JCVSReport

Introduction

JCVSReport will help you find out interesting and vital statistics that detail your group's progress throughout its project life cycle on any of its CVS based projects. These statistics can be used to help optimize your project's workflow on current and future tasks. What's more, all of this functionality is available almost immediately out of the box. And most features can be customized by modifying nothing more than a single line. So while there may be other software packages out there that claim to perform similar features, we are confident that you will find JCVSReport to be a better choice. That is, when compared to other software packages, JCVSReport is more:

- **Functional**: JCVSReport is a highly functional piece of software. That is, the possibilities for statistical collection are almost endless. A metric can be simple, and count the number of lines of code in a particular project over time, or complex and measure Noncomment Source Statements (NCSS), or even so far as to measure *Cyclomatic Complexity*. In fact, our software comes with 18 prepackaged metrics, outlined in our *prepackaged metrics* section below.

- **Configurable and Customizable:** While our software does contain 18 pre-packaged metrics, one may only be interested in a subset of these at any given time. This is done by performing a simple modification to one line in a single configuration file, supplied at the command line. Do a few particular users need to use a different repository, or have some other differing requirements? No problem, all they need to do is specify a separate configuration file while running the program. One can also easily modify graph sizes, titles, whether it is a line, bar, or pie chart, whether it is sampled over time or per developer, etc.

- **Simple to Use:** We have designed our software with simplicity in mind. This means that we use automatic configuration wherever possible. For example, all database queries are automatically generated. When user input is required, it is minimal. That is, most desired functionality and configuration changes can be achieved by changing a single word or line.

- **Expandable:** No single person could predict every kind of statistic one may want to use. As such, we have designed our system to be expandable. For example, if one wanted to collect the number of lines containing a certain keyword pertaining to their particular project, this could be done through the straight-forward creation of a new 3 line .properties file. Or perhaps one would like to find the number of occurrences of loops nested more than two or three levels deep. Again, this could be specified through a new .properties file.

As an evidential example to these 4 claims, consider the Carnegie Mellon Software Engineering Institute's definition of cyclomatic complexity[1], a metric that we include:

> "Pioneered in the 1970s by Thomas McCabe of McCabe & Associates fame, cyclomatic complexity essentially represents the number of paths through a particular section of code, which in object-oriented languages applies to methods. Cyclomatic complexity's formal academic equation from graph theory is as follows:
>
> $CC = E - N + P$
>
> where E represents the number of edges on a graph, N the number of nodes, and P the number of connected components."

The cyclomatic complexity metric is as useful as it is complex. So how do we implement it? With a single regular expression, stored in a .properties file. That's the power of JCVSReport!

What has been Implemented

Our project has been implemented through several well-defined packages:

1) `ca.utoronto.JCVSReport`: This package contains Presentation.java, which is what is invoked from the command line. It contains other files which parse the configuration file, and from its contents will dynamically call the appropriate classes contained in the packages below.

2) `ca.utoronto.JCVSReport.report`: This package contains all generalized code related to bar graphs, pie charts, and time series (line) graphs. It is not linked to any particular metric, but instead serves as a simple graphing interface to the rest of the system.

3) `ca.utoronto.JCVSReport.report.properties`: This package consists entirely of graphing `.properties` files. That is, *`filename`*`.properties` lists all properties of a graph including its title, dimensions, labels, and the metrics that will be included in the final report, if *`filename`* is included from the configuration file. All metrics specified in this .properties file are found in the `ca.utoronto.JCVSReport.metric.properties` package. For further information on report `.properties`, please see the installation document.

4) `ca.utoronto.JCVSReport.metric.properties`: Containing only .properties file, the packages purpose is to explain how to collect the specified metric. For example, the `LinesOfCode.properties` file contains:

[1] http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html

```
class=ca.utoronto.JCVSReport.metric.RegexLineCounter
title=Lines of Code
regex=\\n
```

The class makes reference to the metric collection engine. As the project currently stands `.RegexLineCounter`, `.RegexMatchCounter` and `.SyntaxTreeRegExMatchCounter` are the three metric collection engines. For further information on metric `.properties` files, please see the installation document.

