Unified Code Count – Java (UCC-J)

User Guide

v.2018.05

May 2018

Version History

Date	Author	Version	Changes
11/6/2016	CSSE	1.0	Initial Version
5/30/2018	CSSE	1.1	Updates to Setup Instructions

Table of Contents

1 Introduction

This document provides information for using the UCC-J version 2016.

1.1 Product Overview

Most software cost estimation models including the COCOMO® model require some sizing of software code as an input. Ensuring consistency across independent organizations in the rules used to count software cost code is often difficult to achieve. Given this context, there are two primary uses of the UCC-J system – (1) counting source lines of code (SLOC) and (2) differencing baselines. An end user of the UCC-J can use the system to count source lines of code to collect metrics or use it to difference two baselines and generate differential reports.

UCC-J counts SLOC (source lines of code) in accordance with Software Engineering Institute's (SEI) code counting standards. UCC-J output metrics include Physical and Logical SLOC, blank lines, comments, compiler directives, executable instructions, keywords, differencing, duplicates, and cyclomatic complexity.

UCC-J is intended so that non-technical as well as technical users can easily obtain SLOC and metrics results. The US Government is an advocate of UCC-J as a standard software metrics tool.

UCC-J is developed using Java programming language to make it portable and to allow it to run independent of end user's environment configuration. It is currently distributed as an executable Java Archive (JAR) file which runs in a Java Virtual Machine (JVM). Since Java is the implementation language, a Java VM will be needed to run UCC-J. End users need to have Java Standard Edition (SE) Runtime Environment (JRE) 8 installed in order to successfully execute UCC-J. The user is responsible for using the executable version.

2 System Requirements

The following hardware, operating system, and software configurations and capabilities are recommended to successfully execute UCC-J.

2.1 Hardware

The following hardware capabilities are needed to use UCC-J:

- RAM: 512MB
    - Recommended JVM RAM Allocation for running UCC- G on baselines up to 1,000 files (~200 MB)
        - Minimum: 64MB

        - Recommended: 128MB
    - Recommended JVM RAM Allocation for running UCC-J on baselines larger than 10,000 files (~2GB)
        - Minimum: 256MB
        - Recommended: 512MB
- Hard Disk Free Space:
    - 2x the space of the file list or directory for a counter run (for a 100MB file list, 200MB is required)
    - 2x the space of each file list or directory for a differencer run (for a 100MB directory A and a 150MB directory B, 500MB is required)

2.2 Operating Systems

UCC-J will run on following operating systems:

- Windows
- Linux (Redhat Enterprise, Fedora, Debian, Suse Enterprise, Ubuntu, ...)
- Unix (HP-UX, IBM AIX, Sun Solaris, IRIX, ...)
- Mac OS

2.3 Software

Below software packages need to be pre-installed for UCC-J to run:

- Java Runtime Environment version 8 or higher
- Java 8 System Requirements:
    - RAM: 128 MB
    - Disk Space: 124 MB for JRE; 2 MB for Java Update
- Eclipse Mars or later (http://www.eclipse.org/downloads/packages/ Eclipse IDE for Java Developers)
- Log4j 2

3 Installation

3.1 JRE Installation

Java SE Runtime Environment is needed in order to run Java programs. UCC-J runs with JRE version 8. It can be downloaded at - http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html. Select the correct Operating System (OS) (32-bit or 64-bit) and install once downloaded.

4 UCC-J Compilation

The UCC-J source code is made available on https://csse.usc.edu/ucc_wp/ . Download the source code along with all external libraries provided with them.

Note: UCC-J is also distributed as an executable Java Archive (JAR) file as part of UCC-J's release. Compilation is not required if users prefer to use the JAR file. They may skip to Section 5.

4.1 Updating the Default JRE in Eclipse

Once Eclipse is opened, verify the Java SE Development Kit 8 from step 2 above is set as the default Eclipse JRE. To do this, click on *Window -> Preferences -> Java -> Installed JREs*.

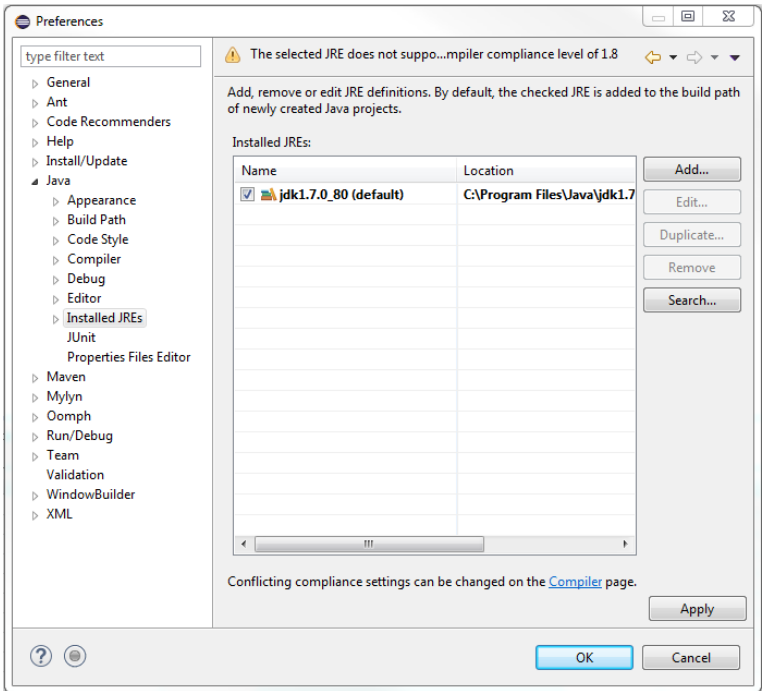

Figure 1 – Eclipse Installed JRE Options Tab

Here, verify the default JRE is the one from Step 2 (*jdk1.8.xx)*, if not:

1. Select the *Add* button.
2. Select *Standard VM* and click *Next*.
3. For "JRE home", click the *Directory…* button and navigate to the directory in which you installed the JDK in Step 2.
   For example C:\Program Files\Java\jdk1.8.0_66
4. The "JRE name" field should now be supplied by default
5. Click *Finish*

6. Finally, select the *jdk1.8.xx* **JDK** as the default Eclipse JRE (*For example, NOT jre1.7.xx*), and click *Finish*.

4.2 Importing and Starting UCC-J Project with Eclipse

The next step is to import the project into Eclipse. To do this, we will need to create a new project. Select *File -> New -> Java Project.*

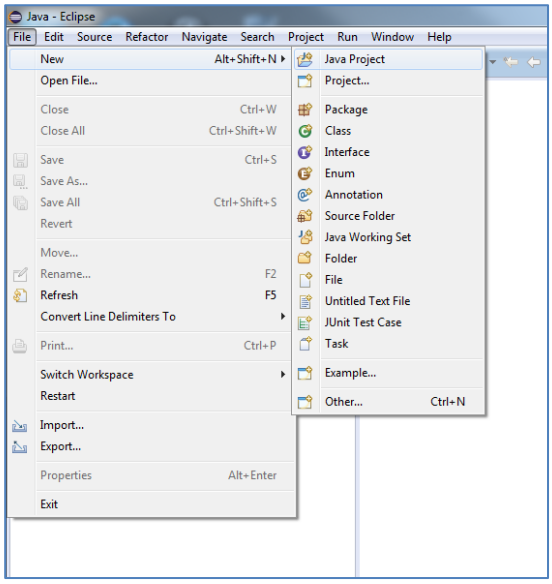

Figure 2 – Creating a New Project – Go-to

The following menu will pop up.

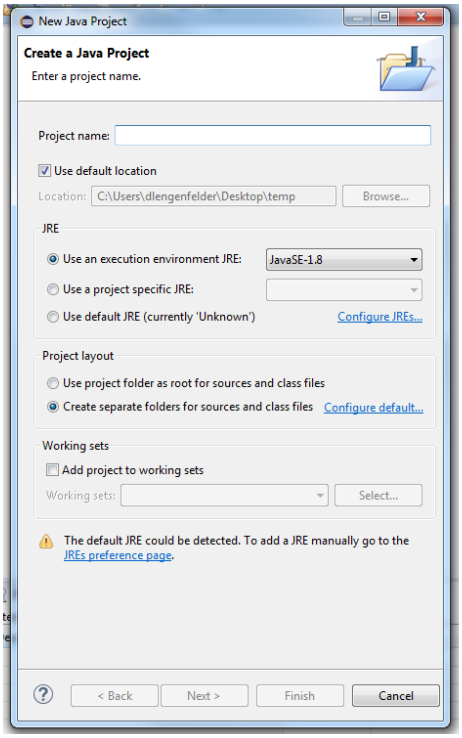

Figure 3 – Creating a New Project – Options Menu

Name the project (*e.g. UCC-J_2018.05*) and select the options depicted below. Make sure to select the project specific JRE radio button and link the JDK installed from Step 2 & 4 above. Also, make sure the *"Use project folder as root for sources and class files"* radio button is selected.

