CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2004
Doc 8 Adapter & Strategy
Contents

References

Design Patterns: Elements of Resuable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison Wesley, 1995, pp. 139-150, 315-324

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, Addision-Wesley, 1998, pp. 105-120, 339-354

Adapter
Motivating Adapter
Java CGI & Servlets

Both Java CGI and servlets are used for server-side processing of certain HTML requests, like processing HTML forms

Servlets have greater functionality and are faster, but require special Web servers or servers with special extensions

To help write Java CGI programs there is class sdsu.net.CGI

It would be useful in moving code between servers to avoid having to rewrite the code

One Problem

One issue is access to the CGI environment variables

There are about 20 common CGI environment variables

In servlets one has an HttpRequest class that has a getX() method for each CGI environment variable

sdsu.net.CGI class returns a hash table with one entry per CGI environment variable

Solution

We can write a wrapper around HttpRequest to make it act like a hash table

The Wrapper or Adapter

```
class CGIAdapter extends Hashtable
  {
  Hashtable CGIvariables = new Hashtable( 20);

  public CGIAdapter( HttpRequest CGIEnvironment )
    {
    CGIvariables.put( "AUTH_TYPE" ,
                CGIEnvironment.getAuthType());

    CGIvariables.put( "REMOTE_USER" ,
                CGIEnvironment.getRemoteUser());

    etc.
    }

  public Object get(Object key)
    {
    return CGIvariables.get( key );
    }

    etc.
  }
```

Going the other Direction

Adapting servlet code to normal CGI requires extracting the CGI environment variables out of the hash table and putting them into an object that implements the public interface of the HttpRequest class

```
class HTTPAdapter extends HttpRequest
  {
  Hashtable CGIvariables;

  public HTTPAdapter( Hashtable CGIEnvironment )
    {
    CGIvariables = CGIEnvironment;
    }

  public String getAuthType()
    {
    return (String) CGIvariables.get( "AUTH_TYPE" );
    }

  public String getRemoteUser()
    {
    return (String) CGIvariables.get( "REMOTE_USER" );
    }

    etc.
  }
```

Adapter

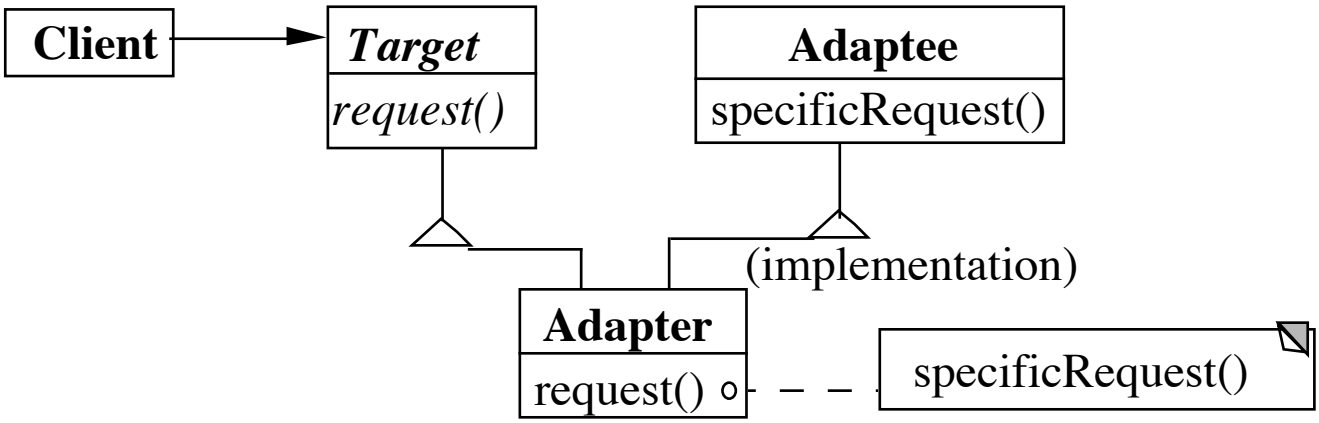
The adapter pattern converts the interface of a class into another interface.

