Template (simple use of inheritance)

Algorithm expressed as a sequence of operations.
These operations must be replaced easily in the source

operation 1
operation 2
...

Template Method (trivial)

Different concrete classes implement different algorithms with the same sequence of operations
**Strategy**

- A user class requires one algorithm chosen among a set of algorithms for its execution.
- The algorithm is performed within some context, which contains the data.
- At initialization time, the current algorithm is set and then called by delegation.
- Template with delegation
- Similar to bridge
  \((\text{Bridge} \rightarrow \text{structure}; \ \text{Strategy} \rightarrow \text{behavior})\)

---

**Strategy**

1. Set the algorithm reference in the context
2. The algorithm is called from the context
3. The algorithm may have access to the context
Strategy (usage)

// Three contexts following different strategies
context = new Context(new ConcreteStrategyA());
context.execute();

context = new Context(new ConcreteStrategyB());
context.execute();

context = new Context(new ConcreteStrategyC());
context.execute();

Builder (similar to the strategy)
Builder

```
Builder
<< interface >>
Builder
buildPart()

ConcreteBuilder
buildPart()
getResult()
```

for all objects in structure {
    builder.buildPart()
}

Chain of responsibility

This pattern allows a request to pass along a chain of objects until one of them handles it, thus giving multiple objects a chance to handle the request.
Chain of responsibility

Client
\(\text{aClient} \rightarrow \text{aHandler} \rightarrow \text{aConcreteHandler} \rightarrow \text{anotherConcreteHandler}\)

\(\text{successor}\)

\(\text{handleRequest()}\)

\(\text{ConcreteHandler1} \rightarrow \text{ConcreteHandler2}\)

\(\text{handleRequest()}\)

\(\text{handleRequest()}\)
Command

1. First class operation

2. The command is provided with the object that must be processed

3. The command may be stored and performed later

Command Pattern

ConcreteCommand_1
- document
- execute()

ConcreteCommand_2
- document
- execute()

ConcreteCommand_3
- document
- execute()

Document
- action()

The command is defined within an object. It can thus be passed over to other objects, it can be stored in a list, called at any time later and so on. It is thus a mean to create a command as a first class operation. The command also binds a command to a receiver (document)
Command Pattern: *undo*

Useful to create undoable commands:

- The command receives the state of the document before the command is executed.
- It defines an undo function that restores the previous state.
- The commands are kept in a list, in the order of their execution.
- Can be combined with the memento pattern.

```
Command Pattern

Invoker

Command
execute()

ConcreteCommand
execute()

Receiver
action()

<<create and add command>>

Initializer

<<create>>

The commands can be stored in invokers, as in the observer pattern, but here, the command contains a reference to a document:

```
new ConcreteCommand(receiver)
```
Command: when to use it

- A document is read and then operations triggered by menu items are performed when a user clicks them (~callback, listener)

- One must prepare a context and have operations performed at a later time

- Support undo (stack the commands)

- Support transactions (which may have to be undone)

Memento

- Object used to save and later restore a state

- The caretaker should not see the state, it must only store and retrieve it.

- Can be used by a Command object to maintain the state of the undo.
Prototype

• Several objects with few variations must be created
• Their attribute values are only known at run time
• One instantiates a prototype and then clones it

Cloning objects in Java

```java
public class ToBeCloned implements Cloneable {
    String s = "xxx";
    int[] i = new int[] { 0,4,2 };

    public ToBeCloned clone() {
        try {
            ToBeCloned temp = (ToBeCloned) super.clone();
            temp.i = i.clone(); // an array must be cloned explicitly
            return temp;        // - Java defines clone() in arrays
        } catch (CloneNotSupportedException cnse) {
            return null;
        }
    }
    // for special constructs, the program must copy the contents explicitly
}
```
Prototype

```
operation() {
    prototype = new ConcretePrototypeC();
    p = prototype.clone();
}
```

Proxies

- **Remote proxy**
  (proxy that forwards the calls to an object located in a remote site)

- **Virtual proxy**
  (e.g. an object that represents a figure that is loaded only when the figure appears on the current page. Used for optimization purpose)

- **Protection proxy**
  (an object that intercepts all calls to an object and checks if the caller has the right to call it)

- **Handle**
  (e.g. to maintain the number of links to the object to destroy it when the last link has been freed, referencing a persistent object to load it when it is first referenced or to map it on a key, to lock it, and so on)
Proxy

Client
request()

Subject
request()

Proxy
request()

RealSubject
request()

Iterator

Head
Item
Item
Item
Item
Item

<<create>>

Iterator
Iterator

- Provides a way to traverse an aggregation according to different policies, without introducing the policy into the list object
- Filtering can be introduced
- The iterator references the list and vice-versa: a creator method in the list must connect the two (can be implemented as a factory)

It is important to have a remove operation within the iterator, otherwise one cannot remove an element during the scan, because if the current element is removed without the iterator knowing it, it would not skip the missing element.
Iterator: class diagram

```
Iterator: class diagram

Aggregate
createIterator()

Client

Iterator
first()
next()
isDone()
currentItem()

ConcreteAggregate
createIterator()

ConcreteIterator
return new ConcreteIterator(this)
```