Asynchronous Networking & Custom Protocols

Topics

- Datagrams
- Multicast
- HTTP
- Text-based Custom Protocols
- State Pattern
- Adapter Pattern

Datagram

- The basic unit of information passed across TCP/IP network
  - Source and destination addresses
  - Data
- UDP sends and receives datagrams

Socket

- Socket - end of a pipe that can send and receive data from host
  - Pair of sockets create a pipe
- Browser & Server dialog
  - Connect to www.washington.edu:80
    - Ask for connection - get a socket
  - Send request
  - Receive response
  - Close socket

Multicast

- "Broadcast" a datagram to any number of other computers in a group
- A group is specified by:
  - A class D IP address and a specified port
    - `224.0.0.0` to `239.255.255.255` (`224.0.0.0` reserved)
  - A port

Java Networking

- `java.net`
  - Addressing
  - Datagrams
  - Sockets
- `java.io`
  - Standard IO streams are used with sockets

InetAddress

- **Represents host (IP)**
  - `getAddress()`
  - `getHostAddress()`
  - `getHostName()`
- **Static lookup methods**
  - `getLocalhost()`
  - `getByName( String hostname )`
  - `getAllByName( String hostname )`

DatagramPacket

- **Represents a datagram**
  - get/setAddress()
  - get/setPort()
  - get/setData()
  - get/setLength()

DatagramSocket

- **A socket for sending and receiving datagram packets**
  - `send( DatagramPacket p )`
  - `receive( DatagramPacket p )`
  - `connect( InetAddress addr, int port )`
  - `close()`
  - `getInetAddress()`
  - `getPort()`
  - `getLocalAddress()`
  - `getLocalPort()`

UdpEchoServer

```
public class UdpEchoServer {
  private int mPort;
  public UdpEchoServer( int port ) {
    mPort = port;
  }

  public void start() {
    DatagramSocket udpSock = null;
    try {
      udpSock = new DatagramSocket( mPort );
      byte[] buf = new byte[1024];
      DatagramPacket packet = new DatagramPacket( buf, buf.length );
      while( true ) {
         udpSock.receive(packet);
         String msg = new String(packet.getData(), packet.getOffset(),
                                                   packet.getLength());
         msg = msg.toUpperCase();

        packet.setData(msg.getBytes());
         packet.setLength(msg.length());
         udpSock.send(packet);
      }
```

UdpEchoServer

```
  } catch( IOException ex ) {
        System.err.println( "Server error: " + ex );
   } finally {
     if( udpSock != null ) udpSock.close();
   }
}
```

UdpEchoClient

```
public class UdpEchoClient {
  private String mIpAddress;
  private int mPort;

  public UdpEchoClient( String ipAddress, int port ) {
    mIpAddress = ipAddress;
    mPort = port;
  }

  public void start()
  {
    DatagramSocket sock = null;
    try {
      sock = new DatagramSocket();
      byte[] buf = new byte[1024];
      DatagramPacket packet;
      packet = new DatagramPacket( buf, buf.length,
                    InetAddress.getByName(mIpAddress), mPort );
      BufferedReader br = new BufferedReader(
                           new InputStreamReader( System.in ) );
```

UdpEchoClient

```
      while( true ) {
         System.out.print("Enter string to be echoed ('q' to quit): ");
         String line = br.readLine();
         if( "q".equals(line) ) break;
            byte[] bytes = line.getBytes();
            packet.setData(bytes);
            packet.setLength(bytes.length);
            sock.send( packet );
            sock.receive( packet );
            System.out.println("Echo: " +
                (new String(packet.getData(), 0, packet.getLength())));
      }
    } catch( IOException ex ) {
      System.out.println( "Server error: " + ex );
    } finally {
      if( sock != null ) {
        sock.close();
      }
    }
  }
```

MulticastSocket

- **Supports multicast transmissions**
- **Extends** DatagramSocket
  - joinGroup(InetAddress multicastAddr)
  - leaveGroup(InetAddress multicastAddr)
  - send(DatagramPacket p)
  - setTimeToLive(int ttl)

