# Installing and Running JCVSReport

## Step #1: Check Prerequisites

In order to run JCVSReport, you must have the following:

- CVS: The official version of CVS must be installed and in your path
- Java: You need version 1.4 of Java.
- A RAW copy of a CVS repository.

In order to compile JCVSReport, you will also need:

- A copy of the JCVSReport software source files.
- Ant: We tested it with Ant 1.6.1.
- Java Compiler: Version 1.4 of the Java SDK at a minimum

If you are missing one or more of the above prerequisites, please see **Appendix A: Installing Prerequisites**.

## Step #2: Setup your configuration file

For JCVSReport to run, it requires a configuration file to be specified on the command line. The file tells our software which CVS repository to use, as well as which metrics you'd like included in your final report.

Below is a sample configuration file, sample_config.txt, followed by an explanation of how to customize it to your personal uses:

```
# Change cvsroot to point to your local CVS repository.
cvsroot=/u/csc408h/winter/pub/repo/c408h11

# Change cvsmodule to be the name of your project inside
# your CVS repository
cvsmodule=408project

# A list of the graphs you want to generate
graphs=CodeSize,CodeComplexity,TestSize,DocumentationSize
```

**cvsroot**: The path must refer to the repository location. In the above case, we show the path to our CVS repository on CDF.

**cvsmodule:** This refers to the name of the project within the `cvsroot` directory that you would like to gather statistics on. In our case, the project name was "408project".

**graphs:** This is where you set which graphs are to be generated and displayed. The graphs are specified through a comma delimited list, where the contents refer to the name of the `.properties` file. In this case, `CodeSize` refers to the `CodeSize.properties` file, located within the `ca.utoronto.JCVSReport.report.properties` package. See

**Appendix B: Graph Configuration** for more information on the graphs that can be displayed here.

## Step #3 – Executing JCVSReport

If you are working within a local checkout of the JCVSReport repository, simply go to that directory and type:

```
/bin/sh JCVSReport –c sample_config.txt
```

This will run either the packaged JAR file, or your latest compile. The `–c` option specifies where the configuration file can be found. An optional option `–u` can also be specified to tell JCVSReport to simply update its database, and not recreate it from scratch.

If you would like to test JCVSReport, simply type:

```
/bin/sh JCVSReport –t
```

After the tests are run, the application will let you know whether or not the tests were successful. If you would like extended information on any errors that might be reported, you will need to install Eclipse and run the tests manually from within the IDE.

If you would like to recompile JCVSReport, simply type 'ant'. A new directory called 'dest' will be created with the contents of your compile. The provided shell script will use this compile in place of the JAR.

## Step #4 – View your report

After following all of the above, a report.html file will be written out in the same directory that the program was run from, along with all of the images that report.html makes reference to. You can view this report using any standard web browser.

# Appendix A: Installing Prerequisites

## Copy the RAW CVS Repository to your local system.

For the purposes of this course, the CVS repository will be located at:

```
/u/csc408h/winter/pub/repo/c408hXX
```

Where *XX* would be replaced by the numerals assigned to your group account by the Professor. Make sure to copy the whole folder to a local folder. This step is necessary because our system gathers statistics from the raw repository, which in the case of a local install is not accessible without this copy.

The easiest way to do this is probably to `tar` up the repository and then transfer it over using SCP or SFTP. This can be done with:

```
    tar -czf ~/cvsrepository.tgz $CVSROOT
```

In the case that the $CVSROOT environment variable is not set (or in the case that it refers to a remote file system), one can achieve the same thing with:

```
    tar -czf ~/cvsrepository.tgz /u/csc408h/winter/pub/repo/c408hXX
```

When you extract the archive to your own local directory, you must make sure that the path to it is at least 10 characters long, due to some limitations in the bloof database that is part of our system. If the path is not long enough, you will get the following error when the software first starts up:

```
net.sf.bloof.scm.cvsplugin.InvalidRepositoryLocationException:
The repository location is null or too short:
```

Furthermore, make sure to use the usual Linux tar software to extract, or cygwin's tar. Other windows applications such as WinRAR seem to damage the repository if the paths are too long.

### Install CVS on your home machine.

Our software makes extensive use of the CVS package, and as such its installation is very important.

If you are using Linux, you will need the official distribution of CVS. This will likely be available with your Linux distribution but it can also be downloaded from www.gnu.org/software/cvs/.

If using a windows based machine, then it is best to use Cygwin port of CVS (www.cygwin.com), as some of the other distributions of the software (such as cvsNT) suffer from compatibility issues with our software.

*Note:* Make sure that you add the CVS binary to your path, so our system can run CVS easily.

## Appendix B: Graph Configuration

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.

## Appendix C: Advanced Graph Configuration

Above it was explained that the final rendered document will contain graphs, specified in the `sample_config.txt` file via the `graphs=` property. It is yet to be explained how this is mapped to an actual graph. To demonstrate this, we will go through an example. Lets say we defined `graphs=Lines,CodeSize`

JCVSReport will then look in the package directory `ca.utoronto.JCVSReport.report.properties` for two files: `Lines.properties`, and `CodeSize.properties`.

