6.005 Elements of Software Construction
Fall 2008

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

Problem

One way to handle the problem of reliably storing large files is to break them into pieces that are placed on multiple disks or machines. At the same time, a way to download a data stream more quickly is to break it into parts and download the different parts from different peers. You're probably familiar with this idea from BitTorrent, the peer-to-peer file sharing protocol, and it's used in a variety of other contexts. By downloading streams in multiple small parts,

the chance of the entire download failing is reduced (since each part can be restarted individually) and load is spread more evenly across the network.

In this project, you will build a *multipart* downloader that assembles a data stream from multiple, potentially endless, parts streaming individually from multiple machines. It allows the same part to be stored redundantly in multiple locations so it can be resilient to failures. Since the multipart streams you will be downloading may be unbounded and never end, your program will assemble the stream incrementally from its parts as they are downloaded, displaying the file or streamed sequence of files (an animated sequence of images, for example) as the download progresses.

In later projects, you'll be asked to refine the problem itself. In this case, however, the problem is largely fixed. You are given a particular protocol format (described below) and required to implement a single method, `Multipart.openStream()` (although you will likely wish to design and use other classes to help with this task). Thus your task is essentially the implementation of a simple API for multipart data transfer. Being an API ('application programmer interface') simply means your solution can be accessed 'programmatically', that is by method calls, rather than through a GUI, for example. Separating out an API is generally good practice because it allows the code to be used in a variety of contexts, separates the user interface from the core functionality, and makes testing easier. We are providing you with a simple GUI that can be easily combined with the API. The `openStream()` method of the API returns an InputStream for reading from the multipart stream; the GUI then simply reads serially from this stream, but it could be used more ambitiously, for interleaving the downloading of multiple streams, for example.

The breakdown of the stream into parts is specified in a special manifest stream. Your code will have to parse this stream, using it to determine which parts to download. Furthermore, these parts can themselves be manifest streams, in which case your code should recursively stream the sub-parts. Both this parsing and the downloading process itself are naturally expressed as state machines.

A variety of failures can occur, and it will be up to you to figure out what they are and decide how they should be handled.

Purpose

The purpose of this project is to help you (1) become familiar with the basic tools of software development in Java -- the language, the Eclipse IDE, and the JUnit testing framework; (2) get you started programming in Java in a simple imperative style; (3) learn how to conceive

of software systems and the environments in which they operate as state machines, and to record designs and environmental assumptions with state machine diagrams; (4) begin to appreciate the importance of testing in building high quality software; (5) acquire some familiarity with some important computer system concepts and technologies, such as URLs and HTTP, redundancy and fault tolerance.

Specification

Manifest Streams

A *manifest stream* specifies how a given stream is broken into parts.

Your program can assume that a URL points to a manifest stream if and only if the stream has the content type `text/parts-manifest` or its URL ends with the suffix `.parts`.

Each part in a manifest stream is separated by a line containing two dashes. For example, the file `picture.jpg` may be broken into three parts all stored on the same machine:

```
http://mymachine.mit.edu/picture.jpg-part1
--
http://mymachine.mit.edu/picture.jpg-part2
--
http://mymachine.mit.edu/picture.jpg-part3
```

To download this stream, your program would download each part in succession, thus recreating the original file.

The manifest stream can give alternatives, so that a part, or several parts, can be stored redundantly in different locations:

```
http://mymachine.mit.edu/picture.jpg-part1
http://yourmachine.mit.edu/picture.jpg-part1
--
http://mymachine.mit.edu/picture.jpg-part2
http://yourmachine.mit.edu/picture.jpg-part2
--
http://mymachine.mit.edu/picture.jpg-part3
http://yourmachine.mit.edu/picture.jpg-part3
```

In this case, the three parts are all stored on the machine `mymachine.mit.edu` and also on the machine `yourmachine.mit.edu`; for each part, if the first machine is not accessible, your

downloader should try the second.

Manifest streams can also be recursive. For example, the manifest stream `http://mymachine.mit.edu/endless.txt.parts` might look like:

```
http://mymachine.mit.edu/verse.txt
http://yourmachine.mit.edu/verse.txt
http://hermachine.mit.edu/verse.txt
--
http://yourmachine.mit.edu/chorus.txt
--
http://mymachine.mit.edu/endless.txt.parts
```

Thus the last part of this manifest stream refers back to itself, so a client reading from your `Multipart` implementation would receive an endless stream alternating between the contents of `verse.txt` and the contents of `chorus.txt`.

