Checksums

Design Patterns: Composite, Observer, Strategy, Template method, Builder, Visitor, Memento

Introduction

Create a solution which recursively scans a target directory and calculates the hashes of all of its files.

The results of each scan must be written to a text file. This text file consists of lines in the following format:

<hash> <relative-path>

Here `<hash>` is the calculated hash and `<relative-path>` is the path of the file relative to that of the target directory. For example, an imaginary output file may look like this:

836289C997463D0487C0CAA2A609991F file1.dat
189089AFB437AD7F6C4F38BBF51E5953 somedir/file2.txt
...

Your solution should support different checksum algorithms. The user should be able to specify an algorithm when running a scan. It should be easy to add or remove such algorithms to/from the solution at a later stage.

The solution should have two modes of operation, depending on how the user selects to treat symbolic links (in Unix-based systems) or shortcuts (.lnk files in Windows systems):

- calculating the checksum of the actual symbolic link or shortcut (i.e., the contents of the file), or
- traversing the target of the symbolic link or shortcut.

The solution should display a progress indicator when it is performing a scan.

Step 1

The progress indicator displays only the name of the file that is being processed at the moment.

Step 2

The progress indicator displays the name of the current file and the number of bytes processed so far.

Step 3

The progress indicator displays progress percentage and estimated time remaining.

There should be an option to pause the scan at any point in time and restore it at a later stage. The program should provide an option to display a report of all files and their sizes to be scanned before the actual scan is started.

Hints

Individual file checksums

Create an interface for the checksum calculating algorithms. For example:

```
interface ChecksumCalculator {
   public String calculate(InputStream is);
}
```

Implement a subclass (or subclasses, if necessary) for each supported algorithm.

Do not implement the logic of the algorithms on your own. Instead, use a well-established library for your chosen language. For example:

- Java: `java.security.DigestInputStream` or Apache Commons `DigestUtils`.

Note that the checksum calculator operates on a stream. In a real application this stream will be associated with a file. In your unit tests use a stream derived from a string:

- Java: https://www.baeldung.com/convert-string-to-input-stream

Use an existing hash calculator to find out the hashes for a set of strings and use those to test your own calculator objects. For example, here is a list of possible tests for MD5:

MD5 checksum	String
`900150983cd24fb0d6963f7d28e17f72`	`"abc"`
`86fb269d190d2c85f6e0468ceca42a20`	`"Hello world!"`

Directory iteration

Since one of the requests is to have a progress percentage, we would like to build an in-memory representation of the directory structure that the algorithm will traverse so that we can better estimate the amount of bytes remaining to be processed. Use the Composite pattern to represent files and directories with a common abstract ancestor representing abstract files.

Store the file path and the size of the file in bytes. For directories, store the sum of all sizes of the files it contains.

Use the Builder pattern to build the in-memory directory structure. Create an abstract builder class and two concrete builder instances that treat symbolic links (symlinks in Unix-based systems) or shortcuts (.lnk files in Windows) differently:

- create a regular file node, if the user has selected NOT to follow symbolic links or shortcuts
- build a subtree for the target, if the user has selected to follow symbolic links or shortcuts

Please note that in the latter case the algorithm will need to implement cycle detection to avoid endless recursion.

Use the Visitor design pattern to implement directory iteration of the created structure. Create an abstract visitor class, which provides two different `visit` methods: for visiting files and for visiting directories.

Use the Template method pattern to implement a method for processing a regular file node and applying an abstract algorithm to it. Implement a concrete visitor instance `HashStreamWriter`, which uses the Strategy design pattern to calculate a checksum by invoking a corresponding Checksum calculator. Implement a second visitor instance `ReportWriter`, which outputs a report of all files that will be traversed and their size.

To enumerate files and directories on the file system use the functionality provided by your selected programming language.

For testing purposes, design `HashStreamWriter` to write its results to a stream. For example:

- Java: you can use `Writer` or `OutputStreamWriter`.

This will allow you to check the output of a scan. In your tests, use a stream redirected to a local buffer. You can check that buffer, once the scan finishes, to see if the results are correct.
To test the class, one option is to mock the file system. Another is to create several sample directories. For example, one empty directory, one which contains files and subdirectories, etc. Make the sample directories a part of your project, i.e. place them inside the directory structure, as a test resource for your project (e.g. in `src/test/resource` if using Maven). Also, the sample directories should be pushed to whatever source control system you are using.
When writing tests that use the sample directories, make sure that you are NOT hardcoding absolute paths. Instead, write the necessary code to detect the location of the test resources on the particular computer on which the build is taking place. For example, check this article on Baeldung.

Use an existing tool, such as `md5sum` in Linux, or `Get-FileHash` in Windows PowerShell to manually calculate the expected results for each sample directory. After that write the tests in

such a way, as to compare the results computed by your code and the one obtained with the existing tool.

Progress indicator

Step 1

To solve Step 1, implement the Observer pattern for `HashStreamWriter`. It should report each new file that it finds and processes.

Implement another class, say, `ProgressReporter`, which will observe a `HashStreamWriter`. It will receive notifications for all files discovered by `visitDiretory` method of the `HashStreamWriter` and simply print their names to STDOUT, like this:

```
Processing file1.dat...
Processing somedir/file2.txt...
...
```

Make sure you implement the pattern by providing a base class `Observable` and an interface `Observer`, instead of implementing the code directly into the other classes.

