CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2002
Doc 9 State, Proxy & Adapter
Contents

Design Principle 2 2
State 3
  Example - SPOP 3
  Intent 10
  Applicability 10
  Issues 11
    How much State in the State 11
    Who defines the state transitions? 12
    Sharing State Objects 14
    Creating and Destroying State Objects 17
    Changing the Context Class for Real 18
  Consequences 19
  State Verses Strategy 20
Proxy 21
  Structure 21
  Dynamics 22
  Reasons for Object Proxies 24
  Smalltalk Proxy Tricks 28
Adapter 32
  Motivating Adapter 32
  Adapter 36
  Consequences 40

References

Design Patterns: Elements of Resuable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison Wesley, 1995, pp. 139-150, 207-217, 305-314

Pattern-Oriented Software: A System of Patterns, Buschman, Meunier, Rohnert, Sommerlad, Stal, 1996, pp. 263-275

Advanced C++: Programming Styles and Idioms, James Coplien, 1992, pp. 58-72

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, Addision-Wesley, 1998, pp. 105-120, 213-222, 327-338

Smalltalk Scaffolding Patterns, Doble & Auer in Pattern Languages of Program Design 4, Addison Wesley, Eds. Harrison, Foote, Rohnert, 2000, pp. 199-219

Copyright ©, All rights reserved. 2002 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.

Design Principle 2

Favor object composition over class inheritance

Composition

- Allows behavior changes at run time
- Helps keep classes encapsulated and focused on one task
- Reduce implementation dependencies

Inheritance

```
class A {
   Foo x
   public int complexOperation() { blah }
}

class B extends A {
   public void bar() { blah}
}
```

Composition

```
class B {
   A myA;
   public int complexOperation() {
      return myA.complexOperation()
   }

   public void bar() { blah}
}
```

State
Example - SPOP
Simple Post Office Protocol

SPOP is used to download e-mail from a server

SPOP supports the following command:

- USER <username>
- PASS <password>
- LIST
- RETR <message number>
- QUIT

USER & PASS Commands

USER with a username must come first
PASS with a password or QUIT must come after USER

If the username and password are valid, then the user can use other commands

LIST Command

Arguments: a message-number (optional)

If it contains an optional message number then returns the size of that message

Otherwise return size of all mail messages in the mailbox

RETR Command

Arguments: a message-number

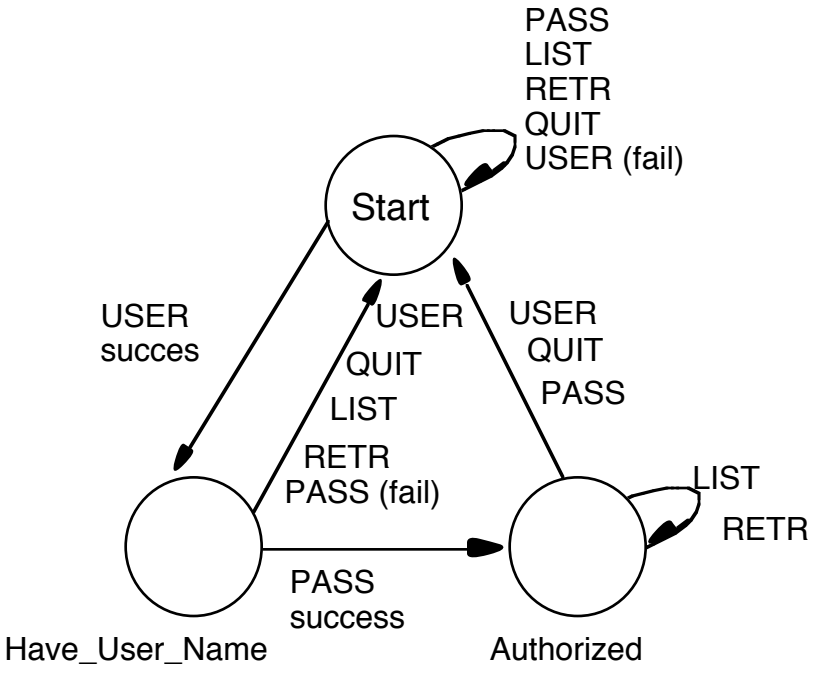
Returns: the mail message indicated by the number

