷 Real-World Analogy

- **Pizza Delivery System**:
  - Pizza types: Pepperoni, Veggie, Chicken...
  - Preparation styles: American, Italian...
  - Instead of creating every combination as a subclass (e.g., `AmericanPepperoniPizza`),
    use **composition** to link `Pizza` and `Restaurant` hierarchies.

🔷 Structure (UML in Text Form)

```
Abstraction (Restaurant)
  + deliver()

Refined Abstractions
  - AmericanRestaurant
```

```
  - ItalianRestaurant

Implementation Interface (Pizza)
  + prepare()

Concrete Implementations
  - PepperoniPizza
  - VeggiePizza

Client
  - links Restaurant with Pizza
  - calls deliver()
```

🔷 Java-Style Code Example

```
// Implementation Interface
interface Pizza {
    String prepare();
}

// Concrete Implementations
class PepperoniPizza implements Pizza {
    @Override
    public String prepare() {
        return "Pepperoni Pizza";
    }
}

class VeggiePizza implements Pizza {
    @Override
    public String prepare() {
        return "Veggie Pizza";
    }
}
```

```
// Abstraction
abstract class Restaurant {
    protected Pizza pizza;

    public Restaurant(Pizza pizza) {
        this.pizza = pizza;
    }

    public abstract void deliver();
}

// Refined Abstractions
class AmericanRestaurant extends Restaurant {
    public AmericanRestaurant(Pizza pizza) {
        super(pizza);
    }

    @Override
    public void deliver() {
        System.out.println("Delivering " + pizza.prepare() + " in American styl
e.");
    }
}

class ItalianRestaurant extends Restaurant {
    public ItalianRestaurant(Pizza pizza) {
        super(pizza);
    }

    @Override
    public void deliver() {
        System.out.println("Delivering " + pizza.prepare() + " in Italian style.");
    }
}
```

```
// Client
public class BridgePatternDemo {
    public static void main(String[] args) {
        Pizza pepperoni = new PepperoniPizza();
        Pizza veggie = new VeggiePizza();

        Restaurant american = new AmericanRestaurant(pepperoni);
        Restaurant italian = new ItalianRestaurant(veggie);

        american.deliver(); // Delivering Pepperoni Pizza in American style.
        italian.deliver();  // Delivering Veggie Pizza in Italian style.
    }
}
```

🔷 Output

```
Delivering Pepperoni Pizza in American style.
Delivering Veggie Pizza in Italian style.
```

🔷 Benefits

- ✅ Avoids **class explosion** from multiple inheritance dimensions.
- ✅ Enables **runtime flexibility**: switch pizza or restaurant independently.
- ✅ Follows **Single Responsibility Principle** → each hierarchy handles its own concern.
- ✅ Follows **Open-Closed Principle** → add new pizzas or restaurants without modifying existing code.

🔷 Use Cases

- UI toolkits (e.g., different platforms and rendering engines).
- Messaging systems (e.g., message types and delivery channels).

- Game engines (e.g., character types and control schemes).
- Payment systems (e.g., payment methods and gateways).

✅ **Summary**:

The **Bridge Pattern** is perfect when you need to **decouple two dimensions of variation**. It keeps your code **modular, scalable, and flexible**, especially when combinations grow exponentially.

📣 Decorator Pattern – Study Note

🔷 Definition

- **Structural Design Pattern**
- Lets you **dynamically add new behaviors** to objects by wrapping them in decorator classes.
- Promotes **composition over inheritance**.

🔷 Real-World Analogy

- **Food Delivery Notifications**:
  - Base notifier sends **email**.
  - Customers want **WhatsApp**, **Facebook**, or **SMS** too.
  - Instead of creating every combination as a subclass, use **decorators** to wrap and extend behavior.

🔷 Structure (UML in Text Form)

```
Component Interface (INotifier)
  + send(String message)

Concrete Component
  - Notifier (sends email)
```

```
Base Decorator
  - BaseNotifierDecorator (wraps INotifier)

Concrete Decorators
  - WhatsAppDecorator
  - FacebookDecorator

Client
  - wraps Notifier with decorators
  - calls send()
```