TimeServer

```
public class TimeServer {
  private static final int ONE_SECOND = 1000;
  private String mIpAddress;
  private int mPort;

  public TimeServer( String ipAddress, int port ) {
    mIpAddress = ipAddress;
    mPort = port;
  }
```

TimeServer

```
public void start() {

  MulticastSocket multiSock = null;
   try {
      InetAddress group = InetAddress.getByName(mIpAddress);
      multiSock = new MulticastSocket();
      multiSock.joinGroup(group);
      byte[] buf = new byte[256];
      DatagramPacket packet = new DatagramPacket(buf, buf.length,
                                                 group, mPort);
      System.out.println("Server ready...");
      while (true) {
         String ds = (new Date()).toString().trim();
         byte[] bytes = ds.getBytes();
         packet.setData(bytes);
         packet.setLength(bytes.length);
         multiSock.send(packet);
         Thread.currentThread().sleep(ONE_SECOND);
      }
```

TimeServer

```
      } catch (IOException ex) {
         System.out.println("Server error: " + ex);
      } catch (InterruptedException ex) {
         System.out.println("Server error: " + ex);
      } finally {
         if (multiSock != null) {
            multiSock.close();
      }
   }

}
```

TimeClient

```
public class TimeClient {
  private String mIpAddress;
  private int mPort;

  public TimeClient( String ipAddress, int port ) {
     mIpAddress = ipAddress;
     mPort = port;
  }

  public void start() {
     MulticastSocket multiSock = null;
     try {
        InetAddress group = InetAddress.getByName( mIpAddress );
        multiSock = new MulticastSocket( mPort );
        multiSock.joinGroup( group );
        byte[] buf = new byte[128];
        DatagramPacket packet = new DatagramPacket( buf, buf.length );
```

TimeClient

```
      System.out.println( "Get 10 time messages..." );
      for( int i=0; i < 10; i++ ) {
         multiSock.receive( packet );
         System.out.println( "TimeOfDay: " +
            (new String(packet.getData(), 0, packet.getLength())));
      }
      multiSock.leaveGroup(group);
    }
    catch( IOException ex ) {
      System.err.println( "Server error: " + ex );
    }
    finally {
      if( multiSock != null ) multiSock.close();
    }
  }
```

Socket - Review

- **Represents a TCP/IP socket**
- **Supports connection oriented communications**
  - `getInputStream()`
  - `getOutputStream()`
  - `close()`
  - `shutdownInput()`
  - `shutdownOutput()`

ServerSocket - Review

- **A listening socket waiting for a "normal" socket to connect**
- **Accepts connection, providing a new socket for communications**
  - `Socket accept()`

Protocol

- Stateless protocol
  - Server doesn't keep the state for a browser
  - Connection is closed after every request
- Stateful protocols
  - Server maintains state information
    - Typically an open connection is maintained
    - May use embedded information in protocol

Protocol

- Other well known protocols
  - HTTP Hyper-Text Transfer Protocol
  - FTP – File Transfer Protocol
  - SMTP – Simple Mail Transfer Protocol
  - POP3 – Post Office Protocol (version 3)
  - IMAP – Internet Message Access Protocol
  - IIOP – Internet Inter-ORB Protocol
  - JRMP – Java Remote Method Protocol

Java Networking

- Doesn't provide support for most protocols, except:
  - HTTP
  - IIOP
  - JRMP
- Implementation of protocol is your job

Universal Resource Locator

- Convenient form of locating resources
  - http://www.washington.edu:80/index.html
    - Protocol is HTTP
    - Host is www.washington.edu
    - Port is 80
    - Resource is index.html
  - Other URLs
    - rmi://192.0.2.24:8099/myServer
    - ftp://localhost

Universal Resource Locator

- `java.net.URL` **and** `URLConnection`
  - **Provide support for forming URLs**
- `URLDecoder`, `URLEncoder`
  - **Utilities for manipulating URL resource component**
- `URLStreamHandler`
  - **Handle streaming resource specified by URL**
  - **Convenient way of implementing URL resources**

