6.005 Elements of Software Construction
Fall 2008
Lab 1: Multipart Data Transfer

In this lab, you will learn about using Subversion for collaborative development and you will explore the Multipart File Transfer code we provide for Project 1. You will also learn about the possible failure conditions for retrieving URLs, and write unit tests that trigger or simulate those failures.

Before Lab

Before coming to lab, **read this lab handout, Project 1, and any linked references**. You won't have time in lab to read, so do it first.

Collaborative Development with Subversion

The following exercises were in the last lab, but very few people got to them. These exercises should familiarize you with several important aspects of collaborative, multi-person development using a version control system such as Subversion.

Who, What, Where, When, and Why

In addition to storing your source code files and the changes made to them, Subversion stores information about who changed what, when, in which files—and if the person doing the changing was playing nice and writing good commit log messages, it can even tell you why the changes were made.

In the Package Explorer of Eclipse, browse to one of the files you modified and committed during today's lab, or during the previous lab. Right-click the file, and select "**Team → Show History**." A new view will appear (at the bottom of the Eclipse window, if you don't drag it somewhere else). After a few moments of discussion with the Subversion repository, that view will list all the revisions committed to the repository that affected the selected file.

Clicking on a particular revision will show you both the files that were added ("A"), modified ("M"), or deleted ("D") by that commit, as well as the commit comment written by the committer. Whenever you use version control to collaborate, write commit comments!

Shared Repositories

Speaking of using version control to collaborate... let's check out a new Subversion repository that, unlike your personal repository, is accessible to the entire class. Refer to the previous lab and the SVN documentation and add the following repository:

```
svn+ssh://athena.dialup.mit.edu/mit/6.005/svn/groups/oop_java/everyone
```

In that repository is a project named `collab_svn`—**check it out**.

Once you've checked out the project, return to the Java perspective and **run the `Main` class** in package `collab` (for example, by right-clicking and selecting "Run As... → Java Application").

The `Main` class `main` method looks for all the Java classes in the `collab` package, creates a new instance of each one, and calls the method `toString()` on each instance.

Some of your classmates may already have added their own classes to the `collab` package, but there should be at least one other class from one of the teaching assistants. Using the teaching assistant class as a template, **create a new class** in the `collab` package that is named with your Athena username (but follows the Java convention of CamelCaseCapitals for class names). One way to do this is to right-click "collab" and choose "Add → Class." Have the `toString()` method of this class `return` a `String` of your choosing. You should be able to run the `Main` class again and see your message in the list.

Updating

Right-click on the "collab_svn" project and choose "Team → Update." Alternatively, select the "Team → Synchronize with Repository" option to use the synchronization screen seen in the last lab, where you can see updates before they happen. Right-click and choose "Update" to update files or directories.

Updating gives you the latest version of something (a file, or a directory or directory tree full of files) from the repository. Relevant reading from the SVN documentation includes the sections on How Working Copies Track the Repository & Mixed Revision Working Copies.

Committing

Before continuing, **make sure you can run the `Main` class successfully** and see everyone's messages, because you are about to check in. As mentioned in the previous lab, checking in broken code will surely earn you the ire of other students who, should they update and receive your nonfunctional changes, will be unable to continue working.

Check in your work. Since you have created a new file, you need to add it to version control. **Eclipse does not automatically put new files into Subversion**; you have to tell it that you want to commit each new file. (Once you do that, all future changes you make to that file will be committed.) One way to add a new file to subversion is to right-click on it and select "Team → Add to Version Control". You also have a chance to add files using the list of checkboxes in the commit dialog itself.

When you try to check in, your attempt may fail with an error from Subclipse because your working copy is out of date. That means somebody else in the class committed since the last time you updated, so Subversion doesn't know what to do with the differences. Update your working copy, and try the commit again.

All is Well

Find a partner, and make sure both you and your partner have received each other's code by committing and updating; you should both be able to run the `Main` class and see your partner's message in the console.

Merging

With the help of your partner, you will now simulate a situation where the use of Subversion becomes more complex. Make sure both you and your partner have updated versions of the shared project. **Pick one of the Java files added by either you or your parter**. You will both edit this same file, but you will do it on different machines. One of you (say, Alice) should add a comment line near the top of Alice's copy of the file, right after the line "`package collab;`" The other person (Bob) should add a comment line at the end of Bob's copy of the file file. Be sure each person makes no other changes.

Pick one of you to commit first (say, Alice), and **commit**. Once Alice commits, Bob will **update**. Even though Bob has modified the same file Alice committed, because the changes were in different parts of the file, Subversion will merge the changes automatically. Now Bob can **commit**, and Alice should **update**. Once this process is done, both you and your partner should have identical source files with both of the extra comments.