🔷 Java-Style Code Example

```
// Component Interface
interface INotifier {
    void send(String message);
}

// Concrete Component
class Notifier implements INotifier {
    @Override
    public void send(String message) {
        System.out.println("Email sent: " + message);
    }
}

// Base Decorator
abstract class BaseNotifierDecorator implements INotifier {
    protected INotifier wrapped;

    public BaseNotifierDecorator(INotifier wrapped) {
        this.wrapped = wrapped;
    }
```

```
    @Override
    public void send(String message) {
        wrapped.send(message);
    }
}

// Concrete Decorators
class WhatsAppDecorator extends BaseNotifierDecorator {
    public WhatsAppDecorator(INotifier wrapped) {
        super(wrapped);
    }

    @Override
    public void send(String message) {
        super.send(message);
        System.out.println("WhatsApp message sent: " + message);
    }
}

class FacebookDecorator extends BaseNotifierDecorator {
    public FacebookDecorator(INotifier wrapped) {
        super(wrapped);
    }

    @Override
    public void send(String message) {
        super.send(message);
        System.out.println("Facebook message sent: " + message);
    }
}

// Client
public class DecoratorPatternDemo {
    public static void main(String[] args) {
        INotifier baseNotifier = new Notifier();
```

```
        INotifier whatsappNotifier = new WhatsAppDecorator(baseNotifier);
        INotifier facebookNotifier = new FacebookDecorator(whatsappNotifier);

        facebookNotifier.send("Your order has been dispatched!");
    }
}
```

🔷 Output

```
Email sent: Your order has been dispatched!
WhatsApp message sent: Your order has been dispatched!
Facebook message sent: Your order has been dispatched!
```

🔷 Benefits

- ✅ Adds behavior **without modifying existing classes**.
- ✅ Supports **runtime flexibility**: wrap objects dynamically.
- ✅ Follows **Single Responsibility Principle** → each decorator handles one concern.
- ✅ Follows **Open-Closed Principle** → add new decorators without changing existing code.

🔷 Use Cases

- UI components (e.g., scrollbars, borders, shadows).
- Logging, caching, or security wrappers.
- Notification systems (email, SMS, push).
- Data transformation pipelines.

✅ **Summary**:

The **Decorator Pattern** is perfect when you want to **extend object behavior at runtime** without altering its structure. It enables **layered enhancements**, keeps code **modular**, and supports **flexible composition**.

🛡️ Proxy Pattern – Study Note

🔷 Definition

- **Structural Design Pattern**
- Provides a **substitute or placeholder** for another object.
- Controls access to the original object, allowing pre/post-processing.

🔷 Real-World Analogy

- **Proxy Server**:
  - Intercepts requests between user and website.
  - Adds security, filtering, caching, and privacy.
  - Similarly, a proxy class can intercept method calls and add logic before/after delegating to the real object.

🔷 Structure (UML in Text Form)

```
Service Interface
  + connectTo(String site)
  + download(String videoName)

Real Service
  - RealInternet
  - RealVideoDownloader

Proxy
  - ProxyInternet
```

```
  - ProxyVideoDownloader

Client
  - uses Service Interface
```

🔷 Java-Style Code Example

🌐 Internet Access Proxy

```
// Service Interface
interface Internet {
    void connectTo(String site);
}

// Real Service
class RealInternet implements Internet {
    @Override
    public void connectTo(String site) {
        System.out.println("Connecting to " + site);
    }
}

// Proxy
class ProxyInternet implements Internet {
    private RealInternet realInternet = new RealInternet();
    private static final Set<String> bannedSites = Set.of("banned.com", "blocke
d.org");

    @Override
    public void connectTo(String site) {
        if (bannedSites.contains(site)) {
            System.out.println("Access Denied to " + site);
        } else {
            realInternet.connectTo(site);
```

```
        }
    }
}
```

📹 Video Downloader Proxy

```
// Service Interface
interface VideoDownloader {
    Video download(String videoName);
}

// Video class
class Video {
    private String name;
    private String metadata;

    public Video(String name, String metadata) {
        this.name = name;
        this.metadata = metadata;
    }

    public String getInfo() {
        return "Video: " + name + ", Metadata: " + metadata;
    }
}

// Real Service
class RealVideoDownloader implements VideoDownloader {
    @Override
    public Video download(String videoName) {
        System.out.println("Downloading " + videoName + " from YouTube...");
        return new Video(videoName, "HD, 1080p, 5min");
    }
}
```

```
// Proxy
class ProxyVideoDownloader implements VideoDownloader {
    private RealVideoDownloader realDownloader = new RealVideoDownloader
();
    private Map<String, Video> cache = new HashMap<>();

    @Override
    public Video download(String videoName) {
        if (cache.containsKey(videoName)) {
            System.out.println("Returning cached video: " + videoName);
            return cache.get(videoName);
        }
        Video video = realDownloader.download(videoName);
        cache.put(videoName, video);
        return video;
    }
}
```