URLConnection

- **Abstract class, basics of connection to a resource**
- **Concrete subclasses**
  - HttpURLConnection
    - **Provides support for HTTP 1.0**
    - **Proxy setting properties**
      - proxySet
      - proxyHost
      - proxyPort
  - JarURLConnection
    - jar:http://host/file.jar!/package/classname.class
      - **Represents a combination URL**

HTTP Example

```
.
.
.
/*
Properties sysProp = System.getProperties();
sysProp.put( "proxySet", "true" );
sysProp.put( "proxyHost", "www-proxy.foo.com" );
sysProp.put( "proxyPort", "35000" );
*/

URL url = new URL( "http://www.washington.edu" );
URLConnection conn = url.openConnection();
// actually an HttpURLConnection
InputStream in = conn.getInputStream();
InputStreamReader rdr = new InputStreamReader( in );
.
.
.
```

Custom Text-Based Protocols

- Support clients and servers written in any language
- Client and server exchange text messages
- Each is responsible for parsing and interpreting
  - Internet standard end of line marker "/r/n"

Custom Text-Based Protocols

- **Tools for parsing**
  - `StringTokenizer`
  - `String.split`
  - `BufferedReader`
- **Always flush after writing to socket**

Defining the Protocol

- Protocol may be simple
  - First token identifies command or operation
  - Remaining tokens provide data for operation
    - `quote:IBM`
- Or slightly more complex
  - A set of name value pairs – properties
    - `oper=quote`
    - `symbol=IBM`
  - This is how HTTP is designed (mostly)
- May also be very complex

Break

Server Implementation

```
...
ServerSocket servSock = new ServerSocket(80);
while(true)
{
   Socket sock = servSock.accept(); // will block
   CHttpRequest req = new CHttpRequest(sock);
   (new Thread(req)).start();
}
...
```

```
public class CHttpRequest implements Runnable
{
   public void run()
   {
      PrintWriter out =
         new PrintWriter(sock.getOutputStream(),true);
      out.println("<HTML> my web page … </HTML>");
   }
}
```

Server Implementation

- Clients don't have to wait
- Still haven't implemented protocol
- Let's change our server
  - Commands
    - `loveme`, `hateme`, `why` and `quit`
    - Server maintains state
    - Client simply prints responses

Server Implementation

```
public class Request implements Runnable
{
   public void run()
   {
      InputStreamReader in =
             new InputStreamReader(sock.getInputStream());
      outStrm = sock.getOutputStream();
      PrintWriter wrtr = new PrintWriter(outStrm, true);


     ...

      while(fMore)
      {
         String command = in.readLine();
         if( command.equals("loveme") )
            wrtr.println("I love you!");
         ...
      }
   }
```

Server Implementation

- **Maintaining state**

```
public class RequestState
{
   String state = “Don’t know”;
   public String loveme()
   {
      state = “Because you are wonderful!”;
      return “I love you!”;
   }
   . . .
   public String why()
   {
      return state;
   }
}
```

Server Implementation

```
public class Request implements Runnable
{
   RequestState rs = new RequestState();
   public void run()
   {  ...
      while(fMore)
      {
         String command = in.readLine();
         if( command.equals("loveme") )
         {
            out.println( rs.loveme() );
         }
         else if( command.equals("why") )
            out.println( rs.why() );
         ...
      }
   }
```

Server Implementation

- **Can also use reflection**
  - `loveme` **command can correspond to** `loveme()`

```
public class Request implements Runnable
{
   RequestState rs = new RequestState();
   public void run()
   {
      ...
      while(fMore)
      {
         rs.getClass().
            getMethod(command, null).invoke(rs,null);
         ...
      }
   }
```

Client Implementation

- **Must send commands**
  - `loveme`, `why`, `hateme` **and** `quit`
  - **Using strings**
- **Essentially those commands actually execute against the state object**
- **What if client could just call** `rs.loveme()`

