Construction

Objectives

The objectives of this assignment are:

(1) Implement your FCS design with high-quality code and thorough unit tests
(2) Gain experience doing a task breakdown
(3) Gain experience using a revision control system (CVS)
(4) Gain experience using a unit testing framework (JUnit)
(5) Gain experience using a code coverage tool (Clover)

Assignment

This assignment requires that you complete the following tasks (in no particular order).

Update your Design Document based on feedback from the design review

Fix any problems that were found in your design during the design review and update your Design Document accordingly.

Implement your design

Write code that implements your updated design. The code should be of high quality. It will be graded based on the criteria listed in the Grading section.

Write automated unit tests

Write unit tests for your code using the JUnit framework. Use Clover to measure the level of code coverage achieved by your unit tests. Your grade will be partially based on how complete your unit tests are according to Clover's coverage metrics.

Automate your build and test process

You must automate all steps required to compile, package, and unit test your software. Specifically, you must automate the following tasks:

(1) compiling your source code
(2) packaging your software (e.g., packaging .class files in a JAR file)
(3) compiling and running your unit tests
(4) generating a Clover code coverage report
(5) deleting generated files (e.g., clean)

You may use shell scripts, make, Ant, Perl, or whatever you want to automate your build process, as long as it can be invoked from the command-line (no IDEs, please). Whatever technology you use to automate your builds should be installed by default in the Linux open labs so the TA can easily build and run your software.

For example, you might write a script, make file, or Ant build file that supports the following targets:

`compile`: compile the code

`package`: package .class files in a JAR file
`test`: compile and run unit tests, and generate a Clover code coverage report
`clean`: delete all generated files

Update your User Manual

Update your User Manual to reflect any changes you have made in the user-visible behavior of your system.

Use CVS for file sharing and revision control

All files should be checked into CVS. Using CVS will make it easy to share files with your teammates and to track multiple versions of your files. When your project is passed off, we will check your files out from CVS in order to build and run your system.

Do a detailed task breakdown

Before starting work on the tasks above, it will help to spend some time planning exactly how you will get everything done. Without this initial planning, you will have a vague idea of what needs to be done, but you won't have a clear idea of what order to do things in, and you'll run the risk of overlooking important items. Creating a plan beforehand will give you a complete understanding of what tasks need to be done and the order in which they need to be done.

Schedule a meeting with your team at which you will plan out the construction phase. Create a table containing all of the tasks that must be accomplished in order to complete the construction phase. Tasks are relatively small units of work that need to be done, usually taking between one and a few days each. Example tasks might be:

- Write build script
- Implement Check-In File use case
- Write unit tests for the DiffEngine class
- Update user manual
- etc.

In your task table you should list the following pieces of information for each task:

- Task name
- Task description
- List of pre-requisite tasks (other tasks that must be done before this one can begin)
- Team member who is assigned to complete the task

For example,

Task	Description	Pre-Requisites	Engineer
T1	Create source tree directory structure in CVS	None	Bill
T2	Write make file	T1	Bill
T3	Learn how to use JUnit	None	Jane
T4	Learn how to use Clover	None	Jane

T5	Create unit test driver program	T2, T3, T4	Jane
Etc.			

Deliverables

Submit one hard copy of your Detailed Task Breakdown. This is due early in the construction phase, and will be graded as soon as you turn it in. Everything else will be passed off and graded at the end of the construction phase.

At the end of the construction phase, submit one hard copy of your updated Design Document and four hard copies of your updated User Manual (one for me, and two for your quality assurance team). All of your files should also be checked into CVS. The TA will retrieve your files from CVS when passing you off. Your CVS repository should contain at least the following files:

(1) Source code
(2) Unit test code and data
(3) Libraries that your code depends on (JUnit, Clover, etc.)
(4) Automated build files (scripts, make files, ant files, etc.)

You might also want to put your Design Document and User Manual files into CVS.

Grading

Your task breakdown will be graded based on Completeness (did you leave anything out?), Feasibility (does your plan make sense?), and Readability (is your task breakdown easy to read?).

At the end of the construction phase, you will pass off your FCS implementation with the TA. During pass off the TA will:

(1) Retrieve the contents of your CVS repository
(2) Compile your system
(3) Compile and run your unit tests
(4) Record the level of code coverage achieved as reported by Clover (the higher the coverage, the better your grade will be)
(5) Exercise each of the main FCS use cases to verify that your system works. The purpose of this is to estimate how complete your implementation is, not to exhaustively test it (that will come in the next assignment).
(6) Verify that the User Manual has been updated to match the final implementation

At a later time the TA will:

(1) Evaluate the quality of the source code
(2) Verify that JUnit and Clover were used correctly
(3) Verify that the Design Document has been updated to match the final implementation

Code Evaluation Criteria

Your source code will be evaluated based on the following criteria:

1. Header comment on each class and method

Every class and method in your program should have a header comment. For a class the header comment should describe the purpose of the class. For a method the header comment should describe the purpose of the method and document each of its parameters and return value.

2. Explanatory comment on each class member variable

The declaration of each class member variable should include a comment explaining the meaning and purpose of the variable. This only needs to be done for class member variables, not for local variables declared within methods.

Here is an example of requirements 1 and 2:

```
/**
 * The URL class represents a web address such as
 *
 *     http://www.byu.edu:80/index.html.
 *
 * In addition to being able to store a URL, this class knows how to parse
 * out the pieces of a URL and how to resolve relative URLs.
 *
 */
public class URL {

      private String scheme;        // the URL's scheme (e.g., "http")
      private String host;          // the URL's host (e.g., "www.byu.edu")
      private int port;             // the URL's port number (e.g., 80)
      private String path;          // the URL's path (e.g., "/index.html")

      .
      .
      .

      /**
       * Resolve a relative URL with respect to this URL
       *
       * @param relativeURL the relative URL that is to be resolved
       * @return the absolute URL representing the resolved URL
       * @throws MalformedURLException
       *
       */
      public URL ResolveRelative(String relativeURL);

      .
      .
      .
}
```

3. Effective class, method, and variable names

Names chosen for classes, methods, and variables should effectively convey the purpose and meaning of the named entity.

4. Effective top-down decomposition of algorithms

Code duplication should be avoided by factoring out common code into separate routines.

Routines should be highly cohesive. Each routine should perform a single task or a small number of highly related tasks. Routines that perform multiple tasks should call different subroutines to perform each subtask. Routines should be relatively short in most cases. [Rule of Thumb: Many routines will be less than 20 lines. Almost all routines will be less than 50 lines. Routines longer than 100 lines should be rare.]

5. Code layout should be readable and consistent

The layout of your code should be readable and consistent. This means things like placement of curly braces, code indentation, wrapping of long lines, layout of parameter lists, etc.

6. Effective source tree directory structure

The files for your project should be effectively organized into subdirectories. For example, you might have subdirectories with names like `src`, `lib`, `test`, `docs`, etc. for organizing the different kinds of files in your project.

7. Effective file organization

Your source code should be effectively organized into multiple files. By default, each class should be placed in a separate file. Lumping all of your code in one or two files is not acceptable.