5) `ca.utoronto.JCVSReport.metric`: This package contains all files related to finding metrics. More specifically, it contains two classes:

```
ca.utoronto.JCVSReport.metric.RegexLineCounter
ca.utoronto.JCVSReport.metric.SyntaxTreeRegexMatchCounter
```

These two classes act as the engine that collects statistics based on the information contained in its supplied metric `.properties` files, as described above.

`RegexLineCounter` matches simple regular expressions against its supplied .java source files. `SyntaxTreeRegexMatchCounter` also matches regular expressions. However, instead it matches them against an abstract syntax tree representation of the respected .java source file. For more information on how to write your own customized metrics, please see the installation document.

Current Metrics:

It was mentioned previously that our program contains many metrics. We will now describe what they are. To include one with your report, simply add the name to the `graph=` section of your configuration file.

- **NCSS:** The number of non-commenting source statements.
- **CyclomaticComplexity**: The number of possible execution paths through your code.
- **LinesOfCode**, **LinesOfComments**, **LinesOfJavaDoc**, **LinesWithTabs**, and **LinesWithTrailingSpaces** are straightforward.
- **NumberOfClasses**, **NumberOfMethods**, and **NumberOfImports**, all count the number of occurrences of class, method, or import declarations respectively.
- **NumberOfTestClasses**, and **NumberOfTestMethods** count the number of test classes and test methods respectively. To do this, the regular expressions consider any class whose name ends with ‘test’ to be test classes.

Furthermore, if the class extends `'testCase'`, it is also considered a test class. All methods that start with "test" are considered to be test methods because that is the standard JUnit syntax.

The above 12 metrics are single metrics. That is, if included, only a single metric will be displayed on the graph. There are 8 predefined graphs which include multiple metrics, as follows:

- **CodeComplexity**: Includes the CyclomaticComplexity, NumberOfMethods, NumberOfClasses metrics. By looking at the relation between the cyclomatic complexity of your software and the number of methods contained within it, you can determine the average complexity of each method in your code.

- **CodeSize**: Includes LinesOfCode and NCSS metrics

- **Conflicts**: Describes the number of conflicting or merged updates An update is considered to be merged if two people make changes to the same file and CVS automatically merges them. If the changes cannot be merged automatically then it will be considered conflicting. This information is gathered from the CVS history file.

- **Coupling**: Displays NumberOfImports, and NumberOfClasses. By looking at the relation between these two metrics, you can get a quick estimate of the coupling of your software.

- **DocumentationSize**: Displays LinesOfComments and LinesOfJavaDoc

- **Lines**: Displays LinesOfCode, LinesOfComments, LinesOfJavaDoc, LinesWithTabs, and LinesWithTrailingSpaces

- **Operations**: Find the number of *updates* and *commits* in the repository.

- **TestSize**: Displays NumberOfTestMethods and NumberOfTestClasses

By default, all statistics are sampled over time, and displayed as a line graph. This behaviour can be changed by appending one of several extensions listed below. The extensions are as follows:

- **`–BarGraph:`** Displays your final results as a bar graph instead of a time series. This graph only is applicable when you are displaying multiple metrics on a single graph.

- **`–PieGraph:`** Displays your final results as a pie chart instead of a time series. This graph only is applicable when you are displaying multiple metrics on a single graph.

- **`-PerDeveloper`**: The contribution of each developer to any of the above metrics can be easily viewed by appending "-PerDeveloper" to the metric name. For example, you can view how much each developer contributed to the number of lines of code in the project by listing "LinesOfCode-PerDeveloper" as one of your metrics. If you are displaying only one metric, then these results will be displayed on a pie chart. If you are displaying multiple metrics, then these results will be displayed as a bar graph.

- **`-PerDeveloperBarGraph`**: This is the same as above, except that it will be forced to be displayed as a bar graph.