Use the Adapter pattern when

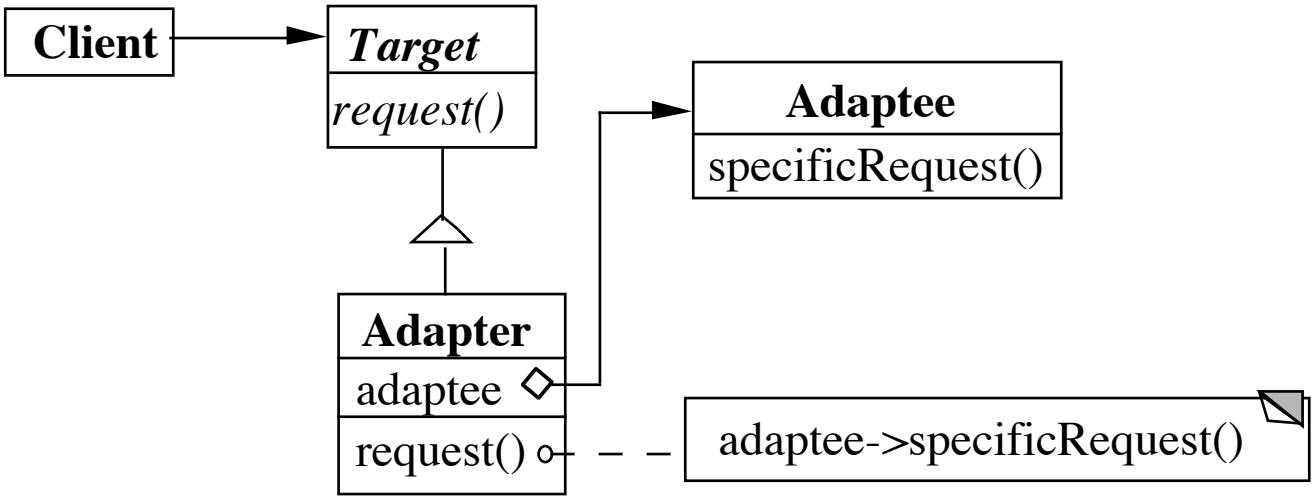
- You want to use an existing class and its interface does not match the one you need
- You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is classes that don't necessarily have compatible interfaces
- You need to use several existing subclasses, but it's impractical to adapt their interface by subclassing everyone. An object adapter can adapt the interface of its parent class

Adapter has two forms:
- Class Adapter
- Object Adapter

Class Adapter



Object Adapter



Class Adapter Example

```
class OldSquarePeg {
   public:
      void squarePegOperation()
         { do something }
}

class RoundPeg {
   public:
      void virtual roundPegOperation = 0;
}

class PegAdapter: private OldSquarePeg,
            public RoundPeg  {
   public:
      void virtual roundPegOperation()  {
         add some corners;
         squarePegOperation();
      }
}

void clientMethod() {
   RoundPeg* aPeg = new PegAdapter();
   aPeg->roundPegOperation();
}
```

Object Adapter Example

```
class OldSquarePeg{
   public:
      void squarePegOperation() { do something }
   }

class RoundPeg{
   public:
      void virtual roundPegOperation = 0;
   }

class PegAdapter:  public RoundPeg     {
   private:
      OldSquarePeg* square;

   public:
      PegAdapter()  { square = new OldSquarePeg; }

      void virtual roundPegOperation()    {
         add some corners;
         square->squarePegOperation();
         }
   }
```

Consequences

A Class adapter uses inheritance so

- Only adapts a class and all its parents, not all its subclasses
- Lets Adapter override some of Adaptee's behavior
- Does not introduce an additional pointer indirection

An object adapter uses object composition so

- Lets a single Adapter work with many Adaptees
- Makes it harder to override Adaptee behavior as the Adapter may not know with Adaptee it is working with

Other issues:

- How much adapting does the Adapter do?
- Pluggable adapters
- Using two-way adapters

How Much Adapting does the Adapter do?

