

Project 2

CMSC 433
Programming Language Technologies and Paradigms
Fall 2002

Due Sunday, October 13, 2002

Introduction

For updates to this writeup, see Section 4.

Increasing numbers of applications—include cell phones, routers, and industrial control applications—are choosing to use HTTP, HTML, and other WWW protocols to perform operations like configuration, control, query, and more. In this assignment, we will build a server that uses the web protocols to communicate. Rather than fixing the server to serve files, as happens for most webservers, we will abstract away from what is being served, and focus on the server infrastructure itself. This way, the same code can be used to write web-based servers for files, router configuration, industrial machine control, and so on.

This project has two main goals:

- To familiarize you with some object-oriented *design patterns*. In particular, you will be implementing the Template pattern, the Observer pattern, and the Proxy pattern in this assignment. In addition, you will make use of the Typesafe Enum pattern, the Iterator pattern, and the Decorator pattern (by virtue of using Java I/O).
- To gain experience with using/maintaining your existing code. In particular, we will utilize the Log library from the first project.

A secondary goal is to gain familiarity with HTTP and the WWW client/server approach. This knowledge is relevant to the way applications are written today.

All provided Java files for this assignment can be found at <http://www.cs.umd.edu/class/fall2002/cmssc433-0201/hw2/java/>. You can also get this assignment in PDF and postscript. If you wish, you can use our implementation of Log rather than your own; we will send out our implementation to all students via e-mail.

1 Basic Webserver

The first thing you must do is write the basic webserver. The general structure of the webserver will be just as with any server: there will be a infinite loop

that listens with a `ServerSocket` on a given port, accepts TCP connections, and then processes requests on those connections. Each time you get a connection, you will do the following:

- *Parse the HTTP request.* Connections are made to web servers using the *HyperText Transfer Protocol*, or HTTP. To parse this request, you will use a class we have provided, `HttpGetRequestSimple`, and call its static `readRequest` method with the connection's `InputStream` and `OutputStream` as arguments. The result will be that it returns an `HttpRequest` object, that stores the key aspects of an HTTP request: the requested file, the HTTP version number, and operation; for this project, only the requested file will be of interest. If something goes wrong, an error message is sent to the client on the `OutputStream`, and `readRequest` will return `null`.
- *Invoke all relevant handlers on the request.* The webserver will have a collection of *handlers* that have been registered to deal with web requests. You will invoke each handler with the `HttpRequest` object and an `StringBuffer` as arguments. These handlers will then process the request and provide output data on the string buffer. This is the key part of the server implementation, and will be explained in more detail below.
- *Send the HTTP response.* Assuming all relevant handlers completed successfully, you will combine all of their output into a single message and create an `HttpResponse` object, using the `HttpResponse` class's static `successResponse` method. If something goes wrong in the process, you will create an `HttpResponse` object using the `errorResponse` method instead. In both cases, the Strings provided to these methods should be HTML (for a primer on HTML, see <http://www.htmlprimer.com/>). After the `HttpResponse` object is created, it can be sent to the client using its `sendResponse` method on the client's output stream. At this point, you will close the connection.

The key part of the implementation is the calling of handlers to process the request. This constitutes an instance of the Observer pattern. That is, a number of handlers, which are the *observers*, will attach to the server, which is the *subject*, indicating that they are interested in processing web events. When web events come in, the server will *update* each interested handler with the information so that it can act and send a response. In our case, the *update* is a call to the handler's `handleRequest` method, described below. The responses of each of the invoked handlers are combined and a single HTTP response that is sent to the client for display.

