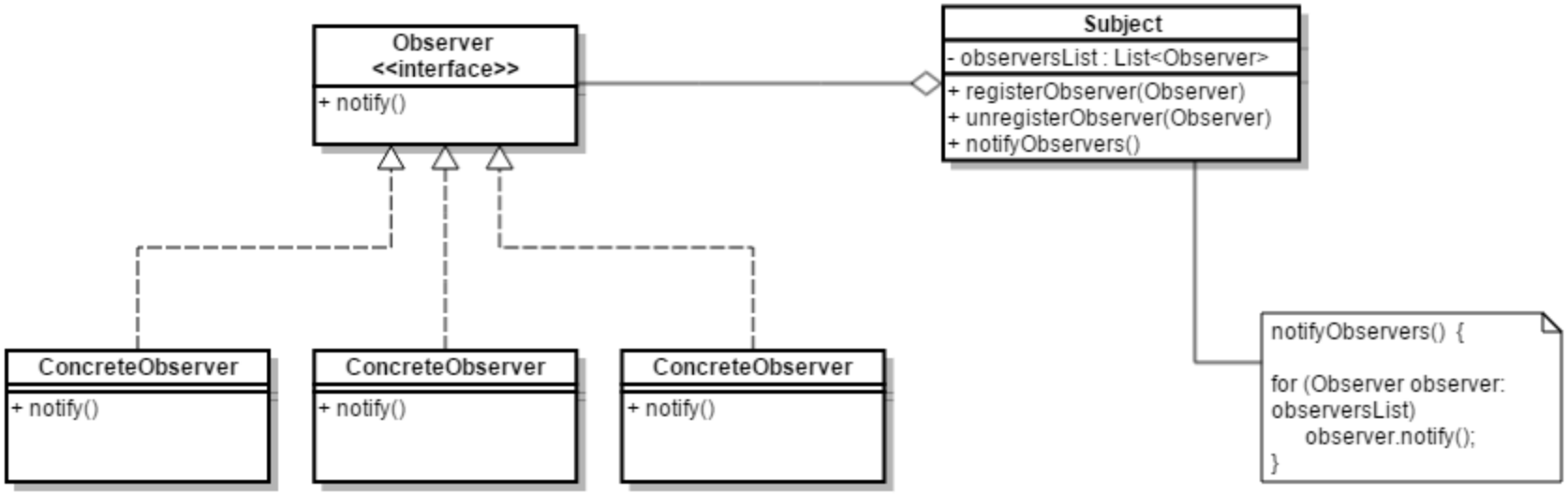
Design Patterns Part 4

by Vlad Ungureanu

Agenda

Design Patterns

- Observer
- Strategy
- Template Method
- Visitor


Observer
<<interface>>
+ notify()
Subject
- observersList : List<Observer>
+ registerObserver(Observer)
+ unregisterObserver(Observer)
+ notifyObservers()
ConcreteObserver
+ notify()
ConcreteObserver
+ notify()
ConcreteObserver
+ notify()
notifyObservers() {
for (Observer observer:
observersList)
observer.notify();
}

Observer – Implementation 1

```
public abstract class CustomObserver {
        protected Target target;
        public abstract void update();
}


public class PropertyChangeObserver extends CustomObserver {

        public PropertyChangeObserver(Target target) {
          this.target = target;
          this.target.attach(this);
        }

        @Override
        public void update() {
          System.out.println("Property : " +(target.getState()));
        }
}
```

Observer – Implementation 1

```
public class ValueChangeObserver extends CustomObserver {

    public ValueChangeObserver(Target target) {
      this.target = target;
      this.target.attach(this);
    }

    @Override
    public void update() {
      System.out.println("Value : " +(target.getState()));
    }
}
```

Observer – Implementation 1

```
public class Target {
        private List<CustomObserver> observers = new ArrayList<CustomObserver>();
        private int state;
        public int getState() {
          return state;
        }
        public void setState(int state) {
          this.state = state;
          notifyAllObservers();
        }
        public void attach(CustomObserver observer) {
          observers.add(observer);
        }

        public void notifyAllObservers() {
           for (CustomObserver observer : observers) {
                   observer.update();
           }
        }
}
```