The first file `Lines.properties` is a predefined metric, packaged with the software. We will go through its contents to learn how to create new metric property files, such as `MyNewMetric.properties`.

### Specifying Graph Characteristics

The first 5 lines, `title`, `ylabel`, `xlabel`, `height`, and `width`, all specify the graphs title, y-axis label, x-axis label, and the graphs height and width respectively. For example:

```
title=Lines of code over time
ylabel=Lines
xlabel=Date
height=300
width=500
```

## Specifying Graph Metrics Sources

The next line contains a `metric` attribute, which specifies which .properties metrics will be used in collecting data. These other .properties files are located within the `ca.utoronto.JCVSReport.metric.properties` package. For example:

```
metrics=LinesOfCode,LinesOfComments,LinesOfJavaDoc
```

The above would refer to `LinesOfCode.properties`, `LinesOfComments.properties`, and `LinesOfJavaDoc.properties`, all within the `ca.utoronto.JCVSReport.metric.properties` package. All .properties files within the `ca.utoronto.JCVSReport.metric.properties` package are titled *metric properties files*, and will be explained in more detail in the next section.

## Metric Properties Files

*Metric Properties Files* specify exactly what is to be matched and counted, and where from. For example, the previous report `.properties` file mentioned above, `Lines.properties`, had a line:

```
metrics=LinesOfCode,LinesOfComments,LinesOfJavaDoc
```

And we already know this means it looks for LinesOfCode.properties in the `ca.utoronto.JCVSReport.metric.properties` package. A typical metric properties file contains three lines, and looks like the following:

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

The first line specifies what class should be used as the matching engine. In this example, we use `RegexLineCounter`, which uses regular expressions against a plain text .java file. We can see from the third line that this metric just matches to the end of a line character.

The `title` property just sets the default name for the metric in the event that it is included directly from the sample_config.txt file, instead of through the report .properties file described earlier.

Lets see another example of a metric .properties file that uses the `RegexLineCounter` class. In this case, we will look at `LinesOfComments.properties.`

```
class=ca.utoronto.JCVSReport.metric.RegexLineCounter
title=Lines of Comments
regex=(?:/\\*(?s:.*?)\\*/|//).*\n
```

It is identical to the previous file, except for the regular expression which is used to match to comments.

The `class` line can also point to:

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

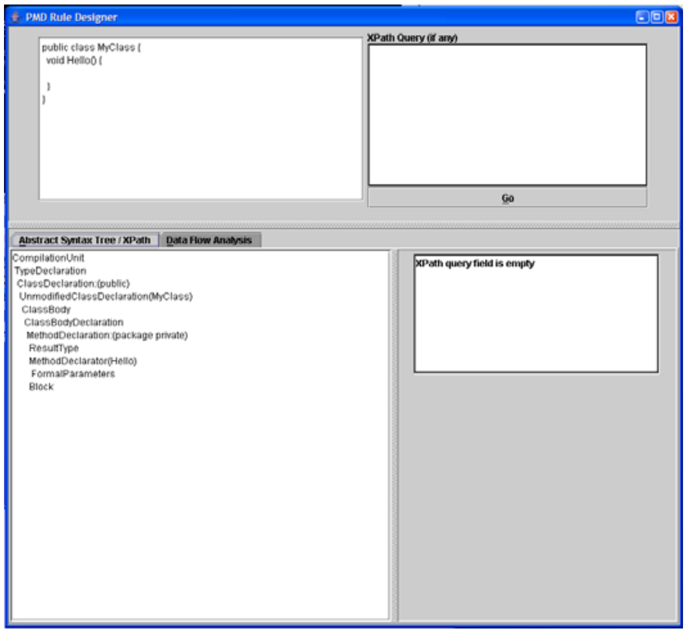
Which is the other matching engine of JCVSReport. This matching engine also uses regular expressions, but it uses them against an *abstract syntax tree* generated by a java compiler. Let us look at the file `NumberOfClasses.properties` which counts the number of classes in a given file.

```
class=ca.utoronto.JCVSReport.metric.SyntaxTreeRegexMatchCounter
title=Number of Classes
regex=ClassDeclaration:
```

As can be seen, these metrics are easily written. You may be wondering how we knew to use `ClassDeclaration:` as our regular expression. One can find these things out by downloading the PMD tools from http://pmd.sourceforge.net/, extracting, and running

```
pmd-2.3\bin\designer.bat or pmd-2.3\bin\designer.sh
```

A window is open, in which you can put java source code. You can click on a 'go' button, and the resulting parse tree is shown in another window pane. By exploring around with different pieces of code, you can learn what kind of parse tree to expect, and design a matching regular expression for that. A screen capture is shown on the right.



```
public class MyClass {
 void Hello() {

 }
}
```

```
CompilationUnit
 TypeDeclaration
  ClassDeclaration:(public)
   UnmodifiedClassDeclaration(MyClass)
    ClassBody
     ClassBodyDeclaration
      MethodDeclaration:(package private)
       ResultType
       MethodDeclarator(Hello)
        FormalParameters
       Block
```

XPath query field is empty

Clearly these combined tools are extremely powerful in coming up with new metrics.