Java 5: Distributed Java Applications

Introduction to Distributed Computing

Welcome to the O'Reilly School of Technology Java 5: Distributed Java Applications Course!

Course Objectives

When you complete this course, you will be able to:

- extend your stand-alone Graphical User Interfaces to communicate with a remote server.
- develop a multi-threaded server that simultaneously supports a number of connected clients.
- design a protocol using an XML XSD specification.
- develop client- and server-side controllers that follow the protocol.
- develop effective JUnit test cases to validate the execution of these controllers.
- develop a testing framework that maximizes code coverage of JUnit test cases.

In this Java course, you will develop a client/server distributed Java application from the ground up. Here you will exercise all of your Java skills to implement a graphical client that communicates with a remote back-end server using XML messages. You will learn the tradeoffs that are common in client/server systems and gain valuable insights into how to design your own distributed, multi-threaded applications.

From the very first lab, you will be developing a client/server application, adding new features and functionality with each successive lab. You will learn by following the design and implementation of the application in the lab. Each quiz will validate that you learned the key information and the projects, performed at your pace, will describe useful extensions to the main development of the overall project.

Lesson Objectives

When you complete this lesson, you will be able to:

- create a ComputationServer application.
- create the ComputationClient.
- make the client send command requests and process the response.
- use JUnit test cases together with the EclEmma code coverage plugin to identify the code that runs during testing.

Welcome to the O'Reilly School of Technology's Advanced Java course. Although it's unlikely that this fifth course in the Java series is your first OST course, we like to include a description of how OST works in all of our courses, just in case. Feel free to skip these first sections if you know you've got a solid understanding of our tools and methods, and instead start at the "Introduction to Distributed Computing" section.

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

```
White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add looks like this.

If we want you to remove existing code, the code to remove ~~will look like this~~.

We may also include instructive comments that you don't need to type.
```

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

```

The plain black text that we present in these INTERACTIVE boxes is
provided by the system (not for you to type). The commands we want you to type look lik
e this.
```

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:

```
Gray "Observe" boxes like this contain information (usually code specifics) for you to
observe.
```

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

Introduction to Distributed Computing

In this course, you'll implement a complex client/server application from start to finish.

You'll learn how an application is developed. When you look at software code, it's almost impossible to determine the *order* in which the code was developed. There are many different approaches you might take to develop a particular application. In this coures, we're going develop an application under these constraints:

- As you add each new capability to the application, you will always have working code to demonstrate. By verifying that the code *works* at every step, you can be reasonably sure that the final application will work.
- As you develop new functionality for the application, you will validate the proper execution using JUnit test cases. Unit test cases are essential for working on any large system; they become the baseline against which you measure your progress.

Testing

A good programmer delivers high-quality code that has been tested using a set of unit tests. Depending on the programming language, there are a number of unit testing frameworks available. For this course, we'll use JUnit, the industry standard for Java. The original JUnit (version 3.0) will suffice for this project, but we encourage you to review the capabilities of version 4.0 on your own as well.

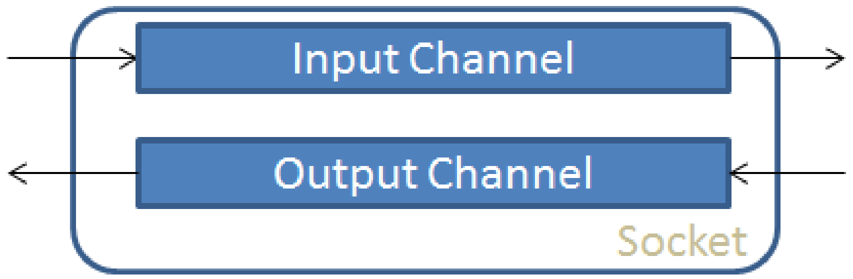
One of the best practices to follow is to separate the code being built from the testing code. Eclipse provides an extremely useful capability to support this practice. Each Java project in Eclipse has a source folder labeled **/src**. You can add any number of source folders to a Java project and the classes contained within these folders are *overlaid* with each other. You will create a source folder **/test** to store all JUnit tests; the package hierarchy of this **/test** source folder is identical to the **/src** folder. This allows you to write test cases that validate protected and package private methods of Java classes without running the risk of exposing either data or methods to other classes unnecessarily. In fact, none of the attributes or methods developed in this course are labeled **private** for just this reason.

Code Coverage

While testing is essential to confirming the quality of your code, you must use other means to validate the implementation, and you must always be concerned about the quality of your test cases. Specifically, how do you know that your test cases truly exercise the code you are writing? There are many code coverage toolkits available that let you determine whether an individual line of Java code has been executed. For this course, you'll use the freely available EclEmma Eclipse plugin. EclEmma identifies which Java statements execute. Based on this information, you can either write additional test cases or validate (with a code review) that the non-executed code is still correct. In many cases, there are some exceptional scenarios that are nearly impossible to automate using a test case; however, upon inspecting the code manually, you can determine that the code would operate properly if these exceptional scenarios happen to occur.

Socket Abstraction

The client/server architecture depends on a reliable connection-oriented communication such as internet sockets. Think of a socket as the endpoint of communications between processes across a network. Take a look at the figure below; because the connection is bidirectional, there is both an *input* channel and an *output* channel:



Let's assume this socket is on the client to represent the communication to the server. You write information onto the *Input Channel* to be transmitted to the server, then the client reads information from the *Output Channel*, that was written to the socket by the server. The socket abstraction can break down though. For example, the client can be delayed indefinitely when writing to the input channel if the output channel is overloaded. One way this can happen is if your client only sends information to the socket without retrieving any information from the Output Channel. You can predict when that will happen by using getSendBufferSize() on your socket to determine the size of its buffer. In practice, with well-written clients and servers, you won't encounter this problem.

Note To view the Java API, click the API icon (API) in the toolbar at the top of the screen. From there, you can find detailed information about any classes or methods we talk about in this course.

Sample Client/Server Application

Let's walk through a stripped-down client/server system to review its fundamental elements. We'll start with a system where the server echoes a string back to the client. You'll upgrade the client (and server) to enable remote users to submit small computation requests for processing, such as addition (+), multiplication (*), subtraction (-), or division (/) of two integer values.

We'll incorporate these key features of the client/server architecture in our ComputationServer application:

- Once the server runs, it awaits incoming requests from a client.
- The client performs rudimentary "error checking" of requests sent to the server. For example, ComputationClient sends only well-formed requests to process integers.
- The server must be robust in the case of ill-formed requests (even though the client should perform error checking) and degrade gracefully.
- The protocol between the client and server must be specified clearly and understood by both parties.

Create a Java project in Eclipse, name it **DistributedApp**, and assign it to the **Java5_Lessons** working set. Eclipse creates a **/src** folder for you.

Create a **ComputationServer** class in the default package of the **/src** folder as shown:

CODE TO TYPE: /src/ComputationServer.java

```
import java.io.*;
import java.net.*;

public class ComputationServer {

  public static void main(String[] args) throws IOException {
    ServerSocket serverSocket = new ServerSocket(7434);
    System.out.println("Server awaiting client connections");

    Socket client = serverSocket.accept();
    BufferedReader fromClient = new BufferedReader(new InputStreamReader(client.getInpu
tStream()));
    PrintWriter toClient = new PrintWriter (client.getOutputStream(), true);

    while (true) {
      try {
        String str = fromClient.readLine();
        if (str == null) { break; }

        toClient.println(str);
      } catch (IOException ioe) {
        // should any interruption occur, stop server
        break;
      }
    }

    client.close();
    serverSocket.close();
    System.out.println("Server done.");
  }
}
```

To run ComputationServer, right-click the **ComputationServer.java** file in the **/src** source folder and select **Run As | Java Application** (or click the icon).

The message, "Server awaiting client connections" appears in the Console tab at the bottom of your Eclipse window.

Take a closer look at the first part of the main method:

OBSERVE:

```
public static void main(String[] args) throws IOException {
    ServerSocket serverSocket = new ServerSocket(7434);
    System.out.println("Server awaiting client connections");

    Socket client = serverSocket.accept();
    BufferedReader fromClient = new BufferedReader(new InputStreamReader(client.getInpu
tStream()));
    PrintWriter toClient = new PrintWriter(client.getOutputStream(), true);
```

ComputationServer receives a client connection request using the ServerSocket.accept method that listens for a connection to be made to the socket and accepts it. The ComputationServer implementation forms the minimal possible implementation of a server. This server checks in at just 30 lines. Once a connection is established, the server creates an object, toClient, to communicate with the connecting client.

In the call to the **PrintWriter** constructor, the second parameter (**true**) ensures that strings written using **println** are flushed automatically. If you didn't do this, you'd need to flush the bytes in the PrintWriter manually to make sure the socket received the data properly.

Note Throughout this course, any classes created in the default package are assumed to be throw-away code or code whose only purpose is to demonstrate an idea or principle.

Let's take a closer look at the main **while** loop:

OBSERVE:

```
  while (true) {
      try {
        String str = fromClient.readLine();
        if (str == null) { break; }

        toClient.println(str);
      } catch (IOException ioe) {
        // should any interruption occur, stop server
        break;
      }
  }
```

fromClient is a BufferedReader associated with the input stream of the client socket; the server reads String lines from **fromClient** that represent the commands from the client. This simple server does nothing more than echo input strings back to the client using the **toClient** PrintWriter associated with the output stream of the client socket. Printing strings to the PrintWriter sends the text back to the remote client.

In the examples in this lab, you will run all applications on the virtual server, so the host name will always be "localhost." The first question you might have is about how to make sense of the **Socket client = serverSocket.accept();** code fragment. Does it return the server's socket? If not, what socket is returned? The key idea to remember is that a socket is merely a convenient networking abstraction. In other words, the socket constructed and returned by the accept method invocation is used by the server to manage the communication (both input and output) with the specific remote client communicating with the server. The server will create such a socket object for each connected client.

In this basic first example, ComputationServer can service just a single client at a time. Even worse, once that client has completed its use of the server, the server exits. The code was written this way intentionally so we could first focus on the essential *socket* structures necessary for client/server communication. Now you're ready to complete the sample application.

Create a **ComputationClient** class in the default package of the **/src** folder as shown:

/src/ComputationClient.java

```
import java.io.*;
import java.net.*;
import java.util.*;

public class ComputationClient {

  public static void main(String[] args) throws Exception {
    Socket server = new Socket ("localhost", 7434);

    PrintWriter toServer = new PrintWriter (server.getOutputStream(), true);
    BufferedReader fromServer = new BufferedReader (new InputStreamReader(server.getInp
utStream()));

    System.out.println("Type string to be echoed back by server");
    Scanner sc = new Scanner(System.in);
    while (sc.hasNextLine()) {
      String str = sc.nextLine();
      toServer.println(str);

      String value = fromServer.readLine();
      System.out.println("Server sends: " + value);
    }

    server.close();
  }
}
```

Let's look closer:

OBSERVE:

```
    Socket server = new Socket ("localhost", 7434);

    PrintWriter toServer = new PrintWriter (server.getOutputStream(), true);
    BufferedReader fromServer = new BufferedReader (new InputStreamReader(server.getInp
utStream()));

    System.out.println("Type string to be echoed back by server");
    Scanner sc = new Scanner(System.in);
    while (sc.hasNextLine()) {
     String str = sc.nextLine();
        toServer.println(str);

        String value = fromServer.readLine();
        System.out.println("Server sends: " + value);
    }
```

The client code communicates with the server by opening up a socket to **"localhost"** on the pre-arranged port number **7434**. This server object is the client-side abstraction by which the client communicates with the server. We create a PrintWriter object (**toServer**) by connecting to **the output stream of the server socket**. The client sends requests to the server by writing to this PrintWriter. We use a BufferedReader object (**fromServer**) to receive output from the server.

It seems odd to say that we're receiving output when you see that an InputStreamReader object is being created, but remember that on the client we're reading input *from the server*, which is creating the output.

ComputationClient contains an inner **while** loop that reads input from the keyboard and sends the text strings to the server. Assuming that you still have ComputationServer running, run ComputationClient and type **testing** as shown:

INTERACTIVE SESSION:

```
Type string to be echoed back by server
testing
Server sends: testing
```

The PrintWriter masks all exceptions that might arise. To know for sure whether the println command sent the communication properly, you need to invoke manually checkError() as shown:

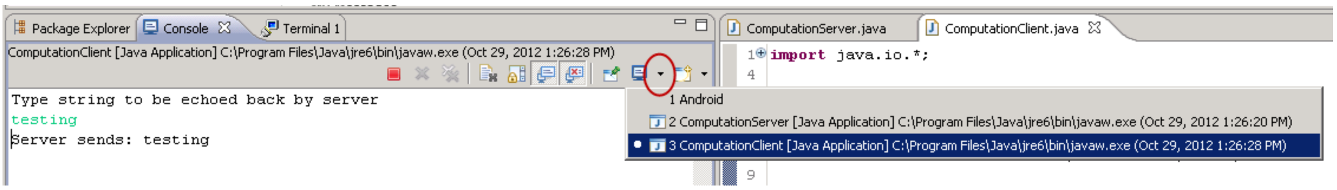
CODE TO TYPE: /src/ComputationClient.java

```java
import java.io.*;
import java.net.*;
import java.util.*;

public class ComputationClient {

  public static void main(String[] args) throws Exception {
    Socket server = new Socket ("localhost", 7434);

    PrintWriter toServer = new PrintWriter (server.getOutputStream(), true);
    BufferedReader fromServer = new BufferedReader (new InputStreamReader(server.getInp
utStream()));

    System.out.println("Type string to be echoed back by server");
    Scanner sc = new Scanner(System.in);
    while (sc.hasNextLine()) {
      String str = sc.nextLine();
      toServer.println(str);
      if (toServer.checkError()) {
        System.err.println("Server communication no longer available.");
        break;
      }

      String value = fromServer.readLine();
      System.out.println("Server sends: " + value);
    }

    server.close();
  }
}
```

Note The AutoFlush feature of the PrintWriter ensures that strings written are immediately flushed onto the communication channel whenever the **println()** method is invoked. If you use the **print** method and include a "\n" character in the string being written (**toServer.print(str + "\n")**), it won't trigger the flush automatically (**toServer.flush()**).

ComputationServer reads a string from the connecting client using the BufferedReader object. If null is returned, then the client has disconnected from the server. When ComputationServer is done with the client, we close the client socket; if the client tries to read input from the (now disconnected) server, null is returned.

Terminate the client, either by typing **Ctrl-z** in the console window or clicking the 🟥 Terminate icon. To observe that process in Eclipse, click the down-pointing arrow on the **Display Selected Console** icon (🖥) in the Console panel. You can switch the selected console to review the output of both processes.



ComputationClient [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Oct 29, 2012 1:26:28 PM)

```
Type string to be echoed back by server
testing
Server sends: testing
```

```java
import java.io.*;
```

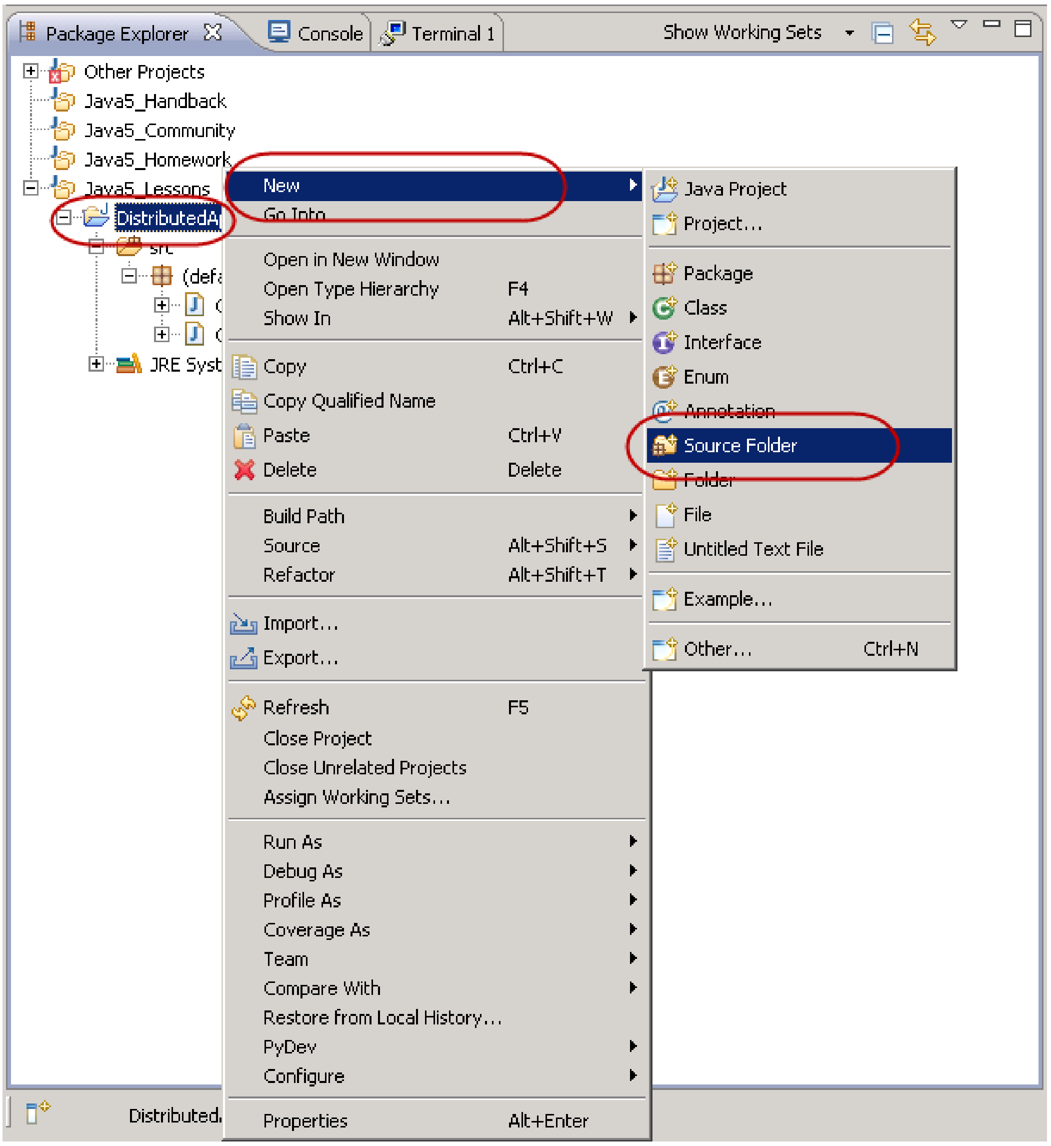
1 Android
2 ComputationServer [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Oct 29, 2012 1:26:20 PM)
3 ComputationClient [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Oct 29, 2012 1:26:28 PM)

Validate that ComputationServer prints "Server done." as its final output, which demonstrates that it terminates normally. Both ComputationServer and ComputationClient should stop executing. This simplified client contains less than 30 lines of Java code.

Testing

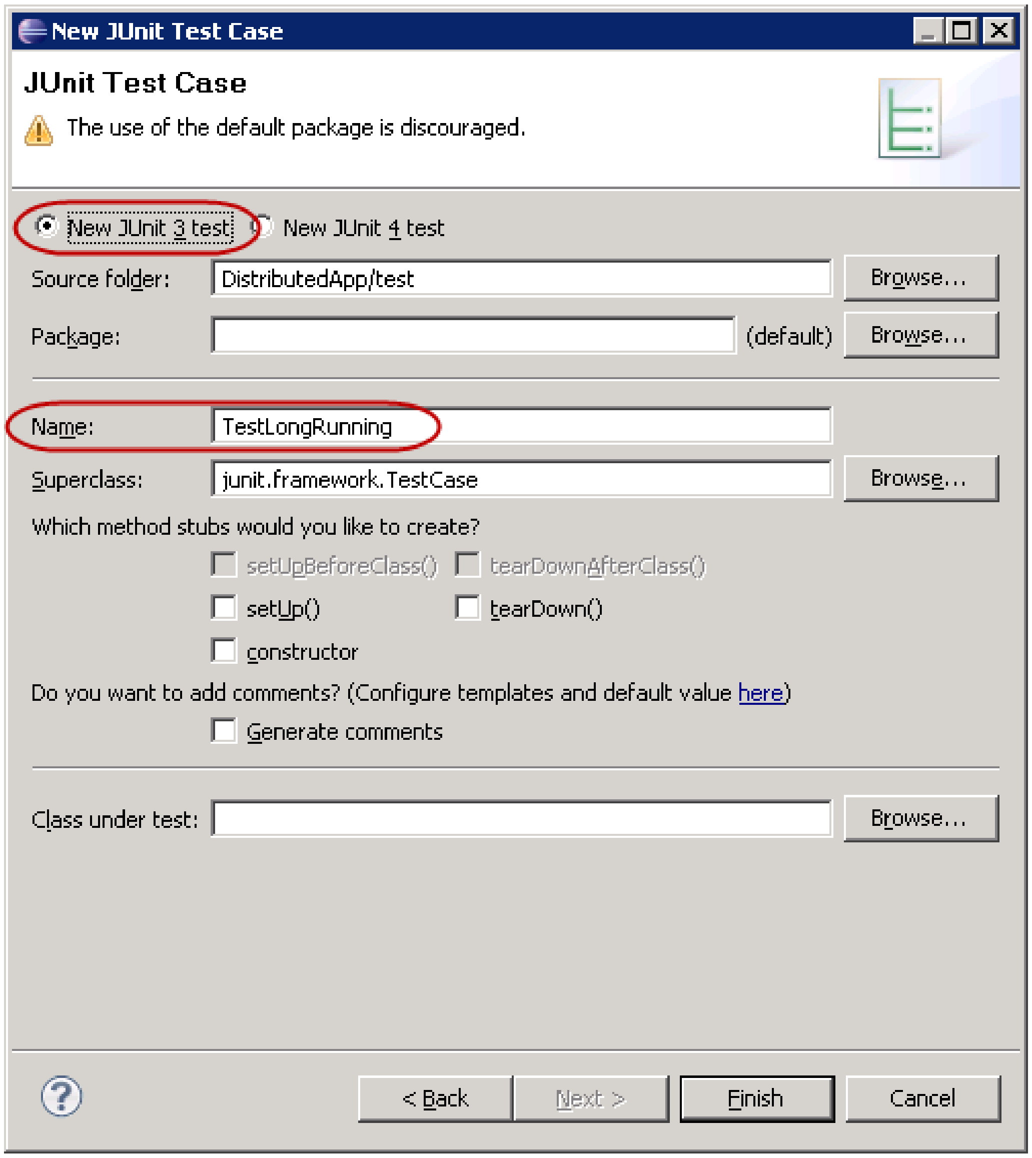
When you develop applications, always maintain a working implementation. To help with this process you'll develop JUnit test cases with each lab. For this first lab, there is a **TestLongRunning** test case that demonstrates one way to validate the server's proper behavior automatically.

Create a source folder in which to place your JUnit test case classes. To do that, right-click on your top-level (**DistributedApp**) project and select **New | Source Folder**.



When prompted, enter **test** as the folder name.

Create the **TestLongRunning** class by right-clicking on the **/test** source folder icon and selecting **New | Other**. In the dialog that appears, open **Java | Junit | JUnit Test Case** and click **Next**. Then, select the **New JUnit 3 test** radio button, enter **TestLongRunning** as the test case name, and click **Finish**.


New JUnit Test Case

JUnit Test Case

The use of the default package is discouraged.

[✓ New JUnit 3 test] New JUnit 4 test

Source folder: DistributedApp/test

Package: (default)

Name: TestLongRunning

Superclass: junit.framework.TestCase

Which method stubs would you like to create?

setUpBeforeClass() tearDownAfterClass()

setUp() tearDown()

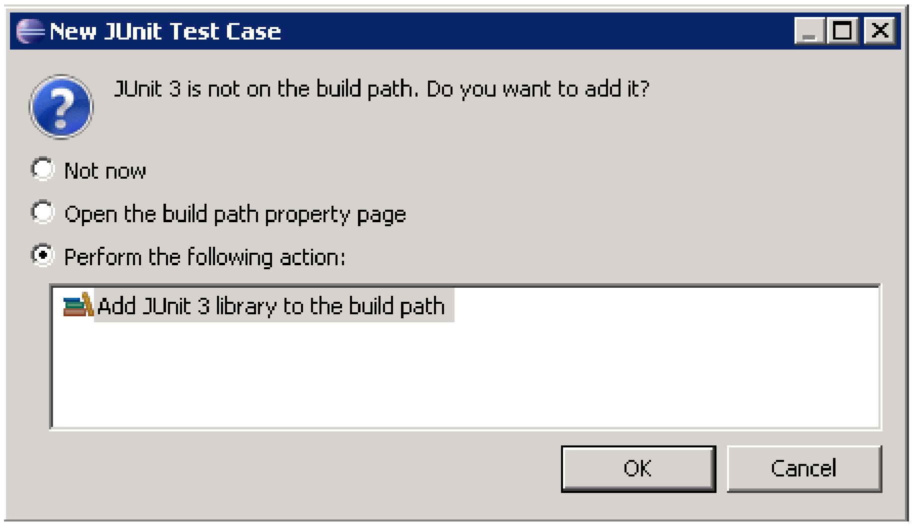
constructor

Do you want to add comments? (Configure templates and default value here)

Generate comments

Class under test:

Eclipse will prompt you to add the JUnit 3 library to the build path. Click **OK**.


New JUnit Test Case

JUnit 3 is not on the build path. Do you want to add it?

Not now

Open the build path property page

[✓ Perform the following action:]

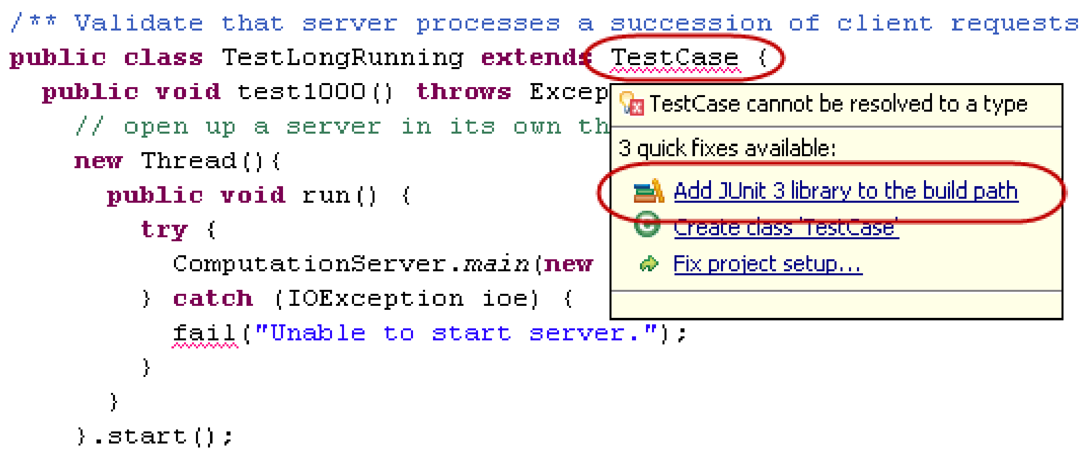
Add JUnit 3 library to the build path

Type **TestLongRunning** as shown:

CODE TO TYPE: /test/TestLongRunning.java

```
import java.io.*;
import java.net.*;
import junit.framework.TestCase;

/** Validate that server processes a succession of client requests. */
public class TestLongRunning extends TestCase {

  public void test1000() throws Exception {
    // open up a server in its own thread of execution.
    new Thread(){
      public void run() {
        try {
          ComputationServer.main(new String[]{});
        } catch (IOException ioe) {
          fail("Unable to start server.");
        }
      }
    }.start();

    // connect away and retrieve input
    Socket server = new Socket ("localhost", 7434);

    PrintWriter toServer = new PrintWriter (server.getOutputStream(), true);
    BufferedReader fromServer = new BufferedReader (new InputStreamReader(server
.getInputStream()));

    for (int i = 0; i < 1000; i++) {
      System.out.println(i);
      toServer.println(i);

      int value = Integer.valueOf(fromServer.readLine());
      assertEquals (i, value);
    }

    server.close();
  }
}
```

If this code doesn't compile, make sure you've added JUnit 3 libraries to your build path. The simplest way to configure your project is to hover your mouse over the word **TestCase**, which may have a wavy red line unerneath it. Eclipse has many self-help features to increase your productivity. The next image we'll see shows the Eclipse pop-up window that appears just below the code in question. Move your mouse to select the first option, **Add JUnit 3 library to the build path**. All compiler errors will disappear:



```
/** Validate that server processes a succession of client requests. */
public class TestLongRunning extends TestCase {
  public void test1000() throws Excep
    // open up a server in its own th
    new Thread(){
      public void run() {
        try {
          ComputationServer.main(new
        } catch (IOException ioe) {
          fail("Unable to start server.");
        }
      }
    }.start();
```

TestCase cannot be resolved to a type

3 quick fixes available:

Add JUnit 3 library to the build path

Create class 'TestCase'

Fix project setup...

Let's look more closely at this test case, which illustrates the full sequence of actions required to demonstrate a typical client/server interaction:

OBSERVE:

```
// open up a server in its own thread of execution.
new Thread(){
  public void run() {
    try {
      ComputationServer.main(new String[]{});
    } catch (IOException ioe) {
      fail("Unable to start server.");
    }
  }
}.start();
```

The Java threading model allows you to spawn a new thread of control any time. While the syntax looks a bit obscure, the above code shows how you create a new anonymous (which meaning that the name of the class isn't necessary) subclass of Thread whose run() method executes the ComputationServer main method. As you know, this main method requires an array of String objects, which is handled by **creating an empty String array**. Next, the test case "simulates" a client connection:

OBSERVE:

```
    // connect away and retrieve input
    Socket server = new Socket ("localhost", 7434);

    PrintWriter toServer = new PrintWriter (server.getOutputStream(), true);
    BufferedReader fromServer = new BufferedReader (new InputStreamReader(server
.getInputStream()));
```

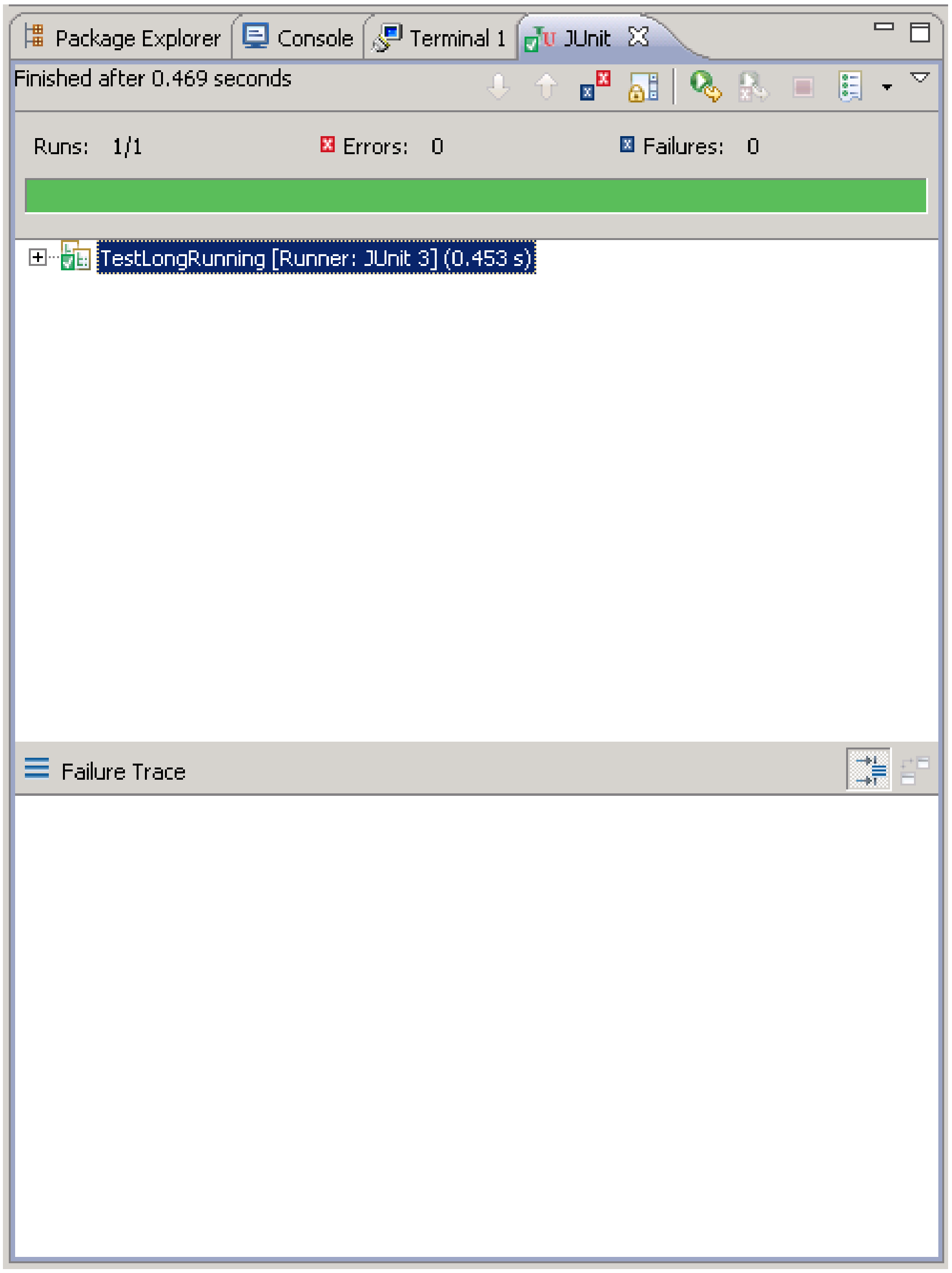
The connecting **server** Socket opens a connection to the ComputationServer executing in its own thread. The **toServer** PrintWriter is the object used to communicate to the server, while the **fromServer** BufferedReader is the object for reading responses back from the server. Everything comes together in the final loop, which issues requests to the server, one at a time, and closes the socket when it's done:

OBSERVE:

```
for (int i = 0; i < 1000; i++) {
  System.out.println(i);
  toServer.println(i);

  int value = Integer.valueOf(fromServer.readLine());
  assertEquals (i, value);
}

server.close();
```

This code loops 1000 times, each time sending a number to the server using **toServer** and then reading back from the server a string to verify (using **assertEquals**) that the value returned is the same as the value sent. Once that's done, the **server** is closed.

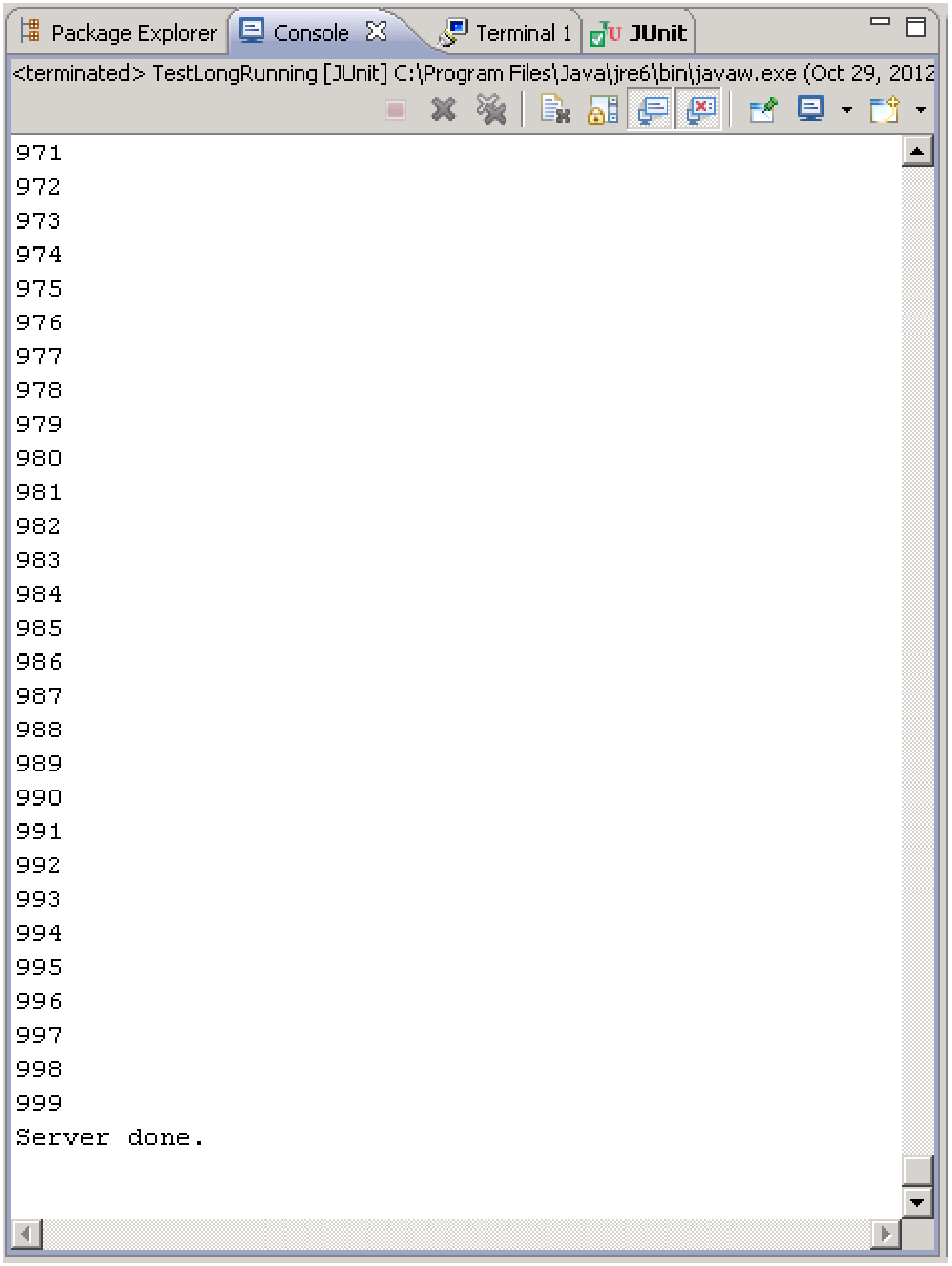
Run this test case by right-clicking the **TestLongRunning.java** file and selecting **Run As | JUnit Test**. This test case spawns the server in a separate thread and then opens up a socket communication to this server and initiates a sequence of 1000 operation requests. This test case contains both the instantiation of the server and client—both useful and necessary for testing a client/server application. The console will show the output (numbers from 0 to 999). Eclipse will switch to the JUnit tab:



Finished after 0.469 seconds

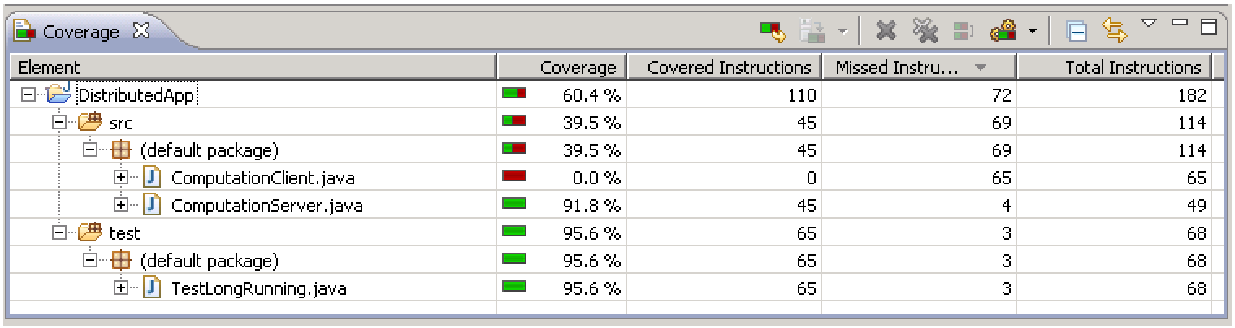
Runs: 1/1   Errors: 0   Failures: 0

TestLongRunning [Runner: JUnit 3] (0.453 s)



```
<terminated> TestLongRunning [JUnit] C:\Program Files\Java\jre6\bin\javaw.exe (Oct 29, 2012
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
Server done.
```

Once you have your test case running correctly (note the green bar), switch back to the Package Explorer tab, right-click this file again, and select **Coverage As | JUnit Test** to rerun all tests cases to generate a report of the code that ran. The summary code appears at the bottom of Eclipse in a Coverage tab, as shown below (your numbers may vary slightly depending on how you typed in the code):

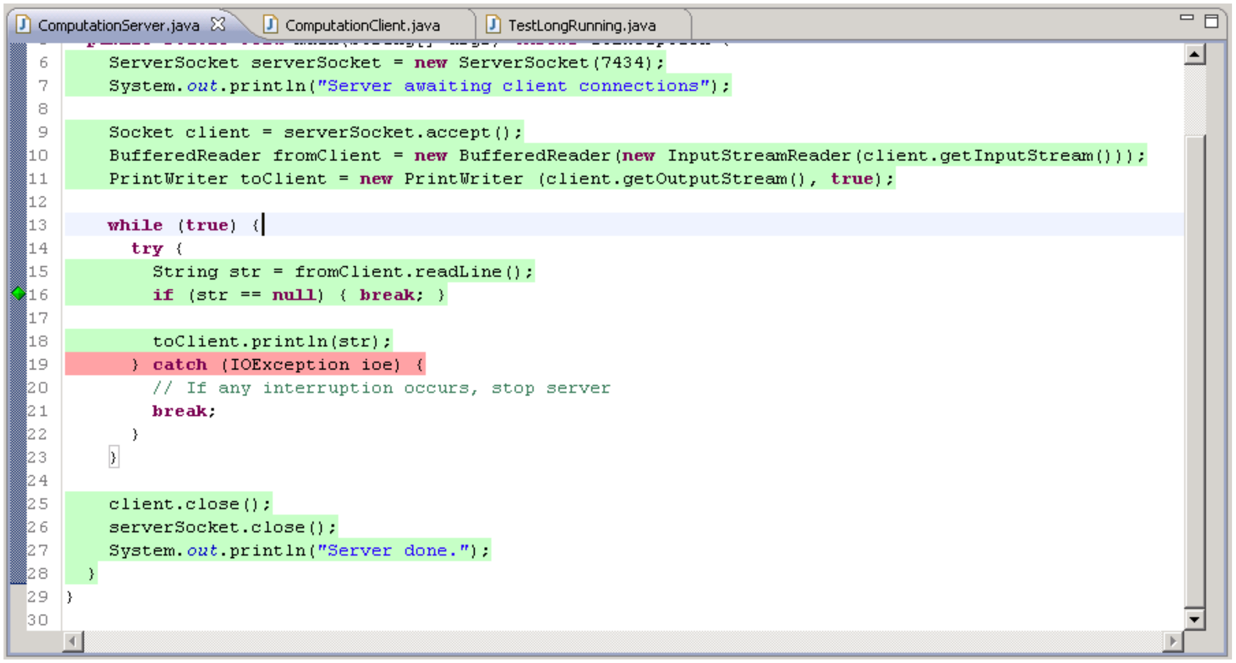


Element	Coverage	Covered Instructions	Missed Instru...	Total Instructions
DistributedApp	60.4 %	110	72	182
src	39.5 %	45	69	114
(default package)	39.5 %	45	69	114
ComputationClient.java	0.0 %	0	65	65
ComputationServer.java	91.8 %	45	4	49
test	95.6 %	65	3	68
(default package)	95.6 %	65	3	68
TestLongRunning.java	95.6 %	65	3	68

In this summary report, the ComputationClient has no associated test cases because it has no lines that executed. Since our goal is to achieve 80% coverage of each individual Java file within a lab, it seems clear that we need more test cases. As you work through the labs in this course, you'll find that we often design and implement code in specific ways to make sure that it can be tested using JUnit test cases. For this simple server, we left out a lot of error-handling code, which means that the code coverage was quite high (91.8%)

with just a single test case.

To review individual files and find out which lines of code executed, open the Java files and review the color-coding for each line. For example, you can see in our example that the IOException exception handler did not execute in ComputationServer. Exceptional cases are hard to test, so we want to make an effort to implement code that can be tested automatically.



```
    ServerSocket serverSocket = new ServerSocket(7434);
    System.out.println("Server awaiting client connections");

    Socket client = serverSocket.accept();
    BufferedReader fromClient = new BufferedReader(new InputStreamReader(client.getInputStream()));
    PrintWriter toClient = new PrintWriter (client.getOutputStream(), true);

    while (true) {
      try {
        String str = fromClient.readLine();
        if (str == null) { break; }

        toClient.println(str);
      } catch (IOException ioe) {
        // If any interruption occurs, stop server
        break;
      }
    }

    client.close();
    serverSocket.close();
    System.out.println("Server done.");
  }
}
```

EclEmma color-codes your source files like this:

- Red shaded regions did not execute. For example, this test did not exercise any code from the ComputationClient class, which registers a coverage of 0.0%.
- Green shaded regions executed. Review the ComputationServer class to see the code that executed. As you can see, none of the exception or error handling code executed. This is often the hardest code to test.
- Yellow shaded regions indicate code that could execute under various conditions.

A yellow line suggests that there was a logical conditional contained in that line that was not evaluated under all possible values. This usually happens when you execute only one side of a logical conditional statement (such as **if** or **case**).

Note To turn off the shaded color after running EclEmma, place your cursor in the Java code file and make a change. I often just place my cursor at the end of any row, or within a documentation block, and add a space.

One weakness of the EclEmma plugin is that no coverage data is recorded if youterminate an application manually that you have launched using the **Coverage As** feature. Additionally, it may often be difficult to determine why some lines of code remain marked in red even though you are pretty sure that the code did execute. We'll use EclEmma to validate that 80% of the written code is executed by the JUnit test cases developed throughout this course. Setting the threshold at 80% keeps you honest as a programmer and keeps you from having to write lots of test code to validate strange exceptional situations (but note that you must still review all non-executed code to make sure it functions properly). Ultimately, we are concerned with the code coverage reported for classes in the **src** source folder.

Adding in Computation Logic

You are now ready to complete this lab by making ComputationServer a functional calculator. First, specify the protocol between the client and server. Let's have the client send three string lines of input. The first line contains the operator to be performed, the second line contains an integer operand1, and the third line contains an integer operand2. Thus the ComputationServer only needs to read three lines of input (assuming they are all present) and then compute the answer. Then, ComputationServer writes two string lines of output to the client. The first line contains a zero (0) or a negative one (-1) declaring the success or failure of the operation. The second line contains either a result (if successful) or an error string (if failure). Here are two scenarios with examples:

.

Client communication to server	Server response
+ 6 9	0 15
/ 1 0	-1 Unable to process request: (/ 1 0)

Make these changes to ComputationServer:

CODE TO TYPE: /src/ComputationServer.java

```
import java.io.*;
import java.net.*;

public class ComputationServer {

  public static void main(String[] args) throws IOException {
    ServerSocket serverSocket = new ServerSocket(7434);
    System.out.println("Server awaiting client connections");

    Socket client = serverSocket.accept();
    BufferedReader fromClient = new BufferedReader(new InputStreamReader(client.
getInputStream()));
    PrintWriter toClient = new PrintWriter (client.getOutputStream(), true);

    while (true) {
~~      try {~~
~~        String str = fromClient.readLine();~~
~~        if (str == null) { break; }~~

~~        toClient.println(str);~~
~~      } catch (IOException ioe) {~~
~~        // should any interruption occur, stop server~~
~~        break;~~
~~      }~~
      String op=null, s1=null, s2=null;

      try {
        op = fromClient.readLine();
        s1 = fromClient.readLine();
        s2 = fromClient.readLine();
      } catch (Exception e) {
        System.err.println("Closing Client connection.");
        break;
      }

      // communication terminated prematurely
      if (op == null || s1 == null || s2 == null) {
        System.err.println("Closing Client connection.");
        break;
      }

      Integer int1=null, int2=null;
      try {
        int1 = Integer.valueOf(s1);
        int2 = Integer.valueOf(s2);

        // support four operations (multiply, divide, add, subtract).
        if (op.equals("*")) { output(toClient, int1 * int2); }
        else if (op.equals("/")) { output (toClient, int1 / int2); }
        else if (op.equals ("+")) { output (toClient, int1 + int2); }
        else if (op.equals ("-")) { output (toClient, int1 - int2); }
        else { outputError (toClient, "Bad Operator:" + op); }
      } catch (NumberFormatException nfe) {
        String errMsg = "Unable to interpret integer:" + nfe.getMessage();
        System.err.println(errMsg);
        outputError(toClient, errMsg);
        break;
      } catch (Exception e) {
        // internal server error. Try to continue and keep processing
        String errMsg = "Unable to process request: (" + op + " " + int1 + " " +
 int2 + ")";
        System.err.println(errMsg);
        outputError(toClient, errMsg);
      }
    }
    client.close();
    serverSocket.close();
    System.out.println("Server done.");
  }

  static void output(PrintWriter toClient, int value) {
    toClient.println(0);
    toClient.println(value);
  }

  static void outputError(PrintWriter toClient, String error) {
    toClient.println(-1);
    toClient.println(error);
  }
}
```

The while loop has been expanded to process a few operations.

The helper methods, **output** and **outputError**, properly encapsulate the protocol needed to respond to the client's requests. The primary server loop is changed to read three strings from the client, and exit immediately if all three are not present; three variables record the operation and the two values:

OBSERVE:

```
  String op=null, s1=null, s2=null;

  try {
    op = fromClient.readLine();
    s1 = fromClient.readLine();
    s2 = fromClient.readLine();
  } catch (Exception e) {
    System.err.println("Closing Client connection.");
    break;
  }

  // communication terminated prematurely
  if (op == null || s1 == null || s2 == null) {
    System.err.println("Closing Client connection.");
    break;
  }
```

To process the request, a dense **if ... then** statement considers a number of alternatives. In each pre-arranged case (that is, "*", "/", "+", and "-") the server sends the computed result back to the client with **output**. If an unexpected operator is requested, it uses **outputError** to describe the failed attempt:

OBSERVE:

```
     Integer int1=null, int2=null;
     try {
       int1 = Integer.valueOf(s1);
       int2 = Integer.valueOf(s2);

       // support four operations (multiply, divide, add, subtract).
       if (op.equals("*")) { output(toClient, int1 * int2); }
       else if (op.equals("/")) { output (toClient, int1 / int2); }
       else if (op.equals ("+")) { output (toClient, int1 + int2); }
       else if (op.equals ("-")) { output (toClient, int1 - int2); }
       else { outputError (toClient, "Bad Operator:" + op); }
     } catch (NumberFormatException nfe) {
       String errMsg = "Unable to interpret integer:" + nfe.getMessage();
       System.err.println(errMsg);
       outputError(toClient, errMsg);
       break;
     } catch (Exception e) {
       // internal server error. Try to continue and keep processing
       String errMsg = "Unable to process request: (" + op + " " + int1 + " " + i
nt2 + ")";
       System.err.println(errMsg);
       outputError(toClient, errMsg);
     }
```

With a similar set of changes to ComputationClient, you can now have the client send command requests and process the response. The response consists of two lines containing strings. The first value is a 0 (success) or -1 (failure) and the ComputationClient outputs the result or the error message.

CODE TO TYPE: /src/ComputationClient.java

```
import java.io.*;
import java.net.*;
import java.util.*;

public class ComputationClient {

  public static void main(String[] args) throws Exception {
    Socket server = new Socket ("localhost", 7434);

    PrintWriter toServer = new PrintWriter (server.getOutputStream(), true);
    BufferedReader fromServer = new BufferedReader (new InputStreamReader(server
.getInputStream()));

~~    System.out.println("Type string to be echoed back by server");~~
    Scanner sc = new Scanner(System.in);
    while (true~~sc.hasNextLine()~~) {
      ~~String str = sc.nextLine();~~
~~      toServer.println(str);~~
      ~~if (toServer.checkError()) {~~
~~        System.err.println("Server communication no longer available.");~~
~~        break;~~
~~      }~~

      try {
        System.out.println("Command> ");
        String op = sc.nextLine();
        Integer int1 = Integer.valueOf(sc.nextLine());
        Integer int2 = Integer.valueOf(sc.nextLine());

        toServer.println(op);
        toServer.println(int1);
        toServer.println(int2);

        Integer response = Integer.valueOf(fromServer.readLine());
        String value = fromServer.readLine();
~~        System.out.println("Server sends: " + value);~~
        if (response == 0) {
          System.out.println(value);
        } else if (response == -1) {
          System.err.println(value);
        } else {
          System.err.println("Received unknown response from server:" + response
);
        }
      } catch (Exception e) {
        System.err.println("error: " + e.getMessage());
        break;
      }
    }

    server.close();
  }
}
```

Let's see how this revised code performs. Execute ComputationServer and then ComputationClient. Type three lines of input from the earlier table ("+", "6", and "9") and observe that **15** appears as output. Then type three more lines of input ("/", "1", "0") and observe the error that appears. Also, try to submit invalid numbers, such as ("+", "6.4", "13.2"), to see the server's response. Oh wait! If you try this case, the client will detect the error first and immediately exit, so the server never gets the invalid command in the first place. Understanding this concept is important in client/server systems because the client should pre-process commands before they are sent to the server (to make sure they are valid). Even so, the server must have defensive logic in place to deal with invalid input as you saw in the above code.

You must now update TestLongRunning to reflect the updated logic. Modify the **for** loop to test all four of the mathematical operators, as shown:

CODE TO TYPE: /test/TestLongRunning.java

```
import java.io.*;
import java.net.*;
import junit.framework.TestCase;

/** Validate that server processes a succession of client requests. */
public class TestLongRunning extends TestCase {

  public void test1000() throws Exception {
    // open up a server in its own thread of execution.
    new Thread(){
      public void run() {
        try {
          ComputationServer.main(new String[]{});
        } catch (IOException ioe) {
          fail("Unable to start server.");
        }
      }
    }.start();

    // connect away and retrieve input
    Socket server = new Socket ("localhost", 7434);

    PrintWriter toServer = new PrintWriter (server.getOutputStream(), true);
    BufferedReader fromServer = new BufferedReader (new InputStreamReader(server
.getInputStream()));

    String[] ops = { "+", "-", "/", "*" };
    for (int i = 0; i < 1000; i++) {
      System.out.println(i);
      toServer.println(ops[i%4]);
      toServer.println(i);
      toServer.println(1);

      int response = Integer.valueOf(fromServer.readLine());
      int value = Integer.valueOf(fromServer.readLine());
      ~~assertEquals (i, value);~~
      assertEquals (0, response);
      switch (i%4) {
        case 0: assertEquals (i+1, value); break;
        case 1: assertEquals (i-1, value); break;
        case 2: assertEquals (i, value); break;
        case 3: assertEquals (i, value); break;
      }
    }

    server.close();
  }
}
```

Terminate any console sessions that are running, and verify that your code works by running the revised JUnit test case. And that's it for this lab. Let's review:

- You developed a stripped-down client/server application.
- You learned how to use JUnit test cases together with the EclEmma code coverage plugin to identify the code that runs during testing. With this information, you can make informed decisions about additional test cases to write.

Eclipse Concepts

This lab included screenshots showing how to use Eclipse to perform common tasks, such as creating new source folders, packages, and classes. For the rest of this course, we'll assume that you can complete those tasks without specific guidance. For the record, here are the Eclipse tasks you will use for the duration of this course:

- Create a new source folder.

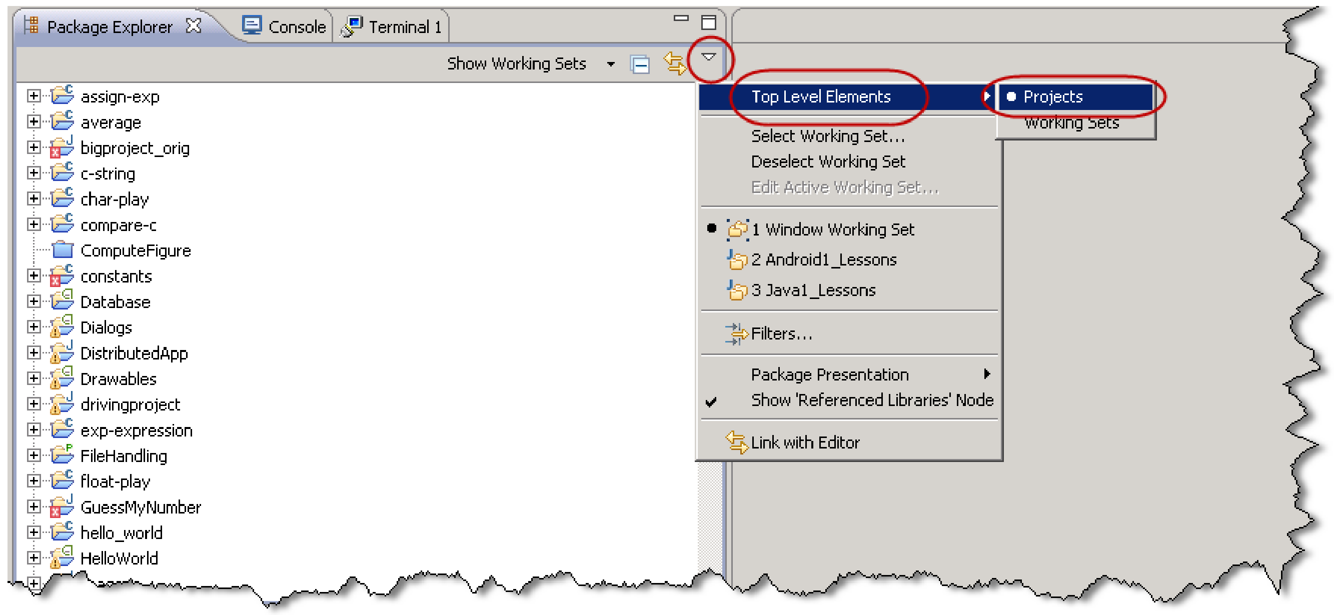
- Create a new package in a source folder.
- Create a new class.
- Create a new interface.
- Execute a Java class.
- Launch the EclEmma plugin.
- Launch all JUnit test cases.
- Launch individual JUnit test case.