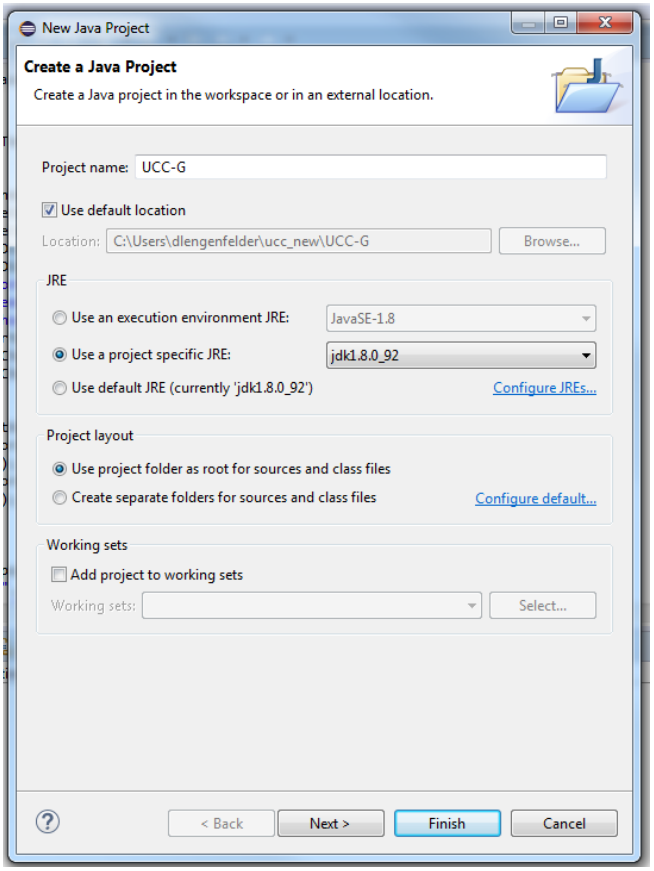

Figure 4 – Creating a New Project – New Project Options

Click “Next”, and you should see a pop-window as depicted below.

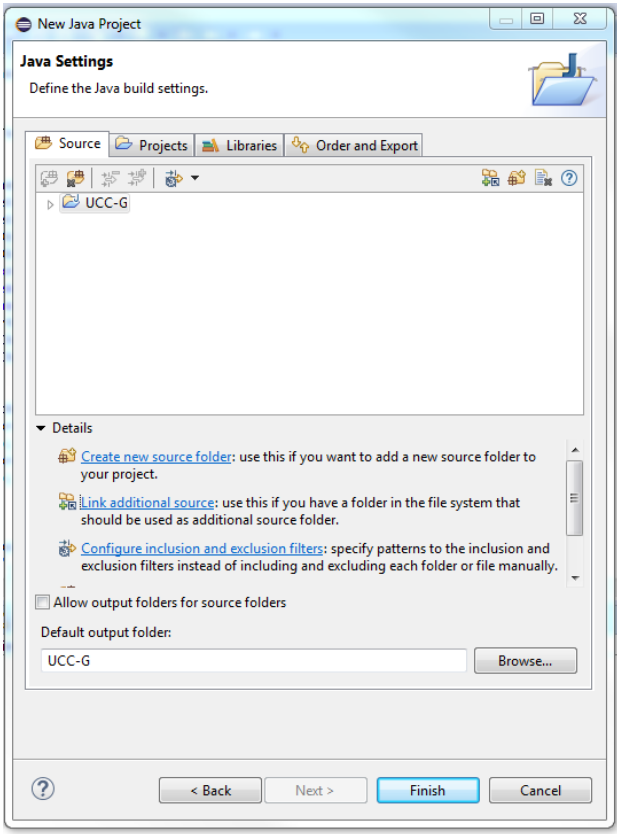

Figure 5 – Creating a New Project – New Project Options Continued

Click on "Link additional source". We will need to pull the source code into this new code hierarchy. You should get the following pop-up window.

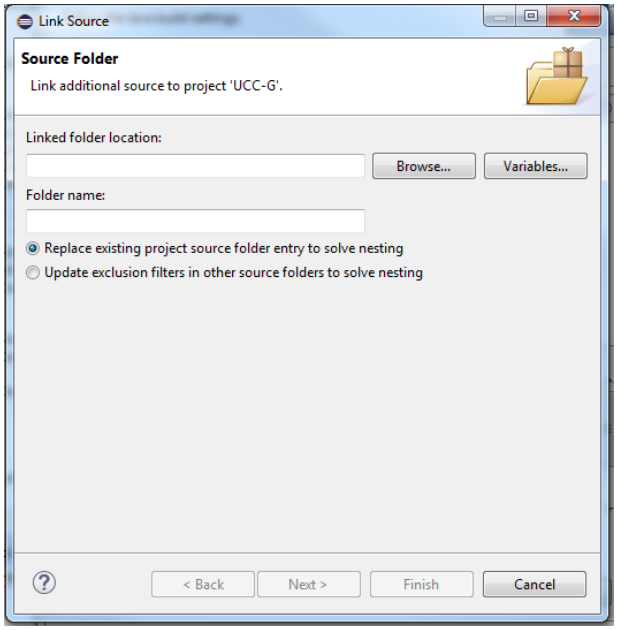

Figure 6 – Creating a New Project – New Project Options Continued

Click “Browse”, and navigate to the folder containing the source you wish to import *(For example: ucc-j_2018.05/src)*. Click *“Finish”.*

Next, select the “Libraries” tab, and click the “Add External JARs…” button. Navigate to the project’s “lib” folder as downloaded from SVN (*…/U2C2/UCC_X/lib)* where you will find three jar files.

1. commons-lang3-3.4.jar
2. log4j-api-2.5.jar
3. log4j-core-2.5.jar

Select all JARs located in the lib folder (<shift>+select) and click the Open button. (More about Log4j can be found in section 9 below.)

Finally, click the “Finish” button to complete your environment setup.

4.3 Importing the Code Style Formatter into the Eclipse Project

To import the Eclipse code style formatter to the project, select
Project -> Properties -> Java Code Style -> Formatter

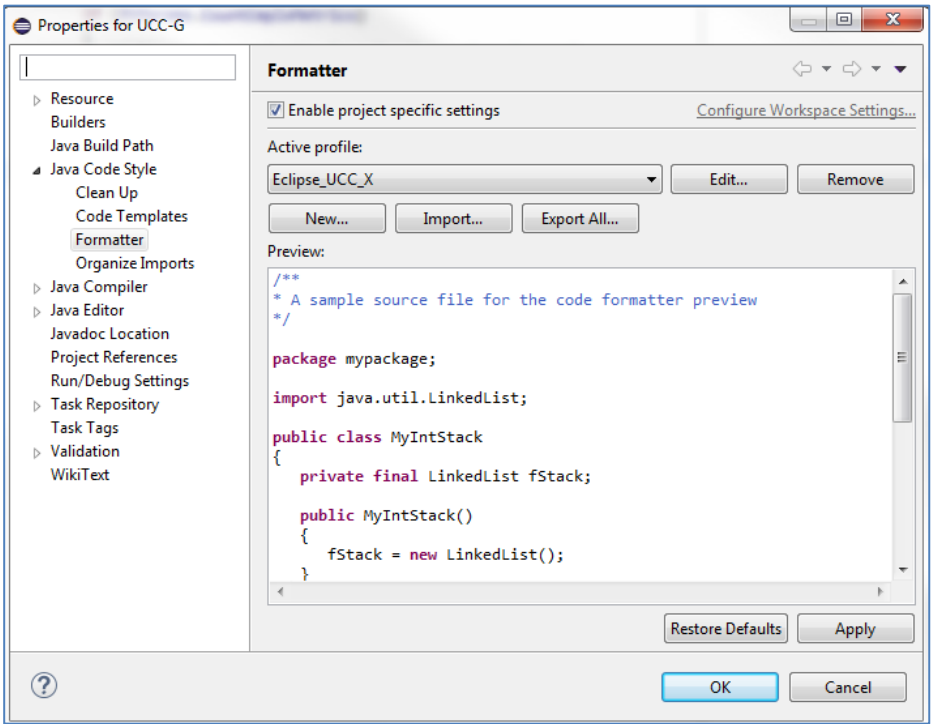

Figure 7 – Importing Eclipse code style formatter

Then, on the Formatter window (See **Figure 7** above)
1) Select **“Enable project specific settings”** checkbox
2) Click **Import** button

3) Select **ucc-j_2018.05/UCC_X_EclipseFormatter.xml**
4) Click **Apply** and then click **OK**

To format a project with the imported Code Style Formatter (See **Figure 8**):