Observer – Implementation 2

```
public class Observer implements java.util.Observer {

    public void update(Observable obs, Object x) {
      System.out.println("update(" + obs + "," + x + ");");
    }
}

public class ObservableTarget extends Observable {

    public void changeSomething(int value) {
      setChanged();
      notifyObservers(value);
    }
}
```

Observer – Implementation

```
public static void main(String[] args) {
        // custom observer
        Target target = new Target();

        PropertyChangeObserver propertyChangeObserver = new PropertyChangeObserver(target);
        ValueChangeObserver valueChangeObserver = new ValueChangeObserver(target);
        System.out.println("First state change: 15");
        target.setState(15);

        // language specific observer
        ObservableTarget observableTarget = new ObservableTarget();
        observableTarget.addObserver(new Observer());
        observableTarget.changeSomething(5);
}
```

Observer – Why ?

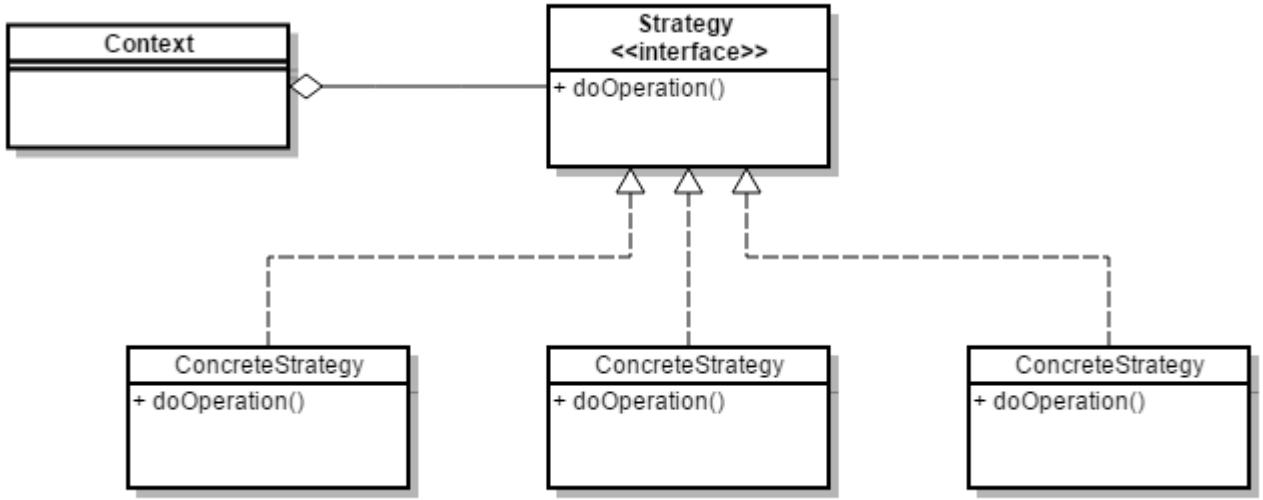
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

- Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.

- The "View" part of Model-View-Controller.

Observer – Why not ?

- Actors make better observers
- Some observer implementations (drag and drop) introduce bad practices (no separation of concern, they cannot hold state)
- Chains of observers are difficult to manage
- Might trigger side effects


Context
Strategy
<<interface>>
+ doOperation()
ConcreteStrategy
+ doOperation()
ConcreteStrategy
+ doOperation()
ConcreteStrategy
+ doOperation()

Strategy – Implementation

```
public interface OperationStrategy {
        int calculate();
}

public class Strategy {

      private OperationStrategy operationStrategy;

      public void setOperationsStrategy(OperationStrategy operationStrategy){
      this.operationStrategy = operationStrategy;
      }

      public int estimate(){
      return 5 * operationStrategy.calculate();
      }

}
```

Strategy – Implementation

```
public class NormalStrategy implements OperationStrategy{

    @Override
    public int calculate() {
      return 5;
    }

}

public class ComplexStrategy implements OperationStrategy{

    @Override
    public int calculate() {
      return 10;
    }

}
```