Doing Your Homework

For each lab, there are quiz questions and project objectives for you to complete to demonstrate your understanding of the lab material.

Most of the homework objectives in this course will require you to modify the example projects from the lab, but the next lab will continue with the example project as we left it in the previous lab, so you'll want to copy the example projects to a new "branch" project to submit to your instructor for each lab's homework assignment.

To copy the DistributedApp project to another project, in the Package Explorer, click the down-pointing white arrow and select **Top Level Elements | Projects**:

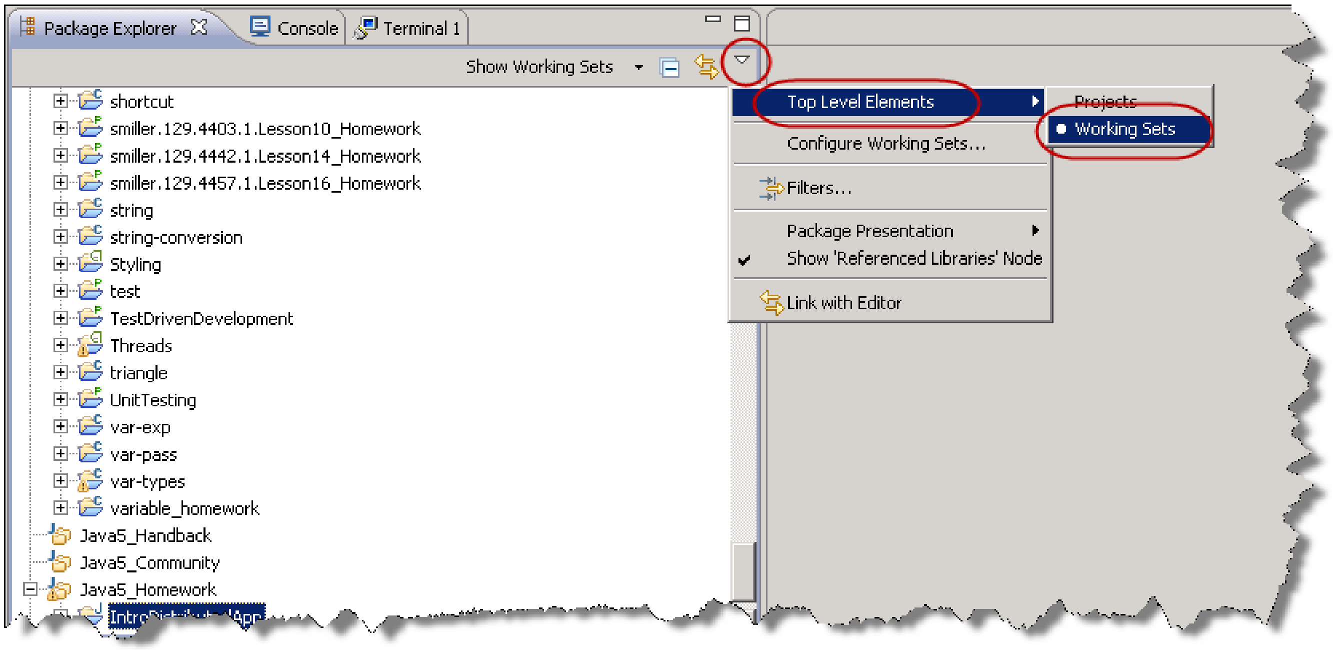


Find your **DistributedApp** project in the list, right-click it, and select **Copy**.

Select **Edit | Paste** from the Eclipse top menu, and give the copy a new name, such as **IntroDistributedApp**.

Right-click the new **IntroDistributedApp** project and assign it to the **Java5_Homework** working set.

Click the downward-pointing white arrow again and select **Top Level Elements | Working Sets**:



And just like that you're on your way! See you in the next lesson...

Server Essentials

Lesson Objectives

In this lesson you will:

- use threads within a server to support multiple connecting clients.
- identify exceptional problems that occur in the communication between clients and servers.
- write code to exercise the performance of a client/server system.

Image Annotation Repository Application

Our goal for this course is to construct a software system that enables groups of users to upload image files to a shared repository and browse through those images.

When you start a project, you'll want to identify your specific requirements in advance. Often though, project requirements change midstream, so you need to learn to build systems with designs that are flexible enough to support unexpected change.

One way to think about this aspect of the application is to compare your process to the process an architect uses when building an arch. An architect couldn't build an arch without first assembling a scaffolding to support the arch as it is being built. Of course, once the scaffolding is removed, your average person won't know exactly how the arch was built and is left to wonder at how such an amazing structure was built! A similar situation exists for users of the application that you're about to build. They'll use it, but they probably won't understand how was built. After you conquer the labs in this course, you'll be a kind of architect and you'll know exactly how ths particular client/server system was built.

This table illustrates the requirements of the application you'll be developing:

R#	Description
R1	Server must allow up to 30 concurrent users to connect and browse the images stored there.
R2	Client must be able to support any of the standard built-in Java image formats (such as PNG or JPG).
R3	Server can be configured to limit the maximum size of any individual image file (default: 5MB).
R4	Server can be configured to limit the total number of files stored on the shared repository (default: 1,000).
R5	A user connecting to a server must provide a user name and password.
R6	A user can upload up to a fixed number of images to the repository (default: 100).
R7	A user can delete any image that he has added to the repository; a user cannot delete images added by another user.
R8	A user can self-register an account with the server.
R9	During the client-server communication, the user's password never appears in plaintext format.
R10	A user account is considered *inactive* if the user has not connected to the server within a fixed time period (default: 14 days).
R11	Each user account has a unique string identifier composed of alphanumeric characters (a-zA-Z0-9). The server only stores the hashed value of the password and therefore does not know it.

With this set of established requirements as your blueprint, you will implement a core set of Java classes to construct a fully functioning server and client. Instead of just reading the code for a fully implemented and constructed system, you are going to build the system, step by step. Throughout the process, you will maintain a working system that offers a subset of the desired functionality. Of course, the proposed set of labs is just one way to build this software application, but it will give you insight into the overall development of an application. As you work through the labs, you may even question the design decisions that have been made along the way—this is good! Your skepticism is a sign of your expanding knowledge and ability to come up with all kinds of different ways to solve problems. We want to ask questions, experiment and make mistakes. The essence of becoming a professional is learning from those mistakes. We'll always explain why the code was designed in the way that it was and ultimately you can determine for yourself which way works best for your needs.

Multi-Threaded Server Application

Given our current knowledge of single-threaded servers, we know we need to satisfy requirement **R1** to ensure that the server application can allow up to 30 concurrent users to connect and browse the images stored there. Your first task then, is to write a multi-threaded server and test its execution using JUnit. The Java threading model will help you to accomplish that. Let's break this task into smaller units of work:

- Construct a server object from the prior lab's standalone server. (This will simplify how servers are coded and tested.)
- Extract the server code that processes requests so it can be executed within its own **RepositoryThread** class.
- Write a separate **ServerLauncher** class to launch the server.

In your **DistributedApp** project's **/src** folder, create a new package named **server**.

In the **server** package, create a **RepositoryServer** class as shown:

CODE TO TYPE: /src/server/RepositoryServer.java

```java
package server;

import java.io.*;
import java.net.*;

public class RepositoryServer {
  ServerSocket serverSocket = null;
  int          state        = 0;

  public void bind() throws IOException {
    serverSocket = new ServerSocket(9172);
    state = 1;
  }

  public void process() throws IOException {
    while (state == 1) {
      Socket client = serverSocket.accept();

      new RepositoryThread(client).start();
    }

    shutdown();
  }

  void shutdown() throws IOException {
    if (serverSocket != null) {
      serverSocket.close();
      serverSocket = null;
      state = 0;
    }
  }
}
```

This class represents a server object that responds to client requests for the Image Repository. The primary methods of this object are bind() (which initializes a ServerSocket object to listen to client requests), process() (which spawns threads to respond to client requests), and shutdown (which has the server shut down so it no longer processes requests). You'll see a compilation error because you have not yet created the **RepositoryThread** class.

RepositoryServer manages a ServerSocket object that is associated with a specific port number on the machine on which it executes. A network *port number* is like a post office box number used to direct mail to a specific recipient. Clients seeking to connect to a specific server must know both the hostname and the specific port number used by that server.

OBSERVE:

```
public void process() throws IOException {
while (state == 1) {
  Socket client = serverSocket.accept();

  new RepositoryThread(client).start();
}

void shutdown() throws IOException {
  if (serverSocket != null) {
    serverSocket.close();
    serverSocket = null;
    state> = 0;
  }
}
```

RepositoryServer constructs a ServerSocket to listen for client requests; when **process()** is invoked, it responds to client requests to connect, as long as the **state variable is 1**. Once **shutdown()** is invoked, **state** is set to 0 and the **process()** method can terminate. Note that the only way for the server to exit is to have one of the instantiated RepositoryThreads call **shutdown()**. Of course, you can still terminate RepositoryServer execution externally, in Eclipse.

The **process()** method loops repeatedly while RepositoryServer is accepting connections (while **state == 1**). The call to **serverSocket.accept()** blocks until client code requests a socket connection, at which point it constructs and returns a Socket object representing the communication channel to/from that client. Finally, when **shutdown()** is invoked, the socket is closed and **state is reset to 0**.

Client/Server Protocol

Next, we'll define the protocol between the client and the server. To start, we'll set it up so the client sends a request as a single string on a line by itself, and the server responds with two string lines: the first line contains 0 for success or -1 for failure. The second line contains the results of the request (for success) or an error message (for failure). Admittedly, this protocol structure is not going to last in the long run, but rather than getting bogged down in developing complex protocols right away, we can start with this basic concept and then figure out later how to improve it.

The first client request will be to return the number of images in the repository. Since you're starting with an empty repository (right?), the response will be 0 until you add functionality to upload images. This lab focuses on changes that are made to the server, so the client code we see will be intentionally artificial (we haven't constructed the client code yet, so we have "pretend" code that looks like it comes from a client, which doesn't yet exist). In the next listing, the client makes three SIZE requests, sleeping for a second in between each request. The Thread.sleep invocations are introduced to demonstrate that the server is able to handle multiple client requests simultaneously.

In the **/test** folder, create a **client** package, and in it, create a **RepositoryClient** class. This class is in the **/test** folder because it will serve as "scaffolding" to use during testing:

CODE TO TYPE: /test/client/RepositoryClient.java

```
package client;

import java.io.*;
import java.net.*;

public class RepositoryClient {

  public static void main(String[] args) throws Exception {
    Socket server = new Socket ("localhost", 9172);

    PrintWriter toServer = new PrintWriter (server.getOutputStream(), true);
    BufferedReader fromServer = new BufferedReader (new InputStreamReader(server
.getInputStream()));

    for (int num = 0; num < 3; num++) {
      toServer.println("SIZE");
      if (!toServer.checkError()) {
        int response = Integer.valueOf(fromServer.readLine());
        String value = fromServer.readLine();
        if (response == 0) {
          System.out.println(num + ": Number of Images: " + value);
        } else if (response == -1) {
          System.err.println(value);
        } else {
          System.err.println("Received unknown response:" + response);
        }
      }
    }

    server.close();
  }
}
```

Let's look closer at the client/server communication pattern.

OBSERVE:

```
  public static void main(String[] args) throws Exception {
    Socket server = new Socket ("localhost", 9172);

    PrintWriter toServer = new PrintWriter (server.getOutputStream(), true);
    BufferedReader fromServer = new BufferedReader (new InputStreamReader(server
.getInputStream()));

    toServer.println("SIZE");
    if (!toServer.checkError()) {
      int response = Integer.valueOf(fromServer.readLine());
      String value = fromServer.readLine();
      if (response == 0) {
        System.out.println(num + ": Number of Images: " + value);
      } else if (response == -1) {
        System.err.println(value);
      } else {
        System.err.println("Received unknown response:" + response);
      }
    }
```

After validating that a SIZE request was sent to the server properly using **checkError**, the code reads two consecutive String lines from the server (**response** and **value**).

The **main method is declared to throw an Exception**, which simplifies the method by avoiding the need to write exception handlers; to see what you were able to avoid, temporarily eliminate the **throws Exception** declaration and you'll see that the code contains six compiler errors. It's common to take shortcuts like this in such scaffolding code. Put the throws declaration back in place. Now you're ready to complete the server.

In the **/src** folder **server** package, create the **RepositoryThread** class as shown:

CODE TO TYPE: /src/server/RepositoryThread.java

```
package server;

import java.io.*;
import java.net.*;

public class RepositoryThread extends Thread {

  Socket client;
  BufferedReader fromClient;
  PrintWriter toClient;

  RepositoryThread (Socket s) throws IOException {
    fromClient = new BufferedReader(new InputStreamReader(s.getInputStream()));
    toClient = new PrintWriter (s.getOutputStream(), true);
    client = s;
  }

  public void run() {
    try {
      while (true) {
        String request = fromClient.readLine();
        if (request == null) {
          break;
        }

        if (request.equals("SIZE")) {
          output("0");
        } else {
          // internal server error. Try to continue and keep processing
          outputError("Unable to process request: " + request);
          continue;
        }
      }
    } catch (IOException ioe) {
      System.err.println("Thread processing terminated:" + ioe.getMessage());
    }

    try {
      fromClient.close();
      toClient.close();
      client.close();
    } catch (IOException ioe) {
      System.err.println("Unable to close connection:" + ioe.getMessage());
    }
  }

  void output(String result) {
    toClient.println(0);
    toClient.println(result);
  }

  void outputError(String error) {
    toClient.println(-1);
    toClient.println(error);
  }
}
```

You will recognize most of this class from the prior lab—it's repackaged here as a standalone class that extends the java.lang.Thread class. Subclasses of **Thread** provide a run() method that executes once the thread starts. In the case of **RepositoryThread**, the **run()** method retrieves the request from the client as a single string on a line by itself. Currently, the class only understands SIZE requests, in which case it returns "0" as the value for the result.

In our example, each **RepositoryThread** object maintains its own state: a Socket object is used to communicate with the client, a BufferedReader object is used to retrieve String input from the client, and a PrintWriter object is used to send String responses to the client.

OBSERVE:

```
public void run() {
  try {
    while (true) {
      String request = fromClient.readLine();
      if (request == null) {
        break;
      }
    ...
```

The run() method contains logic that you've already seen. In Java, calling the start() method on a thread (as is done by RepositoryServer) causes that thread to execute its run() method. As long as the thread is reading requests from the client, it will continue to execute. However, once **fromClient.readLine()** returns **null**, the loop will break and the thread will close the socket, thereby closing communication to the client. The RepositoryThread class maintains the three class variables necessary to process it. Finally, it has two helper output methods.

OBSERVE:

```
RepositoryThread (Socket s) throws IOException {
  fromClient = new BufferedReader(new InputStreamReader(s.getInputStream()));
  toClient = new PrintWriter (s.getOutputStream(), true);
  client = s;
}
```

When the RepositoryThread object is constructed, it sets up the **BufferedReader** and **PrintWriter** objects. Its run() method executes repeatedly, fetching request strings from the remote client and returning output, using the output() and outputError() helper methods. The thread terminates when it receives **null** as input from the client (which happens when the client severs the connection).

Several suboptimal decisions have been made in the implementation of RepositoryThread. One of them will cause you to have to modify this class every time a new request is added to the client/server protocol. This would be like having to upgrade the circuit box in your house whenever you bought a new appliance. In later labs, you'll eliminate this problem in your code. A second issue arises because the protocol is overly simplistic, and as such, you will need to upgrade the way requests and responses are issued between the client and the server. You have to do this because image data sent from the server to the client will be in binary format, and you will be unable to confine the bytes of an image to a single line of text sent to the client. Still, there is no easy way to completely implement any application, so it's often best to chart a slow and steady path towards your end goal.

In the **/src** folder **server** package, create a **ServerLauncher** class as shown to instantiate and configure the RepositoryServer object:

CODE TO TYPE: /src/server/ServerLauncher.java

```
package server;

public class ServerLauncher {

  public static RepositoryServer create() throws Exception {
    RepositoryServer server = new RepositoryServer();
    server.bind();
    return server;
  }

  public static void main(String[] args) throws Exception {
    RepositoryServer server = create();

    System.out.println("Server awaiting client connections");
    server.process();
    System.out.println("Server shutting down.");
  }
}
```

This class contains logic that should be separate from the code that makes up the server.

Can this code handle thirty concurrent requests? The Java documentation for the ServerSocket class confirms that its ServerSocket(int) constructor supports up to 50 incoming connections, well above that requirement.

Which classes should have main methods? I recommend extracting all **public static void main(String[] args)** methods so they exist in "launching" classes such as ServerLauncher, otherwise they will be buried too deep within your code base. Note that the **main()** method is allowed to throw an Exception, which simplifies the code. Finally, the **create()** method does everything while setting up a server, except for initiating processing; this method will be useful during testing.

Testing and Code Coverage

Now that the server has been properly restructured, you can write test cases to validate the code's implementation. The primary test case to review is testMultipleClients() because it demonstrates three clients connecting to the same server. These test cases will exist within the **/test** folder, so you need to create a **server** package under that source folder.

In the **/test** folder **server** package, create the **TestServer** JUnit test case as shown:

CODE TO TYPE: /test/server/TestServer.java

```
package server;

import java.io.*;
import client.*;
import junit.framework.TestCase;

public class TestServer extends TestCase {
  public void testMultipleClients() throws Exception {
    RepositoryServer server = launchServer();

    launchClient();
    launchClient();
    launchClient();

    // wait until everything done.
    Thread.sleep(10000);

    server.shutdown();
    assertEquals (0, server.state);
  }

  public static void launchClient() {
    new Thread() {
      public void run() {
        try {
          RepositoryClient.main(new String[]{});
        } catch (Exception e) {
          System.err.println("Unable to launch test client.");
        }
      }
    }.start();
  }

  public static RepositoryServer launchServer() throws Exception {
    final RepositoryServer server = ServerLauncher.create();
    assertEquals (1, server.state);
    new Thread() {
      public void run()  {
        try {
          server.process();
        } catch (IOException ioe) {
          System.err.println("Server completed.");
        }
      }
    }.start();

    // wait until server is ready.
    Thread.sleep(2000);

    return server;
  }
}
```

Make sure you understand the two helper methods in this test case, because you'll use them in all of your future test cases.

The launchServer() method builds on the the **create method in ServerLauncher** to instantiate and bind a RepositoryServer object. However, you can't just run the **process method**, because that "blocks" all activity while it waits for client requests; you need to execute **process** within its own **Thread**, as shown below. This logic instantiates a **new Thread** and executes its **start** method, which ultimately forces the **run** method to execute, thus having this thread block while the remainder of the launchServer() method can continue. So the method has to wait for two seconds (we chose this amount of time arbitrarily), for the server to be properly instantiated.

OBSERVE:

```
public static RepositoryServer launchServer() throws Exception {
  final RepositoryServer server = ServerLauncher.create();
  assertEquals (1, server.state);
  new Thread() {
    public void run()  {
      try {
        server.process();
      } catch (IOException ioe) {
        System.err.println("Server completed.");
      }
    }
  }.start();

  // wait until server is ready.
  Thread.sleep(2000);
  assertEquals(1, server.state);

  return server;
}
```

In similar fashion, launchClient spawns a **new thread** that executes **RepositoryClient** in its own thread of control:

OBSERVE:

```
public static void launchClient() {
  new Thread() {
    public void run() {
      try {
        RepositoryClient.main(new String[]{});
      } catch (Exception e) {
        System.err.println("Unable to launch test client.");
      }
    }
  }.start();
}
```

When you run the TestServer test case, each of the three clients sends its first request to the server for processing, after which each sleeps for a second. You can see the requests are intermingled in the output, but each client still has three requests. Thus, these requests are all being handled concurrently. You need to wait 10 seconds for the entire test case to complete:

OBSERVE: Output from TestServer

```
0: Number of Images: 0
0: Number of Images: 0
1: Number of Images: 0
1: Number of Images: 0
0: Number of Images: 0
2: Number of Images: 0
2: Number of Images: 0
1: Number of Images: 0
2: Number of Images: 0
Server Completed.
```

Generate EclEmma code coverage for the TestServer test case. You will have to wait the full ten seconds until the test case completes sleeping, but then you will see that you have increased code coverage of RepositoryServer to over 80% (our target threshold). RepositoryThread is still stuck at less than 60% and you can see that all of the non-executed code blocks occur in error handling situations. The only way to demonstrate their execution is to refine the client code (as you did in this lab with the server). You will do that in the next lab.

Because the **testMultipleClients** test case depends on the proper execution of Thread.sleep() statements, such a test case is not ideal. We differentiate between several different types of test cases: those that validate proper execution, those that are performance tests that evaluate the execution time of given functionality, and those that create error or exceptional situations to determine the robustness of the implementation.

Performance Tests

Is it possible to write a JUnit test case to validate requirement **R1**? Well, it's not entirely appropriate to do so. The R1 requirement is best validated using a performance stress test rather than a JUnit test case, because the purpose of a JUnit test case is to validate the correct execution of the code.

Create a new source folder named **performance**, and in it, create a **server** package. In this folder, you'll place code that validates performance (not correctness) tests. Another reason to separate these classes from JUnit test cases is that Eclipse offers a convenient capability to run "All JUnit test cases" quickly for a project or a source code folder; these performance classes should not be executed every time.

In the **/performance** folder **server** package, create the **ConcurrentUserPerformance** class. This is *not* a JUnit test case, but you can take advantage of the helper static methods you have already written.

CODE TO TYPE: /performance/server/ConcurrentUserPerformance.java

```
package server;

import java.io.*;
import java.net.*;

public class ConcurrentUserPerformance {
  public static void main(String[] args) throws Exception {
    Socket[] connections = new Socket[40];
    PrintWriter[] writers = new PrintWriter[40];
    BufferedReader[] readers = new BufferedReader[40];

    RepositoryServer server = TestServer.launchServer();

    for (int i = 0; i < connections.length; i++) {
      connections[i] = new Socket ("localhost", 9172);
      writers[i] = new PrintWriter (connections[i].getOutputStream(), true);
      readers[i] = new BufferedReader (new InputStreamReader(connections[i].getI
nputStream()));
    }

    for (int i = 0; i < connections.length; i++) {
      for (int j = 0; j < connections.length; j++) {
        writers[j].println("SIZE");
        String rc = readers[j].readLine();
        String val = readers[j].readLine();
        System.out.println("C" + j + " communicates (" + rc + ":" + val + ")");
      }
    }

    for (int i = 0; i < connections.length; i++) {
      connections[i].close();
    }
    server.shutdown();
  }
}
```

ConcurrentUserPerformance does not use threads (aside from the thread executing the RepositoryServer), rather it makes a fixed number of client connections (in this case 40) and exercises these connections in serial fashion:

OBSERVE:

```
   Socket[] connections = new Socket[40];
   PrintWriter[] writers = new PrintWriter[40];
   BufferedReader[] readers = new BufferedReader[40];

   RepositoryServer server = TestServer.launchServer();

   for (int i = 0; i < connections.length; i++) {
     connections[i] = new Socket ("localhost", 9172);
     writers[i] = new PrintWriter (connections[i].getOutputStream(), true);
     readers[i] = new BufferedReader (new InputStreamReader(connections[i].getIn
putStream()));
   }
```

This code **instantiates a RepositoryServer**, then constructs and connects 40 sockets to that server. The **writers[]** and **readers[]** arrays store the PrintWriter and BufferedReader objects used to communicate requests to the server and read responses back from the server.

After the socketsset of sockets comes two nested **for** loops that generate 40 * 40 = 1600 SIZE requests to be processed by the server. Each request consists of printing "SIZE" to **writers[j]**, followed by reading two string responses from **readers[j]**:

OBSERVE: How to generate and process 1,600 requests

```
  for (int i = 0; i < connections.length; i++) {
    for (int j = 0; j < connections.length; j++) {
      writers[j].println("SIZE");
      String rc = readers[j].readLine();
      String val = readers[j].readLine();
      System.out.println("C" + j + " communicates (" + rc + ":" + val + ")");
    }
  }
```

Run ConcurrentUserPerformance to verify that each of the 40 connections was able to submit repeated SIZE requests.

Most excellent work! That's the end of this lesson. Go ahead and work through the homework and project like you always do, and I'll see you in the next lesson!

Client Essentials

Lesson Objectives

In this lesson you will:

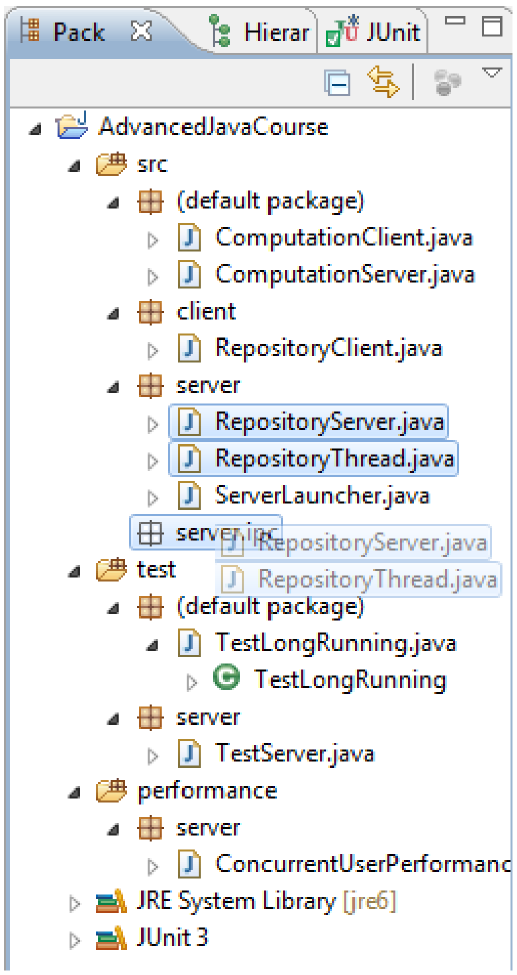
- design an inter-process communication (IPC) layer from the existing classes on top of which to design a Java Swing client to connect to remote server.

Preparing an Inter-Process Communication Layer

So far, we've developed a multi-threaded RepositoryServer and a command-line RepositoryClient that issues requests to that server. Now, we'll design a Graphical User Interface (GUI) client that provides capabilities users expect. If you review the full set of capabilities that you'll need for the final application, you'll know where to start! (The list of requirements can be found in a table at the beginning of the serverEssentials lesson.) In this lab you'll implement only the primary window and splash screen. In the process, you'll restructure your code to encapsulate the client's interactions to the server in an *Inter-Process Communication* (IPC) layer.

The IPC layer is the infrastructure used to communicate between the client and server. We won't need to change that code much once it's created, and it's helpful to know that it's stable code. To guard against unexpected changes, modify the existing classes to encapsulate and hide their implementation.

In the **/src** folder, create a package named **server.ipc**, and move the **RepositoryThread** and **RepositoryServer** classes into this package. To move the classes, hold down the **Ctrl** key, click on the two classes (**RepositoryThread** and **RepositoryServer**), then drag both classes into the new **server.ipc** package. The action on the screen will look like this:



When prompted by Eclipse, check the **Update references to 2 moved element(s)** box.

Ideally, you'll design interfaces against which to program, and then you can forget the underlying implementation details and get to the business of making the application "do stuff." Instead of thinking about sockets and input/output streams, you'll work with an interface that encapsulates all protocol behavior into a single interface. Before you move on, fix the code that broke because of the refactoring. Not surprisingly, your test case and performance test no longer compile.

Create **server.ipc** subpackages in the **/test** and **/performance** folders. Now move **TestServer** into the **/test/server.ipc** package and **ConcurrentUserPerformance** into the **/performance/server.ipc** package.

Before continuing, take some time to clean up some code that you won't need in the future, specifically, the classes that were placed in the default package. Delete **ComputationClient** and **ComputationServer** in the **/src** folder, and **TestLongRunning** in the **/test** folder.

Create the **IProtocolHandler** interface in the **/src/server.ipc** package as shown:

/src/server.ipc/IProtocolHandler.java

```
package server.ipc;

import java.io.*;

public interface IProtocolHandler {

  /** Process the protocol using socket's input and output. Return false to terminate,
true to continue. */
  boolean process(BufferedReader fromSocket, PrintWriter toSocket);
}
```

As is common in object-oriented projects, the name of the interface starts with a capital **I** to clearly identify that it is an interface. This interface enables the real logic of the protocol to be "outsourced" to a handler class that you're about to design. In this way, the IPC layer is responsible only for making the initial connection; after that, a protocol handler will know when to read and write from the socket involved in the communication.

Note Eclipse offers a helpful layout when you have multiple packages using a hierarchical naming pattern. To "nest" packages in the Package Explorer properly, click the white drop-down arrow and select **Package Presentation | Hierarchical**.

In the **/src/server** package, create a class named **ProtocolHandler**, that implements **IProtocolHandler**. You'll recognize most of this code from the **RepositoryThread** class that you wrote in the last lab:

CODE TO TYPE: /src/server/ProtocolHandler.java

```
package server;

~~import java.io.BufferedReader;~~
~~import java.io.PrintWriter;~~
~~import server.ipc.IProtocolHandler;~~
import java.io.*;
import server.ipc.*;

public class ProtocolHandler implements IProtocolHandler {

  @Override
  public boolean process(BufferedReader fromSocket, PrintWriter toSocket) {
    ~~// TODO Auto-generated method stub~~
~~    return false;~~
    try {
      String request = fromSocket.readLine();
      if (request == null) {
        return false;
      }

      if (request.equals("SIZE")) {
        output(toSocket, "0");
      } else {
        // internal server error. Try to continue and keep processing
        outputError(toSocket, "Unable to process request: " + request);
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
      return false;
    }

    return true;
  }

  void output(PrintWriter toSocket, String value) {
    toSocket.println(0);
    toSocket.println(value);
  }

  void outputError(PrintWriter toSocket, String error) {
    toSocket.println(-1);
    toSocket.println(error);
  }
}
```

Look over the **process()** method. It **reads a single line of input from the BufferedReader** associated with the connecting client's socket. Then, based on the input received, the server generates a **successful response** using **output** or a **failed response** using **outputError**:

OBSERVE:

```
  public boolean process(BufferedReader fromSocket, PrintWriter toSocket) {
    try {
      String request = fromSocket.readLine();
      if (request == null) {
        return false;
      }

      if (request.equals("SIZE")) {
        output(toSocket, "0");
      } else {
        // internal server error. Try to continue and keep processing
        outputError(toSocket, "Unable to process request: " + request);
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
      return false;
    }

    return true;
  }
```

Currently, **process()** only handles SIZE requests. While the core logic of ProtocolHandler is identical to the former RepositoryThread implementation, it should be encapsulated in its own class as shown, because you don't want the logic needed to process messages from the client to be buried deeply within low-level IPC code.

For your final modification, you need to tell the RepositoryThread about your ProtocolHandler object. Start by modifying **ServerLauncher** as shown below. (You will modify the RepositoryServer constructor to take in an instance of the ProtocolHandler class to be used to interpret the protocol):

CODE TO TYPE: /src/server/ServerLauncher.java

```
package server;

import server.ipc.*;

public class ServerLauncher {

  public static RepositoryServer create() throws Exception {
    RepositoryServer server = new RepositoryServer(new ProtocolHandler());
    server.bind();
    return server;
  }

  public static void main(String[] args) throws Exception {
    RepositoryServer server = create();

    System.out.println("Server awaiting client connections");
    server.process();
    System.out.println("Server shutting down.");
  }
}
```

The code will result in a compiler error until you make a few modifications to the **RepositoryServer** class. Until then, RepositoryServer will hold onto the ProtocolHandler object and use it whenever a client connects. You need to add a constructor to take the protocol object and then update the code that launches the RepositoryThread objects for processing:

CODE TO TYPE: /src/server.ipc/RepositoryServer

```
package server.ipc;

import java.io.*;
import java.net.*;

public class RepositoryServer {
  ServerSocket serverSocket = null;
  int          state        = 0;
  IProtocolHandler protocolHandler;

  public RepositoryServer(IProtocolHandler ph) {
    protocolHandler = ph;
  }

  public void bind() throws IOException {
    serverSocket = new ServerSocket(9172);
    state = 1;
  }

  public void process() throws IOException {
    while (state == 1) {
      Socket client = serverSocket.accept();

      new RepositoryThread(client, protocolHandler).start();
    }

    shutdown();
  }

  void shutdown() throws IOException {
    if (serverSocket != null) {
      serverSocket.close();
      serverSocket = null;
      state = 0;
    }
  }
}
```

The final change will be to the **RepositoryThread** class:

CODE TO TYPE: /src/server.ipc/RepositoryThread

```
package server.ipc;

import java.io.*;
import java.net.*;

public class RepositoryThread extends Thread {
  Socket client;
  BufferedReader fromClient;
  PrintWriter toClient;
  IProtocolHandler handler;

  RepositoryThread (Socket s, IProtocolHandler h) throws IOException {
    fromClient = new BufferedReader(new InputStreamReader(s.getInputStream()));
    toClient = new PrintWriter (s.getOutputStream(), true);
    client = s;
    handler = h;
  }

  public void run() {
    ~~try {~~
      ~~while (true) {~~
        ~~String request = fromClient.readLine();~~
        ~~if (request == null) {~~
          ~~break;~~
        ~~}~~

        ~~if (request.equals("SIZE")) {~~
          ~~output("0");~~
        ~~} else {~~
          ~~// internal server error. Try to continue and keep processing~~
          ~~outputError("Unable to process request: " + request);~~
          ~~continue;~~
        ~~}~~
      ~~}~~
    ~~} catch (IOException ioe) {~~
      ~~System.err.println("Thread processing terminated:" + ioe.getMessage());~~
    ~~}~~
    // have handler manage the protocol until it decides it is done.
    while (handler.process(fromClient, toClient)) {

    }

    try {
      fromClient.close();
      toClient.close();
      client.close();
    } catch (IOException e) {
      System.err.println("Unable to close connection:" + e.getMessage());
    }
  }

  ~~void output(String result) {~~
    ~~toClient.println(0);~~
    ~~toClient.println(result);~~
  ~~}~~

  ~~void outputError(String error) {~~
    ~~toClient.println(-1);~~
    ~~toClient.println(error);~~
  ~~}~~
}
```

You deleted all of the logic that had previously processed the protocol and replaced it with a simple loop to use the new protocol handler code. The RepositoryThread class recognizes the ProtocolHandler object as an instance of a class that implements IProtocolHandler, which insulaties the IPC layer from the actual business logic further.

Validate that all test cases and performance tests operate properly. You can do this in two ways: to run all test cases associated with your project, right click on the **DistributedApp** project and select menu item **Run As | JUnit Test**. Alternatively, you can right-click, one by one, on the test and performance packages, and select the **Run As | JUnit Test** menu item.

Preparing a Standalone Client GUI

The clients we've written so far are not very useful. We need to add a Graphical User Interface (GUI). You are already familiar with the Java Swing approach for developing Java GUIs. You only need to understand part of the full Swing API to develop reasonably usable client applications. To design the skeleton of a client application, we'll:

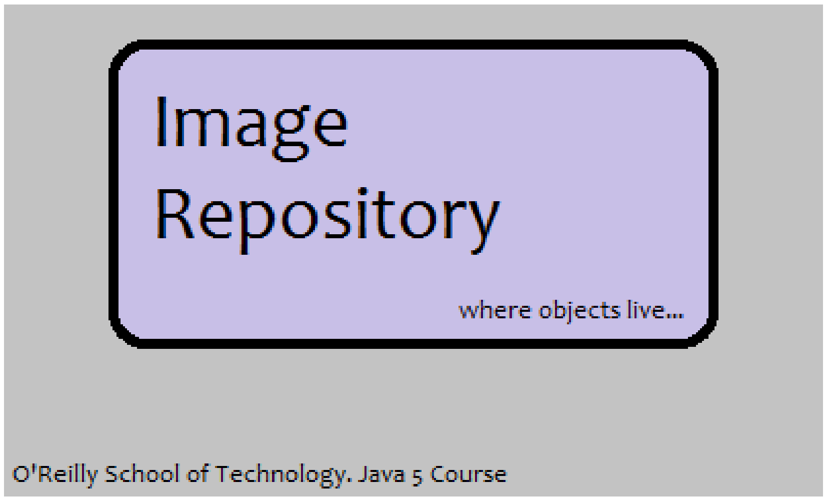
1. write a splash screen that flashes briefly when the application launches.
2. write a Menu-based Swing application with all commands in place.
3. write code to read from (and write to) the user directory to store preferences to use whenever the application runs.

Let's get started!

Writing a Java Splash Screen

The Java Virtual Machine (VM) can be cumbersome at startup, which makes it challenging to write a splash screen. Fortunately, with the Java SE 6 release, the Java VM added a command-line argument to immediately display a pre-selected image in a centered window when launching a Java application. If that were the only capability we had though, it would be a poor splash screen; modern applications often show configuration information or the status of initialization routines as well. You will be able to add logic to manipulate the splash screen. As you can see in the SplashScreen documentation, you can provide an image file (either GIF, JPEG, or PNG) that is displayed immediately upon execution. Even better, once your real windows start appearing, the splash screen automatically hides itself.

In your **DistributedApp** project, create an **/images** folder. Then, download the image abelow s **repositorySplash.png** in the new **/images** folder. (You can modify the image as you like, as long as you keep the same dimensions, especially of the inner rounded rectangle):



To download the image, right-click it, click **Save picture as...**, and navigate to your /images folder (in **Computer/V:/workspace/DistributedApp/images**). To confirm that the image was saved properly, right click on the **/images** folder and select menu item **Refresh**; Eclipse will now show this file within the **/images** folder.

Create a **client** package in the **/src** folder.

Create a **SplashScreenLogic** class in the **/src** folder **client** package. Type the code below, which demonstrates the ability to post messages to the gray space below the "Image Repository" rounded rectangle:

CODE TO TYPE: /src/client/SplashScreenLogic.java

```
package client;

import java.awt.*;

public class SplashScreenLogic {

  public static void update (String s) {
    Graphics g = null;
    SplashScreen splash = SplashScreen.getSplashScreen();
    if (splash != null) {
      g = splash.createGraphics();
    }

    // no splash screen? Well, at least we record the message to stdout.
    if (g == null) {
      System.out.println (s);
      return;
    }

    g.setColor(new Color(195, 195, 195));
    g.fillRect(60, 180, 300, 30);

    g.setColor(Color.black);
    g.drawString(s, 60, 190);
    splash.update();

    g.dispose();
  }
}
```

Swing application programmers often fail to properly dispose of Graphics objects that they have constructed. While this is most common with the getGraphics() method, it can also happen with the **createGraphics()** method as shown here. Here, we invoke **dispose() on Graphics object g** so it doesn't waste system resources.

OBSERVE:

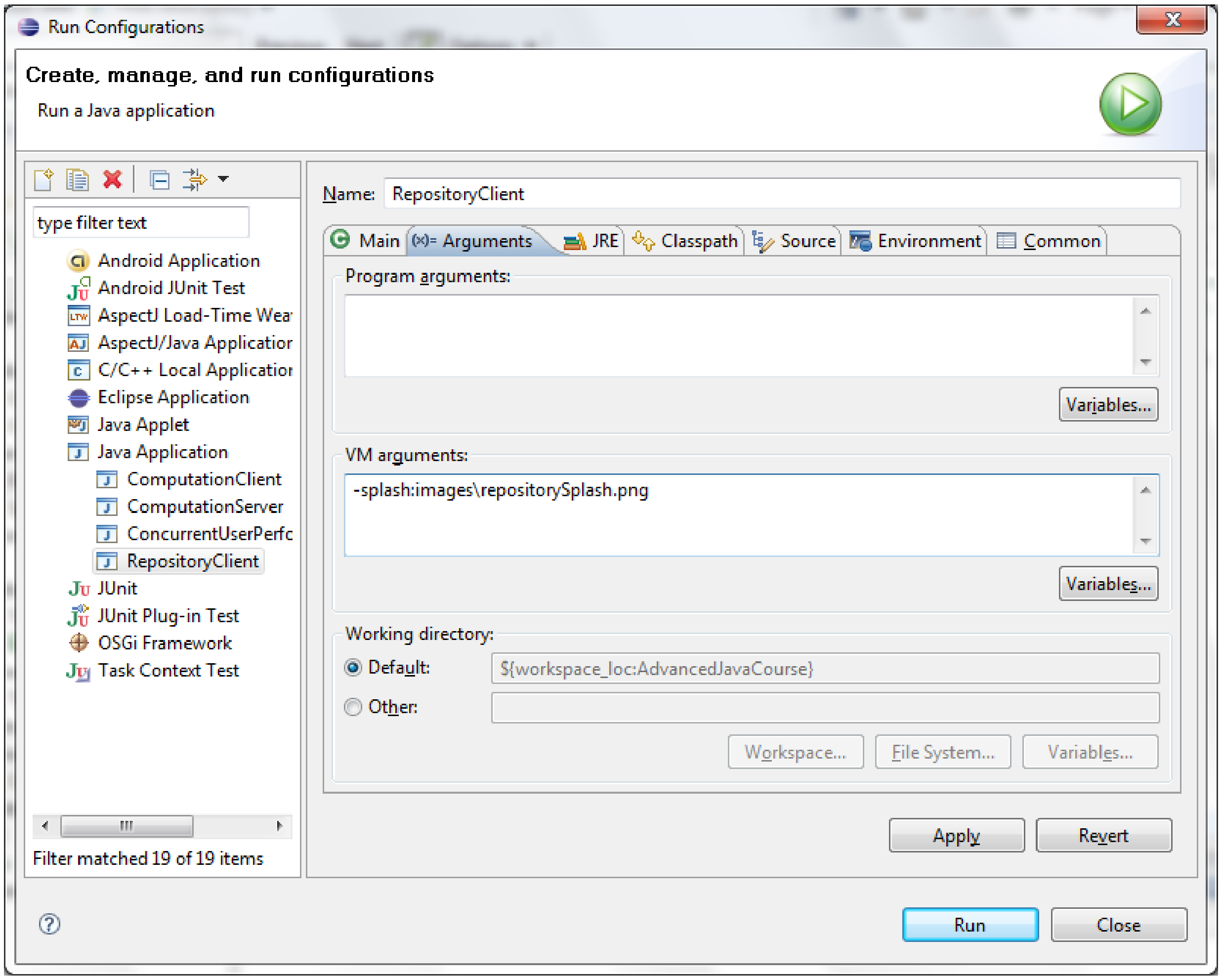
```
  public static void update (String s) {
    Graphics g = null;
    SplashScreen splash = SplashScreen.getSplashScreen();
    if (splash != null) {
      g = splash.createGraphics();
    }

    // no splash screen? Well, at least we record the message to stdout.
    if (g == null) {
      System.out.println (s);
      return;
    }

    g.setColor(new Color(195, 195, 195));
    g.fillRect(60, 180, 300, 30);

    g.setColor(Color.black);
    g.drawString(s, 60, 190);
    splash.update();

    g.dispose();
  }
```

Now, modify **RepositoryClient** so it will write messages to the splash screen:

CODE TO TYPE: /test/client/RepositoryClient

```
package client;

import java.io.*;
import java.net.*;

public class RepositoryClient {

  public static void main(String[] args) throws Exception {
    SplashScreenLogic.update ("connecting to localhost::9172");
    delay(250);
    Socket server = new Socket ("localhost", 9172);

    SplashScreenLogic.update ("connected to localhost::9172");
    delay(250);

    PrintWriter toServer = new PrintWriter (server.getOutputStream(), true);
    BufferedReader fromServer = new BufferedReader (new InputStreamReader(server
.getInputStream()));
    SplashScreenLogic.update ("initializing with server...");
    delay(250);

    for (int num = 0; num < 3; num++) {
      toServer.println("SIZE");
      if (!toServer.checkError()) {
        Integer response = Integer.valueOf(fromServer.readLine());
        String value = fromServer.readLine();
        if (response == 0) {
          System.out.println((num+1) + ": Number of Images: " + value);
        } else if (response == -1) {
          System.err.println(value);
        } else {
          System.err.println("Received unknown response:" + response);
        }
      }
    }

    server.close();
    SplashScreenLogic.update ("closing");
    delay(250);
  }

  /** Delay for a time. */
  static void delay(int ms) {
    try { Thread.sleep(ms); } catch (InterruptedException ie) { }
  }
}
```

Run the **ServerLauncher**, and then the **RepositoryClient**. (We inserted delays to make it possible to read the messages.) If you weren't running a server, you will get an exception, but nothing "graphical" happened. That's because there is still one more step we need to take. To activate the splash screen feature, you need to supply a specific command line argument to the Java VM. This is a bit awkward, but the designers of Java recognized that it would be the best way to avoid the lengthy initialization sequence of the Java VM. From the **Run** menu in Eclipse, select **Run Configurations...** and, under the **Java Application** grouping, locate the **RepositoryClient** entry (which should be the last one executed). Switch to the **Arguments** tab and enter **-splash:images\repositorySplash.png** in the VM Arguments section:



Name: RepositoryClient

VM arguments:

```
-splash:images\repositorySplash.png
```

Working directory: Default: ${workspace_loc:AdvancedJavaCourse}

Click **Apply** and then **Run**. The splash screen should appear; if you see an exception, you might need to run **ServerLauncher** and then relaunch **RepositoryClient** to see the subsequent steps. Of course this is just an example; the whole application lives "within the splash screen," but you get the point.

After a long coding session, be sure to run all existing JUnit test cases to verify that they all pass. (First, make sure to terminate the execution of any RepositoryServer that may be running.) Now, take a break, stretch your legs, then move on to the homework for this lesson!

Writing Your Swing Application

Lesson Objectives

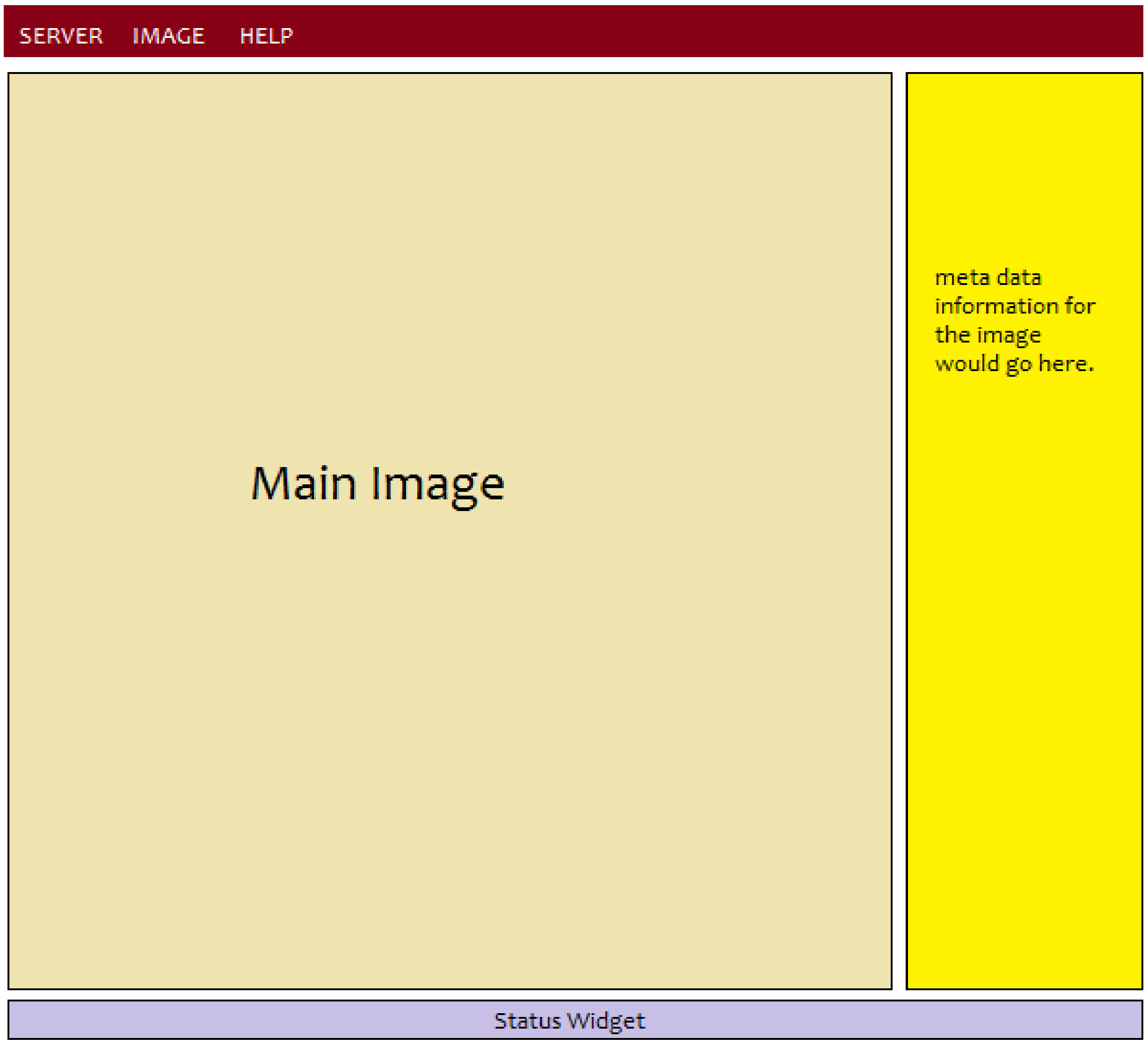
In this lesson you will:

- design a Java Swing client to connect to a remote server.

Writing a Swing Application Skeleton

Now let's get down to the business of writing a GUI with a menu bar to access commands, and a window where the user will browse the images stored on the server. For the layout, we'll take advantage of the GroupLayout class. This is the *go-to* class when designing Swing GUIs. After years of fumbling around with AWT layouts and third-party layout libraries, I was amazed at the versatility of GroupLayout; you'll be amazed too! The basic premise of this layout manager is that GUI layouts are fundamentally composed of symmetric rectangular regions; by treating each axis independently (horizontal and vertical), the code is both clear and concise, that is, once you get used to reading the code fragments.

Based on the requirements we set for this lab, we'll write a GUI that allows users to browse through the images in a repository. Let's assume that we need standard navigation ability where the user can advance to the next image, return to the previous image, go to the very first image, or to the very last image. You will display each image in the largest section of the GUI, and you'll reserve space to include metadata about the image. All of these GUI elements apply to the entire application, not just this particular lab. You'll include a status widget at the bottom of the window as well. Ultimately, the GUI will look like this:



You'll build this GUI incrementally over the next few labs.

In the **/src** folder, create the **client.gui** package.

In the **/src/client.gui** package, create an **ImageRepositoryClient** class that extends **javax.swing.JFrame** (which is needed for any Swing window-based GUI) as shown:

CODE TO TYPE: /src/client.gui/ImageRepositoryClient.java

```
package client.gui;

import java.awt.*;
import javax.swing.*~~JFrame~~;
import javax.swing.GroupLayout.Alignment;

/** Primary GUI window for the client application. */
public class ImageRepositoryClient extends JFrame {
  JScrollPane imgPanel;
  JTextArea imgMetaData;
  JTextField status;

  public ImageRepositoryClient() {
    super("Image Repository Client");
    initMenuBar();
    initLayout();
  }

  void initMenuBar() {
    JMenuBar mb = new JMenuBar();

    JMenu server = new JMenu ("Server");
    mb.add(server);

    JMenu image = new JMenu ("Image");
    mb.add(image);

    setJMenuBar(mb);
  }

  void initLayout() {
    setSize (600, 600);

    JPanel p = new JPanel();
    GroupLayout layout = new GroupLayout(p);
    p.setLayout(layout);
    layout.setAutoCreateGaps(true);
    layout.setAutoCreateContainerGaps(true);

    layout.setHorizontalGroup(layout.createParallelGroup(Alignment.CENTER).
        addGroup(layout.createSequentialGroup().
          addComponent(imagePanel()).
          addComponent(imageMetaData(), GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT
_SIZE, GroupLayout.PREFERRED_SIZE)).
        addComponent(statusBar()));

    layout.setVerticalGroup(layout.createSequentialGroup().
        addGroup(layout.createParallelGroup(Alignment.CENTER).
          addComponent(imagePanel()).
          addComponent(imageMetaData())).
        addComponent(statusBar(), GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE,
 GroupLayout.PREFERRED_SIZE));

    add(p);
  }

  JScrollPane imagePanel() {
    if (imgPanel == null) {
      imgPanel = new JScrollPane();
      imgPanel.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS
_NEEDED);
      imgPanel.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEE
DED);
      imgPanel.setPreferredSize(new Dimension(416, 520));
    }
    return imgPanel;
  }

  JTextArea imageMetaData() {
    if (imgMetaData == null) {
      imgMetaData = new JTextArea();
      imgMetaData.setEditable(false);
      imgMetaData.setPreferredSize(new Dimension(160, 520));
    }

    return imgMetaData;
  }

  JTextField statusBar() {
    if (status == null) {
      status = new JTextField(132);
      status.setEditable(false);
    }

    return status;
  }
}
```

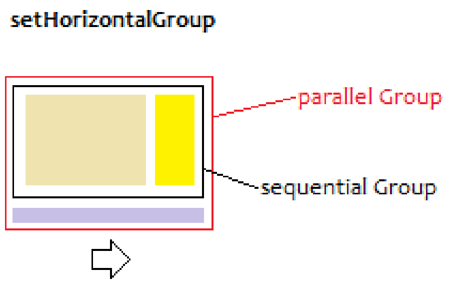
This class is responsible for constructing the GUI elements, including the menu bar and the frame's contents. We had to type a lot of code there, and we need to go over one tricky bit of logic. Pay particular attention to the widgets created here: **imgPanel**, which will present the images in the repository **imgMetaData**, which represents the textual metadata for the image being viewed, and **status**, which contains status information about the execution of the application.

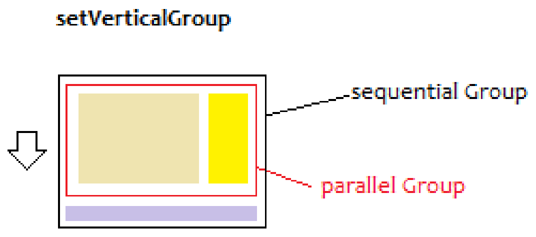
Let's talk about the methods in this class. It's standard practice to have **imagePanel()** and **imageMetaData()** methods that either create or return the widget created earlier, which simplifies ordering constraints that may be present in initialization code. By creating a class attribute to store the widgets being created, you can reference these objects later. Note how the specialized logic for each widget is encapsulated; for example, **imgMetaData** is constructed to be non-editable. The initMenuBar() method is pretty self-explanatory. Let's move on and take a look at initLayout():

OBSERVE:

```
void initLayout() {
    setSize (600, 600);

    JPanel p = new JPanel();
    GroupLayout layout = new GroupLayout(p);
    p.setLayout(layout);
    layout.setAutoCreateGaps(true);
    layout.setAutoCreateContainerGaps(true);

    layout.setHorizontalGroup(layout.createParallelGroup(Alignment.CENTER).
        addGroup(layout.createSequentialGroup().
          addComponent(imagePanel()).
          addComponent(imageMetaData(), GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT
_SIZE, GroupLayout.PREFERRED_SIZE)).
        addComponent(statusBar()));

    layout.setVerticalGroup(layout.createSequentialGroup().
        addGroup(layout.createParallelGroup(Alignment.CENTER).
          addComponent(imagePanel()).
          addComponent(imageMetaData())).
        addComponent(statusBar(), GroupLayout.PREFERRED_SIZE, GroupLayout.DEFAULT_SIZE,
 GroupLayout.PREFERRED_SIZE));

    add(p);
  }
```

The initLayout() method follows common Swing practice by constructing a JPanel object that is added to the enclosing ImageRepositoryClient class. Every JPanel object needs a LayoutManager; in this case you'll use GroupLayout. The key method invocations are setHorizontalGroup and setVerticalGroup. GroupLayout divides the layout by considering these two axes independently. This enables you to write sophisticated layouts that automatically stretch and shrink as the window is resized; if you've ever written a Java GUI using the default Abstract Windowing Toolkit (AWT), you'll recognize that this is a major upgrade to AWT's earlier layout managers.





Together, setHorizontalGroup and setVerticalGroup constrain the layout of widgets in the JPanel. As you look at the horizontal grouping of the widgets above, note that there is a sequential group = {**imgPanel** ; **imgMetaData** } from left to right. This group (from left to right) parallels **status** since the group is "on top of" **status**. In describing this horizontal layout, we have started from the inside and worked our way out. In the layout invocation above, you can see that we start with the outermost parallel group and work our way in. The indentation is critical to understanding. Each group starts a new indentation level, and all components in the same group have the same indentation. Note that a group itself can be considered just another component. The only further constraint is that **imgMetaData** must maintain a fixed width based on its preferred size, as determined by the three extra parameters.

In the vertical grouping of the widgets above, there is a parallel inner group = {**imgPanel** || **imgMetaData** } from top to bottom. This group is followed by **status** at the bottom. The final constraint is that the height of **status** must remain fixed. These visual cues allow you to understand the invocation to setVerticalGroup. If you view the proposed image from top to bottom, you'll see that there are parallel groupings; **imgPanel** and **imgMetaData** are side by side, and both are on top of **status**.

Because you don't know the size of the images to be stored in the repository (and you don't know how the user will choose to resize the window), we instantiate the JScrollPane object in **imagePanel**(). It will use scrollbars to display whatever image is placed in it, automatically.