Conflicts

While SVN does a good job of merging unrelated changes to the same source file, it is not magic.

In the same file you've been editing with your partner, both you and your partner should **modify the `String` returned by `toString()` to have *different* values**.

Pick one person to commit first. Once the change is committed, the other person should update, and you should both take a look at the result, which should contain a **conflict**.

First, notice that a blue-ish square-ish icon has been added to the conflicted file in the Package Explorer. This indicates that the file contains a conflict that Subversion should not merge automatically. Until you inform Subversion that the conflict has been manually resolved, removing that conflict icon, it will refuse to allow you to commit the file.

Second, several extra files have appeared, with the different bits of code Subversion couldn't fit together. (The SVN documentation on Resolve Conflicts (Merging Others' Changes) explains what these files are.)

To merge the conflict manually, right-click on the file and pick "**Team → Edit Conflicts**," which brings up a two-paned editor. On the left hand is your (local) code, and on the right is your partner's (committed) code.

Edit the left side to your mutual, conflict-resolving satisfaction, and save. Then do "**Team → Mark Resolved**." The extra files will be deleted, and the conflict icon will disappear. Check that the resolved version runs fine—don't break the build!—and commit it.

Fall Back!

One final note: the Replace With... menu may come in handy if you realize you need to fall back to a previous version of your work. Just remember to commit **working code** with **good commit comments**, and it should always be easy to restore order to any mess you find yourself in.

✓**Checkpoint.** Find a TA or another member of the course staff and review your work on collaborative development.

Project Layout

Now to start working with Project 1.

Check out the `multipart` project from your group SVN repository. Your group SVN repository is located at

```
svn+ssh://athena.dialup.mit.edu/mit/6.005/svn/groups/multipart/groupK
```

where *K* is your assigned group number.

Take a moment to study the layout of the project's various resources. The project layout will be similar in later assignments.

- **src** is a source folder. It contains the Java source files for the application, a skeleton implementation of the multipart downloader that you will fill in, and utility classes.
- **`lib`** contains libraries required by the project. For this project it just contains the JUnit library, used for unit testing.

Run the application by right-clicking on `src/ui/Main.java` and choosing Run As → Java Application You should see the application's interface appear in a new window. It can't do anything right now, but it will be able to download complex multipart streams once you have finished the project.

Implement a Simple Downloader

Now, let's get back to doing something. First let's locate the entry point to the multi-part downloader code.

Locate the entry point to the downloader. Run the program again, and try to begin a download. In the Console in Eclipse, you should see a long stack trace, all caused by an exception thrown in `Multipart`. Clicking on the line number in the stack trace will bring you to the offending line in `src/multipart/Multipart.java`. As described in the project assignment, this `openStream()` method is the entry point to your code, called by the GUI when a download is initatiated.

Give some simple functionality to `openStream()`. To see how the GUI works, we can at least give openStream() the ability to open 'normal' single-part files.

Look through the javadoc for `java.net.URL` to find how to build a URL from the given String and how to get an InputStream from a URL.

Fill in `openStream()` such that it returns an InputStream reading directly from the target of the given url. There are a couple ways to do this, but it can be done in as little as one line.

Look at the spec for `openStream()`, found directly above the method in the code, or in the `Multipart` javadocs. When does your `openStream()` implementation behave as specified?

Try out your simple downloader. Now the GUI should be able to succesfully ask your `Multipart.openStream()` method for an InputStream from a given url, at least for normal files. For multi-part manifest files, the stream will of course just give the contents of the manifest, not the data it describes.

You can try the program on any url, but it can only preview `txt`, `png`, `gif`, and `jpg` files. As discussed in the project, the GUI can also view or animate sequences of such files, one at a time, as encoded in a file-sequence stream. Some urls you can try:

```
http://mit.edu/6.005/www/fa08/project1/cheerup.jpg
http://mit.edu/6.005/www/fa08/project1/here/bettyboop.png-seq
http://mit.edu/6.005/www/fa08/project1/empty.txt.parts
```

The last of these is a multipart manifest file, but in its current state the program will jsut display the contents of the manifest, as opposed to the data represented by the manifest.

Note that you will probably want to revert `Multipart` instead of committing it, to avoid a Subversion conflict with your project partners, and because you will be designing and implementing a different approach than the simple downloader you've just built. To revert a file to the version stored in the repository, right-click on it and choose Team > Revert.

Your job in Project 1 is to write functionality for the multipart downloader. You will hook your new functionality into the application through the `Multipart.openStream()` method which the UI calls. This does not mean you should implement just one method, or even one class, but this method is the sole *entry point* to your code.

