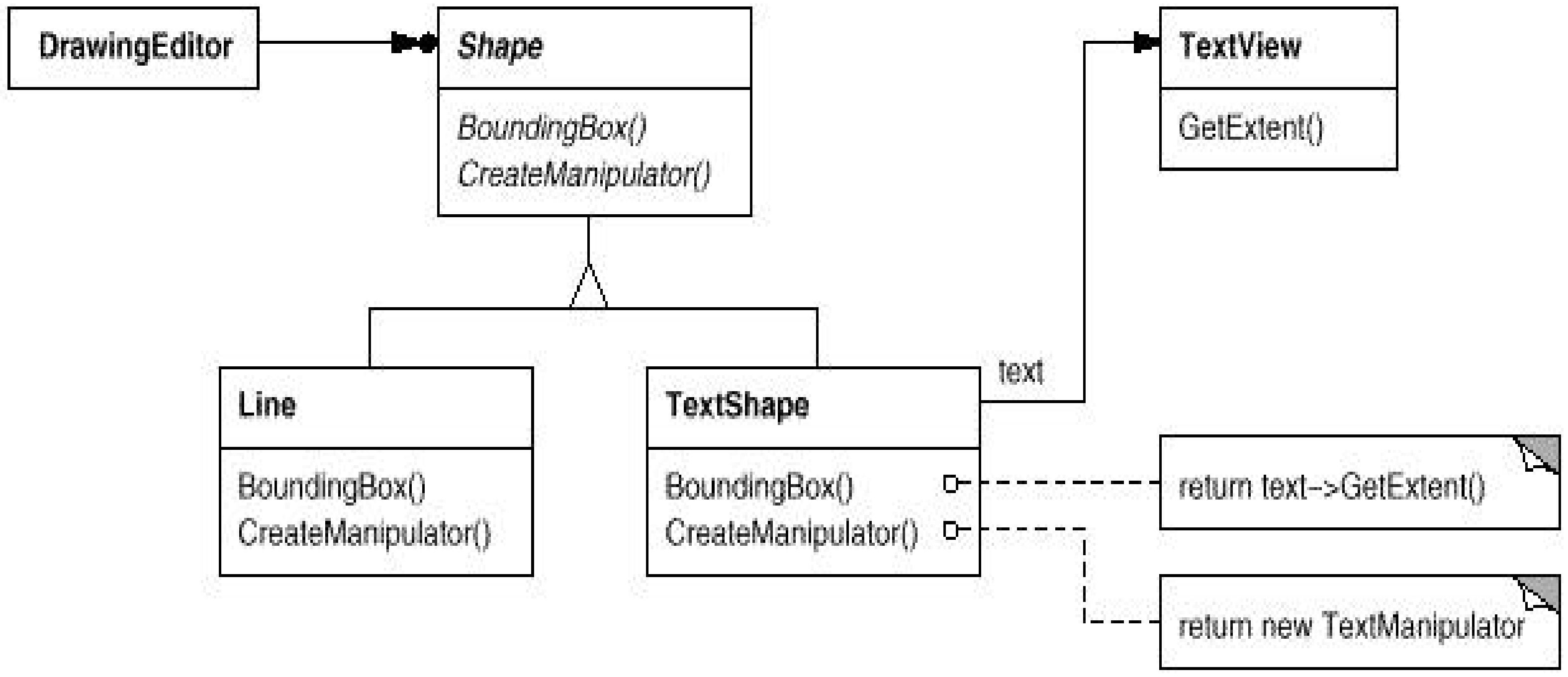
The Adapter Pattern

The Adapter Pattern

- Intent
  - ⇨ Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Also Known As
  - ⇨ Wrapper
- Motivation
  - ⇨ Sometimes a toolkit or class library can not be used because its interface is incompatible with the interface required by an application
  - ⇨ We can not change the library interface, since we may not have its source code
  - ⇨ Even if we did have the source code, we probably should not change the library for each domain-specific application