GroupLayout allows fine-grained control for resizing. In the invocation to setHorizontalGroup, when adding the **imgMetaData** widget, you add three optional parameters. Specifically, in the final application, you want to ensure that when you resize the application frame, the size of the metadata panel on the right size remains a fixed horizontal width. The three parameters reflect the minimum allowed size, the preferred size, and the maximum size. Here, minimum=maximum guarantees a fixed width.

Create the **ClientLauncher** class in the **/src/client** package to launch the GUI application:

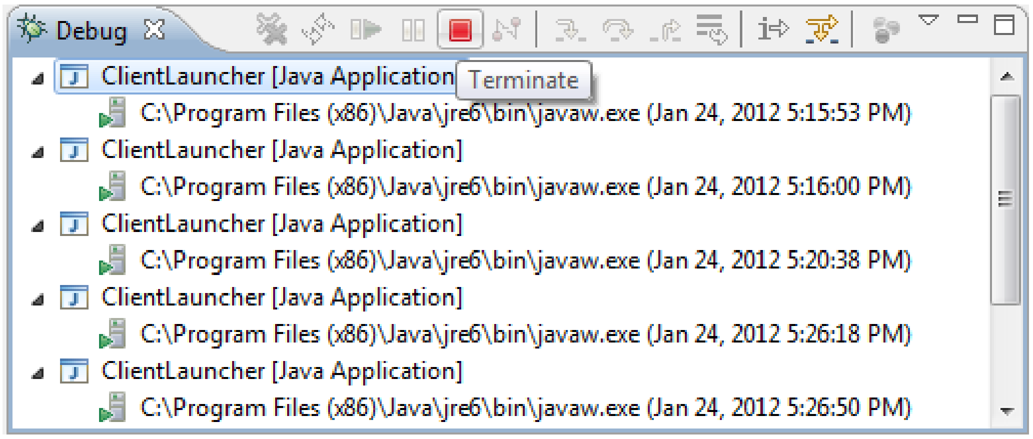
CODE TO TYPE: /src/client/ClientLauncher.java

```
package client;

import client.gui.*;

public class ClientLauncher {
  public static void main(String[] args) {
    ImageRepositoryClient irc = new ImageRepositoryClient();
    irc.setVisible(true);
  }
}
```

Our launcher will become more complex, but this is a good start. The launcher constructs an instance of the ImageRepositoryClient and makes it visible. This is how Swing applications are run.

When you run **ClientLauncher**, the widgets appear as expected; when you resize the frame, the image panel on the left grows while the metadata panel on the right remains fixed in width, and the status field maintains a fixed height. Close the Frame by clicking on the red X in the upper right corner. What happened to the nice splash screen? That's right, you also need to update the run configuration for **ClientLauncher** (as you did in a prior lab) by adding **-splash:images/repositorySplash.png** to the VM arguments used when executing that class. Now when you launch the **ClientLauncher**, the splash screen flashes momentarily before the main application launches. The splash screen may be visible for only a fraction of a second. As your application becomes more complex, the time to launch will increase and the splash screen will remain visible longer.

If you've been running the client examples above, and you haven't closed the client frames, do that now. Notice in Eclipse that you still have processes executing! You can see this in the console window, which still has a red box icon, which allows you to terminate the recently launched application. Switch to the Debug perspective (**Window | Open Perspective | Debug**); you'll see an entry in the Debug tab for every launched client. Use the Eclipse icon to terminate each of these applications, one by one. Select each **Client Launcher** instance and click on the red **Terminate** icon until all executables are terminated:



ClientLauncher [Java Application]
C:\Program Files (x86)\Java\jre6\bin\javaw.exe (Jan 24, 2012 5:15:53 PM)
ClientLauncher [Java Application]
C:\Program Files (x86)\Java\jre6\bin\javaw.exe (Jan 24, 2012 5:16:00 PM)
ClientLauncher [Java Application]
C:\Program Files (x86)\Java\jre6\bin\javaw.exe (Jan 24, 2012 5:20:38 PM)
ClientLauncher [Java Application]
C:\Program Files (x86)\Java\jre6\bin\javaw.exe (Jan 24, 2012 5:26:18 PM)
ClientLauncher [Java Application]
C:\Program Files (x86)\Java\jre6\bin\javaw.exe (Jan 24, 2012 5:26:50 PM)

How did this happen? The answer lies within ClientLauncher. The Swing GUI takes control once you make your first JFrame window visible. Until further action is taken, however, it will not relinquish control. Fix that now by giving the Swing GUI the ability to (a) ask the user to confirm the close window request, and (b) ask the user whether they want to remember this as default behavior in the future.

Switch back to the Java perspective (**Window | Open Perspective | Java**) and modify the ClientLauncher class as shown:

CODE TO TYPE: /src/client/ClientLauncher.java

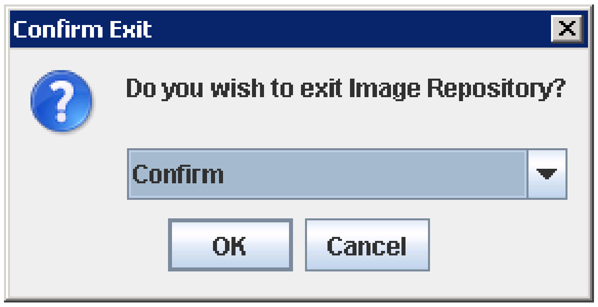
```
package client;

import java.awt.event.*;
import javax.swing.*;
import client.gui.*;

public class ClientLauncher {

  public static void main(String[] args) {
    final ImageRepositoryClient irc = new ImageRepositoryClient();
    irc.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

    final ImageIcon icon = new ImageIcon("images/help_32.png");
    irc.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        String[] choices = { "Confirm", "Confirm and don't ask me again" };
        String s = (String) JOptionPane.showInputDialog (irc,
                      "Do you wish to exit Image Repository?\n ",
                      "Confirm Exit", JOptionPane.PLAIN_MESSAGE,
                      icon, choices, choices[0]);
        if (s == null) {
          return;
        }
        irc.dispose();
      }
    });

    irc.setVisible(true);
  }
}
```

This code depends on an icon loaded from an external file. We recommend you store all of the images for your application in a central location like the **/images** folder; in this case, download and save the icon file there. I've used a free 32x32 icon, but you can use any 32x32 image:

The second argument to the showInputDialog() invocation ends in /n, which creates extra space to make the dialog box appear less cluttered. If you don't include the trailing space, the extra line is trimmed away. Go figure!

Right now you're interested in windowClosing actions; note how this code uses an anonymous class to extend WindowAdapter to override just the one method needed. The irc variable is marked final so the anonymous class can access this object. When you override the windowClosing method, you can deny the user's request by returning. To confirm the user request, dispose of the ImageRepositoryClient frame manually by calling dispose() on it. Make sure to tell Swing that the irc frame is not to be closed automatically; you do that by invoking the setDefaultCloseOperation with the JFrame.DO_NOTHING_ON_CLOSE argument.

Now when the user closes the ImageRepositoryClient window, the windowClosing method is executed, using a standard JOptionPane method to display a dialog that reqires the user to act.



Do you wish to exit Image Repository?

If the user clicks cancel (or closes the dialog), showInputDialog returns null and the windowClosing method returns without disposing of the irc frame, otherwise the irc frame is disposed. In Swing, once the last visible window is disposed, the Java VM can exit, so there is no need to invoke System.exit() here. There are no lingering processes when you close the window.

We've made some nice progress. Now we'll give our application the ability to store user preferences persistently, and give the user a chance to quit the ImageRepositoryClient application without requiring any confirmation.

Persistent User Preferences

You can store application-specific information in the user's home directory, which is the most efficient way to ensure that the file is stored in a user-accessible location, regardless of platform. Using the Java API, you can determine the user's home directory through the System property **user.home**. It is common to create file names that begin with a period (.), which makes these files "hidden." The Preferences helper class provides the necessary functionality for your client. It isn't clear which package should hold this class, because we could use preferences on either the client or the server side of the final application. While the class is dealing with client-based preferences now, we might want to use it for server-based preferences later. We'll create a new package named "util," and create the Preferences class there, so it's not tied to client or server, but placed in its own neat little utility package.

In the **/src** folder, create a **util** package, and in that package, create a **Preferences** class. As you type in this class, pay special attention to the methods you're writing, because they define the minimal behavior required for this class:

CODE TO TYPE: /src/util/Preferences.java

```
package util;

import java.io.*;
import java.util.*;

public class Preferences {
  static Properties props     = null;
  static String propFileName  = ".imageRepository.properties";
  static String homedir       = "user.home";

  public static String get(String name) {
    if (props == null) { load(); }
    return (String) props.get(name);
  }

  public static String set(String name, String value) {
    if (props == null) { load(); }
    String oldValue = (String) props.put(name, value);
    persist();
    return oldValue;
  }

  public static String remove(String name) {
    if (props == null) { load(); }
    String oldValue = (String) props.remove(name);
    persist();
    return oldValue;
  }

  static boolean load() {
    File file = new File (System.getProperty(homedir), propFileName);

    // silently accept first time if preferences file can't be found
    props = new Properties();
    if (!file.exists()) { return true; }

    try {
      props.loadFromXML(new FileInputStream(file));
      return true;
    } catch (Exception e) {
      System.err.println("Unable to load preferences from:" + file);
      return false;
    }
  }

  static void persist() {
    File file = new File (System.getProperty(homedir), propFileName);
    try {
      FileOutputStream fos = new FileOutputStream (file);
      props.storeToXML(fos, "Saved on " + new Date().toString());
    } catch (Exception e) {
      System.err.println("Unable to save preferences to:" + file);
    }
  }
}
```

This class manages a Properties object, **props**. The first three methods manipulate this object, as shown:

OBSERVE:

```
public class Preferences {
  static Properties props      = null;
  static String propFileName  = ".imageRepository.preferences";
  static String homedir       = "user.home";

  public static String get(String name) {
    if (props == null) { load(); }
    return (String) props.get(name);
  }

  public static String set(String name, String value) {
    if (props == null) { load(); }
    String oldValue = (String) props.put(name, value);
    persist();
    return oldValue;
  }

  public static String remove(String name) {
    if (props == null) { load(); }
    String oldValue = (String) props.remove(name);
    persist();
    return oldValue;
  }

  ...
```

`Preferences` persists the stored Properties object whenever the **set** or **remove** method is called. These methods each load() the Properties object from disk first, whenever **props** is null. This somewhat lazy form of evaluation ensures that **props** is properly configured when needed. After **set** and **remove** update **props**, the persist() method is called to store the information persistently to disk. Let's look at the load() code first:

OBSERVE:

```
  static boolean load() {
    File file = new File (System.getProperty(homedir), propFileName);

    // silently accept first time if preferences file can't be found
    props = new Properties();
    if (!file.exists()) { return true; }

    try {
      props.loadFromXML(new FileInputStream(file));
      return true;
    } catch (Exception e) {
      System.err.println("Unable to load preferences from:" + file);
      return false;
    }
  }
```

load() reads a set of preferences using the built-in loadFromXML() method of the Properties class. The location of the persistent file is computed using the global Java property **user.home**, which always tells you the current user's home directory on the file system. The method returns true when it loads the information successfully. When there is no stored file, load() handles the situation properly, because your program has never been run. The final piece of this class is the persist() method:

Storing Properties object to disk

```
  static void persist() {
    File file = new File (System.getProperty(homedir), propFileName);
    try {
      FileOutputStream fos = new FileOutputStream (file);
      props.storeToXML(fos, "Saved on " + new Date().toString());
    } catch (Exception e) {
      System.err.println("Unable to save preferences to:" + file);
    }
  }
```

This method complements the load() method. Note how a FileOutputStream object is created for storing the XML representation of the Properties object.

Because many preferences are boolean values, we'll add two helper methods to the class, and take advantage of java.lang.Boolean:

CODE TO TYPE: src/util/Preferences.java

```
...
  public static String set(String name, boolean b) {
    return set(name, Boolean.toString(b));
  }

  public static boolean isTrue(String name) {
    if (props == null) { load(); }
      return Boolean.parseBoolean((String)props.get(name));
  }
...
```

Now let's integrate this logic with ClientLauncher. The updated listing below shows you just what needs to be changed. As you can see, this code checks whether the preference already exists, and ensures that the ConfirmOnExit preference is set if the user chooses that option:

CODE TO TYPE: /src/client/ClientLauncher.java

```
package client;

import java.awt.event.*;
import javax.swing.*;
import client.gui.*;
import util.*;

public class ClientLauncher {
  static final String preference_confirmOnExit = "ConfirmOnExit";

  public static void main(String[] args) {
    final ImageRepositoryClient irc = new ImageRepositoryClient();
    irc.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

    final ImageIcon icon = new ImageIcon("images/help_32.png");
    irc.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        if (!Preferences.isTrue(preference_confirmOnExit)) {
          String[] choices = { "Confirm", "Confirm and don't ask me again" };
          String s = (String) JOptionPane.showInputDialog (irc,
                        "Do you wish to exit Image Repository?\n ",
                        "Confirm Exit", JOptionPane.PLAIN_MESSAGE,
                        icon, choices, choices[0]);
          if (s == null) {
            return;
          } else if (s.equals (choices[1])) {
          // remember this in the future.
            Preferences.set(preference_confirmOnExit, true);
          }
        }
        irc.dispose();
      }
    });
    irc.setVisible(true);
  }
}
```

Run **ClientLauncher** and close it; when closing the main window, choose the **Confirm and don't ask me again** option and press the **OK** button. After execution, check your home directory and look over the XML file named **.imageRepository.preferences** (click **File** in the upper right and navigate to your "V:" drive; right-click the file and select **Open**; if prompted, browse to and select **Notepad** as the viewer):

OBSERVE:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>Saved on Sun Feb 10 15:09:00 EST 2013</comment>
<entry key="ConfirmOnExit">true</entry>
</properties>
```

If you execute the application again and close the frame, you will no longer be prompted for a confirmation and the application will exit cleanly. To reinstate the requirement to request confirmation, delete the **.imageRepository.preferences** file in your home directory.

Note The JDK has a powerful java.util.prefs.Preferences class that allows "applications to store and retrieve user and system preference and configuration data. This data is stored persistently in an implementation-dependent backing store. Typical implementations include flat files, OS-specific registries, directory servers and SQL databases. The user of this class needn't be concerned with details of the backing store." On a Linux machine, user preferences would be stored in **$HOME/.java/.userPrefs/prefs.xml**; on a Windows desktop, information would be stored in the Windows registry. We have shown how to write your own preferences functionality.

Testing

Everything looks great, but before we consider this lab a success, let's write some tests. The one way to feel entirely confident about your code is to write test cases that demonstrate proper behavior. The Preferences class makes testing a challenge because it works with persistent information. In addition, if you aren't careful, testing the Preferences class will overwrite the image repository preferences file stored in the user's directory! What is a tester to do? Well, you can take advantage of the way the Eclipse workspace is set up, where test cases are stored in the **/test** folder. In particular, you can "rename" the preferences file so that all test cases are processed independently of the normal running code.

Create a **util** package in the **/test** source folder.

In the **/test** folder **util** package, create a **TestPreferences** JUnit test case as shown:

CODE TO TYPE: /test/util/TestPreferences.java

```
package util;

import java.io.*;
import junit.framework.TestCase;

public class TestPreferences extends TestCase {
  File propFile;
  static String oldPropName;
  static String testPropName = ".testProps";
  final static String librarySize = "LibrarySize";

  // ensure preferences file will be in test location
  protected void setUp () {
    oldPropName = Preferences.propFileName;
    Preferences.propFileName = testPropName;
    Preferences.props = null;
    propFile = new File (System.getProperty(Preferences.homedir), testPropName);
    if (propFile.exists()) {
      assertTrue (propFile.delete());
    }
  }

  // delete test location file and restore original name
  protected void tearDown() {
    if (propFile.exists()) {
      assertTrue (propFile.delete());
    }
    Preferences.propFileName = oldPropName;
  }

  public void testSinglePreference() {
    assertFalse (propFile.exists());
    assertNull (Preferences.get(librarySize));

    Preferences.set(librarySize, "1000");
    assertTrue (propFile.exists());

    assertEquals ("1000", Preferences.get(librarySize));
    assertEquals ("1000", Preferences.set(librarySize, "1500"));
    assertEquals ("1500", Preferences.remove(librarySize));
    assertNull (Preferences.get(librarySize));
  }
}
```

setUp() and tearDown() are used to rename the **Preferences.propFileName** value, so these test case methods can execute independently of the production code:

OBSERVE:

```
...
  // ensure preferences file will be in test location
  protected void setUp () {
    oldPropName = Preferences.propFileName;
    Preferences.propFileName = testPropName;
    Preferences.props = null;
    propFile = new File (System.getProperty(Preferences.homedir), testPropName);
    if (propFile.exists()) {
      assertTrue (propFile.delete());
    }
  }

  // delete test location file and restore original name
  protected void tearDown() {
    if (propFile.exists()) {
      assertTrue (propFile.delete());
    }
    Preferences.propFileName = oldPropName;
  }
...
```

Before each test case method, setUp() ensures that there is no preferences file, **propFile**, on disk. tearDown() will also delete the file on disk at the completion of each test case method, so you can be sure that each test case method will start with a pristine file system. Now let's take a closer look at the testSinglePreference test case method:

OBSERVE:

```
  public void testSinglePreference() {
    assertFalse (propFile.exists());
    assertNull (Preferences.get(librarySize));

    Preferences.set(librarySize, "1000");
    assertTrue (propFile.exists());

    assertEquals ("1000", Preferences.get(librarySize));
    assertEquals ("1000", Preferences.set(librarySize, "1500"));
    assertEquals ("1500", Preferences.remove(librarySize));
    assertNull (Preferences.get(librarySize));
  }
```

This test case method covers two cases. First, **when there is not yet a preferences file (that is, the propFile instantiated in setUp does not exist on disk)**. In this case, returning the librarySize preference must return null. Second, **once a preference is set (in this case, librarySize), the properties file is created on disk; thereafter, this preference can be updated and Preferences.get() will retrieve its current value**. This test case also confirms that the value of an unknown (or removed) preference is returned as null.

In addition, this test case confirms that you can add a single preference to an empty preferences file, verify that it exists, change its value, and then verify that it no longer exists once it has been deleted. Run this test case to verify that your code works. What's this? The test case fails? The test case fails in the tearDown() method when you're trying to delete the preferences file on disk. You can check to make sure that this file exists on your computer, so why can't you delete it? This is one of the most common mistakes Java programmers make when using OutputStream objects. There seems to be no easy to way to figure out what's going wrong, but if you don't close an output stream then it will remain open until the Java VM exits. Normally this isn't a problem, but it becomes one when you try to delete the underlying file from disk. Where did you open an output stream? Go to the persist() method in Preferences and you will see the subtle defect as shown below. This method creates a FileOutputStream, but fails to close it! Add the single line of code to close the FileOutputStream, and the test case passes:

CODE TO TYPE: /src/util/Preferences.java

```
...
  static void persist() {
    File file = new File (System.getProperty(homedir), propFileName);
    try {
      FileOutputStream fos = new FileOutputStream (file);
      props.storeToXML(fos, "Saved on " + new Date().toString());
      fos.close();
    } catch (Exception e) {
      System.err.println("Unable to save preferences to:" + file);
    }
  }
...
```

This example demonstrates that you need to write and run test cases after you complete key functionality. When you do that, you can ensure that your code works right and validate that it continues to work later whenever changes happen.

Now run this test case through EclEmma code coverage; the test coverage for the Preferences class is around 60%. We'll need to improve this. There are some places in our code in Preferences that did not execute. You'l want to investigate those scenarios and write test cases for them:

- Loading up a sample (and valid) preferences file from disk.
- Exercising the special methods to handle boolean preferences.

Add two more test case methods to the end of the **TestPreferences** class:

CODE TO TYPE: /test/util/TestPreferences.java

```
...
  public void testLoadWorks() {
    if (!propFile.exists()) {
      assertNull (Preferences.get(librarySize));
    }
    Preferences.set("LibrarySize", "1000");

    // clear out from Preferences
    Preferences.props = null;

    assertEquals ("1000", Preferences.get("LibrarySize"));
  }

  public void testBooleanPreferences() {
    String booleanAtt = "SomeBooleanAtt";
    Preferences.set(booleanAtt, true);
    assertTrue (Preferences.isTrue(booleanAtt));
    assertEquals ("true", Preferences.remove(booleanAtt));
    assertFalse (Preferences.isTrue(booleanAtt));
  }
...
```

After adding these methods, you can see that the coverage still remains too low, at about 78%. The largest unexecuted logic occurs when there is a problem loading up the preferences file. Add this test case to the end of **TestPreferences**, which constructs a "corrupted" XML file that cannot be loaded properly:

CODE TO TYPE: /test/util/TestPreferences.java

```
...
  public void testGarbagePropsFile() {
    try {
      PrintWriter pw = new PrintWriter (propFile);
      pw.println("GARBAGE");
      pw.close();
    } catch (FileNotFoundException fnfe) {
      fail ("Unable to create sample props file.");
    }

    assertNull (Preferences.get("Testing"));
  }
...
```

Run coverage using this test case and you'll see that once again, there is an unexpected error within the tearDown() method. What could have gone wrong this time? Given the problems you saw earlier with the FileOutputStream, you're robably not surprised that there is a corresponding problem when dealing with FileInputStream. Review the load() method and you'll see an invocation to loadFromXML, but when you review the documentation for this method it claims "The specified stream is closed after this method returns." However, as you have just found out, the stream is only closed if the method returns successfully, not when an exception is thrown! The next modification you make will ensure that the file is closed, even upon an exception. Now when you rerun all test cases within EclEmma you'll see a coverage of over 87%. This is a solid testing performance. (The "Unable to load preferences from:\\beam\~\.testProps" warning message in the console appears because of the testGarbagePropsFile test case, as it should):

CODE TO TYPE: /src/util/Preferences.java

```
...
  static boolean load() {
    File file = new File (System.getProperty(homedir), propFileName);

    // silently accept first time if preferences file can't be found
    props = new Properties();
    if (!file.exists()) { return true; }

    FileInputStream fis = null;
    try {
      fis = new FileInputStream(file);
      props.loadFromXML(~~new FileInputStream(file)~~fis);
      return true;
    } catch (Exception e) {
      System.err.println("Unable to load preferences from:" + file);
      try { fis.close(); } catch (IOException ioe) { }
      return false;
    }
  }
...
```

You've missed some exceptional cases in the test case. The new line of code added to the exception handler within the load() method is shown in yellow because the IOException exception was never thrown, thus the empty exception handler was never executed. The other yellow-marked regions are similar. The only red-marked region is within the persist() method.

Now you've got the beginnings of a GUI in place. Good work. Work through your lesson assignments now and I'll see you again soon!

Server-Side Application Model

Lesson Objectives

In this lesson you will:

- use the application model for this ImageRepository application.
- store and load images from disk and transfer these images from client to server using the existing IPC layer.
- use a client to add an image to the repository.
- test the use of the repository with one or more images.

Server-Side Application Model

Okay, you've got your GUI skeleton in place. Now you're ready to add logic to the RepositoryServer. At this point, you'll want to review the requirements and ask yourself key questions about the information that the server needs to store. For example:

- Will images be stored by name? If so, does the name have to conform to a specific format?
- Will images have an "index number" reflecting their position? This seems like it could be hard to implement because deleting an individual image will cause successive images to be renumbered. Perhaps instead the image will have numbers that won't be reused when an image is deleted (causing gaps).
- How will images be stored on disk? Will each image be placed in its own file or will multiple images be stored together?
- How will metadata for each image be stored, such as when it was uploaded, format information, or which user uploaded it?

First you have to decide whether to have a single repository of all images maintained by a server or to allow a server to host multiple repositories. Whatever you decide, a client will only connect to a single repository at a time. With either choice there will be tradeoffs:

- If a server is restricted to hosting just a single repository, you will simplify the server code itself, but the environment will become more complicated, because you will need to launch new servers for each individual repository.
- If a server hosts multiple repositories, more clients are connected to a single server and the throughput for that server may be at risk.

In this lesson, you'll customize the server to use a specific directory containing images as a repository, or use a default image repository when executed. Even so, you need to figure out a way to prevent multiple servers from executing over the same image repository at the same time (more on this later).

Because disk space is not an issue for most applications, you'll store each image in its own file. This sets up another tradeoff: because you're making it simpler to access individual files, the server will use the file system to store potentially thousands of files containing images. This approach is preferable to developing a technique to store all image data within one single file. In addition, it's easier to test a repository in which images are stored in their own files.

Lastly, you can't rely on the sorted image names (or the file names) to set their order in the repository. Instead, you'll construct an *index* that maintains the order of the images and all corresponding metadata. You might consider placing all metadata inside a database (but that's beyond the scope of this course). For now, you'll develop an internal API that could ultimately be reimplemented to use a database.

Repository Selection

In the **/src** folder, create a **server.model** package, where you'll create your classes. The design proposed here is not the only way to approach the problem, but it offers one solution. By creating a package for the application model, you maintain separation between these classes and the rest of the code. When working with complex applications, do not let code become too interwoven, otherwise you risk having code that "only works when everything works." You may have heard of the Model/View/Controller (MVC) paradigm, especially in the context of Graphical User Interfaces. The design introduced in these labs is similar to MVC.

In the **/src** folder **server.model** package, create a **Repository** class:

CODE TO TYPE: /src/server.model/Repository.java

```
package server.model;

import java.io.*;

public class Repository {

  final File storage;
  int count;

  public Repository(File storage) throws IOException {
    if (storage.exists() && storage.isDirectory()) {
      this.storage = storage;
    } else {
      throw new IOException ("Storage for repository must be an existing directo
ry.");
    }
  }

  public void add(byte[] image, String name) {
    System.out.println("Adding " + name + " [" + image.length + " bytes]");
    count++;
  }

  public int size() {
    return count;
  }
}
```

This first implementation of the Repository class consists mostly of scaffolding. This is a good way to initiate your designs when you'll be implementing code with increasing complexity. By marking the **storage** attribute as final, you prevent any changes to the storage reference during execution.

The Repository object will not be constructed if the storage directory doesn't exist. You must construct a Repository object with an actual directory, otherwise the constructor will throw a FileNotFoundException. Because of this invariant, the final version of the class will be easier to write.

Our code contains two initial methods: one allows you to add an image whose bytes are stored in a byte[] array and another allows you to query the number of images stored in the repository. For now, this (mostly scaffolding) code just maintains a running count.

The Repository starts with a basic interface. The first implementation represents an image with a byte[] array. We won't bother to design a complex class to represent an image until we're sure that we need it.

Create a folder in the **DistributedApp** project named **Repository** which represents the default repository.

Then, modify the create() method of **ServerLauncher** to construct a **Repository** object using a default file location as shown:

CODE TO TYPE: /src/server/ServerLauncher.java

```
package server;

import java.io.*;
import server.ipc.*;
import server.model.*;

public class ServerLauncher {
  public static final String defaultLocation = "Repository";

  public static RepositoryServer create() throws Exception {
     return create(new File (defaultLocation));
  }

  public static RepositoryServer create(File dir) throws Exception {
    Repository repository = new Repository(dir);
    RepositoryServer server = new RepositoryServer(repository, new ProtocolHandl
er(repository));
    server.bind();
    return server;
  }

  public static void main(String[] args) throws Exception {
    RepositoryServer server = create();

    System.out.println("Server awaiting client connections");
    server.process();
    System.out.println("Server shutting down.");
  }
}
```

These modifications require the RepositoryServer to become aware of the Repository object introduced in this lab; this makes sense because of the central role that the RepositoryServer plays. The secondary change is that the ProtocolHandler object is constructed with a Repository object; this is done to allow the handler to access the repository as needed during processing. Update **RepositoryServer** as shown:

CODE TO TYPE: /src/server.ipc/RepositoryServer.java

```
package server.ipc;

import java.io.*;
import java.net.*;
import server.model.*;

public class RepositoryServer {
  ServerSocket serverSocket = null;
  int state = 0;
  IProtocolHandler protocolHandler;
  Repository repository;

  public RepositoryServer(Repository rep, IProtocolHandler ph) {
    protocolHandler = ph;
    repository = rep;
  }

  public void bind() throws IOException {
    serverSocket = new ServerSocket(9172);
    state = 1;
  }

  public void process() throws IOException {
    while (state == 1) {
      Socket client = serverSocket.accept();

      new RepositoryThread(client, protocolHandler).start();
    }

    shutdown();
  }

  void shutdown() throws IOException {
    if (serverSocket != null) {
      serverSocket.close();
      serverSocket = null;
      state = 0;
    }
  }
}
```

Now you need to make a few changes to **ProtocolHandler** so that the **ServerLauncher** will compile. Modify your code as shown:

CODE TO TYPE: /src/server/ProtocolHandler.java

```
package server;

import java.io.*;
import server.ipc.*;
import server.model.*;

public class ProtocolHandler implements IProtocolHandler {
  final Repository repository;

  public ProtocolHandler (Repository r) {
    repository = r;
  }

  public boolean process(BufferedReader fromSocket, PrintWriter toSocket) {
    try {
      String request = fromSocket.readLine();
      if (request == null) {
        return false;
      }

      if (request.equals("SIZE")) {
        output(toSocket, "0");
      } else {
        // internal server error. Try to continue and keep processing
        outputError(toSocket, "Unable to process request: " + request);
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
      return false;
    }

    return true;
  }

  void output(PrintWriter toSocket, String value) {
    toSocket.println(0);
    toSocket.println(value);
  }

  void outputError(PrintWriter toSocket, String error) {
    toSocket.println(-1);
    toSocket.println(error);
  }
}
```

We'll draw inspiration from the Test-Driven Development (TDD) community; first we'll create a JUnit test case that validates the expected behavior. In writing the test case, you will complete the remaining code for this lab.

During testing, you shouldn't depend on using normal repositories; create a test repository to use only for test cases. This is useful because then you can create, add, or destroy repositories as needed during testing.

Create a folder in your **DistributedApp** project named **TestRepository** to use for all test cases. Modify **TestServer** as shown:

CODE TO TYPE: /test/server.ipc/TestServer.java

```
package server.ipc;

import java.io.*;
import server.ServerLauncher;
import server.ipc.RepositoryServer;
import client.*;
import junit.framework.TestCase;

  public class TestServer extends TestCase {

    public static final String testRepository = "TestRepository";

    ...

    public static RepositoryServer launchServer() throws Exception {
      final RepositoryServer server = ServerLauncher.create(new File (testReposi
tory));
      ...
```

The create() method in **ServerLauncher** allows for straightforward customization.

In the **/test** folder's **server.ipc** package, create the **TestAddBehavior** test case. This test case is a bit raw and you will have to modify it later. This type of incremental change is pretty common to a client/server system, that is, where you add a new message between the client and server. It requires changes to both sides of the client/server system, but if you do it right, you won't have to make any changes to the underlying IPC infrastructure.

CODE TO TYPE: /test/server.ipc/TestAddBehavior.java

```
package server.ipc;

import java.io.*;
import java.net.*;
import util.*;
import junit.framework.TestCase;

public class TestAddBehavior extends TestCase {
  RepositoryServer server;
  Socket client;

  protected void setUp() throws Exception {
    server = TestServer.launchServer();
    client = new Socket("localhost", 9172);
  }

  protected void tearDown() throws Exception {
    server.shutdown();
    client.close();
  }

  public void testAddBehavior() throws Exception {
    PrintWriter toServer = new PrintWriter (client.getOutputStream(), true);
    BufferedReader fromServer = new BufferedReader (new InputStreamReader(client
.getInputStream()));

    // Protocol for sending SIZE
    toServer.println("SIZE");
    expectSuccess("0", fromServer);

    // Protocol for sending an image
    toServer.println("ADD-BEGIN");
    toServer.println("sampleImage");
    File f = new File("images", "repositorySplash.png");
    toServer.println(ImageEncoding.encode(f));
    toServer.println("\nADD-DONE");
    expectSuccess(null, fromServer);

    // Expect repository with 1 image
    toServer.println("SIZE");
    expectSuccess("1", fromServer);
  }

  public static void expectSuccess (String expect, BufferedReader fromServer) th
rows IOException {
    int response = Integer.valueOf(fromServer.readLine());
    String value = fromServer.readLine();

    if (response == 0) {
      if (expect != null) {
        assertEquals (expect, value);
      } else {
        System.out.println("received:" + value);
      }
    } else {
      fail ("(response:" + response + ") received " + value + " not " + expect);
    }
  }
}
```

You'll recognize much of the logic below from earlier labs. This code will not compile immediately because of a missing class (**util.ImageEncoding**) that you'll write in just a few minutes. This test case class has three main parts that we'll investigate now:

OBSERVE:

```
public class TestAddBehavior extends TestCase {
  RepositoryServer server;
  Socket client;

  protected void setUp() throws Exception {
    server = TestServer.launchServer();
    client = new Socket("localhost", 9172);
  }

  protected void tearDown() throws Exception {
    server.shutdown();
    client.close();
  }

  ...
```

The **setUp** method is defined by JUnit to be the method that executes immediately before each individual test case method. In this case, there is only one test case method, **testAddBehavior**. Here, **setUp** launches a server and client to be used by the test case method. The complementary **tearDown** method executes immediately after each test case method. The implementation shown above properly terminates both the server and client.

Next, check out the expectSuccess() helper method, which validates that the client receives a successful response:

OBSERVE:

```
  public static void expectSuccess (String expect, BufferedReader fromServer) th
rows IOException {
    int response = Integer.valueOf(fromServer.readLine());
    String value = fromServer.readLine();

    if (response == 0) {
      if (expect != null) {
        assertEquals (expect, value);
      } else {
        System.out.println("received:" + value);
      }
    } else {
      fail ("(response:" + response + ") received " + value + " not " + expect);
    }
  }
```

This helper method reads two string lines from the **BufferedReader fromServer** object. If the **response** value is 0 (zero), the server has declared a successful response. Remember, a failure is recorded using a value of "-1." The method does one more check to determine whether **expect** is non-null. If it is, then expectSuccess() confirms that the **value** matches the expected outcome, **expect**.

OBSERVE:

```
  public void testAddBehavior() throws Exception {
    PrintWriter toServer = new PrintWriter (client.getOutputStream(), true);
    BufferedReader fromServer = new BufferedReader (new InputStreamReader(client
.getInputStream()));

    // Protocol for sending SIZE
    toServer.println("SIZE");
    expectSuccess("0", fromServer);

    // Protocol for sending an image
    toServer.println("ADD-BEGIN");
    toServer.println("sampleImage");
    File f = new File("images", "repositorySplash.png");
    toServer.println(ImageEncoding.encode(f));
    toServer.println("\nADD-DONE");
    expectSuccess(null, fromServer);

    // Expect repository with 1 image
    toServer.println("SIZE");
    expectSuccess("1", fromServer);
  }
```

The test case method defines the **toServer** object to use for communicating requests to the server, and the **fromServer** object to use for receiving the response strings from the server. In our test case, first we send a **SIZE** request to the server; the expected successful response is "**0**," reflecting the number of images in the repository. Then we define the **ADD-BEGIN ... ADD-DONE** protocol and send an image using it. Once completed, the expected response is a success, although there is no specific value of interest to the test case. Finally, we send another **SIZE** request to the server, and this time the successful response must be "**1**."

The TestAddBehavior() test case verifies that when you add an image to an empty repository, the repository will contain 1 image. This observation and testing is carried out exclusively using the underlying network communication that you have already designed. Always test your code this way; first write (and test!) underlying utility classes and then build upon and use these classes in each subsequent test case. In doing so, you develop and test code incrementally in the exact environment in which it will be run.

You already have the **SIZE** message implemented, now let's review the proposed ADD message. Since the ProtocolHandler class is in complete control, the handler must support this ADD message structure:

- Send a line to the server with the string **ADD-BEGIN**, indicating that an image is about to be sent.
- Send a line with the name of the image ("sampleImage" in this case).
- Encode the raw bytes of the image ("repositorySplash.png" in this case) on subsequent lines, using the ImageEncoding class that you will define next.
- A final **ADD-DONE** line terminates the message.

This logic works only if the terminal string "ADD-DONE" does not appear in the image encoding. So how do you transmit an image of binary data from the client to the server over an ASCII protocol?

In the **/src** folder **util** package, create a new **ImageEncoding** class. There are two possible implementations for you to consider, either option works. Email attachments use MIME (Multipurpose Internet Mail Extensions) encoding to send binary data using an otherwise ASCII protocol. The problem is that this capability is not standard in the JDK. Here are the two solutions to this problem:

- Install a freely available open source library for the encoding.
- Access a "hidden" class in the JDK against the express wishes of the Java designers.

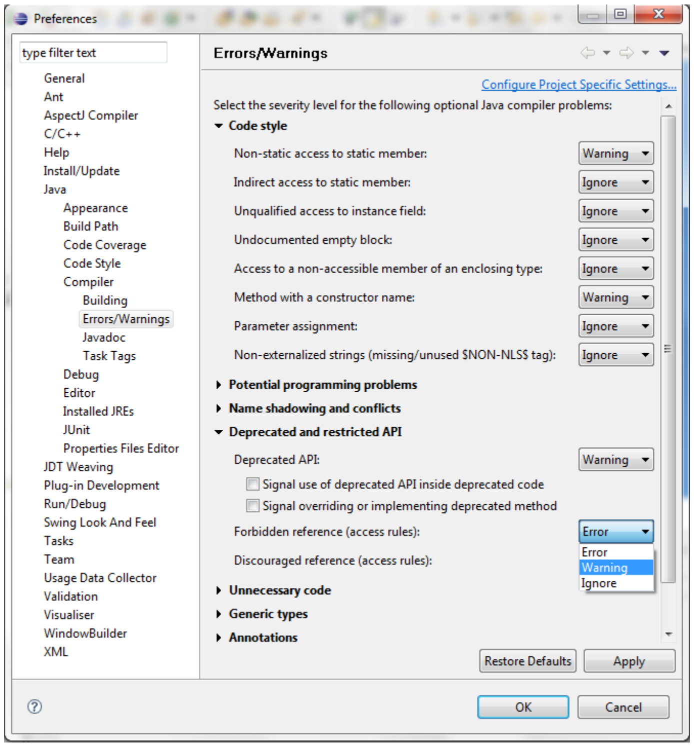
Option 1:Access a Hidden Class

The JDK comes with a sun.misc.BASE64Encoder class for encoding binary files. However, this class is in a **sun.*** package, which means you can't depend on its availability. It may not be present on a different operating system, but still, you can access this class (and find online documentation about it). Eclipse has a customizable compiler feature that might identify such attempts at using BASE64Encoder class as errors, however. Type in this code for **ImageEncoding** (if it compiles without any problems, great!):

CODE TO TYPE: /src/util/ImageEncoding.java

```
package util;

import java.io.*;
import sun.misc.BASE64Decoder;
import sun.misc.BASE64Encoder;

public class ImageEncoding {

  public static String encode (File f) throws IOException {
    if (f == null) {
      throw new IOException ("No image selected for encoding.");
    }
    FileInputStream fis = null;
    try {
      fis = new FileInputStream(f);
      byte[] bytes = new byte[(int) f.length()];
      fis.read(bytes);
      return new BASE64Encoder().encode(bytes);
    } finally {
      fis.close();
    }
  }

  public static byte[] decode (String str) throws IOException {
    return new BASE64Decoder().decodeBuffer(str);
  }
}
```

We close InputStream objects because there is no documentation or indication about whether Base64Encoder's encode() method closes the input stream once it's done. The **finally** block will close the output stream properly regardless of whether the write completed or threw an exception. The above code might show a compiler error with this warning: **Access restriction: The type BASE64Encoder is not accessible due to restriction on required library C:\Program Files (x86)\Java\jre6\lib\rt.jar**. If this happens, select **Window | Preferences** and expand the entries for **Java | Compiler | Errors/Warnings**. Expand the "Deprecated and restricted API" section, and change the "Forbidden reference (access rules)" to **Warning**, as shown:



Errors/Warnings

Select the severity level for the following optional Java compiler problems:

Code style

Non-static access to static member:	Warning
Indirect access to static member:	Ignore
Unqualified access to instance field:	Ignore
Undocumented empty block:	Ignore
Access to a non-accessible member of an enclosing type:	Ignore
Method with a constructor name:	Warning
Parameter assignment:	Ignore
Non-externalized strings (missing/unused $NON-NLS$ tag):	Ignore

Potential programming problems

Name shadowing and conflicts

Deprecated and restricted API

Deprecated API:	Warning

Signal use of deprecated API inside deprecated code

Signal overriding or implementing deprecated method

Forbidden reference (access rules):	Error (Error / Warning / Ignore)
Discouraged reference (access rules):	

Unnecessary code

Generic types

Annotations

When you're alerted that this action will require a full build of the project, click "agree," and your code compiles cleanly.

The next section shows an alternative that uses a freely available open-source implementation. If you're already famiiar with it, feel free to skip to the TestAddBehavior test case.

Option 2: Install a Free Open-Source Class

There are numerous open-source implementations that you can use for MIME-encoding files. Here is one authored by Robert Harder that I've chosen because it is free and without restrictions. You can download a Zip file from his website and save it to the file system of your project (which should be V:\workspace\DistributedApp). When you refresh your **DistributedApp** project in Eclipse, this zip file will appear. Double-click on the file to open it, then double-click on the Base64.java file within the zip file. A notepad application should appear containing the Java code. Select all of the text in the notepad and copy the text to your clipboard.

Create a **net.iharder** package in the **/src** folder.

Create a `Base64` class in that package, and replace that class definition with the contents of the clipboard. The result will be a Base64 class in the net.iharder package. Now you can define the following **ImageEncoding** class in the **util** package:

CODE TO TYPE: /src/util/ImageEncoding2.java

```
package util;

import java.io.*;
import net.iharder.*;

public class ImageEncoding {

  public static String encode (File f) throws IOException {
    return Base64.encodeFromFile(f.getAbsolutePath());
  }

  public static byte[] decode (String str) throws IOException {
    return Base64.decode(str);
  }
}
```

The structure of both options is similar. In fact, they are drop-in replacements of each other. You can use either one.

TestAddBehavior Test Case

Now, make a few changes to the **ProtocolHandler** class. First, the handler needs to have a reference to the **Repository** object, that's the purpose of the added constructor. Without the repository, the handler will not be able to function properly. Second, we want to update the handler to process add image requests. The added code reads one line at a time, appending each string together until it encounters the termination line ("ADD-DONE"). Then the ImageEncoding class decodes this string into the original byte array, which is added to the repository. The logic for counting the lines of the encoded representation is here just to give you an idea of the size (number of lines) of the encoding:

CODE TO TYPE: /src/server/ProtocolHandler.java

```java
package server;

import java.io.*;
import server.model.*;
import server.ipc.*;
import util.*;

/** Implementation of protocol. */
public class ProtocolHandler implements IProtocolHandler {
  final Repository repository;

  public ProtocolHandler(Repository repository) {
    this.repository = repository;
  }

  public boolean process(BufferedReader fromSocket, PrintWriter toSocket) {
    try {
      String request = fromSocket.readLine();
      if (request == null) {
        return false;
      }

      if (request.equals("SIZE")) {
        output(toSocket, "~~0~~" + repository.size());
      } else if (request.equals("ADD-BEGIN")) {
        String name = fromSocket.readLine();
        StringBuilder full = new StringBuilder();
        int num = 0;
        while (true) {
          String line = fromSocket.readLine();
          if (line.equals ("ADD-DONE")) { break; }

          full.append(line);
          num++;
        }

        byte[] bytes = ImageEncoding.decode(full.toString());
        repository.add(bytes, name);
        output (toSocket, bytes.length + " bytes received in " + num + " lines."
);
      } else {
        // internal server error. Try to continue and keep processing
        outputError(toSocket, "Unable to process request: " + request);
      }
    } catch (IOException ioe) {
      ioe.printStackTrace();
      return false;
    } catch (RuntimeException re) {
      outputError(toSocket, re.getMessage());
    }

    return true;
  }

  void output(PrintWriter toSocket, String value) {
    toSocket.println(0);
    toSocket.println(value);
  }

  void outputError(PrintWriter toSocket, String error) {
    toSocket.println(-1);
    toSocket.println(error);
  }
}
```

Instead of returning "0" when receiving a SIZE request, the server now returns a string containing the actual number of images in the repository, using the **size()** method of the **repository**. The second change is more complex:

OBSERVE:

```
if (request.equals("ADD-BEGIN")) {
  String name = fromSocket.readLine();
  StringBuilder full = new StringBuilder();
  int num = 0;
  while (true) {
    String line = fromSocket.readLine();
    if (line.equals ("ADD-DONE")) { break; }

    full.append(line);
    num++;
  }

  byte[] bytes = ImageEncoding.decode(full.toString());
  repository.add(bytes, name);
  output (toSocket, bytes.length + " bytes received in " + num + " lines.");
```

First, the server **reads one line of input to represent the name of the desired image file**. Then it **contatenates all strings received from the client between the "ADD-BEGIN" and "ADD-DONE" lines** (excluding "ADD-BEGIN" and "ADD-DONE" themselves). Using the **ImageEncoding** logic to convert the encoded string into a **byte[]** array, the server adds the image to the repository and outputs a message (of success) to the client, recording the number of bytes in the image. This protocol is well-defined, efficient to implement, and enforced by both client and server.

Run the **TestAddBehavior** test case to validate that we've got correct behavior. Now, run the same test case in EclEmma code coverage; you'll see that **ProtocolHandler** (75% coverage) and **Repository** (88% coverage) are missing only a few lines of error logic. This is a good start!

You've added an additional RuntimeException handler within the process() method, which will protect the server. If the server runs into any problems while processing a client request, an appropriate error message is returned to the client. The power of Exceptions allows you to have logic, in one place, that handles a range of potential error situations.

Completing Repository Functionality

Now you'll complete the functionality of Repository for storing the image persistently in a file. You'll also create an index file to represent all images in the repository. Consider how you'll determine the name of the file on disk for each uploaded image. In the testing example, the name of the image from the client's point of view was "sampleImage." However, if multiple clients upload different files, you can't have the repository store them with the same filename—the newest image would always obliterate the older one. In a client/server system, you can expect that clients don't communicate with each other, so we can't readily enforce a global naming scheme. We have an way to work past this problem and it offers an interesting twist —the MD5 cryptographic hash function. While MD5 is no longer "cryptographically secure," it can be used here to compute a "fingerprint" for an image to create a statistically unique 16-byte hash value. If it turns out that two clients upload the exact same image (bit for bit), then both requests will be computed to the exact same hash value, which can be used to advise the second client that the image already exists in the repository.

In the **/src** folder **util** package, create a **Fingerprint** class as shown:

CODE TO TYPE: /src/util/Fingerprint.java

```
package util;

import java.security.*;

public class Fingerprint {
  public static String getFingerPrint(byte[] bytes) {
    String defaultFingerprint = "ffffffffffffffffffffffffffffffff";
    try {
      MessageDigest md5 = MessageDigest.getInstance("MD5");
      md5.update(bytes, 0, bytes.length);

      StringBuilder sb = new StringBuilder();
      for (byte b : md5.digest()) {
        String hex = Integer.toHexString(b);
        if (hex.length() == 1) {
          sb.append("0").append(hex);
        } else if (hex.length() == 2) {
          sb.append(hex);
        } else {
          // negative byte values appear as something like "ffffffc0";
          sb.append(hex.substring(6));
        }
      }

      return sb.toString();
    } catch (Exception e) {
      return defaultFingerprint;
    }
  }
}
```

This class contains the functionality to compute a string representing the MD5 fingerprint of a **byte[]** array. It relies on the default algorithm implementation already found in the JDK. The JDK documentation contains a list of the supported algorithms. As you can see, you need only provide the bytes to the md5 algorithm, which return the fingerprint as an array of bytes, which is converted into a hex string as shown below. As documented in the md5sum application, the string returned by **Fingerprint** is identical to the value returned by md5sum.

The Fingerprint class contains the logic that converts the byte[] array message digest created using Java's default implementation of the MD5 algorithm. Placing this implementation in its own class simplifies the rest of your code and makes it possible to switch fingerprint algorithms efficiently. You can compare the output of this class directly to the output of 'md5sum' to verify that the values are identical.

To store an image in a file, the repository computes its fingerprint and attempts to create a file on disk in which to store the bytes; if thie attempt fails, then the image is already present in the repository and an Exception should be thrown. Modify **Repository** as shown (we'll want this method to return the computed fingerprint of the added image later, so we'll make that change now too):

CODE TO TYPE: /src/server.model/Repository.java

```
package server.model;

import java.io.*;
import util.*;

public class Repository {

  final File storage;
  int count;

  public Repository(File storage) throws IOException {
    if (storage.exists() && storage.isDirectory()) {
      this.storage = storage;
    } else {
      throw new IOException ("Storage for repository must be an existing directo
ry.");
    }
  }

  public ~~void~~String add(byte[] image, String name) {
    String fp = Fingerprint.getFingerPrint(image);
    File f = new File (storage, fp);
    if (f.exists()) {
      throw new IllegalStateException("That image already exists in the reposito
ry");
    }

    FileOutputStream fos = null;
    boolean failed = false;
    try {
      fos = new FileOutputStream (f);
    } catch (FileNotFoundException e) {
      failed = true;
    }

    try {
      if (!failed) {
        fos.write(image);
      }
    } catch (IOException e) {
      failed = true;
    } finally {
      try {
        fos.close();
      } catch (IOException ioe) {
        ioe.printStackTrace();
        failed = true;
      }
    }

    if (failed) {
      throw new IllegalStateException("Unable to construct image file. Contact A
dministrator.");
    }
    System.out.println("Adding " + name + " [" + image.length + " bytes]");
    count++;
    return fp;
  }

  public int size() {
    return count;
  }
}
```

The add() method attempts to store the image to disk in a file named for that image's fingerprint. Once the output file is created using the FileOutputStream (fos) object, that output stream must be closed. Note how the **finally** clause in the exception handler will be invoked regardless of whether the fos.write(image) method throws an IOException. The only trick here is the use of a separate try/catch handler in the **finally** block to handle errors that may arise when fos.close() executes.

When you run the **TestAddBehavior** test case now, a file is created in the **TestRepository** folder in Eclipse. You won't actually be able to see this file until you refresh the folder (because the file was created without Eclipse being aware of it). To do that, right-click the **TestRepository** folder and select **Refresh**. You'll see a single file with a name that consists of a long hex string—if you used the splash image in the test case, the name is **c00bc1ed28fabdbcebc3e4735decc83e**, which is the MD5 fingerprint for the image file.

Note The Repository class is unable to recover properly if there are problems in storing persistently, which is why the class is designed to throw an unchecked IllegalStateException when an image is added to the Repository. The server must handle these error situations.

Everything looks pretty good, but when you run the test case again, it fails, declaring "That image already exists in the repository." If you delete the image file in the **/TestRepository** folder manually, you can rerun the test case and it will succeed. Let this be a lesson in repeatability. Test cases are important because you can execute them automatically, at a moment's notice, to reaffirm your (increasing) confidence in your code!

I like what I'm seeing so far! Get going on the homework for this lesson and when you're done, I'll meet you in the next lesson!

In the next lab you will fix the TestAddBehavior test case and complete the Repository class to properly maintain a persistent index of files in the repository.

Java Object Serialization

Lesson Objectives

In this lesson you will:

- use Java Object Serialization for persistent storage
- use the Java Collections Framework.

Java Object Serialization

From its inception, Java offered support for linearizing objects into bytes and then restoring the original objects later. Known as the Object Serialization model, this functionality can be used to transmit objects over a network, or store objects persistently to disk. In this lesson, you'll learn about the benefits and challenges of using Object Serialization, and the best times to implement it.

We need an index to maintain the order of the images in the repository, and the metadata associated with each image. You'll develop an **Index** class for this purpose and use Java's built-in ability to store the **Index** object persistently to disk.

The trick to devising effective data structures is to envision the various ways in which the information is retrieved and updated. Our index maintains the ordering of the images and, for now, the only metadata associated with each image is its name, fingerprint, and size in bytes. The index will allow for iterative access through the entire set, as well as retrieval by ordered position and by fingerprint. Since we want the metadata to be extensible, we will use a Properties object for all metadata information.

in the **/src** folder's **server.model** package, create an **Index** class as shown:

CODE TO TYPE: /src/server.model/Index.java

```
package server.model;

import java.util.*;

public class Index implements Iterable<String> {
  // Order of keys determines order in repository
  ArrayList<String> keys             = new ArrayList<String>();
  Hashtable<String,Properties> meta = new Hashtable<String,Properties>();

  public Properties getMetaData(String key) {
    Properties md = meta.get(key);
    if (md == null) { return new Properties(); }
    return md;
  }

  public Properties setMetaData(String key, Properties props) {
    Properties old = meta.get(key);
    meta.put(key, props);
    return old;
  }

  public boolean add(String key) {
    if (keys.contains(key)) {
      return false;
    }
    keys.add(key);
    return true;
  }

  public Iterator<String> iterator() {
    return keys.iterator();
  }

  public int size() {
    return keys.size();
  }
}
```

By creating this class, you expose methods for the conceived behavior of the index. This class implements the **Iterable** interface, which allows you to iterate over all of the keys in the index using an enhanced **for** loop.

Here are the essential parts of this class:

OBSERVE:

```
  public class Index implements Iterable<String> {
     ...

     public Iterator<String> iterator() {
      ...
     }
  }
```

When Index implements Iterable, it declares that it contains an iterator() method which constructs an Iterator over its aggregate elements. In addition, the type of element returned by the Iterator is declared to be String (using the Java generics capability). The enhanced **for** loop that takes advantage of this capability would look like this:

OBSERVE:

```
  for (String key : idx) {
    System.out.println(key);
  }
```

The above code would print out (in order) the keys for the images stored by the Index object, idx:

OBSERVE:

```
public class Index implements Iterable<String> {
  // Order of keys determines order in repository
  ArrayList<String> keys            = new ArrayList<String>();
  Hashtable<String,Properties> meta = new Hashtable<String,Properties>();

  public Properties getMetaData(String key) {
    Properties md = meta.get(key);
    if (md == null) { return new Properties(); }
    return md;
  }

  public Properties setMetaData(String key, Properties props) {
    Properties old = meta.get(key);
    meta.put(key, props);
    return old;
  }

  public boolean add(String key) {
    if (keys.contains(key)) {
      return false;
    }
    keys.add(key);
    return true;
  }

  public Iterator<String> iterator() {
    return keys.iterator();
  }

  public int size() {
    return keys.size();
  }
}
```

The **keys** attribute is an **ArrayList** because you want to preserve the order of images in the repository by key. The **meta** attribute is an associative **Hashtable** that allows random access by key to retrieve or set the metadata associated with each image. Although **setMetaData**'s primary reponsibility is to store a Properties object for the given key, with minimal programming effort, you can have it return the prior Properties object that had been associated with the given key. This is a common pattern with get/set methods that eliminates the need to call getMetaData separately in order to get the former value before updating it.

Repository still doesn't store metadata persistently to survive from one server execution to the next. Also, the Index should be reloaded from persistent storage whenever the Repository is constructed. We want to integrate Index with Repository.

Given the final Repository class from the prior lab, you need to store the Index to disk when it changes and load up the complete Index object whenever a Repository is constructed. It makes sense to store the Index object in a file within the directory that contains the repository image files. Let's proceed with this task in stages. Make these changes to the **Repository** class to store the Index object to disk (and load it from disk). The code won't compile until all missing methods are in place:

CODE TO TYPE: /src/server.model/Repository.java

```
package server.model;

import java.io.*;
import java.util.*;
import util.*;

public class Repository {
  final File storage;
  Index index;
  static final String indexFileName = "indexFile";
  final File indexFile;
  ~~int count;~~

  public Repository(File storage) throws IOException {
    if (storage.exists() && storage.isDirectory()) {
      this.storage = storage;
      indexFile = loadIndex();
    } else {
      throw new IOException (~~"Storage for repository must be an existing directory."~~Sto
rageNotDirectory);
    }
  }

  public String add(byte[] image, String name) {
    String fp = Fingerprint.getFingerPrint(image);
    File f = new File (storage, fp);
    if (f.exists()) {
      throw new IllegalStateException(~~"That image already exists in the repository"~~Alre
adyExistsImage);
    }

    FileOutputStream fos = null;
    boolean failed = false;
    try {
      fos = new FileOutputStream (f);
    } catch (FileNotFoundException fnfe) {
      failed = true;
    }

    try {
      if (!failed) {
        fos.write(image);
      }
    } catch (IOException ioe) {
      failed = true;
    } finally {
      try {
        fos.close();
      } catch (IOException ioe) {
        ioe.printStackTrace();
        failed = true;
      }
    }

    if (failed) {
      throw new IllegalStateException(~~"Unable to construct image file. Contact Administ
rator."~~UnableToWriteFile);
    }
    ~~System.out.println("Adding " + name + " [" + image.length + " bytes");~~
    ~~count++;~~
    Properties props = new Properties();
    props.put("name", name);
    props.put("totalBytes", image.length);
    props.put("fingerPrint", fp);
    index.add(fp);
    index.setMetaData(fp, props);
    storeIndex();
    return fp;
  }

  boolean storeIndex() {
    FileOutputStream fos;
    try {
      fos = new FileOutputStream(indexFile);
    } catch (FileNotFoundException fnfe) {
      System.err.println("Unable to store index file to:" + indexFile);
      return false;
    }

    ObjectOutputStream oos = null;
    try {
      oos = new ObjectOutputStream(fos);
      oos.writeObject(index);
    } catch (IOException ioe) {
      System.err.println("Errors encountered while storing index file to:" + indexFile)
;
      ioe.printStackTrace();
      return false;
    } finally {
      try {
        oos.close();
      } catch (IOException ioe) {
        System.err.println("Errors encountered while closing index file.");
      }
    }

    return true;
  }

  File loadIndex() {
    index = new Index();

    File idxFile = new File (storage, indexFileName);
    if (idxFile.exists()) {
      FileInputStream fis;
      try {
        fis = new FileInputStream(idxFile);
      } catch (FileNotFoundException fnfe) {
        return null;
      }