1.1 The `HttpHandler` class

Each handler will be a subclass of the abstract class `HttpHandler`:

```
public abstract class HttpHandler implements Comparable {
    protected HandlerOrder orderConstraint;
```

```

public int compareTo(Object that) { ... }
public boolean handleRequest(HttpServletRequest request,
                             StringBuffer out)
    throws HandlerException
{
    if (matchRequest(request)) {
        writeResponse(out);
        return true;
    }
    return false;
}
public abstract boolean isExclusive();
protected abstract boolean matchRequest(HttpServletRequest request);
protected abstract void writeResponse(StringBuffer out)
    throws HandlerException;
}

```

To properly subclass this class, you will have to implement its three abstract methods: `isExclusive`, `matchRequest`, and `writeResponse`:

- `isExclusive` returns a boolean indicating whether or not it is an “exclusive” handler. That is, it should not be invoked if another exclusive handler has already been invoked for the same request. Likewise, no other exclusive handler should be invoked once this one is. Non-exclusive handlers can be invoked at any time. Making sure that things work this way is the responsibility of the server (i.e. it is not encapsulated in the `Handler` object itself).
- `matchRequest` is used to determine whether this handler is interested in responding to the given request. It is given the `HttpServletRequest` object as an argument, and it can look at the contents of the object to decide whether or not it needs to act. If it plans to write a response (using `writeResponse`, next), it may need to store some information for use in writing that response.
- Finally, `writeResponse` will write HTML to the given output buffer, with the intention of displaying it at the client side. It is assumed that this method will only be called if `matchRequest` has been called first. If legal output could not be generated due to an error, then `writeResponse` should throw the exception `HandlerException` (see Figure 1.1). This exception contains two fields, a string describing the problem in plain-text, and an HTML error message to send back to the client. The server will use the plain-text message for its own accounting, and will send the HTML part back to the client.

The methods `matchRequest` and `writeResponse` participate in the Template pattern. That is, `Handler` defines a method `handleRequest`, that takes an `HttpServletRequest` and an `StringBuffer` as its arguments, which calls these two

```

public class HandlerException extends Exception {
    private String htmlMsg;
    public HandlerException(String msg, String htmlMsg) {
        super(msg);
        this.htmlMsg = htmlMsg;
    }
    public String getHtmlMsg() { return htmlMsg; };
}

```

Figure 1: The `HandlerException` class

methods in the proper way. When the server implements the Observer pattern, it will need to store a collection of `HttpHandlers`. When it iterates over this collection, it will invoke the `handleRequest` method of each handler, which will in turn invoke the appropriate `matchRequest` and `writeResponse` methods. These methods are listed as `protected` because they should not be invoked directly, but instead only by `handleRequest` or perhaps a subclass of `HttpHandler`.

So that handlers can coexist properly, they may need to be called in a particular order. For this reason, the `HttpHandler` class implements the `Comparable` interface, so that handlers can be sorted. This way, the server can invoke each handler in sorted order. The `compareTo` function, implemented in `HttpHandler`, uses the public field `orderConstraint` to determine its order. This field is of class `HandlerOrder`, which implements the Typesafe Enum pattern (see Bloch's *Effective Java* book, pps. 104–114). In particular, a handler can have `orderConstraint` `EARLY`, `NO_PREFERENCE`, or `LATE`. By sorting the handlers and running them in sorted order, the server will run `EARLY` handlers first, then `NO_PREFERENCE` handlers, and finally `LATE` handlers.

Also, when updating (that is, invoking the `handleRequest` method of) each of the `HttpHandlers`, the server should take care to respect the `isExclusive` status of each handler. That is, it should keep track of whether an exclusive handler has been run, and if so, not run any other exclusive handlers (non-exclusive handlers can be run at any time).

Finally, when implementing the update part of the Observer pattern in the server, you should use the Iterator pattern. That is, you should store handlers in some abstract structure that can be iterated over. This is easy to do using the provided Java libraries, such as the Container classes. Storing your handlers in a `Collection`, and then iterating over them using the `Collection`'s `iterator` method (to return an `Iterator`), will easily satisfy this requirement.

1.2 Handling Errors

If `handleRequest` throws the exception `HandlerException` (see Figure 1.1), an error has occurred. As mentioned above, this exception contains two fields, a string describing the problem in plain-text, and an HTML error message to send back to the client. The server will use the plain-text message for its own

accounting, and will send the HTML part back to the client. Processing should stop at that point, and the error created (using `HttpResponse.errorResponse`) and sent.