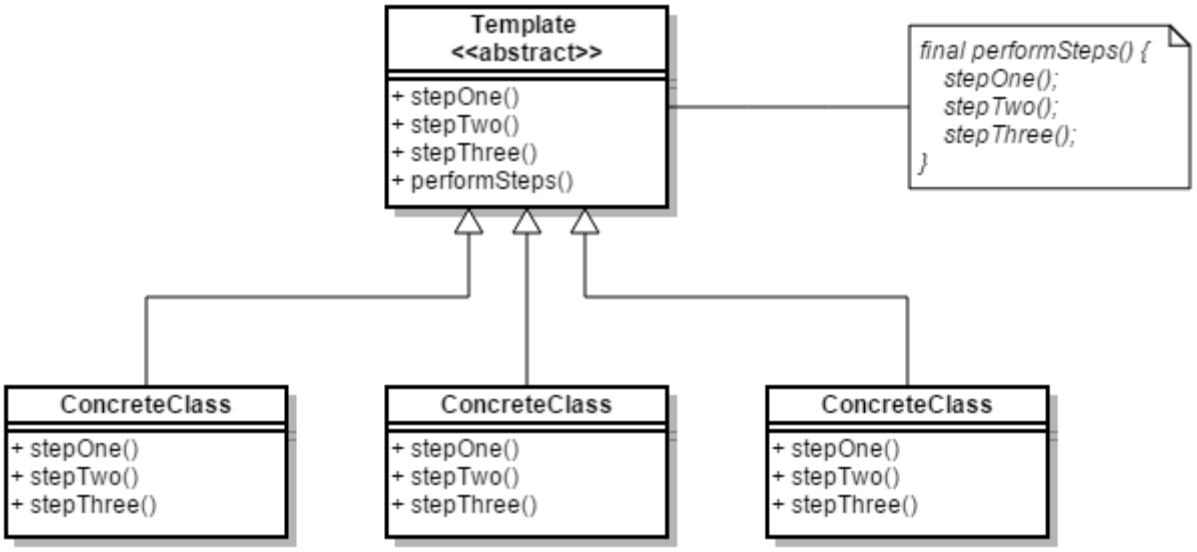
Strategy – Why ?

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

- Capture the abstraction in an interface, bury implementation details in derived classes.

Strategy – Why not ?

- State, Strategy, Bridge, Decorator (and to some degree Adapter) have similar solution structures
- Extra layer of complexion
- Might hide complex operations
- If multiple decorators are use intent might be obscured


Template
<<abstract>>
+ stepOne()
+ stepTwo()
+ stepThree()
+ performSteps()
final performSteps() {
stepOne();
stepTwo();
stepThree();
}
ConcreteClass
+ stepOne()
+ stepTwo()
+ stepThree()
ConcreteClass
+ stepOne()
+ stepTwo()
+ stepThree()
ConcreteClass
+ stepOne()
+ stepTwo()
+ stepThree()

Template Method – Implementation

```
public abstract class TemplateMethod {

    public abstract void start();
    public abstract void finish();
    public abstract void task();

    public void use(){
      this.start();
      this.task();
      this.finish();
    }
}
```

Template Method – Implementation

```
public class PublicDevice extends TemplateMethod {

    @Override
    public void start(){
      System.out.println("Public Start!");
    }
    @Override
    public void finish(){
      System.out.println("Public Finish!");
    }
    @Override
    public void task(){
      System.out.println("Public Task!");
    }
}
```

Template Method – Implementation

```
public class SecureDevice extends TemplateMethod{

    @Override
    public void start() {
      System.out.println("Secure Start!");
    }
    @Override
    public void finish() {
      System.out.println("Secure Finish!");
    }
    @Override
    public void task() {
      System.out.println("Secure Task!");
    }

}
```

Template Method – Why ?

