Design patterns

Creational patterns:

1.Singleton

Solves 2 problems at the same time

1.ensures that the class has 1 instance, the most common reason for this is to control the access to a shared resource (example: a database)

Here's how it works: imagine that you created an object, but after a while decided to create a new one. Instead of receiving a fresh object, you'll get the one you already created.

2.Provide a global access point to that instance

Solution

1 make the default constructor private to prevent other objects from using the new operator with the Singleton class

2 Create a static creation method that acts as a constructor. Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object. If your code has access to the Singleton class, then it's able to call the Singleton's static method. So, whenever that method is called, the same object is always returned.

How to implement:

```
public class Singleton
{
    private static Singleton instance;

    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            {
                instance = new Singleton();
            }
            return instance;
        }
    }
}

// Usage:
Singleton singleton = Singleton.Instance;
```

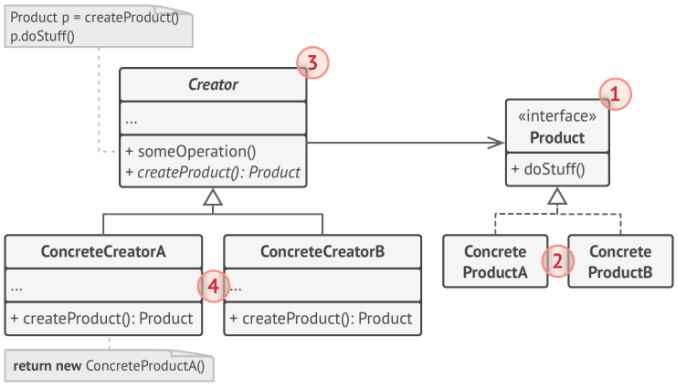
1.Add a private static field to the class for storing the singleton instance.
2. Declare a public static creation method for getting the singleton instance.
3. Implement “lazy initialization” inside the static method. It should create a new object on its first call and put it into the static field. The method should always return that instance on all subsequent calls.
4. Make the constructor of the class private. The static method of the class will still be able to call the constructor, but not the other objects
5. Go over the client code and replace all direct calls to the singleton’s constructor with calls to its static creation method.

2.Factory

Imagine you create a logistics app that initially handles only trucks, with most of the code centered in the Truck class. As the app gains popularity, you get requests to add sea transport. However, because the code is tightly coupled to Truck, adding a Ship class would require extensive codebase changes. Future additions of other transport types would also need similar updates, resulting in a tangled codebase full of conditionals switching behaviors by transport type.

Solution

The Factory Method pattern suggests that you replace direct object construction calls (using the new operator) with calls to a special factory method. Don’t worry: the objects are still created via the new operator, but it’s being called from within the factory method. Objects returned by a factory method are often referred to as “products.”, the point of doing this is that now you can override the factory method in a subclass and make it return a different object There’s a slight limitation though: subclasses may return different types of products only if these products have a common base class or interface(ships and trucks have to implement the same interface in order to make road and sea logistics classes create and return them in the create product method). Also, the factory method in the base class should have its return type declared as this interface.



creator:logistics
concretecreatorA:road logistics
concretecreatorB:sea logistics
concreteproductA:truck
concreteproductB:ship
creatproduct()in class concretecreatorA
returns a new truck
createproduct in class concretecreatorB
returns a new ship

1. The **Product** declares the interface, which is common to all objects that can be produced by the creator and its subclasses.

2. **Concrete Products** are different implementations of the product interface.

3. The **Creator class** declares the factory method that returns new product objects. It's important that the return type of this method matches the product interface.

4. **Concrete Creators** override the base factory method so it returns a different type of product.

1. First, you need to create some storage to keep track of all of the created objects.

2. When someone requests an object, the program should look for a free object inside that pool.

3. ... and then return it to the client code.

4. If there are no free objects, the program should create a new one (and add it to the pool).

How to implement

1.Make all products follow the same interface. This interface should declare methods that make sense in every product.

2. Add an empty factory method inside the creator class. The return type of the method should match the common product interface.

3. In the creator's code find all references to product constructors. One by one, replace them with calls to the factory method, while extracting the product creation code into the factory method. You might need to add a temporary parameter to the factory method to control the type of returned product.

4. Now, create a set of creator subclasses for each type of product listed in the factory method. Override the factory method in the subclasses and extract the appropriate bits of construction code from the base method.

6. If, after all of the extractions, the base factory method has become empty, you can make it abstract. If there's something left, you can make it a default behavior of the method

```
public class ConcreteCreator : Creator
{
    public override IProduct FactoryMethod()
    {
        return new ConcreteProduct();
    }
}

// Usage:
Creator creator = new ConcreteCreator();
creator.SomeOperation();
```

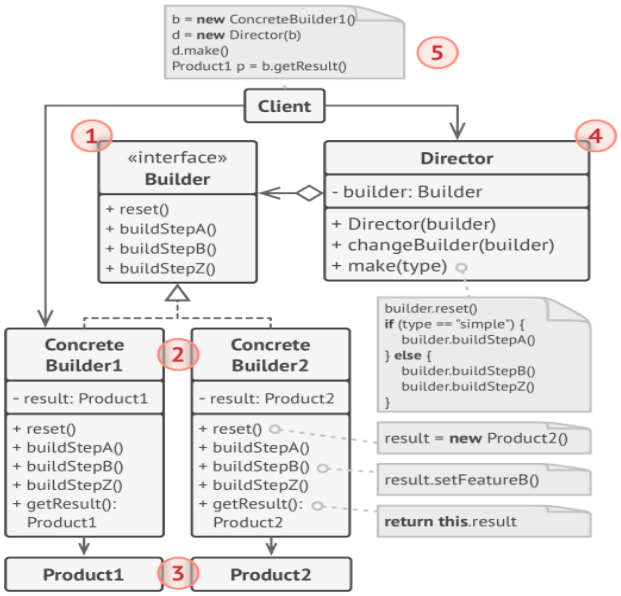
```
public interface IProduct
{
    void Operation();
}

public class ConcreteProduct : IProduct
{
    public void Operation()
    {
        Console.WriteLine("ConcreteProduct.Operation");
    }
}

public abstract class Creator
{
    public abstract IProduct FactoryMethod();

    public void SomeOperation()
    {
        IProduct product = FactoryMethod();
        product.Operation();
    }
}
```

