

Lesson 4 Template Method and Prototype

Template Method

1. Intent

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

2. Motivation

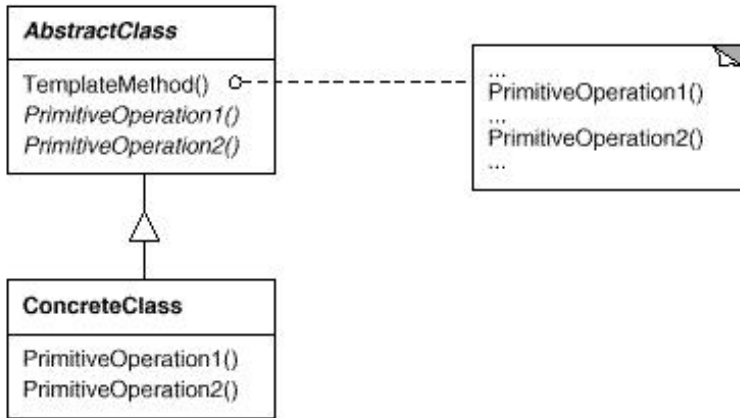
Sometimes you want to specify the order of operations that a method uses, but allow subclasses to provide their own implementations of some of these operations

3. Applicability

Use the Template Method pattern:

- a. To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
- b. To localize common behavior among subclasses and place it in a common class (in this case, a superclass) to avoid code duplication. This is a classic example of “code refactoring.”
- c. To control how subclasses extend superclass operations. You can define a template method that calls "hook" operations at specific points, thereby permitting extensions only at those points. The Template Method is a fundamental technique for code reuse.

4. Structure



5. Participants

a. AbstractClass

- Defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.
- Implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in **AbstractClass** or those of other objects.

b. ConcreteClass

- Implements the primitive operations to carry out subclass-specific steps of the algorithm.

6. How to implement the Template Method Pattern

- ### a. Abstract class with a 'skeleton' implementation of an algorithm. (In a template method, the parent class calls the operations of a subclass and not the other way around.)

This is an inverted control structure that's sometimes referred to as "the Hollywood principle," as in, "Don't call us, we'll call you".

```

public abstract class AOrderProcessor {

    public final void processOrder(IOrder order) {
        String orderDetails = getOrderDetails(order);
        float shippingFee = getShippingFee(order);
        int shippingTime = getShippingTime(order);
        sendConfirmation(orderDetails, shippingFee, shippingTime);
    }

    //by weight, volume or whichever-is-higher depending on
    //partnering agent
    public abstract float getShippingFee(IOrder order);

    //by distance and shipment type(air, sea or land...)
    public abstract int getShippingTime(IOrder order);

    public abstract void sendConfirmation(String orderDetails,
        float shippingFee, int shippingTime);

    private String getOrderDetails(IOrder order) {
        String od = "";
        od += "Thank you for shopping with us!";
        od += "Here is the detailed information on your order:";
        // more detailed information item by item here from 'order'
        // object.
        return od;
    }
}

```

b. Implement the Concrete subclass by overriding the abstract methods.

```

/**for bulky and heavy items but buyers are willing to wait
 * longer to reduce the shipping cost.
 * Logistics company uses sea containers.
 */
public class SlowOrderProcessor extends AOrderProcessor {

    @Override
    public float getShippingFee(IOrder order) {
        // calculate shipping cost based on order details
        return shippingFee;
    }

    @Override
    public int getShippingTime(IOrder order) {
        // get shipping time based on shipping address
        return shippingTime;
    }

    @Override

```

```

public void sendConfirmation(String orderDetails, float
    shippingFee, int shippingTime) { // send by email
}

```

Lab 4-1

A program stores/prints large alphabets by using characters instead of pixels (see below). Suppose the program stores the definition of each letter in a separate file and takes advantage of symmetries to reduce file sizes. For example, the following files define the letters 'A', 'B' and 'C', which represent all 3 cases of symmetry.

VERTICAL	NONE	HORIZONTAL
5	9	9
14	14	8
10	10	10
20	20	20
-----	-----	-----
-----+	+++++---	---++++--
---++	++-+++-	--++-++-
--++-	++-+++-	-++-+++-
--++-	++-+++-	-++-+++-
-++--	++-+++-	-++-+++-
-++--	++-+++-	-++-+++-
-++--	+++++---	-++-+++-
-++++	++-+++-	
-++--	++-+++-	
-++--	++-+++-	
-++--	++-+++-	
-++--	++-+++-	
-++--	+++++---	
-++--	+++++---	

The first line defines the symmetry type. The 2nd and the 3rd lines define the numbers of columns and rows, for example, in the letter 'A' file there are 5 columns and 14 rows. The 5th and 6th define the size of the complete letter. Complete letters are represented with a 10 x 20 matrix. Implement the program using a Template Method pattern to construct/print a complete letter from its definition. The skeleton algorithm is specialized for letters with a vertical axis of symmetry (like 'A'), horizontal axis (like 'C') or no symmetry (like 'B').