The adapter may have to work very little or a great deal to adapt the Adaptee to the Target

The Adapter may just map one operation to another

```
class PegAdapter:  public RoundPeg     {
   private:
      OldSquarePeg* square;

   public:
      PegAdapter() { square = new OldSquarePeg;}

      void roundPegOperation()       {
         square->squarePegOperation();
      }
}
```

The Adapter may have to work hard if the Target operation does not have a comparable operation in the Adaptee

Pluggable Adapters

In the CGI example we adapted a class with getX() methods to a hash table interface

It is likely that we may adapt a class with getX() methods to a hashtable in the future

It would be nice to write one class to do all such adapting

This class would be given a list of keys to getX methods and an Adaptee object

```
HttpRequest CGIEnvironment = getHttpRequest();
PluggableHashAdapter sample =
   new PluggableHashAdapter( CGIEnvironment );

sample.adapt( "AUTH_TYPE" , getAuthType );
sample.adapt( "REMOTE_USER" , getRemoteUser );
etc.

sample.get( “REMOTE_USER” );
```

Pluggable Adapters are used in interface components, where we know in advance that we will adapt the component to other interfaces

Pluggable Adapters are common in Smalltalk, were it is easier to map strings to method calls

Using two-way Adapter

In the SquarePeg-RoundPeg example the SquarePeg is adapted to the RoundPeg

So a SquarePeg can be used where a RoundPeg is needed, but not the other way around.

A two-way adapter would also allow a RoundPeg be used in place of the SquarePeg

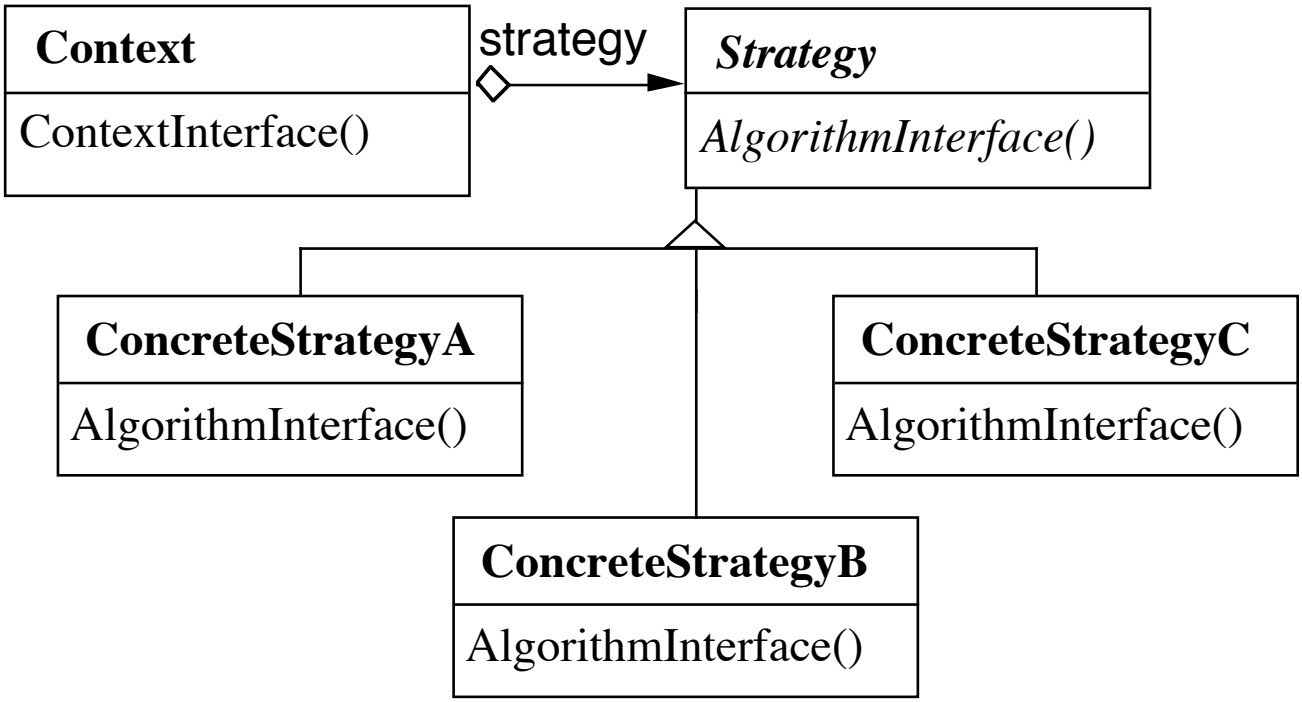
```
class OldSquarePeg {
  public:
    void virtual squarePegOperation() { blah }
}

class RoundPeg {
  public:
    void virtual roundPegOperation() { blah }
}

class PegAdapter: public OldSquarePeg, RoundPeg {
  public:
    void virtual roundPegOperation() {
      add some corners;
      squarePegOperation();
    }
    void virtual squarePegOperation() {
      add some corners;
      roundPegOperation();
    }
}
```

