**Diamond Problem**

Java does not allow a class to inherit more than one class (multiple inheritance). This is known as the diamond problem.

It is an ambiguity that can rise as a result of multiple inheritance.

Problem for OOP languages.

Also known as the deadly diamond of death.

**Solution for the diamond problem:**

Default methods & interfaces.

The advantage of interfaces is that it can have the same default methods with the same name and signature in two different interfaces. It allows us to implement these two interfaces, from a class. We must override the default methods explicitly with its interface name.

Example:

InterfaceName.**super**.methodName();

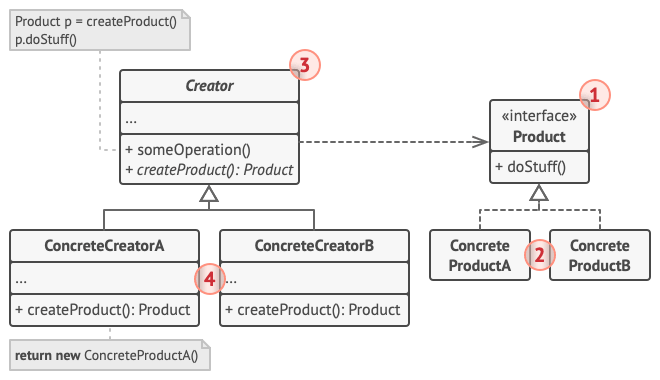
**Creational Patterns**

**Objective:** To provide various object creation mechanisms to increase flexibility and decrease reuse of existing code.

1. Factory Method Pattern: Imagine having a transportation app based on trucks. But later you want to add different types of vehicles like ships. Code gets messy. How to handle it?

Solution:

Replace direct object construction(new Truck()) with a factory method. From now on get the direct objects from the factory. For no confusion later add an interface (Vehicle) that can act as the base class.



1 → Product: interface for all

2 → ConcreateProduct: Different implementations of product.

3 → Creator: Return new product objects. Can be abstract to force all classes to implement in their own ways. (Does Not have to create new instances it can return existing objects from a cache, pool or another source)

4 → Concrete Creators: Override Base factory to return a different type of Product.

When to use:

1. When you don't know the exact dependencies of the objects.
2. When you want to provide users of your library to extend its internal components.
3. When you want to reuse existing objects instead of rebuilding them. (to save resources)

Pros:

1. Avoid tight coupling
2. Single Responsibility principle: Move product creation code into one place.
3. Open closed principle: Can introduce new types of product.

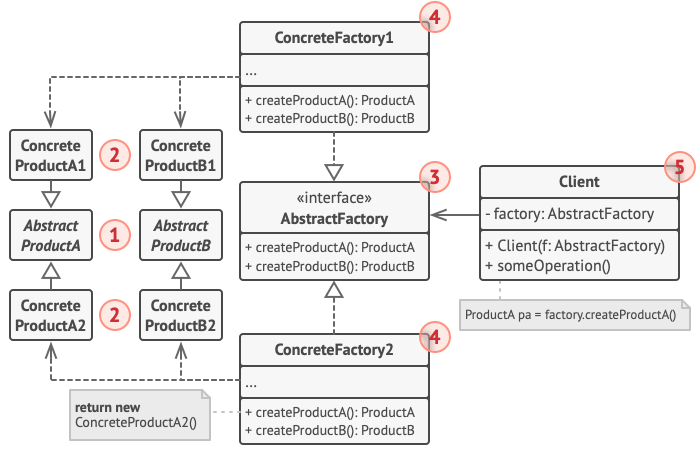
Cons:

1. Code may become complicated if you need to introduce new products.
2. Abstract Factory Pattern

Objective: Lets you produce families of related objects without specifying their concrete classes.

Problem: You produce chairs and tables. You produce them in modern and victorian style. You need a way to produce chairs so when customer orders chairs, they dont get some in modern style and some in victorian.

Solution: Have chair and table interface. İmplement them in modern and victorian style. Have an abstract factory that is implemented by different factories to produce in victorian and modern.



1 → Abstract Products: Interfaces for a set of distinct but related products. (Chair, Table)

2 → Concrete Products: Implementations of abstract products. All products must be implemented in all variants. (VictorianChair, ModernChair,...)

3 → Abstract Factory: Interface that declares a set of creation methods for each of the abstract products. (FurnitureFactory)

4 → Concrete Factories: Implement creation methods of the abstract factory. Each concrete factory corresponds to a specific variant of products. (VictorianFactory, ModernFactory) – creation method should return abstract product.

5 → The Client can work with any concrete factory/product variant, as long as it communicates with their objects via abstract interfaces.

When to use:

1. When your code needs to work with various families of related products.
2. When you have a set of Factory Methods.

Pros:

1. Products you are getting from a factory are compatible.
2. Avoid tight coupling
3. Single responsibility principle
4. Open/closed principle

Cons:

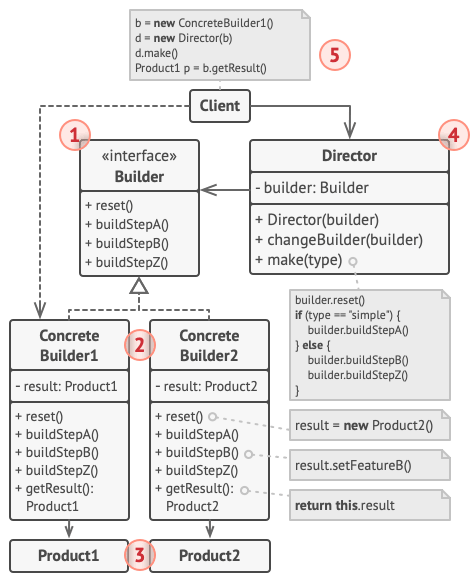
1. Code may get complicated with added classes and interfaces.
2. Builder Pattern

Objective: Helps you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

Problem: You want to build a house. With 4 walls, a door and son on. But what if you want to build a bigger house or a house with a pool. One solution might be to extend the house class and move on like that but this can get complicated quickly. Another solution might be to have a big constructor and non used parts could be fed as null but it makes the constructor ugly.