      ObjectInputStream ois = null;
      try {
        ois = new ObjectInputStream(fis);
        index = (Index) ois.readObject();
      } catch (IOException ioe) {
        System.err.println("Problems encountered in loading Index file (" + idxFile + "
).");
        ioe.printStackTrace();
      } catch (ClassNotFoundException cnfe) {
        System.err.println ("Index file (" + idxFile + ") is not a valid Index object."
);
      } finally {
        try {
          ois.close();
        } catch (IOException ioe) {
          ioe.printStackTrace();
        }
      }
    }

    return idxFile;
  }
  public int size() {
    return ~~count~~index.size();
  }

  public static final String AlreadyExistsImage = "That image already exists in the rep
ository.";
  public static final String UnableToWriteFile = "Unable to construct image file. Conta
ct Administrator.";
  public static final String StorageNotDirectory = "Storage for repository must be a di
rectory.";
  public static final String StorageDoesNotExistPrefix = "Storage for repository doesn'
t exist:";
}
```

We added a lot of new code here; let's discuss some of the more important items:

One significant change is the set of error strings that are defined because of the principle of "single point of control." An exception message string is defined in only one place, which makes testing your code more efficient. (You'll see the benefit of that firsthand when you complete the test cases in this lab.)

In addition, now the Repository class is responsible for storing persistently and loading the Index object as it is updated. The order of objects in the repository will be based on the order of keys in the index. Whenever the Repository changes, the storeIndex() method can be called to store information persistently to disk:

OBSERVE:

```
  boolean storeIndex() {
    FileOutputStream fos;
    try {
      fos = new FileOutputStream(indexFile);
    } catch (FileNotFoundException fnfe) {
      System.err.println("Unable to store index file to:" + indexFile);
      return false;
    }

    ObjectOutputStream oos = null;
    try {
      oos = new ObjectOutputStream(fos);
      oos.writeObject(index);
    } catch (IOException ioe) {
      System.err.println("Errors encountered while storing index file to:" + indexFile)
;
      ioe.printStackTrace();
      return false;
    } finally {
      try {
        oos.close();
      } catch (IOException ioe) {
        System.err.println("Errors encountered while closing index file.");
      }
    }

    return true;
  }
```

This method uses the java.io classes to interact with the file system. We create a **FileOutputStream** object to access the designated indexFile on disk. Now we can use the writeObject() method of **ObjectOutputStream** to write a Serializable object to a file. The code attempts to cover several exceptional circumstances. Also, the **finally** block ensures that the output stream is closed properly upon completion.

When the Repository is instantiated the first time, the Index object must be loaded from disk, as described by the loadIndex() method:

Add loadIndex() method to Repository class

```
  File loadIndex() {
    index = new Index();

    File idxFile = new File (storage, indexFileName);
    if (idxFile.exists()) {
      FileInputStream fis;
      try {
        fis = new FileInputStream(idxFile);
      } catch (FileNotFoundException fnfe) {
        return null;
      }

      ObjectInputStream ois = null;
      try {
        ois = new ObjectInputStream(fis);
        index = (Index) ois.readObject();
      } catch (IOException ioe) {
        System.err.println("Problems encountered in loading Index file (" + idxFile + "
).");
        ioe.printStackTrace();
      } catch (ClassNotFoundException cnfe) {
        System.err.println ("Index file (" + idxFile + ") is not a valid Index object."
);
      } finally {
        try {
          ois.close();
        } catch (IOException ioe) {
          ioe.printStackTrace();
        }
      }
    }

    return idxFile;
  }
```

The loadIndex() method mirrors the storeIndex() method. The only difference is that it first instantiates a new Index object in case there is no persistent indexFile on disk.

If you launch the **TestAddBehavior** test case now, you'll see the following output on the console.

OBSERVE:

```
Errors encountered while storing index file to:TestRepository\indexFile
java.io.NotSerializableException: server.model.Index
 at java.io.ObjectOutputStream.writeObject0(Unknown Source)
 at java.io.ObjectOutputStream.writeObject(Unknown Source)
 at server.model.Repository.storeIndex(Repository.java:126)
 at server.model.Repository.add(Repository.java:72)
 at server.ProtocolHandler.process(ProtocolHandler.java:42)
 at server.ipc.RepositoryThread.run(RepositoryThread.java:30)
Server Completed.
```

The exception trace tells you that you have not properly made the Index class serializable. Fortunately, you only need to make this one change to the **Index** class to fix that:

CODE TO TYPE: /src/server.model/Index.java

```
public class Index implements Iterable<String>, java.io.Serializable {
 ...
}
```

This is known as a "marker" interface because it has no methods and it is used to mark objects so that the Java VM can to store the object. Only those classes specifically tagged as implementing java.lang.Serializable can be stored to disk. If any object could be serialized, it would expose the internal private data of those objects to prying eyes, and pose a real security risk.

Before you run the test case again, note that there is a "partially written" **indexFile** file in the **/TestRepository** folder. If you don't see it listed, **Refresh** the **/TestRepository** folder and it will become visible. Open it in Eclipse's text editor and take a look at the string **java.io.NotSerializableException**. Delete that file. If you don't the **loadIndex** method you just wrote will fail when attempting to load the partial file. Also, delete the image file in the **/TestRepository** folder, otherwise the test case will fail because "That image already exists in the repository."

Now re-run the **TestAddBehavior** test case and refresh the **/TestRepository** folder; you'll see an **indexFile** file. Open it in Eclipse to see that this is a quasi-binary stored representation. While you may not be able to interpret all of the characters, you'll see some strings that are recognizeable English, revealing the internal state of the objects that are being stored to disk.

Re-run the **TestAddBehavior** test case, and you may be surprised to see it fail. We wrote the test case assuming that the repository was empty. The setUp() method for this test case will likely erase all files (and the newly-persisted index file). Modify **clearTestRepository()** method to the **TestServer** class to delete all files as found in the repository directory (and this includes the persistent index file, which is being placed in the same directory) as shown:

CODE TO TYPE: test/server.ipc/TestServer.java

```
public class TestServer extends TestCase {

  public static final String testRepository = "TestRepository";

  public static void clearTestRepository() {
    File dir = new File(testRepository);
    File[] existing = dir.listFiles();
    for (File f : existing) {
      if (!f.isDirectory()) {
        assertTrue (f.delete());
      }
    }
  }

  ...
}
```

Now, modify **TestAddBehavior** to provide some useful helper methods for our test cases:

CODE TO TYPE: /test/server.ipc/TestAddBehavior.java

```
package server.ipc;

import java.io.*;
import java.net.*;
import server.model.*;
import util.*;
import junit.framework.TestCase;

public class TestAddBehavior extends TestCase {
  RepositoryServer  server;
  Socket            client;
  PrintWriter       toServer;
  BufferedReader    fromServer;

  void startClient() throws Exception {
    client = new Socket ("localhost", 9172);
    toServer = new PrintWriter (client.getOutputStream(), true);
    fromServer = new BufferedReader (new InputStreamReader(client.getInputStream()));
  }

  void stopClient() throws Exception {
    client.close();
    client= null;
  }

  void stopServer() throws Exception {
    server.shutdown();
    server = null;
  }

  protected void setUp() throws Exception {
...
```

The above methods allow you to start and stop a client, as well as the server. You'll continue to use the TestServer.launchServer() method to start the server. The additional toServer and fromServer objects are used to communicate from these clients to the server. Modify the setUp() method of **TestAddBehavior** to take advantage of these methods:

CODE TO TEST: test/server.ipc/TestAddBehavior.java

```
...
  protected void setUp() throws Exception {
    TestServer.clearTestRepository();
    server = TestServer.launchServer();
    ~~client = new Socket("localhost", 9172);~~
    startClient();
  }

  protected void tearDown() throws Exception {
    ~~server.shutdown();~~
    ~~client.close();~~
    stopServer();
    stopClient();
  }
...
```

Delete these files before constructing a Repository instance, otherwise the repository will pre-load the index file and the deletion would have no effect. Because of these JUnit methods, you need to modify the testAddBehavior() test case method in **TestAddBehavior** to remove otherwise duplicate functionality that has been moved to setUp() and tearDown():

CODE TO TEST: /test/server.ipc/TestAddBehavior.java

```
  public void testAddBehavior() throws Exception {
    ~~PrintWriter toServer = new PrintWriter (client.getOutputStream(), true);~~
~~    BufferedReader fromServer = new BufferedReader (new InputStreamReader(client.getInp~~
~~utStream()));~~

    // Protocol for sending SIZE
    toServer.println("SIZE");
    expectSuccess("0", fromServer);

    // Protocol for sending an image
    toServer.println("ADD-BEGIN");
    toServer.println("sampleImage");
    File f = new File("images", "repositorySplash.png");
    toServer.println(ImageEncoding.encode(f));
    toServer.println("\nADD-DONE");
    expectSuccess(null, fromServer);

    // Expect repository with 1 image
    toServer.println("SIZE");
    expectSuccess("1", fromServer);
  }
```

You'll also need to test the more complex behavior that takes place when you stop and start multiple clients, or even servers.

We'll create some building block methods that will be useful in any test case method you will write. These methods issue requests to the server to compute the size of the repository and add a new image to the repository. Modify **TestAddBehavior** to support more comprehensive tests:

CODE TO TYPE: /test/server.ipc/TestAddBehavior.java

```
...
  void requestSIZE() throws IOException {
    toServer.println("SIZE");
  }

  void requestADD(String name, File f) throws IOException {
    toServer.println("ADD-BEGIN");
    toServer.println(name);
    toServer.println(ImageEncoding.encode(f));
    toServer.println("\nADD-DONE");
  }
...
```

Now add test case that verifies a sequence of activities between a client and server:

CODE TO TYPE: /test/server.ipc/TestAddBehavior.java

```
...
  public void testBasicAddBehavior() throws Exception {
    requestSIZE();
    TestAddBehavior.expectSuccess("0", fromServer);

    File f = new File ("images", "repositorySplash.png");
    requestADD("sampleImage", f);
    TestAddBehavior.expectSuccess(null, fromServer);

    requestSIZE();
    TestAddBehavior.expectSuccess("1", fromServer);

    stopClient();
    stopServer();

    server = TestServer.launchServer();
    startClient();

    requestSIZE();
    TestAddBehavior.expectSuccess("1", fromServer);

    requestADD("sampleImage", f);
    expectFailure(Repository.AlreadyExistsImage, fromServer);
  }
...
```

The test repository is cleared out and a server and client are executed. Then an image is added to the repository and the size of the repository is confirmed to be 1. Then both client and server are stopped and restarted. Now the repository contains a single image, attempts to add the same image again, and fails.

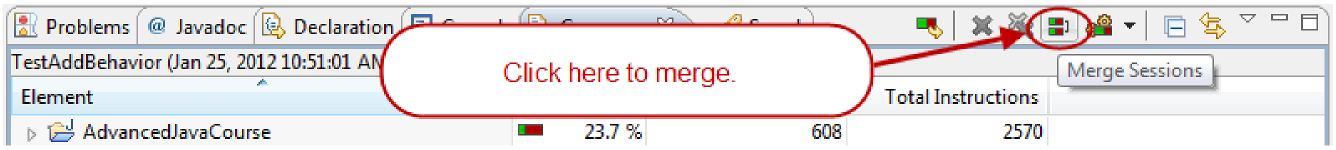
To complete the **TestAddBehavior** test case, add an **expectFailure** method as shown:

CODE TO TYPE: /test/server.ipc/TestAddBehavior.java

```
...
  public static void expectFailure (String expect, BufferedReader fromServer) throws IO
Exception {
    int response = Integer.valueOf(fromServer.readLine());
    String value = fromServer.readLine();

    if (response == -1) {
      if (expect != null) {
        assertEquals (expect, value);
      } else {
        System.out.println("received:" + value);
      }
    } else {
      fail ("(response:" + response + ") received " + value + " not " + expect);
    }
  }
...
```

This is nearly identical to the expectSuccess() method. The only difference is that it checks to make sure that the response from the server is -1.

This test case represents a significant investment to ensure basic functionality. Execute the **TestServer** and **TestAddBehavior** test cases to validate the correct behavior. Generate code coverage using EclEmma for each of these test cases. The EclEmma reporting panel shows you the individual reports for each individual test case, but you really want the combined results. Fortunately, the EclEmma panel contains the ability to merge multiple runs together to produce the result you need. On the panel, there is a "Merge Sessions" button with two stacked red/green bars:



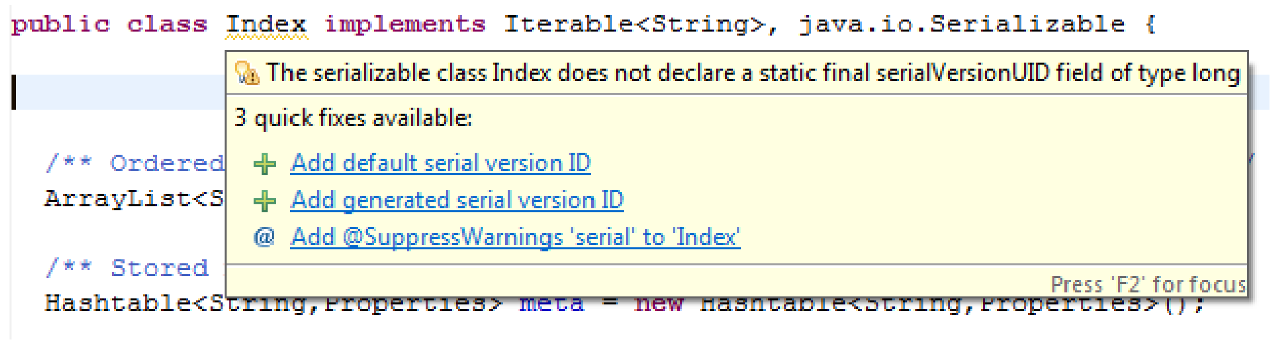
Click this button to bring up a dialog box where you can select the number of EclEmma sessions to merge to produce a single report of coverage.

You can see that the code coverage is rising (at least on the server side) and most of the non-executing code is found in exception handlers. You still have your work cut out for you, but you're on your way! You can continue to merge new runs, but these reports are not persistent. That is, if you exit Eclipse you start with a fresh slate when you come back.

There is one final point we need to cover regarding Java's Serialization model. If the Index class is modified after an instance has been written to disk, the default behavior is to assume that the persistently saved object cannot be de-serialized because of the risk of an inconsistent state. The complex solution to this problem would be to design specialized writeObject() and readObject() methods (you can explore these further in available tutorials on Java). A basic, and likely sufficient solution is to define a **static final long serialVersionUID** with the serializable class. Every instance written to disk embeds this serialVersionUID; the identifier is validated when the object is de-serialized. If you can guarantee that the only changes to a class are methods or **transient** attributes, the de-serialization should succeed.

Note You can mark an attribute **transient** to ensure that the Java Serializable mechanism ignores the attribute when storing and loading that object.

In Eclipse, you may have noticed that the Java editor of the Index class showed this warning:



```
public class Index implements Iterable<String>, java.io.Serializable {

    /** Ordered
    ArrayList<S

    /** Stored
    Hashtable<String,Properties> meta = new Hashtable<String,Properties>();
```

The serializable class Index does not declare a static final serialVersionUID field of type long

3 quick fixes available:

- Add default serial version ID
- Add generated serial version ID
- Add @SuppressWarnings 'serial' to 'Index'

Press 'F2' for focus

Select the second "Add generated serial version ID" option; a static attribute is added to **Index** (your value will likely be different):

OBSERVE:

```
  /** Serial version UID will enable loading from disk even when new methods are added.
 */
  private static final long serialVersionUID = -4153322746301327742L;
```

As long as you maintain the integrity of existing persisting attributes, future compilation of this class will not affect the way an Index object is loaded. If a structural change makes the new class version incompatible with an older stored version (and you are required to maintain backward compatibility), you will need to write specialized writeObject() and readObject() methods to support the loading of serializable objects with the older version.

That does it for our lesson on serialization. head over to your project adn work on that. WHen you're ready, move on to the next lesson.

Using XML to Specify a Protocol

Lesson Objectives

In this lesson you will:

- write an XML schema file to represent the protocol.

XML as Protocol Specification

Even though we have implemented only two messages in the system, we need to have an unambiguous way to specify the protocol so that both client and server know how to communicate. It's overly optimistic to assume that our code contains all relevant details. At the same time, it is impractical to write a protocol design document that must be updated each time the protocol changes. Ideally, we'll use a special language to specify the protocol that can become part of the code used by both client and server. The overall philosophy that expects us to use ASCII text to represent complex structures is simply outdated and ineffective. The industry standard for capturing complex structures textually is the eXtensible Markup Language (XML). Still, you have probably heard of it or even used it yourself at some time in your professional career. If you're interested in learning about XML in detail, contact us to find out more about our XML courses. You can also search for yourself and read up on the intricacies and power of XML. In this lab, you'll learn just enough to be able to support the client and server as they seek to communicate with each other.

The cornerstone for the protocol specification is an XML schema definition (XSD) file. This file defines a *schema* that can be used to validate that a given XML string is well-formed and valid. In general practice, we speak of an XML *document*, but for our purposes we'll use the term XML *string*, because those fragments are single messages. Using a schema definition, we can validate that a given string conforms to that schema before the client sends a command to the server. Similarly, before the server responds in kind, it must validate each XML string being sent to the client.

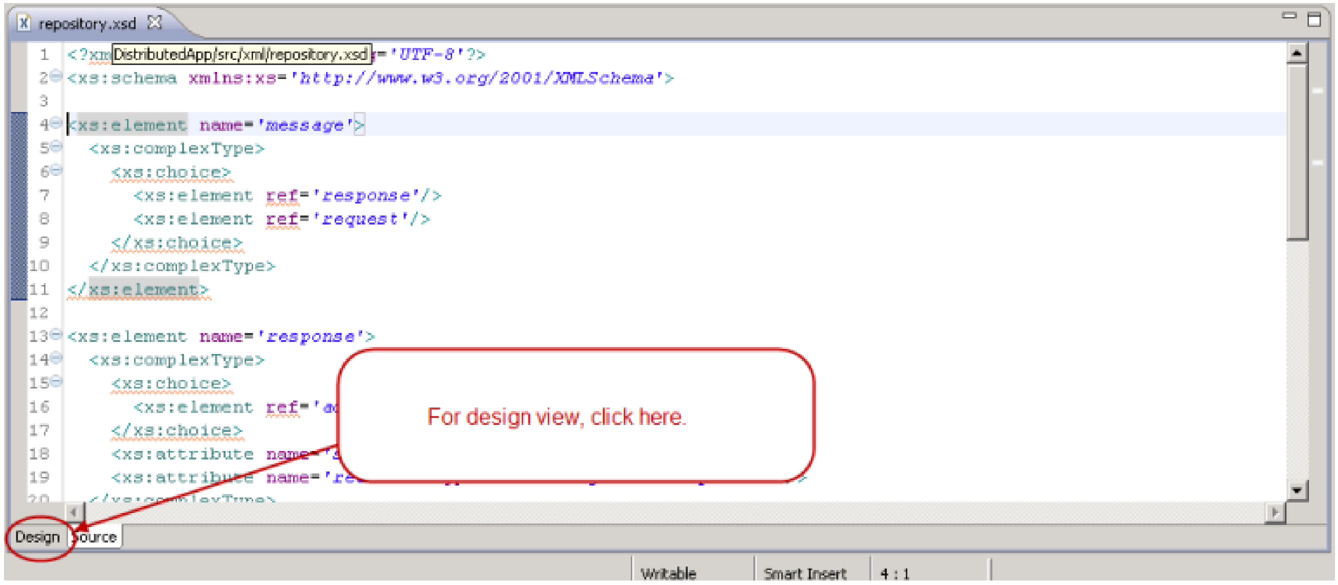
In the top folder of your **DistributedApp** project, create a file named **repository.xsd**. If Eclipse opens the editor in Design mode, click the **Source** tab at the bottom of the editor window, and then type the XML schema definition as shown below:

CODE TO TYPE: /repository.xsd

```
<?xml version='1.0' encoding='UTF-8'?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>

<xs:element name='message'>
  <xs:complexType>
    <xs:choice>
      <xs:element ref='response'/>
      <xs:element ref='request'/>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name='response'>
  <xs:complexType>
    <xs:choice>
      <xs:element ref='addResponse'/>
    </xs:choice>
    <xs:attribute name='success' type='xs:boolean' use='required'/>
    <xs:attribute name='reason'  type='xs:string'  use='optional'/>
  </xs:complexType>
</xs:element>

<xs:element name='request'>
  <xs:complexType>
    <xs:choice>
      <xs:element ref='addRequest'/>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name='addRequest'>
  <xs:complexType>
   <xs:sequence>
     <xs:element name='image'/>
   </xs:sequence>
   <xs:attribute name='name' type='xs:string' use='required'/>
  </xs:complexType>
</xs:element>

<xs:element name='addResponse'>
  <xs:complexType>
    <xs:attribute name='numBytes' type='xs:integer' use='required'/>
  </xs:complexType>
</xs:element>
</xs:schema>
```

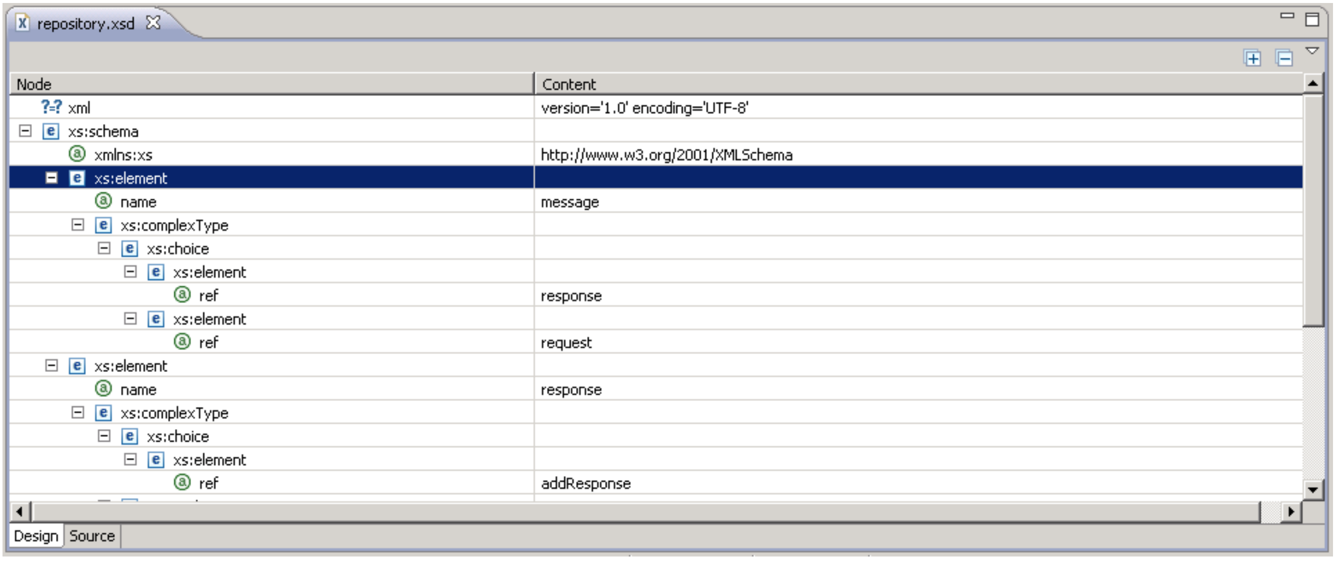
This XSD file supports the two message types envisioned for the system. We won't analyze it in detail, but we'll present enough information to understand how it works with this application. To learn more about XML schemas, see the W3Schools Schema tutorial and W3.org Schema Primer, or contact us for info about our XML course offerings.

Click the **Design** tab at the bottom of the Eclipse editor window (if you don't see the Design tab, close the file, right-click it in the Package Explorer, select **Open With**, select **Other...**, and select **XML Editor**):



```
<?xm DistributedApp/src/xml/repository.xsd = 'UTF-8'?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>

<xs:element name='message'>
  <xs:complexType>
    <xs:choice>
      <xs:element ref='response'/>
      <xs:element ref='request'/>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name='response'>
  <xs:complexType>
    <xs:choice>
      <xs:element ref='a
    </xs:choice>
    <xs:attribute name='s
    <xs:attribute name='re
  </xs:complexType>
```

Expand the elements and components:



Node	Content
?=? xml	version='1.0' encoding='UTF-8'
⊟ e xs:schema	
ⓐ xmlns:xs	http://www.w3.org/2001/XMLSchema
⊟ e xs:element	
ⓐ name	message
⊟ e xs:complexType	
⊟ e xs:choice	
⊟ e xs:element	
ⓐ ref	response
⊟ e xs:element	
ⓐ ref	request
⊟ e xs:element	
ⓐ name	response
⊟ e xs:complexType	
⊟ e xs:choice	
⊟ e xs:element	
ⓐ ref	addResponse

The initial element in this schema is a **message**, which is defined to be either a **response** or a **request**. A **response** is a message that has a boolean **success** attribute and a string **reason** attribute if the response represents a failure. The **reason** attribute is declared to be **optional**, which mean a successful response doesn't need to include a meaningless **reason** attribute value.

A **request** is further subdivided into individual request types. The first (and only, so far) is **addRequest**. Similarly, a **response** is further subdivided into response types; **addResponse** is the only one so far. Each individual **request** (or **response**) is described in terms of the defined attributes and child elements that must be present. **addRequest** has a string **name** attribute and an **image** child element. The question remains: how is the image data going to be "inserted" into the XML message? In an earlier lab, you wrote code that embedded MIME-encoded data, but this only worked because of a sentinel "ADD-DONE" string that was unlikely to be part of the encoding. Similarly, the designers of XML created the ability to encode arbitrary ASCII data using an unparsed section which begins with **<![CDATA[** and ends with **]]>**. The terminating characters **]]>** do not appear in the MIME-encoded data format. This fragment represents a valid **addRequest**:

Valid addRequest XML fragment

```
<request>
  <addRequest name="sampleImage">
    <image>
       <![CDATA[iVBORw0KGgoAAAANSUhEUgAAAZAAAADwCAYAAAAuPDI...
       ]]>
    </image>
  </addRequest>
</request>
```

The next XML fragment describes the proper addResponse when an image is added to the repository successfully. The **success** attribute is part of the **response** element, as specified in the XSD file. Now you may be thinking, "where is the closing '</addResponse>' element which is required for well-formed XML?" Well, the XML designers allow for a shortcut when elements have no children. An element without children can be defined and closed with **/>**.

Valid AddResponse XML fragment

```
<response success="true">
  <addResponse numBytes="9160"/>
</response>
```

Go ahead and create some classes to manage these XML messages. Don't just embed XML string fragments throughout your code, because that would become a nightmare to manage and maintain.

In the **/src** folder, create an **xml** package.

In the **/src** folder's new **xml** package, create a **Message** class to handle most of the functionality required for parsing and constructing XML strings:

CODE TO TYPE: /src/xml/Message.java

```
package xml;

import java.io.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import javax.xml.validation.*;
import org.w3c.dom.*;
import org.xml.sax.*;

public class Message {
  static DocumentBuilder builder;

  static void configure() {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    factory.setNamespaceAware(true);
    SchemaFactory sf = SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
    try {
      factory.setSchema(sf.newSchema(new Source[] {new StreamSource("repository.xsd")})
);
      builder = factory.newDocumentBuilder();
    } catch (Exception e) {
      throw new RuntimeException ("Unable to configure Message");
    }
  }
}
```

Let's review some of the standard boilerplate code in the configure() method:

OBSERVE:

```
  static void configure() {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    factory.setNamespaceAware(true);
    SchemaFactory sf = SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
    try {
      factory.setSchema(sf.newSchema(new Source[] {new StreamSource("repository.xsd")})
);
      builder = factory.newDocumentBuilder();
    } catch (Exception e) {
      throw new RuntimeException ("Unable to configure Message");
    }
  }
```

Java has built-in XML support as defined in the javax.xml.* packages. The above code configures the **Message** class to use the schema as defined by **repository.xsd**. It constructs a singleton **builder** object to parse XML strings conforming to the **repository.xsd** schema definition you just created. This singleton **builder** object must be handled carefully, that's why outside classes are unable to access this object. Write the Message constructor which is the only method that uses the **builder** object:

CODE TO TYPE: /src/xml/Message.java

```
public class Message {
  ...

  public final Node contents;

  public Message (String xmlSource) throws IllegalArgumentException {
    if (builder == null) {
      configure();
    }

    try {
      InputSource is = new InputSource (new StringReader (xmlSource));

      // parse method in builder is not thread safe.
      Document d = null;
      synchronized (builder) {
        d = builder.parse(is);
      }

      // Grab first (and only) child (either request or response)
      NodeList children = d.getChildNodes();
      for (int i = 0; i < children.getLength(); i++) {
        Node n = children.item(i);
        if (n.getNodeType() == Node.ELEMENT_NODE) {
          contents = n;
          return;
        }
      }

      throw new IllegalArgumentException ("XML document has no child node");
    } catch (Exception e) {
      e.printStackTrace();
      throw new IllegalArgumentException (e.getMessage());
    }
  }
}
```

Let's discuss this addition in a little more detail.

OBSERVE:

```
public class Message {
  ...

  public final Node contents;

  public Message (String xmlSource) throws IllegalArgumentException {
    if (builder == null) {
      configure();
    }

    try {
      InputSource is = new InputSource (new StringReader (xmlSource));

      // parse method in builder is not thread safe.
      Document d = null;
      synchronized (builder) {
        d = builder.parse(is);
      }

      // Grab first (and only) child (either request or response)
      NodeList children = d.getChildNodes();
      for (int i = 0; i < children.getLength(); i++) {
        Node n = children.item(i);
        if (n.getNodeType() == Node.ELEMENT_NODE) {
          contents = n;
          return;
        }
      }

      throw new IllegalArgumentException ("XML document has no child node");
    } catch (Exception e) {
      e.printStackTrace();
      throw new IllegalArgumentException (e.getMessage());
    }
  }
}
```

If the **builder** has not yet been configured, the constructor self-configures; this is a useful technique to simplify your application initialization code. Also, access to the **builder.parse()** method is enclosed within a **synchronized (builder)** code block because that method is not "thread-safe." This ensures that with all simultaneous attempts to construct Message objects, no more than one will ever access the **parse()** method at the same time.

The real work for this class takes place in its constructor, which receives an XML string as an argument. The **builder** parses the **xmlSource** string using a StringReader object. The constructor will succeed only if the XML string conforms to the **repository.xsd** schema definition file. Once the parsing completes, the builder returns the root Document node for the Document Object Model (DOM) representing the XML string.

Given an XML string, the **builder** creates a tree-like recursive object that represents the structure and relationships encoded in the string. At this point, there is a single root Node object corresponding to the XML string. Using the existing XML API, you can navigate through the children (and grandchildren) nodes within the structure to locate all elements of the original XML string.

For now, you want just the first Node in the Document. The **for loop in the constructor** shows how to use the XML API to iterate over all of the children for a given Document (or Node) object. This loop retrieves a NodeList object over which you make repeated calls to **Node n = children.item(i)** ("get the ith child"). Once you have located a node of type ELEMENT_NODE, you know you have found either the request or the response, so this node serves as the **contents** for the Message. If no such element exists, then the string doesn't conform and an IllegalArgumentException can be thrown.

Before you use these XML messages, you need to validate that this Message class actually works.

In the **/test** folder, create an **xml** package.

In this new **xml** package, create a **ValidateXMLMessages** test case as shown:

CODE TO TYPE: /test/xml/ValidateXMLMessages.java

```java
package xml;

import server.model.*;
import junit.framework.TestCase;

public class ValidateXMLMessages extends TestCase {

  public void testAddRequest() {
    String addRequestSample = "<request><addRequest name='sampleImage'><image>DATA HERE
</image>" +
                              "</addRequest></request>";
    Message m = new Message(addRequestSample);
    assertEquals ("request", m.contents.getLocalName());
  }

  public void testAddResponseSuccess() {
    String addRequestSample = "<response success='true'><addResponse numBytes='9160'/><
/response>";
    Message m = new Message(addRequestSample);
    assertEquals ("response", m.contents.getLocalName());
  }

  public void testAddResponseFailure() {
    String addRequestSample = "<response success='false' reason='" + Repository.Already
ExistsImage + "'>" +
                              "<addResponse numBytes='0'/></response>";
    Message m = new Message(addRequestSample);
    assertEquals ("response", m.contents.getLocalName());
  }

  public void testFailedRequest() {
    try {
      String notInProtocol = "<request><missingRequest name='sampleImage'><image>DATA H
ERE</image>" +
                             "</missingRequest></request>";
      new Message(notInProtocol);
      fail("Should detect non-existing request.");
    } catch (Exception e) {
      // success
    }
  }
}
```

Each test case method follows a general approach. Let's look more closely at testAddRequest(); the other test case methods are similar:

OBSERVE:

```java
  public void testAddRequest() {
    String addRequestSample = "<request><addRequest name='sampleImage'><image>DATA HERE
</image>" +
                              "</addRequest></request>";
    Message m = new Message(addRequestSample);
    assertEquals ("request", m.contents.getLocalName());
  }
```

The test case method uses the Message constructor to instantiate a Message object from the sample **addRequestSample** string. If this is the first time that a Message object has been constructed, the configure() method presented earlier will be called immediately to properly load up the requisite **repository.xsd** object. Assuming the constructor returns successfully, the **assertEquals JUnit method validates that the (local) name of the Node associated with the contents of the message object is "request"**.

In conjunction, these test cases validate that you can construct an addRequest and two proper responses: one for success and one for failure. For completion, there is a test case for a non-existent request (missingRequest) and a test case where the input string does not contain well-formed XML. Run these test cases to make sure that all succeed.

What's this? testFailedRequest() allows a Message object to be constructed? Let's take a closer look at this test case method:

testFailedRequest test case method

```
 public void testFailedRequest() {
    try {
      String notInProtocol = "<request><missingRequest name='sampleImage'><image>DATA H
ERE</image>" +
                             "</missingRequest></request>";
      new Message(notInProtocol);
      fail("Should detect non-existing request.");
    } catch (Exception e) {
      // success
    }
  }
```

As you can see, there is no missingRequest in the schema definition file, so what is this all about? The issue is a subtle one and occurs because of the way that the XML technology was developed and adopted. *You* are responsible for checking the errors that occur and it is *your job* to determine how to deal with validation errors. To do that, you'll need to register with **builder** an error handler that implements org.xml.sax.ErrorHandler.

In the **/src** folder's **xml** package, create an **XMLHandler** class to deal with validation errors properly:

CODE TO TYPE: /src/xml/XMLHandler.java

```
package xml;

import java.util.*;
import org.xml.sax.*;

public class XMLHandler implements ErrorHandler {
  ArrayList<String> errors = new ArrayList<String>();

  /** Keep record of all errors and continue until failFast is called. */
  public void error(SAXParseException spe) throws SAXException {
    errors.add(spe.toString());
  }

  /** Fail immediately with fatal errors. */
  public void fatalError(SAXParseException spe) throws SAXException {
    throw spe;
  }

  /** Emit warnings as they come and otherwise ignore. */
  public void warning(SAXParseException spe) throws SAXException {
    System.out.println("WARNING: " + spe.toString());
  }

  /** Terminate immediately upon detecting any XML errors. */
  public void failFast() {
    if (errors.size() == 0) {
      return;
    }

    for (String s : errors) {
      System.out.println("ERROR:" + s);
    }
    errors.clear();
    throw new RuntimeException ("Parsing Failed");
  }
}
```

Let's look at this class more closely.

OBSERVE:

```
...
  ArrayList<String> errors = new ArrayList<String>();

  /** Keep record of all errors and continue until failFast is called. */
  public void error(SAXParseException spe) throws SAXException {
    errors.add(spe.toString());
  }

  /** Fail immediately with fatal errors. */
  public void fatalError(SAXParseException spe) throws SAXException {
    throw spe;
  }

  /** Emit warnings as they come and otherwise ignore. */
  public void warning(SAXParseException spe) throws SAXException {
    System.out.println("WARNING: " + spe.toString());
  }

  /** Terminate immediately upon detecting any XML errors. */
  public void failFast() {
    if (errors.size() == 0) {
      return;
    }

    for (String s : errors) {
      System.out.println("ERROR:" + s);
    }
    errors.clear();
    throw new RuntimeException ("Parsing Failed");
  }
...
```

The builder XML parser identifies "warnings," "errors," and "fatal errors" and then invokes the appropriate method on the registered error handler. If no error handler is registered, no validation takes place! Once you register this error handler with the builder, all **warnings are output to System.out**, all **fatal errors cause an exception to be thrown** (thus halting the parsing) and **all errors are collected in an ArrayList object**. After the document has been parsed, you call **failFast** to determine whether any errors occured; if there were any, **they are output to System.out and a RunTimeException is thrown**. The errors **ArrayList must be cleared** to avoid having future parse requests fail incorrectly, since all parsing uses the same builder object (and by extension the same error handler).

Finally, to make sure there will never be concurrent shared usage of this handler, the invocation to **failFast** must occur within a **synchronized** block. Modify the **Message** class as shown:

CODE TO TYPE: /src/xml/Message.java

```
package xml;

import java.io.*;

import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import javax.xml.validation.*;
import org.w3c.dom.*;
import org.xml.sax.*;

public class Message {
  static DocumentBuilder builder;
  static final XMLHandler errorHandler = new XMLHandler();
  public final Node contents;

  public Message (String xmlSource) throws IllegalArgumentException {
    if (builder == null) {
      configure();
    }

    try {
      InputSource is = new InputSource (new StringReader (xmlSource));

      // parse method in builder is not thread safe.
      Document d = null;
      synchronized (builder) {
        d = builder.parse(is);
        errorHandler.failFast();
      }

      // Grab first (and only) child (either request or response)
      NodeList children = d.getChildNodes();
      for (int i = 0; i < children.getLength(); i++) {
        Node n = children.item(i);
        if (n.getNodeType() == Node.ELEMENT_NODE) {
          contents = n;
          return;
        }
      }

      throw new IllegalArgumentException ("XML document has no child node");
    } catch (Exception e) {
      e.printStackTrace();
      throw new IllegalArgumentException (e.getMessage());
    }
  }

  static void configure() {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    factory.setNamespaceAware(true);
    SchemaFactory sf = SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
    try {
      factory.setSchema(sf.newSchema(new Source[] {new StreamSource("repository.xsd")})
);
      builder = factory.newDocumentBuilder();
      builder.setErrorHandler(errorHandler);
    } catch (Exception e) {
      throw new RuntimeException ("Unable to configure Message");
    }
  }
}
```

Running the **ValidateXMLMessages** test case should convince you that each Message object conforms to a valid XML string. Now, before you get too satisfied, add the following testBadXML() test case method to be the first test case method in **ValidateXMLMessages**:

CODE TO TYPE: /test/xml/ValidateXMLMessages.java

```
package xml;

import server.model.*;
import junit.framework.TestCase;

public class ValidateXMLMessages extends TestCase {

  public void testBadXML() {
    try {
      String bad = "<req>Not even XML< ><REQ/>";
      new Message(bad);
      fail("Should detect non-existing request.");
    } catch (Exception e) {
      // success
    }
  }
...
```

This test case method uses a string that isn't valid XML structure. After adding this method, run your test cases. Existing test case methods that passed before might now fail! Surely this is a sign of something more significant. Go back and review the **Message** constructor and you'll see that the outermost exception handler doesn't call **errorHandler.failFast()** properly. Correct this oversight as shown:

CODE TO TYPE: /src/xml/Message.java

```
...
  public Message (String xmlSource) throws IllegalArgumentException {
    if (builder == null) {
      configure();
    }

    try {
      InputSource is = new InputSource (new StringReader (xmlSource));

      // parse method in builder is not thread safe.
      Document d = null;
      synchronized (builder) {
        d = builder.parse(is);
        errorHandler.failFast();
      }

      // Grab first (and only) child (either request or response)
      NodeList children = d.getChildNodes();
      for (int i = 0; i < children.getLength(); i++) {
        Node n = children.item(i);
        if (n.getNodeType() == Node.ELEMENT_NODE) {
          contents = n;
          return;
        }
      }

      throw new IllegalArgumentException ("XML document has no child node");
    } catch (Exception e) {
      errorHandler.failFast();
      e.printStackTrace();
      throw new IllegalArgumentException (e.getMessage());
    }
  }
...
```

Run the tests again and they should all pass.

Status Messages

The XML schema definition file is constructed to be readily extensible as new requests (and responses) are defined. As a general principle, each request will have a corresponding response to allow the client to know that the request was properly received and acted upon. Given this principle, what should be the first request that the client makes when connecting to the server? Or to think of this another way, what should be the first response that is sent to the client? Modify the XSD file to support a **statusRequest** that contains no attributes or other information. This could be used as the first client request. The corresponding **statusResponse** will return information about the current image being viewed and the total number of images in the repository. The schema should be changed to support the following XML fragment:

Sample statusResponse XML fragment

```
<response success="true">
  <statusResponse key="c00bc1ed28fabdbcebc3e4735decc83e" index="1" total="17"/>
</response>
```

This message encodes the state of the repository (having 1 of 17 images) and the specific key of the image being observed by the client. Start by modifying the **repository.xsd** file:

CODE TO TYPE: /repository.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>

<xs:element name='message'>
  <xs:complexType>
    <xs:choice>
      <xs:element ref='response'/>
      <xs:element ref='request'/>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name='response'>
  <xs:complexType>
    <xs:choice>
      <xs:element ref='addResponse'/>
      <xs:element ref='statusResponse'/>
    </xs:choice>
    <xs:attribute name='success' type='xs:boolean' use='required'/>
    <xs:attribute name='reason'  type='xs:string'  use='optional'/>
  </xs:complexType>
</xs:element>

<xs:element name='request'>
  <xs:complexType>
    <xs:choice>
      <xs:element ref='addRequest'/>
      <xs:element ref='statusRequest'/>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name='addRequest'>
  <xs:complexType>
   <xs:sequence>
     <xs:element name='image'/>
   </xs:sequence>
   <xs:attribute name='name' type='xs:string' use='required'/>
  </xs:complexType>
</xs:element>

<xs:element name='addResponse'>
  <xs:complexType>
    <xs:attribute name='numBytes' type='xs:integer' use='required'/>
  </xs:complexType>
</xs:element>

<xs:element name='statusRequest'/>

<xs:element name='statusResponse'>
  <xs:complexType>
    <xs:attribute name='key'   type='xs:string'  use='required'/>
    <xs:attribute name='index' type='xs:integer' use='required'/>
    <xs:attribute name='total' type='xs:integer' use='required'/>
  </xs:complexType>
</xs:element>
</xs:schema>
```

Of course, you have to extend the **ValidateXMLMessages** test case to validate that you can properly parse both of these messages:

CODE TO TYPE: /test/xml/ValidateXMLMessages.java

```
...
  public void testStatusRequest() {
    String statusSample = "<request><statusRequest/></request>";
    Message m = new Message(statusSample);
    assertEquals ("request", m.contents.getLocalName());
  }

  public void testStatusResponse() {
    String statusResponse = "<response success='true'><statusResponse key='asdkjhkas' i
ndex='1' total='17'/></response>";
    Message m = new Message(statusResponse);
    assertEquals ("response", m.contents.getLocalName());
  }
...
```

Now, relaunch all JUnit test cases to validate your code so far. Defining a protocol is a necessary first step toward the implementation of the overall application. Once the skeleton support is in place, you can begin to add pairs of messages (that is, requests and responses). Along the way, adhere to the philosophy that has guided you so far: testing incrementally as you go. Instead of trying to codify the entire protocol in advance, you are now ready to begin integrating the protocol into the application, as you will do in the next lab.

Get busy with this lesson's homework and I'll see you soon!

XML Protocol Implementation

Lesson Objectives

In this lesson you will:

- send requests and receive responses over the IPC layer.
- retrieve information from XML objects using the Java XML API.

XML Protocol Implementation

Now that you've defined the XML protocol, you need to modify the client to send XML requests, and the server to receive them and send XML responses in return. Note that you don't need to change the IPC layer itself, only the **ProtocolHandler** class. The changes are substantial because they reflect a different abstraction: the transfer of an XML message instead of the current line-by-line transfer of ASCII text.

The **process()** method in **ProtocolHandler** is still responsible for handling the protocol using the raw input and output objects provided by the IPC layer, but now it concatenates each line of input until a full request is seen. To facilitate that process, the client makes sure that each request is sent with a trailing new-line character. This ensures that the **process()** method will read a line of ASCII text at some point, that ends with **"</request>"** in the XML representation of the request. From the raw XML, a **Message** object is constructed and handed off to a **process(Message)** method, which acts on the request and returns the appropriate response. This response **Message** is converted back into a raw XML string to be written to the socket which communicates back to the client.

Your first task is to replace **ProtocolHandler** with the implementation below. You can delete the **output()** and **outputError()** methods and replace the existing **process()** method as shown:

CODE TO TYPE: /src/server/ProtocolHandler.java

```
package server;

import java.io.*;
import server.ipc.*;
import server.model.*;
import util.*;
import xml.*;
import org.w3c.dom.*;

public class ProtocolHandler implements IProtocolHandler {
  final Repository repository;
  public static final String endRequest = "</request>";

  public ProtocolHandler (Repository r) {
    repository = r;
  }

  public boolean process(BufferedReader fromSocket, PrintWriter toSocket) {
    try {
      String line = fromSocket.readLine();
      if (line == null) { return false; }
      StringBuilder buf = new StringBuilder(line);
      while (!buf.substring(buf.length() - endRequest.length(), buf.length()).equals(en
dRequest)) {
        line = fromSocket.readLine();
        if (line == null) { return false; }
        buf.append(line);
      }

      Message request = new Message (buf.toString());
      Message response = process (request);
      toSocket.println(response.toString());
      return !toSocket.checkError();
    } catch (Exception e) {
      e.printStackTrace();
      return false;
    }
  }
}
```

This code won't compile until you write a **process(Message)** method (we'll do that in just a bit). First, check out the revised **process(BufferedReader, PrintWriter)** method which contains the **while** loop for processing input strings from **BufferedReader**:

OBSERVE:

```
  String line = fromSocket.readLine();
  if (line == null) { return false; }
  StringBuilder buf = new StringBuilder(line);

  while (!buf.substring(buf.length()-endRequest.length(), buf.length()).equals(endReque
st)) {
    line = fromSocket.readLine();
    if (line == null) { return false; }
    buf.append(line);
  }
```

This code concatenates string lines read from the client until the most recently read string is terminated by the "</request>" string (the **endRequest** constant attribute stores this value). The rather complicated-looking condition to the **while** loop, checks to see if the **buf StringBuilder** object ends with the **endRequest** string. If it does not, the program reads another string line from the **fromSocket** and appends it to **buf**.

Once that **while** loop completes, **buf** contains the full XML string request from a client. The remaining code in the method constructs a **Message** object to be processed by a **process(Message)**, and the resulting **Message** response is returned to the client. Let's write this **process(Message)** method. Start by entering this method skeleton:

CODE TO TYPE: /src/server/ProtocolHandler.java

```
...
  public Message process (Message request) {
    Node child = request.contents.getFirstChild();
    if (child.getLocalName().equals ("addRequest")) {
      String name = child.getAttributes().getNamedItem("name").getNodeValue();
      Node imageNode = child.getFirstChild();

      // TODO: Fill in processing of addRequest
      return null;
    }

    return null;  // unknown request? No idea what to do.
  }
...
```

Take a closer look:

OBSERVE:

```
  public Message process (Message request) {
    Node child = request.contents.getFirstChild();
    if (child.getLocalName().equals ("addRequest")) {
      String name = child.getAttributes().getNamedItem("name").getNodeValue();
      Node imageNode = child.getFirstChild();

      // TODO: Fill in processing of addRequest
      return null;
    }

    return null;  // unknown request? No idea what to do.
  }
```

We **use the XML API to process the Message DOM object**. The method above contains the essential XML API calls that you'll use when dealing with XML objects. The parsed XML is represented as a tree-like data structure with nodes that represent the elements of the XML document being parsed. The **Message** class already stores as **contents**, the node associated with the request (or response) message. This node corresponds to the **"<request>"** (or **"<response>"**) element in **repository.xsd**. So, in order to identify the request, we look at the child of the **contents** node. The Java API provides all the methods we need to go through a DOM:

1. Given a **Node** with children, getFirstChild() returns the first child node; this convenience method helps you navigate down a tree quickly when you know there is only a single child.
2. getAttributes() returns a NamedNodeMap which is in essence a hash table with keys that are the attribute names, with values that are Nodes that store the attributes' values in XML.
3. Given a **Node**, getNodeValue() returns the associated **String** value; this allows you to get the value for an attribute.
4. getTextContent() returns the text associated with an element. As a markup language, XML allows fragments of the form <Object>SomeValueHere</Object>. Use **getTextContent()** on the **Node** for the given Object element to retrieve the text associated with the element (in this case, "SomeValueHere").

You can use these API methods when completing the implementation of **process(Message)**:

CODE TO TYPE: /src/server/ProtocolHandler.java

```
  public static final String CorruptedImageData = "Encoded image data appears to be cor
rupted.";

  public Message process (Message request) {
    Node child = request.contents.getFirstChild();
    if (child.getLocalName().equals ("addRequest")) {
      String name = child.getAttributes().getNamedItem("name").getNodeValue();
      Node imageNode = child.getFirstChild();

      String xmlResp;
      try {
        byte[] bytes = ImageEncoding.decode(imageNode.getTextContent());
        repository.add(bytes, name);
        xmlResp = "<response success='true'><addResponse numBytes='" + bytes.length + "
'/></response>";
      } catch (IOException e) {
        xmlResp = "<response success='false' reason='" + CorruptedImageData + "'>" +
                "<addResponse numBytes='0'/></response>";
      } catch (IllegalStateException e) {
        xmlResp = "<response success='false' reason='" + Repository.AlreadyExistsImage
+ "'>" +
                "<addResponse numBytes='0'></addResponse></response>";
      }

      return new Message(xmlResp);
      ~~// TODO: Fill in processing of addRequest~~
      ~~return null;~~
    }

    return null;  // unknown request? No idea what to do.
  }
```

The code above adds the constant **CorruptedImageData**, which stores the error message for later testing:

OBSERVE:

```
  public static final String CorruptedImageData = "Encoded image data appears to be cor
rupted.";
```

Check out the core logic of this new code:

OBSERVE:

```
  byte[] bytes = ImageEncoding.decode(imageNode.getTextContent());
  repository.add(bytes, name);
  xmlResp = "<response success='true'><addResponse numBytes='" + bytes.length + "'/></r
esponse>";
```

This code uses the **imageNode.getTextContents()** API call to retrieve the string that contains the MIME-encoded bytes for the image being sent by the client. Next, the **ImageEncoding** class you created earlier decodes this string into a proper **byte[]** array to be inserted into the repository.

To try out this new capability, modify **RepositoryClient** as shown:

CODE TO TYPE: /test/client/RepositoryClient.java

```
package client;

import java.io.*;
import java.net.*;
import xml.*;
import util.*;

public class RepositoryClient {

  public static void main(String[] args) throws Exception {
    SplashScreenLogic.update ("connecting to localhost::9172");
    delay(250);
    Socket server = new Socket ("localhost", 9172);

    SplashScreenLogic.update ("connected to localhost::9172");
    delay(250);

    PrintWriter toServer = new PrintWriter (server.getOutputStream(), true);
    BufferedReader fromServer = new BufferedReader (new InputStreamReader(server.getInp
utStream()));
    SplashScreenLogic.update ("initializing with server...");
    delay(250);

~~    for (int num = 0; num < 3; num++) {~~
~~      toServer.println("SIZE");~~
~~      if (!toServer.checkError()) {~~
~~        Integer response = Integer.valueOf(fromServer.readLine());~~
~~        String value = fromServer.readLine();~~
~~        if (response == 0) {~~
~~          System.out.println((num+1) + ": Number of Images: " + value);~~
~~        } else if (response == -1) {~~
~~          System.err.println(value);~~
~~        } else {~~
~~          System.err.println("Received unknown response:" + response);~~
~~        }~~
~~      }~~
~~    }~~

    File f = new File("images", "repositorySplash.png");
    String encoding = ImageEncoding.encode(f);
    String xmlAddRequest = "<request><addRequest name='sampleImage'>" +
         "<image>\n<![CDATA[" + encoding + "\n]]></image></addRequest></request>";

    new Message(xmlAddRequest);
    toServer.println(xmlAddRequest);

    processResponse(fromServer);

    server.close();
    SplashScreenLogic.update ("closing");
    delay(250);
  }

  /** Delay for a time. */
  static void delay(int ms) {
    try { Thread.sleep(ms); } catch (InterruptedException ie) { }
  }
}
```

This code won't compile until you write the **processResponse(Message)** method, but let's check out the logic anyway:

OBSERVE:

```
  File f = new File("images", "repositorySplash.png");
  String encoding = ImageEncoding.encode(f);
  String xmlAddRequest = "<request><addRequest name='sampleImage'>" +
        "<image>\n<![CDATA[" + encoding + "\n]]></image></addRequest></request>";

  new Message(xmlAddRequest);
  toServer.println(xmlAddRequest);

  processResponse(fromServer);
```

Using existing logic, this code computes the MIME-encoded string for the specific image. In this case, it constructs an XML **addRequest** that embeds this encoded string using the familiar **CDATA** construct. You invoke **new Message(xmlAddRequest)** to validate the XML string. If there had been a problem in the XML encoding, this constructor invocation would've throw an Exception. If there is no Exception, the XML is valid—which means the string can be output directly to the server via the **toServer** object. Every request must be terminated by an end-of-line character, so it's essential to use the **println** method when writing to the server socket.

The **processResponse()** method you'll create next is similar to code in **ProtocolHandler**. First, you'll aggregate lines of input until a full response is retrieved, which you'll pass on to the **Message** constructor. This code allows you to retrieve the **success** and **reason** attributes from the response efficiently:

CODE TO TYPE: /test/client/RepositoryClient.java

```
public class RepositoryClient {
  ...

  public static String endResponse = "</response>";

  static void processResponse(BufferedReader fromServer) throws IOException {
    try {
      StringBuilder buf = new StringBuilder(fromServer.readLine());
      while (!buf.substring(buf.length() - endResponse.length(), buf.length()).equals(e
ndResponse)) {
        buf.append(fromServer.readLine());
      }
      Message response = new Message(buf.toString());
      String sval = response.contents.getAttributes().getNamedItem("success").getNodeVa
lue();
      if (Boolean.valueOf(sval)) {
        System.out.println("Success");
      } else {
        System.out.println("Error:" + response.contents.getAttributes().getNamedItem("r
eason").getNodeValue());
      }

    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
  }

  ...
}
```

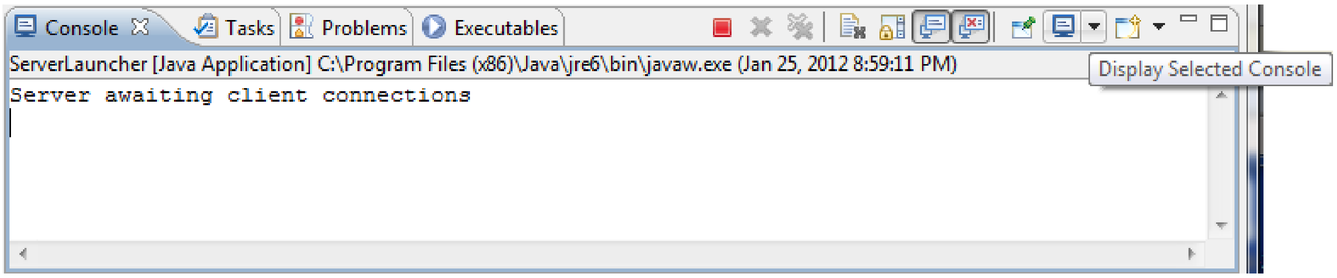
OBSERVE:

```
public class RepositoryClient {
  ...

  public static String endResponse = "</response>";

  static void processResponse(BufferedReader fromServer) throws IOException {
    try {
      StringBuilder buf = new StringBuilder(fromServer.readLine());
      while (!buf.substring(buf.length()-endResponse.length(), buf.length()).equals(end
Response)) {
        buf.append(fromServer.readLine());
      }
      Message response = new Message(buf.toString());
      String sval = response.contents.getAttributes().getNamedItem("success").getNodeVa
lue();
      if (Boolean.valueOf(sval)) {
        System.out.println("Success");
      } else {
        System.out.println("Error:" + response.contents.getAttributes().getNamedItem("r
eason").getNodeValue());
      }

    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
  }

  ...
}
```

This code is similar to the code you wrote for the server to process the client request. The **highlighted code determines whether the response is a success** by finding the value of the **success** attribute in the response.

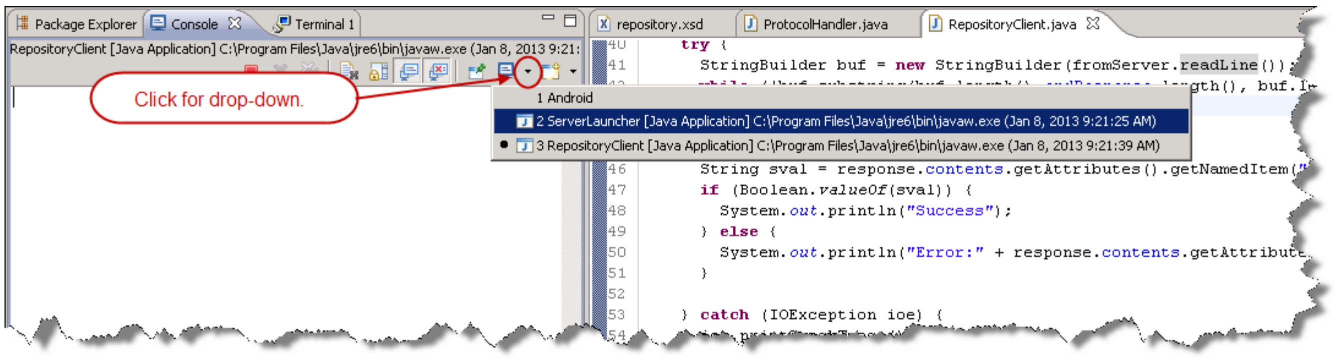
Note Before continuing, refresh the **Repository** folder and delete any Index and image files there.

