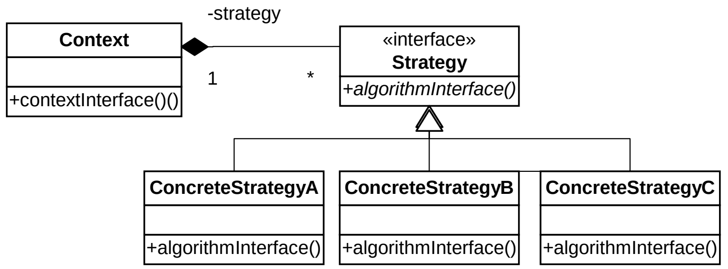
Behavioral Patterns - the rest

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.

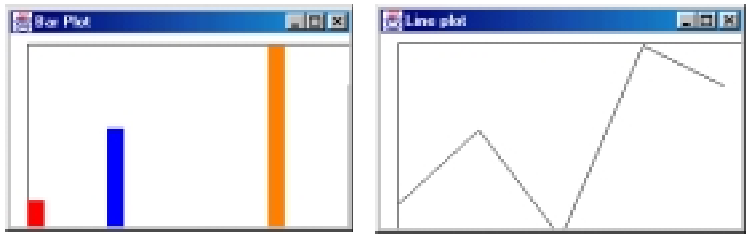
Strategy (aka Policy)

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



Structure

Example - Strategy



Example - Strategy

```
public abstract class PlotStrategy extends JFrame {
  ...
  public abstract void plot(float xp[], float yp[]);
  ...
}
```

```
public class Context {
    private PlotStrategy plotStrategy;
    public void setBarPlot() {
        plotStrategy = new BarPlotStrategy();}
    public void setLinePlot() {
        plotStrategy = new LinePlotStrategy(); }
    public void plot() { plotStrategy.plot(x, y);}
}
```

Applicability - Strategy

- When many related classes differ only in their behaviour.
- You need different variants of an algorithm.
- To encapsulate algorithm specific datastructures.
- A class defines many behaviours and these appear as multiple conditional statements. Move related conditional branches into their own Strategy class.

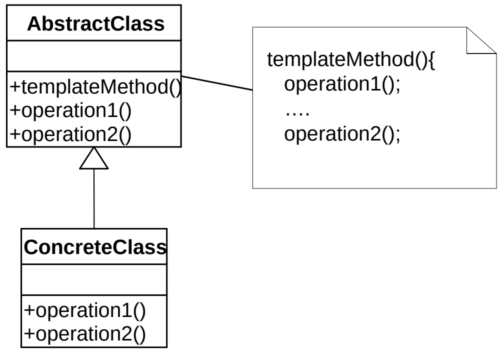
Consequences - Strategy

- Defines families of related algorithms.
- Its is an alternative to direct sub-classing. We could sub-class Context - “hard-wired”.
- Strategies eliminate conditional statements.
- Can have choice of implementations for the same behaviour.
- Disadvantages: Communication overhead - Context-Strategy + increased #objects.

Template Method

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses refine steps without changing algorithm structure.

Structure



Template Method - Example

```
public abstract class AbstractCollection
implements Collection {
  public abstract Iterator iterator();
  public abstract int size();
  public boolean contains(Object o) {
      Iterator e = iterator();
      if (o==null) {
          while (e.hasNext())
            if (e.next()==null) return true;
      } else {
          while (e.hasNext())
            if (o.equals(e.next())) return true;
      }
      return false;
  }
  ...
}
```

The AbstractCollection class from java.util -Tutorial 6, includes examples of template methods:

Applicability - Template method

- Implement the invariant part of algorithm once and leave parts that vary to subclasses.
- Factor out common behaviour in subclasses - “refactor to generalize”.
- Control subclass extensions - “hook” operations.

Consequences - Template method

- Fundamental technique for code re-use.
- Leads to an inverted control structure in which base class calls sub-class methods:- “Hollywood Principle - Don’t call us, we’ll call you”.
- Template methods call:
  - concrete operations
  - primitive operations
  - factory methods
  - **hook operations** - default behaviour that can be over-ridden by subclasses.

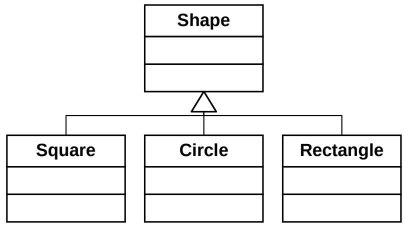
Visitor

Represent an operation to be performed on the elements of an object structure. Visitor permits a new operation to be defined without changing the classes of the elements on which it operates.

An external class is created to act on the data in other classes.