1) Select project
2) Click **Source**
3) Click **Format**

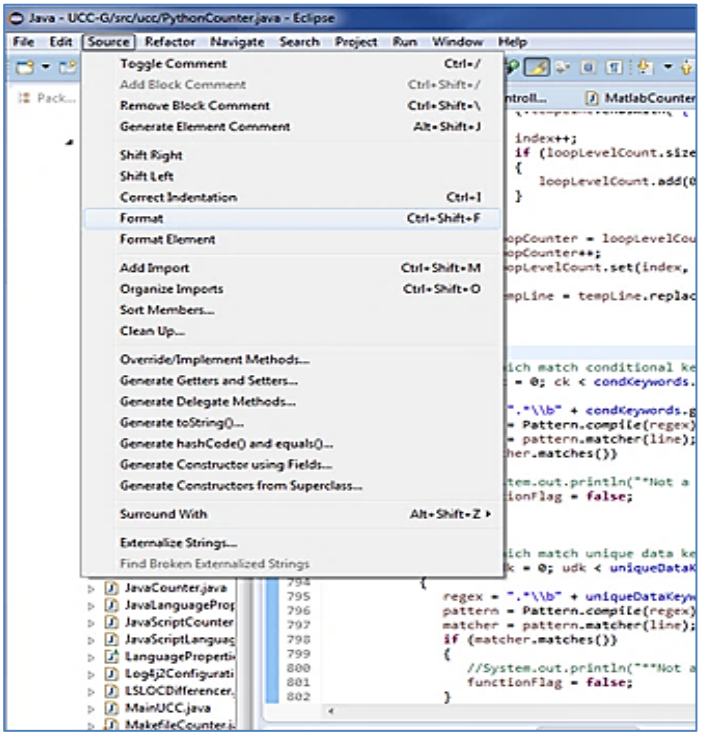

Figure 8 – Formatting an entire Eclipse project

To format a single file in the project (See **Figure 9**):

1) Right click on the file to format
2) Click **Source**
3) Click **Format**

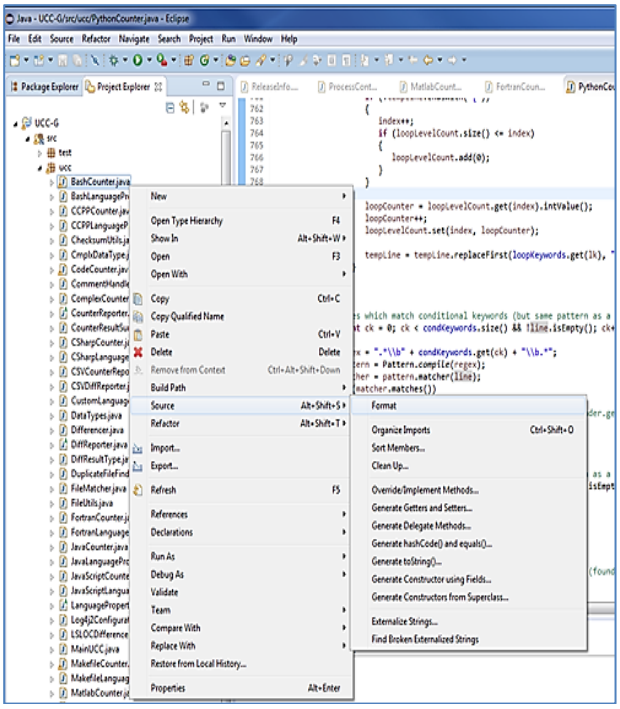

Figure 9 – Formatting a single file

5 UCC-J Execution

To execute UCC-J with the JAR file, run below command from where the JAR file is stored along with switch options described in section 5.1.

java –jar <jar file name> <switches>

For example, to run counting feature of UCC-J on a baseline directory, execute below command:

java –jar ucc-j.jar –dir C:\dev\baselineA\src

5.1 Command Line Switch Options

UCC-J functional user requirements are characterized by the different operations requested by end users. UCC-J operations are specified by end users via use of command line arguments. UCC-J command line format, along with various switches are:

[-v] [-h] [-d [-i1 fileListA.txt] [-i2 fileListB.txt] [-t #]] [-tdup #] [-trunc #] [-cf] [-dir <dirA> [dirB] <filespecs> [-import <file>]] [-outdir outDir] [-ascii] [-extfile extFile] [-unified] [-nodup] [-nocomplex] [-nolinks] [-export <language> [-outdir <outDir>]]

Table 1 lists the switches and their usage notes. For in detail description of what each switch does, refer section 8.

Table 1 – Command Line Switch Options

Switches	Usage Notes
	No argument is given at command line. the tool reads and counts all files listed in the text file *filelist.txt*, *filelist.dat, fileList.txt, or fileList.dat*
-v	Displays the version number of the UCC-J being executed.
-h [<switch_name>]	Displays help information on UCC-J usage with different switch options. Optionally, if –h is followed by a switch, information on how to use that specific switch is displayed.
-d	If specified, UCC-J will run the differencing function. Otherwise, the counting function is executed.
-i1 fileListA.txt	Input list filename containing filenames and/or directories in *Baseline* A to be counted if -d is not present or compared if -d is present. The file is in plain text format, one filename or directory name per line. If a directory name is specified, the directory is searched recursively for all countable files.
-i2 fileListB.txt	Input list filename containing filenames in *Baseline* B. If the -d switch is present, the differencing function will be invoked and fileListA.txt/.dat and fileListB.txt/.dat or filelistA.txt/.dat and filelistB.txt/.dat will be compared. Normally, *Baseline* B is the newer or the current version of the program, as compared to *Baseline* A. The file format is same as *Baseline* A. Used only if the option –d is present.
-t #	Specify the modification threshold, the percentage of common characters between two lines of code to be compared over the length of the longest line. If two lines have the percentage of common characters

	equal or higher than the specified threshold, they are matched and counted as modified. Otherwise, they are counted as one SLOC deleted and one SLOC added. The valid values range from 0 to 100 and default to 60 (same as *-t 60*). Used only if the option –d is present.
-tdup #	Specify the threshold percentage for duplicated files of the same name. This specifies the maximum percent match between two files of the same name in a baseline to be considered duplicates. By default, this threshold is 60 – spacing and comments are not considered.
-dir	Specify directories containing files to be counted and/or compared. If this argument is provided, the input list files are ignored. If the –d is not present; UCC-J looks for dirA and filespecs. If the -d is present; UCC-J looks for dirA, dirB and filespecs. The directories are searched recursively for files that match the filespecs. See section 4.2.1.1 for examples.
dirA	Name of the directory. If the option -d is provided, dirA is the directory of *Baseline* A. Otherwise; it specifies the directory to be counted with the counting function.
dirB	Name of the directory of *Baseline* B, used only if the option -d is given.
filespecs	Specifications of file extensions to be counted/compared; wildcard characters “?” and “*” are allowed. Use a space between *filespecs*; for example, *.cpp *.c
-import <file>	Imports specified file containing language properties for a custom language. The custom language is included when code counter is run on a given directory or a file list.
-outdir <dirname>	Allows user to specify a directory name to be prepended to the output reports.

	The reports will be generated in the specified directory. This allows the user to set up a batch job with multiple UCC-J runs and not have the output reports overwritten.
-extfile <extfile>	Allows user to specify the name of a file that contains languages and extensions to map to that language counter. This allows the user to include non-default extensions, or remove default extensions.
-unified	Directs the UCC-J to print counting results to a single unified language file name TOTAL_outfile.csv.
-nocomplex	Do not process language keywords or report complexity metrics. Using this switch will reduce the processing time.
-nodup	Do not search for duplicate files. Any duplicates will be reported as unique files. This decreases processing time.
-debug <level>	Enables the UCC-J debugger. Outputs are directed to the <time_stamp>_ucc.log file which will appear in the same folder as the UCC-J executable JAR by default if an output directory is not specified. Valid levels are FATAL, ERROR, WARN, INFO, DEBUG, or TRACE. By default, the application is set to the ERROR level. All levels under the specified level will be logged. For example, if the level is set to INFO, any log outputs with INFO, WARN, ERROR, or FATAL level would be outputted and log statements with DEBUG or TRACE would be ignored.
-export <language_name>	Exports language properties for the specified language to a text file. The output text file is stored in the directory specified with –outdir option or defaults to current working directory. Output file name will be [language_name]LanguageProperties.txt.

	language_name used with this switch must match the names defined in Table
-ascii	Generate reports in text format (.txt). Default format is .csv
-cf	Support handling ClearCase filenames. The ClearCase application appends version information to the filename, starting from '@@'. This option requires the UCC-J tool to handle the original filename instead of the ClearCase-modified filename.
-nolinks	Skip Unix symbolic links to prevent multiple counting of same files as links.
-trunc #	Truncate threshold, specifying the maximum number of characters allowed in a logical SLOC. Additional characters will be truncated. The default value is 10,000, and zero is for no truncation. Performance can be significantly degraded if truncation is too high.

Table 2 - Language names to use with -export switch

Language Names		
ADA	HTML	RUBY
ASP	IDL	SCALA
ASSEMBLY	JAVA	SQL
BASH	JAVASCRIPT	VB

C_CPP	JSP	VB_SCRIPT
COLDFUSION	MAKEFILE	VERILOG
COLDFUSION_SCRIPT	MATLAB	VHDL
CSHARP	NEXTMIDAS	XMIDAS
CSHELL	PASCAL	XML
CSS	PERL	
DOS_BATCH	PHP	
FORTRAN	PYTHON	

5.2 Counting and Differencing Details and Examples

5.2.1 Counting Source Files

The counting function is executed using the command line with no *-d* switch. There are two alternative ways to specify the source files of the target program: using the *-dir* switch and using the file list (*fileList.txt*).