3.Builder

Imagine a complex object that requires laborious, step-by-step initialization of many fields and nested objects. Such initialization code is usually buried inside a monstrous constructor with lots of parameters. Or even worse: scattered all over the client code. For example, let's think about how to create a House object. To build a simple house, you need to construct four walls and a floor, install a door, fit a pair of windows, and build a roof. But what if you want a bigger, brighter house, with a backyard and other goodies (like a heating system, plumbing, and electrical wiring)?

Solution

u extract the object construction code out of its own class and move it to separate objects called builders. The pattern organizes object construction into a set of steps (buildWalls , buildDoor , etc.). To create an object, you execute a series of these steps on a builder object. The important part is that you don't need to call all of the steps. You can call only those steps that are necessary for producing a particular configuration of an object.



Director The director class defines the order in which to execute the building steps, while the builder provides the implementation for those steps. In addition, the director class completely hides the details of product construction from the client code. The client only needs to associate a builder with a director, launch the construction with the director, and get the result from the builder

1. The **Builder** interface declares product construction steps that are common to all types of builders.

2. **Concrete Builders** provide different implementations of the construction steps. Concrete builders may produce products that don't follow the common interface.

3. **Products** are resulting objects. Products constructed by different builders don't have to belong to the same class hierarchy or interface.

4. The **Director** class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.

5. The **Client** must associate one of the builder objects with the director. Usually, it's done just once, via parameters of the director's constructor. Then the director uses that builder object for all further construction. However, there's an alternative approach for when the client passes the builder object to the production method of the director. In this case, you can use a different builder each time you produce something with the director.

How to implement

1. Make sure that you can clearly define the common construction steps for building all available product representations. Otherwise, you won't be able to proceed with implementing the pattern.

2. Declare these steps in the base builder interface.

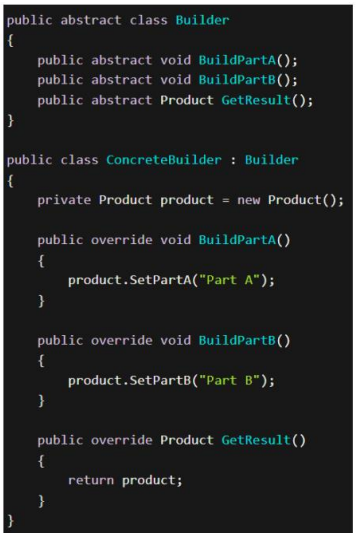
3. Create a concrete builder class for each of the product representations and implement their construction steps. Don't forget about implementing a method for fetching the result of the construction. The reason why this method can't be declared inside the builder interface is that various builders may construct products that don't have a common interface. Therefore, you don't know what would be the return type for such a method. However, if you're dealing with products from a single hierarchy, the fetching method can be safely added to the base interface.

4. Think about creating a director class. It may encapsulate various ways to construct a product using the same builder object.

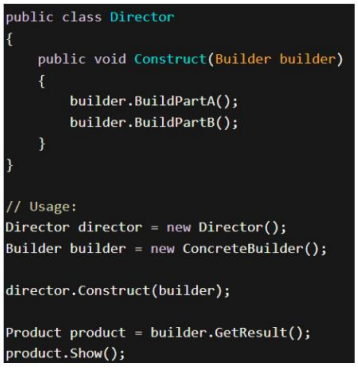
5. The client code creates both the builder and the director objects. Before construction starts, the client must pass a builder object to the director. Usually, the client does this only once, via parameters of the director's constructor. The director uses the builder object in all further construction. There's an alternative approach, where the builder is passed directly to the construction method of the director.

6. The construction result can be obtained directly from the director only if all products follow the same interface. Otherwise, the client should fetch the result from the builder.

```
public class Product
{
    private string partA;
    private string partB;

    public void SetPartA(string partA)
    {
        this.partA = partA;
    }

    public void SetPartB(string partB)
    {
        this.partB = partB;
    }

    public void Show()
    {
        Console.WriteLine($"Part A: {partA}\nPart B: {partB}")
    }
}
```



```
public abstract class Builder
{
    public abstract void BuildPartA();
    public abstract void BuildPartB();
    public abstract Product GetResult();
}

public class ConcreteBuilder : Builder
{
    private Product product = new Product();

    public override void BuildPartA()
    {
        product.SetPartA("Part A");
    }

    public override void BuildPartB()
    {
        product.SetPartB("Part B");
    }

    public override Product GetResult()
    {
        return product;
    }
}
```



```
public class Director
{
    public void Construct(Builder builder)
    {
        builder.BuildPartA();
        builder.BuildPartB();
    }
}

// Usage:
Director director = new Director();
Builder builder = new ConcreteBuilder();

director.Construct(builder);

Product product = builder.GetResult();
product.Show();
```