- **`-PerDeveloperTimeSeries:`** This option will allow you to display your results Per Developer, as a time series. This graph is only applicable when you are displaying a single metric on a single graph.

What has been tested

We took a two-pronged approach to testing JCVSReport:

- **Unit Tests**: We wrote unit tests for all of the major classes in JCVSReport. These unit tests typically instantiated a few examples of each class and tested the boundary conditions on the functions.

- **Functionality Tests**: To facilitate writing test cases, we created a test repository and manually counted each statistic on that repository. We then wrote test cases which compared our manual results to the JCVSReport results. This gave us a basis to check the reliability of our metrics, and to exercise our collection engines.

Test Directory

All of the unit tests and functionality tests are located under the `408project/test` directory. This directory has the same structure as the `408project/src` directory and all of the tests for source files in the `src` directory are located under the same relative path under the test directory.

The test directory also includes a sample CVS repository that was used for functionality testing, as explained above.

CVS Repository for testing

The sample CVS repository can be found within the `408project/test/test_cvsrepository` folder. This test repository contains one module called `project` for which statistics were manually gathered, and compared against those collected through our software, during functionality testing.

This testing project contains two java files, one JUnit test file, one Python file, and a single C file. The Python and C file were included to make sure that JCVSReport will still work with repositories that contain other source files, in addition to java files. The `test_cvsrepository` directory also contains a `README` file containing statistics for the testing project that were calculated by hand for each of the revisions of the java file in the testing repository.

Our tests have been implemented in the following five packages.

- **ca.utoronto.JCVSReport**
  - Tests the configuration file interface
  - Tests the database connection interfaces
  - Tests the main program class (Presentation)

- **ca.utoronto.JCVSReport.cvsPlugin**
  - Tests the CVS history parser
  - Tests the CVS plugin

- **ca.utoronto.JCVSReport.metric**:
  - Tests the three different metric gathering engines, RegexLineCounter, SyntaxTreeRegexMatchCounter, and RegexMatchCounter.
  - Tests internal representations of metric data, including our metric revision map and set classes

- **ca.utoronto.JCVSReport.report**:
  - Tests all basic graph functionality, and time series graph functionality.
  - (A current limitation is that our pie and bar graph are not tested in our unit tests, but are tested in our functionality tests)

- **ca.utoronto.SimpleProcess**:
  - We test that our program will correctly handle commands that don't exist, do exist, don't work, etc.
  - We make sure that our class for managing different buffers (StreamGobbler) functions within specifications.

Current limitations:

While the capabilities of our program are quite impressive, our program does have some limitations. We outline these limitations below:

- For the time series graphs, our sampling granularity is over each day. While it would be nice to be able to sample at coarser or finer details, this has not yet been implemented.
- When sampling over time, statistics collection begins when the CVS repository began, and continues until the current date. Future

implementations may allow one to specify different starting and stopping dates.

- The `SyntaxTreeRegexMatchCounter` engine will only work if it is possible to compile the respective .java file. This is because it relies on a Java compiler to create an Abstract Syntax Tree which can be parsed using regular expressions. Since a compiler cannot produce this tree if there are syntactic errors in the code, we haven't the ability or the plans to fix this issue. However, we have taken special provisions in our code to skip over the problematic file in question, if it cannot compile.
- We provide the option to update our database system instead of rebuilding the entire thing, by supplying a `-u` option from the command line. Currently, this only functions properly if the database actually exists, and that the specified configuration file does not request new metrics than what were previously collected.
- Our system can only store the statistics in relation to a single repository at a time.
- While we provide an easy to run `.jar` file, newer metrics cannot easily be added or modified using this method. This is easily fixed however if the user recompiles using `ant`, as described in our installation document.
- Our software has been extensively tested with Java 1.4, and no other version.
- We haven't directly tested the Bloof system very much, but have tried to mitigate any potential issues by testing it against our own functionality.
- Our software only works with the official CVS distribution. This distribution comes standard with many Linux distributions, and it is also available for Windows via the Cygwin package. Our software does not work with non-standard CVS distributions such as CVSNT.