In addition, you must deal with the case in which no handler generates any output. This could happen for two reasons: either no handler matched the request (all calls to `handleRequest` returned false), or some handlers matched the request, but no output was generated (that is, the `StringBuffer` used by these handlers remained empty). In the first case, you should send back an error message, using `HttpResponse.errorResponse` saying `Failed to match request requestName` where `requestName` was the client's request. In the second case, send back a normal message using `HttpResponse.successResponse` saying `No output generated for request requestName`.

1.3 The BasicServer class

Now that we have described what you must implement, here is the class you must implement that will encapsulate this functionality:

```
public class BasicServer {
    public BasicServer(int port);
    public void attach (HttpHandler handler);
    public void detach (HttpHandler handler);
    public void run ();
}
```

The constructor takes as an argument the TCP port that the server will listen on for connections. The `attach` method is used to “register” a handler with the server, and the `detach` method is used to remove a registered handler. Finally, the `run` method is used to actually start the server; it will contain the infinite server loop that accepts connections on the given port, parses the HTTP request, invokes the handlers, and sends the response.

This class will be used by particular servers you will build, described in Section 3.

2 Handlers

Handlers are classes that encapsulate the part of any server that varies. That is, the abstract functionality provided by any server is roughly the same: the server accepts connections, parses requests, and sends some responses. However, different servers handle different kinds of requests and have different corresponding responses. For example, a web server parses HTTP messages containing URL's and often provides HTML pages as responses for viewing in a browser, while an NFS file server parses NFS requests and responds with the appropriate files. A time server might always return the current time for a requested timezone.

Handlers define the part of each server that differs. In our project, we will always use HTTP as the means of communication, but rather than just respond

with HTML files, we might respond with something different, depending on the request (we'll describe this idea more in Section 2.1).¹ This way, we can build HTTP-based servers that do different things by reusing the `BasicServer` class, but attaching different `HttpHandlers`.

The class `HttpHandler` is an abstract class, as described above, so the handlers you implement will be subclasses of it. Each handler you write will have the following form:

```
public class MyHandler extends HttpHandler {
    public MyHandler(java.util.regex.Pattern pat,
                    HandlerOrder orderConstraint,
                    ...);

    public boolean isExclusive();
    protected boolean matchRequest(HttpRequest request);
    protected void writeResponse(StringBuffer out)
        throws HandlerException;
}
```

That is, the constructor will take as arguments a pattern to match against, along with an order constraint to indicate when it should be processed relative to other handlers. The pattern will be used in your implementation of the `matchRequest` method to determine if the given `HttpRequest` has a `RequestName` that is of interest to your handler. The `...` indicates that you may need to pass other arguments to the constructor depending on the handler that you write. These additional arguments will be objects that constitute the handler's state. You will then have to implement the three abstract methods of `HttpHandler` as described above.

You may have noticed that if you did things this way, you would duplicate a lot of code. That is, the code that implements the constructor and `matchRequest` will be largely the same across different handlers you might write, since it all is based on regular expression matching. Therefore, if you wish you can create an abstract subclass of `HttpHandler` called `EventHandler`, which will serve as the parent for your own handlers, rather than `HttpHandler` itself. `EventHandler` can implement general methods that won't change across handlers, such as `matchRequest` and `isExclusive` (assuming your other handlers will subclass `EventHandler`):

```
public abstract class EventHandler extends HttpHandler {
    private java.util.regex.Pattern pattern;
    private boolean exclusiveFlag;
    public EventHandler (java.util.regex.Pattern pat,
                        HandlerOrder orderConstraint,
                        boolean exclusiveFlag) { ... }
    public boolean isExclusive() { return exclusiveFlag; }
    protected boolean matchRequest(HttpRequest request) { ... }
```

¹In this project, the format of the output will always be HTML.

```
        protected abstract void writeResponse(StringBuffer out)
            throws HandlerException;
    }
```

Note that each handler's `matchRequest` might have to capture some state from the `HttpRequest` object for use in the `writeResponse` method. As such, you may have to override `EventHandler`'s `matchRequest` method when subclassing it.

