Checksum Calculator

Atanas Semerdzhiev, Kalin Georgiev, Petar Armyanov, Trifon Trifonov

Edition 0.1, 2024-11-04: Still a work in progress

Table of Contents

Project Description

In this project you will develop a simple checksum calculator. It can be used to determine the checksums of one or more files, or to verify the integrity of their contents against a previously generated list of checksums.

The program will receive its input as a list of command line arguments.

At least two modes of operation must be supported (you can implement more if you wish):

1. Checksum calculation.
2. Verification against a list of already generated checksums.

For the purposes of the project, all file operations must be performed in binary mode.

The program will present the user with a progress indicator, which reports simple statistics about what has been processed already and what remains.

There must also be an option to pause a scan at any point in time and to resume it at a later stage.

Checksum Calculation

The default behavior of the program is to calculate checksums.

The user can specify the target item to be analyzed (file or directory) by using the `--path` switch. If this switch is not specified, the program uses the current working directory.

- If the target is a file, only its checksum must be calculated.
- If it is a directory, then its contents must be iterated recursively and checksums must be calculated for all files.

The user can specify which algorithm to apply by using the `--algorithm` switch. Your design should make it easy to add or remove algorithms (e.g., md5, sha1, sha2-512, and others). For the purposes of this project you do not need to implement the algorithms yourself. Instead, you can use an appropriate library.

Verification

In this mode the program uses a list of already calculated checksums and verifies them against one or more existing files. To activate it the user supplies the `--checksums` switch and specifies a file containing the checksums to be used.

The user can explicitly specify the target file or directory to be verified by using the `--path` switch. If this switch is not specified, the program uses the current working directory.

Please, note that four cases are possible:

1. A file that is in the target does not have an analog in the checksum file (new file).
2. A file that is in the checksum file does not have an analog in the target (removed file).
3. A file that is in the target has an analog in the checksum file, but the checksums do not match (changed file).
4. A file that is in the target has an analog in the checksum file and the checksums match (unchanged file).

In this case the program should list each file on a separate line and indicate its status. For example:

```
> checksum-calculator --checksums=checksums.txt
./test/file1: OK
./test/file2: MODIFIED
./test/file3: NEW
./test/file4: REMOVED
./test/file5: OK
```

Error-Handling

When implementing the program, proactively think about the possible errors that can occur and how to handle them. For example, what should happen if the user specifies a non-existent file or directory? Or if the user supplies the `--checksums` switch, but specifies a file that is not a checksum file? Or if the user specifies a target that cannot be read?

The program must graciously handle all such cases and provide meaningful error messages to the user.

If the error is critical and the program cannot continue, it must exit with a non-zero exit code.

Symbolic Links

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

Progress Indication

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

For the purposes of the project, implement this functionality in three steps. Each step will require you to revisit the design of the solution and adjust it to accommodate the more complex task.

Step 1

The progress indicator simply displays 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 for the entire task (i.e., number of bytes processed so far divided by the number of bytes in all files that need to be processed).

Report Formats

Add support for multiple report formats. For example, the user can specify that the output should be in Text, HTML, XML, Markdown or JSON format.

For the purposes of the project implement at least two formats. One of them must be the plain text format used in the `*sum` family of applications in Linux:

> The default mode is to print a line with checksum, a space, a character indicating input mode ('*' for binary, ' ' for text or where binary is insignificant), and name for each FILE.
>
> — Linux Man Pages, md5sum

Here is what a sample output from `md5sum` may look like:

```
> md5sum /bin/* -b
bf1cde9c94c301cdc3b5486f2f3fe66b */bin/zip
72dcfc86d29028446e52f33a8937cd69 */bin/zipcloak
b820b8cb1e28403decc1734fc19a26d0 */bin/zipdetails
ddf51a6f1d31b015b654033bd027c94a */bin/zipgrep
7ac7b307c04fcd5ee4a4bb6f3c0b70b5 */bin/zipinfo
```

Decide for yourself what other format(s) to implement. For example, here is what an XML report may look like:

XML

```xml
<checksums>
  <item>
    <mode>binary</mode>
    <checksum>bf1cde9c94c301cdc3b5486f2f3fe66b</checksum>
    <path>/bin/zip</path>
    <size>203768</size>
  </item>
  <item>
    <mode>binary</mode>
    <checksum>72dcfc86d29028446e52f33a8937cd69</checksum>
    <path>/bin/zipcloak</path>
    <size>72088</size>
  </item>
</checksums>
```

Add support for an additional switch called "--format" that will allow the user to specify the desired format.

When the switch is not specified, the program should default to the simple text format described above.

Hints

Command Line Processing

It is not necessary to manually parse the command line arguments. Instead, you can use an appropriate library. For example:

Java

- Apache Commons CLI

C++

- Templatized C++ Command Line Parser Library (TCLAP)

C#

- Parse the Command Line with `System.CommandLine` | Microsoft Learn

Individual file checksums

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

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

Implement subclasses 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`
- Apache Commons `DigestUtils`

C#

- `System.Security.Cryptography` Namespace
- C# – Get a file's checksum using any hashing algorithm

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, to verify the behavior of the calculators, 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

One of the requests is to have a progress indicator that reports the actual percentage of work completed. To accommodate this, we would like to build an in-memory representation of the directory structure to be traversed. This will enable us to 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 each file in bytes. For directories store the sum of all sizes of the items it contains (may be either files or directories).

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 example:

Java

- How do I iterate through the files in a directory and it's sub-directories in Java? | Stack Overflow
- Interface FileVisitor<T> | Oracle
- Class SimpleFileVisitor<T> | Oracle

C++ * Filesystem Library (since C++17) | cppreference.com * `<filesystem>` | Microsoft Learn

C#

- How to: Enumerate directories and files | Microsoft Learn
- Best way to iterate folders and subfolders | Stack Overflow

For testing purposes, design `HashStreamWriter` to write its results to a stream. 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 directory-iterating 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 hard-coding 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 for Java.

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 the `visitDirectory` 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.

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

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 below for an example.

Note that at this point 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.

Skills and Competencies

Programming concepts:

- Assertions
- Exception handling
- Exception safety guarantees
- Invariants
- OOP principles
- Polymorphism
- SOLID
- Streams
- TDD
- Unit testing
- Working with the filesystem
- Hash functions, checksums

Design Patterns

- Composite,
- Observer,
- Strategy
- Template method
- Builder
- Visitor
- Memento