5.2.1.1 Using the -dir command line switch

This command requires the tool to count all source files contained in the folder *project1*:

java –jar <jar file name> -dir project1

This command requires the tool to count all C/C++ source files contained in the folder *project1*:

java –jar <jar file name> -dir project1 *.cpp *.h *.c *.hpp *.cc

This command requires the tool to difference all source files contained in the folder *project1* with those contained in *project2*:

java –jar <jar file name> -d -dir project1 project2

This command requires the tool to difference all C/C++ source files contained in the folder *project1* with those contained in *project2*:

java –jar <jar file name> -d -dir project1 project2 *.cpp *.h *.c *.hpp *.cc

Under Unix/Linux when using the –dir option, any wildcards must be enclosed within quotes. Otherwise, the wildcards will be expanded on the command line and erroneous results will be produced. For example: ucc -d -dir baseA baseB *.cpp should be written as ucc -d -dir baseA baseB "*.cpp".

5.2.1.2 Using fileList.txt

java –jar <jar file name>

This command requires the tool to find the default file list named *fileList.txt, fileList.dat, filelist.txt, or filelist.dat* in the working directory and count all source files listed in it.

The default file list, for example, *fileList.txt* contains a list of source files to be counted, one filename per line. You can create the file *fileList.txt* using one of the following commands

- Unix:

  ls -1 <filespecs> > fileList.txt

  or, use the following to obtain full directory/pathname specification

  find *[Directory]* -name '<filespecs>' > fileList.txt

  to append this file with additional filenames use:

  find *[Directory]* -name '<filespecs>' >> fileList.txt

- MS-DOS:

  dir/B > fileList.txt *[Directory]**<filespecs>*
  or, use the following to obtain full directory/pathname specification

  along with files in all subdirectories:

  dir/B/S > fileList.txt *[Directory]**<filespecs>*

  to append this file with additional filenames use:

  dir/B/S > fileList.txt *[Directory]**<filespecs>*

Where,

- *Directory* is the directory name relative to the current working directory. *Directory* is optional, and it is not given, the working directory is implied.
- *filespecs* is the file specifications, and wildcard chars ? * are allowed. Use a space between two filespecs, for examples, *.cpp *.c

5.2.1.3 Specify a filename containing the input list of files

UCC -i1 <fileList>

This command requires the tool to find the file <fileList.txt or fileList.dat> in the working directory and read it to obtain the input source files listed in it. Since the format of these files is the same as fileList.txt, you can use the commands described in 5.2.1.2 to create them.

An advantage to this method is that the input file list can be given a descriptive name, such as JustCppFiles.txt, and multiple input file lists can be stored in the same directory.

5.2.2 Differencing Baselines

In this function, source files in the baselines will be matched and compared to determine the counts for SLOC added, deleted, modified, or unmodified.

To run this function, UCC-J must be called with the *-d* switch.

5.2.2.1 Using List Files

Compare source files of Baseline A and Baseline B contained in files *fileListA.txt/.dat* and *fileListB.txt/.dat or filelistA.txt/.dat and filelistB.txt/.dat.*

java –jar <jar file name> –d

By default, the files *fileListA.txt* and *fileListB.txt* contains source filenames in *Baseline* A and *Baseline* B, respectively. You can specify different filenames by using the *-i1* and *-i2* command line switches.

java –jar <jar file name> -d -i1 fileA.txt -i2 fileB.txt

(Since the format of these files is the same as *fileList.txt*, you can use the commands described in 4.2.1.2 to create them).

5.2.2.2 Using the -dir command line switch

java –jar <jar file name> -d -dir <dirname1> <dirname2> filespec1 filespec2 … filespecn

For example:

java –jar <jar file name> -d -dir code1.0 code1.2 *.cpp *.c *.hpp *.h *.cc

requires the tool to match and compare all source files with extensions of *.cpp, *.c, *.hpp, *.h, or .cc contained in the folder *code1.0* and *code1.2*.

Below is a console output for differencing two linux kernel versions using UCC-J.

```
Processing differencer request...
Processing baseline A counter request...
Checksum calculation execution time: 95.72354 (s)
Performing duplicate identification..............DONE
Duplicate check execution time: 40.97634 (s)
PSLOC and LSLOC Counting execution time: 6368.28522 (s)
```

6 UCC-J Output Files and Results Data Mappings

UCC-J output reports file format will depend on the user input. UCC-J supports .csv and also supports .txt file format specified with –ascii option. Default format is .csv. If a user specifies -ascii option, the reports will be produced in text format (.txt) and the output files will have .txt extensions. Default format is .csv.

Output files for both Counter and Differencer operations with .csv extension as an example are shown in Table and Table below. Please note that if a user selects –nodup option, output files that begin with name "*Duplicate*" will not be generated.

<LANG> is the name of the language of the source files, e.g., C_CPP for C/C++ files and Java for Java files.

Table 3 - Counter Operation Outputs

File Name	Results Data Set
outfile_summary.csv	P-SLOC, L-SLOC, number of files summary per language type. This file also includes a summary at the end with total number of files, total P-SLOC, and total L-SLOC counts in the baseline.
<LANG>_outfile.csv (for ex. C_CPP_outfile.csv, JAVA_outfile.csv, etc.)	Detailed line counts – total lines, blank lines, comments lines, compiler directives, data declarations, executable instructions, P-SLOC and L-SLOC per <LANG> type files in the baseline. This file also contains a total tally of all the

	above counts for all types of <LANG> type files in the baseline.
outfile_cplx.csv	Complexity metrics – number of math operations, logical operations, preprocessors, assignment operations, pointer operations, and level 1 through level 5 loops - summary for each file in the baseline.
outfile_cyclomatic_cplx.csv	Total, average, and risk type cyclomatic complexity metrics for each file in the baseline. Also, cyclomatic complexity count with risk type for all functions in the baseline.
DuplicatePairs.csv	Duplicate files found in the baseline, paired with original files
Duplicates-outfile_summary.csv	P-SLOC, L-SLOC, number of files summary per language type for duplicate source code files in the baseline. This file also includes a summary at the end with total number of files, total P-SLOC, and total L-SLOC counts of duplicate files in the baseline.
Duplicates-<LANG>_outfile.csv	Detailed line counts – total lines, blank lines, comments lines, compiler directives, data declarations, executable instructions, P-SLOC and L-SLOC per <LANG> type duplicate files in the baseline. This file also contains a total tally of all the above counts for all types of <LANG> type duplicate files in the baseline.
Duplicates-outfile_cplx.csv	Complexity metrics – number of math operations, logical operations, preprocessors, assignment operations, pointer operations, and level 1 through level 5 loops - summary for each duplicate file in the baseline
Duplicates-outfile_cyclomatic_cplx.csv	Total, average, and risk type cyclomatic complexity metrics for each duplicate file in the baseline. Also, cyclomatic complexity count with risk type for all functions of duplicate files in the baseline.
outfile_uncounted_files.csv	List of files that were not counted along with the reason (if known)
YYYY-MM-DD-HH:MM:SS-ucc.log	Log file listing errors that occurred while running UCC. Filename will include time stamp indicating the beginning of UCC-J run to differentiate it from other runs of UCC-J program.

Table 4 - Differencer Operation Outputs

File Name	Result Data Set
MatchedPairs.csv	A list of file pairs from baseline A and baseline B that were compared with each other for differencing metrics
outfile_diff_results.csv	Differencing metrics between baseline A and baseline B for files listed in the MatchedPairs.csv. Results include counts for new lines, deleted lines, modified lines, and unmodified lines when comparing a file pair. Results also include language type and file names. A summary of counts for total new lines, total deleted lines, total modified lines, and total unmodified lines is at the end of the file.
Baseline-A-outfile_summary.csv	P-SLOC, L-SLOC, number of files summary per language type for files in baseline A. This file also includes a summary at the end with total number of files, total P-SLOC, and total L-SLOC counts in baseline A.
Baseline-A-<LANG>_outfile.csv (for ex. Baseline-A-C_CPP_outfile.csv, Baseline-A-JAVA_outfile.csv, etc.)	Detailed line counts – total lines, blank lines, comments lines, compiler directives, data declarations, executable instructions, P-SLOC and L-SLOC per <LANG> type files in baseline A. This file also contains a total tally of all the above counts for all types of <LANG> type files in baseline A.
Baseline-A-outfile_cplx.csv	Complexity metrics – number of math operations, logical operations, preprocessors, assignment operations, pointer operations, and level 1 through level 5 loops - summary for each file in baseline A.
Baseline-A-outfile_cyclomatic_cplx.csv	Total, average, and risk type cyclomatic complexity metrics for each file in baseline A. Also, cyclomatic complexity count with risk type for all functions in baseline A.
Baseline-B-outfile_summary.csv	P-SLOC, L-SLOC, number of files summary per language type for files in baseline B. This file also includes a summary at the end with total number of files, total P-SLOC, and total L-SLOC counts in baseline B.