QUIT Command

Arguments: none

Updates mail box to reflect transactions taken during the transaction state, then logs user out

If session ends by any method except the QUIT command, the updates are not done



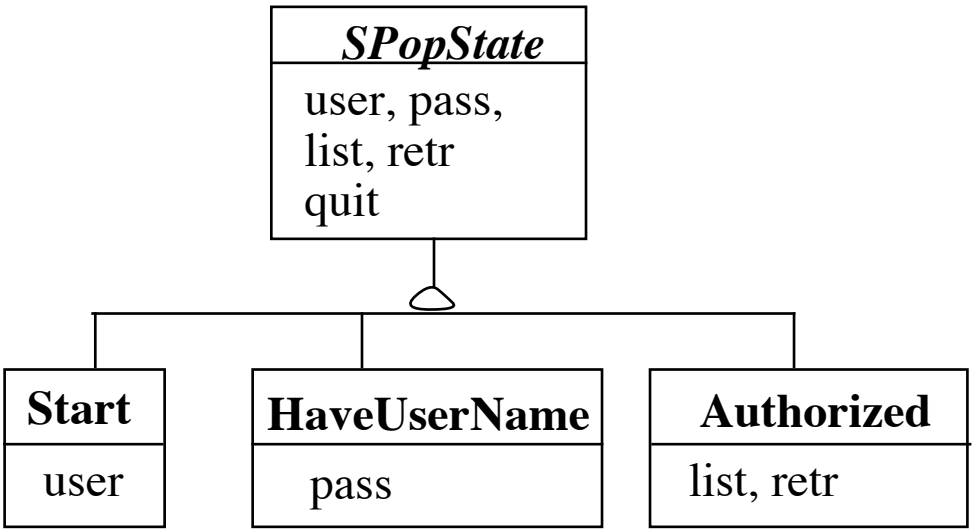
The Switch Statement

```
class SPop
   {
   static final int HAVE_USER_NAME = 2;
   static final int START = 3;
   static final int AUTHORIZED = 4;

   private int state = START;

   String userName;
   String password;

   public void user( String userName ) {
      switch (state) {
         case START:  {
            this.userName = userName;
            state = HAVE_USER_NAME;
            break;
         }
         case HAVE_USER_NAME:
         case AUTHORIZED:{
            endLastSessionWithoutUpdate();
            goToStartState()
            }
         }
      }
```

Implementation with Switch Statement Cont.

```
public void pass( String password )
  {
  switch (state)
    {
    case START:  {
      giveWarningOfIllegalCommand();
      }
    case HAVE_USER_NAME: {
      this.password = password;
      if ( validateUser() )
        state = AUTHORIZED;
      else {
        sendErrorMessageOrWhatEver();
        userName = null;
        password = null;
        state = START;
      }
    case AUTHORIZED: {
      endLastSessionWithoutUpdate();
      goToStartState()
    }
    }
  }
etc.
}
```

Using Polymorphism Implementation




```
class SPop {
   private SPopState state = new Start();

   public void user( String userName ) {
      state = state.user( userName );
   }

   public void pass( String password ) {
      state = state.pass( password );
   }

   public void list( int messageNumber ) {
      state = state.list( massageNumber );
   }

   etc.
```

SPopStates

Defines default behavior

```
abstract class SPopState {
   public SPopState user( String userName ) {
      return goToStartState();
   }

   public SPopState pass( String password ) {
      return goToStartState();
   }

   public SPopState list( int massageNumber ) {
      return goToStartState();
   }

   public SPopState retr( int massageNumber ) {
      return goToStartState();
   }

   public SPopState quit(  ) {
      return goToStartState();
   }

   protected SPopState goToStartState()  {
      endLastSessionWithoutUpdate();
      return new StartState();
   }
}
```