Solution: Extract the object construction code out of its own class and move it to separate objects called builders.

Organize object construction into a set of steps (buildWalls, buildDoor).



1 → Builder: Interface declares product construction steps that are common to all types of builders.

2 → Concrete Builders: Provide different implementations of the construction steps. May produce products that don't follow a common interface.

3 → Products: Resulting object.

4 → Director: Defines which order to call the construction steps.

5 → Client: Have a builder. Give it to directors constructor. Build and get the result from the builder.

When to use:

1. When you want to get rid of telescoping constructor. (No long constructors or multiple constructors)
2. When you want the code to be able to create different representations of some product (wooden house, stone house)
3. Construct composite trees or other complex objects → A builder does not expose the unfinished product while running construction steps. This prevents the client code from fetching an incomplete result.

Pros:

1. Run steps recursively.
2. Reuse the same construction for various representations.
3. Single Responsibility Principle.

Cons:

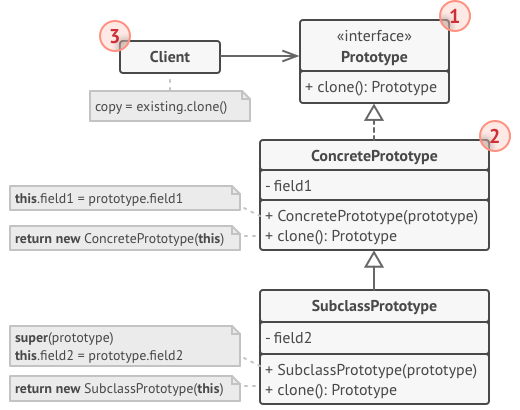
1. Overall complexity of the code increases.
2. Prototype Pattern

Objective: Lets you copy existing objects without making your code dependent on their classes.

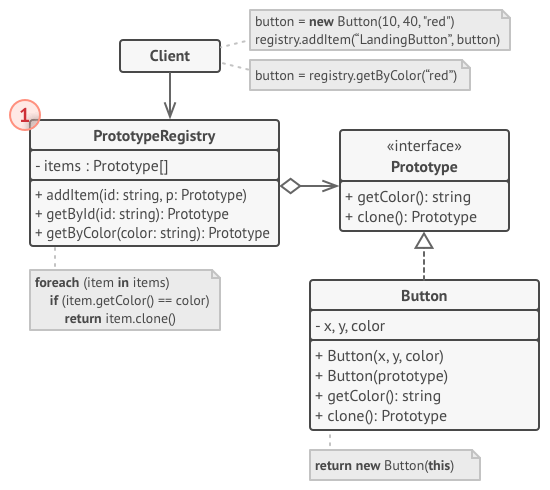
Problem: You want to copy an object exactly. You need to go over all the fields and set them individually. Another problem is you have to know the class you are duplicating so you depend on the class. Another catch is you may only have the interface and not the concrete class which would be a problem.

Solution: An object that supports cloning is called a *prototype*. In these classes you have a “clone method” that clones the object.

Basic implementation:



Prototype registry implementation??



1 → Prototype Registry: It stores a set of pre-built objects that are ready to be copied. Simplest : name → prototype hash map.

When to use:

1. When your code shouldnt depend on the concrete classes of objects that you need to copy.
2. When you want to reduce the number of subclasses that only differ in the way they initialize their respective objects.

Pros:

1. Clone without coupling to their concrete classes.
2. Get rid of repeated initialization code in favor of cloning pre-built prototypes.
3. Produce complex- objects more conveniently.

Cons:

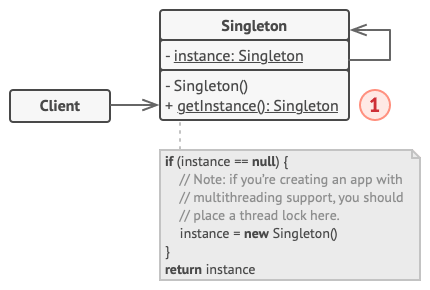
1. Cloning complex objects that have circular references might be very tricky.
2. Singleton Pattern

Objective: To ensure that a class has only one instance while providing a global access point.

Problem: Single Responsibility Principle. → Ensure that a class hast just one instance. & Provide a global access point to that instance.

Solution: 2 Steps:

1. Make a default cınctructor private to prevent other objects to create an instance of it.
2. Create a static creation method that acts as a constructor. This method calls the private constructor to create an instance and saves it in a static field.



When to use:

1. When a class needs to have only one instance. (Single database object)
2. When you need stricter control over global variables.

Pros:

1. You can be sure that class has only a single instance.
2. You gain global access point to that instance.
3. Class initialized only once when it is requested for the first time.

Cons:

1. Violates single responsibility principle.
2. Requires special treatment for multithreaded environment.
3. May be diffucult to write unit tests. To mock objects it relies on constructors but in this case it is private.

Less Used:

1. Pool
2. Simple Factory
3. Static Factory