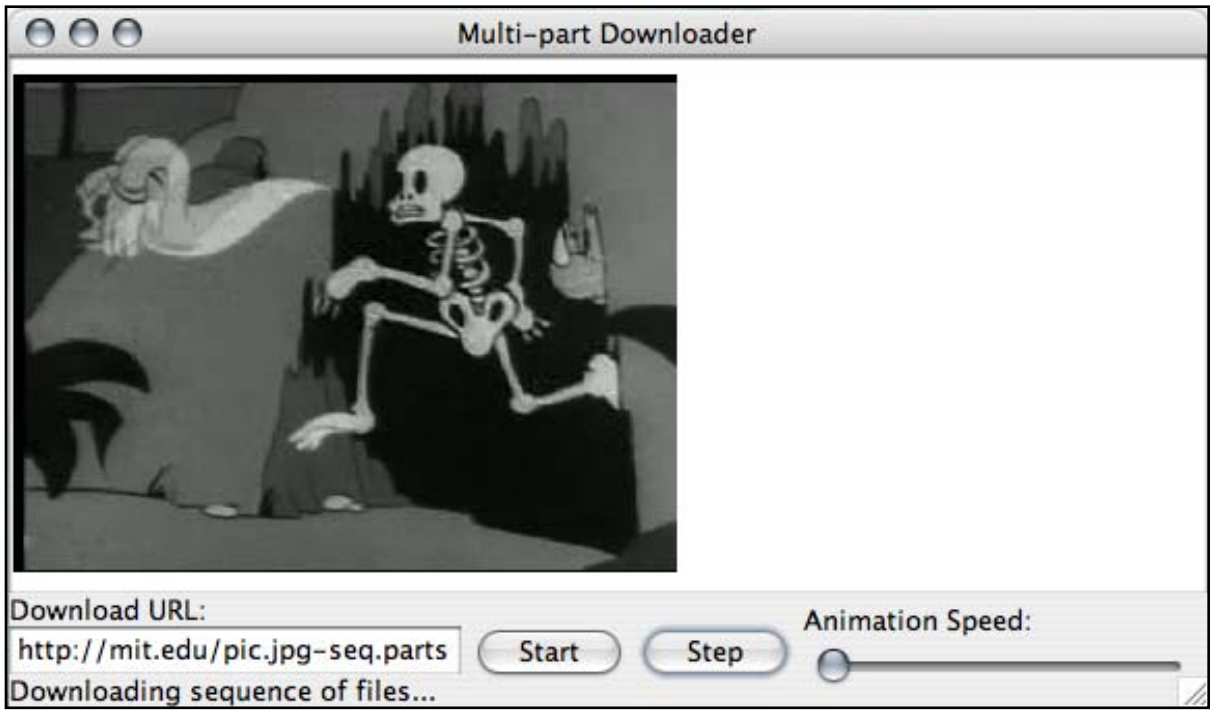
Note also that in the preceding manifest stream there are three alternatives for `verse.txt` but only one for `chorus.txt` and `endless.txt.parts`. Like in BitTorrent, certain parts may only be hosted by certain servers. Then again, it should never hurt if a nonexistent or perhaps even malformed URL is listed as an additional alternative source, since your code should be able to handle this and try a different alternative.

Finally, keep in mind that we are assuming all URLs point to potentially unbounded streams, not necessarily finite-length files. Thus another way to produce the endless series of verses and choruses would be an endless manifest stream (generated by a program running at the URL, for example):

```
http://mymachine.mit.edu/verse.txt
--
http://yourmachine.mit.edu/chorus.txt
--
http://mymachine.mit.edu/verse.txt
--
http://yourmachine.mit.edu/chorus.txt
--
[...]
```

and so on...

Application Behavior



The GUI offers a text field and an adjacent button marked *Start*. The user enters a URL in the text field pointing to a manifest stream (which may be local or on another machine) and clicks on the button. The application then downloads the stream part-by-part. If the stream results in a single finite file, the file is downloaded in its entirety and then displayed in the window. The GUI also supports (potentially endless) file-sequence streams, described below. Each file in these sequences is downloaded and displayed in succession, creating, for example, an image animation or textual progression. The *Step* button downloads and displays the next file in a sequence stream, and the *Animate* slider controls automatic stepping, for example to watch an animated sequence of image files.

API

The GUI component of the application is provided for you. Your task is to implement the method `Multipart.openStream()`, satisfying the specification in the javadocs for the `Multipart` class. You may want to create other classes as well to help with this task, but the sole *entry point* where the GUI will call your code is `Multipart.openStream()`.

Handling faults

A variety of failures may occur during execution, which your code should deal with gracefully. These include network failures (e.g. the program cannot download a file), manifest file syntax errors, and other failure scenarios.