Can also be though of as an OO way of implementing a switch statement…

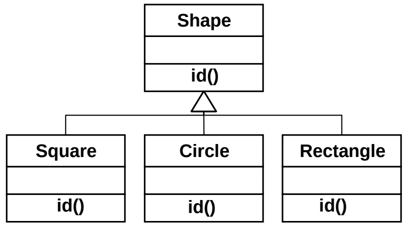
Motivation - Visitor



Suppose we want a method Selected to print out the object class when we click on it.

```
void selected(Shape obj) {
   if (obj instanceof Circle)
     System.out.println("its a Circle");
   else if (obj instanceof Square)
     System.out.println("its a Square");
   else
     System.out.println("its a Rectangle");
}
```

Motivation - Visitor



However, for 100 shapes, we have up to 99 comparisons, try...

```
abstract class Shape {
    abstract int id()
}

class Circle Extends Shape
{   ...
    static final int ID = 1;
    int id() { return ID; }
}
```

```
class Square Extends Shape
{   ...
    static final int ID = 2;
    int id() { return ID; }
}

class Rectangle Extends Shape
{   ...
    static final int ID = 3;
    int id() { return ID; }
}
```

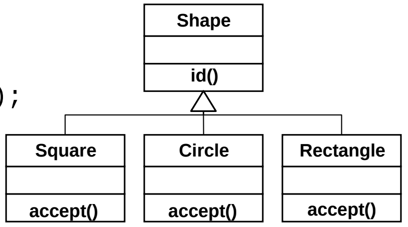
Motivation - Visitor

```
void selected(Shape obj) {
   switch (obj.id()) {
   case Circle.ID:
     System.out.println("its a Circle"); break;
   case Square.ID:
     System.out.println("its a Square"); break;
   case Rectangle.ID:
     System.out.println("its a Rectangle"); break;
}
```

This solution has the problem of leaving the management of unique ID's to the programmer - easy to make mistake + ID is redundant since we can identify class of an object using instanceof

Example - visitor

```
interface Visitor {
   void visit(Circle obj);
   void visit(Square obj);
   void visit(Rectangle obj);
}
```

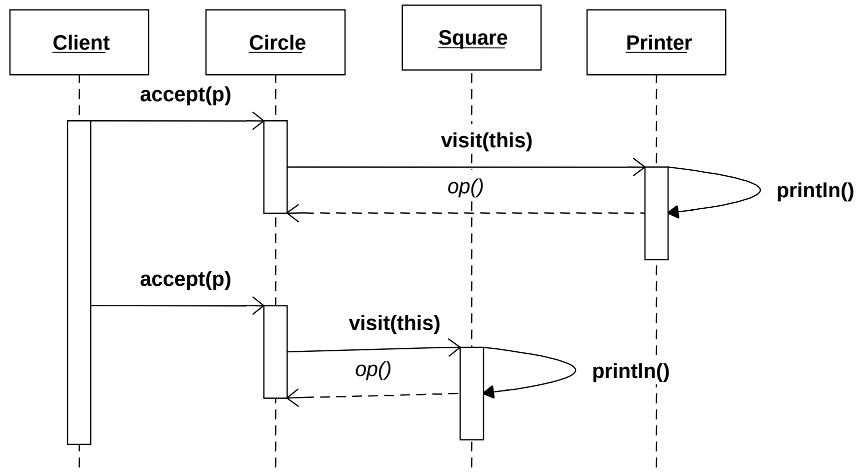


```
abstract class Shape {
  ...
  abstract void accept(Visitor v);
}

class Circle Extends Shape
{  ...
   void accept(Visitor v)
   {v.visit(this);}
}
```

Example - visitor

```
class Printer extends Visitor {
   void visit(Circle obj) {
      System.out.println("its a Circle");
   }
   void visit(Square obj) {
      System.out.println("its a Square");
   }
   void visit(Rectangle obj) {
      System.out.println("its a Rectangle");
   }
}

//using the Printer Visitor
void selected(Shape obj) {
   obj.accept(new Printer());
}
```

Collaboration - Visitor



Double Dispatch

Applicabiliy - Visitor

- Essentially, when the classes defining an object structure rarely change, but you want to to define new operations over the structure.
  (e.g. the compiler-compiler sablecc generates set of classes - which accept visitors - to represent abstract syntax tree. Users write visitor classes to implement static semantic checks, pretty printing, code generation etc.

Consequences - Visitor

- Visitor makes adding new operation easy - add new visitor class.
- Visitor gathers related operations and separates unrelated ones.
- Adding new concrete elements (e.g. Shape classes) is difficult.
- Can break encapsulation - visitors most have access to enough visited element state to perform their function.

Discussion Point

"Most problems in computer science can be solved by another level of indirection".