Baseline-B-<LANG>_outfile.csv (for ex. Baseline-B-C_CPP_outfile.csv, Baseline-B-JAVA_outfile.csv, etc.)	Detailed line counts – total lines, blank lines, comments lines, compiler directives, data declarations, executable instructions, P-SLOC and L-SLOC per <LANG> type files in baseline B. This file also contains a total tally of all the above counts for all types of <LANG> type files in baseline B.
Baseline-B-outfile_cplx.csv	Complexity metrics – number of math operations, logical operations, preprocessors, assignment operations, pointer operations, and level 1 through level 5 loops - summary for each file in baseline B.
Baseline-B-outfile_cyclomatic_cplx.csv	Total, average, and risk type cyclomatic complexity metrics for each file in baseline B. Also, cyclomatic complexity count with risk type for all functions in baseline B.
Duplicates-A-DuplicatePairs.csv	Duplicate files found in baseline A, paired with original files
Duplicates-A-outfile_summary.csv	P-SLOC, L-SLOC, number of files summary per language type for duplicate source code files in baseline A. This file also includes a summary at the end with total number of files, total P-SLOC, and total L-SLOC counts of duplicate files in baseline A.
Duplicates-A-<LANG>_outfile.csv	Detailed line counts – total lines, blank lines, comments lines, compiler directives, data declarations, executable instructions, P-SLOC and L-SLOC per <LANG> type duplicate files in baseline A. This file also contains a total tally of all the above counts for all types of <LANG> type duplicate files in baseline A.
Duplicates-A-outfile_cplx.csv	Complexity metrics – number of math operations, logical operations, preprocessors, assignment operations, pointer operations, and level 1 through level 5 loops - summary for each duplicate file in baseline A.
Duplicates-A-outfile_cyclomatic_cplx.csv	Total, average, and risk type cyclomatic complexity metrics for each duplicate file in baseline A. Also, cyclomatic complexity count with risk type for all functions of duplicate files in baseline A.
Duplicates-B-DuplicatePairs.csv	Duplicate files found in baseline B, paired with original files

Duplicates-B-outfile_summary.csv	P-SLOC, L-SLOC, number of files summary per language type for duplicate source code files in baseline B. This file also includes a summary at the end with total number of files, total P-SLOC, and total L-SLOC counts of duplicate files in baseline B.
Duplicates-B-<LANG>_outfile.csv	Detailed line counts – total lines, blank lines, comments lines, compiler directives, data declarations, executable instructions, P-SLOC and L-SLOC per <LANG> type duplicate files in baseline B. This file also contains a total tally of all the above counts for all types of <LANG> type duplicate files in baseline B.
Duplicates-B-outfile_cplx.csv	Complexity metrics – number of math operations, logical operations, preprocessors, assignment operations, pointer operations, and level 1 through level 5 loops - summary for each duplicate file in baseline B.
Duplicates-B-outfile_cyclomatic_cplx.csv	Total, average, and risk type cyclomatic complexity metrics for each duplicate file in baseline B. Also, cyclomatic complexity count with risk type for all functions of duplicate files in baseline B.
outfile_uncounted_files.csv	List of files that were not counted along with the reason (if known)
YYYY-MM-DD-HH:MM:SS-ucc.log	Log file listing errors that occurred while running UCC. Filename will include time stamp indicating the beginning of UCC-J run to differentiate it from other runs of UCC-J program.

7 Counting Standards

The UCC-J counts physical and logical SLOC and other metrics according to published counting standards which are developed at CSSE so that the logic behind the metrics being produced is clear to all participants. The counting standard documents are separate documents and are included in the UCC-J release. The counting standards are derived from the latest available ANSII standard language specification for each language counted by the UCC. Users should note that non-ANSII standard compilers may have commands which are outside of the ANSII standard specification. The results from using UCC-J on non-ANSII standard code cannot be guaranteed.

The counting standards documents for each language provide detailed information of what is counted, and how items are counted, so that all users can understand the operations and outputs of the UCC. Definitions are included of what is considered to be a blank line, comment line, and executable line of code for each language. The document describes in detail the physical and logical SLOC counting rules. Physical SLOC are counted at one per line. Logical SLOC counting rules are grouped by structure and the order of precedence is defined.

The items being measured, in order of precedence (numbered), are:

- 1. Executable lines,
- Non-executable lines
    - 2. Declaration (Data) lines
    - 3. Compiler directives
    - Comments
        - 4. On their own lines
        - 5. Embedded
        - 6. Banners
        - 7. Empty comments
        - 8. Blank lines

A table of logical SLOC counting rules is provided, and further specifies the order of precedence for the various types of executables lines. The rules define precisely when a count occurs, and a comments section gives further explanation.

The counting standards define for each language what keywords are counted and tallied in the output report. The keywords include compiler directives, data keywords, and executable keywords. Compiler directives are statements that tell the compiler how to compile a program but not what to compile. Data keywords define data declarations, which describe storage elements and the format that will be used to interpret data contained in them. Executable keywords are execution control statements. The specified compiler directives, data keywords, and executable keywords are counted and included in output reports. The data keywords are specific to each language.

8 Language Support

UCC-J currently supports below programming languages listed in Table 2.

8.1 File Extensions

UCC-J tool supports multiple programming languages. The tool maps a file to a programming language based on its file extension. Below table depicts the languages supported and their corresponding file extensions.

Table 2 - Language to File Extensions Mapping

Programming Language	File Extensions
ADA	.ada, .a, .adb, .ads
ASP	.asp, .aspx
ASSEMBLY	.asm, .s, .asm, .ppc
BASH	.sh, .ksh
C_CPP	.c, .cc, .cpp, .h, .hpp, .hh, .ipp
COLDFUSION	.cfm
COLDFUSION_SCRIPT	.cfs
CSHARP	.cs
CSHELL	.csh, .tcsh
CSS	.css
DOS_BATCH	.bat, .cmd, .btm
FORTRAN	.f90, .F90, .f95, .F95, .f03, .F03, .hpf, .f, .for, .ftn
HTML	.html, .htm
IDL	.pro, .sav
JAVA	.java
JAVASCRIPT	.js
JSP	.jsp
MAKEFILE	mak, .make, .gmk, .am, .in, .win, (if a file extension does not exist, file names containing the word “makefile” are processed)

MATLAB	.m
NEXTMIDAS	.mm
PASCAL	.pas, .p, .pp, .pa3, .pa4, .pa5, .pascal
PERL	.pl, .pm
PHP	.php
PYTHON	.py
RUBY	.rb, .rbw
SCALA	.scala, .sc
SQL	.sql, .SQL
VB	.vb
VB_SCRIPT	.vbs, .vbe, .wsf, .wsc
VERILOG	.v
VHDL	.vhd, .vhdl
XMIDAS	.txt
XML	.xml

It may be desirable to associate an additional extension to a language counter, or to disassociate a particular extension from a language counter. This can be done using the *-extfile <filename>* option on the command line.

8.2 Basic Assumption and Definitions

8.2.1 Data Files

Data files shall contain only blank lines and data lines. Data lines are counted using the physical SLOC definition.

8.2.2 Source Files

Source code files may contain blank lines, comment lines (whole or embedded), compiler directives, data lines, or executable lines. Source code files have to be compiled successfully to ensure the integrity of the inclusive syntax.

8.2.3 SLOC Definitions and Counting Rules

Please refer to the counting standard documents.

8.2.4 TAB

A Tab character is treated as a blank character upon input.

8.2.5 Blank Line

A blank line is defined as any physical line of the source file that contains only blank, Tab, or form feed characters prior to the occurrence of a carriage return (EOLN).

8.2.6 Total Sizing

The total sizing of analyzed source code files in terms of the SLOC count contains the highest degree of confidence. However, the sizing information pertaining to the sub classifications (compiler directives, data lines, executable lines) has a somewhat lower level of confidence associated with them.

Misclassifications of the sub classifications of SLOC may occur due to:

1) user modifications to the UCC-J tool,
2) syntax and semantic enhancements to the parsed programming language,
3) exotic usage of the parsed programming language, and
4) integrity of the host platform execution environment.

Additionally, in some programming languages a single SLOC may contain attributes of both a data declaration and an executable instruction simultaneously. These occurrences represent events beyond the control of the UCC-J tool designer and may cause the inclusive parsing capabilities of the tool to misclassify a particular SLOC. For these reasons, the counts of sub-classifications should be regarded as an approximation and not as a precise count. In only the physical SLOC definition does the sum of the sub-classification counts equal the total physical SLOC count.

8.2.7 Keyword Count

The search for any programming language specific keywords over a physical line of code for purposes of incrementing the tally of occurrences shall include the detection of

multiple keywords of the same type, e.g., two occurrences of the keyword READ on the same physical line.

The search for any programming language specific keywords over a physical line of code for purposes of incrementing the tally of occurrences shall include the detection on multiple keywords of different types, e.g., occurrences of keywords READ and WRITE on the same physical line.

Keywords found within comments (whole or embedded) or string literals shall not be included in the tally count.

9 Switch Usage Detail

This section will describe each switch in detail along with information on how to tailor the switch usage for various execution requirements. When applicable, performance implications will be described as well.

9.1 -v

Displays the version number of the UCC-J being executed.

9.2 -h [<switch_name>]

Displays help information on UCC-J usage with different switch options. Optionally, if –h is followed by a switch, information on how to use that specific switch is displayed.

9.3 -import <file>

Imports specified file containing language properties for a custom language. The custom language is included when code counter is run on a given directory or a file list.

9.4 -d

If specified, UCC-J will run the differencing function. Otherwise, the counting function is executed.

The UCC-J can be used to count SLOC within files, but it can also be used to difference two baselines of files. When just counting and not differencing is desired, use the UCC-J command without the -d switch. Counting is the default. When differencing, counting is performed and reported, and additional reports are produced which compare the files in two baselines and determine, for each file, how many lines of code were added, deleted, modified, or not modified. For counting and differencing, use the UCC-J -d command.