Abstracting Server Implementation

```
public interface RequestState {
    public String loveme();
    public String why();
    public String hateme();
    public String quit();
}

public class Request implements RequestState {
    String state = “Don’t know”;
    public String loveme() {
        state = “Because you are wonderful!”;
        return “I love you!”;
    }
    . . .
    public String why() {
        return state;
    }
```

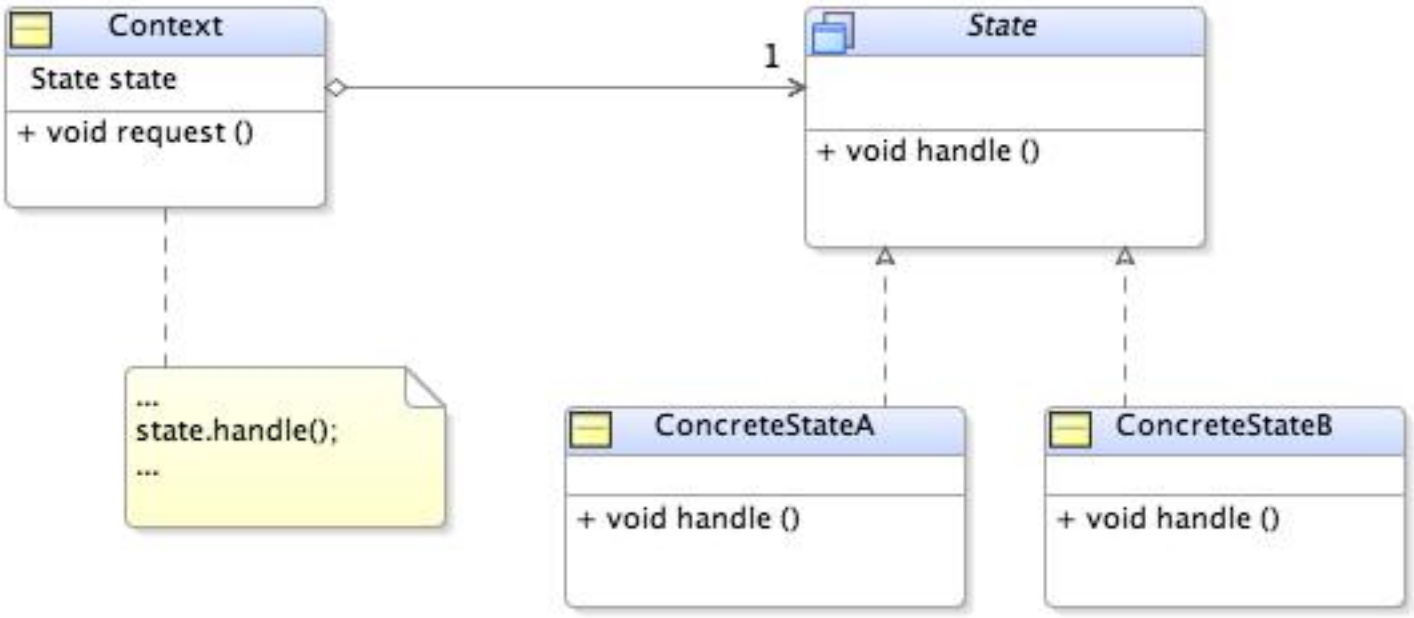
Abstracting Client Implementation

```
public class RequestStub implements RequestState
{
   public String loveme()
   {
      ...
      // send request to the server
      out.println("loveme");
      String response = in.readLine();
      ...
      return response;
   }
}
```

- Note that client and server implement the same interface - this is key

Another Abstraction
State Pattern

- Implement each state's behavior in it's own class



State Pattern

```
public interface RequestState
{
   public RequestState loveme( PrintWriter out );
   public RequestState hateme( PrintWriter out );
   public RequestState why( PrintWriter out );
   public RequestState quit( PrintWriter out );
}
```