2.1 Handler classes you will implement

Once you have implemented the web server, you need to implement some handlers to use it. These handlers are described below, roughly listed in the following way:

1. The name of the handler class, which roughly describes what it does.
2. The pattern that you must match against in order to run the handler. This is a regular expression that matches the `requestName` of the `HttpRequest` object given to `matchRequest`. Sometimes no pattern (or only part of a pattern) will be given, in the case the handler should get the pattern in its constructor.
3. Whether or not the handler is exclusive. Note that exclusivity is particular to the functionality of the handler, while the ordering that handlers are run is particular to the application that uses them.
4. A description of what the handler does, including what its legal inputs are, and how to format its output to send back to the client's web browser.

Here are the handlers:

1. `GetFileHandler`. Matches `"getFile/*.*`" and is exclusive. Call the part of the `requestName` that matches the `.*` of the pattern *filename*. That is, if the request was `"getFile/foo.html"` then *filename* is `foo.html`. The handler will complete successfully if *filename* ends with the suffix `.html` and if a file named *filename* exists in the server's current directory (that is, the directory that the server was started in). In this case, the `GetFileHandler` will return the contents of that file. Otherwise it will return an error (by invoking `HttpResponse.errorResponse` with an appropriate indication of what went wrong).
2. `LogEventHandler`. Matches `".*"` and is nonexclusive. Stores the string representation of the `HttpRequest` object it receives in a `Log` (as implemented in project 1). The `Log` object to use is specified in the constructor for `LogEventHandler`.
3. `GetRecordsHandler`. First, consider the constructor to this class:

```

public GetRecordsHandler extends ... {
    public GetRecordsHandler(Log log, String logName,
                             HandlerOrder orderConstraint) { ... }
    ...
}

```

The match pattern for this handler is then “getRecords/*logName*/*”, where *logName* is the string passed to the constructor above. The *.** part of the match pattern is called the *filter*. This handler is exclusive.

The intent of this handler is to extract and print out all of the records from the Log passed to the constructor that match the *filter*, which is expected to itself be a regular expression. For example, if the request was “getRecords/HTTPlog/methodEntry.*”, and if there was a `GetRecordsHandler` created with `logName` “HTTPlog”, then since this handler will have match pattern “getRecords/HTTPlog/*” it will run and thereby extract all of the records from log that match the pattern `methodEntry.*`.

For this to work, you should change the implementation of your LocalLog class so that if a window size of 0 is given, it will return all records in the log in an array. You can then find all of the records in that array that match the pattern.

Once the handler extracts the records, it needs to send printable versions back. You should do this as an HTML table. In particular, you should simply format each `LogRecord` as a row in the table, delimited by `<tr><td></td></tr>` tags. For example:

```

<table>
<tr><td>Mon Aug 26 13:51:50 EDT 2002 / #1 / Entering: downheap</td></tr>
<tr><td>Mon Aug 26 13:51:50 EDT 2002 / #2 / Exiting: downheap</td></tr>
</table>

```

This defines a two-row table. You can then pass this string to `HttpResponse.successResponse`, to create an `HttpResponse` object to then send to the client. For more on HTML tables, see <http://www.htmlprimer.com/lesson6.shtml>.

4. `RemoteGetFileHandler`. Matches “remoteGetFile/*” and is exclusive. This handler acts as a proxy for a second web server. The part of the pattern that matches *.** is the *url*. `RemoteGetFileHandler` will execute an HTTP request on the remote server indicated by *url*. It returns to the client browser whatever it receives from the second web server. For example, if the request was “remoteGetFile/www.cs.umd.edu/index.html”, then `RemoteGetFileHandler` would open a connection to `http://www.cs.umd.edu/index.html`, and download the page. It would then send this page back to its own sender. If the request cannot be completed for some reason (say, if the *url* is ill-formed, or you cannot connect to the remote site, among others), and appropriate error message should be returned using `HttpResponse.errorResponse`.