Validate that you have coded the XML logic properly, by running **ServerLauncher** and then **RepositoryClient**. What's this? It seems as if the **RepositoryClient** just hangs and doesn't return. Even the splash screen is stuck on "Initializing with server..."



```
ServerLauncher [Java Application] C:\Program Files (x86)\Java\jre6\bin\javaw.exe (Jan 25, 2012 8:59:11 PM)
Server awaiting client connections
```

In the console tab, there's a pull-down menu on the right that allows you to "Display Selected Console." Use this menu to switch between the running applications; you can see that the server is running properly ("Server awaiting client connections"), but there is no output on the client side. Given the **RepositoryClient**, it appears that the server received a request, but a response was never returned to the client. Terminate the execution of both the client and server applications.



RepositoryClient [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 8, 2013 9:21:

2 ServerLauncher [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 8, 2013 9:21:25 AM)

3 RepositoryClient [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 8, 2013 9:21:39 AM)

```
    try {
      StringBuilder buf = new StringBuilder(fromServer.readLine());
                                                  gth(), buf.l
      String sval = response.contents.getAttributes().getNamedItem("
      if (Boolean.valueOf(sval)) {
        System.out.println("Success");
      } else {
        System.out.println("Error:" + response.contents.getAttribute
      }

    } catch (IOException ioe) {
```

You might choose to set breakpoints now and then execute these applications in the Eclipse debugger to identify where the problem occurs. I recommend doing that. You want to become familiar with debugging single and multiple Java applications. For now, turn your attention to the **ProtocolHandler** code that you revised at the beginning of this lab—specifically, these two lines:

OBSERVE:

```
Message response = process (request);
toSocket.println(response.toString());
```

Compare those two lines with the changes introduced in **RepositoryClient** as shown below:

OBSERVE:

```
new Message(xmlAddRequest);
toServer.println(xmlAddRequest);
```

The first line validates the XML string properly, but the constructed **Message** object is promptly ignored, because it's not needed. The second line takes the (now validated) XML string and prints it right to the **PrintWriter** used to communicate to the server. Review the **ProtocolHandler** code now and you'll see that it depends upon the **Message** class having a working toString() method! Whoops! You've just encountered one of the most common defects in Java: it omits a necessary **toString** method.

Add the logic needed for **toString** to **Message** as shown below. As with most XML technologies, you might be surprised at how unnecessarily complicated it appears; nonetheless, this is the standard way to convert a DOM into its string representation:

CODE TO TYPE: /src/xml/Message.java

```
package xml;

import java.io.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import javax.xml.transform.dom.*;
import javax.xml.validation.*;
import org.w3c.dom.*;
import org.xml.sax.*;

public class Message {
  static DocumentBuilder builder;
  static final XMLHandler errorHandler = new XMLHandler();
  static Transformer transformer;
  public final Node contents;

  public Message (String xmlSource) throws IllegalArgumentException {
    if (builder == null) {
      configure();
    }

    try {
      InputSource is = new InputSource (new StringReader (xmlSource));

      // parse method in builder is not thread safe.
      Document d = null;
      synchronized (builder) {
        d = builder.parse(is);
        errorHandler.failFast();
      }

      // Grab first (and only) child (either request or response)
      NodeList children = d.getChildNodes();
      for (int i = 0; i < children.getLength(); i++) {
        Node n = children.item(i);
        if (n.getNodeType() == Node.ELEMENT_NODE) {
          contents = n;
          return;
        }
      }
      throw new IllegalArgumentException ("XML document has no child node");
    } catch (Exception e) {
      errorHandler.failFast();
      e.printStackTrace();
      throw new IllegalArgumentException (e.getMessage());
    }
  }

  static void configure() {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    factory.setNamespaceAware(true);
    SchemaFactory sf = SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
    try {
      factory.setSchema(sf.newSchema(new Source[] {new StreamSource("repository.xsd")})
);
      builder = factory.newDocumentBuilder();
      builder.setErrorHandler(errorHandler);
    } catch (Exception e) {
      throw new RuntimeException ("Unable to configure Message");
    }

    TransformerFactory tf = TransformerFactory.newInstance();
    try {
      transformer = tf.newTransformer();
    } catch (TransformerConfigurationException tce) {
      tce.printStackTrace();
    }
  }

  public String toString() {
    DOMSource domSource = new DOMSource(contents);
    StringWriter writer = new StringWriter();
    StreamResult result = new StreamResult(writer);
    try {
      transformer.transform(domSource, result);
      return writer.toString();
    } catch (Exception e) {
      return "";
    }
  }
}
```

If you haven't terminated the client and server applications, do that now. Also, delete any files in the **Repository** folder. Now relaunch **ServerLauncher** and execute **RepositoryClient**. The client output in Eclipse should say, "Success." Verify the logic of the repository by executing **RepositoryClient** again to detect that the image is already part of the repository and that this second request is to be denied. Your output should read, "Error:That image already exists in the repository." Be sure to terminate the server application before continuing.

Extending Protocol Implementation with Status Messages

Modify the **RepositoryClient** to issue a **statusRequest** request at startup:

CODE TO TYPE: /test/client/RepositoryClient

```
...
  public static void main(String[] args) throws Exception {
    SplashScreenLogic.update ("connecting to localhost::9172");
    delay(250);
    Socket server = new Socket ("localhost", 9172);

    SplashScreenLogic.update ("connected to localhost::9172");
    delay(250);

    PrintWriter toServer = new PrintWriter (server.getOutputStream(), true);
    BufferedReader fromServer = new BufferedReader (new InputStreamReader(server.getInp
utStream()));
    SplashScreenLogic.update ("initializing with server...");
    delay(250);

    String xmlStatusRequest = "<request><statusRequest/></request>";
    new Message(xmlStatusRequest);
    toServer.println(xmlStatusRequest);
    processResponse(fromServer);

    File f = new File("images", "repositorySplash.png");
    String encoding = ImageEncoding.encode(f);
    String xmlAddRequest = "<request><addRequest name='sampleImage'>" +
         "<image>\n<![CDATA[" + encoding + "\n]]></image></addRequest></request>";

    new Message(xmlAddRequest);
    toServer.println(xmlAddRequest);

    processResponse(fromServer);

    server.close();
    SplashScreenLogic.update ("closing");
    delay(250);
  }
...
```

Run **ServerLauncher** and then run **RepositoryClient**. A **NullPointerException** appears in the Eclipse Console in the **process** method of the **ProtocolHandler** class:

OBSERVE:

```
Server awaiting client connections
Exception in thread "Thread-0" java.lang.NullPointerException
  at server.ProtocolHandler.process(ProtocolHandler.java:38)
  at server.ipc.RepositoryThread.run(RepositoryThread.java:30)
```

If you click on the link to the **process** method in the exception stack trace, you will find that the **response** object returned by **process()** is **null**. Of course! You haven't yet modified the **ProtocolHandler** class to deal with a **statusRequest** that might be sent to the server. Modify the **process(Message)** method in **ProtocolHandler** as shown:

CODE TO TYPE: /src/server/ProtocolHandler.java

```
...
  public Message process (Message request) {
    Node child = request.contents.getFirstChild();
    if (child.getLocalName().equals ("addRequest")) {
      String name = child.getAttributes().getNamedItem("name").getNodeValue();
      Node imageNode = child.getFirstChild();
      String xmlResp;
      try {
        byte[] bytes = ImageEncoding.decode(imageNode.getTextContent());
        repository.add(bytes, name);
        xmlResp = "<response success='true'><addResponse numBytes='" + bytes.length + "
'/></response>";
      } catch (IOException e) {
        xmlResp = "<response success='false' reason='" + CorruptedImageData + "'>" +
          "<addResponse numBytes='0'/></response>";
      } catch (Exception e) {
        xmlResp = "<response success='false' reason='" + e.getMessage() + "'>" +
              "<addResponse numBytes='0'/></response>";
      }

      return new Message(xmlResp);
    } else if (child.getLocalName().equals("statusRequest")) {
      String xmlResp = "<response success='true'>" +
            "<statusResponse key='SomeKey' index='1' total='" + repository.size() + "'/
>" +
            "</response>";
      return new Message(xmlResp);
    }

    return null;     // unknown request? No idea what to do.
  }
}
...
```

The server responds to a **statusRequest** with a **statusResponse** that records information about the repository. This code also contains a scaffolding element in that **statusResponse** requires a **key** value to reflect the fingerprint of the "current image" being viewed by the client. Until that logic is implemented, a dummy **SomeKey** value is used instead, to get the string to pass XML validation.

Now terminate and re-run the **ServerLauncher** and **RepositoryClient**. The client produces two output messages; the first is **Success** (the response to the statusRequest) and the second is either **Error:That image already exists in the repository** (if you didn't clear out the repository), or **Success** (if you did).

You've powered through more than half of this advanced Java course. Good job! You'll continue extending your code to add new messages (requests and responses). This table shows how far we've come toward meeting the original set of requirements and goals we set:

R#	Status	Description
R1	DONE	Server must allow up to 30 concurrent users to connect and browse the images stored there.

R2		Client must be able to support any of the standard built-in Java image formats (such as PNG or JPG).
R3		Server can be configured to limit the maximum size of any individual image file (default: 5MB).
R4		Server can be configured to limit the total number of files stored on the shared repository (default: 1,000).
R5		A user connecting to a server must provide a user name and password.
R6	80%	A user can upload up to a fixed number of images to the repository (default: 100).
R7		A user can delete any image that he has added to the repository; a user cannot delete images added by another user.
R8		A user can self-register an account with the Server.
R9		During the client-server communication, the user's password never appears in plaintext format.
R10		A user account is considered *inactive* if the user has not connected to the server within a fixed time period (default: 14 days).
R11		Each user account has a unique string identifier composed of alphanumeric characters (a-zA-Z0-9). The server only stores the hashed value of the password and therefore does not know it.

The foundation has been laid and you are ready to tackle the remaining requirements with confidence!

On a final note, since you're keeping score at home, what is your code coverage? Let's find out. Be sure to terminate any running programs (such as **ServerLauncher**). First, execute all test cases within the **test** source folder and observe that the existing **TestAddBehavior** test case no longer works because it was written before the XML protocol was defined. To update, replace the old "SIZE" requests with the **statusRequest** added in this lab. Let's get started by fixing the methods to test and validate the success or failure of a response. Make the following code changes to **TestAddBehavior** to reimplement **expectSuccess**.

CODE TO TYPE: /test/server.ipc/TestAddBehavior.java

```
package server.ipc;

import java.io.*;
import java.net.*;
import server.model.*;
import util.*;
import xml.*;
import junit.framework.TestCase;

public class TestAddBehavior extends TestCase {
  ...
  public static final String endResponse = "</response>";

  public static Message processResponse(BufferedReader fromServer) throws IOException {
    // Accumulate all input until terminating </response>
    StringBuilder buf = new StringBuilder(fromServer.readLine());
    while (!buf.substring(buf.length() - endResponse.length(), buf.length()).equals(end
Response)) {
      buf.append(fromServer.readLine());
    }
    return new Message(buf.toString());
  }

  public static Message expectSuccess (BufferedReader fromServer) throws IOException {
    Message response = processResponse(fromServer);
    String sval = response.contents.getAttributes().getNamedItem("success").getNodeValu
e();
    assertTrue (Boolean.valueOf(sval));
    return response;
  }

  ...
}
```

Let's take a closer look at the revised expectSuccess() method. You've already seen the logic for processResponse().

expectSuccess() uses **processResponse**:

OBSERVE:

```
  public static Message expectSuccess (BufferedReader fromServer) throws IOException {
    Message response = processResponse(fromServer);
    String sval = response.contents.getAttributes().getNamedItem("success").getNodeValu
e();
    assertTrue (Boolean.valueOf(sval));
    return response;
  }
```

This method returns the **Message** response received from the server so that it can be inspected in detail by your test case methods. The required **XML API calls that extract attributes from the XML response are highlighted in red**. Now, replace requestSIZE() with requestSTATUS() in **TestAddBehavior** as shown:

CODE TO TYPE: /test/server.ipc/TestAddBehavior.java

```
  ~~void requestSIZE() throws IOException {~~
~~    toServer.println("SIZE");~~
~~  }~~

  public static void requestSTATUS (PrintWriter out) {
    String xmlStatusRequest = "<request><statusRequest/></request>";
    new Message(xmlStatusRequest);
    out.println(xmlStatusRequest);
  }
```

This method is **public static**, so it can be reused by future test case methods that you write. Now rewrite the **requestADD** method to use the XML protocol:

CODE TO TYPE: /test/server.ipc/TestAddBehavior.java

```
  public static void requestADD(PrintWriter out, String name, File f) throws IOExceptio
n {
~~    // send splash file as IMAGE using (ADD, NAME, IMAGE encoded)~~
~~    toServer.println("ADD BEGIN");~~
~~    toServer.println(name);~~
~~    toServer.print(ImageEncoding.encode(f);~~
~~    toServer.println("\nADD DONE");~~
    String encoding = ImageEncoding.encode(f);
    String xmlAddRequest = "<request><addRequest name='" + name + "'>" +
          "<image>\n<![CDATA[" + encoding + "\n]]></image></addRequest></request>";

    new Message(xmlAddRequest);
    out.println(xmlAddRequest);
  }
```

This revised method uses familiar logic for building XML strings; it also validates the XML before sending the string along to the server. Now you can update the **testAddBehavior** test case method to take full advantage of these updated helper methods:

CODE TO TYPE: /test/server.ipc/TestAddBehavior.java

```
  public void testAddBehavior() throws Exception {
    ~~// Protocol for sending SIZE~~
~~    toServer.println("SIZE");~~
    requestSTATUS(toServer);
    expectSuccess(~~"0", ~~fromServer);

    ~~// Protocol for sending an image~~
~~    toServer.println("ADD BEGIN");~~
~~    toServer.println("sampleImage");~~
    File f = new File("images", "repositorySplash.png");
~~    toServer.println(ImageEncoding.encode(f));~~
~~    toServer.println("\nADD DONE");~~
    requestADD(toServer, "sampleImage", f);
    expectSuccess(~~null, ~~fromServer);

    ~~// Expect repository with 1 image~~
~~    toServer.println("SIZE");~~
    requestSTATUS(toServer);
    Message response = expectSuccess(~~"1", ~~fromServer);
    String sval = response.contents.getFirstChild().getAttributes().getNamedItem("total
").getNodeValue();
    assertEquals ("1", sval);
  }
```

These changes clean up the code used for communication between the client and the server, and takes advantage of the **Message** class representing XML strings. To complete your test cases, make similar changes to support the revised **testBasicAddBehavior** test case method. First, rewrite the **expectFailure()** method in **TestAddBehavior**:

CODE TO TYPE: /test/server.ipc/TestAddBehavior.java

```
  Message expectFailure (String expect, BufferedReader fromServer) throws IOException {
    Message response = TestAddBehavior.processResponse(fromServer);
    String sval = response.contents.getAttributes().getNamedItem("success").getNodeValu
e();
    assertFalse (Boolean.valueOf(sval));
    String reason = response.contents.getAttributes().getNamedItem("reason").getNodeVal
ue();
    assertEquals (expect, reason);
    return response;
  }
```

Now you're ready to tackle rewriting the **testBasicAddBehavior** test case method:

CODE TO TYPE: /test/server.ipc/TestAddBehavior.java

```
  public void testBasicAddBehavior() throws Exception {
    Message r;
~~    requestSIZE();~~
~~    expectSuccess("0", fromServer);~~
    requestSTATUS(toServer);
    r = expectSuccess(fromServer);
    String count = r.contents.getFirstChild().getAttributes().getNamedItem("total").get
NodeValue();
    assertEquals ("0", count);

    File f = new File("images", "repositorySplash.png");
    requestADD(toServer, "sampleImage", f);
    expectSuccess(~~null, ~~fromServer);

~~    requestSIZE(); ~~
~~    expectSuccess("1", fromServer);~~
    requestSTATUS(toServer);  // expect one file in repository.
    r = expectSuccess(fromServer);
    count = r.contents.getFirstChild().getAttributes().getNamedItem("total").getNodeVal
ue();
    assertEquals ("1", count);

    stopClient();
    stopServer();

    server = TestServer.launchServer();
    startClient();

~~    requestSIZE();~~
~~    expectSuccess("1", fromServer);~~
    requestSTATUS(toServer);
    r = expectSuccess(fromServer);
    count = r.contents.getFirstChild().getAttributes().getNamedItem("total").getNodeVal
ue();
    assertEquals ("1", count);

    requestADD(toServer, "sampleImage", f);
    expectFailure(Repository.AlreadyExistsImage, fromServer);
  }
}
```

Execute EclEmma on all test cases in the **test** source folder. If you have kept the test cases up-to-date as each lab progressed, then you'll see a table that aggregates coverage by package:

Package	Coverage	Covered Instructions	Total Instructions
client	9.6 %	12	125
client.gui	0.0 %	0	135
server	81.0%	149	184
server.ipc	87.7 %	100	114
server.model	55.2 %	203	368
util	87.8 %	195	222
xml	85.9 %	177	206

Aside from the code that shows 0.0% coverage, you have produced 72% coverage of the **src** source folder. Expand the packages in the EclEmma Coverage Report and you'll see that some classes have no coverage (and likely will have no test cases to cover them) such as **ClientLauncher**; also **ImageRepositoryClient** isn't covered by a test class yet. Finally, inspect the coverage for the **Repository** class, because the code that didn't execute is contained only within Exception handlers. Instead of trying to reach 80% coverage in this class, you might want to construct sample test cases that deal with specific situations (for example, if the persistent files for the repository are READ ONLY when executing the program).

User Authentication

Lesson Objectives

In this lesson you will:

- store user state on the server side of a client-server application.

User Authentication

Now that you have the communication framework for a client server application, it's time to upgrade the server so that it's aware of each individual user. Currently the server is multi-threaded to allow multiple clients to upload images, but there is no way for the server to differentiate between these clients. The common solution is to provide user accounts with credentialed information that must be provided at connect time. In this lesson, you'll extend the communication protocol to require an initial login message; thereafter, the server will associate that user with the thread spawned to process the client requests. You also need to ensure that two clients are unable to connect simultaneously to the same server using the same credentials; this will require a user manager to oversee which users are currently logged in.

Let's start with the **loginRequest** that you need to add to the protocol. Here's a sample XML fragment of a valid **loginRequest**:

OBSERVE:

```
<request>
  <loginRequest user='user00' password='6e5aa8fe26c43b164d6308b0b942deb2'/>
</request>
```

The password will never be sent as plain text; we'll send an MD5-fingerprint of the actual password. Begin your code by defining several new test case methods in **ValidateXMLMessages**. Don't worry about typing in the "exact" hexadecimal string representing the hashed password! Just make sure it's a non-empty string:

CODE TO TYPE: /test/xml/ValidateXMLMessages.java

```
...
  public void testLoginRequest() {
    String login = "<request><loginRequest user='user00' password='6e5aa8fe26c43b164d63
08b0b942deb2'/></request>";
    Message m = new Message(login);
    assertEquals ("request", m.contents.getLocalName());
  }

  public void testLoginResponseFailure() {
    String login = "<response success='false' reason='Invalid credentials'><loginRespon
se user='user00'/></response>";
    Message m = new Message(login);
    assertEquals ("response", m.contents.getLocalName());
  }

  public void testLoginResponseSuccess() {
    String login = "<response success='true'><loginResponse user='user00'/></response>"
;
    Message m = new Message(login);
    assertEquals ("response", m.contents.getLocalName());
  }
...
```

Naturally, these test cases won't pass yet, because you have to modify the **repository.xsd** file to include definitions for these messages. These changes will do the trick:

CODE TO TYPE: /repository.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>

<xs:element name='message'>
  <xs:complexType>
    <xs:choice>
      <xs:element ref='response'/>
      <xs:element ref='request'/>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name='response'>
  <xs:complexType>
    <xs:choice>
      <xs:element ref='addResponse'/>
      <xs:element ref='statusResponse'/>
      <xs:element ref='loginResponse'/>
    </xs:choice>
    <xs:attribute name='success' type='xs:boolean' use='required'/>
    <xs:attribute name='reason'  type='xs:string'  use='optional'/>
  </xs:complexType>
</xs:element>

<xs:element name='request'>
  <xs:complexType>
    <xs:choice>
      <xs:element ref='addRequest'/>
      <xs:element ref='statusRequest'/>
      <xs:element ref='loginRequest'/>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name='addRequest'>
  <xs:complexType>
   <xs:sequence>
     <xs:element name='image'/>
   </xs:sequence>
   <xs:attribute name='name' type='xs:string' use='required'/>
  </xs:complexType>
</xs:element>

<xs:element name='addResponse'>
  <xs:complexType>
    <xs:attribute name='numBytes' type='xs:integer' use='required'/>
  </xs:complexType>
</xs:element>

<xs:element name='statusRequest'/>

<xs:element name='statusResponse'>
  <xs:complexType>
    <xs:attribute name='key'   type='xs:string'  use='required'/>
    <xs:attribute name='index' type='xs:integer' use='required'/>
    <xs:attribute name='total' type='xs:integer' use='required'/>
  </xs:complexType>
</xs:element>

<xs:element name='loginRequest'>
  <xs:complexType>
   <xs:attribute name='user'     type='xs:string' use='required'/>
   <xs:attribute name='password' type='xs:string' use='required'/>
  </xs:complexType>
</xs:element>
```

```
<xs:element name='loginResponse'>
  <xs:complexType>
    <xs:attribute name='user' type='xs:string' use='required'/>
  </xs:complexType>
</xs:element>
</xs:schema>
```

The above changes define the structure of **loginRequest** and **loginResponse** messages. Note that all of the attributes are required. When you edit the XSD file, try to maintain the structure such that requests and responses are listed in pairs; this will make it easier for you to find elements in the file in the future. Before continuing, ensure that the **validateXMLMessages** test case passes.

Now you have to make a fundamental decision regarding **loginRequest**. Specifically, there can be only a single **loginRequest** message during a client session, but it should not be handled by the **ProtocolHandler** class that you've been building. Do you see why? The authentication of a user is part of the server's responsibility, whereas **ProtocolHandler** manages all actions on behalf of a given user. There will be no way for a message to be sent to the server to "spoof" some other user's identification, so you need to have the **RepositoryThread** process the first **loginRequest** message fundamentally from the client; once the user is authenticated, then **ProtocolHandler** can get involved.

As you consider making this change, you'll see the misplaced implementation in **ProtocolHandler**, for assembling **Message** objects from the underlying socket. Move this logic into the **RepositoryThread** class to avoid inadvertent errors on behalf of the user logic being implemented in **ProtocolHandler**. This is a good example of a potentially dangerous leakage of the **Message** abstraction from the underlying IPC layer. Fix it by modifying the **IProtocolHandler** interface. Instead of exposing raw access to the Input/Output streams made available by the socket, this revised interface takes an XML request and returns an XML response. If this method ever returns **null**, the server should disconnect the client. Pause for a moment to digest this change— I'll wait, it's important. It's common to refactor systems to hide details using better abstractions that are discovered over the course of a project's lifetime. Modify **IProtocolHandler** as shown:

CODE TO TYPE: /src/server.ipc/IProtocolHandler

```
package server.ipc;

~~import java.io.*;~~
import xml.*;

public interface IProtocolHandler {

  ~~/** Process the protocol using socket's input and output. Return false to terminate,~~
~~true to continue. */~~
~~  boolean process(BufferedReader fromSocket, PrintWriter toSocket);~~
  /** Process the given Message request, return Message in response or null to terminat
e protocol. */
  Message process(Message request);
}
```

Once you have made this change to the interface, you can delete the **process(BufferedReader, PrintWriter)** method in **ProtocolHandler**

/src/server/ProtocolHandler.java

```
...
public class ProtocolHandler implements IProtocolHandler {
  ...

  ~~public boolean process(BufferedReader fromSocket, PrintWriter toSocket) {~~
~~      ...~~
~~        return false;~~
~~      }~~
~~  }~~

  ...
}
```

Several of our classes contain code logic for parsing or concatenating XML strings. Instead of spreading this logic around, consolidate the logic into a single class.

In the **src** folder **xml** package, create a **Parser** class as shown:

CODE TO TYPE: /src/xml/Parser.java

```
package xml;

import java.io.*;

public class Parser {
  public final static String loginRequest        = "loginRequest";
  public final static String loginResponse       = "loginResponse";

  public final static String loginUser           = "user";
  public final static String loginPassword       = "password";

  public final static String invalidCredentials  = "Invalid credentials";

  public static Message extractRequest(BufferedReader in) {
    return extractMessage(in, "</request>");
  }

  public static Message extractResponse(BufferedReader in) {
    return extractMessage(in, "</response>");
  }

  static Message extractMessage(BufferedReader in, String terminator) {
    try {
      String line = in.readLine();
      if (line == null) { return null; }
      StringBuilder buf = new StringBuilder(line);
      while (!buf.substring(buf.length() - terminator.length(), buf.length()).equals(te
rminator)) {
        line = in.readLine();
        if (line == null) { return null; }
        buf.append(line);
      }

      return new Message (buf.toString());
    } catch (IOException ioe) {
      return null;
    }
  }
}
```

This class processes input streams to extract the request and response XML messages. Over time it can store an increasing number of string constants associated with the **repository.xsd** schema. For now, the string constants in **Parser** reflect the attributes of the **loginRequest** and **loginResponse** messages:

OBSERVE:

```
public class Parser {
  public final static String loginRequest        = "loginRequest";
  public final static String loginResponse       = "loginResponse";

  public final static String loginUser           = "user";
  public final static String loginPassword       = "password";

  public final static String invalidCredentials  = "Invalid credentials";

  public static Message extractRequest(BufferedReader in) {
    return extractMessage(in, "</request>");
  }

  public static Message extractResponse(BufferedReader in) {
    return extractMessage(in, "</response>");
  }

  static Message extractMessage(BufferedReader in, String terminator) {
    ...
  }
}
```

So far you've seen the **extractMessage()** method in several test cases and actual code. This helper method is used by **extractRequest** and **extractResponse**. Once again, this method succeeds only if the terminator string (in this case either **<request>** or **<response>**) has an end-of-line ('/n') character immediately following it.

To take advantage of this consolidated code, change the **RepositoryThread** class. Let's do that in stages. First, replace the **while** loop in the **run** method:

CODE TO TYPE: /src/server.ipc/RepositoryThread.java

```
package server.ipc;

import java.io.*;
import java.net.*;
import xml.*;
import org.w3c.dom.*;

public class RepositoryThread extends Thread {
  Socket client;
  BufferedReader fromClient;
  PrintWriter toClient;
  IProtocolHandler handler;

  RepositoryThread (Socket s, IProtocolHandler h) throws IOException {
    fromClient = new BufferedReader(new InputStreamReader(s.getInputStream()));
    toClient = new PrintWriter (s.getOutputStream(), true);
    client = s;
    handler = h;
  }

  public void run() {
    ~~// have handler manage the protocol until it decides it is done.~~
~~    while (handler.process(fromClient, toClient)) {~~

~~    }~~

    // authentication by first login message. Stop if not a loginRequest.
    Message m = Parser.extractRequest(fromClient);
    Node child = m.contents.getFirstChild();
    if (!child.getLocalName().equals (Parser.loginRequest)) {
      return;
    }

    // Get authentication information
    String user = child.getAttributes().getNamedItem(Parser.loginUser).getNodeValue();
    String pass = child.getAttributes().getNamedItem(Parser.loginPassword).getNodeValue
();

    // tell client decision and engage handler on successful login
    boolean validated;
    if (!authenticate(user, pass)) {
      m = new Message("<response success='false' reason='" + Parser.invalidCredentials
+ "'>" +
                "<loginResponse user='" + user + "'/></response>");
      validated = false;
    } else {
      m = new Message("<response success='true'><loginResponse user='" + user + "'/></r
esponse>");
      validated = true;
    }

    toClient.println(m.toString());
    if (toClient.checkError()) {
      return;
    }

    // TODO: Fill in processing logic

    try {
      fromClient.close();
      toClient.close();
      client.close();
    } catch (IOException ioe) {
      System.err.println("Unable to close connection:" + ioe.getMessage());
    }
  }
  // TODO: Scaffolding code to validate any user whose name starts with letter.
  public boolean authenticate(String user, String pass) {
    return Character.isLetter(user.charAt(0));
  }
}
```

Let's take a closer look at this added code:

Receiving and processing initial loginRequest method

```
  // authentication by first login message. Stop if not a loginRequest.
  Message m = Parser.extractRequest(fromClient);
  Node child = m.contents.getFirstChild();
  if (!child.getLocalName().equals (Parser.loginRequest)) {
    return;
  }

  // Get authentication information
  String user = child.getAttributes().getNamedItem(Parser.loginUser).getNodeValue();
  String pass = child.getAttributes().getNamedItem(Parser.loginPassword).getNodeValue()
;
```

Using the newly defined **extractRequest** method in **Parser**, **RepositoryThread** first retrieves the initial request from the client. Then using the **Java XML API you have already seen**, the code **determines whether the request is a loginRequest**, stopping immediately if it is not. Once the message is identified as a **loginRequest**, the **user** and **hashed password** are extracted from the request. You can assume that these values are present, otherwise the **Parser** would not have been able to construct the **loginRequest** message in the first place. Right away, you can see the benefit of using a formal schema; your code is simpler to write because you don't have to validate the structure of the data (this task is handled by the XML parsing code).

Next, the code seeks to validate the user with the server. For now, use the scaffolding **authenticate()** method you added earlier to **RepositoryThread**:

Authenticating the user with the server

```
  // tell client decision and engage handler on successful login
  boolean validated;
  if (!authenticate(user, pass)) {
    m = new Message("<response success='false' reason='" + Parser.invalidCredentials +
"'>" +
                    "<loginResponse user='" + user + "'/></response>");
    validated = false;
  } else {
    m = new Message("<response success='true'><loginResponse user='" + user + "'/></res
ponse>");
    validated = true;
  }

  toClient.println(m.toString());
  if (toClient.checkError()) {
    return;
  }
```

Our code constructs either a successful or failed **loginResponse** message, which it then sends back to the client. To complete the implementation, write the code that processes messages from the client if the authentication succeeds; this code replaces the **TODO** comment we had inserted earlier as a reminder:

CODE TO TYPE: /src/server.ipc/RepositoryThread.java

```
...
  // TODO: Fill in processing logic
  if (validated) {
    // have handler manage the protocol until it decides it is done.
    while ((m = Parser.extractRequest(fromClient)) != null) {
      Message response = handler.process(m);
      if (response == null) { break; }

      toClient.println(response.toString());
      if (toClient.checkError()) {
        break;
      }
    }
  }
...
```

This **while** loop extracts message requests from the client and gives them to the **handler** for processing. If **handler.process()** returns **null**, the client's session ends; otherwise the response is sent back to the client and the loop continues.

With the revised code in place, it's time to write test cases to validate that new code. Write test cases for the **Parser** class, which has static methods that pull data from a BufferedReader object. You can construct a BufferedReader object from a fixed **String** by using the StringReader class as shown in this test case:

In the **/test** folder **xml** package, create a **TestParser** test case as shown:

CODE TO TYPE: /test/xml/TestParser.java

```
package xml;

import java.io.*;
import junit.framework.TestCase;

public class TestParser extends TestCase {

  public void testParser() {
    String s = "<response success='false' reason='" + Parser.invalidCredentials + "'><l
oginResponse user='user00'/></response>";
    StringReader sr = new StringReader(s);
    Message m = Parser.extractResponse(new BufferedReader(sr));
    assertEquals ("response", m.contents.getLocalName());
  }
}
```

For this test case, assume that if the XML parsing works properly (at the outermost response level), it will parse the inner specific response properly (in this case, **loginResponse**). Generate EclEmma code coverage for this test case and look at Parser.java; you still need to write a test case to make sure that requests are being parsed similarly. Also, observe the yellow-shaded lines within **extractMessage**. These are **if** statements that execute, but the inner guarded statement does not execute. In addition, there is an Exception handler that never executes. It is really common to have these sorts of "soft" coverage. Add the following multi-line and request test case methods to **TestParser**; the test code coverage of **Parser** reaches 82%.

CODE TO TYPE: /test/xml/TestParser.java

```
  public void testRequest() {
    String s = "<request><loginRequest user='user00' password='askjdhkjhkdjs'/></reques
t>";
    StringReader serializable = new StringReader(s);
    Message m = Parser.extractRequest(new BufferedReader(serializable));
    assertEquals ("request", m.contents.getLocalName());
  }

  public void testMultiLineParser() {
    String s = "<response success='false' reason='" + Parser.invalidCredentials + "'>\n
<loginResponse user='user00'/>\n</response>";
    StringReader serializable = new StringReader(s);
    Message m = Parser.extractResponse(new BufferedReader(serializable));
    assertEquals ("response", m.contents.getLocalName());
  }
```

Be sure you test all of your parsing code, before you test the new login logic.

Now you're ready to upgrade testing code to incorporate the **loginRequest** as the first message from a client. Add **requestLOGIN()** and **responseLOGIN()** method to **TestAddBehavior** and modify **requestSTATUS()** and **requestADD()**, as shown:

CODE TO TYPE: /test/server.ipc/TestAddBehavior.java

```
  public static Message requestLOGIN(String user, String hashedPassword) {
    String s = "<request><loginRequest user='" + user + "' password='" + hashedPassword
+ "'/></request>";
    StringReader sr = new StringReader(s);
    return Parser.extractRequest(new BufferedReader(sr));
  }

  public static Message responseLOGIN(String user, String error) {
    String s = "<response success='";
    if (error == null) {
     s += "false'";
    } else {
     s += "true' reason='" + error + "'";
    }
    s += "><loginResponse user='" + user + "'/></request>";
    StringReader sr = new StringReader(s);
    return Parser.extractRequest(new BufferedReader(sr));
  }

  public static ~~void~~Message requestSTATUS (~~PrintWriter out~~) {
    String xmlStatusRequest = "<request><statusRequest/></request>";
    return new Message(xmlStatusRequest);

~~    out.println(xmlStatusRequest);~~
  }

  public static ~~void~~Message requestADD(~~PrintWriter out, ~~String name, File f) throws IOE
xception {
    String encoding = ImageEncoding.encode(f);
    String xmlAddRequest = "<request><addRequest name='" + name + "'>" +
       "<image>\n<![CDATA[" + encoding + "\n]]></image></addRequest></request>";
    return new Message(xmlAddRequest);
~~    out.println(xmlAddRequest);~~
  }
```

These methods all have the same structure in that they generate a valid **Message** object. Also, these methods are all **public static**, which maximizes their utility in other test cases. The **requestLOGIN()** and **responseLOGIN()** methods take advantage of the **StringReader** class that allows you to construct a **BufferedReader** from a fixed string, rather than from an input stream. The changes you made to **requestAD()D** and **requestSTATUS()**, eliminate the need to pass in a **PrintWriter** to the method.

Now let's g ahead and clean up the **testBasicAddBehavior()** method in **TestAddBehavior** as shown:

CODE TO TYPE: /test/server.ipc/TestAddBehavior.java

```
  public void testBasicAddBehavior() throws Exception {
    Message r;
    toServer.println(requestLOGIN("sample", "hashed-password"));
    r = expectSuccess(fromServer);
    toServer.println(requestSTATUS(~~toServer~~));
    r = expectSuccess(fromServer);
    String count = r.contents.getFirstChild().getAttributes().getNamedItem("total").get
NodeValue();
    assertEquals ("0", count);

    File f = new File ("images", "repositorySplash.png");
    toServer.println(requestADD(~~toServer, ~~"sampleImage", f));
    expectSuccess(fromServer);

    toServer.println(requestSTATUS(~~toServer~~));  // expect one file in repository.
    r = expectSuccess(fromServer);
    count = r.contents.getFirstChild().getAttributes().getNamedItem("total").getNodeVal
ue();
    assertEquals ("1", count);

    stopClient();
    stopServer();

    server = TestServer.launchServer();
    startClient();

    toServer.println(requestLOGIN("sample", "hashed-password"));
    r = expectSuccess(fromServer);
    toServer.println(requestSTATUS(~~toServer~~));
    r = expectSuccess(fromServer);
    count = r.contents.getFirstChild().getAttributes().getNamedItem("total").getNodeVal
ue();
    assertEquals ("1", count);

    // must fail because of duplicate image
    toServer.println(requestADD(~~toServer, ~~"sampleImage", f));
    expectFailure(Repository.AlreadyExistsImage, fromServer);
  }
```

These changes affect only the logic concerning the way test cases write messages to the server. The **testAddBehavior** test case method in **TestAddBehavior** is now redundant because of the extra logic being tested by the **testBasicAddBehavior** test case method, so delete the **testAddBehavior** test case method. Now, add a test case method to **TestAddBehavior** to validate login failures:

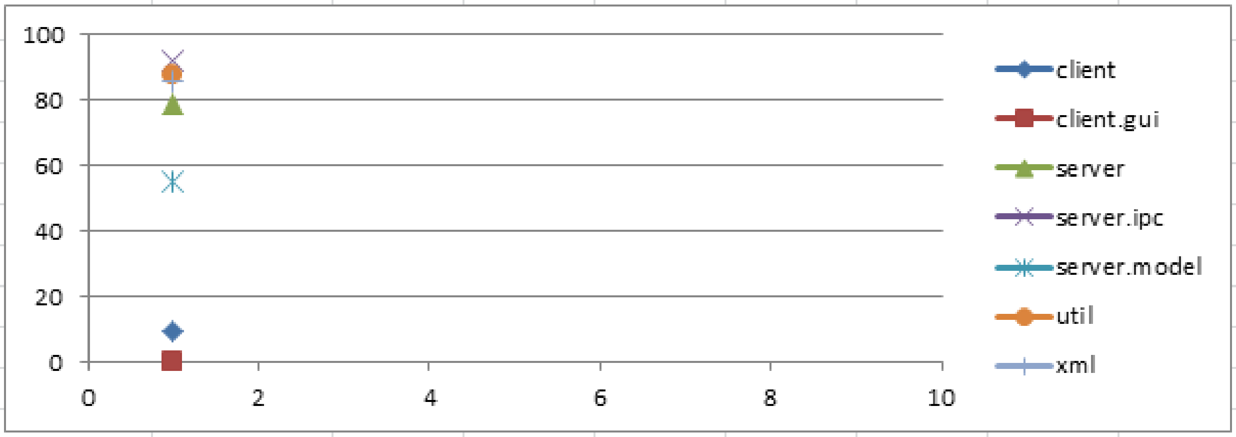
CODE TO TYPE: /test/server.ipc/TestAddBehavior.java

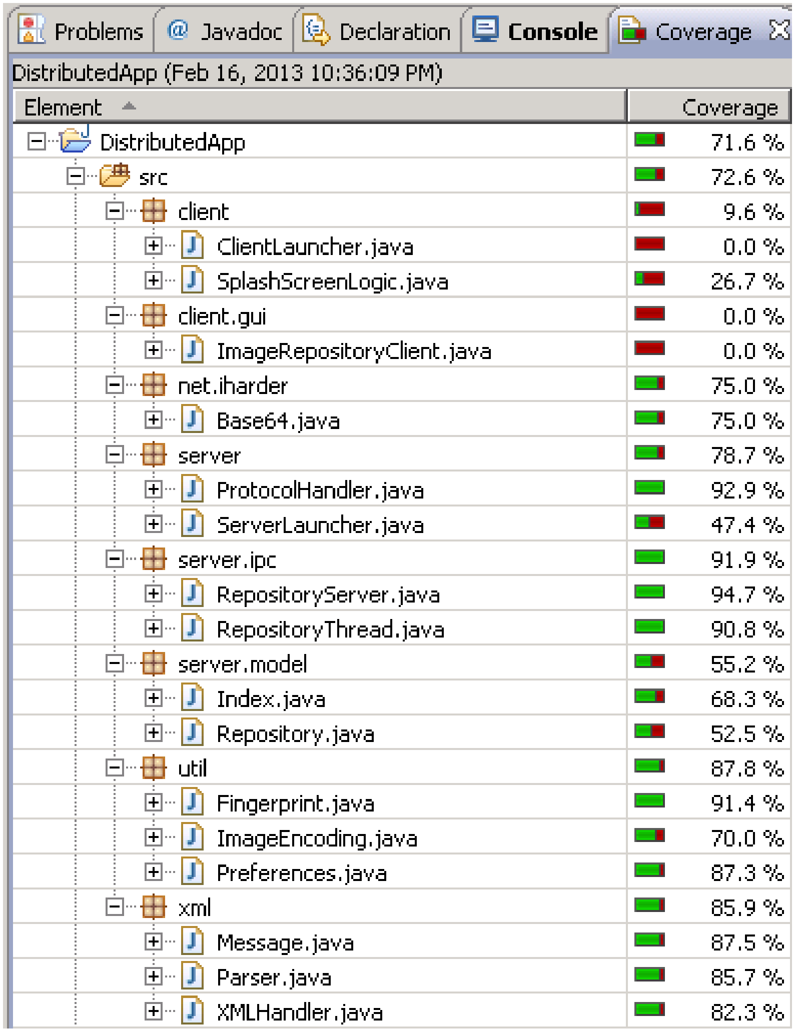
```
...
  // scaffolding denies user names starting with digit
  public void testLoginFailureBehavior() throws Exception {
    toServer.println(requestLOGIN("0startsLetter", "BADBAD"));
    expectFailure(Parser.invalidCredentials, fromServer);
  }
```

Rerun all test cases in the **test** source folder. Make it a habit to run all test cases, because you never know when a minor change causes some seemingly unrelated part of your code to fail.

Now, relaunch all test cases in the **test** source folder using EclEmma to generate code coverage. In the Coverage panel at the bottom of the Eclipse window, expand each of the packages in the **src** folder to check your progress. From this starting point, create a spreadsheet to chart the increase (or decrease!) of coverage at the end of each lecture. These charts can be extremely useful as you determine which new test cases you need to write. Here's an image of the spreadsheet as it is right now. Over the next few labs, you will see percentages go down (because new code has been added) or go up (because new test cases have been written):

	Authenticate Users	Server Sessions	Client Login	Server Menu	Browse Images	Navigate Image	Delete Image
src	72.6						
client	9.6						
client.gui	0						
server	78.7						
server.ipc	91.9						
server.model	55.2						
util	87.8						
xml	85.9						





Problems | Javadoc | Declaration | Console | Coverage

DistributedApp (Feb 16, 2013 10:36:09 PM)

Element	Coverage
DistributedApp	71.6 %
src	72.6 %
client	9.6 %
ClientLauncher.java	0.0 %
SplashScreenLogic.java	26.7 %
client.gui	0.0 %
ImageRepositoryClient.java	0.0 %
net.iharder	75.0 %
Base64.java	75.0 %
server	78.7 %
ProtocolHandler.java	92.9 %
ServerLauncher.java	47.4 %
server.ipc	91.9 %
RepositoryServer.java	94.7 %
RepositoryThread.java	90.8 %
server.model	55.2 %
Index.java	68.3 %
Repository.java	52.5 %
util	87.8 %
Fingerprint.java	91.4 %
ImageEncoding.java	70.0 %
Preferences.java	87.3 %
xml	85.9 %
Message.java	87.5 %
Parser.java	85.7 %
XMLHandler.java	82.3 %

Nicely done! You now have a client that can authenticate its connection with the server and communicate through XML messages. You're well-positioned to enhance the communication protocol with new messages, and add the final functionality required for this application. In the next lab, you'll enhance the server to maintain information about active users (users who are connected to the server). See you there!

Server Sessions

Lesson Objectives

In this lesson you will:

- use the server to associate user information with each spawned thread to manage the proper authentication and execution of the protocol.

Server Sessions

So now you have a working protocol that lets clients connect to the server, but the server still doesn't maintain user-defined sessions for each thread to authenticate users' actions properly in the protocol. You also have scaffolding code in the server-side authentication that needs to be replaced to store and validate users' account information. In this lab, you'll use the Java Serializable mechanism to store user information persistently in a file. Let's get started!

You'll need to design a **UserManager** class to manage and store all user information; it will allow the server to create self-registered accounts. In its constructor, **UserManager** takes as an argument, a java.io.File, in which all persistent information is stored. You've already seen how to store whole objects to disk. In this lesson, you'll store the user manager object itself.

Be aware that user accounts must be deleted automatically after a specified period of inactivity. One way to meet this requirement would be to actively monitor the *last activity time* for any user, and proactively take steps to delete accounts where "time of inactivity" exceeds the server's threshold. We start by storing extra time information for each user. At startup time, the server can discontinue the account for any user that has shown no activity for a specified time period. The only drawback to this approach is that users may log in and stay logged in without any subsequent activity, and thereby retain their accounts for longer than the allowed period of time. This situation can be resolved by including a separate active "reaper" thread that sweeps through the activity of each user and disconnects (and deactivates the accounts of) users who have been inactive for too long.

You can analyze the requirements to determine that for each user you need to store (a) the user id; (b) the hashed password; and (c) the time of last activity. To write and parse time values, choose to store this value in the standard "milliseconds since January 1 1970" format.

Note Although Java supports using "milliseconds since January 1 1970" to store time, you won't encounter the Unix Millenium Bug because it's stored in a 64-bit **long**. That's good to know!

In the **/src** folder **server** package, create the **UserInfo** class as shown:

CODE TO TYPE: /src/server/UserInfo.java

```
package server;

public class UserInfo implements java.io.Serializable {
  final String user;
  final String hashedPassword;
  long  lastAccessTime;

  public UserInfo(String user, String hashedPassword, long access) {
    this.user = user;
    this.hashedPassword = hashedPassword;
    this.lastAccessTime = access;
  }

  public boolean authenticate(String hp) {
    return (hashedPassword.equals(hp));
  }

  public void updateAccessTime(long millis) {
    lastAccessTime = millis;
  }
}
```

This class stores a hashed password and a time of last activity for each user, and supports authentication and updates. Because there is no way to alter a user's account id or hashed password, these attributes are marked as **final**. Make sure **UserInfo** implements **java.io.Serializable**, because you'll need to store instances of this object to disk.

The **UserManager** supports four key capabilities. Start with just two, for now: register new users on demand, and authenticate existing users. We'll implement this class in stages:

In the **/src** folder **server** package, create the **UserManager** class as shown:

CODE TO TYPE: /src/server/UserManager.java

```
package server;

import java.io.*;
import java.util.*;

public class UserManager implements Serializable {
  Hashtable<String,UserInfo> users = new Hashtable<String,UserInfo>();
  transient File storage;

  public UserManager (File f) {
    storage = f;
  }

  public boolean registerUser (String user, String hashedPassword) {
    if (users.containsKey(user)) { return false; }

    UserInfo ui = new UserInfo (user, hashedPassword, System.currentTimeMillis());
    users.put(user, ui);
    return true;
  }

  public boolean removeUser(String user) {
    UserInfo ui = users.remove(user);
    return (ui != null);
  }

  public boolean authenticate (String user, String hashedPassword) {
    UserInfo ui = users.get(user);
    if (ui == null) { return false; }

    return ui.authenticate(hashedPassword);
  }
}
```

OBSERVE:

```
import java.io.*;
import java.util.*;

public class UserManager implements Serializable {
  Hashtable<String,UserInfo> users = new Hashtable<String,UserInfo>();
  transient File storage;

  public UserManager (File f) {
    storage = f;
  }
```

UserManager is instantiated with the specified **java.io.File** into which it should store (and from which it should load) persistent information. By ensuring the constructor requires the **File** object for persistent storage, the class ensures that it can ultimately provide **load()** and **store()** methods requiring no parameters, which can thus be invoked directly by the **RepositoryThread** code. Note that the **storage** attribute is marked **transient**. This tells the Java VM not to write this value to disk during the serialization process. Now, add the **store()** method to **UserManager**:

CODE TO TYPE: /src/server/UserManager.java

```
  public boolean store() {
    FileOutputStream fos;
    try {
      fos = new FileOutputStream(storage);
    } catch (FileNotFoundException fnfe) {
      System.err.println("Unable to store user manager to:" + storage);
      return false;
    }

    ObjectOutputStream oos = null;
    try {
      oos = new ObjectOutputStream(fos);
      oos.writeObject(this);
    } catch (IOException ioe) {
      System.err.println("Errors encountered while storing user manager to:" + storage)
;
      return false;
    } finally {
      try {
        oos.close();
      } catch (Exception e) {
        System.err.println("Errors encountered while closing user manager file.");
      }
    }

    return true;
  }
```

OBSERVE:

```
      oos = new ObjectOutputStream(fos);
      oos.writeObject(this);
```

The **store()** method uses FileOutputStream to store data to a file. Using ObjectOutputStream, the **entire UserManager object is stored to disk**. Because **storage** was marked as transient, its value is not stored. The reason for doing this will be clear when you review and add the **load()** method:

CODE TO TYPE: /src/server/UserManager.java

```
  public void load() {
    if (storage.exists()) {
      FileInputStream fis;
      try {
        fis = new FileInputStream(storage);
      } catch (FileNotFoundException fnfe) {
        users = new Hashtable<String,UserInfo>();
        return;
      }

      ObjectInputStream ois = null;
      try {
        ois = new ObjectInputStream(fis);
        UserManager stored = (UserManager) ois.readObject();
        users = stored.users;
      } catch (Exception e) {
        System.err.println("Problems encountered in loading user manager file (" + stor
age + ").");
      } finally {
        try {
          ois.close();
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    }
  }
```

Because this method is invoked on an instantiated **UserManager** object, you only have to retrieve the **users** hashtable from that stored object to update the set of users for the instantiated **UserManager**. Eventually the server has to be changed to load the **UserManager** object at startup and update the user manager's information during processing (which includes both self-registered new accounts and access time information).

Now let's tackle two more capabilities: update the last access time for a given user, and determine whether a user account is still active based on the length of inactivity. Modify **UserManager** as shown:

CODE TO TYPE: /src/server/UserManager.java

```
public class UserManager {
  ...

  public static long activeThreshold = 14 * 24 * 60 * 60 * 1000;    // Active threshold
 (in milliseconds) is 14 days by default

  public static void setThreshold(long val) {
    activeThreshold = val;
  }

  public void updateAccessTime(String user) {
    UserInfo ui = users.get(user);
    if (ui != null) { ui.updateAccessTime(System.currentTimeMillis()); }
  }

  public boolean isActive (String user) {
    UserInfo ui = users.get(user);
    if (ui == null) { return false; }

    long now = System.currentTimeMillis();
    Long then = ui.lastAccessTime;

    return (now - then) < activeThreshold;
  }

  ...
}
```

OBSERVE:

```
public void updateAccessTime(String user) {
  UserInfo ui = users.get(user);
  if (ui != null) { ui.updateAccessTime(System.currentTimeMillis()); }
}

public boolean isActive (String user) {
  UserInfo ui = users.get(user);
  if (ui == null) { return false; }

  long now = System.currentTimeMillis();
  Long then = ui.lastAccessTime;

  return (now - then) < activeThreshold;
}
```

Whenever a user interacts with the server, the **RepositoryThread** processing that activity calls **updateAccessTime** to update the last activity time for that user. It works as a kind of expiration time. From the initial requirement **R10** you must be able to configure the threshold of time that an account is considered active (the default is 14 days). The default calculation of the **activeThreshold** field and the corresponding **setThreshold()** method for changing this value at runtime are in **UserManager**.

When **updateAccessTime** updates the last access time for a user, **isActive** is able to determine whether the account for a given user should be considered active given the time that has elapsed since that user's last access.

Tip In the **isActive** method, you'll see what at first seems like a weird computation: **(now - then)**, where **now** is a primitive **long**, while **then** is an object of class **Long**. Since Java version 1.5, the JavaVM "unboxes" and "boxes" mixed primitives and objects in expressions automatically.

Now we can include the **UserManager** in your **RepositoryServer**. Modify **ServerLauncher** to instantiate a **UserManager** object on startup, using a predefined file for persistent storage, which will be stored in the **Repository** directory. The **UserManager** object is passed along to the server so the server can give it to the **RepositoryThread** when it executes:

CODE TO TYPE: /src/server/ServerLauncher.java

```
package server;

import server.ipc.*;
import server.model.*;
import java.io.*;

public class ServerLauncher {
  final static String defaultLocation = "Repository";

~~  public static RepositoryServer create() throws Exception {~~
~~      return create(new File (defaultLocation));~~
~~  }~~
~~  ~~
~~  public static RepositoryServer create(File dir) throws Exception {~~
~~    Repository repository = new Repository(dir);~~
~~    RepositoryServer server = new RepositoryServer(repository, new ProtocolHandler(repo~~
~~sitory));~~
~~    server.bind();~~
~~    return server;~~
~~  }~~

  public static void main(String[] args) throws Exception {
    File storage = new File (defaultLocation);
    if (args.length != 0) {
   storage = new File (args[0]);
    }

    UserManager userManager = new UserManager(new File (storage, "userManager"));
    userManager.load();

    Repository repository = new Repository(storage);
    RepositoryServer server = ~~create();~~new RepositoryServer(repository, userManager,
     new ProtocolHandler(repository));

    server.bind();

    // process all requests and exit.
    System.out.println("Server awaiting client connections");
    server.process();
    System.out.println("Server shutting down.");
  }
}
```

For now, ignore the compilation error in **ServerLauncher** (it will be fixed shortly).

Let's work on the self-registration issue. Edit the **repository.xsd** file to add a new optional boolean **register** attribute to the **loginRequest** message. Java presumes that messages with the **register** attribute set to **true** are attempts by the client to register a new account in the system. The user requesting the registration may have selected a duplicate user id; these requests will be denied and the user will be prompted to choose a new user id. Find the XML block for **loginRequest** in **repository.xsd** and modify it as shown:

CODE TO TYPE: /repository.xsd

```
<xs:element name='loginRequest'>
  <xs:complexType>
   <xs:attribute name='user'       type='xs:string'  use='required'/>
   <xs:attribute name='password'   type='xs:string'  use='required'/>
   <xs:attribute name='register'   type='xs:boolean' use='optional'/>
  </xs:complexType>
</xs:element>
```

When a **RepositoryThread** receives a **loginRequest** with **register='true'**, it is directed to open a new account for that user. If the account has already been opened, a failed **loginResponse** is returned to the client; otherwise the account is created and the user remains connected. Since we have modified the protocol, you should add a new attribute to **Parser** just after the definition of **loginPassword**:

CODE TO TYPE: /src/xml/Parser.java

```
...
  public final static String loginUser           = "user";
  public final static String loginPassword       = "password";
  public final static String loginRegister       = "register";

  public final static String invalidCredentials  = "Invalid credentials";
...
```

You'll need to make several changes to the **RepositoryThread** process method to manage the protocol, and extract information from the **loginRequest** message. Modify **RepositoryThread** as shown:

CODE TO TYPE: /src/server.ipc/RepositoryThread.java

```
package server.ipc;

import java.io.*;
import java.net.*;
import server.*;
import xml.*;
import org.w3c.dom.*;

public class RepositoryThread extends Thread {
  Socket client;
  BufferedReader fromClient;
  PrintWriter toClient;
  IProtocolHandler handler;
  String user;
  UserManager manager;

  RepositoryThread (UserManager um, Socket s, IProtocolHandler h) throws IOException {
    fromClient = new BufferedReader(new InputStreamReader(s.getInputStream()));
    toClient = new PrintWriter (s.getOutputStream(), true);
    client = s;
    handler = h;
    manager = um;
  }

  public void run() {
    // authentication by first login message. Stop if not a loginRequest.
    Message m = Parser.extractRequest(fromClient);
    Node child = m.contents.getFirstChild();
    if (!child.getLocalName().equals (Parser.loginRequest)) {
      return;
    }

    // Get authentication information
    ~~String ~~user = child.getAttributes().getNamedItem(Parser.loginUser).getNodeValue();
    String pass = child.getAttributes().getNamedItem(Parser.loginPassword).getNodeValue
();

    // might be self-registration.
    Node registerNode = child.getAttributes().getNamedItem(Parser.loginRegister);
    boolean register = false;
    if (registerNode != null) {
      register = Boolean.valueOf(registerNode.getNodeValue());
    }

    // tell client decision and engage handler on successful login
    boolean validated;
    if (register) {
      if (manager.registerUser(user, pass)) {
        m = new Message("<response success='true'><loginResponse user='" + user + "'/><
/response>");
        validated = true;
      } else {
        m = new Message("<response success='false' reason='" + Parser.invalidCredential
s + "'>" +
              "<loginResponse user='" + user + "'/></response>");
        validated = false;
      }
    } else {
      if (!manager.authenticate(user, pass)) {
        m = new Message("<response success='false' reason='" + Parser.invalidCredential
s + "'>" +
                  "<loginResponse user='" + user + "'/></response>");
        validated = false;
      } else {
        m = new Message("<response success='true'>" +
              "<loginResponse user='" + user + "'/></response>");
```

```
          validated = true;
        }
      }

      toClient.println(m.toString());
      if (toClient.checkError()) {
        return;
      }

      if (validated) {
        // have handler manage the protocol until it decides it is done.
        while ((m = Parser.extractRequest(fromClient)) != null) {
          manager.updateAccessTime(user);
          Message response = handler.process(m);
          if (response == null) { break; }

          toClient.println(response.toString());
          if (toClient.checkError()) {
            break;
          }
        }
      }