Tasks

You should perform the following tasks:

- **Laboratory**. Complete the first lab of this project. A separate handout describes the lab activities.
- **Problem Refinement**. Analyze the problem statement carefully, and resolve any ambiguities or omissions. In particular, you should specify the structure of the manifest file using a grammar, clarify any assumptions that are necessary, and should specify what failures will be handled and how. In most cases, there will be no obvious best way to refine the problem, so you should use your own judgment and justify your decisions carefully (but briefly!). Your work should include a refined specification for the `Multipart` class itself.
- **Abstract Design**. Design a multipart downloader by describing a state machine (with a grammar or a state machine diagram) that describes the behavior of the downloader itself. You will probably find it easiest to start with the assumption that there are no failures, so that you can identify the major modes and the transitions between them, and then to consider the various failures that can occur. You may end up designing two state machines, such as a parser and a downloader.
- **Code Design**. Sketch the code structure using dependency diagrams for **two** possible implementations of your abstract design that differ substantively. List their relative merits, select one of them, and explain why you chose it. You should consider, in particular, whether your design is *testable*; it should be easy, for example, to simulate failures so that you can test your code against them.
- **Design Meeting**. Your TA will arrange to meet with your group during lab time (11am-2pm) on Friday, October 3, the day after your first deliverables are due.
- **Implementation**. Implement your chosen code design in Java. You might find that you want to make changes to your design. You are free to do this, but should record the changes so it is clear what the original design was and what changes you made to it.
- As you write the code, you should include a succinct specification of each method as a javadoc comment, and additionally a comment at the top of each class explaining its role.
- **Test**. Run the staff test suite using the test framework provided. If your program fails any of the tests, correct the problem, and rerun the test suite. Create at least **three** additional test cases of your own, that demonstrate the robustness of your `Multipart` implementation.
- **Reflection**. Write a brief commentary saying what you learned from this experience. What was easy? What was hard? What was unexpected? Briefly evaluate your solution, pointing out its key merits and deficiencies.

Infrastructure

We provide some example manifest streams. In approximate order of complexity, these are:

- http://mit.edu/6.005/www/fa08/project1/cheerup.jpg.parts
- http://mit.edu/6.005/www/fa08/project1/yellowsub.txt-seq.parts
- http://mit.edu/6.005/www/fa08/project1/icarus.jpg.parts
- http://mit.edu/6.005/www/fa08/project1/empty.txt.parts
- http://mit.edu/6.005/www/fa08/project1/bettyboop.png-seq.parts
- http://mit.edu/6.005/www/fa08/project1/hair.jpg.parts
- http://scripts.mit.edu/~6.005/project1/fly.txt-seq.parts.cgi

(**Note** that if you want to look at the contents of these manifests in a browser, you may have to save the files and then open them in a text editor. In particular, Firefox will sometimes just display the URL of the file, as though it were the contents of the file.)

For comparison, some of the original single-part files are in the same directory.

Deliverables and Grading

For the first deadline, your deliverables are:

- the problem refinement;
- the abstract design;
- and the code design;

and for the second deadline:

- the implementation;
- the tests;
- and your reflections.

Your code should be committed in the repository you share with your teammates by the deadline. All other parts of the project should be stored in your repository as two separate PDF documents, one for each deadline.

Grades will be allotted according to the following breakdown: problem refinement -- 20%; abstract design -- 20%; code design -- 20%; implementation -- 20%; testing -- 10%; reflection -- 10%.

Hints

Understanding the protocol. Look at the contents of the manifest files we provide, using a web browser. (You may have to save the files and then open them in a text editor. In particular, Firefox will sometimes just display the URL of the file, as though it were the contents of the file.) The structure is quite simple, and examining the manifest files and the parts of text files (which, unlike image files, have well-formed subparts) will give you an intuition for what your program should be doing.

Extending `InputStream`. Your code design may call for creating a subclass of `java.io.InputStream`. You should understand, by reading the `InputStream` javadoc, why you only need to override `read()` to have a legitimate subclass of `InputStream`. Consider what you would need to do differently if the *implementation* details of `InputStream` weren't so clearly documented. Also consider whether you want to override the `close()` method.

Using local `file` URLs. You can use URLs starting with `file://` to access local files. This could be useful for both debugging and testing.

Manifests (.parts files) are different from sequences (-seq files). Your job in this project is to download, parse, and interpret manifests. The sequence file format is implemented by FileSequenceReader, which you will unit-test in the lab. You can think of the sequence format as a primitive movie format that we've invented (analogous to .mov or .avi). Manifests may include -seq files, but your code shouldn't treat them any differently from other file formats, like .txt or .jpg. In particular, Multipart.openStream() should *not* convert a manifest into a sequence stream (i.e., it should *not* precede each part with its length as a 4-byte integer). Multipart.openStream() should return a stream that simply concatenates all the parts together.