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



Examples

Java Layout Managers for Windows

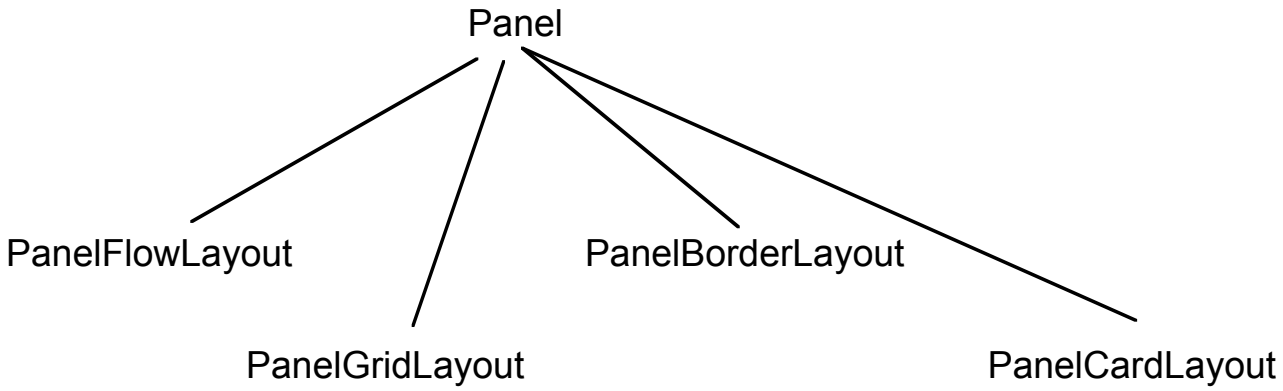
Java Comparators

Smalltalk sort blocks

Java Layout Managers

```
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!

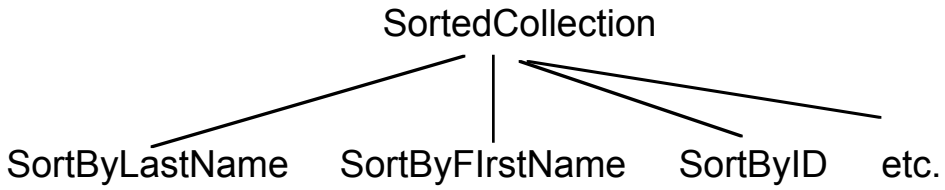
Java Comparators

```
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:

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

  With code like:

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

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

```
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