      // close communication to client.
      try {
        fromClient.close();
        toClient.close();
        client.close();
      } catch (IOException e) {
        System.err.println("Unable to close connection:" + e.getMessage());
      }
    }

~~    // TODO: Scaffolding code to validate any user whose name starts with letter.~~
~~    public boolean authenticate(String user, String pass) {~~
~~      return Character.isLetter(user.charAt(0));~~
~~    }~~
}
```

The new code we added handles the self-registration of user accounts as requested at the outset of this project. It also handles situations where a user tries to self-register an account with a user name that already exists.

In the **run** method of **RepositoryThread**, you've converted the local variable **user** extracted from the **loginRequest** into a class attribute. Because the user string is stored by the **RepositoryThread**, it is not available to the **ProtocolHandler** and therefore cannot be "spoofed" by malicious code. Also, in newly inserted call to **manager.updateAccessTime()** within the **while** loop, you can see that whenever any activity occurs for the user, the thread updates the activity managed by the **UserManager** first.

The scaffolding **authenticate()** method in **RepositoryThread** has been deleted and invocations to it have been replaced with invocations to the **UserManager** implementation. We write scaffolding code to enable development to proceed at a steady, uninterrupted pace. We delete scaffolding code once the real classes are developed.

Now modify **RepositoryServer** so it is given a **UserManager** object when it is constructed; this **UserManager** object is passed to each thread spawned by the **RepositoryServer**:

CODE TO TYPE: /src/server.ipc/RepositoryServer.java

```
package server.ipc;

import java.io.*;
import java.net.*;
import server.model.*;
import server.*;

public class RepositoryServer {
  ServerSocket serverSocket = null;
  int state = 0;
  IProtocolHandler protocolHandler;
  Repository repository;
  UserManager manager;

  public RepositoryServer(Repository rep, UserManager um, IProtocolHandler ph) {
    protocolHandler = ph;
    repository = rep;
    manager = um;
  }

  public void bind() throws IOException {
    serverSocket = new ServerSocket(9172);
    state = 1;
  }

  public void process() throws IOException {
    while (state == 1) {
      Socket client = serverSocket.accept();

      new RepositoryThread(manager, client, protocolHandler).start();
    }

    shutdown();
  }

  void shutdown() throws IOException {
    if (serverSocket != null) {
      serverSocket.close();
      serverSocket = null;
      state = 0;
    }
  }
}
```

Testing

Your code changes cause compilation errors in the test cases, so now you need to integrate the **UserManager** class. The **ConcurrentUserPerformance** class in the **performance** source folder is seriously outdated (it still refers to SIZE messages). This class has served its purpose; it's time to delete it. Review the **TestServer** test case that you need to modify. Change the imports of this test case to import **server.*** and then modify the **launchServer** method as shown below to integrate **UserManager** into the **launchServer** method used during testing:

CODE TO TYPE: /test/server.ipc/TestServer.java

```
package server.ipc;

import java.io.*;
import server.model.*;
~~import server.ServerLauncher;~~
~~import server.ipc.RepositoryServer;~~
import server.*;
import client.*;
import junit.framework.TestCase;

public class TestServer extends TestCase {

  ...

  public static RepositoryServer launchServer() throws Exception {
    ~~final RepositoryServer server = ServerLauncher.create(new File (testRepository));~~
    ~~assertEquals (1, server.state);~~
    Repository repository = new Repository(new File(testRepository));
    UserManager userManager = new UserManager(new File (testRepository, "userManager"))
;
    userManager.load();
    final RepositoryServer server = new RepositoryServer(repository, userManager,
      new server.ProtocolHandler(repository));
    new Thread() {
      public void run()  {
        try {
          server.bind();
          assertEquals (1, server.state);
          server.process();
        } catch (IOException ioe) {
          System.err.println("Server completed:" + ioe.getMessage());
        }
      }
    }.start();

    // wait until server is ready.
    Thread.sleep(2000);

    return server;
  }
}
```

You need some additional test cases to validate the core behavior of **UserManager**.

In the **/test** source folder **server** package, create a new JUnit test case named **TestUserManager** as shown (your **server** package is probably empty at this point, so it may not appear in your Package Explorer. Right-click the **/test** folder, select **New | Other | JUnit | JUnit Test Case**, and include **server** in the Package field):

CODE TO TYPE: /test/server/TestUserManager.java

```
package server;

import java.io.*;
import server.ipc.*;
import junit.framework.TestCase;

public class TestUserManager extends TestCase {
  UserManager userManager;

  protected void setUp() {
    userManager = new UserManager(new File (TestServer.testRepository, "userManager"));
  }

  public void testMembership() {
    assertFalse (userManager.isActive("george"));
    assertTrue (userManager.registerUser("george", "HASH-PASSWORD"));
    assertTrue (userManager.isActive("george"));

    assertTrue (userManager.authenticate("george", "HASH-PASSWORD"));
    assertFalse (userManager.authenticate("george", "BAD-HASH-PASSWORD"));
    assertTrue (userManager.removeUser("george"));
    assertFalse (userManager.isActive("george"));
    assertFalse (userManager.authenticate("george", "HASH-PASSWORD"));
    assertFalse (userManager.removeUser("george"));
  }

  public void testStorage() {
    assertFalse (userManager.isActive("george"));
    assertTrue (userManager.registerUser("george", "HASH-PASSWORD"));
    assertTrue (userManager.store());

    // recreate
    userManager = new UserManager(new File (TestServer.testRepository, "userManager"));
    userManager.load();
    assertTrue (userManager.isActive("george"));
  }

  public void testFaultyTwiceRegistered() {
    assertTrue (userManager.registerUser("george", "HASH-PASSWORD"));
    assertFalse (userManager.registerUser("george", "HASH-PASSWORD"));
  }

  public void testTiming () throws InterruptedException {
    assertTrue (userManager.registerUser("george", "HASH-PASSWORD"));
    UserManager.setThreshold(50);  // 50 milliseconds
    Thread.sleep(250);             // sleep longer than threshold
    assertFalse (userManager.isActive("george"));
    userManager.updateAccessTime("george");
    assertTrue (userManager.isActive("george"));
  }
```

The **testMembership()** test case method issues a sequence of registrations and authentications to validate that the core logic is covered. Each method in **UserManager** returns a meaningful return value which facilitates proper testing.

Validate that all test cases in the **test** source folder pass. What's this? Failed tests within the **TestAddBehavior** test case? Of course! In this lab, you replaced the "scaffolding" **authenticate** method (which validated solely by making sure that the first character of user name was a letter) with the real implementation. So now you have to revisit these test cases. Since you added self-registration **loginRequest** messages in this lab, you will have to validate that code as well:

CODE TO TYPE: /test/server.ipc/TestAddBehavior.java

```
...
  public static Message requestLOGIN(String user, String hashedPassword, boolean self)
{
    String s = "<request><loginRequest user='" + user + "' password='" + hashedPassword
 +
               "' register='" + self + "'/></request>";
    StringReader sr = new StringReader(s);
    return Parser.extractRequest(new BufferedReader(sr));
  }

  public static Message requestLOGIN(String user, String password) {
    return requestLOGIN(user, password, false);
  }
...
```

These methods allow test cases to request a login for an existing account or to self-register one. We keep the original **requestLOGIN()** method with two parameters for convenience and backward compatibility. You can take advantage of these new methods right away in the revised **testBasicAddBehavior()** method:

CODE TO TYPE: /test/server.ipc/TestAddBehavior.java

```
...
  public void testBasicAddBehavior() throws Exception {
    Message r;
    toServer.println(requestLOGIN("sample", "hashed-password", true));
    r = TestAddBehavior.expectSuccess(fromServer);
    ...
  }
```

Now this test case self-registers the **sample** user account, which is used later in the test case during a straight-up login process. However, when we rerun all test cases, this test case fails on the second attempt to login using these same credentials. We can explain this behavior. Earlier, you wrote **load()** and **store()** methods in **UserManager**, but you never wrote the code to invoke **store()**. You have several options; the least efficient would have you invoke **store()** whenever any user information changed (for example, when new accounts were created or the last access time for a user is updated). An alternative would be to use a timer thread to store the **UserManager** object periodically; while useful, this option would be challenging to test within a use case. The simplest option would be to invoke **store()** whenever the server shuts down. Let's do that. Modify the **shutdown()** method in **RepositoryServer** as shown:

CODE TO TYPE: /src/server.ipc/RepositoryServer.java

```
  void shutdown() throws IOException {
    manager.store();
    if (serverSocket != null) {
      serverSocket.close();
      serverSocket = null;
      state = 0;
    }
  }
```

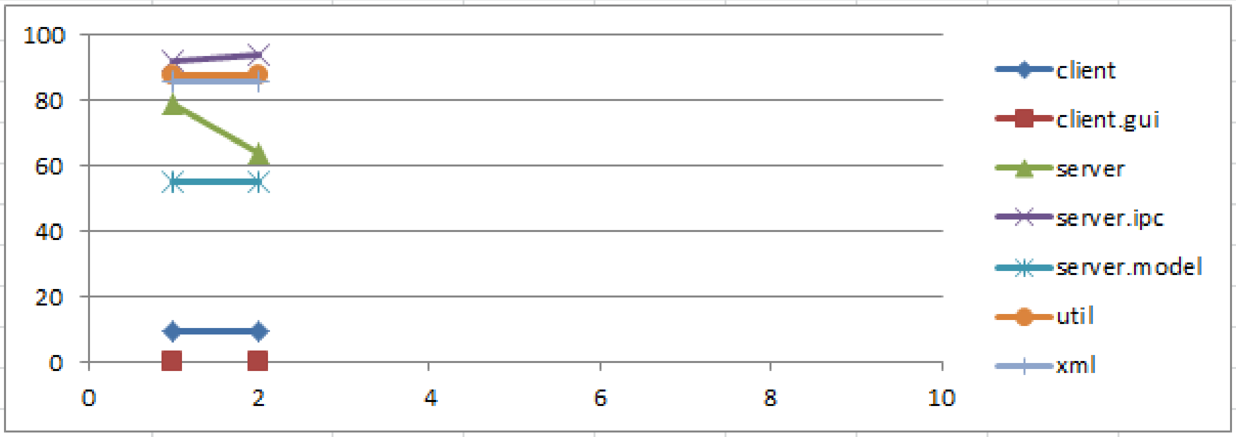
Now rerun your test cases; they all pass. Generate code coverage using EclEmma and review the new code in **RepositoryThread**. You can see that you have not exercised code when self-registration fails. What happens, for example, if someone attempts to self-register an account with a user name that already has a valid account? Add this next method to **TestAddBehavior** to handle this situation. While you're at it, fix the **testLoginFailureBehavior** test case method to eliminate its outdated documentation and misleading arguments, as shown:

CODE TO TYPE: /test/server.ipc/TestAddBehavior.java

```
~~// scaffolding denies user names starting with digit~~
public void testLoginFailureBehavior() throws Exception {
   toServer.println(requestLOGIN("~~0startsLetter~~UnknownUser", "BADBAD"));
   expectFailure(Parser.invalidCredentials, fromServer);
 }

 public void testInvalidSelfRegistration() throws Exception {
   toServer.println(requestLOGIN("user00", "n", true));
   expectSuccess(fromServer);
   stopClient();

   startClient();
   toServer.println(requestLOGIN("user00", "n", true));
   expectFailure(Parser.invalidCredentials, fromServer);
 }
```

Use EclEmma to generate the code coverage for all test cases and your results will look like this:

	Authenticate Users	Server Sessions	Client Login	Server Menu	Browse Images	Navigate Image	Delete Image
src	72.6	72.3					
client	9.6	9.6					
client.gui	0	0					
server	78.7	63.6					
server.ipc	91.9	94					
server.model	55.2	55.2					
util	87.8	87.8					
xml	85.9	85.9					



So now you know how Java deals with server sessions and identification. You've covered some complex Java topics so far. Great work!

Supporting Client Login

Lesson Objectives

In this lesson you will:

- develop the client-side IPC layer.
- write a login dialog that also supports self-registration.
- close a multi-threaded client application.

Supporting Client Login with Improved Client-Side Inter-Process Communication (IPC)

At last we're ready to revamp the client-side GUI application that will support all of the functionality you've added to the server. First, we'll improve the IPC capability of the client, essentially putting into place a multi-threaded system similar to what you did for the server. Then we'll add a login window to initiate the connection to the server. Throughout this process, your server and client will be running on the same (virtual) machine, but once the code is operational, you can execute the client and server code on separate machines.

Because you don't want to synchronously block all client-side activity while waiting for the server to process a request, the client requires an executable ServerAccess thread to read responses returned from the server. Additionally, by having a thread read responses, you make it possible for the client to receive a response asynchronously, without first sending a request.

The ServerAccess class handles the connection to the server. ServerAccess needs to know:

- the complete credentials of the user trying to connect (that is, user name and hashed password)
- the remote machine to which a connection is requested (in our case, "localhost")
- whether the user is requesting to self-register a new account

ServerAccess offers the ability to connect to the remote server, disconnect from that server, and send a request to that server. The **ServerAccess** code pulls in logic that you've seen before.

in the **/src** folder **client** package, create a **ServerAccess** class as shown:

CODE TO TYPE: /src/client/ServerAccess.java

```
package client;

import java.io.*;
import java.net.*;
import xml.*;

public class ServerAccess extends Thread {
  String host;
  String user;
  String hashedPass;
  boolean selfRegister;

  Socket server;
  BufferedReader fromServer;
  PrintWriter toServer;

  boolean isActive = false;

  public ServerAccess(String host, String user, String hashedPass, boolean selfRegister
) {
    this.host = host;
    this.user = user;
    this.hashedPass = hashedPass;
    this.selfRegister = selfRegister;
  }

  public boolean connect() {
    try {
      server = new Socket (host, 9172);
      fromServer = new BufferedReader (new InputStreamReader(server.getInputStream()));
      toServer = new PrintWriter (server.getOutputStream(), true);
      isActive = true;
    } catch (Exception e) {
      System.err.println("Unable to connect to server: " + e.getMessage());
      isActive = false;
      return false;
    }

    start();
    return true;
  }

  public void run() {
    // TODO: Fill in soon
  }

  public void disconnect() {
    isActive = false;
    try {
      server.close();
    } catch (IOException ioe) {
      System.err.println("Unable to close server:" + ioe.getMessage());
    }
  }

  public synchronized boolean sendRequest(Message r) {
    if (!isActive) { return false; }

    toServer.println(r);
    return !toServer.checkError();
  }
}
```

The **ServerAccess** constructor records information that's needed to set up communication with the remote server when **connect()** is invoked. The **isActive** field determines whether the connection to the remote server is active.

Initially, the value of isActive is **false**; a user can send a message to the server using **sendRequest()** only if the connection is active. Let's take a closer look at the **connect()** method in ServerAccess as it launches a thread:

OBSERVE:

```
  public boolean connect() {
    try {
      server = new Socket (host, 9172);
      toServer = new PrintWriter (server.getOutputStream(), true);
      fromServer = new BufferedReader (new InputStreamReader(server.getInputStream()));
      isActive = true;
    } catch (Exception e) {
      System.err.println("Unable to connect to server: " + e.getMessage());
      isActive = false;
      return false;
    }

    start();
    return true;
  }
```

You'll recognize much of this code. It **connects to the remote server and creates toServer and fromServer objects**. Once communication is established, **isActive** is set to **true**. The **start()** invocation causes the **ServerAccess** thread to begin executing its **run** method, which you'll complete now. The **run** method has two parts. Edit run() in **ServerAccess** as shown:

CODE TO TYPE: /src/client/ServerAccess.java

```
...
  public void run() {
    ~~// TODO: Fill in soon~~
    try {
      String selfAtt = "";
      if (selfRegister) { selfAtt = " register='true'"; }
      Message m = new Message("<request>" +
          "<loginRequest user='" + user + "' password='" + hashedPass + "' " + selfAtt
+ "/></request>");
      sendRequest(m);

      while (isActive) {
        // TODO: Fill in soon
      }

    } catch (Exception e) {
      e.printStackTrace();
    }

    disconnect();
  }
...
```

Let's take a closer look:

OBSERVE:

```
  public void run() {
    try {
      String selfAtt = "";
      if (selfRegister) { selfAtt = " register='true'"; }
      Message m = new Message("<request>" +
          "<loginRequest user='" + user + "' password='" + hashedPass + "' " + selfAtt
+ "/></request>");
      sendRequest(m);

      while (isActive) {
        // TODO: Fill in soon
      }

    } catch (Exception e) {
      e.printStackTrace();
    }

    disconnect();
  }
```

The **run()** method constructs a **loginRequest**, which is sent to the server first. After that, so long as **isActive** is true, the **while** loop processes messages. When **isActive** is false (or if an Exception occurs), the **ServerAccess** thread disconnects from the remote server. Now let's complete the run() method:

CODE TO TYPE: /src/client/ServerAccess.java

```
  public void run() {
    try {
      String selfAtt = "";
      if (selfRegister) { selfAtt = " register='true'"; }
      Message m = new Message("<request>" +
          "<loginRequest user='" + user + "' password='" + hashedPass + "' " + selfAtt
+ "/></request>");
      sendRequest(m);

      while (isActive) {
        ~~// TODO: Fill in soon~~
        m = Parser.extractResponse(fromServer);
        if (m == null) {
          break;
        }
        // TODO: For now, just print it to console
        System.out.println(m);
      }

    } catch (Exception e) {
      e.printStackTrace();
    }

    disconnect();
  }
```

The **ServerAccess** thread blocks and waits for a response from the server using the **extractResponse()** method implemented in the **Parser** class. **When a response is read, a Message object is constructed**. If that message object is ever **null**, the client can infer that the connection with the server has been shut down, and so it can request that **ServerAccess** be disconnected as well. For now, this code just outputs the response from the server— you'll fix in the next lesson. There's an interesting alternative situation that takes place when the client chooses to disconnect from the server. In that case, you'll just call **disconnect()** on **ServerAccess** and the thread will exit properly.

OBSERVE:

```
while (isActive) {
  m = Parser.extractResponse(fromServer);
  if (m == null) {
    break;
  }

  // TODO: For now, just print it to console
  System.out.println(m);
}
```

When designing a client/server system, you have to consider when the client terminates the communication, and when the server terminates the communication. You also need to know when a thread will stop. The Java API for threads includes the Thread.stop method, which is less than ideal. Calling **stop** on a Thread is inherently unsafe. Instead, you'll want to find indirect ways to terminate a thread. In the case of **ServerAccess**, the **while** method runs as long as the client is actively connected to the server; once the client drops this connection, **isActive** is set to **false**, and the loop terminates. The thread will terminate properly without stopping the thread manually.

Client Login Window

Our client GUI looks nice, but it has no real functionality. Let's create a login window where users can enter their credentials when connecting to the server. You'll use **GroupLayout** to model the GUI dialog window. This GUI must allow the user to enter this information:

- Remote server host (default: localhost)
- User ID
- Password
- Whether user is self-registering an account (default: no)

You also want this dialog to be "modal," which means that no other GUI processing will be possible until the dialog is closed.

In the **/src** folder **client.gui** package, create a **LoginDialog** class as shown (this is the longest code listing you've had to enter so far—fortunately, you saw a GUI Swing class earlier, so the **GroupLayout** invocations will look familiar):

CODE TO TYPE: /src/client.gui/LoginDialog.java

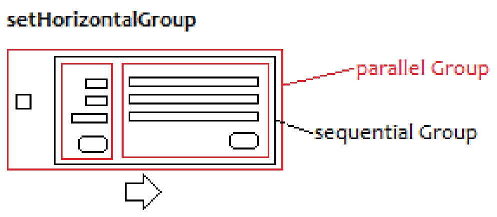
```
package client.gui;

import java.awt.event.*;
import javax.swing.*;
import javax.swing.GroupLayout.Alignment;

public class LoginDialog extends JDialog {
  JTextField      user;
  JPasswordField  pass;
  JTextField      host;
  JCheckBox       register;
  JButton         ok;
  JButton         cancel;
  boolean         isCanceled;

  public LoginDialog(JFrame parent) {
    this (parent, true);
  }

  LoginDialog(JFrame parent, boolean modal) {
    super(parent, "Enter Login Credentials", modal);
    setResizable(false);
    initLayout();
  }

  void initLayout() {
    setSize (400, 200);
    JPanel p = new JPanel();

    GroupLayout layout = new GroupLayout(p);
    p.setLayout(layout);

    // Enable gaps between components and with container for better look.
    layout.setAutoCreateGaps(true);
    layout.setAutoCreateContainerGaps(true);

    JLabel host = new JLabel ("host:");
    JLabel user = new JLabel ("user:");
    JLabel pass = new JLabel ("password:");

    layout.setHorizontalGroup(layout.createParallelGroup(Alignment.CENTER).
        addComponent(getRegisterCheckBox()).
        addGroup(layout.createSequentialGroup().
          addGroup(layout.createParallelGroup(Alignment.TRAILING).
            addComponent (host).
            addComponent (user).
            addComponent (pass).
            addComponent(getCancel())).
          addGroup(layout.createParallelGroup(Alignment.TRAILING).
            addComponent (getHostField()).
            addComponent (getUserField()).
            addComponent (getPasswordField()).
            addComponent (getOK()))));

    layout.setVerticalGroup(layout.createSequentialGroup().
        addGroup(layout.createParallelGroup(Alignment.BASELINE).
          addComponent (host).
          addComponent (getHostField())).
        addGroup(layout.createParallelGroup(Alignment.BASELINE).
          addComponent (user).
          addComponent (getUserField())).
        addGroup(layout.createParallelGroup(Alignment.BASELINE).
          addComponent (pass).
          addComponent (getPasswordField())).
        addGroup(layout.createParallelGroup(Alignment.CENTER).
          addComponent (getCancel()).
          addComponent (getRegisterCheckBox()).
          addComponent (getOK())));

    add(p);
  }

  JTextField getHostField() {
    if (host == null) { host = new JTextField (32);  }
    return host;
  }

  JTextField getUserField() {
    if (user == null) { user = new JTextField (32); }
    return user;
  }

  JPasswordField getPasswordField() {
    if (pass == null) {  pass = new JPasswordField (32);  }
    return pass;
  }

  JCheckBox getRegisterCheckBox() {
    if (register == null) {  register = new JCheckBox ("Self Register");  }
    return register;
  }

  JButton getOK() {
    if (ok == null) { ok = new JButton ("OK"); }
    return ok;
  }

  JButton getCancel() {
    if (cancel == null) { cancel = new JButton ("Cancel"); }
    return cancel;
  }
}
```

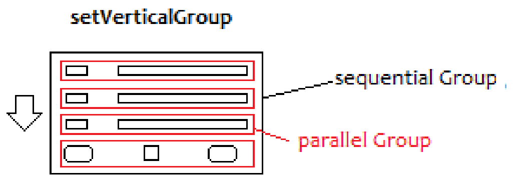
Whew! That was long one, but you probaby recognized that the structure is similar to the **ImageRepositoryClient** class you've already created. The only complication, naturally, is the invocation of **setVerticalGroup** and **setHorizontalGroup**.

Compare the layout code with this example of the actual planned layout, which shows the parallel and sequential groups:

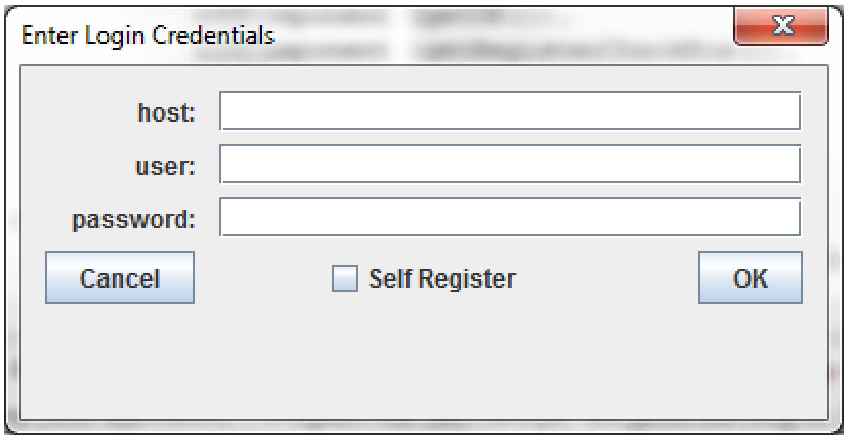
OBSERVE:

```
    layout.setHorizontalGroup(layout.createParallelGroup(Alignment.CENTER).
        addComponent(getRegisterCheckBox()).
        addGroup(layout.createSequentialGroup().
          addGroup(layout.createParallelGroup(Alignment.TRAILING).
            addComponent (host).
            addComponent (user).
            addComponent (pass).
            addComponent(getCancel())).
          addGroup(layout.createParallelGroup(Alignment.TRAILING).
            addComponent (getHostField()).
            addComponent (getUserField()).
            addComponent (getPasswordField()).
            addComponent (getOK()))));

    layout.setVerticalGroup(layout.createSequentialGroup().
        addGroup(layout.createParallelGroup(Alignment.BASELINE).
          addComponent (host).
          addComponent (getHostField())).
        addGroup(layout.createParallelGroup(Alignment.BASELINE).
          addComponent (user).
          addComponent (getUserField())).
        addGroup(layout.createParallelGroup(Alignment.BASELINE).
          addComponent (pass).
          addComponent (getPasswordField())).
        addGroup(layout.createParallelGroup(Alignment.CENTER).
          addComponent (getCancel()).
          addComponent (getRegisterCheckBox()).
          addComponent (getOK()));
```





The tricky part of this invocation is that it's able to place the "Self Register" check box between the "OK" and "Cancel" buttons. This works because the first **layout.createParallelGroup()** invocation chooses **Alignment.CENTER** for its alignment. So, the "Self Register" check box is centered horizontally along with the other widgets.



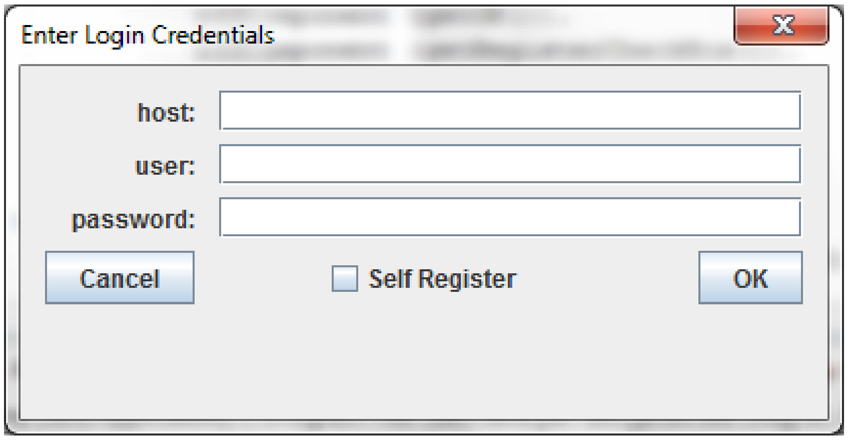
To see **LoginDialog** in action, create the **Temp** class in the default package of the **/src** folder. The **Temp** class shows how to display a **LoginDialog** window and dispose of it when the user closes the window. (You'll delete this class once you complete **LoginDialog**):

CODE TO TYPE: /src/Temp.java

```
import javax.swing.*;
import client.gui.*;

public class Temp {
  public static void main(String[] args) {
    final LoginDialog ld = new LoginDialog(null);
    ld.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    ld.setVisible(true);
    System.out.println("DONE");
  }
}
```

Normally a dialog is opened from within another Java window, but you can always pass in **null** to the constructor. In this case, the dialog appears in the top-left corner of your display. **LoginDialog** has two constructors: the **LoginDialog(JFrame)** constructor creates a modal dialog, which means that no other GUI activity is accepted by your Swing application until this dialog is closed; and the **LoginDialog(JFrame,boolean)** constructor, which gives you the option to create a modeless dialog (which will become necessary later when you write test cases for this class).

Run the **Temp** class. Your dialog appears as shown:



Unfortunately, none of the buttons work, but the top three label/field pairs are structured correctly, and the "Self Register" checkbox is centered horizontally. When you close the **LoginDialog** window you just launched with **Temp**, the word "DONE" appears on the Eclipse console. This behavior demonstrates that GUI threads block whenever a modal dialog is opened.

Users may complete their interactions with a dialog box by pressing the **OK** button; dialog boxes also present the option to **Cancel** (or the user could choose to close the entire dialog window from the window frame). There is also an **isCanceled** attribute in **LoginDialog**.

In GUI applications, you write control handlers to process events (such as mouse clicks and requests to close windows). Modify the **LoginDialog** class as shown to add the first control handler:

CODE TO TYPE: /src/client.gui/LoginDialog.java

```
public class LoginDialog extends JDialog {
  ...

  class CancelAction {
    public void process() {
      isCanceled = true;
      LoginDialog.this.dispose();
    }
  }

  public LoginDialog(JFrame parent) {
    this (parent, true);
  }

  LoginDialog(JFrame parent, boolean modal) {
    super(parent, "Enter Login Credentials", modal);
    setResizable(false);
    initLayout();

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
       new CancelAction().process();
      }
    });

    getCancel().addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
         new CancelAction().process();
      }
    });
  }

  ...
}
```

The **CancelAction** inner class contains a single **process()** method that sets the **isCanceled** attribute for **LoginDialog** to **true**, before disposing of the **LoginDialog** object. The unusual syntax of the **LoginDialog.this.dispose()** statement; this syntax allows the inner class (in this case **CancelAction**) to be able to refer to its container class (that is, **LoginDialog**). Java's use of inner classes makes it possible to write concise code that can be encapsulated to protect access. This **CancelAction** class is used by both the **WindowAdapter** controller associated with closing the window, and the **ActionListener** associated with the **Cancel** button. Because **CancelAction** is a stateless class, you can construct a new **CancelAction** object and invoke its **process()** method. In both cases, you use Java's anonymous classes to register a **WindowListener** and **ActionListener** with their respective Swing elements.

Go ahead and run the **Temp** class (you can dispose of the window by clicking on the **Cancel** button). That's progress! Let's keep going.

LoginDialog needs to have some logic so the client application can retrieve the information entered by the user. In other words, when the user fills in the text fields and clicks **OK**, you need to extract the values from the text fields and store them within the **LoginDialog** object. When you **dispose** of **LoginDialog**, you eliminate only the GUI resources; the **LoginDialog** object still exists in memory. The code that made **LoginDialog** visible in the first place can then retrieve the necessary attributes from the **LoginDialog** object. Add some attributes and methods to **LoginDialog** as shown:

CODE TO TYPE: /src/client.gui/LoginDialog.java

```
package client.gui;

import util.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.GroupLayout.Alignment;

public class LoginDialog extends JDialog {
  String  userValue;
  String  hostValue;
  boolean isRegistered;
  String  hashedPassword;

  ...

  public boolean wasCanceled () { return isCanceled; }
  public String getUserValue() { return userValue; }
  public String getHostValue() { return hostValue; }
  public String getHashedPasswordValue() { return hashedPassword; }
  public boolean isSelfRegistered() { return isRegistered; }

  ...
}
```

These changes make it possible to retrieve all information recorded by the **LoginDialog** object. The next controller you write will be associated with the **OK** button; it will update these values for future retrieval. Make these changes to **LoginDialog**:

CODE TO TYPE: /src/client.gui/LoginDialog.java

```
public class LoginDialog extends JDialog {
  ...

  class OKAction {
    public void process() {
      isCanceled = false;
      hostValue = getHostField().getText();
      userValue = getUserField().getText();
      isRegistered = getRegisterCheckBox().isSelected();

      // Extract password and safely clean it out
      char [] chars = getPasswordField().getPassword();
      byte[] bytes = new byte[chars.length];

      for (int i = 0; i < bytes.length; i++) {
        bytes[i] = (byte) chars[i];
        chars[i] = '\0';
      }
      hashedPassword = Fingerprint.getFingerPrint(bytes);
      LoginDialog.this.dispose();
    }
  }

  public LoginDialog(JFrame parent) {
    this (parent, true);
  }

  public LoginDialog(JFrame parent, boolean modal) {
    super(parent, "Enter Login Credentials", true);
    setResizable(false);
    initLayout();

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
       new CancelAction().process();
      }
    });

    getCancel().addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
         new CancelAction().process();
      }
    });

    getOK().addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        new OKAction().process();
      }
    });

  }

  ...
}
```

The **process()** method in **OKAction** extracts values from the Swing GUI widgets and stores them in the class attributes you just added. When you use the **process()** methid, you can only **extract a char[] array** from a **JPasswordField** object. This is done to allow the caller to clean out the array contents safely and construct the **hashedPassword** using the **Fingerprint** class you've already developed. The designers of the Swing framework recommend the code fragment you've used to extract the characters. This helps to prevent a malicious third-party from retrieving the password, because you essentially zero it out when you extract it in the first place. Note that **LoginDialog** only stores the hashed password, and never the plain-text password, for security reasons. When you write code in this way, you satisfying your obligation to avoid storing (or transmitting) the user's password in plain text.

Change **Temp** to retrieve the values from **LoginDialog** as shown:

CODE TO TYPE: /src/Temp.java

```
import javax.swing.*;
import client.gui.*;

public class Temp {
  public static void main(String[] args) {
    final LoginDialog ld = new LoginDialog(null);
    ld.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    ld.setVisible(true);
    ~~System.out.println("DONE");~~
    System.out.println("canceled:" + ld.wasCanceled());
    System.out.println("user:" + ld.getUserValue());
    System.out.println("password:" + ld.getHashedPasswordValue());
    System.out.println("selfRegister:" + ld.isSelfRegistered());
  }
}
```

Run **Temp** and observe the output responses under different input circumstances; try closing the dialog by clicking **OK** and **Cancel**.

Let's add one more common GUI feature to **LoginDialog** that will enable the **OK** button only when all text fields have content. This requires using a **KeyListener** on all text fields.

CODE TO TYPE: /src/client.gui/LoginDialog.java

```
public class LoginDialog extends JDialog {

  ...

  class OKFilter extends KeyAdapter {
    public void keyReleased(KeyEvent e) {
      validateForm();
    }
  }

  void validateForm() {
    boolean enable = true;
    if (getHostField().getText().length() == 0) { enable = false; }
    if (getUserField().getText().length() == 0) { enable = false; }
    if (getPasswordField().getPassword().length == 0) { enable = false; }

    getOK().setEnabled(enable);
  }

  ...

  LoginDialog(JFrame parent, boolean modal) {
    super(parent, "Enter Login Credentials", modal);
    setResizable(false);
    initLayout();
    validateForm();

    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        new CancelAction().process();
      }
    });

    getCancel().addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        new CancelAction().process();
      }
    });

    getOK().addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        new OKAction().process();
      }
    });

    getPasswordField().addKeyListener(new OKFilter());
    getHostField().addKeyListener(new OKFilter());
    getUserField().addKeyListener(new OKFilter());
  }

  ...
}
```

These small changes, spread across **LoginDialog**, call **validateForm** when each key is released. In doing so, **validateForm** is called whenever the user types a key that alters the value stored in any text field. This sort of field-value checking is common in GUI systems, and it can result in awful code if it's not handled well. The enabling logic for the **OK** button is encapsulated within the **validateForm** method. Each controller that manages user updates must (at some point) invoke **validateForm** to enable (or disable) the **OK** button correctly.

Run **Temp** to verify that the **OK** button is unavailable until you enter data in all three fields.

Testing

The new code for this lesson is split between the client GUI and client IPC layers. You can write a set of test cases to validate **ServerAccess**, but you'll have to wait until the next lesson to write test cases for **LoginDialog**. For now, get started by modifying **RepositoryClient** to use the new protocol and retrieve its credentials and host information from **LoginDialog**:

CODE TO TYPE: /test/client/RepositoryClient.java

```
package client;

import java.io.*;
~~import java.net.*;~~
import javax.swing.*;
import xml.*;
import util.*;
import client.gui.*;

public class RepositoryClient {
  ~~public static String endResponse = "</response>";~~

  ~~static void processResponse(BufferedReader fromServer) throws IOException {~~
    ~~try {~~
      ~~StringBuilder buf = new StringBuilder(fromServer.readLine());~~
      ~~while (!buf.substring(buf.length()-endResponse.length(), buf.length()).equals(end~~
~~Response)) {~~
        ~~buf.append(fromServer.readLine());~~
      ~~}~~
      ~~Message response = new Message(buf.toString());~~
      ~~String sval = response.contents.getAttributes().getNamedItem("success").getNodeVa~~
~~lue();~~
      ~~if (Boolean.valueOf(sval)) {~~
        ~~System.out.println("Success");~~
      ~~} else {~~
        ~~System.out.println("Error:" + response.contents.getAttributes().getNamedItem("r~~
~~eason").getNodeValue());~~
      ~~}~~

    ~~} catch (IOException ioe) {~~
      ~~ioe.printStackTrace();~~
    ~~}~~
  ~~}~~

  public static void main(String[] args) throws Exception {
    final LoginDialog ld = new LoginDialog(null);
    ld.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

    ld.setVisible(true);
    if (ld.wasCanceled()) { System.exit(0); }

    String fp = ld.getHashedPasswordValue();
    String user = ld.getUserValue();
    String host = ld.getHostValue();
    boolean register = ld.isSelfRegistered();

    SplashScreenLogic.update ("connecting to ~~localhost~~" + host + "::9172");
    delay(250);
    ~~Socket server = new Socket ("localhost", 9172);~~
    ServerAccess sa = new ServerAccess(host, user, fp, register);
    if (!sa.connect()) {
      System.err.println ("Unable to connect to server:" + host);
      System.exit(-1);
    }

    SplashScreenLogic.update ("connected to localhost::9172");
    delay(250);

    ~~PrintWriter toServer = new PrintWriter (server.getOutputStream(), true);~~
    ~~BufferedReader fromServer = new BufferedReader (new InputStreamReader(server.getInp~~
~~utStream()));~~
    ~~SplashScreenLogic.update ("initializing with server...");~~
    ~~delay(250);~~

    String xmlStatusRequest = "<request><statusRequest/></request>";
    sa.sendRequest(new Message(xmlStatusRequest));
~~    toServer.println(xmlStatusRequest);~~
~~    processResponse(fromServer);~~

    File f = new File("images", "repositorySplash.png");
    String encoding = ImageEncoding.encode(f);
    String xmlAddRequest = "<request><addRequest name='sampleImage'>" +
         "<image>\n<![CDATA[" + encoding + "\n]]></image></addRequest></request>";

    sa.sendRequest(new Message(xmlAddRequest));

    // wait 5 seconds for everything to complete
    delay (5000);

    ~~toServer.println(xmlAddRequest);~~

~~    processResponse(fromServer);~~

~~    server.close();~~
    sa.disconnect();
    SplashScreenLogic.update ("closing");
    delay(250);
  }

  /** Delay for a time. */
  static void delay(int ms) {
    try { Thread.sleep(ms); } catch (InterruptedException ie) { }
  }
}
```

Launch a server by running **ServerLauncher**, and then run the updated **RepositoryClient** and select to self-register a new account. **RepositoryClient** is intended to test the core logic of the server; don't be concerned that it fails to dispose of the Splash Screen when **LoginDialog** is visible. For the host value, enter **localhost**. Then enter a name and password you'll remember, and check the **Self Register** box.

You'll see output like this in your Console window when you click the **OK** button:

OBSERVE:

```
connecting to localhost::9172
connected to localhost::9172
<?xml version="1.0" encoding="UTF-8"?><response success="true"><loginResponse user="lkj
kl"/></response>
<?xml version="1.0" encoding="UTF-8"?><response success="true"><statusResponse index="1
" key="SomeKey" total="1"/></response>
<?xml version="1.0" encoding="UTF-8"?><response reason="That image already exists in th
e repository." success="false">
    <addResponse numBytes="0"/></response>
closing
```

The third xml message appears only if you have run code in the past that uploaded an image to the Image Repository. If you can recall the user credentials for the account you just created, re-run **RepositoryClient** and login using the same credentials, but this time, do not check the **Self Register** box. Since the server is still running, the account is still active, and you can connect properly.

In the **/test** folder, create a **client.gui** package.

In the **/test** folder **client.gui** package, create a **TestLoginDialog** test case as shown:

CODE TO TYPE: /test/client.gui/TestLoginDialog.java

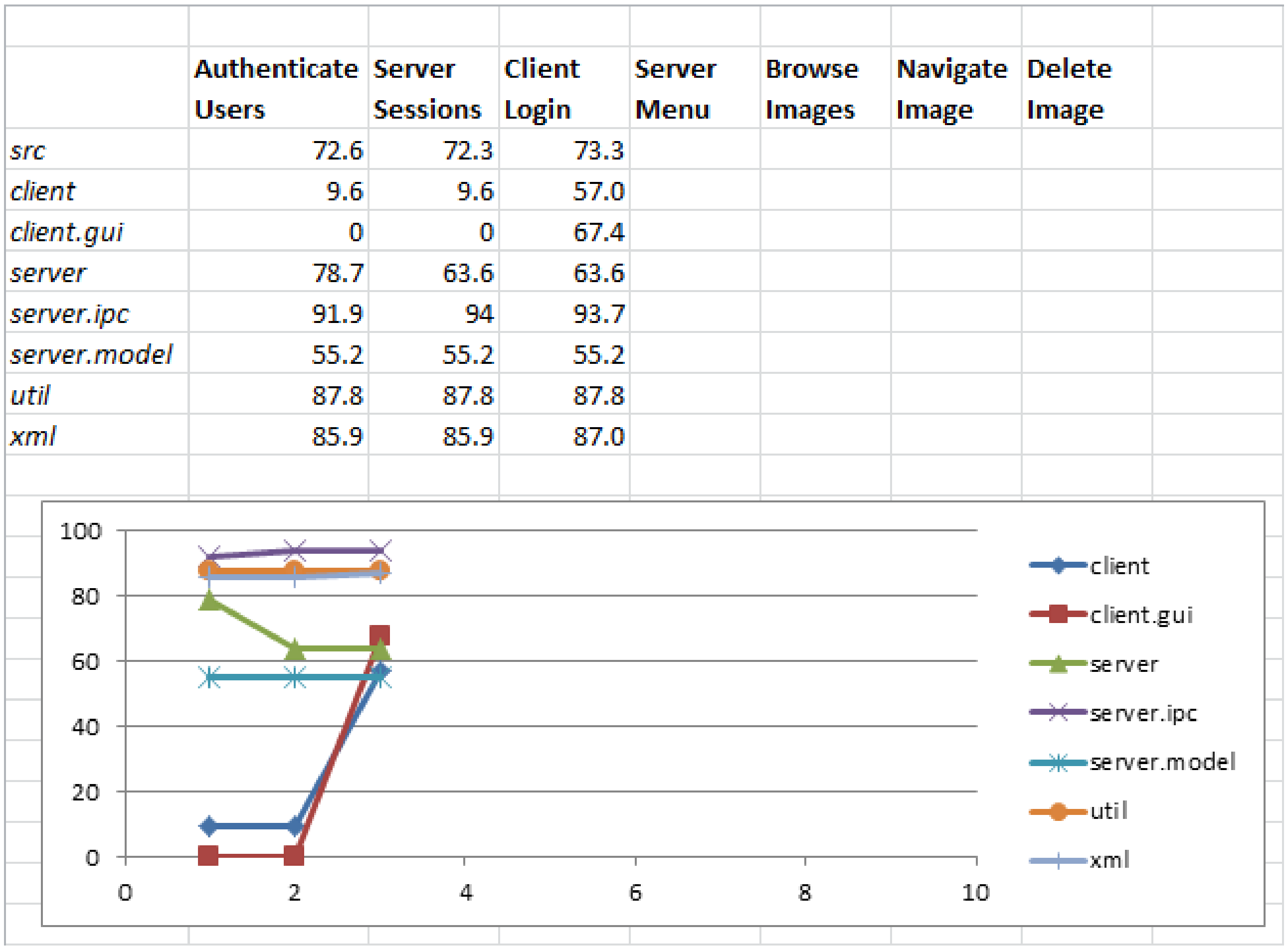
```java
package client.gui;

import util.*;
import junit.framework.TestCase;

public class TestLoginDialog extends TestCase {
  LoginDialog ld;

  protected void setUp () {
    ld = new LoginDialog(null, false);
    ld.setVisible(true);
  }

  protected void tearDown() {
    ld.dispose();
  }

  public void testInitialDisabled() {
    assertFalse (ld.getOK().isEnabled());
  }

  // validate second password.
  public void testRegistrationSituation() {
    assertFalse (ld.getOK().isEnabled());

    ld.getUserField().setText("sample");
    ld.getHostField().setText("localhost");
    ld.getRegisterCheckBox().doClick();

    ld.validateForm();

    assertFalse (ld.getOK().isEnabled());

    // Enter password
    ld.getPasswordField().setText("another");
    ld.validateForm();

    assertTrue (ld.getOK().isEnabled());

    // make the action occur
    ld.new OKAction().process();

    assertEquals ("sample", ld.getUserValue());
    assertEquals (Fingerprint.getFingerPrint("another".getBytes()), ld.getHashedPasswor
dValue());
    assertEquals ("localhost", ld.getHostValue());
    assertFalse (ld.wasCanceled());
    assertTrue (ld.isSelfRegistered());

    assertFalse (ld.isVisible());
  }
}
```

This test case will exercise the essential logic for **LoginDialog**. In past lessons you saw how to merge a number of EclEmma sessions to determine the full code coverage. Now terminate all running applications and launch **ServerLauncher** using EclEmma. Do the same for **RepositoryClient**. Exercise a few features in **LoginDialog**, like selecting (and unselecting) the register checkbox, clearing fields, entering invalid information, and so on. Next, choose to self-register a new account, enter proper credentials, and press **OK**. The dialog disappears and a coverage session will be generated.

Note There is no logic in **ServerLauncher** to automatically shut it down (yet). Terminate the application using the console tab or the Debug perspective. You will be notified that "No coverage data file has been written during this coverage session," because you terminated the application manually. We'll fix this problem in the next lesson.

Now launch all test cases within EclEmma as you've done in past lessons. Due to the changes to **Repository**, the **testMultipleClients** code in **TestServer** is no longer useful; it pops up multiple **LoginDialog** boxes. If you wait ten seconds though, they will go away. (Take a look at the test case method to see why this happens.) Now, use the "merge sessions" option to, well, merge all sessions.

Wow! Look how nice the coverage appears! (Note that your mileage may vary.)



	Authenticate Users	Server Sessions	Client Login	Server Menu	Browse Images	Navigate Image	Delete Image
src	72.6	72.3	73.3				
client	9.6	9.6	57.0				
client.gui	0	0	67.4				
server	78.7	63.6	63.6				
server.ipc	91.9	94	93.7				
server.model	55.2	55.2	55.2				
util	87.8	87.8	87.8				
xml	85.9	85.9	87.0				

You are exercising nearly 70% of the code you have written! Congratulations! You are making excellent progress. Keeep up the good work and see you in shortly.

Client Server Menu

Lesson Objectives

In this lesson you will:

- complete the implementation of the **Server** menu in the Client GUI application.

Client Server Menu

You're in the home stretch! As we prepare to assemble the GUI client application in its entirety, we want to make sure we have all the pieces ready to go. We need the client launching class, **ClientLauncher**, instantiates and makes the primary main client class, **ImageRepositoryClient**, visible. We also want all interaction with the server to occur through the **ServerAccess** class, which can be instantiated on demand. The client already has a **Preferences** class to manage all user customizations persistently.You've already seen "local controller" objects used—specifically by **LoginDialog** to manage user interactions. The final pieces we need to put into action in order to complete the client puzzle are the special controller classes that will manage all key functionality.

First, you'll need to update **ServerAccess** so it does more than just print out messages that it receives from the server. The true logic of this class should be externalized; to do that, you'll need an interface.

In the **/src** folder **client** package, create an **IProcessResponse** interface as shown:

CODE TO TYPE: /src/client/IProcessResponse.java

```
package client;

import xml.*;

public interface IProcessResponse {
  void process (Message response);
}
```

This interface defines a **process** method that will be implemented by a handler to respond to messages received from the server.

Now integrate this interface with **ServerAccess**. Modify your code as shown:

CODE TO TYPE: /src/client/ServerAccess.java

```
package client;

import java.io.*;
import java.net.*;
import xml.*;

public class ServerAccess extends Thread {
  String host = null;
  String user = null;
  String hashedPass = null;
  boolean selfRegister = false;

  Socket server;
  BufferedReader fromServer;
  PrintWriter toServer;

  boolean isActive = false;
  IProcessResponse handler;

  public ServerAccess(String host, String user, String hashedPass, boolean selfRegister
) {
    this.host = host;
    this.user = user;
    this.hashedPass = hashedPass;
    this.selfRegister = selfRegister;
  }

  public boolean connect(IProcessResponse handler) {
    this.handler = handler;

    try {
      server = new Socket (host, 9172);
      fromServer = new BufferedReader (new InputStreamReader(server.getInputStream()));
      toServer = new PrintWriter (server.getOutputStream(), true);
      isActive = true;
    } catch (Exception e) {
      System.err.println("Unable to connect to server: " + e.getMessage());
      isActive = false;
      return false;
    }

    start();
    return true;
  }

  public void run() {
    try {
      String selfAtt = "";
      if (selfRegister) { selfAtt = " register='true'"; }
      Message m = new Message("<request>" +
          "<loginRequest user='" + user + "' password='" + hashedPass + "' " + selfAtt
+ "/></request>");
      sendRequest(m);

      while (isActive) {
        m = Parser.extractResponse(fromServer);
        if (m == null) {
          break;
        }

        ~~// TODO: For now, just print it to console~~
        ~~System.out.println(m);~~
        handler.process(m);
      }

    } catch (Exception e) {
      e.printStackTrace();
    }

    disconnect();
  }

  public void disconnect() {
    isActive = false;
    try {
      server.close();
    } catch (IOException ioe) {
      System.err.println("Unable to close server:" + ioe.getMessage());
    }
  }

  public synchronized boolean sendRequest(Message r) {
    if (!isActive) { return false; }

    toServer.println(r);
    return !toServer.checkError();
  }
}
```

All messages received from the server are delegated to the **handler** for processing.

Integrate **LoginDialog** with the **Server** menu to see these changes in action. There are three menu items to complete:

- **Connect:** connect to a remote server using the **LoginDialog** window.
- **Disconnect:** disconnect from a remote server to make it possible to reconnect to a new server.
- **Quit:** quit the GUI application (after prompting user for confirmation).

Start with the **Quit** menu item, because the **ClientLauncher** code already has most of the logic for this functionality. Extract this logic and place it into a standalone controller class which can be invoked either by the closing of the main application window or the selection of the **Quit** menu item.

In the **/src** folder **client.gui** package, create the **QuitController** class as shown:

CODE TO TYPE: /src/client.gui/QuitController.java

```
package client.gui;

import javax.swing.*;
import util.*;

public class QuitController {
  final static String property_confirmOnExit = "ConfirmOnExit";
  static String imageFile = "images/help_32.png";
  static ImageIcon icon;

  public boolean confirm(ImageRepositoryClient client) {
    if (icon == null) {
      icon = new ImageIcon(imageFile);
    }
    if (!Preferences.isTrue(property_confirmOnExit)) {
      String[] choices = { "Confirm", "Confirm and don't ask me again" };

      String s = (String) JOptionPane.showInputDialog (client,
          "Do you wish to exit Image Repository?\n ",
          "Confirm Exit", JOptionPane.PLAIN_MESSAGE,
          icon, choices, choices[0]);

      if (s == null) {
        return false;
      } else if (s.equals (choices[1])) {
        Preferences.set(property_confirmOnExit, true);
      }
    }
    client.quit();
    return true;
  }
}
```

Ignore the compiler error for now, because you'll add a **quit()** method to **ImageRepositoryClient** soon, then you can modify **ClientLauncher** to use that **quit()** method. Take a look at the method signature for **confirm()**, which returns **true** on success or **false** on denial. You will continue to use this pattern in the other controllers you write for this lesson. Modify **ClientLauncher** to use **QuitController** as shown:

CODE TO TYPE: /src/client/ClientLauncher.java

```
package client;

import java.awt.event.*;
import javax.swing.*;
import client.gui.*;
~~import util.*;~~

public class ClientLauncher {
~~  static final String property_confirmOnExit = "ConfirmOnExit";~~

  public static void main(String[] args) {
    final ImageRepositoryClient irc = new ImageRepositoryClient();
    irc.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

~~    final ImageIcon icon = new ImageIcon("images/help_32.png");~~
    irc.addWindowListener(new WindowAdapter() {

      public void windowClosing(WindowEvent we) {
        if (new QuitController().confirm(irc)) {
          irc.dispose();
        }
~~        if (!Preferences.isTrue(property_confirmOnExit)) {~~
~~          String[] choices = { "Confirm", "Confirm and don't ask me again" };~~
~~          String s = (String) JOptionPane.showInputDialog(irc,~~
~~                  "Do you wish to exit Image Repository\n ",~~
~~                  "Confirm Exit", JOptionPane.PLAIN_MESSAGE,~~
~~                  icon, choices, choices[0]);~~

~~          if (s == null) {~~
~~            return;~~
~~          } else if (s.equals (choices[1])) {~~
~~            // remember this in the future.~~
~~            Preferences.set(property_confirmOnExit, true);~~
~~          }~~
~~        }~~

~~        irc.dispose();~~
      }
    });

    irc.setVisible(true);
  }
}
```

Now **ClientLauncher** is more straightforward, and **QuitController** can be reused in **ImageRepositoryClient**. The **QuitController** doesn't compile yet because you still need to add a **quit()** method to **ImageRepositoryClient**; you'll do that shortly.

All of the code changes you make during this lesson work toward the single purpose of defining the GUI controllers we'll need to handle the application functionality we want. Once we have that in order, our next task is to develop a response handler to process the response messages received by the client as the application proceeds.

The **Connect** and **Disconnect** functionalities are mutually exclusive, so you need to enable their corresponding menu items appropriately. For example, **Disconnect** can only be enabled once the client has connected to a remote server. In the same way that you validated a form before allowing the **OK** button to be enabled earlier, you can have a similar method enable or disable menu items based on the state of the client. So, where should this client state be maintained? Good question! It begins at **ImageRepositoryClient**, so we'll place the **validateMenuBar()** method there, to be invoked whenever the connection state with the server is updated (for instance, during initialization, connection, or disconnection).

Let's make the necessary changes one at a time. First, add some attributes and a method to **ImageRepositoryClient**:

CODE TO TYPE: /src/client.gui/ImageRepositoryClient.java

```
import java.awt.*;
import javax.swing.*;
import javax.swing.GroupLayout.Alignment;
import java.awt.event.*;
import client.*;

/** Primary GUI window for the client application. */
public class ImageRepositoryClient extends JFrame {
  JScrollPane imgPanel;
  JTextArea imgMetaData;
  JTextField status;
  ServerAccess access;
  JMenu     image;
  JMenuItem connect;
  JMenuItem disconnect;
  JMenuItem quit;

  ...

  public void validateMenuBar() {
    connect.setEnabled(access == null);

    disconnect.setEnabled(access != null);
    image.setEnabled(access != null);
  }
}
```

The **validateMenuBar()** method defines the conditions under which the **connect** and **disconnect** menu items are enabled. Whenever there is a non-**null access** object, the client should be able to disconnect from the server and interact with the **image** menu, otherwise, only the **connect** menu item should be enabled.

Now, modify the **initMenuBar()** method to use these attributes. Take care to call **validateMenuBar()** at the end of the constructor to configure the GUI properly before it becomes visible to the user.

CODE TO TYPE: /src/client.gui/ImageRepositoryClient.java

```
  public ImageRepositoryClient() {
    super("Image Repository Client");
    initMenuBar();
    initLayout();
    validateMenuBar();
  }

  void initMenuBar() {
    JMenuBar mb = new JMenuBar();

    JMenu server = new JMenu ("Server");
    connect = new JMenuItem("Connect...");
    server.add(connect);
    connect.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        new ConnectController().connect(ImageRepositoryClient.this);
      }
    });

    disconnect = new JMenuItem("Disconnect...");
    server.add(disconnect);

    quit = new JMenuItem("Quit...");
    server.add(quit);
    quit.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        new QuitController().confirm(ImageRepositoryClient.this);
      }
    });
    mb.add(server);

    ~~JMenu ~~image = new JMenu ("Image");
    mb.add(image);

    setJMenuBar(mb);
  }
```

These changes register two new controllers: one to react to the user's selection of the **Connect...** menu item, and the other to use the above **QuitController**. The **image JMenu** object created now is stored by the **image** class attribute, rather than by a local variable as it was before; this allows you to write code to manipulate the state (that is, whether enabled or disabled) of the menu items later. Finally, you pass the **ImageRepositoryClient** object to the **QuitController** using this code fragment:

OBSERVE:

```
new QuitController().confirm(ImageRepositoryClient.this);
```

Because this fragment exists within the anonymous class defined to implement **ActionListener**, you cannot just pass **this** as an argument. Using **ImageRepositoryClient.this** states your intention to pass the enclosing **ImageRepositoryClient** object.

Now we're ready for the final push! While **ImageRepositoryClient** is the primary class for the GUI application, you have to be careful not to embed too much application logic within it, otherwise you run the risk of not being able to test that application logic properly. Whenever possible, encapsulate control logic in separate controllers; at the same time, you can place some methods, for example **quit()**, into **ImageRepositoryClient**, because certain methods have global impact.

In the **/src** folder **client.gui** package, create a **ConnectController** class as shown:

CODE TO TYPE: /src/client.gui/ConnectController.java