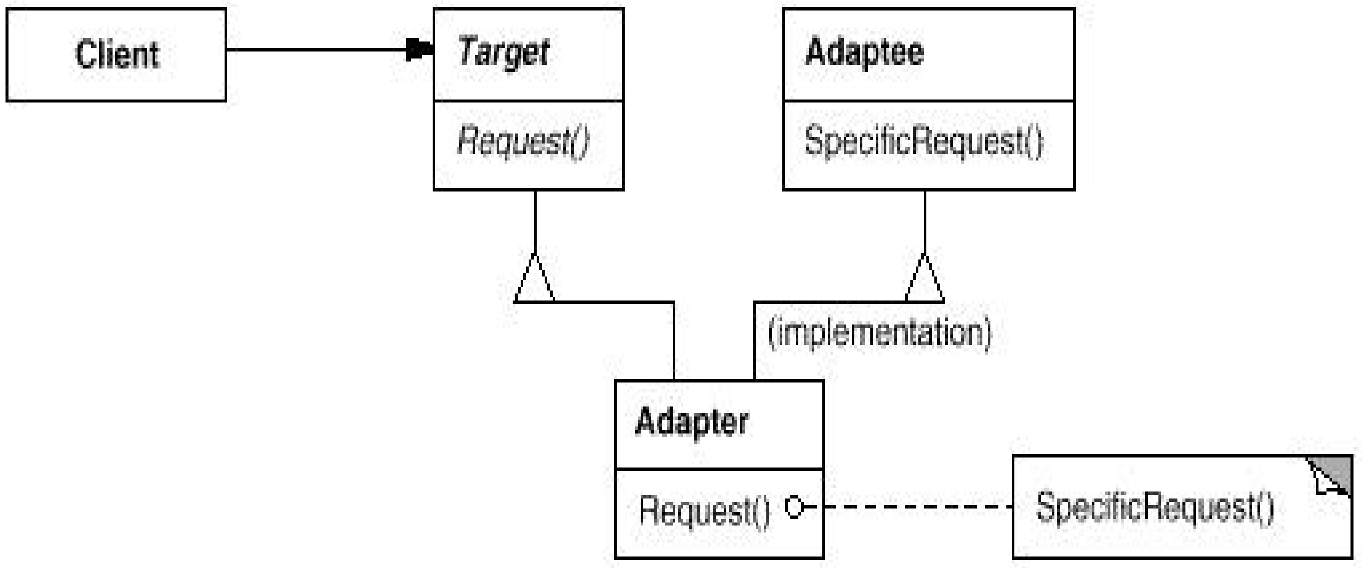
The Adapter Pattern

- Motivation
  - ➩ Example:



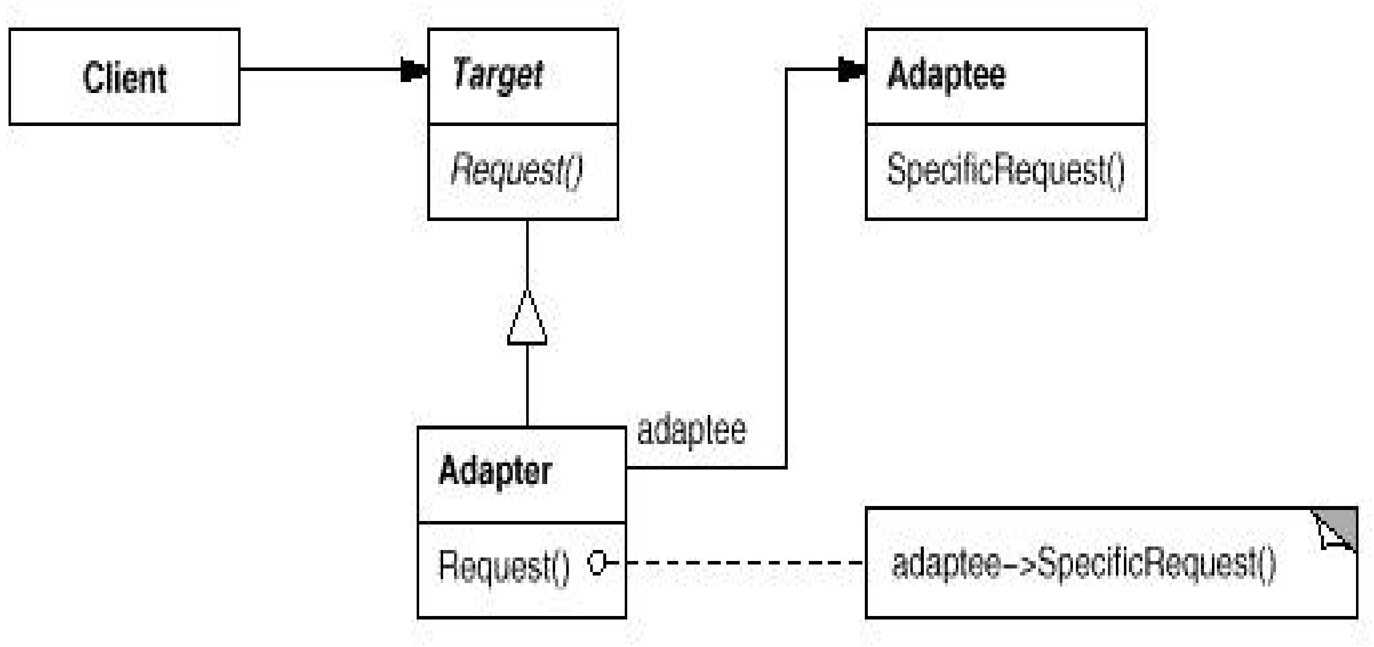
The Adapter Pattern

- Structure
  - ⇨ A class adapter uses multiple inheritance to adapt one interface to another:



The Adapter Pattern

- Structure
  - ⇨ An object adapter relies on object composition:



The Adapter Pattern

- Applicability

  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 classes with incompatible interfaces

- Implementation Issues
  - How much adapting should be done?
    - Simple interface conversion that just changes operation names and order of arguments
    - Totally different set of operations
  - Does the adapter provide two-way transparency?
    - A *two-way adapter* supports both the Target and the Adaptee interface. It allows an adapted object (Adapter) to appear as an Adaptee object or a Target object

Adapter Pattern Example 1

- The classic round pegs and square pegs!
- Here's the SquarePeg class:

```
/**
 * The SquarePeg class.
 * This is the Target class.
 */
public class SquarePeg {
  public void insert(String str) {
    System.out.println("SquarePeg insert(): " + str);
  }
}
```

Adapter Pattern Example 1 (Continued)

- And the RoundPeg class:

```
/**
 * The RoundPeg class.
 * This is the Adaptee class.
 */
public class RoundPeg {
  public void insertIntoHole(String msg) {
    System.out.println("RoundPeg insertIntoHole(): " + msg);
  }
}
```

- If a client only understands the SquarePeg interface for inserting pegs using the insert() method, how can it insert round pegs? A peg adapter!

Adapter Pattern Example 1 (Continued)

- Here is the PegAdapter class:

```
/**
 * The PegAdapter class.
 * This is the Adapter class.
 * It adapts a RoundPeg to a SquarePeg.
 * Its interface is that of a SquarePeg.
 */
public class PegAdapter extends SquarePeg {
  private RoundPeg roundPeg;

  public PegAdapter(RoundPeg peg)  {this.roundPeg = peg;}

  public void insert(String str)  {roundPeg.insertIntoHole(str);}

}
```

Adapter Pattern Example 1 (Continued)

- Typical client program:

```
// Test program for Pegs.
public class TestPegs {
  public static void main(String args[])  {

    // Create some pegs.
    RoundPeg roundPeg = new RoundPeg();
    SquarePeg squarePeg = new SquarePeg();

    // Do an insert using the square peg.
    squarePeg.insert("Inserting square peg...");
```

Adapter Pattern Example 1 (Continued)

```
    // Now we'd like to do an insert using the round peg.
    // But this client only understands the insert()
    // method of pegs, not a insertIntoHole() method.
    // The solution: create an adapter that adapts
    // a square peg to a round peg!
    PegAdapter adapter = new PegAdapter(roundPeg);
    adapter.insert("Inserting round peg...");
  }

}
```

- Client program output:

```
SquarePeg insert(): Inserting square peg...
RoundPeg insertIntoHole(): Inserting round peg...
```

Adapter Pattern Example 2

- Notice in Example 1 that the PegAdapter adapts a RoundPeg to a SquarePeg. The interface for PegAdapter is that of a SquarePeg.
- What if we want to have an adapter that acts as a SquarePeg or a RoundPeg? Such an adapter is called a two-way adapter.
- One way to implement two-way adapters is to use multiple inheritance, but we can't do this in Java
- But we can have our adapter class implement two different Java interfaces!