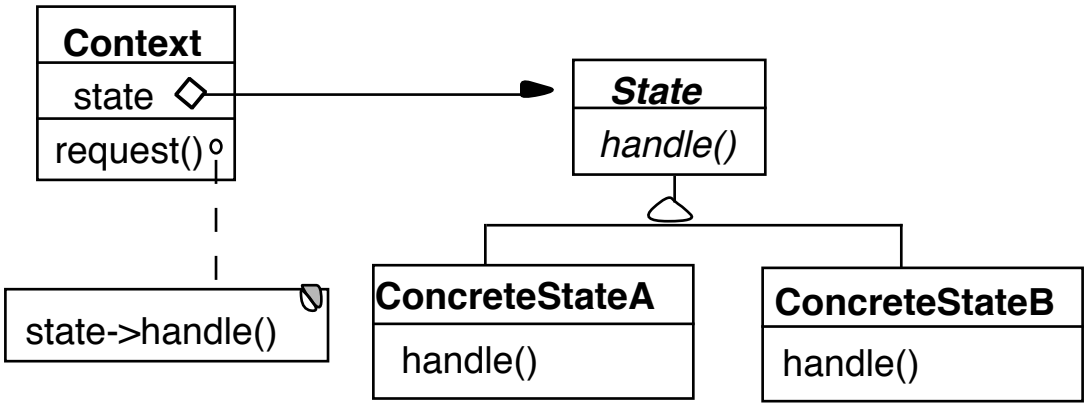
SpopStates - Continued

```
class Start extends SPopState {
   public SPopState user( String userName ) {
      return new HaveUserName( userName );
   }
}

class HaveUserName extends SPopState {
   String userName;

   public HaveUserName( String userName ) {
      this.userName = userName;
   }

   public SPopState pass( String password ) {
      if ( validateUser( userName, password )
         return new Authorized( userName );
      else
         return goToStartState();
   }
}
```

State

Intent

Allow an object to alter its behavior when its internal state changes. The object will appear to change it class.



Applicability

Use the State pattern in either of the following cases:

- An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.

- Operations have large, multipart conditional statements that depend on the object's state. Often, several operations will contain this same conditional structure.

Issues
How much State in the State

In Example:

- SPop is the Context
- SPopState is the abstract State
- Start, HaveUserName are ConcreteStates

All the state & all real behavior is in SPopState & subclasses

This is an extreme example

In general the Context will have data & methods
- Besides State & State methods
- This data will not change states

That is only some aspects of the Context will alter its behavior

Issue
Who defines the state transitions?
The Context

- If the states will be used in different state machines with different transitions

- If the criteria changing states is fixed

```
class SPop
  {
  private SPopState state = new Start();

  public void user( String userName )
    {
    state.user( userName );
    state = new HaveUserName( userName );
    }

  public void pass( String password )
    {
    if ( state.pass( password ) )
      state = new Authorized( );
    else
      state = new Start();
    }
```

Who defines the state transitions?
The State

- More flexible to let State subclasses specify the next state

```
class SPop
   {
   private SPopState state = new Start();

   public void user( String userName )
      {
      state = state.user( userName );
      }

   public void pass( String password )
      {
      state = state.pass( password );
      }

   public void list( int messageNumber )
      {
      state = state.list( massageNumber );
      }
```

IssueSharing State Objects

Multiple contexts (SPops) can use the same state object if the state object has no instance variables

A state object can have no instance variables if:

- The object has no need for instance variables or
- The object stores its instance variables elsewhere

Storing Instance Variables Elsewhere
Variant 1

SPop stores them and passes them to states

```
class SPop
   {
   private SPopState state = new Start();

   String userName;
   String password;

   public void user( String newName )
      {
      this.userName = newName;
      state.user( newName );
      }

   public void pass( String password )
      {
      state.pass( userName , password );
      }
```

Storing Instance Variables Elsewhere
Variant 2

SPop stores them and states get data from SPop

```
class SPop {
   private SPopState state = new Start();

   String userName;
   String password;

   public String userName() { return userName; }

   public String password() { return password; }

   public void user( String newName ) {
      this.userName = newName ;
      state.user( this );
   }

   etc.

class HaveUserName extends SPopState {
   public SPopState pass( SPop mailServer ) {
      String useName = mailServer.userName();
      etc.
   }
}
```

Issue
Creating and Destroying State Objects

Options:

- Create state object when needed, destroy it when it is no longer needed
- Create states once, never destroy them (singleton)