```
package client.gui;

import javax.swing.*;
import client.*;

public class ConnectController {
  public boolean connect(ImageRepositoryClient client) {
    final LoginDialog ld = new LoginDialog(client);
    ld.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

    ld.setVisible(true);
    if (ld.wasCanceled()) {
      return false;
    }

    String fp = ld.getHashedPasswordValue();
    String user = ld.getUserValue();
    String host = ld.getHostValue();
    boolean register = ld.isSelfRegistered();
    return connect(client, host, user, fp, register);
  }

  public boolean connect(ImageRepositoryClient client, String host, String user, String
 fp, boolean register) {
    ServerAccess sa = new ServerAccess(host, user, fp, register);
    if (!sa.connect(new ResponseHandler(client))) {
      return false;
    }

    client.access = sa;
    client.validateMenuBar();
    return true;
  }
}
```

The **ConnectController** displays the **LoginDialog** and requests information from the user. If the user canceled the dialog, **LoginDialog** returns **false** and **ConnectController** returns **false**. If the attempt to connect fails, it also returns **false**. If it returns **true**, you know that the client connected to the server successfully. This particular controller stores the **ServerAccess** object directly with the **client** and then invokes **validateMenuBar** to update the GUI properly.

ConnectController has two methods named **connect**. The first (with just an **ImageRepositoryClient** as an argument) is intended for interactive behavior. However, this method cannot be tested automatically because it requires the user's direct involvement. That's why we write a second **connect** method that takes an additional four arguments whose values are extracted from the **LoginDialog** presented to the user. In this way, the real logic of this controller can be placed in a method that works with automated testing.

ConnectController does not compile because it depends on an undefined class, **ResponseHandler**.

In the **/src** folder **client** package, create a **ResponseHandler** class as shown:

CODE TO TYPE: /src/client/ResponseHandler.java

```
package client;

import client.gui.*;
import xml.*;

public class ResponseHandler implements IProcessResponse {

  ImageRepositoryClient client;

  public ResponseHandler (ImageRepositoryClient client) {
    this.client = client;
  }

  public void process(Message response) {
    String type = response.contents.getFirstChild().getLocalName();

    // handle loginResponse specially
    if (type.equals(Parser.loginResponse)) {
      if (!Parser.success(response)) {
        client.status("Unable to login:" + Parser.reason(response));
      } else {
        client.status("Connected to server.");
      }

      client.validateMenuBar();
      return;
    }

    System.out.println("received:" + response);
  }
}
```

ResponseHandler processes all response messages coming back from the server. You don't want the low-level IPC handling code to be responsible, and you don't want the top-level **ImageRepositoryClient** to embed this logic. Instead, **ResponseHandler** takes on those responsibilities.

The **process()** method deals with all responses coming back from the server. The **loginResponse** must be handled in a particular way to enable and disable menu items. The above code won't compile until you provide some additional helper methods. First, we'll add some methods to **ImageRepositoryClient**:

CODE TO TYPE: /src/client.gui/ImageRepositoryClient.java

```
...
  public void clearStatus() {
    statusBar().setText("");
  }

  public void status(String msg) {
    statusBar().setText(msg);
  }

  public void quit() {
    if (access != null) {
      access.disconnect();
    }
    access = null;
    setVisible(false);
    dispose();
  }
}
```

There are lots of places within the client that you can check to determine whether a response has succeeeded. Instead of embedding the XML-parsing logic throughout your client code, delegate it to the XML utility **Parser** class, as shown:

CODE TO TYPE: /src/xml/Parser.java

```
package xml;

import java.io.*;
import org.w3c.dom.*;

public class Parser {
  public final static String loginRequest        = "loginRequest";
  public final static String loginResponse       = "loginResponse";

  public final static String loginUser           = "user";
  public final static String loginPassword       = "password";
  public final static String loginRegister       = "register";

  public final static String invalidCredentials  = "Invalid credentials";

  public final static String success             = "success";
  public final static String reason              = "reason";

  public static Message extractRequest(BufferedReader in) {
    return extractMessage(in, "</request>");
  }

  public static Message extractResponse(BufferedReader in) {
    return extractMessage(in, "</response>");
  }

  static Message extractMessage(BufferedReader in, String terminator) {
    try {
      String line = in.readLine();
      if (line == null) { return null; }
      StringBuilder buf = new StringBuilder(line);
      while (!buf.substring(buf.length()-terminator.length(), buf.length()).equals(term
inator)) {
        line = in.readLine();
        if (line == null) { return null; }
        buf.append(line);
      }

      return new Message (buf.toString());
    } catch (IOException ioe) {
      return null;
    }
  }

  public static boolean success(Message response) {
    return Boolean.valueOf(response.contents.getAttributes().getNamedItem(success).getN
odeValue());
  }

  public static String reason(Message response) {
    Node r = response.contents.getAttributes().getNamedItem(reason);
    if (r == null) { return ""; }
    return r.getNodeValue();
  }
}
```

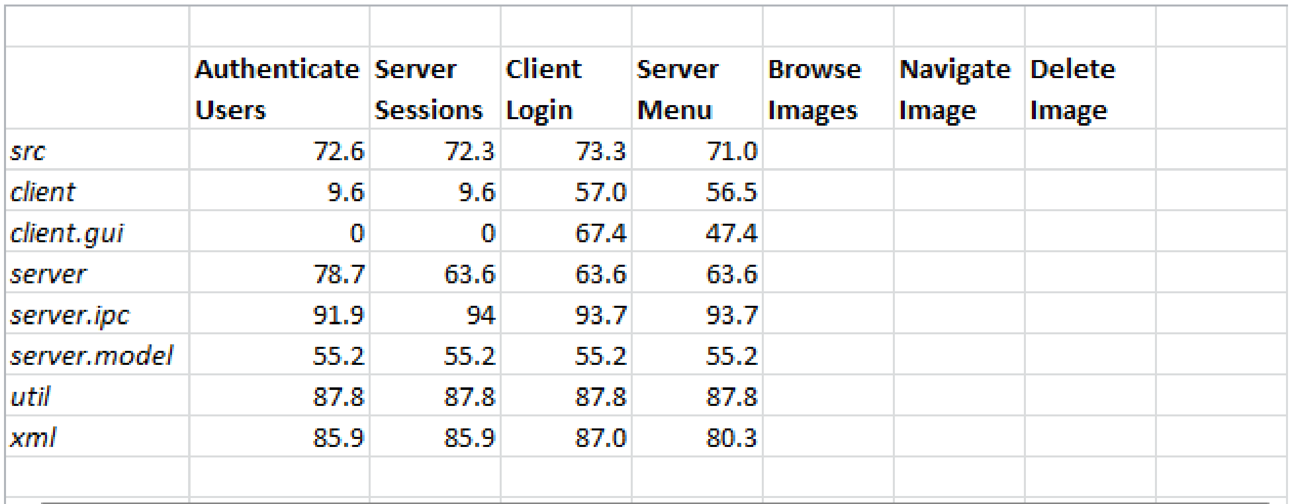
Now you have a fully-functioning client with a **Connect** menu item that's connected to engage the server. Currently, if you want to disconnect the client, you must exit it, but we'll remedy that little inconvenience in the next lesson.

Before you run the tests, notice that **RepositoryClient** is broken! We need to modify it to register an **IProcessResponse** object with the **ServerAccess**. Because this is a support class for testing, it's enough to register a handler that does nothing but output the message to the console. Modify **RepositoryClient** as shown:

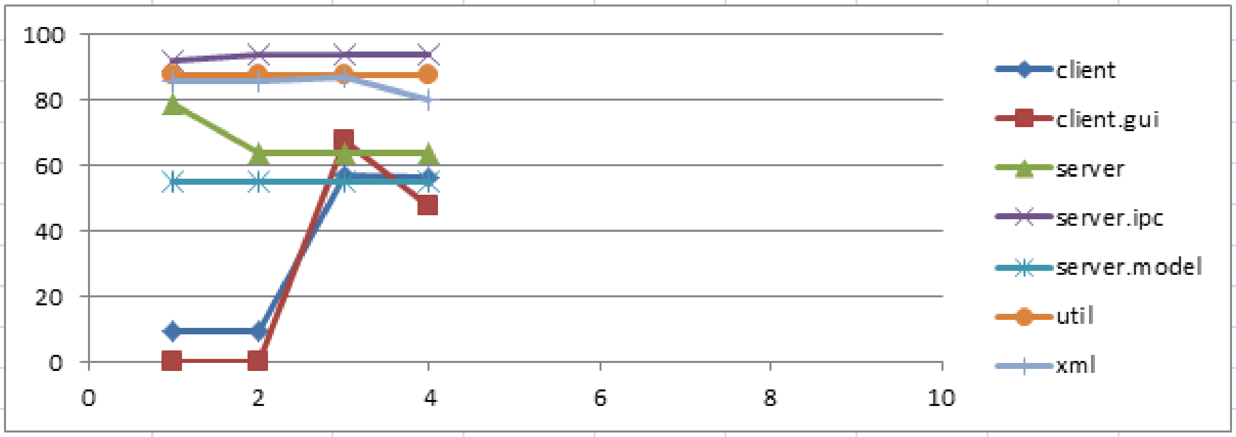
CODE TO TYPE: /test/client/RepositoryClient.java

```
package client;

import java.io.*;
import javax.swing.*;
import xml.*;
import util.*;
import client.gui.*;

public class RepositoryClient {

  public static void main(String[] args) throws Exception {
    final LoginDialog ld = new LoginDialog(null);
    ld.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    ld.setVisible(true);
    if (ld.wasCanceled()) { System.exit(0); }

    String fp = ld.getHashedPasswordValue();
    String user = ld.getUserValue();
    String host = ld.getHostValue();
    boolean register = ld.isSelfRegistered();

    SplashScreenLogic.update ("connecting to " + host + "::9172");
    delay(250);
    ServerAccess sa = new ServerAccess(host, user, fp, register);
    IProcessResponse handler = new IProcessResponse() {
      public void process(Message response) {
        System.out.println(response);
      }
    };
    if (!sa.connect(handler)) {
      System.err.println ("Unable to connect to server:" + host);
      System.exit(-1);
    }

    SplashScreenLogic.update ("connected to localhost::9172");
    delay(250);
...
```

Now it's time to generate our coverage. Run all test cases using EclEmma first, to compute the automatic coverage. Next, run **ServerLauncher** as a stand-alone executable. Run **RepositoryClient** using EclEmma and exercise a few of features in **LoginDialog** before self-registering a new account. You'll have to terminate the **ServerLauncher** manually. Now merge the EclEmma sessions and your coverage results will look something like this:



	Authenticate Users	Server Sessions	Client Login	Server Menu	Browse Images	Navigate Image	Delete Image
src	72.6	72.3	73.3	71.0			
client	9.6	9.6	57.0	56.5			
client.gui	0	0	67.4	47.4			
server	78.7	63.6	63.6	63.6			
server.ipc	91.9	94	93.7	93.7			
server.model	55.2	55.2	55.2	55.2			
util	87.8	87.8	87.8	87.8			
xml	85.9	85.9	87.0	80.3			



Of course there's still planty of work to do. The client still cannot disconnect from the server, for example. This task will be completed in the next lesson. Whenerv you add client code, make sure to add new automated test cases to maintain quality control over your code as you go. Our overall testing progress is quite good, and we want to keep it that way!

Image Browsing

Lesson Objectives

In this lesson you will:

- apply the cycle of message request and message response capabilities between a client and server.

Image Browsing

You're really rolling now. Pretty soon you'll be able to upload images to the repository and see them in the **ImageRepositoryClient** window! You'll start by adding capability that allows the user to select an image file from disk and upload that image to the repository. All the pieces are in place to show the full integration of a request being sent from the GUI and the server generating a proper response. Let's break this task up into steps:

1. Complete the **Image** menu to allow a connected client to select image for upload.
2. Determine the supported Java image formats (requirement **R2**).
3. Allow users to select images (of the appropriate type) from their computer files.
4. Generate an **addRequest** in the client to send to the server for a selected image.
5. Receive a confirmation **addResponse** from the server (either success or failure).

Along the way, we'll follow the pattern we've developed in past lessons to make incremental progress and enable JUnit testing. The goal is to develop a single controller class to handle the steps we've listed above and thereby simplify the control logic in the client.

In the **/src** folder **client** package, create a new **IController** interface as shown:

CODE TO TYPE: /src/client/IController.java

```
package client;

import xml.*;

public interface IController {
  void process (Message request, Message response);
}
```

All client controllers that are responsible for managing the interaction of request/response messages between the client and server must implement the above interface.

This interface allows you to access the originating client request when processing the server's response to that request. In doing so, you will be able to correlate responses that appear from the server. In order to do that, the **ServerAccess** client IPC code needs to keep track of the controller objects that register an interest in a response. Since each request is sent to the server one by one, and in turn processed by the server, each reponse to a client request is returned in order. So, you can use a Queue to match up waiting controllers with the corresponding responses received by the server. Actually, the storage will be a LinkedList, but you get the idea. You can modify **ServerAccess** to make these changes while still supporting the existing "send a request without worrying about a response" behavior already implemented. Go ahead and modify **ServerAccess** as shown:

CODE TO TYPE: /src/client/ServerAccess.java

```
package client;

import java.io.*;
import java.net.*;
import java.util.*;
import xml.*;

public class ServerAccess extends Thread {

  ...

  Queue<Pair> queue = new LinkedList<Pair>();

  class Pair {
    IController  controller;
    Message      request;

    Pair (IController c, Message r) {
      controller = c;
      request = r;
    }
  }

  public synchronized boolean isWaiting() {
    return (!queue.isEmpty());
  }

  ...
}
```

The inner class **Pair** represents an (**IController**, **Message**) pair. The **queue** object maintains a linked list of **Pair** objects, which represent the requests and corresponding controllers waiting for responses to those requests. The **isWaiting()** helper method determines whether there are any **Pair** objects in **queue**.

To take advantage of this new capability, add a **sendRequest** method to **ServerAccess**:

CODE TO TYPE: /src/client/ServerAccess.java

```
...
 public synchronized boolean sendRequest(IController c, Message r) {
    if (!isActive) { return false; }

    toServer.println(r);
    boolean success = !toServer.checkError();
    if (success) {
      queue.add(new Pair(c, r));
    }
    return success;
  }
...
```

This method adds a **Pair** object for the designated **IController** and **Message** to the end of **queue**, if the request was successfully written to the server.

Now, whenever a response is received by the client, it must check the **queue** to see whether there is a waiting **Pair** object in the queue. If there is, then the response is handled by the **IController** object associated with the pair. Modify the **run** method of **ServerAccess** as shown:

CODE TO TYPE: /src/client/ServerAccess.java

```
...
  public void run() {
    try {
      String selfAtt = "";
      if (selfRegister) { selfAtt = " register='true'"; }
      Message m = new Message("<request>" +
          "<loginRequest user='" + user + "' password='" + hashedPass + "' " + selfAtt
+ "/></request>");
      sendRequest(m);

      while (isActive) {
        m = Parser.extractResponse(fromServer);
        if (m == null) {
          break;
        }

        Pair p = queue.poll();
        if (p != null) {
          p.controller.process(p.request, m);
        } else {
          handler.process(m);
        }
      }

    } catch (Exception e) {
      e.printStackTrace();
    }

    disconnect();
  }
```

Let's take a closer look:

OBSERVE:

```
        Pair p = queue.poll();
        if (p != null) {
          p.controller.process(p.request, m);
        } else {
          handler.process(m);
        }
      }
```

This code will **poll the queue object** (in non-blocking fashion) to see if there is a registered **Pair** object in place for the extracted response. **If there is not**, the response is handled using the **handler** object. If there is a **Pair** object in the queue, it is removed from the queue and the **controller associated with the Pair object is given both the original request (p.request) and the response (m) for processing**.

Now let's consider an instance when the client disconnects from the server. The disconnect may take place while **queue** has a registered **Pair** object, so, to disconnect properly, we have to clear out the queue:

CODE TO TYPE: /src/client/ServerAccess.java

```
  public void disconnect() {
    isActive = false;
    queue.clear();

    try {
      server.close();
    } catch (IOException ioe) {
      System.err.println("Unable to close server:" + ioe.getMessage());
    }
  }
```

The **Image** menu needs an **Add** menu item and an associated **AddImageController** to oversee the action associated with this menu item. In the **initMenuBar** method of **ImageRepositoryClient**, make these changes:

CODE TO TYPE: /src/client.gui/ImageRepositoryClient

```
...
    image = new JMenu ("Image");
    JMenuItem add = new JMenuItem("Add...");
    add.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        new AddImageController(ImageRepositoryClient.this).add();
      }
    });
    image.add(add);
    mb.add(image);
...
```

In the **/src** folder **client.gui** package, create an **AddImageController** class. This controller must grab a file selected by the user based on the set of supported image formats. For more information, see this useful tutorial on the subject. You will use a JFileChooser to browse the local disk for a file with a name that matches a given FileNameExtensionFilter. Fortunately, there is a Java API to determine the allowed file types. Note that **AddImageController** must implement **IController** because it will register itself to process the **addResponse** message returned by the server. Let's tackle this class step by step:

CODE TO TYPE: /src/client.gui/AddImageController.java

```
package client.gui;

import java.io.*;
import javax.swing.*;
import javax.swing.filechooser.*;
import javax.imageio.*;

import util.*;
import xml.*;
import org.w3c.dom.*;
import client.*;

public class AddImageController implements IController {

  ImageRepositoryClient client;

  public AddImageController (ImageRepositoryClient client) {
    this.client = client;
  }

  public void process (Message request, Message response) {
    NamedNodeMap map = request.contents.getFirstChild().getAttributes();
    String name = map.getNamedItem("name").getNodeValue();

    if (Parser.success(response)) {
      client.status("Image uploaded to server:" + name);
    } else {
      client.status("Problem adding image:" + name + "(" + Parser.reason(response) + ")
");
    }
  }
}
```

This code represents the framework of a typical **IController** implementation. The **process()** method is invoked by **ServerAccess** when a response is received for a specific request sent by the client. Now add to **AddImageController** the key methods that actually send the request to the server in the first place:

CODE TO TYPE: /src/client.gui/AddImageController.java

```
...
  public boolean add() {
    client.clearStatus();
    FileNameExtensionFilter filter = new FileNameExtensionFilter(
         "Supported Image Types", ImageIO.getReaderFormatNames());

    JFileChooser chooser = new JFileChooser();
    chooser.setFileFilter (filter);
    if (chooser.showOpenDialog(client) != JFileChooser.APPROVE_OPTION) {
      return false;
    }

    File f = chooser.getSelectedFile();
    return add(f);
  }

  public boolean add (File f) {
    try {
      String encoding = ImageEncoding.encode(f);
      String xmlAddRequest = "<request><addRequest name='" + f.getName() + "'>" +
         "<image>\n<![CDATA[" + encoding + "\n]]></image></addRequest></request>";

      return client.access.sendRequest(this, new Message(xmlAddRequest));
    } catch (IOException ioe) {
      client.status("Problem adding image:" + ioe.getMessage());
      return false;
    }
  }
...
```

Let's take a closer look:

OBSERVE:

```
public boolean add() {
    client.clearStatus();
    FileNameExtensionFilter filter = new FileNameExtensionFilter(
         "Supported Image Types", ImageIO.getReaderFormatNames());

    JFileChooser chooser = new JFileChooser();
    chooser.setFileFilter (filter);
    if (chooser.showOpenDialog(client) != JFileChooser.APPROVE_OPTION) {
      return false;
    }

    File f = chooser.getSelectedFile();
    return add(f);
  }

  public boolean add (File f) {
    try {
      String encoding = ImageEncoding.encode(f);
      String xmlAddRequest = "<request><addRequest name='" + f.getName() + "'>" +
         "<image>\n<![CDATA[" + encoding + "\n]]></image></addRequest></request>";

      return client.access.sendRequest(this, new Message(xmlAddRequest));
    } catch (IOException ioe) {
      client.status("Problem adding image:" + ioe.getMessage());
      return false;
    }
  }
```

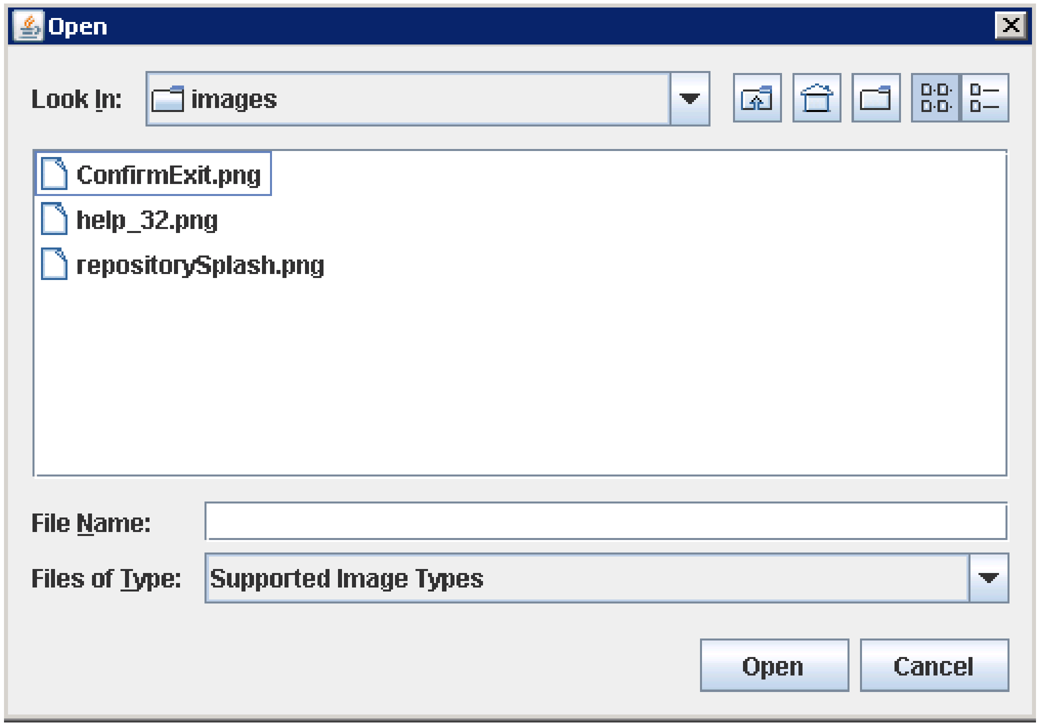
Invoking the **add()** method brings up a **JFileChooser** object that allows the user to select an image file from the file system. Once an image is selected, the method invokes **add(f)** to carry out this logic. This code is probably familiar to you, except for **how the request is sent to the server**. The code invokes the added **sendRequest()** method in

ServerAccess with the appropriate arguments.

You might wonder why there is an **add()** method and an **add(File)** method. Throughout the course, you've written code that can be tested automatically. By structuring our code this way, **add** interacts with the user who selects a file from disk, while **add(File)** represents a programming interface that supports **add**, and (more importantly) enables testing.

process() picks up where the **add(File)** method leaves off. When you use distributed computation, the challenge is to develop control flow patterns that deal with asynchronous communication with a server. Let's try doing that:

Run **ServerLauncher** and execute **ClientLauncher**. Select **Server | Connect**. Enter **localhost** and then a self-registered account and password. Select the new **Image | Add...** menu item. A dialog window appears. Browse to a directory where you have image files, for example, the **/images** directory in the **/workspace/DistributedApp** directory:



Select a file that passes the filter (currently this includes BMP, GIF, JPEG, JPG, PNG, and WBMP formats) and click **Open**. A successful status message appears in the bottom of the **ImageRepositoryClient** window. If you refresh the **Repository** folder within Eclipse, you will see a new file that contains the bytes of the image you selected.

So, you may wonder, "when do you clear the status messages?" There is no good answer that question. One strategy is to clear the status at the outset of any user command (as you see here in the **add** method); if you do that, then when the user chooses to cancel the command, at least you've already cleared any status. If the command completes, then the status will be visible until the user initiates a new command. This will be handled at the start of every GUI controller.

Just for fun, try to add the image again. The response fails because the duplicate image exists in the repository. Be sure to terminate the execution of both **ClientLauncher** and **ServerLauncher** before continuing.

Testing

With the new code, you also need to write some test cases for validation. While you've planned ahead and designed **AddImageController** with testing in mind, there are other challenges. Each JUnit test case class must belong to a package. Within that package, the JUnit test case has special access to "package private" attributes and methods. You have taken advantage of this feature in test cases you've written in past lessons. But what happens when you want to write a test case that must somehow access "package private" attributes and methods from two different packages? You can define special helper classes to be placed in the **/test** source code folder which essentially grant you that kind of access.

In the **/test** folder **server.ipc** package, create the **RepositoryServerAccess** class as shown:

CODE TO TYPE: /test/server.ipc/RepositoryServerAccess.java

```
package server.ipc;

import java.io.*;

public class RepositoryServerAccess {
  public static int size(RepositoryServer server) {
    return server.repository.size();
  }

  public static void shutdown (RepositoryServer server) throws IOException {
    server.shutdown();
  }
}
```

This class makes it possible for test cases to access privileged information without forcing the actual code to make it public. This technique maintains the integrity of your code base, while enhancing the productivity of your test cases.

Your new code enables your test cases to determine the number of images in a given **RepositoryServer** and shut down a **RepositoryServer**.

In the **/test** folder **client.gui** package, create a **TestAddImageController** test class as shown:

CODE TO TYPE: /test/client.gui/TestAddImageController.java

```
package client.gui;

import java.io.*;
import client.*;
import server.ipc.*;
import junit.framework.TestCase;

public class TestAddImageController extends TestCase {

  RepositoryServer server;
  static final String user = "tester";
  static final String imageFile = "images/repositorySplash.png";

  protected void setUp() throws Exception {
    TestServer.clearTestRepository();
    server = TestServer.launchServer();
  }

  protected void tearDown() throws Exception {
    RepositoryServerAccess.shutdown(server);
    server = null;
  }

  public static ImageRepositoryClient loginClient() throws Exception {
    ImageRepositoryClient client = new ImageRepositoryClient();

    client.access = new ServerAccess("localhost", user, user, true);
    if (!client.access.connect(new ResponseHandler(client))) {
      fail ("unable to connect to localhost");
    }

    Thread.sleep(1000);  // give time to connect
    return client;
  }

  public void testImageAdd() throws Exception {
    ImageRepositoryClient client = loginClient();
    assertEquals (0, RepositoryServerAccess.size(server));

    AddImageController c = new AddImageController(client);
    assertTrue (c.add(new File (imageFile)));

    // wait until all responses have been received before continuing test case
    int ctr = 50;
    while (client.access.isWaiting() && ctr-- > 0) {
      Thread.sleep(200);
    }

    // validate repository add succeeded.
    assertEquals (1, RepositoryServerAccess.size(server));
    client.quit();
    client.dispose();
  }
}
```

Here, you validate the capabilities of **AddImageController**. The **TestAddImageController** test case is similar in nature to the **TestSequence** test case you saw earlier, but the level of abstraction has been raised. No longer do you see raw sockets or InputStream objects; instead we're working at the level of GUIs and controllers. The **testImageAdd** test case method relies on the **setUp** method to instantiate a working **RepositoryServer**. Then it self-registers a "tester" user and invokes the **add(File)** method of the instantiated **AddImageController** in the exact same way that the interactive user would. Then the test case waits until the **ServerAccess** object determines that it is no longer waiting for a response from the server; this only happens when the server has responded properly to the **addRequest** generated by the **AddImageController**. Upon completion, the test case validates that there is one image in the repository (using the newly defined **RepositoryAccess** class) before quitting the client. The **tearDown()** method safely shuts down the **RepositoryServer**.

This test case leverages the design of the user interface to create an effective test. Execute the test case to validate that it completes successfully. You can refresh the **TestRepository** folder to see the files that were added as part of this test case.

Browse Repository

The completed functionality allows you to upload images to the repository, but it doesn't support browsing yet. For that to happen, the client must be able to show a current image in the repository, and either advance or return to another image. To support this capability, you need to keep track of which image the client is viewing. You decide whether the server side or the client side will be responsible for this knowledge. In the client/server architecture, it's common to consider the client to be "stateless," and thereby delegate responsibility to the server. Next, the server needs to store the image being viewed for all connected clients; so which class should hold this information? It can't be **RepositoryThread**, because that class is responsible for the information of a specific user. The **UserManager** can authenticate users, but should it be involved in the server-wide state for each connected user? That doesn't seem like such a good idea because the UserManager is on the client side, and server-wide stuff ought to be managed on the server side.

In the **/src** folder **server** package, create the **ClientState** class as shown:

CODE TO TYPE: /src/server/ClientState.java

```
package server;

import server.ipc.*;

public class ClientState {
  final String user;
  final RepositoryThread thread;
  String imageKey;

  public ClientState (String user, RepositoryThread thread) {
    this.user = user;
    this.thread = thread;
  }

  public void setImageKey(String key) {
    this.imageKey= key;
  }

  public String getImageKey() {
    return imageKey;
  }
}
```

The information in **UserInfo** is meant to be persistent, so we need to create a separate class, **ClientState**, that represents transient client information while it is connected to the server.

For now, the client state consists of the unique key generated for each image in the repository.

RepositoryServer will maintain a Hashtable of current connected users and update it as clients connect and disconnect. To enable this information to be accessed globally, it will be stored as a **static** Hashtable in **RepositoryServer** and a set of **static** helper methods will be provided. To allow these static methods to access the **Repository** object associated with the **RepositoryServer**, the **repository** attribute is changed to be **static**. This makes sense because in any given server there will only be a single **RepositoryServer** object. Modify your code as shown:

CODE TO TYPE: /src/server.ipc/RepositoryServer.java

```
package server.ipc;

import java.io.*;
import java.net.*;
import java.util.*;
import server.model.*;
import server.*;

public class RepositoryServer {
  ServerSocket serverSocket = null;
  int state = 0;
  IProtocolHandler protocolHandler;
  static Repository repository;
  UserManager manager;

  static Hashtable<String, ClientState> users = new Hashtable<String, ClientState>();

  public RepositoryServer(Repository rep, UserManager um, IProtocolHandler ph) {
    protocolHandler = ph;
    repository = rep;
    manager = um;
  }

  public void bind() throws IOException {
    serverSocket = new ServerSocket(9172);
    state = 1;
  }

  public void process() throws IOException {
    while (state == 1) {
      Socket client = serverSocket.accept();

      new RepositoryThread(manager, client, protocolHandler).start();
    }

    shutdown();
  }

  void shutdown() throws IOException {
    manager.store();
    if (serverSocket != null) {
      serverSocket.close();
      serverSocket = null;
      state = 0;
    }
  }

  public static boolean register (String user, RepositoryThread thread) {
    if (users.containsKey(user)) { return false; }

    ClientState state = new ClientState(user, thread);
    state.setImageKey(repository.getNthKey(1));
    users.put(user, state);
    return true;
  }

  public static void unregister(String user) {
    users.remove(user);
  }

  public static ClientState getState(String user) {
    return users.get(user);
  }
}
```

The newly added **register()** and **unregister()** methods maintain the **users** Hashtable by associating a **ClientState** object with each client that connects. The initial state for each user is the image key for the first image in the repository. Add the **getNthKey** method to **Repository** in order for this code to compile:

CODE TO TYPE: /src/server.model/Repository.java

```
...
  public String getNthKey(int n) {
    return index.getNthKey(n);
  }
...
```

Add this method to **Index** to allow the arbitrary retrieval of an image by its position:

CODE TO TYPE: /src/server.model/Index.java

```
...
  public String getNthKey(int n) {
    if (n > keys.size()) { return null; }
    return keys.get(n-1);
  }
...
```

Note that you have to subtract 1 when invoking **keys.get(n-1)**, because the **keys ArrayList** uses zero-based indexing. Now when clients log into the server (within **RepositoryThread**) successfully, the thread must register itself with **RepositoryServer** as shown by these updates to **RepositoryThread**:

CODE TO TYPE: /src/server.ipc/RepositoryThread.java

```
...
    if (validated) {
      RepositoryServer.register(user, this);

      // have handler manage the protocol until it decides it is done.
      while ((m = Parser.extractRequest(fromClient)) != null) {
        manager.updateAccessTime(user);
        Message response = handler.process(user, m);
        if (response == null) { break; }

        toClient.println(response.toString());
        if (toClient.checkError()) {
          break;
        }
      }
      RepositoryServer.unregister(user);
    }
...
```

To make this code compile, you have to add the user information to the **IProtocolHandler** interface. **ProtocolHandler** can use the **static** methods of **RepositoryServer** to update client state information. Let's make the necessary changes:

CODE TO TYPE: /src/server.ipc/IProtocolHandler.java

```
package server.ipc;

import xml.*;

public interface IProtocolHandler {
  /** Process the given Message request, return Message in reponse or null to terminate
 protocol. */
  Message process(String user, Message request);
}
```

Now consider the logic that will be contained within the **ProtocolHandler** class. If you're not careful,

ProtocolHandler will swallow all of the logic on the server! You want to fid ways to isolate and encapsulate the appropriate logic in an appropriate class. For starters, instead of burying logic within a complex **if** statement, introduce a **ServerAddImageController** and a **ServerStatusController** on the server side. Modify **ProtocolHandler** to pass in the name of the user on whose behalf the **ProtocolHandler** is executing:

CODE TO TYPE: /src/server/ProtocolHandler.java

```
package server;

~~import java.io.*;~~
import server.ipc.*;
import server.model.*;
~~import util.*;~~
import xml.*;
import org.w3c.dom.*;

public class ProtocolHandler implements IProtocolHandler {
  final Repository repository;
  public static final String endRequest = "</request>";

  public ProtocolHandler (Repository r) {
    repository = r;
  }

  ~~public static final String CorruptedImageData = "Encoded image data appears to be cor~~
~~rupted.";~~

  public Message process (String user, Message request) {
      Node child = request.contents.getFirstChild();
      if (child.getLocalName().equals ("addRequest")) {
        ~~String name = child.getAttributes().getNamedItem("name").getNodeValue();~~
        ~~Node imageNode = child.getFirstChild();~~

        ~~String xmlResp;~~
        ~~try {~~
          ~~byte[] bytes = ImageEncoding.decode(imageNode.getTextContent());~~
          ~~repository.add(bytes, name);~~
          ~~xmlResp = "<response success='true'><addResponse numBytes='" + bytes.length +~~
~~ "'/></response>";~~
        ~~} catch (IOException e) {~~
          ~~xmlResp = "<response success='false' reason='" + CorruptedImageData + "'>" +~~
                ~~"<addResponse numBytes='0'/></response>";~~
        ~~} catch (IllegalStateException e) {~~
          ~~xmlResp = "<response success='false' reason='" + Repository.AlreadyExistsImag~~
~~e + "'>" +~~
                ~~"<addResponse numBytes='0'></addResponse></response>";~~
        ~~}~~

        ~~return new Message(xmlResp);~~
        return new ServerAddImageController(repository).process(user, request);
      } else if (child.getLocalName().equals("statusRequest")) {
        ~~String xmlResp = "<response success='true'>" +~~
          ~~"<statusResponse key='SomeKey' index='1' total='" + repository.size() + "'/>"~~
~~ +~~
          ~~"</response>";~~
        ~~return new Message(xmlResp);~~
        return new ServerStatusController(repository).process(user, request);
      }

      return null;  // unknown request? No idea what to do.
    }
}
```

The logic for adding an image has been removed and will appear in the controller classes that you'll create so that the above code will compile.

In the **/src** folder **server** package, create a **ServerAddImageController** as shown:

CODE TO TYPE: /src/server/ServerAddImageController.java

```
package server;

import java.io.*;
import server.ipc.*;
import server.model.*;
import util.*;
import org.w3c.dom.*;
import xml.*;

public class ServerAddImageController implements IProtocolHandler {
  Repository repository;

  public ServerAddImageController(Repository repository) {
    this.repository = repository;
  }

  public Message process (String user, Message request) {
    Node child = request.contents.getFirstChild();
    String name = child.getAttributes().getNamedItem("name").getNodeValue();
    Node imageNode = child.getFirstChild();

    String xmlResp;
    try {
      byte[] bytes = ImageEncoding.decode(imageNode.getTextContent());
      repository.add (bytes, name);
      xmlResp = "<response success='true'><addResponse numBytes='" + bytes.length + "'/
></response>";
    } catch (IOException ioe) {
      xmlResp = "<response success='false' reason='" + CorruptedImageData + "'>" +
                "<addResponse numBytes='0'/></response>";
    } catch (Exception e) {
      xmlResp = "<response success='false' reason='" + e.getMessage() + "'>" +
                "<addResponse numBytes='0'/></response>";
    }

    return new Message(xmlResp);
  }

  public static final String CorruptedImageData = "Encoded image data appears to be cor
rupted.";
}
```

This code has been extracted from the old **ProtocolHandler** class and encapsulated into its own controller class. So, adding an image to the repository updates the repository itself, but which operations update the client's view? Let's appraoch this piece by piece. First, modify the **statusResponse** to include an image in the response. Update the XSD block for **statusResponse** as follows to support an optional image to be attached to each **statusResponse** message. The image is optional because of the **minOccurs** and **maxOccurs** values defined in our code:

CODE TO TYPE: /repository.xsd

```
<xs:element name='statusResponse'>
  <xs:complexType>
    <xs:sequence minOccurs="0" maxOccurs="1">
      <xs:element name='image'/>
    </xs:sequence>
    <xs:attribute name='key'   type='xs:string'  use='required'/>
    <xs:attribute name='index' type='xs:integer' use='required'/>
    <xs:attribute name='total' type='xs:integer' use='required'/>
  </xs:complexType>
</xs:element>
```

The client's current image key represents the image embedded with each status response. If the repository contains no images, then **key** will be an empty string.

In the **/src** folder **server** package, create a **ServerStatusController** to handle this logic, much like it was done for **addRequest**:

CODE TO TYPE: /src/server/ServerStatusController.java

```
package server;

import java.io.*;
import server.ipc.*;
import server.model.*;
import util.*;
import xml.*;

public class ServerStatusController implements IProtocolHandler {

  Repository repository;

  public ServerStatusController(Repository repository) {
    this.repository = repository;
  }

  public Message process(String user, Message request) {
    ClientState cs = RepositoryServer.getState(user);
    File f = repository.getImage(cs.getImageKey());
    String imageData = "";
    try {
      String encoding = ImageEncoding.encode(f);
      imageData = "<image>\n<![CDATA[" + encoding + "\n]]></image>";
    } catch (Exception e) {
      System.err.println ("Unable to encode image file:" + e.getMessage());
    }

    String xmlResp = "<response success='true'><statusResponse key='" + cs.getImageKey(
) + "' " +
        "index='" + repository.getOrder(cs.getImageKey()) + "' total='" + repository.si
ze() + "'>" +
        imageData + "</statusResponse></response>";
    return new Message(xmlResp);
  }
}
```

This code will work even when there are no images in the repository because **cs.getImageKey()** will return **null**.

To support this controller, you need to add some methods to **Repository** and **Index**. It may seem surprising to discover missing methods from key classes, but as you begin to exercise more and more of the desired functionality, you'll find that missing methods are actually pretty common. Modify the **Repository** class as shown to add a method to retrieve an image file by key, and one to return the index location for a given key:

CODE TO TYPE: /src/server.model/Repository.java

```
...
  public File getImage(String key) {
    if (key == null) { return null; }
    File f = new File (storage, key);
    if (f.exists()) {
      return f;
    }

    return null;
  }

  public int getOrder(String key) {
    return index.getOrder(key);
  }

...
```

The **getImage()** method returns the **File** object that represents that image on disk, or **null** if the image is non-existent. Given a key value, **getOrder** determines the image number in the set. To enable this code to compile you need to add a corresponding method to **Index**:

CODE TO TYPE: /src/server.model/Index.java

```
...
  public int getOrder(String key) {
    for (int i = 0; i < keys.size(); i++) {
      if (keys.get(i).equals(key)) {
        return (i+1);
      }
    }
    return -1;
  }

...
```

The **return (i+1)** adjusts the zero-based indexing scheme within **keys** to return a 1-based index value.

So how can you get a **statusRequest** sent to the server upon successful login? The **ImageRepositoryClient** must send a **statusRequest** once it has been connected properly. Right now, you don't want to bury this logic within the **ResponseHandler**. Instead, revise **ResponseHandler** to use a new **connected** method in **ImageRepositoryClient** that can initialize the client when a successful connection has been established. In essence, you're placing the right functionality in the right place:

CODE TO TYPE: /src/client/ResponseHandler.java

```
package client;

import client.gui.*;
import xml.*;

public class ResponseHandler implements IProcessResponse {

  ImageRepositoryClient client;

  public ResponseHandler (ImageRepositoryClient client) {
    this.client = client;
  }

  public void process(Message response) {
    String type = response.contents.getFirstChild().getLocalName();

    // handle loginResponse specially
    if (type.equals(Parser.loginResponse)) {
      boolean ok = Parser.success(response);
      if (!ok~~Parser.success(response)~~) {
       client.status("Unable to login:" + Parser.reason(response));
      } else {
       client.status("Connected to server.");
      }

      client.connected(ok);
      ~~client.validateMenuBar();~~
      return;
    }

    System.out.println("received:" + response);
  }
}
```

Here we eliminate the need for low-level knowledge of the GUI to leak out into other classes. Add the **connected()** method to **ImageRepositoryClient**, which validates the menu bar, but also fires off a **statusRequest** on a successful connection. If any new initialization code is needed upon successful (or failed) connections, this method will contain that logic:

CODE TO TYPE: /src/client.gui/ImageRepositoryClient

```
..
  public void connected(boolean ok) {
    validateMenuBar();

    if (ok) {
      String xmlStatusRequest = "<request><statusRequest/></request>";
      access.sendRequest(new StatusController(this), new xml.Message(xmlStatusRequest))
;
    }
  }
...
```

A **StatusController** object is constructed to process the request/response cycle in its entirety.

In the **/src** folder **client.gui** package, create the **StatusController** class as shown:

CODE TO TYPE: /src/client.gui/StatusController.java

```
package client.gui;

import java.io.*;
import java.awt.image.*;
import javax.imageio.*;
import org.w3c.dom.*;
import util.*;
import xml.*;
import client.*;

public class StatusController implements IController {

  ImageRepositoryClient client;

  public StatusController(ImageRepositoryClient client) {
    this.client = client;
  }

  public void process(Message request, Message response) {
    Node child = response.contents.getFirstChild();
    Node imageNode = child.getFirstChild();
    if (imageNode == null) {
      return;
    }

    try {
      byte[] bytes = ImageEncoding.decode(imageNode.getTextContent());

      InputStream in = new ByteArrayInputStream(bytes);
      BufferedImage image = ImageIO.read(in);
      client.display(image);
    } catch (IOException ioe) {
      client.status("Unable to decode image from server:" + ioe.getMessage());
    }
  }
}
```

In its **process()** method, the **StatusController** retrieves the image from the **statusResponse** message and requests **ImageRepositoryClient** to update its display. Add the **display()**method to **ImageRepositoryClient** to complete the task:

CODE TO TYPE: /src/client.gui/ImageRepositoryClient

```
...
  public void display(Image image) {
    if (image == null) {
      imgPanel.setViewportView(new JLabel(""));
    } else {
      ImageIcon icon = new ImageIcon(image);
      imgPanel.setViewportView(new JLabel(icon));
    }

    imgPanel.invalidate();
    imgPanel.validate();
    imgPanel.repaint();
  }

...
```

The **display()** method either clears or sets the viewport of the **JScrollPane** containing the current image for the client on the server.

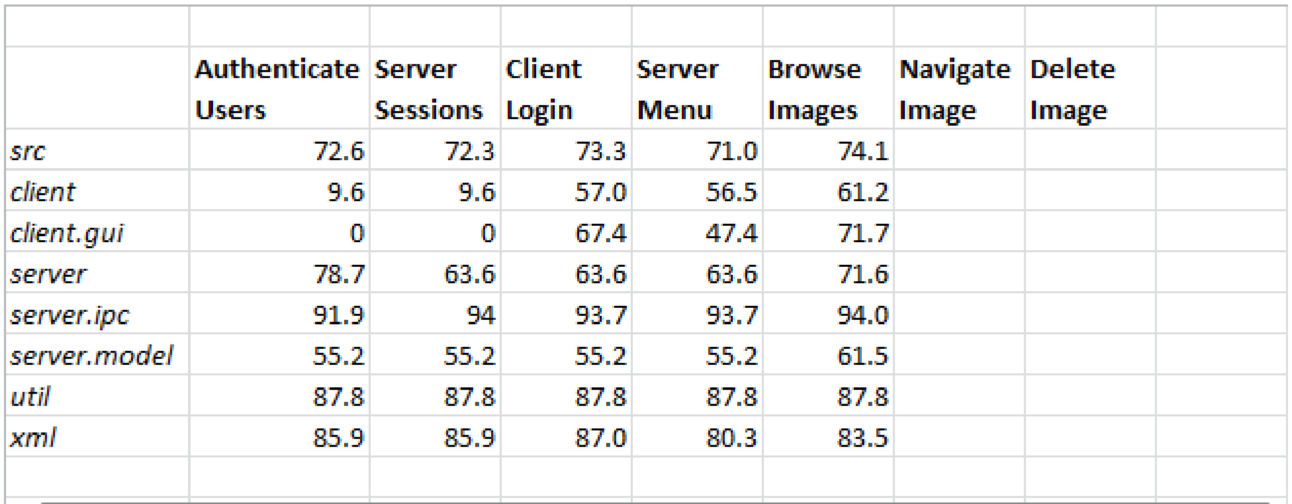
OBSERVE:

```
    imgPanel.invalidate();
    imgPanel.validate();
    imgPanel.repaint();
```

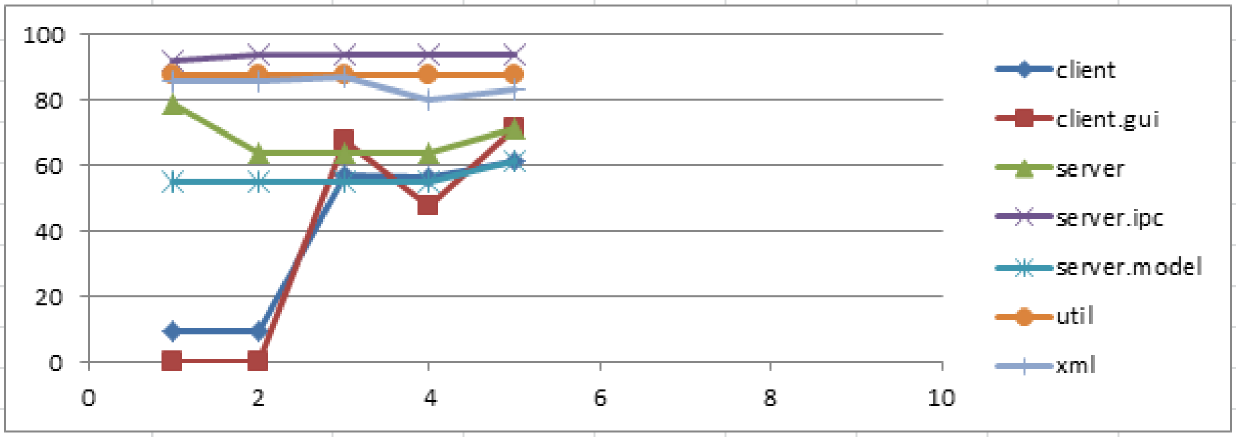
To display the image properly, the **three method invocations to invalidate, validate, and repaint** must be executed as shown. Doing so will ensure that the scrolling region is formatted to enclose the image completely, regardless of its size.

Phew! This lesson has been a marathon, but now we're ready to demonstrate the functionality we've got so far. First, delete all files in the **Repository** folder so you can start from scratch. Run **ServerLauncher** and then run **ClientLauncher**. Connect to the server on localhost, self-register an account, and add a file to the repository. Verify that the status information at the bottom of your client window has changed to confirm that the image was uploaded properly. Then, quit the client application (leave the server running if you like). Run **ClientLauncher** and connect again, either with a new self-registered account or the one that you registered in the previous run. The newly added image appears in the client's image panel.

Terminate both the **ClientLauncher** and **ServerLauncher**. Now run all test cases in coverage to determine your status. We need to attend to writing test cases for the client—this is not surprising given the extensive changes we made! Even so, you've got increasing coverage of your project code without excessive effort, because you've been taking steps along the way to write your test cases. Good thinking!



	Authenticate Users	Server Sessions	Client Login	Server Menu	Browse Images	Navigate Image	Delete Image
src	72.6	72.3	73.3	71.0	74.1		
client	9.6	9.6	57.0	56.5	61.2		
client.gui	0	0	67.4	47.4	71.7		
server	78.7	63.6	63.6	63.6	71.6		
server.ipc	91.9	94	93.7	93.7	94.0		
server.model	55.2	55.2	55.2	55.2	61.5		
util	87.8	87.8	87.8	87.8	87.8		
xml	85.9	85.9	87.0	80.3	83.5		



Navigating Repository Images

Lesson Objectives

In this lesson you will:

- design a customizable XML message and properly synchronize the state of the server and client.

Navigating Images in the Repository

In the last lab, you developed the capability to add an image to the repository and display the user's current image in the **ImageRepositoryClient** window, but the user still can't navigate through the images. We need to allow the user to navigate through the repository, so we'll add these controls: Next, Previous, First, and Last. You can get your navigation tools in place using a single XML message. Modify **repository.xsd** to include a **navigateRequest**:

CODE TO TYPE: /repository.xsd

```
<?xml version='1.0' encoding='UTF-8'?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>

<xs:element name='message'>
  <xs:complexType>
    <xs:choice>
      <xs:element ref='response'/>
      <xs:element ref='request'/>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name='response'>
  <xs:complexType>
    <xs:choice>
      <xs:element ref='addResponse'/>
      <xs:element ref='statusResponse'/>
      <xs:element ref='loginResponse'/>
    </xs:choice>
    <xs:attribute name='success' type='xs:boolean' use='required'/>
    <xs:attribute name='reason'  type='xs:string'  use='optional'/>
  </xs:complexType>
</xs:element>

<xs:element name='request'>
  <xs:complexType>
    <xs:choice>
      <xs:element ref='addRequest'/>
      <xs:element ref='statusRequest'/>
      <xs:element ref='loginRequest'/>
      <xs:element ref='navigateRequest'/>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name='addRequest'>
  <xs:complexType>
   <xs:sequence>
     <xs:element name='image'/>
   </xs:sequence>
   <xs:attribute name='name' type='xs:string' use='required'/>
  </xs:complexType>
</xs:element>

<xs:element name='addResponse'>
  <xs:complexType>
    <xs:attribute name='numBytes' type='xs:integer' use='required'/>
  </xs:complexType>
</xs:element>

<xs:element name='statusRequest'/>

<xs:element name='statusResponse'>
  <xs:complexType>
    <xs:sequence minOccurs="0" maxOccurs="1">
     <xs:element name='image'/>
    </xs:sequence>
    <xs:attribute name='key'   type='xs:string'  use='required'/>
    <xs:attribute name='index' type='xs:integer' use='required'/>
    <xs:attribute name='total' type='xs:integer' use='required'/>
  </xs:complexType>
</xs:element>

<xs:element name='loginRequest'>
  <xs:complexType>
   <xs:attribute name='user'     type='xs:string'  use='required'/>
   <xs:attribute name='password' type='xs:string'  use='required'/>
   <xs:attribute name='register' type='xs:boolean' use='optional'/>
  </xs:complexType>
</xs:element>

<xs:element name='loginResponse'>
  <xs:complexType>
    <xs:attribute name='user' type='xs:string' use='required'/>
  </xs:complexType>
</xs:element>

<xs:simpleType name='directionType'>
  <xs:restriction base='xs:string'>
    <xs:pattern value='next|previous|first|last'/>
  </xs:restriction>
</xs:simpleType>

<xs:element name='navigateRequest'>
  <xs:complexType>
    <xs:attribute name='direction' type='directionType' use='required'/>
  </xs:complexType>
</xs:element>
</xs:schema>
```

Whenever you make changes to **repository.xsd**, you'll want to add attributes to **Parser** to be used in your code. Go ahead and add these attributes to **Parser** now:

CODE TO TYPE: /src/xml/Parser.java

```
...
  public final static String statusResponse      = "statusResponse";

  public final static String direction           = "direction";
  public final static String first               = "first";
  public final static String previous            = "previous";
  public final static String next                = "next";
  public final static String last                = "last";
```

To activate this logic, you need to modify **ImageRepositoryClient** to add some new menu items that support the navigation. Update the **initMenuBar** method of **ImageRepositoryClient** as shown:

CODE TO TYPE: /src/client.gui/ImageRepositoryClient.java

```
...
import java.awt.*;
import javax.swing.*;
import javax.swing.GroupLayout.Alignment;
import java.awt.event.*;
import client.*;
import xml.*;
...
  void initMenuBar() {
    ...
    image = new JMenu ("Image");
    JMenuItem add = new JMenuItem("Add...");
    add.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        new AddImageController(ImageRepositoryClient.this).add();
      }
    });
    image.add(add);
    JMenuItem first = new JMenuItem("First");
    first.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        new NavigateController(ImageRepositoryClient.this).go(Parser.first);
      }
    });
    image.add(first);
    JMenuItem previous = new JMenuItem("Previous");
    previous.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        new NavigateController(ImageRepositoryClient.this).go(Parser.previous);
      }
    });
    image.add(previous);
    JMenuItem next = new JMenuItem("Next");
    next.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        new NavigateController(ImageRepositoryClient.this).go(Parser.next);
      }
    });
    image.add(next);
    JMenuItem last = new JMenuItem("Last");
    last.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        new NavigateController(ImageRepositoryClient.this).go(Parser.last);
      }
    });
    image.add(last);
    mb.add(image);
...
```

These changes invoke a **NavigateController** to go to the first, previous, next, or last image in the repository.

In the **/src** folder **client.gui** package, create a **NavigateController** class as shown:

CODE TO TYPE: /src/client.gui/NavigateController.java

```
package client.gui;

import xml.*;

public class NavigateController {

  ImageRepositoryClient client;

  public NavigateController(ImageRepositoryClient client) {
    this.client = client;
  }

  boolean go (String direction) {
    String xmlNavRequest = "<request><navigateRequest direction='" + direction + "'/></
request>";
    return client.access.sendRequest(new Message(xmlNavRequest));
  }
}
```

The **NavigateController** takes one of the directions (first, previous, next, or last) and constructs an appropriate **navigateRequest** for the server. The idea is to have the server respond with a **statusResponse** so there is no need to register a waiting controller; you can use the existing **sendRequest** method to send the request to the server. On the server side you'll need to modify **ProtocolHandler** to process the **navigateRequest** as follows (while you're at it, remove the **endRequest** attribute; we don't need it anymore):

CODE TO TYPE: /src/server/ProtocolHandler.java

```
package server;

import server.ipc.*;
import server.model.*;
import xml.*;
import org.w3c.dom.*;

public class ProtocolHandler implements IProtocolHandler {
  final Repository repository;
  ~~public static final String endRequest = "</request>";~~

  public ProtocolHandler (Repository r) {
    repository = r;
  }

  public Message process (String user, Message request) {
    Node child = request.contents.getFirstChild();
    if (child.getLocalName().equals ("addRequest")) {
      return new ServerAddImageController(repository).process(user, request);
    } else if (child.getLocalName().equals("statusRequest")) {
      return new ServerStatusController(repository).process(user, request);
    } else if (child.getLocalName().equals("navigateRequest")) {
      return new ServerNavigateController(repository).process(user, request);
    }

    return null;  // unknown request? No idea what to do.
  }
}
```

In the **/src** folder **server** package, create a **ServerNavigateController** class. This controller determines which image is viewed by the client and adjusts accordingly. Finally, the controller activates the **ServerStatusController** which stimulates a **statusResponse** to be sent to the client. Let's write this class in stages. Start with the skeleton code below:

CODE TO TYPE: /src/server/ServerNavigateController.java

```
package server;

import server.ipc.*;
import server.model.*;
import xml.*;
import org.w3c.dom.*;

public class ServerNavigateController implements IProtocolHandler {

  Repository repository;

  public ServerNavigateController(Repository repository) {
    this.repository = repository;
  }

  public Message process(String user, Message request) {
    ClientState cs = RepositoryServer.getState(user);
    String key = cs.getImageKey();
    int num = repository.getOrder(key);

    NamedNodeMap map = request.contents.getFirstChild().getAttributes();
    String direction = map.getNamedItem(Parser.direction).getNodeValue();

    // To Add...

    // return a status response.
    return new ServerStatusController(repository).process(user, request);
  }
}
```

Let's take a closer look:

OBSERVE:

```
  public Message process(String user, Message request) {
    ClientState cs = RepositoryServer.getState(user);
    String key = cs.getImageKey();
    int num = repository.getOrder(key);

    NamedNodeMap map = request.contents.getFirstChild().getAttributes();
    String direction = map.getNamedItem(Parser.direction).getNodeValue();

    // To Add...