Adapter Pattern Example 2 (Continued)

- Here are the interfaces for round and square pegs:

```
/**
 *The IRoundPeg interface.
 */
public interface IRoundPeg {
  public void insertIntoHole(String msg);
}


/**
 *The ISquarePeg interface.
 */
public interface ISquarePeg {
  public void insert(String str);
}
```

Adapter Pattern Example 2 (Continued)

- Here are the new RoundPeg and SquarePeg classes. These are essentially the same as before except they now implement the appropriate interface.

```
// The RoundPeg class.
public class RoundPeg implements IRoundPeg {
  public void insertIntoHole(String msg) {
    System.out.println("RoundPeg insertIntoHole(): " + msg);
  }
}

// The SquarePeg class.
public class SquarePeg implements ISquarePeg {
  public void insert(String str) {
    System.out.println("SquarePeg insert(): " + str);
  }
}
```

Adapter Pattern Example 2 (Continued)

- And here is the new PegAdapter:

```
/**
 * The PegAdapter class.
 * This is the two-way adapter class.
 */
public class PegAdapter implements ISquarePeg, IRoundPeg {
  private RoundPeg roundPeg;
  private SquarePeg squarePeg;

  public PegAdapter(RoundPeg peg) {this.roundPeg = peg;}

  public PegAdapter(SquarePeg peg) {this.squarePeg = peg;}

  public void insert(String str) {roundPeg.insertIntoHole(str);}

  public void insertIntoHole(String msg){squarePeg.insert(msg);}
}
```

Adapter Pattern Example 2 (Continued)

- A client that uses the two-way adapter:

```
// Test program for Pegs.
public class TestPegs {
  public static void main(String args[]) {
    // Create some pegs.
    RoundPeg roundPeg = new RoundPeg();
    SquarePeg squarePeg = new SquarePeg();

    // Do an insert using the square peg.
    squarePeg.insert("Inserting square peg...");

    // Create a two-way adapter and do an insert with it.
    ISquarePeg roundToSquare = new PegAdapter(roundPeg);
    roundToSquare.insert("Inserting round peg...");
```

Adapter Pattern Example 2 (Continued)

```
      // Do an insert using the round peg.
      roundPeg.insertIntoHole("Inserting round peg...");

      // Create a two-way adapter and do an insert with it.
      IRoundPeg squareToRound = new PegAdapter(squarePeg);
      squareToRound.insertIntoHole("Inserting square peg...");
    }
  }
```

- Client program output:

```
SquarePeg insert(): Inserting square peg...
RoundPeg insertIntoHole(): Inserting round peg...
RoundPeg insertIntoHole(): Inserting round peg...
SquarePeg insert(): Inserting square peg...
```

Adapter Pattern Example 3

- This example comes from Roger Whitney, San Diego State University
- Situation: A Java class library exists for creating CGI web server programs. One class in the library is the CGIVariables class which stores all CGI environment variables in a hash table and allows access to them via a get(String evName) method. Many Java CGI programs have been written using this library. The latest version of the web server supports servlets, which provide functionality similar to CGI programs, but are considerably more efficient. The servlet library has an HttpServletRequest class which has a getX() method for each CGI environment variable. We want to use servlets. Should we rewrite all of our existing Java CGI programs??

Adapter Pattern Example 3

- Solution : Well, we'll have to do some rewriting, but let's attempt to minimize things. We can design a CGIAdapter class which has the same interface (a get() method) as the original CGIVariables class, but which puts a wrapper around the HttpServletRequest class. Our CGI programs must now use this CGIAdapter class rather than the original CGIVariables class, but the form of the get() method invocations need not change.

