Design Patterns: Composite, Memento, Template Method, Decorator, Chain of Responsibility, Interpreter

**Composite**

[Skrien §8.7] We need to allow users to group figures together to make a “composite” figure, which can be manipulated as a single figure.

Our `Drawing` class stores all its figures in an `ArrayList` named “figures”. We could create a separate `ArrayList` for each group.

That is, to implement the "Group" menu item action, we can just

- remove all the selected figures from `figures`,
- put those selected figures into a new `ArrayList`, and
- add that `ArrayList` to `figures`.

The problem is that the `Drawing` class assumes that only `Figure` objects are in `ArrayLists`. So all drawing methods will have to be rewritten similarly to this:

```java
if( figures.get(i) instanceof ArrayList)
    ...do the desired processing on the group...
else
    ...do the desired processing on the figure...
```

Things would get worse if a group contained another group.

The Composite pattern solves this. It has

- one or more “leaf” classes, representing individual objects, and
- a “composite” class, representing collections of objects.

Both of these are subclasses of a `component` class.
A client can refer to components without knowing whether they are composites or leaves.

In our drawing program, which are the composites and which are the leaves?

Here is a UML diagram:

![UML Diagram]

**Exercise:** Let’s take a look at a sample method, `drawShape`, in each of the classes. Which class is each in?

```java
public void drawShape(Graphics g) {
    Rectangle rect = getBoundingRect();
    g.drawRect(rect.x, rect.y,
               rect.width, rect.height);
}
```

```java
public void drawShape(Graphics g) {
    for (Iterator<Figure> it = children.iterator();
         it.hasNext();)
        it.hasNext();
    }
```
public void drawShape(Graphics g) {
    Rectangle rect = getBoundingRect();
    g.drawOval(rect.x, rect.y,
               rect.width, rect.height);
}

Composite

Intent: Allow clients to treat individual objects and compositions of objects uniformly.

Problem: The application needs to manipulate a hierarchical collection of “primitive” and “composite” objects. Primitive objects are processed one way, and composite objects are handled differently. It is inelegant to query the “type” of each object before processing it.

Solution: Define an abstract base class (Component) that specifies the behavior that needs to be exercised uniformly across all primitive and composite objects. Make the Primitive and Composite classes subclasses of the Component class.

Memento

A standard feature of an editor is the ability to undo the most recent changes that the user has made.

Let’s add this functionality to our drawing application.

We’ll do this by adding an “undo” stack, which holds previous states of the canvas.

The top of the stack always corresponds to the current state of the canvas.

Now, what two things should we do when a change is undone?
We could make the `DrawingCanvas` handle undoing and redoing. But this gives it a lot of extra responsibility. Is there a more elegant way?

The Memento pattern solves this problem. It allows an external class to handle the states, but doesn’t let this external class manipulate them in any way.

Here is a description of Memento.

**Memento**

*Intent:* Without violating encapsulation, capture and externalize an object’s internal state so that the object can be returned to this state later.

*Problem:* We need to restore an object to its previous state.

*Solution:* Use an Originator, which creates the Mementos (states) and a Caretaker, which performs safekeeping of the Mementos.

*Implementation:* The client requests a Memento from the Originator when it needs to checkpoint the source object’s state. The source object sets the Memento to a characterization of its state. The Caretaker preserves the Memento, but only the Originator can store and retrieve information from the Memento (the Memento is "opaque" to the client and all other objects). If the client subsequently needs to “roll back” the source object’s state, it hands the Memento back to the Originator for reinstatement.


```java
public class UndoRedoHandler {

    private CanvasStateStack undoStack, redoStack;
    private DrawingCanvas canvas;

    public UndoRedoHandler(DrawingCanvas canvas) {
        undoStack = new CanvasStateStack();
        redoStack = new CanvasStateStack();
        this.canvas = canvas;
    }
```
// Store the initial state of the canvas on
// the undo stack
undoStack.push(canvas.createMemento());

...

public void undo() {
    if (undoStack.size() == 1)
        return; // Only current state is on stack

    DrawingCanvas.State canvasState =
        undoStack.pop();
    redoStack.push(canvasState);
    canvas.restoreState(undoStack.peek());
}

public void redo() {
    if (redoStack.isEmpty()) return;

    DrawingCanvas.State canvasState =
        redoStack.pop();
    undoStack.push(canvasState);
    canvas.restoreState(canvasState);
}

We only need to add two methods to the DrawingCanvas class. What are they?

