Object Oriented Programming and Design in Java

Session 22
Instructor: Bert Huang
Announcements

• Homework 5 due last day of class: Mon. May 3rd

• Mon. May 3rd: Final review

• Mon. May 10th, Final exam. 9 AM - noon
  • closed-book/notes, focus on post-midterm material, but material is inherently cumulative
Review

- Threadsafe wrappers for Collections
- Leftover Design Patterns
  - ADAPTER
  - COMMAND
  - FACTORY METHOD
  - PROXY
  - SINGLETON
  - VISITOR
Today’s Plan

- VISITOR pattern
- Networking
- Socket and ServerSocket classes
- Simple text-chat example program
Programming Patterns

MVC

VISITOR

COMPOSITE

PROXY

DECORATOR

COMMAND

ADAPTER

STRATEGY

TEMPLATE-METHOD

FACTORY-METHOD

SINGLETON
Pattern: Visitor

• You're building a hierarchy of classes, and you want to allow new functionality
• but don't want to have clients modify code
• STRATEGY is inadequate if new functionality depends on concrete types
• e.g., file system: DirectoryNode and FileNode
• want to allow client to add operations, e.g., printing operation, disk-space computation
VISITOR

- An object structure contains element classes of multiple types, and you want to carry out operations that depend on the object types
- The set of operations should be extensible over time
- The set of element classes is fixed

Solution
- Define a *visitor* interface that has methods for visiting elements of each of the given types
- Each element class defines an *accept* method that invokes the matching element visitation method on the visitor parameter
- To implement an operation, define a class that implements the visitor interface type and supplies the operation's action for each element type
Double Dispatch

• This pattern uses polymorphism twice to make code very general
  • 1st, element.accept() calls Visitor method based on type of element
  • 2nd, the Visitor method performs operation based on type of Visitor
• Both actions called through interfaces
• Concrete classes need not be known at runtime
Example Visitor

- FileSystem
  - accept()
  - visitFileNode()
  - visitDirectoryNode()

- FileNode
  - accept()
  - Calls visitFileNode

- Directory Node
  - accept()
  - Calls visitDirectoryNode
Double Dispatch in FileSystemNode

client

: Directory Node

: Print Visitor

accept

visitDirectoryNode

Polymorphic selection of node type

Polymorphic selection of visitor type
Programming Patterns

- MVC
- COMPOSITE
- DECORATOR
- VISITOR
- STRATEGY
- TEMPLATE-METHOD
- ADAPTER
- PROXY
- SINGLETON
- FACTORY-METHOD
- COMMAND
Networking

- Modern computing is done over the Internet
- This includes, but is far more general than the World Wide Web and websites transferred over http
- Data over the internet is divided into two types of raw information: application data and network protocol data
- Network protocol data tells the routers, switches and computers where the data came from, where it's headed, how to check for errors, lost data, etc.
The Internet Protocol (IP)

- Computers on the Internet have IP addresses, which are four byte numbers, like 128.59.48.24 (www.columbia.edu)
- Domain Name Servers (DNS) map these IP addresses to easier-to-remember names
- IP transmits data in small chunks known as *packets*, which contain validation information so the receiving computer can tell if the data was corrupted
- If data is corrupted or lost, IP doesn't say what to do about it, which is why many Internet communications also use *Transmission Control Protocol* (TCP)
TCP/IP

- Transmission Control Protocol retries packet transmission if there is a failure
- Programs can abstract away transmission details using TCP/IP
- The protocol is responsible for reliable transmission (or elegantly notifies of an error)
- A lot of details, but the important pieces of a TCP/IP packet are:
  - Sender's IP address and port
  - Receiver's IP address and port
Ports

- Since most computers only have one or two network connections, must distinguish between messages from different programs

- TCP/IP packets have 16-bit number (0, 65535) called a *port*, indicates what program should handle it

- 80 is the Web, 22 is ssh, 993 is IMAP/SSL

- Ports 0-1023 restricted, but we can use the larger numbers for our programs
On each side of a TCP/IP connection, there is a *socket*

Java lets us abstract away the details and work with the sockets

One end is a *server* and the other is a *client*

Each socket has an InputStream and an OutputStream
Socket API

- Common constructor:
  ```java
  new Socket(String host, int port)
  ```
- `InputStream getInputStream()`
- `OutputStream getOutputStream()`
- `void close()`
- IOExceptions everywhere; lots of try/catch blocks in Socket code
- But how do we set up the host? `ServerSocket`
ServerSocket API