Run the JUnit tests

The project contains unit tests that test the multipart downlaoder.

Open `src/multipart/MultipartTest.java`. This class contains the unit tests for `Multipart`. **Run the tests** by right-clicking on `MultipartTest.java` and selecting `Run as... → JUnit test`. The test should run and fail, because `Multipart` isn't actually implemented yet.

Now you have a better feel for how the application works, you know the entry points into which you need to connect your implementation, and you have a way of measuring success by running the JUnit tests.

Create the Project Javadocs

Another important tool to know about is javadoc, which generates documentation from comments written in Java source code. Javadoc comments start with two asterisks, like `/** ... */`, and typically appear before a class or a method to describe the specification of that class or method. The article How To Write Doc Comments for the Javadoc Tool is a useful resource for learning about what you can and should say in a documentation comment.

Run Project → Generate Javadoc... to generate the documentation. Select the `multipart` project, and make sure that the Destination field under Use Standard Doclet points to the `api` directory of the project. Click on Finish and updated documentation will be generated in the `api` directory.

Find the file `api/index.html` and open it to see what the resulting documentation looks like. (If you see the raw HTML instead of a rendered web page, you need to right-click on it and select Open With → Web Browser to view it. If you prefer to use your own web browser to view it, rather than Eclipse's built-in web browser, then Open With → System Editor, or else find your Eclipse workspace directory in your operating system file manager and drill down until you find `api/index.html`.)

Note that people often choose not to commit easily generated content such as javadoc to team repositories, to avoid conflicts, since it consists of many files, and because when someone wants it they can simply generate it. To have SVN permanently ignore the `api/` directory, right-click on it and choose *Team > Add to svn:ignore* (if the command is disabled, then the directory is already being ignored).

Find and read the documentation for the FileSequenceReader class. You will be needing it in the next exercise.

Write Unit Tests for FileSequenceReader

In this exercise, you will get familiar with our made-up file-sequence format and the `FileSequenceReader`, which is a class used by the GUI to break file-sequences into their constituent sub-files.

File-Sequence Streams

As mentioned, our GUI knows how to display either single files or sequences of files. Thus we've made up a simple stream format for encoding a series of finite files, allowing the GUI to display each file in succession just as it would display a single finite file.

The format for file-sequence streams is a size *n0*, followed by data for a file of size *n0*, followed by a size *n1*, followed by data for a file of size *n1*, and so on, either forever or until the stream ends. The sizes are represented by four bytes, starting with the most significant byte (the standard network byte order, known as big-endian).

As an example, consider a file consisting of 5 bytes (shown in hexadecimal):

```
36 2E 30 30 35
```

and a 6-byte file:

```
72 75 6C 65 73 21
```

We would encode the first file followed by the second file as the 19-byte (4-byte length + 5-byte file + 4-byte length + 6-byte file) file sequence:

```
00 00 00 05 36 2E 30 30 35 00 00 00 06 72 75 6C 65 73 21
```

Testing

There is already an example JUnit test for `FileSequenceReader`, in `FileSequenceReaderTest`. It tests the class on a trivial file-sequence consisting of one sub-file of length 0. (Note that this is not the same as a file-sequence consisting of zero sub-files!) It uses a ByteArrayInputStream to create an InputStream backed by a byte array, so that it can simply feed the desired bytes through the stream.

Make a new class in the same package as `FileSequenceReader` to hold your unit tests. Since you are sharing this repository with your partners, and all of you are creating a test class for this exercise, you should name your test class something unique, such as `FileSequenceReaderTest_`*`username`*.

Start by writing two tests for `FileSequenceReader.readOneFile()` that you expect to always succeed. They should provide valid file-sequence streams and then check that these streams are properly broken into sub-files, like in the provided `FileSequenceReaderTest`. Run your tests to make sure they work.

Note that some or all of your tests will either need to catch exceptions and then fail, or be declared as throwing the exceptions, in which case they will fail automatically if the exception is thrown.

Now think about ways that `readOneFile()` might fail, and devise tests for a failure condition to make sure that it fails as desired. You may find the Javadoc for `FileSequenceReader` helpful, and you can see the unit tests in FibonacciTest from the last lab for useful examples of how to test that a particular exception is thrown. Run your test to make sure it works.

✓**Checkpoint.** Find a TA or another member of the course staff and review your work on the downloader code and the unit tests you wrote.

Commit Your Solutions

This is the end of the lab. Be sure to commit your `FileSequenceReader` JUnit tests to your group Subversion repository. **Since both you and your partners are sharing the same repository, be careful to check for conflicts when you commit, and don't break the build for your partners.**