Here is a UML sequence diagram that shows what happens when a change is undone.
1. The **ActionListener** for the Undo menu item signals the **UndoRedoHandler** to undo.

2. The **UndoRedoHandler** then
   a. extracts the **State** to be restored from its undo stack and
   b. signals the **DrawingCanvas** to restore its state to that **State**.

3. Since the internals of the **State** object are private, the **DrawingCanvas** cannot access them, but it can ask the **State** object to restore the canvas’s state for it.

*Exercise:* Come up with another example of Memento.

**Template Method**

*[HFDP, Ch. 8]* Say we have two classes whose code looks similar, but isn't identical.

We can duplicate the code, but change it where changes are needed. Is this a good idea?

Template Method pattern is a way of abstracting similar behavior in various classes and duplicating only the code that differs.

Here’s the example given in the text (p. 282):

```java
Coffee
void prepareRecipe() {
    boilWater();
    brewCoffeeGrinds();
    pourInCup();
    addSugarAndMilk();
}

Tea
void prepareRecipe() {
    boilWater();
    steepTeaBag();
    pourInCup();
    addLemon();
}
```

Do you see the similarity? **How might you avoid code duplication?**

- Have a single code sequence and test the class in the methods that need to use different behavior.
• Or, create an abstract class and make it the superclass; then implement methods differently in Coffee and Tea.

How should we write the code?

```java
public
    prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }
```

What other declarations do we need in this class?

```java
void boilWater() {
    System.out.println("Boiling water …");
}

void pourInCup() {
    System.out.println("Pouring …");
}

abstract void brew();
abstract void addCondiments();
```

What methods do the Coffee and Tea classes need to implement?

Now, let’s draw the class diagram, like it says in the book …

This is the Template Method pattern. Which method in it is a “template method”?

Now, let’s write up the pattern definition like we have done for other patterns before.

**Template Method**

*Intent:* Capture common parts of an algorithm in a single code sequence.
Problem: Two or more algorithms have some similarities, but are not identical. It would be wasteful and error prone to duplicate code for the common parts.

Solution: Define a skeleton of an algorithm in a method, deferring some steps to subclasses. The subclasses provide custom definitions of certain steps, without changing the algorithm's structure.

Implementation: The skeleton is placed in an abstract class. The deferred steps are defined in this class as abstract methods. The subclasses provide implementations for each abstract method.

Other examples of Template Method

(i) A game, where the initialization, alternating play, and testing for winner is the same.

(ii) x

(iii) x

(iv) x

One of the main pitfalls in Template Method is that sometimes processes are almost the same, but not exactly. Certain steps need to be performed in some cases, but not in others.

For example, in the coffee case, a customer may not want sugar and milk.

What's the best way to address this?

Our code now becomes …

```java
public abstract class CaffeineBeverage {
    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        if condimentsWanted() then
            addCondiments();
    }
```
boolean condimentsWanted() { return true; }

The hook method `condimentsWanted()` is given a default implementation, which can be overridden by subclasses.

Template Method helps avoid the pitfall of “dependency rot.” This means that high-level components (methods of the abstract class) “depend on” low-level components, as well as vice versa.

In this context, what does it mean to “depend on”?

Why would this be bad?

The dependency-inversion principle says,

A. High-level modules should not depend on low-level modules. Both should depend on abstractions.
B. Abstractions should not depend upon details. Details should depend upon abstractions.

What example of the dependency-inversion principle did we see earlier this week?

To prevent dependency inversion, we apply the *Hollywood principle*:

How can this be realized in the Template Method pattern?