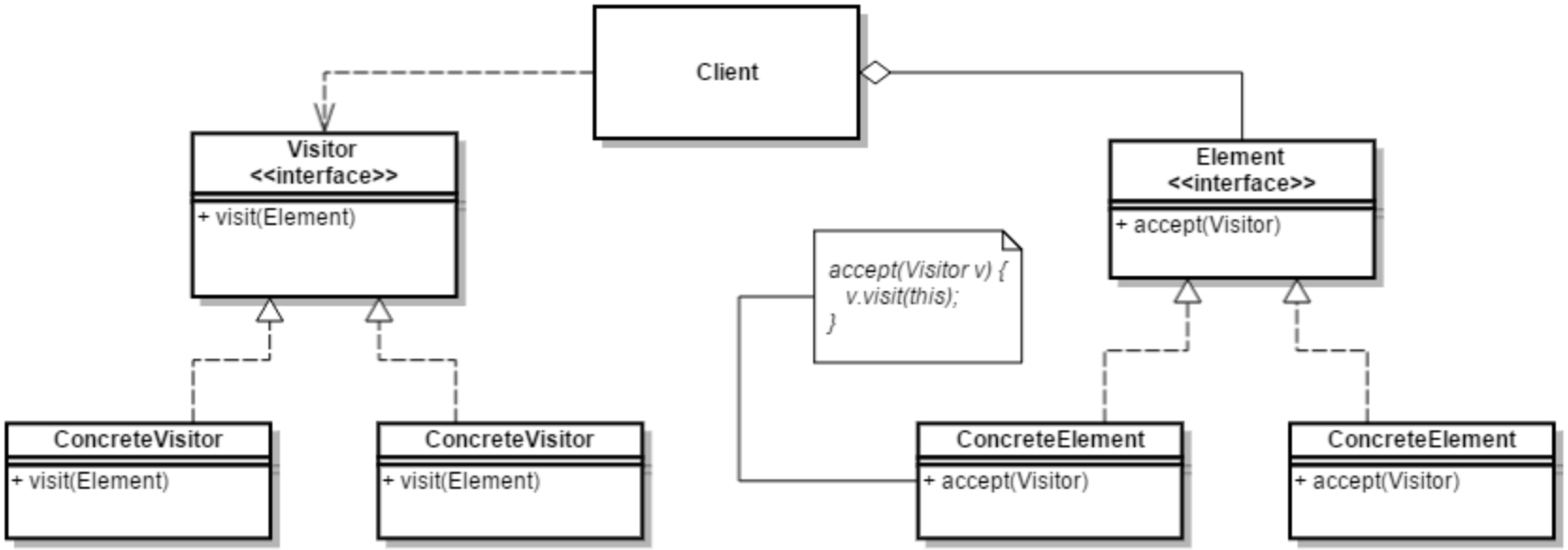
- Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

- Base class declares algorithm 'placeholders', and derived classes implement the placeholders

LEARN stuff IO

Template Method – Why not ?

- Restricts you to a single inheritance in Java
- Extra layer of complexion
- Implementation might hide complex operations
- A generic form of Factory Method


Client
Visitor
<<interface>>
+ visit(Element)
Element
<<interface>>
+ accept(Visitor)
accept(Visitor v) {
v.visit(this);
}
ConcreteVisitor
+ visit(Element)
ConcreteVisitor
+ visit(Element)
ConcreteElement
+ accept(Visitor)
ConcreteElement
+ accept(Visitor)

Visitor – Implementation

```
public interface Visitable {
void accept(Visitor visitor);
}
public interface Visitor {
    void visitSimpleElement(SimpleElement simpleElement);
    void visitComplexElement(ComplexElement complexElement);
}
public class NosyVisitor implements Visitor{
    @Override
    public void visitSimpleElement(SimpleElement simpleElement) {
       System.out.println("Visited simple element!");
    }
    @Override
    public void visitComplexElement(ComplexElement complexElement) {
       System.out.println("Visited complex element!");
    }
}
```

Visitor – Implementation

```
public class ComplexElement implements Visitable{

    @Override
    public void accept(Visitor visitor) {
      visitor.visitComplexElement(this);
    }

}
public class SimpleElement implements Visitable{

    @Override
    public void accept(Visitor visitor) {
      visitor.visitSimpleElement(this);
    }

}
```

Visitor – Implementation

```
public class VisitorMain {

    public static void main(String[] args) {
            List<Visitable> items = new ArrayList<>();
            items.add(new SimpleElement());
            items.add(new ComplexElement());
            NosyVisitor visitor = new NosyVisitor();
            for (Visitable item : items)
            {
                item.accept(visitor);
            }
    }
}
```

Visitor – Why ?

- Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- The classic technique for recovering lost type information.
- Do the right thing based on the type of two objects.
- Double dispatch

LEARN stuff IO

Visitor – Why not ?

- Arguments and the return type of visiting methods have to be known in advance.
- Extra layer of abstraction.
- A lot of code has to be written to prepare the use of visitors.
- If a visitor pattern has not been written in the first time, the hierarchy has to be modified to implement it.

THANK YOU!

Vlad Costel Ungureanu
ungureanu_vlad_costel@yahoo.com

This is a free course from LearnStuff.io
– not for commercial use –