- **Constructor:**
  
  ```java
  new ServerSocket(int port)
  ```

- **Socket accept()** // Listens for a connection to be made and accepts it

- **close();**

- **Get Sockets using accept() and perform logic with them using their InputStream and OutputStream**

- **Usually, wrap with**
  ```java
  new BufferedReader(socket.getInputStream())
  ```

  **and**
  ```java
  new PrintWriter(socket.getOutputStream())
  ```
Simple Two-Way Text Chatting

- TextServer - establishes connection, responsible for exception handling
- TextClient - establishes connection, responsible for exception handling
- TwoWayChatServer - reads input from console and sends it to client, prints messages from client
- TwoWayChatClient - reads input from console and sends it to server, prints messages from server
public class TextServer {

    /**
     * Constructor takes a port number and 
     * opens a single-connection server on that port
     * @param port port to listen on
     */

    public TextServer(int port) {
        try {
            server = new ServerSocket(port);

            clientSocket = server.accept();
            out = new PrintWriter(
                clientSocket.getOutputStream(), true);
            in = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
        }
        catch (IOException e) {
            System.err.println("Error opening server streams");
            System.exit(-1);
        }
    }

    /**
     * Reads a line sent by the client
     * @return the line sent by client
     */

    public String readLine() {
        try {
            return in.readLine();
        }
        catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }

    /**
     * Writes a line to the client
     * @param line String to send to client
     */

    public void writeLine(String line) {
        out.println(line);
    }

    /**
     * Accessor for BufferedReader
     * @return BufferedReader representing input from client
     */

    public BufferedReader getReader() {
        return in;
    }

    /**
     * @return whether server is open
     */

    public boolean isConnected() {
        return !server.isClosed();
    }

    /**
     * Closes the connection to client
     */

    public void closeClient() {
        try {
            clientSocket.close();
        }
        catch (IOException e) { e.printStackTrace(); }
    }

    /**
     * Closes the server
     */

    public void closeServer() {
        try {
            server.close();
        }
        catch (IOException e) { e.printStackTrace(); }
    }

    private PrintWriter out;
    private BufferedReader in;
    private ServerSocket server;
    private Socket clientSocket;
}

TextServer.java
/**
 * Reads a line sent by the client
 * @return the line sent by client
 */

public String readLine()
{
    try {
        try {
            return in.readLine();
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}

/**
 *Writes a line to the client
 * @param line String to send to client
 */

public void writeLine(String line) { out.println(line); }

/**
 * Accessor for BufferedReader
 * @return BufferedReader representing input from client
 */
public class TextServer {

   public TextServer (int port) {
      try {
         server = new ServerSocket(port);
         clientSocket = server.accept();
         out = new PrintWriter(clientSocket.getOutputStream(), true);
         in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
      }
      catch (IOException e) {
         System.err.println("Error opening server streams");
         System.exit(-1);
      }
   }

   public String readLine() {
      try {
         return in.readLine();
      }
      catch (IOException e) {
         e.printStackTrace();
         return null;
      }
   }

   public void writeLine(String line) {
      out.println(line);
   }

   public BufferedReader getReader() {
      return in;
   }

   public boolean isConnected() {
      return !server.isClosed();
   }

   public void closeClient() {
      try {
         clientSocket.close();
      }
      catch (IOException e) { e.printStackTrace(); }
   }

   public void closeServer() {
      try {
         server.close();
      }
      catch (IOException e) { e.printStackTrace(); }
   }

   private PrintWriter out;
   private BufferedReader in;
   private ServerSocket server;
   private Socket clientSocket;
}
TextClient.java
Constructor

```java
public TextClient(String hostname, int port)
{
    try {
        clientSocket = new Socket(hostname, port);

        out = new PrintWriter(
            clientSocket.getOutputStream(), true);

        in = new BufferedReader(
            new InputStreamReader(
                clientSocket.getInputStream()));
    }
    catch (IOException e) {
        System.err.println("Error opening server streams");
        System.exit(-1);
    }
}
```
TwoWayChatServer.java

Main Loop

TextServer server = new TextServer(port);
server.writeLine("Connected to server");

// start back-and-forth chatting
String line;
while((line = server.readLine()) != null) {
    System.out.println(line);
    try {
        server.writeLine(stdin.readLine());
    } catch (IOException e) {
        e.printStackTrace();
    }
}
TextClient client = new TextClient(hostname, port);