How can we prevent the subclass methods from calling methods of the abstract class?

One commentator goes even further. He says that multiple layers of Template Methods are inherently prone to the yo-yo effect

**Template Method vs. Strategy**

What's the difference between Template Method and Strategy? *(HFDP pp. 308–309.)*
• Template Method defines the outline of an algorithm and lets subclasses fill in the parts that vary. It attempts to keep control over the sequence of steps; strategy does not.

• Strategy doesn’t require that the implementing classes be subclasses; it can use composition. Clients can change algorithms at run time by using a different strategy object. This basically lets them appear to change class at run time.

• Template Method depends on methods defined in the superclass (the template method(s)). Strategy doesn’t.

Decorator

[HFDP, Ch. 3] Why is too much subclassing bad?

• Because classes share behavior with their superclasses. It is impossible to change superclasses without changing the subclasses.

• Also, subclassing every time objects vary tends to require a lot of code to be duplicated, violating the DRY principle.

HFDP has a good example on p. 80. Here, a separate subclass is created for each combination of coffee and condiments.

There are four types of coffee:

• HouseBlend
• DarkRoast
• Decaf
• Espresso

There are many condiments, e.g., milk, soy, mocha, whip.

Each subclass has only one method. What is it?

Attempt #2: Let’s use a separate instance variable for each condiment … along with setters and getters, of course.

How should we implement cost()? In the superclass or subclasses?
This is better, but still not ideal. Why? (See p. 84)

What principle have we violated?

How can we solve this problem without using inheritance?

We will just have the with-condiments versions of our object ______ the original version.

Look at the diagram on p. 89. We start out with a cost for DarkRoast coffee.

Then if we want the cost of DarkRoast with Mocha, we give Mocha a cost() method.

Mocha’s ______ method calls the _______ method of Beverage.

In order to do this, Beverage and Mocha should both implement the same interface.

Now, we can do the same for all the other components. They will all have cost() methods. (How should this method be implemented?)

Suppose we have more than one condiment per beverage? We can wrap one component in another.

Beverage w/2 condiments → Beverage w/1 condiment → Beverage

Now let’s take a look at the class diagram.

Another nice thing about Decorator: You can add functionality at any time without disturbing the base object!

Exercise: Let’s think of some other uses of Decorator. Here are some examples submitted by students.

- Cars: You have the base model. Then you can add on what you want (power locks, leather seats, etc.).
- Reading encrypted files: The FileReader class can use the Decorator pattern to read encrypted files.
• Vacation packages: Adding different destinations is similar to
  decorating the coffee.

Chain of Responsibility

[HFDP, §14.3] You know how filters in Unix are so powerful? Take
  this one, for example:

    grep "execute(" `find mydir -name "*.java"` | awk -F: '{print }'
    | sort -u | wc -l

It prints the number of files in mydir that contain the string
  execute(.  

What happens? At each step, a request is passed along from one
  handler to the next.

Each handler can modify the request, by processing what it can, and
  then pass the request to the next handler.

The example in HFDP is for various e-mail handlers that route
  messages to various mailboxes depending on the subject.

Let’s take a look at the class diagram.

Exercise: What are some other examples of Chain of Responsibility?

• Code generation in a compiler.
• Try and Catch in programming languages
• A logging class. Each logging handler decides if any action is
  to be taken at this log level and then passes the message on to
  the next logging handler.
• An interrupt driver: Device drivers that are attached to the same
  interrupt; each get executed along to determine which handler
  should handle the interrupt.

Interpeter

[HFDP, §14.5] We all know the general idea of an interpreter: Read
  commands, and do what they say.
The example in *HFDP* is a duck, which can do one of three things:

- Fly
- Quack
- Turn right

One program for the interpreter is just …

```plaintext
right;
while (daylight) fly;
quack;
```

An interpreter always has some notion of compound commands, or repetition.

Let’s take a look at the class diagram of this interpreter, from p. 621, and the generalized class diagram.

*Exercise:* Can you think of another example?

- Bytecode interpreter
- Playing a musical score