    // return a status response.
    return new ServerStatusController(repository).process(user, request);
```

The **ServerNavigateController** uses the **getState()** method of **RepositoryServer** to determine the state for the given **user**. From the state, you can get the **key**; using the key you can determine the **num** ordinal position of that client (a number from 1 to the size of the repository). From the **request**, you can see how the requested **direction** is extracted. Finally, the existing **ServerStatusController** returns the response **Message** to the client, which will contain the user's current status.

Now, fill in the details of the logic to manipulate the user's state based on the desired **direction** that's included in the **navigateRequest**:

CODE TO TYPE: /src/server/ServerNavigateController.java

```
package server;

import server.ipc.*;
import server.model.*;
import xml.*;

public class ServerNavigateController implements IProtocolHandler {

  Repository repository;

  public ServerNavigateController(Repository repository) {
    this.repository = repository;
  }

  public Message process(String user, Message request) {
    ClientState cs = RepositoryServer.getState(user);
    String key = cs.getImageKey();
    int num = repository.getOrder(key);

    NamedNodeMap map = request.contents.getFirstChild().getAttributes();
    String direction = map.getNamedItem(Parser.direction).getNodeValue();

    ~~// To Add...~~
    if (direction.equals(Parser.first)) {
      num = 1;
    } else if (direction.equals(Parser.previous) && num > 1) {
      num--;
    } else if (direction.equals(Parser.last)) {
      num = repository.size();
    } else if (direction.equals(Parser.next) && num < repository.size()) {
      num++;
    }
    key = repository.getNthKey(num);
    cs.setImageKey(key);

    // return a status response.
    return new ServerStatusController(repository).process(user, request);
  }
}
```

For each of the four cases, **num** is updated accordingly. Next, the **key** for the updated image number is retrieved from the repository using **getNthKey** and you call **setImageKey** to record this information with the client state. Now when the **ServerStatusController** executes, the correct key is included in the **statusResponse**.

We still have one last item to address. The client isn't ready to receive a **statusResponse** message unsolicited from the server. You may recall that the client processes all received messages via a **ResponseHandler** class, which currently processes only the **loginResponse** message. You need to update this class to handle **statusResponse** messages as well. Modify your code as shown:

CODE TO TYPE: /src/client/ResponseHandler.java

```
package client;

import client.gui.*;
import xml.*;

public class ResponseHandler implements IProcessResponse {

  ImageRepositoryClient client;

  public ResponseHandler (ImageRepositoryClient client) {
    this.client = client;
  }

  public void process(Message response) {
    String type = response.contents.getFirstChild().getLocalName();

    // handle loginResponse specially
    if (type.equals(Parser.loginResponse)) {
      boolean ok = Parser.success(response);
      if (!ok) {
        client.status("Unable to login:" + Parser.reason(response));
      } else {
        client.status("Connected to server.");
      }

      client.connected(ok);
      return;
    } else if (type.equals(Parser.statusResponse)) {
      new StatusController(client).process(null, response);
      return;
    }

    System.out.println("received:" + response);
  }
}
```

There was no **statusRequest** object known on the client side, so the invocation to **process** added above takes **null** as its first parameter; the invocation simply asks the client to process the received **statusResponse** message.

Could it really be that straightforward? Well, execute **ServerLauncher** and then execute **ClientLauncher**. Connect to localhost and self-register a new account. Once connected, you'll see the first image in the repository on the client display. Select the **Image | Add** menu item several times to populate the repository. Then begin browsing through the images using the menu item controls. Note that each time you navigate, the new image appears on the client display. You can even execute another instance of **ClientLauncher** and have two separate clients, each navigating through the repository, each adding images, and this can occur simultaneously!

There are a few items that still need our attention:

1. If you connect to the server (not self-registered) and you enter invalid credentials, the **Server** menu bar still appears as if the client had properly connected.
2. Disconnect functionality is not yet implemented.
3. The status bar at the bottom of the screen doesn't change during navigation. (Shouldn't it show which picture in the repository is being viewed? It's time to get that metadata properly displayed on the screen).
4. When adding an image to the repository, that image should become the last one in the repository, but shouldn't the client navigate to that last picture and show it on the client display?

Let's complete these tasks in order. To validate that the client behaves improperly, run **ServerLauncher** and then run **ClientLauncher** and try to connect to **localhost** with new account information. While the status bar at the bottom of the screen shows "Unable to login:Invalid credentials", the **Server** menu shows that it believes the connection was appropriate. So where do you begin to solve this problem? Well, you can look at the **ResponseHandler** because its **process()** method is the one that posts the "invalid credentials" status message, but that method only calls the **connected** method. Review the changes to this method below, which calls **validateMenuBar** after the method configures **access**:

CODE TO TYPE: /src/client.gui/ImageRepositoryClient.java

```
...
public void connected(boolean ok) {
~~  validateMenuBar();~~

  if (ok) {
    String xmlStatusRequest = "<request><statusRequest/></request>";
    access.sendRequest(new StatusController(this), new xml.Message(xmlStatusRequest));
  } else {
    access = null;
  }

  validateMenuBar();
}
...
```

If **access** is set to **null** on failed login attempts, the **validateMenuBar()** method is properly able to adjust to demonstrate whether the client is connected or not. Close down all applications and restart the **ServerLauncher** and **ClientLauncher**. This time note that after failed login attempts, the menu bar is properly updated.

Moving on to the **Server | Disconnect** menu item, you need to write a **DisconnectController** to oversee this functionality. Start at **ImageRepositoryClient** and make these modifications to **initMenuBar()**:

CODE TO TYPE: /src/client.gui/ImageRepositoryClient.java

```
...
  disconnect = new JMenuItem("Disconnect...");
  disconnect.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        new DisconnectController(ImageRepositoryClient.this).confirm();
      }
  });
  server.add(disconnect);
...
```

This new code invokes **DisconnectController** once selected.

In the **/src** folder **client.gui** package, create a **DisconnectController** class as shown:

/src/client.gui/DisconnectController.java

```
package client.gui;

import javax.swing.*;
import util.*;

public class DisconnectController {
  static String property_confirmOnDisconnect = "ConfirmOnDisconnect";
  static String imageFile = "images/help_32.png";
  static ImageIcon icon;

  ImageRepositoryClient client;

  public DisconnectController(ImageRepositoryClient client) {
    this.client = client;
  }

  public boolean confirm() {
    if (icon == null) {
      icon = new ImageIcon(imageFile);
    }
    if (!Preferences.isTrue(property_confirmOnDisconnect)) {
      String[] choices = { "Confirm", "Confirm and don't ask me again" };

      String s = (String) JOptionPane.showInputDialog (client,
          "Do you wish to disconnect from " + client.access.getHost() + "?\n ",
          "Confirm Disconnect", JOptionPane.PLAIN_MESSAGE,
          icon, choices, choices[0]);
      if (s == null) {
        return false;
      } else if (s.equals (choices[1])) {
        // remember this in the future.
        Preferences.set(property_confirmOnDisconnect, true);
      }
    }

    return disconnect();
  }

  boolean disconnect() {
    String host = client.access.getHost();
    client.access.disconnect();
    client.connected(false);

    client.status("Disconnected from " + host);
    return true;
  }
}
```

The above code may seem familiar; it is nearly identical to **QuitController**. To protect the user from inadvertently "disconnecting," you included the same confirmation step as in the **QuitController**. To disconnect the client, disconnect the **ServerAccess** object and tell **ImageRepositoryClient** that it is no longer connected. For this code to compile, add this method to **ServerAccess**:

CODE TO TYPE: /src/client/ServerAccess.java

```
  public String getHost() {
    return host;
  }
```

Test the capability by terminating all existing applications running in Eclipse. Npw execute **ServerLauncher** and **ClientLauncher**. Once you've connected, select **Server | Disconnect**; you'll be asked to confirm the disconnect request. If you choose to disconnect, the menu bar is revalidated to allow you to reconnect later.

The next task to deal with is that the screen is not updated after the client uploads a new image to the repository. The next few changes to **ServerAddImageController** show how to update the **ClientState** for the user adding images:

CODE TO TYPE: /src/server/ServerAddImageController.java

```
package server;

import java.io.*;
import server.ipc.*;
import server.model.*;
import util.*;
import org.w3c.dom.*;
import xml.*;

public class ServerAddImageController implements IProtocolHandler {
  Repository repository;

  public ServerAddImageController(Repository repository) {
    this.repository = repository;
  }

  public Message process (String user, Message request) {
    Node child = request.contents.getFirstChild();
    String name = child.getAttributes().getNamedItem("name").getNodeValue();
    Node imageNode = child.getFirstChild();

    String xmlResp;
    try {
      byte[] bytes = ImageEncoding.decode(imageNode.getTextContent());
      String key = repository.add (bytes, name);
      ClientState cs = RepositoryServer.getState(user);
      cs.setImageKey(key);
      xmlResp = "<response success='true'><addResponse numBytes='" + bytes.length + "'/
></response>";
    } catch (IOException ioe) {
      xmlResp = "<response success='false' reason='" + CorruptedImageData + "'>" +
                "<addResponse numBytes='0'/></response>";
    } catch (Exception e) {
      xmlResp = "<response success='false' reason='" + e.getMessage() + "'>" +
                "<addResponse numBytes='0'/></response>";
    }

    return new Message(xmlResp);
  }

  public static final String CorruptedImageData = "Encoded image data appears to be cor
rupted.";
}
```

There is no room in the **addResponse** message to send back the image, so how can the client see the uploaded image? Remember how on the client side, you detect the successful image addition, so at that point you can send a **statusRequest** to the server to return the current image? Modify the **process** method of the client-side **AddImageController** as follows:

CODE TO TYPE: /src/client.gui/AddImageController.java

```
...
  public void process (Message request, Message response) {
    NamedNodeMap map = request.contents.getFirstChild().getAttributes();
    String name = map.getNamedItem("name").getNodeValue();

    if (Parser.success(response)) {
      client.status("Image uploaded to server:" + name);

      String xmlStatusRequest = "<request><statusRequest/></request>";
      client.access.sendRequest(new StatusController(client), new Message(xmlStatusRequ
est));
    } else {
      client.status("Problem adding image:" + name + "(" + Parser.reason(response) + ")
");
    }
  }
...
```

Pretty cool! Now, as the client adds images, the most recently added image appears. Try this out. Delete all files in the **/Repository** folder and relaunch **ServerLauncher** and **ClientLauncher**. Observe the changed behavior as you add new images to the repository. All of the scaffolding and carefully designed controllers are now building blocks that you can use to satisfy the application requirements.

You still have one more task to perform. Update the status at the bottom of the client display during navigation:

CODE TO TYPE: /src/client.gui/StatusController.java

```
package client.gui;

import java.io.*;
import java.awt.image.*;
import javax.imageio.*;
import org.w3c.dom.*;
import util.*;
import xml.*;
import client.*;

public class StatusController implements IController {

  ImageRepositoryClient client;

  public StatusController(ImageRepositoryClient client) {
    this.client = client;
  }

  public void process(Message request, Message response) {
    Node child = response.contents.getFirstChild();
    Node imageNode = child.getFirstChild();
    if (imageNode == null) {
      client.status("Repository has no images.");
      return;
    }

    int idx = Integer.valueOf(child.getAttributes().getNamedItem("index").getNodeValue(
));
    int total = Integer.valueOf(child.getAttributes().getNamedItem("total").getNodeValu
e());
    try {
      byte[] bytes = ImageEncoding.decode(imageNode.getTextContent());

      InputStream in = new ByteArrayInputStream(bytes);
      BufferedImage image = ImageIO.read(in);
      client.display(image);
      client.status("Image " + idx + " of " + total);
    } catch (IOException ioe) {
      client.status("Unable to decode image from server:" + ioe.getMessage());
    }
  }
}
```

Now when you run the application, you'll see that the status bar information is updated as you navigate among the images in the repository. Once again, having clearly defined controllers means you can identify quickly where you need to make small adjustments in your code, whether during normal development or as requirements change.

Delete the "temporary" classes from the default package—those are no longer part of your project. Now rerun the code coverage on all test cases and confirm that the following classes have ZERO code coverage from any test case. (In the next lab you must make progress towards closing the coverage gap):

OBSERVE:

```
Client
ClientLauncher
SplashScreenLogic
ConnectController
DisconnectController
NavigateController
QuitController
Server
ServerNavigateController
ServerLauncher
```

Let's write some test cases for **NavigateController**. Pattern them after the **TestAddBehavior** test case. Your new test case will extend **TestAddBehavior** to take immediate advantage of its **setUp** and **tearDown** methods, as well as its attributes which are now inherited by **TestNavigationSequence**.

In the **/test** folder **server.ipc** package, create a **TestNavigationSequence** class that extends **TestAddBehavior**, as shown. While typing the code, follow the sequence of actions in the test case method:

CODE TO TYPE: /test/server.ipc/TestNavigationSequence.java

```
package server.ipc;

import java.io.*;
import org.w3c.dom.*;
import xml.*;
import server.*;

public class TestNavigationSequence extends TestAddBehavior {

  static Message requestNAVIGATE (String direction) {
    String xmlNavRequest = "<request><navigateRequest direction='" + direction + "'/></
request>";
    return new Message(xmlNavRequest);
  }

  public void testBriefNavigationSequence() throws Exception {
    String first = "c00bc1ed28fabdbcebc3e4735decc83e";
    String last = "6e3a233232c4c8e0c8bb1c163aa48d9d";
    String user = "user00";

    toServer.println(requestLOGIN(user, "n", true));
    expectSuccess(fromServer);

    // add
    File f = new File("images", "repositorySplash.png");
    toServer.println(requestADD("sampleImage", f));
    expectSuccess(fromServer);

    // verify that client has this first image.
    ClientState state = RepositoryServer.getState(user);
    String key = state.getImageKey();
    assertEquals (first, key);

    // add
    f = new File("images", "help_32.png");
    toServer.println(requestADD("help", f));
    expectSuccess(fromServer);

    // now on second one
    key = state.getImageKey();
    assertEquals (last, key);

    // now navigate to the first
    ServerNavigateController nc = new ServerNavigateController(RepositoryServer.reposit
ory);
    nc.process(user, requestNAVIGATE(Parser.first));
    assertEquals (first, state.getImageKey());

    // validate StatusController works on first
    ServerStatusController sc = new ServerStatusController(RepositoryServer.repository)
;
    Message resp = sc.process(user, TestAddBehavior.requestSTATUS());
    NamedNodeMap map = resp.contents.getFirstChild().getAttributes();
    assertEquals (first, map.getNamedItem("key").getNodeValue());

    // go last
    nc.process(user, requestNAVIGATE(Parser.last));
    assertEquals (last, state.getImageKey());

    // validate StatusController works on first
    sc = new ServerStatusController(RepositoryServer.repository);
    resp = sc.process(user, TestAddBehavior.requestSTATUS());
    map = resp.contents.getFirstChild().getAttributes();
    assertEquals (last, map.getNamedItem("key").getNodeValue());

    // go previous
    nc.process(user, requestNAVIGATE(Parser.previous));
    assertEquals (first, state.getImageKey());

    // go next
    nc.process(user, requestNAVIGATE(Parser.next));
    assertEquals (last, state.getImageKey());
  }
}
```

This test case logic may be familiar. It's a lot to type in all at once, but you can probably "read" the scenario it describes where a client logs in, adds two images, and then navigates among the images in the repository. By extending **TestAddBehavior**, this test case can take advantage of the inherited attributes and methods from that class, as well as the **setUp** and **tearDown** methods for starting and stopping the server and client. Once again, you demonstrate how to compose new functionality from the composition of existing classes. This test case demonstrates the capability on the server side.

In this lab you wrote a number of capabilities on the client side; now demonstrate their effectiveness.

In the **/test** folder **client.gui** package, create a **TestConnection** test case as shown:

CODE TO TYPE: /test/client.gui/TestConnection.java

```
package client.gui;

import server.ipc.*;
import junit.framework.TestCase;

public class TestConnection extends TestCase {
  RepositoryServer server;

  protected void setUp() throws Exception {
    TestServer.clearTestRepository();
    server = TestServer.launchServer();
  }

  protected void tearDown() throws Exception {
    RepositoryServerAccess.shutdown(server);
    server = null;
  }

  public void testConnection() throws Exception {
    ImageRepositoryClient client = new ImageRepositoryClient();

    ConnectController cc = new ConnectController();
    cc.connect(client, "localhost", "tester", "anything", true);

    assertEquals ("localhost", client.access.getHost());

    DisconnectController dc = new DisconnectController(client);
    assertTrue (dc.disconnect());

    assertTrue (client.access == null);
  }
}
```

This test case executes the **ConnectController** and **DisconnectController**. You've already seen the scenario being tested, because you put the building blocks into place.

The **testConnection** test case method creates a new **ImageRepositoryClient** object, and uses the **ConnectController** to attempt to connect that client to the local **RepositoryServer** using a self-registered account for user "tester" with hashedPassword of "anything." After confirming the connection (this test case method uses the **getHost()** method we added earlier in this lab), the client is instructed to disconnect, using **DisconnectController**; you know that **disconnect** succeeds because **client.access** is reset to **null**.

For the final test case, you'll validate the client-side navigation capabilities.

In the **/test** folder **client.gui** package, create a **TestClientNavigation** test case. This test case class introduces a few advanced testing techniques. Let's take them on one step at a time:

CODE TO TYPE: /test/client.gui/TestClientNavigation.java

```
package client.gui;

import java.io.*;
import server.*;
import server.ipc.*;
import xml.*;
import junit.framework.TestCase;

public class TestClientNavigation extends TestCase {
  RepositoryServer server;
  static final String user = "tester";

  protected void setUp() throws Exception {
    TestServer.clearTestRepository();
    server = TestServer.launchServer();
  }

  protected void tearDown() throws Exception {
    RepositoryServerAccess.shutdown(server);
    server = null;
  }

  public static boolean waitForResponse(ImageRepositoryClient client) throws Exception
{
    // wait until all responses have been received before continuing test case
    int ctr = 50;
    while (client.access.isWaiting() && ctr-- > 0) {
      Thread.sleep(200);
    }
    return !client.access.isWaiting();
  }

  public static boolean waitUntilKeySet(ClientState state, String target) throws Except
ion {
    int ctr = 50;
    while (!target.equals(state.getImageKey()) && ctr-- > 0) {
      Thread.sleep(200);
    }
    return target.equals(state.getImageKey());
  }
}
```

Let's take a closer look:

OBSERVE:

```
  public static boolean waitForResponse(ImageRepositoryClient client) throws Exception
{
    // wait until all responses have been received before continuing test case
    int ctr = 50;
    while (client.access.isWaiting() && ctr-- > 0) {
      Thread.sleep(200);
    }
    return !client.access.isWaiting();
  }

  public static boolean waitUntilKeySet(ClientState state, String target) throws Except
ion {
    int ctr = 50;
    while (!target.equals(state.getImageKey()) && ctr-- > 0) {
      Thread.sleep(200);
    }
    return target.equals(state.getImageKey());
  }
```

This test case involves a client and the server. Without making additional changes to your code, it's hard to determine when to check to see whether a request sent by the client has been processed by the server properly. If a controller sends a request to the server, and expects to receive the response, then the **ServerAccess** object maintains the registered controller until it can process the response. The **waitForResponse()** method allows you to wait (in a non-blocking fashion) until **client.access.isWaiting()** is **false**. To avoid becoming stuck in an infinite loop, this method will check once every 200 milliseconds, until 10 seconds have elapsed, to determine whether **client.access** is still waiting. The method returns **true** when the client is no longer waiting, or **false** when something has gone wrong and the client has not received a response within 10 seconds.

Similarly, the **waitUntilKeySet()** method waits until the given **ClientState** has its image key changed to **target** (within a total elapsed time of 10 seconds). This method is used for requests that are not expecting a returned response to a registered controller.

Is **waitForResponse()** more complicated than it needs to be? Maybe. Of course, you would like to write a simpler method that stays in the **while** loop until the client is no longer waiting, like this:

OBSERVE:

```
  public static boolean waitForResponse(ImageRepositoryClient client) {
    while (client.access.isWaiting());
    return !client.access.isWaiting();
  }
```

This won't work though, because the **ServerAccess** thread and the JUnit thread executing the **waitForResponse** method will clash. Don't believe me? After you've completed the test case, modify the **waitForResponse** as shown above and see what happens.

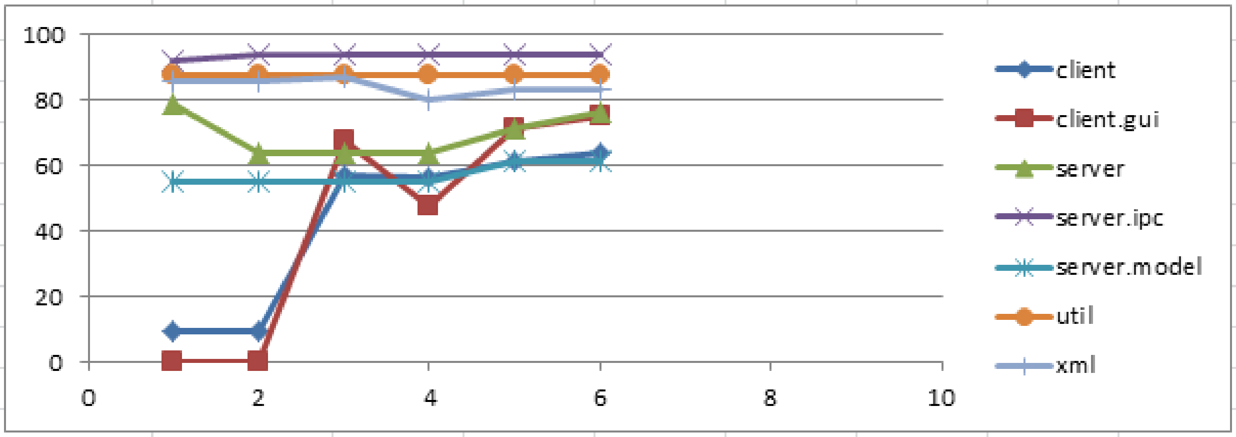
Now add this test case method to **TestClientNavigation**:

CODE TO TYPE: /test/client.gui/TestClientNavigation.java

```
...
  public void testNavigation() throws Exception {
    String first = "c00bc1ed28fabdbcebc3e4735decc83e";
    String last = "6e3a233232c4c8e0c8bb1c163aa48d9d";

    ImageRepositoryClient client = new ImageRepositoryClient();

    // connect and add two images.
    ConnectController cc = new ConnectController();
    cc.connect(client, "localhost", user, "anything", true);

    AddImageController c = new AddImageController(client);
    assertTrue (c.add(new File ("images/repositorySplash.png")));
    assertTrue (waitForResponse(client));
    assertEquals (1, RepositoryServerAccess.size(server));

    assertTrue (c.add(new File ("images/help_32.png")));
    assertTrue (waitForResponse(client));
    assertEquals (2, RepositoryServerAccess.size(server));

    // make some navigation requests from the client. Go to the first one in repository
.
    // Note that navigation sends back Status messages so this controller is not one
    // that "waits" for a response from the server. Instead we wait for state to change
    NavigateController nc = new NavigateController(client);
    assertTrue (nc.go(Parser.first));

    ClientState state = RepositoryServer.getState(user);
    assertTrue(waitUntilKeySet(state, first));

    assertTrue (nc.go(Parser.last));
    assertTrue(waitUntilKeySet(state, last));

    DisconnectController dc = new DisconnectController(client);
    assertTrue (dc.disconnect());
    assertTrue (client.access == null);
  }
```

Once this test case method is completed, launch all test cases in the **/test** source folder to validate that they all pass. Next, generate code coverage using EclEmmma. Some classes, such as **ClientLauncher** and **ServerLauncher**, still have no automatic coverage because they are just launching code that begins the client and server applications, respectively. In Eclipse, there is no easy way to write a JUnit test case that uses parameters to the Java VM to execute the **SplashScreenLogic**. We have to stick to our basic testing principles that tell us that classes with coverage that is below 80% must be inspected manually.

	Authenticate Users	Server Sessions	Client Login	Server Menu	Browse Images	Navigate Image	Delete Image
src	72.6	72.3	73.3	71.0	74.1	75.2	
client	9.6	9.6	57.0	56.5	61.2	63.6	
client.gui	0	0	67.4	47.4	71.7	75	
server	78.7	63.6	63.6	63.6	71.6	76.2	
server.ipc	91.9	94	93.7	93.7	94.0	94	
server.model	55.2	55.2	55.2	55.2	61.5	61.5	
util	87.8	87.8	87.8	87.8	87.8	87.8	
xml	85.9	85.9	87.0	80.3	83.5	83.5	



Deleting Repository Images

Lesson Objectives

In this lesson you will:

- use an authentication policy to allow or disallow remote functionality.

Deleting Images in the Repository

You've made it to the final lesson of Java 5. Nice going! Now you'll implement the functionality that allows users to delete images in the repository and complete the core functionality of the application. Given the structure of the client/server approach, you may have already guessed that we'll be adding some new XML messages and writing client and server controllers. Before we launch into this task, we'll need to define a policy that controls the deletion of images:

1. Users should not be able to delete images that they themselves did not upload. You will identify additional metadata that can be stored with each image to support this functionality.
2. Users should not be able to delete an image that is currently being viewed by another user.

If we stick to this defined policy, we improve our users' experience. To request the deletion of an image, a user must be looking at that image currently and then select **Image | Delete**, which is a new menu item added to **ImageRepositoryClient**. Make these change to the **initMenuBar()** method of **ImageRepositoryClient**:

CODE TO TYPE: /src/client.gui/ImageRepositoryClient.java

```
...
    image = new JMenu ("Image");
    JMenuItem add = new JMenuItem("Add...");
    add.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        new AddImageController(ImageRepositoryClient.this).add();
      }
    });
    image.add(add);
    JMenuItem delete = new JMenuItem("Delete...");
    delete.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        new DeleteImageController(ImageRepositoryClient.this).delete();
      }
    });
    image.add(delete);
    image.add(new JSeparator());

    JMenuItem first = new JMenuItem("First");
    first.addActionListener(new ActionListener() {
...
```

The XML **deleteRequest** coming from the client doesn't need any information, because the server will determine the image to delete from the requesting user's **ClientState**. The **deleteResponse** should contain the name of the file (not the unreadable key) to tell the user that the operation succeeded. Make these changes to **repository.xsd**:

CODE TO TYPE: /repository.xsd

```
<?xml version='1.0' encoding='UTF-8'?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>

<xs:element name='message'>
  <xs:complexType>
    <xs:choice>
      <xs:element ref='response'/>
      <xs:element ref='request'/>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name='response'>
  <xs:complexType>
    <xs:choice>
      <xs:element ref='addResponse'/>
      <xs:element ref='statusResponse'/>
      <xs:element ref='loginResponse'/>
      <xs:element ref='deleteResponse'/>
    </xs:choice>
    <xs:attribute name='success' type='xs:boolean' use='required'/>
    <xs:attribute name='reason'  type='xs:string'  use='optional'/>
  </xs:complexType>
</xs:element>

<xs:element name='request'>
  <xs:complexType>
    <xs:choice>
      <xs:element ref='addRequest'/>
      <xs:element ref='statusRequest'/>
      <xs:element ref='loginRequest'/>
      <xs:element ref='navigateRequest'/>
      <xs:element ref='deleteRequest'/>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name='addRequest'>
  <xs:complexType>
   <xs:sequence>
     <xs:element name='image'/>
   </xs:sequence>
   <xs:attribute name='name' type='xs:string' use='required'/>
  </xs:complexType>
</xs:element>

<xs:element name='addResponse'>
  <xs:complexType>
    <xs:attribute name='numBytes' type='xs:integer' use='required'/>
  </xs:complexType>
</xs:element>

<xs:element name='statusRequest'/>

<xs:element name='statusResponse'>
  <xs:complexType>
    <xs:sequence minOccurs="0" maxOccurs="1">
     <xs:element name='image'/>
    </xs:sequence>
    <xs:attribute name='key'   type='xs:string'  use='required'/>
    <xs:attribute name='index' type='xs:integer' use='required'/>
    <xs:attribute name='total' type='xs:integer' use='required'/>
  </xs:complexType>
</xs:element>

<xs:element name='loginRequest'>
```

```
  <xs:complexType>
   <xs:attribute name='user'     type='xs:string'  use='required'/>
   <xs:attribute name='password' type='xs:string'  use='required'/>
   <xs:attribute name='register' type='xs:boolean' use='optional'/>
  </xs:complexType>
</xs:element>

<xs:element name='loginResponse'>
  <xs:complexType>
    <xs:attribute name='user' type='xs:string' use='required'/>
  </xs:complexType>
</xs:element>

<xs:simpleType name='directionType'>
  <xs:restriction base='xs:string'>
    <xs:pattern value='next|previous|first|last'/>
  </xs:restriction>
</xs:simpleType>

<xs:element name='navigateRequest'>
  <xs:complexType>
    <xs:attribute name='direction' type='directionType' use='required'/>
  </xs:complexType>
</xs:element>

<xs:element name='deleteRequest'/>

<xs:element name='deleteResponse'>
  <xs:complexType>
    <xs:attribute name='name' type='xs:string' use='required'/>
  </xs:complexType>
</xs:element>
</xs:schema>
```

When you make changes to the XML, you need to make similar changes to **Parser** for the key constants. Add these values to **Parser**:

CODE TO TYPE: /src/xml/Parser.java

```
...
  public final static String name                = "name";
  public final static String value               = "value";
...
```

The client sends a **deleteRequest** to the server and receives (and processes) a **deleteResponse** in return.

In the **/src** folder **client.gui** package, create a **DeleteImageController** class as shown:

CODE TO TYPE: /src/client.gui/DeleteImageController.java

```
package client.gui;

import org.w3c.dom.*;
import xml.*;
import client.*;

public class DeleteImageController implements IController {

  ImageRepositoryClient client;

  public DeleteImageController (ImageRepositoryClient client) {
    this.client = client;
  }

  public boolean delete() {
    client.clearStatus();

    String xmlDeleteRequest = "<request><deleteRequest/></request>";
    return client.access.sendRequest(this, new Message(xmlDeleteRequest));
  }

  public void process(Message request, Message response) {
    if (Parser.success(response)) {
      NamedNodeMap map = response.contents.getFirstChild().getAttributes();
      String name = map.getNamedItem(Parser.name).getNodeValue();
      client.status("deleted image " + name + " from repository.");

      String xmlStatusRequest = "<request><statusRequest/></request>";
      client.access.sendRequest(new StatusController(client), new Message(xmlStatusRequ
est));
    } else {
      client.status("Unable to delete image:" + Parser.reason(response));
    }
  }
}
```

This code uses the same two-part structure as **AddImageController** did. The **delete** method sends the delete request to the server while the **process** message handles the response.

Because the server maintains the **ClientState** for the connected user, the logic of the client-side **DeleteImageController** is relatively straightforward.

The real functionality happens in the server. As you might expect, you'll change **ProtocolHandler** to respond to the **deleteRequest** that arrives. So, what if a user is viewing the first image in the repository and that user selects to delete the image just as a new client is logging into the system? As the current server is designed, each individual request coming to the server is handled by its own thread, which operates concurrently with all other threads in the system. It is entirely likely that the server will decide that it is safe to delete the first image in the repository when, in reality, a concurrently executing thread has just sent that image to be viewed by the newly connected user. You need some way to guarantee that an individual thread has the server's attention while processing. Using the **synchronized** key word, you can ensure that no two threads execute the **process** method at the same time. Because the **ProtocolHandler** class is the central handler in the server, it's the most convenient place to restrict concurrent access in a fine-grained way:

CODE TO TYPE: /src/server/ProtocolHandler.java

```
package server;

import server.ipc.*;
import server.model.*;
import xml.*;
import org.w3c.dom.*;

public class ProtocolHandler implements IProtocolHandler {
  final Repository repository;

  public ProtocolHandler (Repository r) {
    repository = r;
  }

  public synchronized Message process (String user, Message request) {
    Node child = request.contents.getFirstChild();
    if (child.getLocalName().equals ("addRequest")) {
      return new AddImageController(repository).process(user, request);
    } else if (child.getLocalName().equals("statusRequest")) {
      return new StatusController(repository).process(user, request);
    } else if (child.getLocalName().equals("navigateRequest")) {
      return new NavigateController(repository).process(user, request);
    } else if (child.getLocalName().equals("deleteRequest")) {
      return new ServerDeleteImageController(repository).process(user, request);
    }

    return null;  // unknown request? No idea what to do.
  }
}
```

The **ServerDeleteImageController** server-side controller must ensure that no other users are viewing the exact same image. If this is confirmed, the image is deleted; otherwise the request is denied and an appropriate response is sent back. The client display of the user requesting the deletion will depict the next image in the repository. To make this controller work, you will have to make a number of small changes to your existing code. Let's get started.

In the **/src** folder **server** package, create the **ServerDeleteImageController** class as shown (you will implement this class in stages):

CODE TO TYPE: /src/server/ServerDeleteImageController.java

```
package server;

import java.util.*;
import server.ipc.*;
import server.model.*;
import xml.*;

public class ServerDeleteImageController implements IProtocolHandler {

  Repository repository;

  public ServerDeleteImageController(Repository repository) {
    this.repository = repository;
  }

  public Message process(String user, Message request) {
    String xmlResp = "";
    ClientState cs = RepositoryServer.getState(user);
    String key = cs.getImageKey();
    if (key == null) {
      xmlResp = "<response success='false' reason='" + EmptyRepository + "'>" +
            "<deleteResponse name=''/></response>";
      return new Message(xmlResp);
    }

    // To Complete...
  }

  public static final String EmptyRepository = "Repository is empty.";
  public static final String AnotherViewer = "Another user is viewing the image.";
}
```

The **process()** method above responds to **deleteRequest** messages coming from the client. This code describes how to handle the unexpected case, where there are no images in the repository yet a **deleteRequest** message was received by the server. You call **RepositoryServer.getState()** to retrieve the **ClientState** object associated with the given user. Only when the repository is empty is there no **key** associated with the **ClientState** object. The only action to take is to return a failed response.

To complete the **process** method, you need to add logic that verifies that no other client is actively viewing the image that is about to be deleted. Instead of burying this logic deep within **RepositoryServer**, add a method to **RepositoryServer** to expose the full set of connected users. This flexible method allows you to perform an operation over all connected users of the system:

CODE TO TYPE: /src/server.ipc/RepositoryServer.java

```
...
  public static Collection<ClientState> users() {
    return users.values();
  }
...
```

You will use this method to determine whether any other connected user is viewing the exact same image as the one requested for deletion. Finally, you want **Repository** to allow you to set and get the metadata associated with an image in the repository by key value as shown:

CODE TO TYPE: /src/server.model/Repository.java

```
...
  public Properties getMetaData(String key) {
    return index.getMetaData(key);
  }

  public Properties setMetaData(String key, Properties md) {
    Properties old = index.setMetaData(key, md);
    storeIndex();
    return old;
  }
...
```

These methods expose the underlying Properties object associated with each image. They delegate the get/set requests to the **Index** of the repository, making sure to call **storeIndex()** to persist all changes. Now modify the **process** method of **ServerDeleteImageController** to validate that no other client is accessing the same image as the one requested for deletion:

CODE TO TYPE: /src/server/ServerDeleteImageController.java

```
...
 public Message process(String user, Message request) {
    String xmlResp = "";
    ClientState cs = RepositoryServer.getState(user);
    String key = cs.getImageKey();
    if (key == null) {
      xmlResp = "<response success='false' reason='" + EmptyRepository + "'>" +
            "<deleteResponse name=''/></response>";
      return new Message(xmlResp);
    }

    int num = repository.getOrder(key);
    Properties props = repository.getMetaData(key);

    // Verify that no other user is concurrently viewing the image
    for (ClientState other : RepositoryServer.users()) {
      if (other == cs) { continue; }

      if (key.equals (other.getImageKey())) {
        xmlResp = "<response success='false' reason='" + AnotherViewer + "'>" +
              "<deleteResponse name='" + props.getProperty(Parser.name) + "'/></respons
e>";
        return new Message(xmlResp);
      }
    }
    // To Complete...
  }
```

Let's take a closer look.

OBSERVE:

```
    // Verify that no other user is concurrently viewing the image
    for (ClientState other : RepositoryServer.users()) {
      if (other == cs) { continue; }

      if (key.equals (other.getImageKey())) {
        xmlResp = "<response success='false' reason='" + AnotherViewer + "'>" +
              "<deleteResponse name='" + props.getProperty(Parser.name) + "'/></respons
e>";
        return new Message(xmlResp);
      }
    }
```

The **RepositoryServer.users()** method returns the collection of users currently connected to the server. This method was provided to offer the greatest flexibility in dealing with other connected users. The enhanced for loop **for (ClientState other : RepositoryServer.users())** iterates over all **ClientState** objects associated with other connected users. **If any of these *other* ClientState objects has an image key equal to the image key being requested for deletion**, then you must respond with a failed **deleteResponse** message, this time declaring the reason to be **AnotherViewer**. The statement **if (other == cs) { continue; }** ensures that you skip over the **other ClientState** when you get to the **ClientState** associated with the current user requesting the deletion.

Once all checks have passed, complete the logic that actually deletes the object, as shown:

CODE TO TYPE: /src/server/ServerDeleteImageController.java

```
...
  public Message process(String user, Message request) {
    String xmlResp = "";
    ClientState cs = RepositoryServer.getState(user);
    String key = cs.getImageKey();
    if (key == null) {
      xmlResp = "<response success='false' reason='" + EmptyRepository + "'>" +
            "<deleteResponse name=''/></response>";
      return new Message(xmlResp);
    }

    int num = repository.getOrder(key);
    Properties props = repository.getMetaData(key);

    // Verify that no other user is concurrently viewing the image
    for (ClientState other : RepositoryServer.users()) {
      if (other == cs) { continue; }

      if (key.equals (other.getImageKey())) {
        xmlResp = "<response success='false' reason='" + AnotherViewer + "'>" +
              "<deleteResponse name='" + props.getProperty(Parser.name) + "'/></respons
e>";
        return new Message(xmlResp);
      }
    }

    ~~// To Complete...~~
    try {
      repository.delete(key);
      if (repository.size() == 0) {
        cs.setImageKey(null);
      } else {
        if (num > repository.size()) {
          num = repository.size();
        }
        cs.setImageKey(repository.getNthKey(num));
      }

      xmlResp = "<response success='true'>" +
              "<deleteResponse name='" + props.getProperty(Parser.name) + "'/></respons
e>";
    } catch (Exception e) {
      xmlResp = "<response success='false' reason='" + e.getMessage() + "'>" +
            "<deleteResponse name='" + props.getProperty(Parser.name) + "'/></response>
";
    }

    return new Message(xmlResp);
  }
```

This code won't compile until you write a **delete()** method in **Repository** that deletes an image by **key**. The **num** value is the ordinal position of the given image in the repository. Once this image is deleted (via **repository.delete(key)**), you can retrieve the key for the next image in the repository by using the same **num** value, then save this key as the image key for the **ClientState** requesting the deletion. In this final case, the returning **deleteResponse** is successful unless an exception was thrown when attempting to delete the image.

The above code also handles the situation where the user deletes the last image in the repository. The code updates

num when that happens.

Now modify **Repository**. Begin by writing a **delete()** method:

CODE TO TYPE: /src/server.model/Repository.java

```
...
  public void delete (String key) {
    File f = new File (storage, key);
    if (f.exists()) {
      if (!f.delete()) {
        throw new IllegalStateException ("Unable to delete image:" + key);
      }
    }

    index.delete(key);
    storeIndex();
 }
...
```

The **delete** method in **Repository** depends on adding a corresponding method to **Index**:

CODE TO TYPE: /src/server.model/Index.java

```
...
  public void delete(String fp) {
    if (!keys.contains(fp)) {
      return;
    }

    keys.remove(fp);
    meta.remove(fp);
  }
...
```

To see whether your changes were implemented properly, run **ServerLauncher** and **ClientLauncher** and make sure you are the only connected client. If the **Repository** folder is empty, add some images. Verify that you are looking at the first image in the repository. Now request to delete each image that you see, one at a time. You can review the progress of the server by refreshing the **Repository** folder after each delete request. You can add images and then delete them, until the repository is empty.

Now connect two clients to the same repository and have each of them view the first image in the repository. If you have deleted all images, then go ahead and add one! Now, try to have one of the clients attempt to delete the first image in the repository. As you will see, attempts to delete an image are denied as long as another client is viewing that image.

Upgrade Protocol to Display Metadata

To display metadata information on the right side of the client display, you need to upgrade the protocol. The metadata is stored by the server; you can "piggyback" the information with the **statusResponse** message by making a small change to **repository.xsd**. These changes add a sub-child to **image** that allows you to have a sequence of **metadata** tag information for each image. The metadata will come from the **Index**:

CODE TO TYPE: /repository.xsd

```
<xs:element name='metadata'>
  <xs:complexType>
    <xs:attribute name='name'  type='xs:string' use='required'/>
    <xs:attribute name='value' type='xs:string' use='required'/>
  </xs:complexType>
</xs:element>

<xs:element name='statusResponse'>
  <xs:complexType>
    <xs:sequence minOccurs="0" maxOccurs="1">
      <xs:element name='image'/>
      <xs:sequence minOccurs="0">
        <xs:element name='metadata'/>
      </xs:sequence>
    </xs:sequence>
    <xs:attribute name='key'   type='xs:string'  use='required'/>
    <xs:attribute name='index' type='xs:integer' use='required'/>
    <xs:attribute name='total' type='xs:integer' use='required'/>
  </xs:complexType>
</xs:element>
```

The fundamental structure of the **metadata** schema element is a **(name, value)** pair. The change to **statusResponse** allows for any number (possibly zero) of **(name, value)** pairs to be associated with the **image** being included in the **statusResponse**. To take advantage of this new capability, modify **ServerStatusController** as shown:

CODE TO TYPE: /src/server/ServerStatusController.java

```
package server;

import java.io.*;
import java.util.*;
import server.ipc.*;
import server.model.*;
import util.*;
import xml.*;

public class ServerStatusController implements IProtocolHandler {

  Repository repository;

  public ServerStatusController(Repository repository) {
    this.repository = repository;
  }

  public Message process(String user, Message request) {
    ClientState cs = RepositoryServer.getState(user);
    File f = repository.getImage(cs.getImageKey());
    ~~String imageData = "";~~

    // Default message in case the repository is empty
    String xmlResp = "<response success='true'><statusResponse key='" + cs.getImageKey(
) + "' " +
                     "index='" + repository.getOrder(cs.getImageKey()) + "' total='" +
repository.size() + "'>" +
                     "</statusResponse></response>";
    if (repository.size() == 0) {
      return new Message(xmlResp);
    }

    try {
      String encoding = ImageEncoding.encode(f);
      String metadata = "";
      Properties props = repository.getMetaData(cs.getImageKey());
      for (String name : props.stringPropertyNames()) {
        metadata += "<metadata name='" + name + "' value='" + props.getProperty(name) +
 "'/>";
      }
      String imageData = "<image>\n<![CDATA[" + encoding + "\n]]>" + metadata + "</imag
e>";
      xmlResp = "<response success='true'><statusResponse key='" + cs.getImageKey() + "
' " +
                "index='" + repository.getOrder(cs.getImageKey()) + "' total='" + repos
itory.size() + "'>" +
                imageData + "</statusResponse></response>";
    } catch (Exception e) {
      ~~System.err.println ("Unable to encode image file:" + ioe.getMessage());~~
      xmlResp = "<response success='false' reason='" + UnableToEncode + "'>" +
                "<statusResponse key='" + cs.getImageKey() + "' " +
                "index='" + repository.getOrder(cs.getImageKey()) + "' total='" + repos
itory.size() + "'>" +
                "</statusResponse></response>";
    }

    ~~String xmlResp = "<response success='true'><statusResponse key='" + cs.getImageKey(~~
~~) + "' " +~~
~~        "index='" + repository.getOrder(cs.getImageKey()) + "' total='" + repository.si~~
~~ze() + "'>" +~~
~~        imageData + "</statusResponse></response>";~~
    return new Message(xmlResp);
  }

  public static final String UnableToEncode = "Unable to encode image file";
}
```

The above code constructs a **metadata** XML fragment which is inserted as a sub-child to the **image** XML tag.

On the client side, you need to modify the **StatusController** to extract the metadata from the **statusResponse** it receives:

CODE TO TYPE: /src/client.gui/StatusController

```
package client.gui;

import java.io.*;
import java.awt.image.*;
import javax.imageio.*;
import org.w3c.dom.*;
import util.*;
import xml.*;
import client.*;

public class StatusController implements IController {

  ImageRepositoryClient client;

  public StatusController(ImageRepositoryClient client) {
    this.client = client;
  }

  public void process(Message request, Message response) {
    Node child = response.contents.getFirstChild();
    Node imageNode = child.getFirstChild();
    if (imageNode == null) {
      client.status("Repository has no images.");
      client.meta(null);
      client.display(null);
      return;
    }

    client.meta(null);
    NodeList metaNodes = imageNode.getChildNodes();
    for (int i = 0; i < metaNodes.getLength(); i++) {
      Node n = metaNodes.item(i);
      if (n.getNodeType() != Node.ELEMENT_NODE) { continue; }

      String name = n.getAttributes().getNamedItem("name").getNodeValue();
      String value = n.getAttributes().getNamedItem("value").getNodeValue();

      client.meta(name + " = " + value + "\n");
    }

    int idx = Integer.valueOf(child.getAttributes().getNamedItem("index").getNodeValue(
));
    int total = Integer.valueOf(child.getAttributes().getNamedItem("total").getNodeValu
e());
    try {
      byte[] bytes = ImageEncoding.decode(imageNode.getTextContent());

      InputStream in = new ByteArrayInputStream(bytes);
      BufferedImage image = ImageIO.read(in);
      client.display(image);
      client.status("Image " + idx + " of " + total);
    } catch (IOException ioe) {
      client.status("Unable to decode image from server:" + ioe.getMessage());
      client.meta(null);
      client.display(null);
    }
  }
}
```

Everything in an XML message is represented as a node, which means that you have to determine the type of each child node to make sure it is a true **ELEMENT_NODE**, and thus a **metadata** child of the **image** element. Once you do that, you extract the **(name, value)** value from the node and append that information to the client's metadata frame, on the right side of the client's GUI.

The above code also fixes a problem. You may have noticed that when you delete the last image in the repository, the screen still shows the image even though the status says, "Repository has no images." If there is no image to display, **StatusController** now clears the image and metadata.

To support this method, add a **meta()** method to **ImageRepositoryClient** that clears or appends text to the panel on the right side of the image:

CODE TO TYPE: /src/cilent.gui/ImageRepositoryClient.java

```
...
  public void meta(String string) {
    if (string == null) {
      imgMetaData.setText("");
    } else {
      imgMetaData.append(string);
    }
  }
...
```

Close down your client and server applications (if they are still running) and execute **ServerLauncher** and **ClientLauncher** to connect the client to the server. As you navigate through the images, you'll see metadata information in the right panel. For now, there are only two pieces of metadata; add some more, namely, the size of the image in bytes (both original and encoded) and the name of the user who uploaded the image. Modify the server-side **ServerAddImageController** as shown:

CODE TO TYPE: /src/server/ServerAddImageController.java

```
package server;

import java.io.*;
import java.util.*;
import server.ipc.*;
import server.model.*;
import util.*;
import org.w3c.dom.*;
import xml.*;

public class ServerAddImageController implements IProtocolHandler {
  Repository repository;

  public ServerAddImageController(Repository repository) {
    this.repository = repository;
  }

  public Message process (String user, Message request) {
    Node child = request.contents.getFirstChild();
    String name = child.getAttributes().getNamedItem("name").getNodeValue();
    Node imageNode = child.getFirstChild();

    String xmlResp;
    try {
      byte[] bytes = ImageEncoding.decode(imageNode.getTextContent());
      String key = repository.add (bytes, name);

      Properties props = repository.getMetaData(key);
      props.setProperty("user", user);
      props.setProperty("size", "" + bytes.length);
      props.setProperty("encoded-size", "" + imageNode.getTextContent().length());
      repository.setMetaData(key, props);

      ClientState cs = RepositoryServer.getState(user);
      cs.setImageKey(key);
      xmlResp = "<response success='true'><addResponse numBytes='" + bytes.length + "'/
></response>";
    } catch (IOException ioe) {
      xmlResp = "<response success='false' reason='" + CorruptedImageData + "'>" +
                "<addResponse numBytes='0'/></response>";
    } catch (Exception e) {
      xmlResp = "<response success='false' reason='" + e.getMessage() + "'>" +
                "<addResponse numBytes='0'/></response>";
    }

    return new Message(xmlResp);
  }

  public static final String CorruptedImageData = "Encoded image data appears to be cor
rupted.";
}
```

Now continue adding images; the user identifiers are now associated with each image. You're ready to make the final constraint check to make sure that a user deleting an image is actually the one who uploaded that image in the first place.

Preventing Multiple Login Requests

In this lesson you faced the challenge of concurrent access to the repository; there is a similar issue regarding the way a user logs into the system. You need to be sure that these two cases are prevented:

- Two clients simultaneously submit self-registration requests for the same user id.
- A user attempts to use the same login credentials as a user who is currently logged in.

Since each **RepositoryThread** operates independently and concurrently, you have to consider where these threads can synchronize their actions to prevent the two cases described above from happening. In **RepositoryThread**, you can see an opportunity in the **run** method where you can synchronize multiple threads according to the way they access **userManager**. Instead of defining a **synchronized** method, you introduce a **synchronized** block to ensures that only one concurrent thread operates within this block. If two threads are trying to self-register, then the first one in will succeed and the second will block and fail when it gets its chance. If two threads are trying to log in using the same user credentials, then the first one through will succeed and register itself with **RepositoryServer** (note how the **register** invocation is moved). With these changes, you can be sure that when the second thread is allowed to continue, it will detect a logged-in user with the same user id, and exit in failure (so sad):

CODE TO TYPE: /src/server.ipc/RepositoryThread.java

```
package server.ipc;

import java.io.*;
import java.net.*;
import org.w3c.dom.*;
import xml.*;
import server.*;

public class RepositoryThread extends Thread {
  Socket client;
  BufferedReader fromClient;
  PrintWriter toClient;
  IProtocolHandler handler;
  String user;
  UserManager manager;

  RepositoryThread (UserManager um, Socket s, IProtocolHandler h) throws IOException {
    fromClient = new BufferedReader(new InputStreamReader(s.getInputStream()));
    toClient = new PrintWriter (s.getOutputStream(), true);
    client = s;
    handler = h;
    manager = um;
  }

  public void run() {
    // authentication by first login message. Stop if not a loginRequest.
    Message m = Parser.extractRequest(fromClient);
    Node child = m.contents.getFirstChild();
    if (!child.getLocalName().equals (Parser.loginRequest)) {
      return;
    }

    // Get authentication information
    String user = child.getAttributes().getNamedItem(Parser.loginUser).getNodeValue();
    String pass = child.getAttributes().getNamedItem(Parser.loginPassword).getNodeValue
();

    // might be self-registration.
    Node registerNode = child.getAttributes().getNamedItem(Parser.loginRegister);
    boolean register = false;
    if (registerNode != null) {
      register = Boolean.valueOf(registerNode.getNodeValue());
    }

    // tell client decision and engage handler on successful login
    boolean validated;
    synchronized (manager) {
      if (register) {
        if (manager.registerUser(user, pass)) {
          m = new Message("<response success='true'><loginResponse user='" + user + "'/
></response>");
          validated = true;
          RepositoryServer.register(user, this);
        } else {
          m = new Message("<response success='false' reason='" + Parser.invalidCredenti
als + "'>" +
                "<loginResponse user='" + user + "'/></response>");
          validated = false;
        }
      } else {
        if (!manager.authenticate(user, pass)) {
          m = new Message("<response success='false' reason='" + Parser.invalidCredenti
als + "'>" +
                  "<loginResponse user='" + user + "'/></response>");
          validated = false;
        } else {
          if (RepositoryServer.getState(user) != null) {
            m = new Message("<response success='false' reason='" + DuplicateLogin + "'>
" +
                                "<loginResponse user='" + user + "'/></response>");
            validated = false;
          } else {
            m = new Message("<response success='true'><loginResponse user='" + user + "
'/></response>");
            validated = true;
            RepositoryServer.register(user, this);
          }
        }
      }
    }

    toClient.println(m.toString());
    if (toClient.checkError()) {
      ~~return;~~
      validated = false;
      RepositoryServer.unregister(user);
    }

    if (validated) {
      ~~RepositoryServer.register(user, this);~~

      // have handler manage the protocol until it decides it is done.
      while ((m = Parser.extractRequest(fromClient)) != null) {
        manager.updateAccessTime(user);
        Message response = handler.process(user, m);
        if (response == null) { break; }

        toClient.println(response.toString());
        if (toClient.checkError()) {
          break;
        }
      }
      RepositoryServer.unregister(user);
    }

    try {
      fromClient.close();
      toClient.close();
      client.close();
    } catch (IOException ioe) {
      System.err.println("Unable to close connection:" + ioe.getMessage());
    }
  }

  public static final String DuplicateLogin = "User is already connected";
}
```

Once this code is complete, run the **ServerLauncher** and execute two **ClientLauncher** applications. Have the first client self-register an account. Have the second client try to log in using the same credentials, and the server will detect the duplicate login.

This has been a long lesson, but you still need to write one final test case using the new functionality that you added in this lesson. The next test case validates that you can add two images, and delete them one at a time.

In the **/test** folder **server.ipc** package, create a **TestDeletion** class that extends **TestAddBehavior**. While you type the code, try to follow the narrative sequence of actions in the test case method:

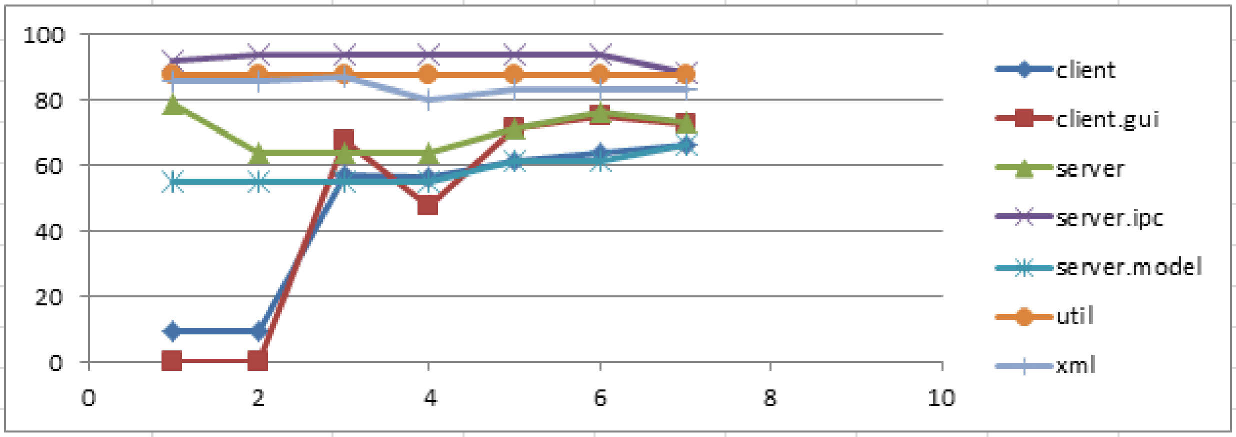
CODE TO TYPE: /test/server.ipc/TestDeletion.java

```
package server.ipc;

import java.io.File;

import server.ClientState;
import server.ServerDeleteImageController;
import server.ServerNavigateController;
import xml.*;

public class TestDeletion extends TestAddBehavior {

  static Message requestDELETE () {
    String xmlNavRequest = "<request><deleteRequest/></request>";
    return new Message(xmlNavRequest);
  }

  public void testDeletions() throws Exception {
    String splashFP = "c00bc1ed28fabdbcebc3e4735decc83e";

    String helpFP = "6e3a233232c4c8e0c8bb1c163aa48d9d";
    String user = "user00";

    toServer.println(requestLOGIN(user, "password", true));
    expectSuccess(fromServer);

    File f = new File("images", "repositorySplash.png");
    toServer.println(requestADD("sampleImage", f));
    expectSuccess(fromServer);

    ClientState state = RepositoryServer.getState(user);
    assertEquals (splashFP, state.getImageKey());

    f = new File("images", "help_32.png");
    toServer.println(requestADD("help", f));
    expectSuccess(fromServer);

    assertEquals (helpFP, state.getImageKey());

    // now go to first and delete
    ServerNavigateController nc = new ServerNavigateController(RepositoryServer.reposit
ory);
    nc.process(user, TestNavigationSequence.requestNAVIGATE(Parser.first));
    assertEquals (splashFP, state.getImageKey());

    ServerDeleteImageController dc = new ServerDeleteImageController(RepositoryServer.r
epository);
    dc.process(user, requestDELETE());

    assertEquals (helpFP, state.getImageKey());

    // Now delete last one
    dc.process(user, requestDELETE());
    assertTrue (state.getImageKey() == null);
  }
}
```

This test case takes advantage of many of the controllers and scaffolding test case methods you have developed in this course. See how all of your earlier work makes it possible to write effective test cases such as this one? Excellent!

Re-execute all JUnit test cases for your project to validate that they all pass; then, re-execute using EclEmma to determine coverage of the code.

	Authenticate Users	Server Sessions	Client Login	Server Menu	Browse Images	Navigate Image	Delete Image
src	72.6	72.3	73.3	71.0	74.1	75.2	74.7
client	9.6	9.6	57.0	56.5	61.2	63.6	66.5
client.gui	0	0	67.4	47.4	71.7	75	72.8
server	78.7	63.6	63.6	63.6	71.6	76.2	73.0
server.ipc	91.9	94	93.7	93.7	94.0	94	88.3
server.model	55.2	55.2	55.2	55.2	61.5	61.5	66.1
util	87.8	87.8	87.8	87.8	87.8	87.8	87.8
xml	85.9	85.9	87.0	80.3	83.5	83.5	83.5



The results above are truly outstanding. If you get a chance to review the code statistics (give or take a few lines) you will see that this project contains:

- More than 30 classes totaling 2000+ lines of Java code.
- More than 10 test cases totaling 800+ lines of Java code.
- A schema file totaling 100+ lines of XSD.

Your test cases written throughout the project assure nearly 75% of your code. The code that doesn't execute is mostly code that cannot be tested manually (for example, **ServerLauncher** and **ClientLauncher**) or cannot be executed in testing (**SplashScreenLogic**). Some controllers are entirely GUI-based (**QuitController**).

The code with coverage that's below 80% is due largely to extensive error handling (for example, **Repository**) to handle the interaction with the file system, none of which can be tested readily using JUnit. I hope you feel a real sense of satisfaction at this point, you've come a long way! I also hope you can see the limitless possibilities provided by the design of this client/server system. You can add new messages and controllers to handle them, based on the template shown repeatedly in these lessons. And so, we end where we began, with an emphasis on a solid design with implementation that leads to readily tested code—even GUIs—and through that process, you can now write distributed applications. Well done!