// start back-and-forth chatting
String line;
while ((line = client.readLine()) != null)
{
    System.out.println(line);
    try {
        client.writeLine(stdin.readLine());
    } catch (IOException e) {
        e.printStackTrace();
    }
}

client.close();
Multithreading

- These programs work, but the conversation must alternate back and forth between the client and server
- We need multithreading to allow remote messages to be displayed immediately while waiting for System.in input
- ThreadedBufferedReaderPrinter - Runnable: continually prints output from BufferedReader ASAP
- ThreadedChatServer - reads input from console and sends it to client, starts TBRP thread
- ThreadedChatClient - reads input from console and sends it to server, starts TBRP thread
public class ThreadedBufferedReaderPrinter implements Runnable {

    /**
     * Constructor takes the BufferedReader to print
     * @param reader the BufferedReader to print
     */
    public ThreadedBufferedReaderPrinter(BufferedReader reader) {
        this.reader = reader;
    }

    public void run() {
        String line;
        try {
            while (!Thread.interrupted() &&
                    (line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    BufferedReader reader;
}
ThreadedChatClient
Main Loop

// hostname and port loaded

TextClient client = new TextClient(hostname, port);

// Start printing thread
Thread t = new Thread(new ThreadedBufferedReaderPrinter(client.getReader()));
t.start();

// start chatting
while (client.isConnected()) {
    try {
        client.writeLine(stdin.readLine());
    } catch (IOException e) {
        e.printStackTrace();
    }
}
// port loaded

TextServer server = new TextServer(port);
server.writeLine("Connected to server");

// Start printing thread
Thread t = new Thread(new 
    ThreadedBufferedReaderPrinter(server.getReader()));
t.start();

// start chatting
while (server.isConnected()) {
    try {
        server.writeLine(stdin.readLine());
    } catch (IOException e) {
        e.printStackTrace();
    }
}

ThreadedMultiChatServer

- Handle multiple connections with threads
  - while (true)
    - accept connection
    - start thread to handle connection
- Multiple clients can connect to the chat server
- Each client managed by a thread, when any client sends a message, bounce to all connected clients
- Store client OutputStreams in a List, all client-handling threads share the list
public class MultiChatHandler implements Runnable {

    public MultiChatHandler(BufferedReader reader,
            List<PrintWriter> outputs, InetAddress addr)
    {
        this.reader = reader;
        this.outputs = outputs;
        name = addr.toString();

        printAll("A new client connected.");
    }

    public void run() {
        while (!Thread.interrupted()) {
            String line = null;
            try {
                line = reader.readLine();
            } catch (IOException e) {
                e.printStackTrace();
            } System.out.println(line);
            printAll(line);
        }
    }

    private void printAll(String line)
    {
        for (PrintWriter pw : outputs)
            pw.println(name + ": "+ line);
    }
}

MultiChatHandler
public class MultiChatHandler implements Runnable {
    public MultiChatHandler(BufferedReader reader, List<PrintWriter> outputs, InetAddress addr) {
        this.reader = reader;
        this.outputs = outputs;
        name = addr.toString();
        printAll("A new client connected.");
    }

    public void run() {
        while (!Thread.interrupted()) {
            String line = null;
            try {
                line = reader.readLine();
            } catch (IOException e) {
                e.printStackTrace();
            }
            System.out.println(line);
            printAll(line);
        }
    }

    /**
     * Print something to all connected clients
     * @param line
     */
    private void printAll(String line) {
        for (PrintWriter pw : outputs) {
            pw.println(name + ": " + line);
        }
        System.out.println(line);
    }

    BufferedReader reader;
    List<PrintWriter> outputs;
    String name;
}
List<PrintWriter> allOut = new ArrayList<PrintWriter>();

while(true) {
    try {
        Socket client = server.accept();
        allOut.add(new PrintWriter(client.getOutputStream(), true));
        BufferedReader in = new BufferedReader(
            new InputStreamReader(client.getInputStream()));

        Thread t = new Thread(new
            MultiChatHandler(in, allOut, client.getInetAddress()));
        t.start();
    } catch (IOException e) {
        System.err.println("Error connecting client.");
    }
}

ThreadedMultiChatServer Main Loop
Reading

• Horstmann Ch. 10 for Patterns

• http://java.sun.com/docs/books/tutorial/networking/sockets/index.html

• Optional: Ch. 22.1-22.4 in Big Java by Horstmann if you still have it from 1004

• http://www.cs.columbia.edu/~bert/courses/1007/code/networking/