State Pattern

```
public class IntroState implements RequestState {
   public RequestState loveme( PrintWriter out ) {
      out.println( "I love you." );
      return new CLoveMeState();
   }
   public RequestState hateme( PrintWriter out ) {
      out.println( "I don't hate you." );
      return new CHateMeState();
   }
   public RequestState why( PrintWriter out ) {
      out.println( "Why what?" );
      return this;
   }
   public RequestState quit( PrintWriter out ) {
      out.println( "Bye." );
     return null;
   }
}
```

State Pattern

```
public class Request implements Runnable
{
   RequestState rs = new IntroState();
   public void run()
   {  ...
      while( rs == null )
      {
         String command = br.readLine();
         if( command.equals( "loveme" ) )
            rs = rs.loveme( wrtr );
         ...
      }
   }
```

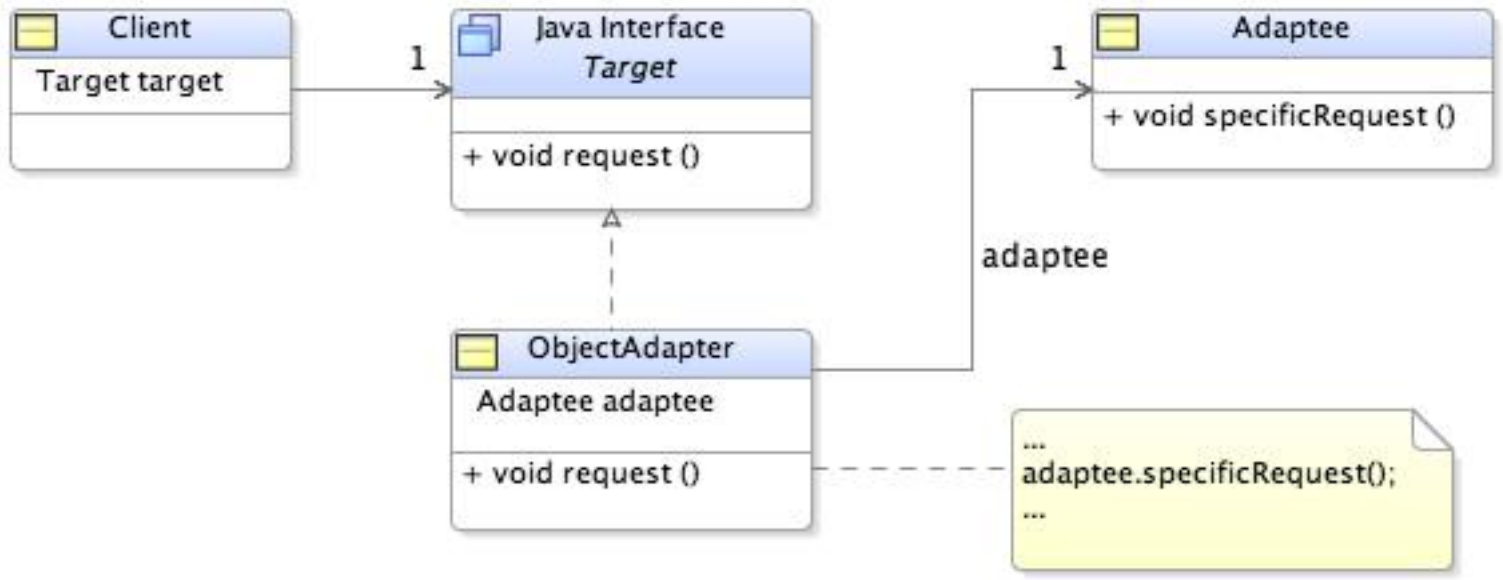
Adapter Problem

- A component implements the behavior we want
- The component provides an interface different than required
- Analogous to electrical outlets
  - Different interfaces
    - USA
    - Europe
  - Adapters are available