Differencing is a powerful capability that allows code changes between two baselines (or versions) of code to be monitored. Both physical and logical SLOC are included to provide insight into the extent of work completed between baselines.* (See **Note**, below)

The differencing process matches files between the two baselines using an algorithm that ensures the best possible match is found for all files. The matched file pairs are compared to each other line by line to determine how many SLOC have been added, deleted, modified, or are unmodified. All SLOC from any files in Baseline A that are not matched to a file in Baseline B are considered deleted, and all SLOC from any files in Baseline B that are not matched to a file in Baseline A are considered added.

The user is given the ability to tailor how the UCC-J determines if a SLOC has been modified using the -t # switch. For more information, see section 8.5.

The following relationships hold:

Baseline A SLOC = Deleted SLOC + Modified SLOC + Unmodified SLOC

Baseline B SLOC = New SLOC + Modified SLOC + Unmodified SLOC

Differencing will add a significant amount of time to the processing. It also will require more memory. If large numbers of files are in each baseline, there is a possibility that memory will become completely used and the process will hang. This problem may be addressed in a variety of ways.

The user may be able to run the process on a computer that has more memory than the computer that hung, or they may be able to add memory to the computer that hung.

The user may divide the input files into a number of smaller sets. If the user orders the smaller sets by language types, the duplicate file function will still work appropriately. If the files of a language type must be placed in separate sets, the duplicate file function may not find all duplicate pairs. If multiple sets are used, the user may need to aggregate the output files, as this will not be done automatically. The -outdir switch may be used to direct the outputs of sequential UCC-J executions to different directories, enabling the user to more easily specify multiple UCC-J runs. See section 9.10 for more information.

The user may choose to disable the duplicate file process by adding the -nodup switch to the UCC-J command line. The search for duplicate files will not be performed, resulting in a performance speedup.

*__Note__: In most cases, only logical SLOC is differenced with the –d option. For most languages, this removes exclude keywords and characters (e.g. else, end if, }, {, etc.) which gives a more accurate differencing result. An example of LSLOC differencing for a C++ file would be as follows: we are given the input code

```
#include <iostream>
using namespace std;

int main ()
{
  // for loop execution
  for( int a = 10; a < 20; a = a + 1 )
  {
     cout << "value of a: " << a << endl;
  }

  return 0;
}
```

However, the code that is run through the differencer would be

```
#include <iostream>
using namespace std;
int main ()
  for( int a = 10; a < 20; a = a + 1 )
     cout << "value of a: " << a << endl;
  return 0;
```

This cuts down on the false-positives of code modification (add, modified, deleted results) as well as the total code we difference, thereby improving performance.

There are three exceptions where physical SLOC needs to be differenced: XML, HTML, and ColdFusion. These languages count LSLOC by the number of open tags (e.g. <p>, <table>, <code>, <cffunction>, etc.), however the level of effort is determined by what falls within those tags, i.e. the PSLOC. Therefore, no reformatting of the code is performed before differencing files of these three language types.

9.5 -i1 fileListA.txt

Input list filename containing filenames in *Baseline* A.

The fileListA.txt file is required to be a plain text format file containing a list of filenames to be processed, one per line. The files may be specified as full or relative directory/pathname specification. The UCC-J will open the file specified by the fileListA.txt argument, read each line and attempt to open each file specified. If a file cannot be opened, an error message is generated, and a tally is kept and reported in the output report.

When counting, as specified by the lack of -d switch on the command line, the UCC-J will expect just the -i1 switch and not the -i2 switch. When differencing, as specified by the inclusion of the -d switch on the command line, the UCC-J will expect to find the -i2 switch. If these conditions are not met, an error is generated.

9.6 -i2 fileListB.txt

Input list filename containing filenames in *Baseline* B.

If the -d switch is present, the differencing function will be invoked and fileListA.txt and fileListB.txt will be compared. Normally, *Baseline* B is the newer or the current version of the program, as compared to *Baseline* A. The file format is same as *Baseline* A.

9.7 -t

Specifies the modification threshold.

The user is given the ability to tailor how the UCC-J determines if a SLOC has been modified using the –t # switch. The # is the modification threshold and can be any number between 0 and 100, with the default being 60 (same as -t 60). The modification threshold specifies a percentage; i.e. -t 60 indicates a modification threshold of 60%. If two SLOC have the percentage of common characters equal or higher than the specified threshold, as compared over the length of the longest line, they are matched and counted as modified. Otherwise, they are counted as one SLOC deleted and one SLOC added. In this example, using -t 60, if 60% of the characters of the longest line match with the compared line, the lines are considered modified in Baseline B. If less than 60% of the characters in the longest line match with the compared line, Baseline A counts one SLOC deleted, and Baseline B counts one SLOC added. If the compared lines are exactly the same, the lines are counted as unmodified.

9.8 -tdup

Specifies the threshold percentage for identical SLOC when comparing two files within a baseline for one file to be considered a duplicate of the other.

This switch specifies the maximum percent match allowed between the SLOC in two files of the same name within a baseline to be considered duplicates. Blank lines and comments are not considered. By default, this threshold is 60. Valid values are numbers between 1 and 100. The threshold corresponds to the percentage of SLOC which may be the same when two files are compared in order to be considered duplicates. For example, -tdup 20 would mean that a file is a duplicate of another file if 20% or more of the SLOC are same.

Duplicate file processing is computationally expensive. A switch -nodup is available to inhibit the duplicate processing in order to reduce execution time.

Files must be in the same baseline, but not necessarily in the same directory, in order to be duplicates. The files do not need to have the same name; however, if the file names are different, they have to be exactly the same, including comments and blank lines, to be identified as duplicates. Files with the same name, but not in the same subdirectory, are

considered duplicates if the code is identical, even if there are differences in the comments and/or blank lines.

Duplicate files are counted and reported separately. One purpose for this command is to isolate SLOC counts for files which did not require development, but were duplicated for a variety of reasons which could include for configuration management purposes, or were computer generated.

9.9 -dir <dirA> [<dirB>] [filespecs …]

Specify directories containing files to be counted and/or compared.

The -dir switch causes the UCC-J to look for files to be counted in the specified directory and its subdirectories. If extensions are provided, it will select only files with those extensions. Any default or specified input file lists will be ignored.

If the -d (for differencing) is not present, the UCC-J will only look for <dirA> and filespecs. If the -d is present, the UCC-J will look for dirA, dirB and filespecs. The directories are searched recursively for files that match the filespecs.

dirA Name of the top level directory. If the option *-d* is provided, dirA is the directory of *Baseline* A. Otherwise, it specifies the directory to be counted with the counting function. The directory search is recursive so that all subdirectories under the top level directory are searched.

dirB Name of the top level directory of *Baseline* B, used only if the option *-d* is given. The directory search is recursive so that all subdirectories under the top level directory are searched.

filespecs Specifications of file extensions to be counted/compared; wildcard chars ? * are allowed. Use a space between *filespecs*; for example, *.cpp *.c

Examples:

java –jar <jar file name> dirA *.*

This command will do counting only, since there is no -d, and will look in the directory dirA recursively to find all files. Any files with extensions recognized by the UCC-J will be counted

java –jar <jar file name> dirA

This command is equivalent to UCC-J dirA *.*. It will do counting only, and will look in the directory dirA recursively to find all files. Any files with extensions recognized by the UCC-J will be counted.

java –jar <jar file name> dirA *.c *.cpp *.f *.for *.f77 *.f90 *.f03 *.hpf

This command will do counting only, since there is no –d, and will look in the directory dirA recursively, and will process all files found with the extensions specified (.c, .cpp, .f, .for, .f77, .f90, .f03, *.hpf). Notice that the file extensions are separated by a space but have no comma between.

java –jar <jar file name> -d dirA *.*

This command is improperly formed, as it specifies -d for differencing, but two directories must be provided.

java –jar <jar file name> -d dirA dirB *.*

This command will difference the files found in dirA with the files found in dirB. The *.* direct the UCC-J to process all files with an extension recognized by the UCC-J.

java –jar <jar file name> -d dirA dirB *.java *.sql

This command will difference files with the extension .java or .sql found in directories dirA and dirB recursively. No other files will be processed.

java –jar <jar file name> -d dirA dirB

This command will difference files found in directories dirA and dirB recursively with any extension that UCC-J recognizes.

9.10 -outdir <dirname>

Prepend the dirname to the output files created.

This command allows the user to specify a directory name to be prepended to the output reports. If the directory does not exist, UCC-J will create it. The reports will be generated in the specified directory. This allows the user to set up a batch job with multiple UCC-J runs and not have the output reports overwritten.

Examples:

java –jar <jar file name> -outdir test1

This command will look for the default input list fileListA.txt, count the SLOC within the files, and write the output reports into the directory test1.

java –jar <jar file name> -outdir test2 -i1 test2files

This command will open the input list of files in test2files, read them in, count the SLOC, and write the output reports into directory test2.

9.11 -extfile <extfilename>

The UCC-J uses a file's extension to associates files to be counted with language counting modules. One or more extensions may be associated with a given language counter. The table in section 0 lists the current languages and their associated file extensions.