Hint: Your program takes an array as the input. Think of the task as multiple sub-tasks: identify symmetry type, reconstruct the letter and print it.

The Prototype Pattern

Programmers create a new object by cloning the prototypical instance to avoid resource-intensive operations or repeating the complex logic in building it every time. Thus the Prototype Pattern decouples the client from knowing all the details of the product classes it is dealing with.

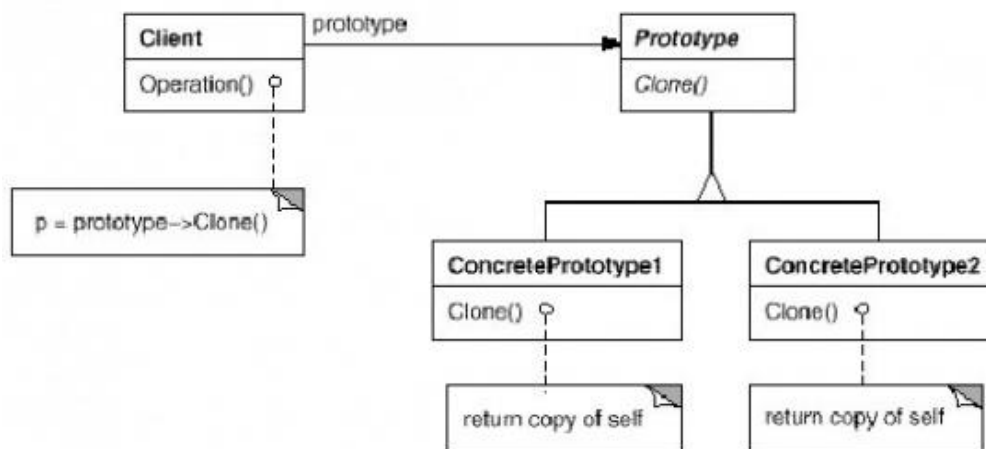
1. Intent

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

2. Motivation

- The Prototype Pattern allows you to make new instances by copying existing/template instances.
- When your application has to create objects without knowing their type or any details of how to create them.
- When creating a large/complex object that is resource intensive, cloning works better than using the constructor sometimes.

3. Structure



4. Participants

- a. Prototype
 - declares an interface for cloning itself.
- b. ConcretePrototype
 - implements an operation for cloning itself.
- c. Client
 - creates a new object by asking a prototype to clone itself.

5. Applicability

Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented (think about scenario1 for the Proxy pattern); and

- a. when the classes to instantiate are specified at run-time, for example, by dynamic loading; or
- b. to avoid building a class hierarchy of factories that parallels the class hierarchy of products; or

Scenarios in real life:

- a. Copy and Paste: copy an image and press 'ctrl+v' many times, you get multiple copies of it.
- b. In Word when you select 'Insert->Symbol', you get a symbol inserted into your document (like \neq , $\sqrt{\quad}$, \geq).
- c. When an employer hires employees, they give each a copy of the employment contract made out of the 'contract template'.

- d. A trading company gives samples of products (simple enough to replicate) to its partner factory, the factory produces large quantities of them.
- e. 3D printing.

6. Consequences

- a. It hides the concrete product classes from the client, thereby reducing the number of names clients know about.
- b. It lets a client work with application-specific classes without modification.
- c. Prototypes let you incorporate a new concrete product class into a system simply by registering a prototypical instance with the client. That's a bit more flexible than other creational patterns, because a client can install and remove prototypes at run-time.
- d. A client can exhibit new behavior by delegating responsibility to the prototype.
- e. This kind of design lets users define new "classes" without programming.
- f. Factory Method often produces a hierarchy of Creator classes that parallels the product class hierarchy. The Prototype pattern lets you clone a prototype instead of asking a factory method to make a new object. Hence you don't need a Creator class hierarchy at all.

7. How to implement the Prototype pattern?

a. Prototype interface

```
public interface Prototype {  
    public Prototype doClone();  
}
```

b. Prototype concrete class

```
public class Person implements Prototype {  
  
    String name;  
  
    public Person(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public Prototype doClone() {  
        return new Person(name);  
    }  
  
    public String toString() {  
        return "This person is named " + name;  
    }  
}
```

c. Client

```
public static void main(String[] args) {  
  
    Person person1 = new Person("Fred");  
    System.out.println("person 1:" + person1);  
    Person person2 = (Person) person1.doClone();  
    System.out.println("person 2:" + person2);  
  
}
```


Lab 4-2

Implement a deep copy for the clone() method of Employee class below.

```
public class Employee implements Cloneable, Serializable {  
  
    private int id;  
    private String Lastname;  
    private String Firstname;  
    private String streetAddress;  
    private String city;  
    private String state;  
    private String zipcode;  
    private Employee supervisor;  
    private Employee staff[];  
  
    @Override  
    protected Object clone(){  
        //implement deep copy here  
    }  
}
```