In order to implement `RemoteGetFileHandler`, you should look into the `java.net` package, particularly the `URL` class and the `URLConnection` class.

5. `StartTimerHandler`. For each `StartTimerHandler` object, there will be a `EndTimerHandler` object, described below. The idea is that `StartTimerHandler` will start a timer, and then later, following the execution of some notable event, an `EndTimerHandler` will run that stops the timer and notes the elapsed time.

Each pair of handlers shares the same state: a `Timer` object, which you must implement:

```
public class Timer {
    public void startTimer() { ... }
    public void endTimer() { ... }
    public long elapsedTimeMS() { ... }
}
```

When you call `startTimer`, it gets the current time, and stores it in some private field. When `endTimer` is called, it also grabs the current time. Finally, when `elapsedTimeMS` is called, it returns the difference, in milliseconds, between the starting time and the ending time. To get the current time, you can use `java.util.Date`, or `System.currentTimeMillis`, or whatever you wish. The result of `elapsedTimeMS` must be in milliseconds, however.

The `StartTimerHandler`'s match condition will be set by its constructor (see below), and it is nonexclusive. When a `StartTimerHandler` matches a request, it will call `startTimer` on its `Timer` object. The match condition and the `Timer` object to use should be provided in the constructor:

```
public StartTimerHandler extends ... {
    public StartTimerHandler(Timer timer, String pattern,
                             HandlerOrder orderConstraint) { ... }
    ...
}
```

6. `EndTimerHandler`. Its match condition will be set by its constructor (see below), and it is nonexclusive. It also takes a `Timer` object in its constructor, and a `Log` object:

```
public EndTimerHandler extends ... {
    public EndTimerHandler(Timer timer, String pattern, Log log,
                             HandlerOrder orderConstraint) { ... }
    ...
}
```

When a `EndTimerHandler` matches a request, it will call `endTimer` on its `Timer` object. It then will get the elapsed time by calling the object's `elapsedTimeMS` method. At this point, it will create `LogRecord` event to store in its `log`, where the event name will have the form

```
httpReq + ": " + elapsedTime
```

where `httpReq` is the `HttpRequest` object that this `EndTimerHandler` matched against, and `elapsedTime` is the elapsed time returned by the `Timer`.

3 Applications

Using these handlers, you will write three server applications. These servers will be accessed by HTTP clients, like Netscape. For each application you will do the following:

1. Create a `BasicServer` instance.
2. Create all necessary handlers.
3. Attach the handlers to the server.
4. Invoke the server's `run()` method.

Each application will be encapsulated within a class. Each application class and the necessary handlers are described below. Note that in all cases, the `public static main()` method for your server should get the port number to pass to `BasicServer`'s constructor from the command line (i.e., start the server with `java -Dport=portnum ServerClassName`). The port number can be extracted in your server with the `Integer.getInteger()` method. For testing purposes, so everyone uses different ports, test your server using port `x33yy`, where `x` is your section number (either 1 or 2) and `yy` is the last two digits of your account number.

1. `PageServer`. This application will provide basic web server functions, and will log all requests it receives. It will create a `GetFileHandler`, `RemoteGetFileHandler`, `GetRecordsHandler`, and `LogEventHandler`. Handler order constraints are `NO_PREFERENCE` for all handlers. The main method of `PageServer` should create a new `LocalLog` (say of size 100) to provide to the `LogEventHandler` constructor, and it should pass the same `LocalLog` object to the `GetRecordsHandler`, and use the `String WebLog` for the `logName` parameter.