The -extfile switch allows users to modify the file extension mapping by specify which file extensions are to be associated with the language counters. This gives the user more flexibility in determining which files will be counted. It also will be useful as more languages are added to the UCC-J, as some extensions may be used by more than one language.

Data files will be only be counted for physical SLOC, and only if the users uses an -extfile command to specify the data file extensions. Refer to section 9.11.1 for more information. The file named by <extfile> is a list that associates user-specified extensions with UCC-J language counters.

The extfile command allows a user to create and specify the name of a file that maps file extensions to UCC-J language counters. This allows the user to include non-default extensions, or remove default extensions. There are no spaces between the language, the equal sign, or the extensions. The extensions are separated by a comma. Comments may be placed within square brackets, for example [comment], and may be placed anywhere in the extfile.

Each line in the file should have the form:

<language>=.<ext1>[,.<ext2>,…<extn>]

The -extfile option gives the user great flexibility in tailoring specific runs. For instance, suppose a user wishes to count only the files written in Fortran 77 as indicated by having the extension .f77. Placing the Fortran=.f77 command into the extfile would accomplish that. If a user wishes to count Fortran files generated with a non-standard extension, such as .foo. The user can place the command Fortran=.foo in the extfile.

Examples:

Java=.java,.javax

This line would cause any file with the extension .java or .javax to be counted with the java language counter. No other extensions will be counted with the java language counter. All other languages will use the default extensions.

Ada=.ada,.a

Fortran=.f,.for

These lines would cause the Ada counter to count only files with extensions of .ada or .a. All other Ada files with other extensions will be ignored. Also, the Fortran counter will count only files with extensions of .f or .for. All Fortran files with other extensions will be ignored.

9.11.1 File Extension Mapping Names

For reference, the current file extension mappings are shown in Table 2. Be sure to use the Internal UCC-J Language Name in the -extfile mapping as defined in Table . For example, don't use C/C++, use C_CPP.

9.12 -unified

Directs the UCC-J to print counting results to a single unified language file name TOTAL_outfile.csv.

If the -unified command is not specified, a report is produced for each language and is named <Lang>_outfile.csv, which is an Excel format.

9.13 -debug <level>

Enables the UCC-J debugger. Outputs are directed to the <time_stamp>_ucc.log file which will appear in the same folder as the UCC-J executable JAR by default if an output directory is not specified.

Valid levels are FATAL, ERROR, WARN, INFO, DEBUG, or TRACE.

By default, the application is set to the ERROR level.

All levels under the specified level will be logged. For example, if the level is set to INFO, any log outputs with INFO, WARN, ERROR, or FATAL level would be outputted and log statements with DEBUG or TRACE would be ignored.

9.14 -nocomplex

Do not process or report keywords or complexity metrics.

Using this switch will reduce the processing time.

9.15 -export <language_name>

Exports language properties for the specified language to a text file. The output text file is stored in the directory specified with –outdir option or defaults to current working directory. Output file name will be [language_name]LanguageProperties.txt.

language_name used with this switch must match the names defined in Table .

9.16 -nodup

Do not search for duplicate files.

The duplicate file processing is both memory and processor intensive, as the process analyzes each file and compares to potential duplicates line by line. If the user does not need to separate out duplicate files within a baseline, the -nodup switch will disable the duplicate search decreasing processing time significantly. Any duplicates will be reported as unique files.

9.17 -ascii

Generate reports in text format (.txt). The default report format is .csv, which directly opens into MS Excel.

9.18 -cf

Support handling ClearCase filenames.

The ClearCase application appends version information to the filename, starting from '@@'. This option requires the UCC-J tool to handle the original filename instead of the ClearCase-modified filename by stripping off the '@@' and any characters after that and before the extension.

9.19 -nolinks

Do not follow symbolic links on Unix based systems.

This disables following symbolic links to directories and counting of links to files. This can prevent duplicate file counts on Unix/Linux systems.

9.20 -trunc

Truncation threshold.

The truncation threshold specifies the maximum number of characters allowed in a logical SLOC. Additional characters will be truncated. The default value is 10,000. If -trunc 0 is specified, no truncation is done. Performance can be significantly degraded if truncation is too high.

10 Log4j Usage

UCC-J uses Log4j 2 to perform application logging:

- User Guide: http://logging.apache.org/log4j/2.x/log4j-users-guide.pdf
- Website: http://logging.apache.org/log4j/2.x/

UCC-J is configured to use a Programmatic Log4j2 configuration, bypassing the need for an external configuration file. This is done to allow the default logging level to be changed via a switch parameter from the UCC-J command line. The configuration can be found in the Log4j2ConfigurationFactory class and is called (along with the logging level) from the MainUCC class.

UCC-J Log4j2 Setup:

a) From Eclipse, add the files to your project classpath (this is also covered at the end of Section 7 above). First, select Project -> Properties from the global menu

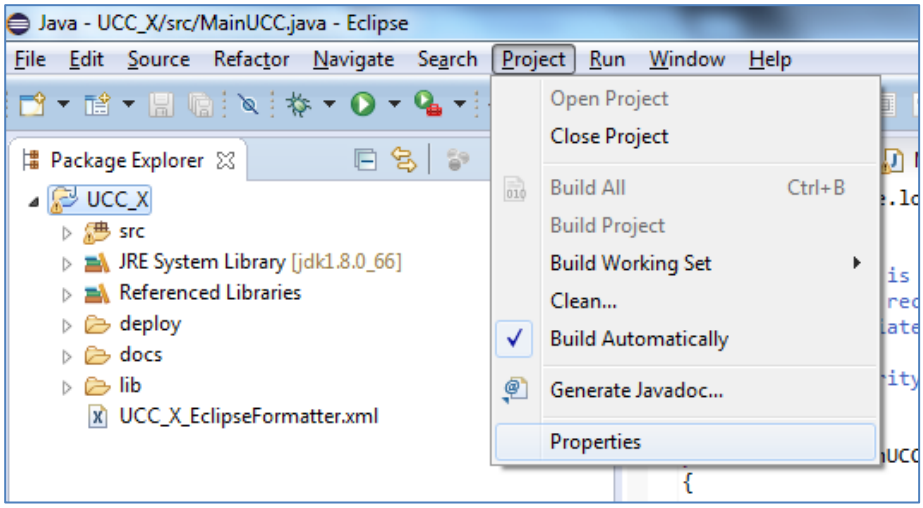

Figure 10 – Properties option under Project menu in Eclipse

b) From the left-side menu, select “Java Build Path” then from the right-side pane, select the “Libraries” tab.

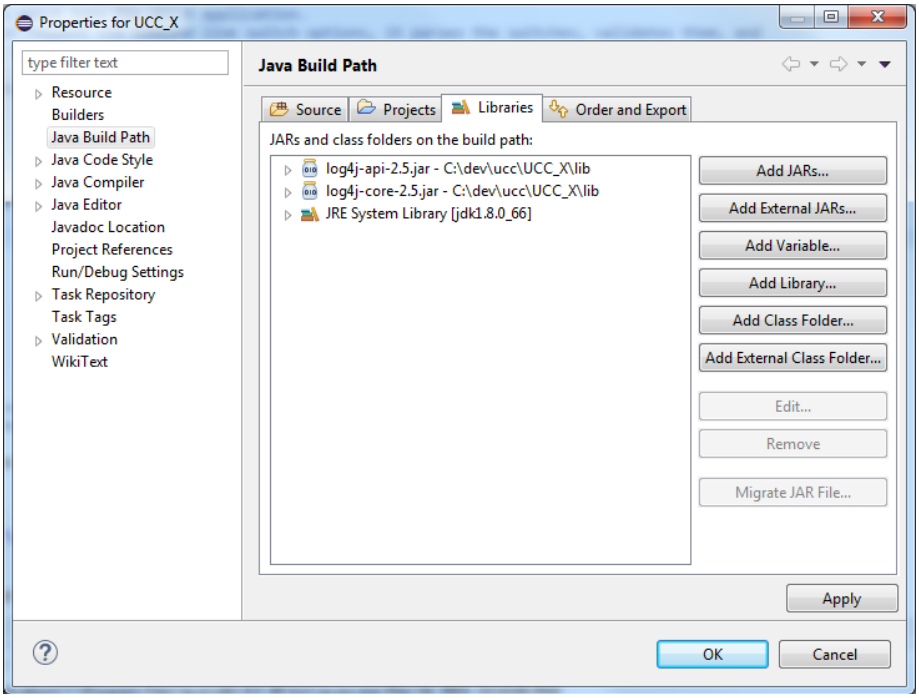

Figure 11 – Java build path Libraries tab under project properties in Eclipse

c) Next, click the “Add External JARs…” button on the right hand side (2nd button from top), and locate the JAR files identified in Step 1 (located in your UCC-J package in the lib folder). Finally, click “Open” and your project classpath is now updated.