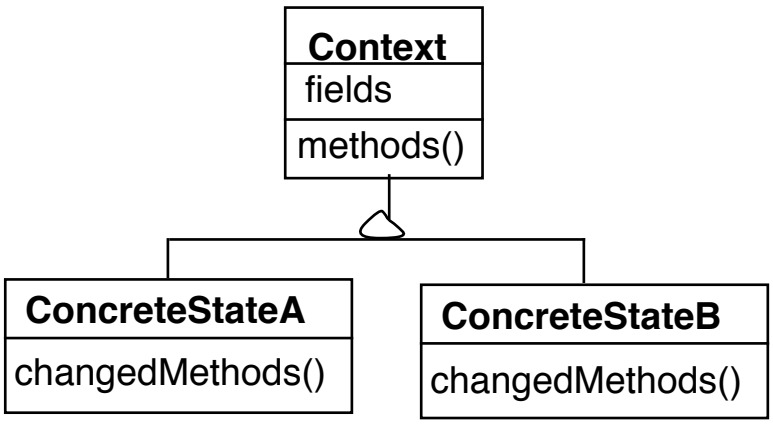
Issue
Changing the Context Class for Real

Some languages allow an object to change its class

- CLOS (Common Lisp Object System)
- Cincom's VisualWorks Smalltalk

```
| context |
context := Start new.
context changeClassTo: HaveUserName.
context changeClassTo: Authorized.
```

So why not forget State pattern and use:



In VisualWorks Smalltalk

- Subclassing and modifying ConcreteState becomes hard

In CLOS the State pattern may not be needed

Consequences

- It localize state-specific behavior and partitions for different states
- It makes state transitions explicit
- State objects can be shared

State Verses Strategy

How to tell the difference

Rate of Change

Strategy

Context object usually contains one of several possible ConcreteStrategy objects

State

Context object often changes its ConcreteState object over its lifetime

Exposure of Change

Strategy

All ConcreteStrategies do the same thing, but differently

Clients do not see any difference in behavior in the Context

State

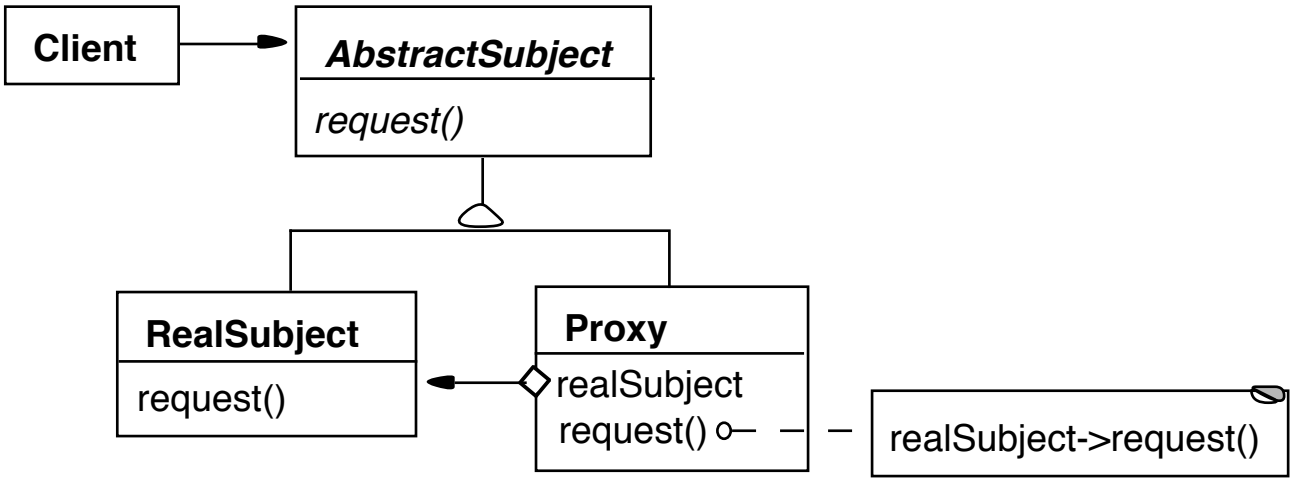
ConcreteState act differently

Clients see different behavior in the Context

Proxy

proxy n. pl prox-ies The agency for a person who acts as a substitute for another person, authority to act for another