👨‍💻 Client Code

```
public class ProxyPatternDemo {
    public static void main(String[] args) {
        // Internet Proxy
        Internet internet = new ProxyInternet();
        internet.connectTo("example.com");
        internet.connectTo("banned.com");

        // Video Proxy
        VideoDownloader downloader = new ProxyVideoDownloader();
        downloader.download("PizzaRecipe");
        downloader.download("PizzaRecipe"); // Cached
    }
}
```

🔷 Output

```
Connecting to example.com
Access Denied to banned.com
Downloading PizzaRecipe from YouTube...
Returning cached video: PizzaRecipe
```

🔷 Benefits

- ✅ Adds **security, caching, logging, or access control**.
- ✅ Keeps client code **clean and decoupled**.
- ✅ Follows **Open-Closed Principle** → add new proxies without modifying service or client.
- ✅ Can **delay or manage lifecycle** of the real object.

🔷 Use Cases

- Access control (firewalls, filters).
- Lazy initialization or resource management.
- Caching and performance optimization.
- Remote proxies (e.g., RPC, web services).
- Logging and monitoring wrappers.

✅ **Summary**:

The **Proxy Pattern** is perfect when you need to **intercept and manage access** to another object. It enables **controlled delegation**, supports **runtime flexibility**, and keeps your architecture **clean and extensible**.

Would you like me to also prepare a **comparison table of Proxy vs Decorator vs Adapter**, since they all wrap objects but serve different purposes?

Absolutely, Tanjim! Here's a comprehensive table that maps **design patterns** to their **typical use cases** in software development. This will help you quickly

identify which pattern suits which kind of task — from UI rendering to database access, caching, messaging, and more.

🧠 Design Patterns & Their Use Cases

Pattern	Category	Typical Use Case
Singleton	Creational	Managing shared resources like **database connections**, **logging**, **config**
Factory Method	Creational	Creating objects without specifying exact class — e.g., **UI components**, **parsers**
Abstract Factory	Creational	Creating families of related objects — e.g., **cross-platform UI kits**, **themes**
Builder	Creational	Constructing complex objects step-by-step — e.g., **HTML/XML documents**, **reports**
Prototype	Creational	Cloning objects — e.g., **game entities**, **document templates**, **deep copies**
Adapter	Structural	Bridging incompatible interfaces — e.g., **XML ↔ JSON converters**, **legacy APIs**
Bridge	Structural	Decoupling abstraction from implementation — e.g., **UI rendering**, **device drivers**
Composite	Structural	Tree-like structures — e.g., **file systems**, **menus**, **organization charts**
Decorator	Structural	Adding behavior dynamically — e.g., **notification layers**, **UI styling**, **logging**
Facade	Structural	Simplifying complex subsystems — e.g., **library wrappers**, **API gateways**
Flyweight	Structural	Memory optimization — e.g., **text rendering**, **game tiles**, **book metadata**
Proxy	Structural	Access control, caching — e.g., **firewalls**, **lazy loading**, **video streaming**
Chain of Responsibility	Behavioral	Event handling pipelines — e.g., **middleware**, **form validation**, **logging chains**

Pattern	Category	Typical Use Case
Command	Behavioral	Encapsulating actions — e.g., **undo/redo**, **task queues**, **remote control**
Interpreter	Behavioral	Parsing and evaluating expressions — e.g., **math engines**, **DSLs**, **regex**
Iterator	Behavioral	Traversing collections — e.g., **graph traversal**, **custom data structures**
Mediator	Behavioral	Centralizing communication — e.g., **UI components**, **chat rooms**, **air traffic control**
Memento	Behavioral	State saving/restoration — e.g., **undo/redo**, **checkpoints**, **draft recovery**
Observer	Behavioral	Event subscription — e.g., **UI listeners**, **notification systems**, **pub/sub**
State	Behavioral	State-dependent behavior — e.g., **UI modes**, **workflow engines**, **game states**
Strategy	Behavioral	Swappable algorithms — e.g., **sorting**, **payment methods**, **compression**
Template Method	Behavioral	Defining skeleton of an algorithm — e.g., **report generation**, **data import/export**
Visitor	Behavioral	Externalizing operations — e.g., **syntax trees**, **insurance messaging**, **file scanning**

✅ **Quick Examples:**

- **Singleton** → `DatabaseConnection.getInstance()`
- **Observer** → `onClickListener` in Android
- **Strategy** → `PaymentService.setStrategy(new PayPalStrategy())`
- **Decorator** → `new FacebookDecorator(new WhatsAppDecorator(new Notifier()))`
- **Proxy** → `new ProxyInternet().connectTo("banned.com")`

Would you like me to generate a **cheat sheet version** of this table with icons and color-coded categories for your study notes? Or maybe a **PDF-style layout** for printing and revision?