CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2002
Doc 8 Strategy

Contents

Strategy ....................................................................................................................... 2
Intent ......................................................................................................................... 2
Applicability ........................................................................................................ 8
Consequences .......................................................................................................... 9
Implementation ..................................................................................................... 10

References

Design Patterns: Elements of Reusable Object-Oriented Software, 
Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, pp. 315-314

The Design Patterns Smalltalk Companion, Alpert, Brown, 

Copyright ©, All rights reserved. 2001 SDSU & Roger Whitney, 5500 Campanile Drive, San 
Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opi.shtml) license 
defines the copyright on this document.
**Strategy Intent**

Define a family of algorithms, encapsulate each one, and make them interchangeable.

Strategy lets the algorithm vary independently from clients that use it.

**Structure**

```
Context
  ContextInterface()

Strategy
  AlgorithmInterface()

ConcreteStrategyA
  AlgorithmInterface()

ConcreteStrategyB
  AlgorithmInterface()

ConcreteStrategyC
  AlgorithmInterface()
```
Examples

Java Layout Managers for Windows

Java Comparators

Smalltalk sort blocks

Java Layout Managers

```java
import java.awt.*;
class FlowExample extends Frame {

    public FlowExample( int width, int height ) {
        setTitle( "Flow Example" );
        setSize( width, height );
        setLayout( new FlowLayout( FlowLayout.LEFT) );

        for ( int label = 1; label < 10; label++ )
            add( new Button( String.valueOf( label ) ) );
        show();
    }

    public static void main( String args[] ) {
        new FlowExample( 175, 100 );
        new FlowExample( 175, 100 );
    }
}
```
Why Not use Inheritance?

But there are:

• 20 different Layout classes
• At least 39 subclasses of Component using layouts

So using inheritance would require 780 classes!
Java Comparators

import java.util.Comparator;
import java.util.*;
class Student {  
    String name;
    
    public Student( String newName) { name = newName;}
    
    public String toString() { return name; }
}

final class StudentNameComparator implements Comparator {
    
    public int compare( Object leftOp, Object rightOp ) {
        String leftName = ((Student) leftOp).name;
        String rightName = ((Student) rightOp).name;
        return leftName.compareTo( rightName );
    }

    public boolean equals( Object comparator ) {
        return comparator instanceof StudentNameComparator;
    }
}

public class Test {
    public static void main(String args[]) {
        Student[] cs596 = { new Student( "Li" ), new Student( "Swen" ),
                        new Student( "Chan" ) };
        //Sort the array
        Arrays.sort( cs596, new StudentNameComparator() );
    }
}
Smalltalk SortBlocks

| list |
list := #( 1 6 2 3 9 5 ) asSortedCollection.
Transcript
  print: list;
  cr.
list sortBlock: [:x :y | x > y].
Transcript
  print: list;
  cr;
  flush.
Why Not use Inheritance

There are arbitrarily many ways to sort

So get arbitrarily many

- Subclasses of SortedCollection or
- Comparator classes (blocks)

But with comparators (blocks) one can:

- Combine different comparators
- Sort the same list with different comparators
Applicability

Use the Strategy pattern when

• You need different variants of an algorithm

• An algorithm uses data that clients shouldn't know about

• A class defines many behaviors, and these appear as multiple switch statement in the classes operations

• Many related classes differ only in their behavior
Consequences

- Families of related algorithms
- Alternative to subclassing of Context
  What is the big deal? You still subclass Strategy!
- Eliminates conditional statements

Replace in Context code like:

```java
switch ( flag ) {
    case A: doA(); break;
    case B: doB(); break;
    case C: doC(); break;
}
```

With code like:

```java
strategy.do();
```

- Gives a choice of implementations
- Clients must be aware of different Strategies

```java
SortedList studentRecords = new SortedList(new ShellSort());
```

- Communication overhead between Strategy and Context
- Increase number of objects
Implementation

• Defining the Strategy and Context interfaces

  How does data flow between them

  Context pass data to Strategy

  Strategy has point to Context, gets data from Context

  In Java use inner classes

• Strategies as template parameters

  Can be used if Strategy can be selected at compile-time and does not change at runtime

  SortedList<ShellSort> studentRecords;

• Making Strategy objects optional

  Give Context default behavior

  If default used no need to create Strategy object