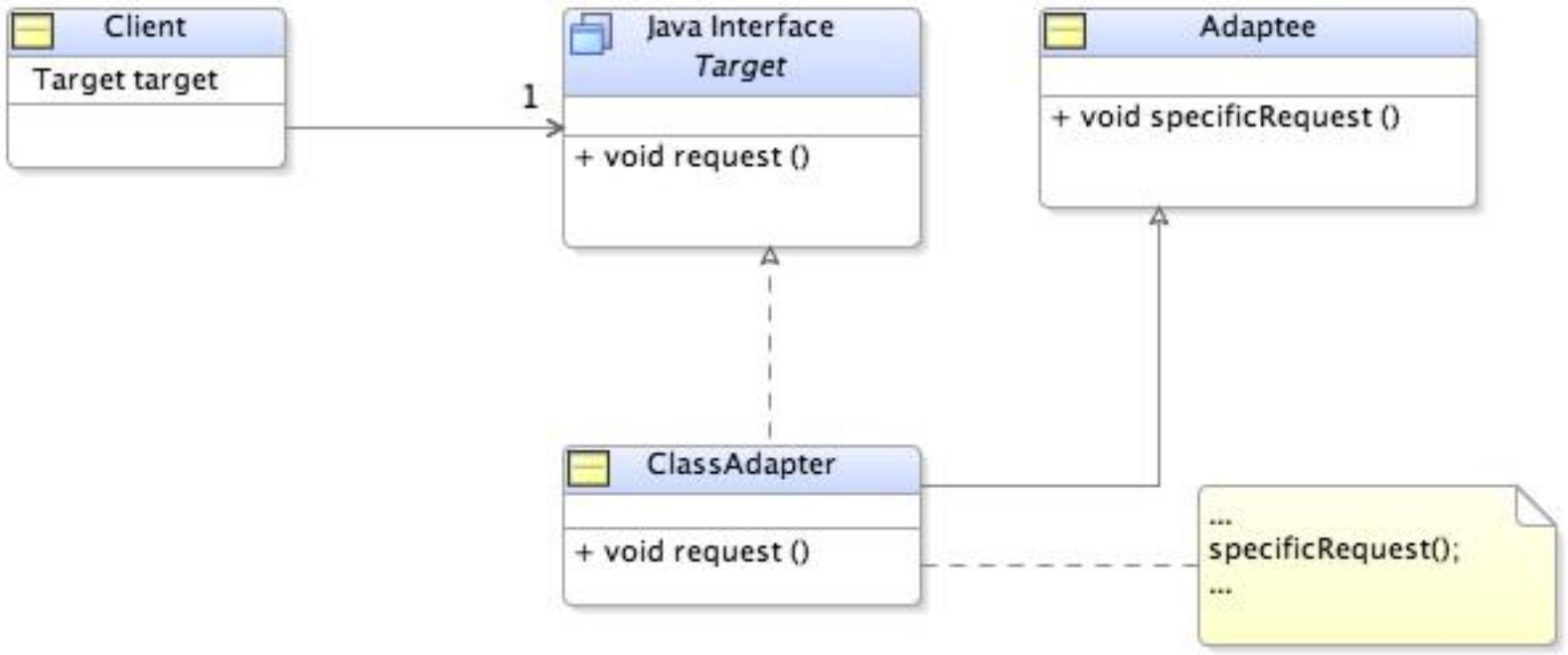
Adapter Solution

- The Adapter Patterns provides software "adapters"
- Two varieties
  - Class adapters
    - Uses multiple (or interface) inheritance to convert one interface to another
  - Object adapters
    - Uses composition to convert one interface to another

Adapter Structure (object)



Adapter Structure (class)



Adapter Solution

- Defining characteristics
  - Class adapters realize the target interface and inherits from the adapted class
  - Object adapters realize the target interface and delegates to the adapted class
  - Methods implemented in the adapter to satisfy target interface invoke methods of the adapted class

Adapter Solution

- **Discussion**
  - **Adapts one interface to another by placing an intermediary between them**
  - **Class and object patterns are complimentary not competitors**

Adapter Type		Problem
Class	Object	
No	Yes	Adapt a set of classes all having the same interface
No	Yes	Adapt to a final class
Yes	No	Simplifies overriding adapted class behavior