> **NOTE** Java has an implementation of an `Observer` interface and an `Observable` class. However, as this project is intended as an exercise on the Observer design pattern, please, do not use them, but rather implement your own versions. Also, keep in mind that they have both been deprecated as of Java 9.

Step 2

For Step 2 things will get a bit more complicated, because right now the solution consists of three distinct layers:
`ProgressReporter` → `HashStreamWriter` → `ChecksumCalculator`

First, note that it is the checksum calculator, which processes the individual files and thus it is the only one that "knows" how many bytes of the file have been processed so far.
So, first implement Observer for the checksum calculator(s). A calculator should report the number of bytes processed so far. Probably it will be an overkill to report each individual byte. This will be very inefficient. Instead, do so for entire chunks of information. For example. you can report once for every 1KB processed (or several KBs, or for each MB, etc.). Visually it may look like this:

Processing file1.txt… 56 byte(s) read Processing somedir/file2.txt… 1000 byte(s) read

TIP

In order to display dynamic progress on the same line use the `\r` character at the beginning of the output and use a printing function which does NOT automatically append a new line to the output. Check the excerpt at the end of the document for an example.

Now, note that there is no direct link between the progress reporter and the checksum calculator. Here are some possible ways to resolve this:

- Make it so that `ProgressReporter` observes `HashStreamWriter`, which in turn observes the checksum calculator. `HashStreamWriter` forwards all progress reported by the checksum calculator to `ProgressReporter`.
  In this case `HashStreamWriter` is itself an observer.
  The code may look like this:

```
class HashStreamWriter extends Observable implements Observer {

    private ChecksumCalculator calc;

    public HashStreamWriter(ChecksumCalculator calc) {
        this.calc = calc;

        // Attach the current class as an observer
        calc.attachObserver(this);
    }

    public void onNewFile(String path) {
        notify(this, new NewFileMessage(path));
    }

    @Override
    public void update(Observable source, Message m) {
        // Forward the message to the observers of
        // DirectoryHashStreamWriter.
        notify(source, m);
    }
}
```

- Make it so that when `ProgressReporter` subscribes to `HashStreamWriter`, `HashStreamWriter` also subscribes it to the checksum calculator. In this way `ProgressReporter` will directly receive all status updates from the calculator directly. It will still receive all updates from `HashStreamWriter` too.
  In this case there is no need for `HashStreamWriter` to observe the checksum calculator.
  The code may look like this:

```
class HashStreamWriter extends Observable {
```

```
    private ChecksumCalculator calc;

    public HashStreamWriter(ChecksumCalculator calc) {
        this.calc = calc;
        // No need to attach the current class as observer
    }

    public void onNewFile(String path) {
        notify(this, new NewFileMessage(path));
    }

    @Override
    public void registerObserver(Observer observer) {
        // Call the parent function to register
        // observer for our own object.
        super.registerObserver(observer);

        // Attach observer to the calculator too.
        calc.registerObserver(observer);
    }
}
```

Another thing to consider is how to structure the messages exchanged between observables and observers. You will probably want to use different types of messages for each case. Then the observer can decide what to do based on the actual types of the sender and/or the message. For example:

```
class ProgressReporter implements Observer {

    String currentPath = "(nothing)";
    Integer bytesRead = new Integer(0);

    public void update(Observable sender, Object message) {

        if(sender instanceof ChecksumCalculator) {
            // There is additional progress on the current file.
            // Assuming an Integer was been passed as a message.
            bytesRead = (Integer)message;
        }
        else if(sender instanceof HashStreamWriter) {
            // A new file was found. Output a LF character to start a new line.
            System.out.print('\n');

            // Assuming a String was been passed
            currentPath = (String)message;
            bytesRead = new Integer(0);
        }
        else {
            throw new IllegalArgumentException("Unexpected message");
        }

        refreshDisplay();
    }

    private void refreshDisplay() {
        System.out.print(
```

```
            "\rProcessing " +
            currentPath +
            "... " +
            bytesRead.intValue() +
            " byte(s) read";
        );
    }

}
```

Step 3

In order to calculate the estimated total progress percentage, modify the `ProgressReporter` class to store the total number of bytes processed and the total time elapsed from the start of the process. Initialize the `ProgressReporter` with the total number of bytes expected to be read, which should be stored in the root node of the in-memory directory structure. Estimate the time remaining by calculating the average speed of bytes processed per second.

Pausing and restoring

Use the Memento pattern to store the state of the scanning progress. Modify the `HashStreamWriter` class so that it can create a Memento object with its current state and to restore its state from a memento object. Modify the `ProgressReporter` class so that it can store and restore progress information in the Memento object. Use the Observer pattern so that the `HashStreamWriter` and `ChecksumCalculator` classes can detect requests to pause execution. The classes should query an observable at an appropriate interval (e.g. every file and/or every 1 million bytes processed) to check whether there has been an asynchronous request to pause the process.

In order to implement the pausing and restoring behaviour, the scanning process should be implemented in a separate thread, and the user can enter pause commands in the main thread.