Structure



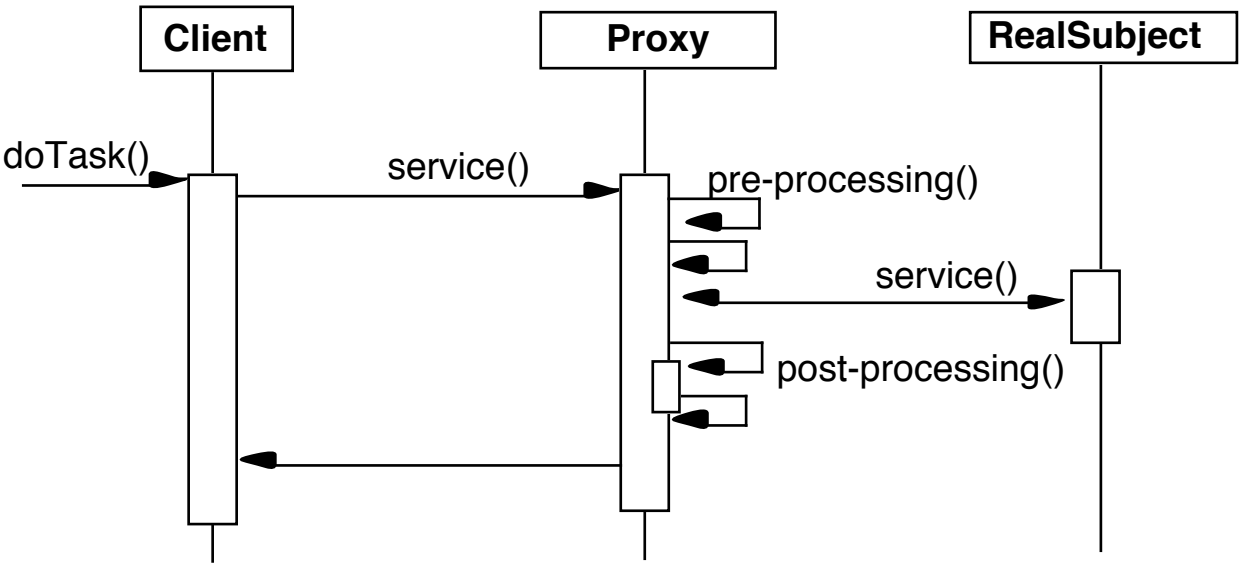
The Pattern

The proxy has the same interface as the original object

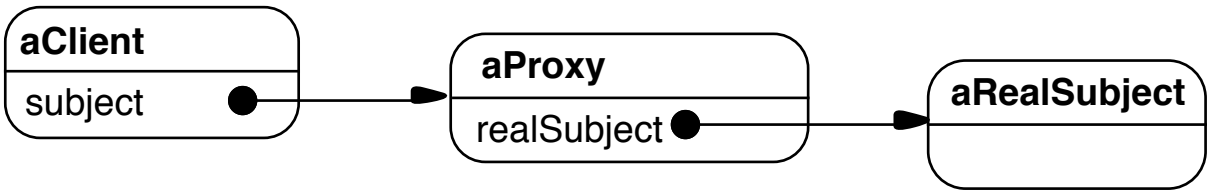
Use common interface (or abstract class) for both the proxy and original object

Proxy contains a reference to original object, so proxy can forward requests to the original object

Dynamics



Runtime Objects



Sample Proxy

```
public class Proxy
  {
  Foo target;

  public float bar(int size )
    {
    preprocess here
    float answer =  target.bar( size);
    postProcess here
    return answer;
    }

  other methods as needed
  }
```