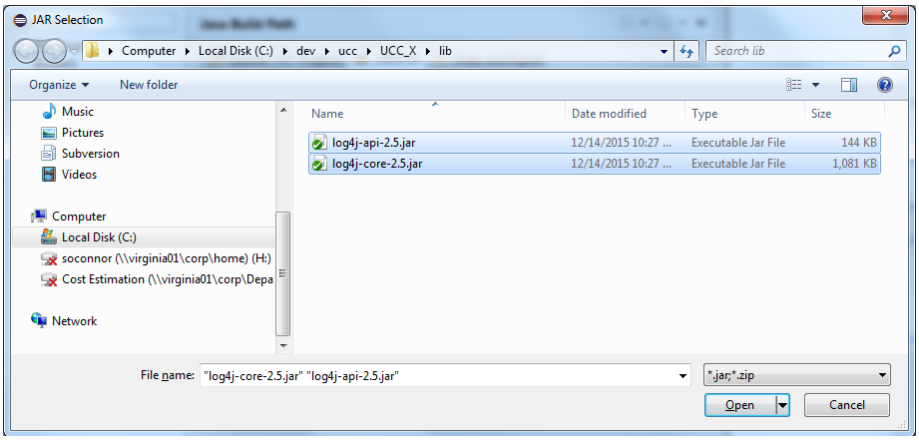

Figure 12 – Log4j jar location

To use log4j2 to perform logging outputs within a Java class file, it requires the following:

a) Import the following libraries at the top of your Java class file:

```
import org.apache.logging.log4j.Logger;
import org.apache.logging.log4j.LogManager;
```

b) Instantiate the "Logger" class after the class signature (pass in the file's class name to the method constructor, in this example MainUCC is the class name and must match the entry to the getLogger constructor):

```
public class MainUCC
{
  private static final Logger logger = LogManager.getLogger(MainUCC.class);
```

c) Call the logger wherever a notification is required, and pass the log level. For example:

```
logger.info("UCC-X Logging Enabled: User set to Level.FATAL");
```

The following are all of the default log4j output levels (with corresponding integer rankings). The application can be configured to a particular level, where it outputs all lower rankings. For example, if the application was set to the "INFO" level, any log outputs with INFO, WARN, ERROR or FATAL ratings would be outputted (any logger statements using DEBUG or TRACE would be ignored).

i. FATAL 100
ii. ERROR 200
iii. WARN 300
iv. INFO 400
v. DEBUG 500
vi. TRACE 600

d) The application's overall log4j2 ranking level can be set within the MainUCC class file's ParseUserRequest method.

To enable log4j2 when executing UCC, use the "-debug" command line argument. The "-debug" argument can appear at any point in the list of arguments provided to the program. UCC will always search for and enable logging first prior to processing other arguments. For convenience, the –debug argument can appear in any order from the command line when combined with other arguments.

To view UCC's outputted log4j file from Eclipse, use the Eclipse LogViewer Plugin: http://marketplace.eclipse.org/content/logviewer

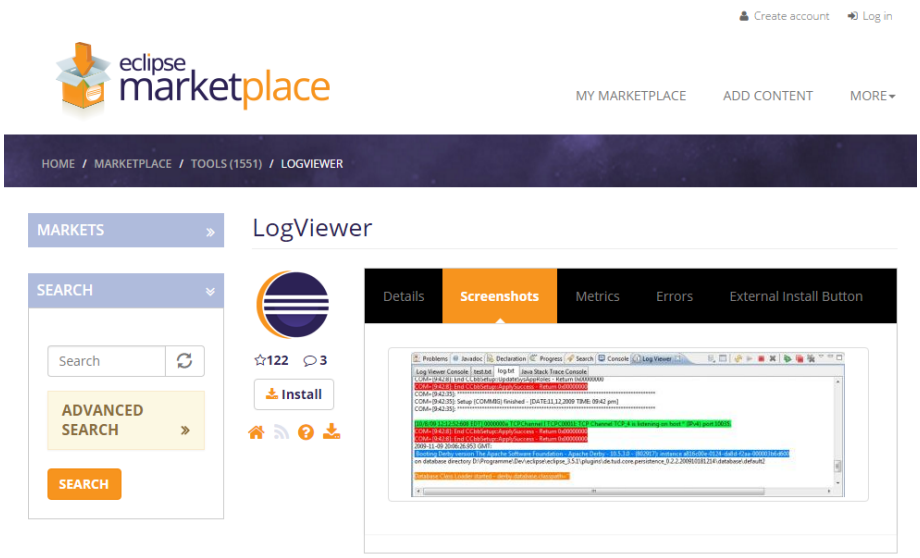

Figure 13 – LogViewer Plugin

a) From the link above, hover over the "Install" link, left-click and drag it to your Ecipse IDE to perform the plugin installation.

b) Once installed, you can launch the LogViewer from Eclipse's Console Menu. Click the down arrow to the right of the new window icon and select "4 Log Viewer Console":

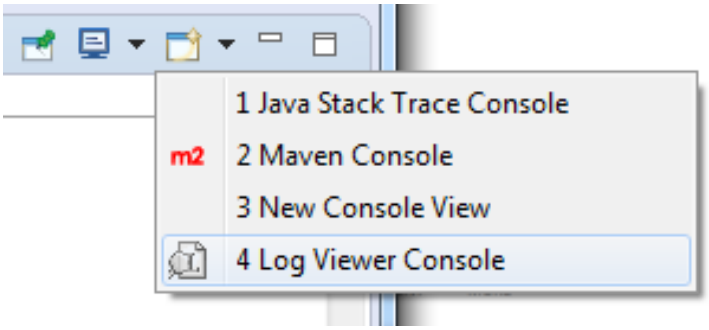

Figure 14 – Log viewer console option in Eclipse

c) Select the newly created "Log Viewer" tab in the console section, then drag a file you wish to monitor from Windows Explorer to the LogViewer console window in Eclipse. When running UCC from Eclipse, the "ucc.log" file will be created under the UCC_X project root folder "…\ucc\UCC_X". The LogViewer console will tail an external file, and refresh changes to that file automatically. This will permit real time feedback from the UCC log4j system while using the Eclipse IDE.

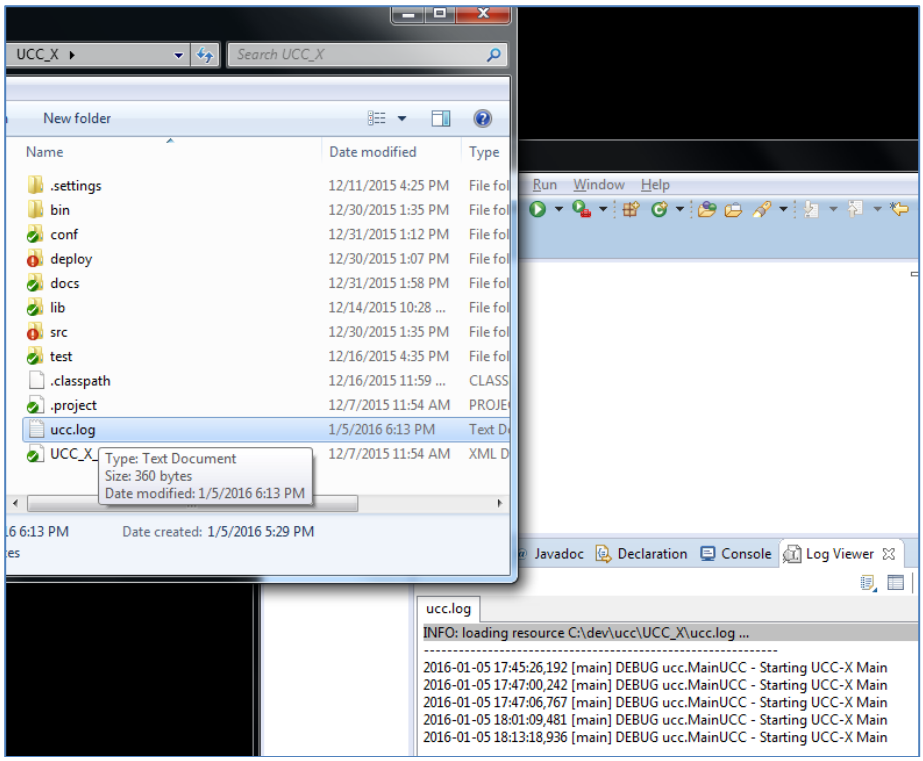

Figure 15 – Log Viewer and ucc.log location

11 References

[1] R.E. Park, "Software Size Measurement: A Framework for Counting Source Statements", Technical Report CMU/SEI-92-TR-20 ESC-TR-92-020, 1992.

[2] B. Boehm, C. Abts, S. Chulani, "Software development cost estimation approaches: A survey", *Annals of Software Engineering, 2000*.

12 Appendices

12.1 Appendix A: Acronyms List

Acronyms	Definitions
ASCII	American Standard Code for Information Interchange
CMU	Carnegie Mellon University
CSS	Cascading Style Sheet
CSV	Comma Separated Values

HTML	HyperText Markup Language
IDL	Interactive Data Language
JAR	Java Archive
JRE	Java Runtime Environment
JSP	Java Server Pages
JVM	Java Virtual Machine
LSLOC	Logical Source Line Of Code
OS	Operating System
PHP	PHP: Hypertext Preprocessor
RAM	Random Access Memory
SE	Standard Edition
SEI	Software Engineering Institute
SLOC	Source Line of Code
SQL	Structured Query Language
UCC	Unified Code Count
UCC-J	Unified Code Count – Java version
USG	United States Government
VB	Visual Basic
VHDL	VHSIC Hardware Description Language
XML	Extensible Markup Language