Adapter Pattern Example 3 (Continued)

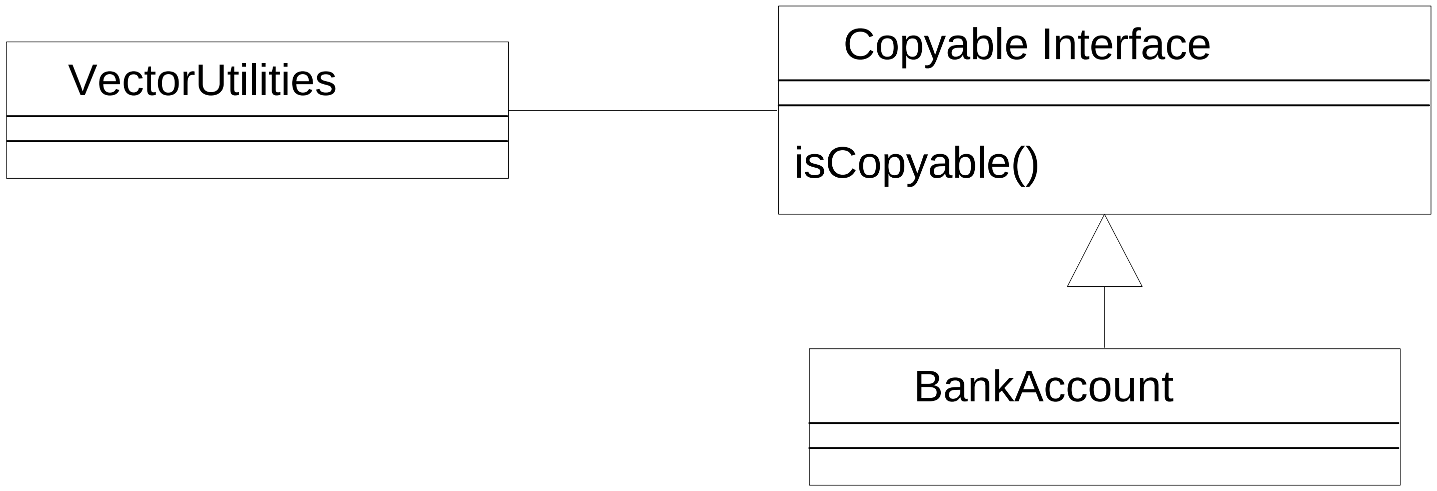
- Here's a snippet of the CGIAdapter class:

```
public class CGIAdapter  {
  Hashtable CGIVariables = new Hashtable(20);

  public CGIAdapter(HttpServletRequest CGIEnvironment) {
    CGIVariables.put("AUTH_TYPE", CGIEnvironment.getAuthType());
    CGIVariables.put("REMOTE_USER",
                     CGIEnvironment.getRemoteUser());
    etc.
  }

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

}
```

- Note that in this example, the Adapter class (CGIAdapter) itself constructs the Adaptee class (CGIVariables)

Adapter Pattern Example 4

- Consider a utility class that has a copy() method which can make a copy of an vector excluding those objects that meet a certain criteria. To accomplish this the method assumes that all objects in the vector implement the Copyable interface providing the isCopyable() method to determine if the object should be copied or not.



Adapter Pattern Example 4 (Continued)

- Here’s the Copyable interface:

```
public interface Copyable {
  public boolean isCopyable();
}
```

- And here’s the copy() method of the VectorUtilities class:

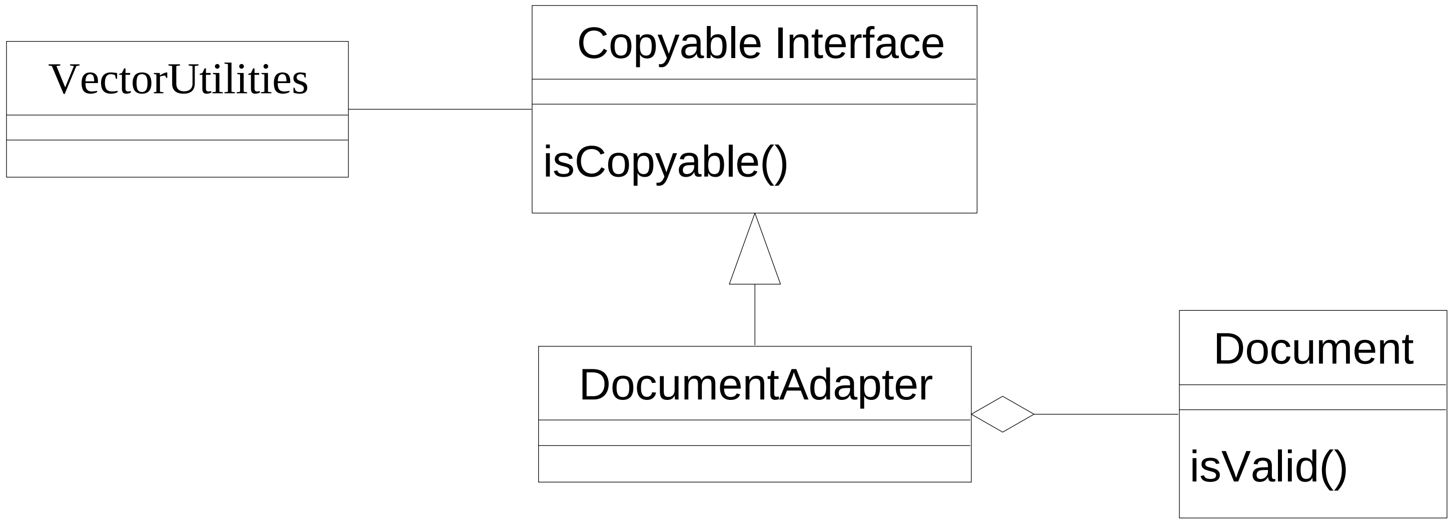
```
public static Vector copy(Vector vin) {
  Vector vout = new Vector();
  Enumeration e = vin.elements();
  while (e.hasMoreElements()) {
    Copyable c = (Copyable) e.nextElement();
    if (c.isCopyable())
      vout.addElemet(c);
  }
  return vout;
}
```

Adapter Pattern Example 4 (Continued)

- But what if we have a class, say the Document class, that does not implement the Copyable interface. We want to be able perform a selective copy of a vector of Document objects, but we do not want to modify the Document class at all. Sounds like a job for (TA-DA) an adapter!
- To make things simple, let's assume that the Document class has a nice isValid() method we can invoke to determine whether or not it should be copied

Adapter Pattern Example 4 (Continued)

- Here's our new class diagram:



Adapter Pattern Example 4 (Continued)

- And here is our DocumentAdapter class:

```
public class DocumentAdapter implements Copyable {

  private Document d;

  public DocumentAdapter(Document d) {
    document = d;
  }

  public boolean isCopyable() {
    return d.isValid();
  }

}
```

Adapter Pattern Example 5

- Do you see any Adapter pattern here?

```
public class ButtonDemo {

  public ButtonDemo() {
    Button button = new Button("Press me");
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
         doOperation();
      }
    });
  }
  public void doOperation() { whatever }

}
```

Adapter Pattern Example 5 (Continued)

- Button objects expect to be able to invoke the actionPerformed() method on their associated ActionListener objects. But the ButtonDemo class does not have this method! It really wants the button to invoke its doOperation() method. The anonymous inner class we instantiated acts as an adapter object, adapting ButtonDemo to ActionListener!
- Recall that there are some AWT listener interfaces that have several methods which must be implemented by an event listener. For example, the WindowListener interface has seven such methods. In many cases, an event listener is really only interested in one specific event, such as the Window Closing event.

Adapter Pattern Example 5 (Continued)

- Java provides “adapter” classes as a convenience in this situation. For example, the WindowAdapter class implements the WindowListener interface, providing “do nothing” implementation of all seven required methods. An event listener class can extend WindowAdapter and override only those methods of interest
- And now we see why they are called adapter classes!