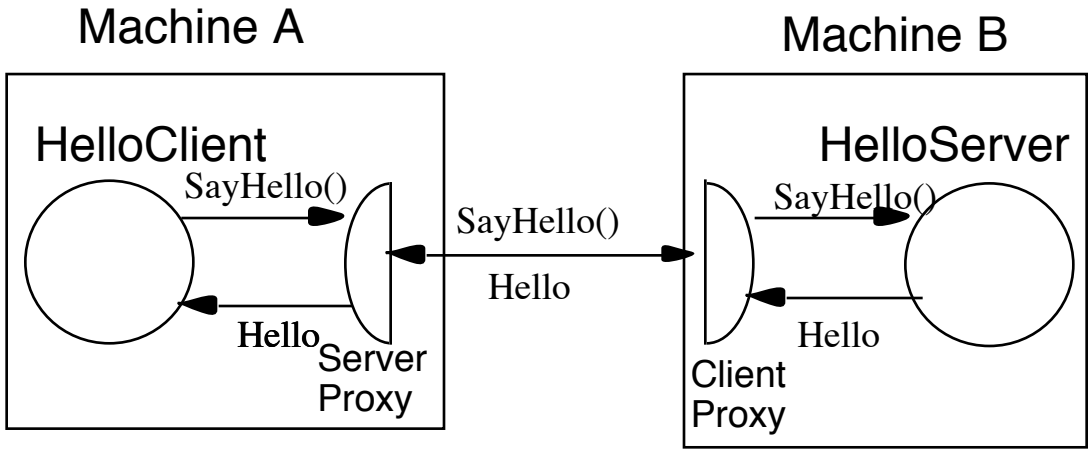
Preprocessing & post-processing depend on purpose of the proxy

Reasons for Object Proxies
Remote Proxy

The actual object is on a remote machine (remote address space)

Hide real details of accessing the object

Used in CORBA, Java RMI




```
public class HelloClient  {
   public static void main(String args[])  {
      try  {
         String server = getHelloHostAddress( args);
         Hello proxy = (Hello) Naming.lookup( server );
         String message = proxy.sayHello();
         System.out.println( message );
      }
      catch ( Exception error)
         {  error.printStackTrace(); }
   }
}
```

Reasons for Object Proxies Continued

Virtual Proxy

- Creates/accesses expensive objects on demand
- You may wish to delay creating an expensive object until it is really accessed
- It may be too expensive to keep entire state of the object in memory at one time

Protection Proxy

- Provides different objects different level of access to original object

Cache Proxy (Server Proxy)

- Multiple local clients can share results from expensive operations: remote accesses or long computations

Firewall Proxy

- Protect local clients from outside world

Synchronization Proxy

- Synchronize multiple accesses to real subject

```
public class Table {
   public Object elementAt( int row, int column ){ blah}

   public void setElementAt(Object element, int row, int column )
      { blah}
}

public class RowLockTable {
   Table realTable;
   Integer[] locks;

   public RowLockTable( Table toLock) {
      realTable = toLock;
      locks = new String[ toLock.numberOfRows() ];
      for (int row = 0; row< toLock.numberOfRows(); row++ )
         locks[row] = new Integer(row);
      }

   public Object elementAt( int row, int column ) {
      synchronized ( locks[row] ) {
         return realTable.elementAt( row, column);
      }
   }

   public void setElementAt(Object element, int row, int column ){
      synchronized ( locks[row] )     {
         return realTable.setElementAt(element, row, column);
      }
   }
}
```

Counting Proxy

Delete original object when there are no references to it

Prevent accidental deletion of real subject

Collect usage statistics

Sample use is making C++ pointer safe

Smalltalk Proxy Tricks

When an object is sent a message

The object's class and the object's class’s superclasses are searched for the method

If the method is not found the object is sent the message:

    doesNotUnderstand:

This method in Object raises an exception

Prototyping of a Proxy

One can use doesNotUnderstand: to implement a pluggable proxy

Example

```
Object subclass: #Proxy
   instanceVariableNames: 'target '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Whitney-Examples'
```

Class Method

```
on: anObject
   ^super new target: anObject
```

Instance Methods

```
doesNotUnderstand: aMessage
   ^target
      perform: aMessage selector
      withArguments: aMessage arguments

target: anObject
   target := anObject
```

Examples of Using the Proxy

```
| wrapper |
wrapper := Proxy on: Transcript.
wrapper open.
wrapper show: 'Hi mom'.

| wrapper |
wrapper := Proxy on: 3.
wrapper + 5.

| wrapper |
wrapper := Proxy on: 'Hi '.
wrapper , ' mom'.
```

Why just for Prototyping

doesNotUnderstand:

- Can be hard to debug
- Is slower than regular message send

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

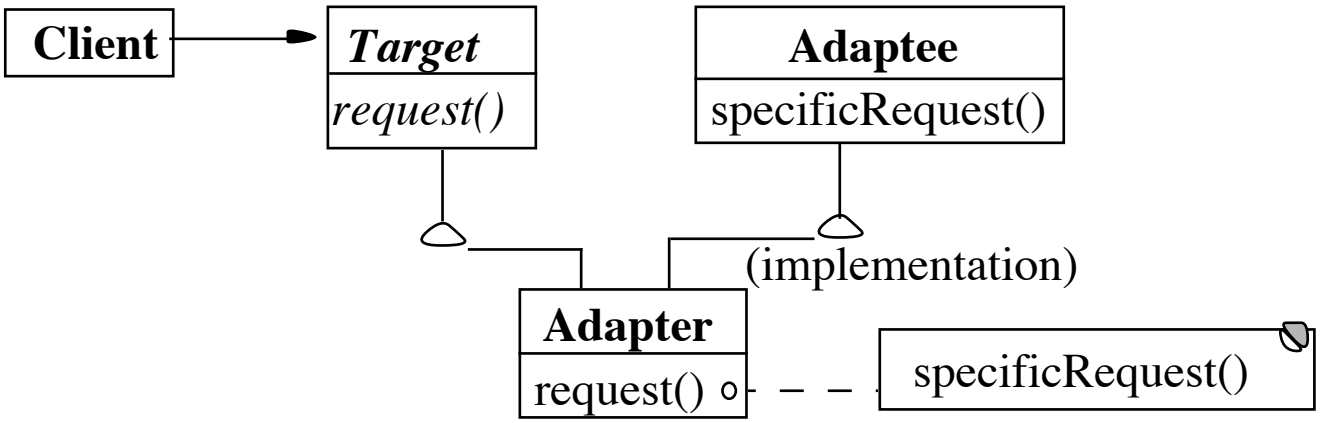
The adapter pattern converts the interface of a class into another interface.

Use the Adapter pattern when

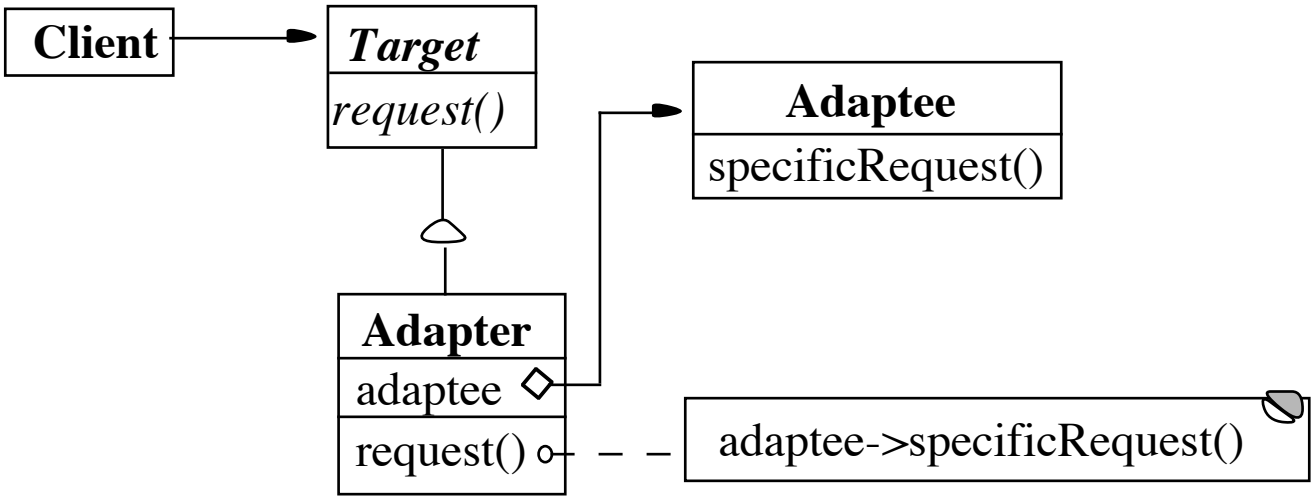
- You want to use an existing class and its interface does not match the one you need
- You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is classes that don't necessarily have compatible interfaces
- You need to use several existing subclasses, but it's impractical 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();
    }
}
```