As a result, clients can issue HTTP requests of the following form:

- `http://ServerAddr:portNum/getFile/fileName`
- `http://ServerAddr:portNum/remoteGetFile/URL`

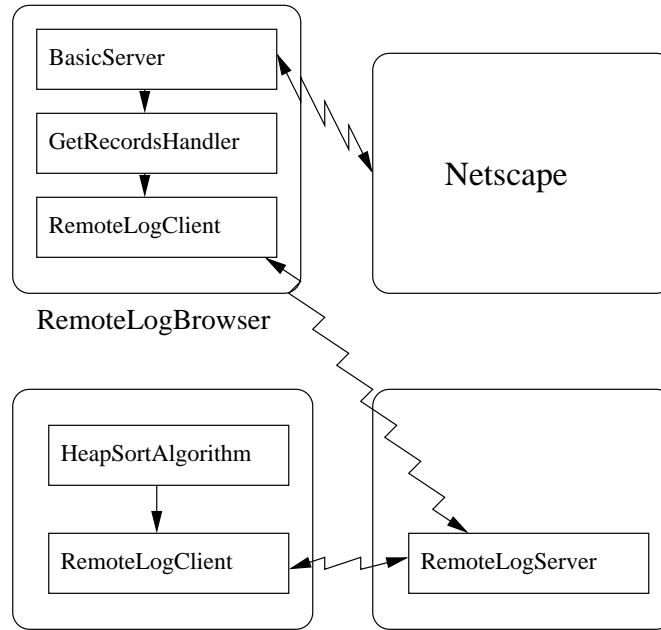


Figure 2: RemoteLogBrowser: A Server Application to Query a Remote Log

- `http://ServerAddr:portNum/getRecords/WebLog/pattern`

Remember that the URL part should not include the `http://` prefix.

2. `PageServerPlusTimer`. This application will extend your basic server with timers. You will create the handlers `GetFileHandler`, `RemoteGetFileHandler`, `GetRecordsHandler`, and `StartTimerHandler`, and `EndTimerHandler` (you will not use `LogEventHandler`). Here, the functionality of the server is the same, but rather than log an event for each request that comes in, you will log how long it takes to process the request. Handler orders are `EARLY` for `StartTimerHandler`; `NO_PREFERENCE` for `GetFileHandler`, `RemoteGetFileHandler`, and `GetRecordsHandler`; and `LATE` for `EndTimerHandler`. The main method of `PageServerPlusTimer` should create a new `LocalLog` (say of size 100) to provide to the `EndTimerHandler` constructor, and it should pass the same `LocalLog` object to the `GetRecordsHandler`, and use the String `WebLog` for the `logName` parameter, as above. In addition, main will have to create a `Timer` object to be shared (given to the constructors) of both the `StartTimerHandler` and the `EndTimerHandler`. Both of these objects should use pattern `*`. Clients issue HTTP requests as in the previous server.
3. `RemoteLogBrowser`. This application will provide web access to the log of a running program. To do this, you will create a `GetRecordsHandler`, with

order `NO_PREFERENCE`, and will pass to it an instance of `RemoteLogClient`, rather than `LocalLog`. Give it the name `RemoteLog`. This `RemoteLogClient` will connect to the same `RemoteLogServer` that some other program that is running, e.g. `HeapSortAlgorithm` from project 1.² This is depicted in Figure 2. This way, the `RemoteLogBrowser` provides a way for you to look at the log being created by another program. Clients can issue HTTP requests of the following form:

- `http://ServerAddr:portNum/getRecords/RemoteLog/pattern`

When starting up your `RemoteLogBrowser`, you should be able specify the port it should use to connect to the `RemoteLogServer` (that is, the port the `RemoteLogServer` was started with). Do this with the option `-Dlogport=logportnum`. Therefore, you should be able to start your server as follows:

```
java -Dport=portnum -Dlogport=logportnum RemoteLogBrowser
```

This specifies that the `RemoteLogBrowser` is listening for HTTP requests on port `portnum` and will connect to the `RemoteLogServer` listening on port `logportnum`.

One way to implement this option is by using the following code, to be at the start of the main method in `RemoteLogBrowser`:

```
int remoteLogPort = Integer.getInteger("logport", n).intValue();
```

The variable `n` contains the default port to use (which you can set based on your userid, as specified above). You then use `remoteLogPort` when creating the `RemoteLogClient` object. As for the host to connect to, you can either assume `localhost` (that is, the `RemoteLogBrowser` will be running on the local machine), or you can use a similar parameter passing mechanism (that defaults to `localhost`) for specifying the hostname.

4 Amendments and Notes

This section contains useful information that has come up since the project assignment was posted.

4.1 Building

The code we provided you uses libraries that are *deprecated*, meaning that Java will eventually not support them. As a result, when you compile the files, you'll get the warning

²For testing, run this `RemoteLogServer` using port `x32yy`, where `x` is your section number (either 1 or 2) and `yy` is the last two digits of your account number.

Note: Some input files use or override a deprecated API.
Note: Recompile with `-deprecation` for details.

Don't worry about this.

4.2 Testing

You should test your project by connecting to your web server with a web browser. For this to work on the CSIC machines, it is likely that you will have to start the web browser on the same machine you are running the server. If you are working locally on a machine in 3107 CSIC, or if your ssh session forwards X11, you will be able to type `netscape` at the command prompt, and Netscape will appear on your desktop. Then access the URL

```
http://localhost:portNum/pattern
```

where `portNum` is the port your server is listening on, and `pattern` is the URL, as specified above.

If your ssh connection does not forward X11 (i.e. running Netscape doesn't work), you can use a text-based browser, like `links`. In particular, you can do

```
links http://localhost:portNum/pattern
```

from the command line prompt, and it should connect to your server. We have found that Netscape is more forgiving with its HTTP than is `links`. In particular, `links` will often display "connection reset by peer" after displaying the correct results—don't worry about this.

Finally, you should test all of your file handlers (`GetFileHandler` and `RemoteGetFileHandler`) using simple `.html` pages, without pictures or other embedded links. In particular, if the pages have embedded pictures, then additional requests will be sent by the browser to your server, which will not be able to handle them (since they won't end in `.html`). The result will be incomplete pages, and/or errors reported by browser that were sent from your server. Any files from Hicks' course web page will work, since they are all simple HTML (to try them using `GetFileHandler`, you will need to copy the `.html` file to your home directory). For another remote file, try <http://gcc.gnu.org/onlinedocs/gcc/>.

4.3 Bugs

Here we list the bugs in the original specification that are fixed in the current version.

1. There was a bug in the specification in Section 1.1. The old text read that if `readRequest` returns false, then an error has occurred. This is incorrect; `readRequest` will return false when the handler does not match the request, and so does not process it. Errors are indicated when `readRequest` throws a `HandlerException`.

2. There was no description as to what to do if no handler generates any output. This has been added in a new Section 1.2 (along with the corrected text for Bug 1).
3. Some of the constructors specified in Section 2.1 were ill-specified: they did not include an `orderConstraint` in the constructor. This has been rectified in the above writeup.
4. The prototype for `EventHandler` in Section 2 was incorrect: it included the line `public HandlerOrder orderConstraint`. This line is not needed (and should be deleted), because this field will be inherited from `HttpHandler`.
5. We have indicated that your `RemoteLogBrowser` should be able to accept java option `-Dlogport=logportnum` so that you can specify when starting it up what `RemoteLogServer` it should connect to. See the description of `RemoteLogBrowser` for details.
6. The description of `RemoteLogBrowser` failed to give a name for the remote log it was accessing, and incorrectly specified the form of the URL to use. See the description for more details.

Here are some more minor bugs:

1. There was a typo in the description of `EndTimerHandler`: we referred to it method `stopTimer` rather than `endTimer`.
2. The prototype for the constructor to the `EventHandler` class had a spurious `void` in front of it.
3. There was a typo in the description of `PageServerPlusTimer` that referred to `PageServer` rather than `PageServerPlusTimer`.
4. You can assume that the `RemoteLogServer` is running on `localhost`; see the description of `RemoteLogBrowser`.