AEROSPACE REPORT NO.
TOR-2010(3906)-72

Unified Code Counter (UCC) Software Design

July 10, 2012

Ryan E. Pfeiffer
Systems Analysis and Tools
Engineering Applications Department

Prepared for:

U.S. Government

Contract No. FA8802-09-C-0001

Authorized by: National Systems Group

AEROSPACE
Assuring Space Mission Success

AEROSPACE REPORT NO.
TOR-2010(3906)-72

Unified Code Counter (UCC) Software Design

July 10, 2012

Ryan E. Pfeiffer
Systems Analysis and Tools
Engineering Applications Department

Prepared for:

U.S. Government

Contract No. FA8802-09-C-0001

Authorized by: National Systems Group

AEROSPACE
Assuring Space Mission Success

AEROSPACE REPORT NO.
TOR-2010(3906)-72

Unified Code Counter (UCC) Software Design

Approved by:

[illegible]

Gretchen Lindsay, Systems Director
Policy Plans and Resource Analysis
Mission Innovations
Mission Enabling Operations Division
National Systems Group

SL0024(1, 5824, 33, GN)

Acknowledgments

I would like to thank all those who have been involved in the UCC project. Thanks to Dennis Hamme and Mark Kirtley of The Aerospace Corporation for their program support. Thanks to all of the Aerospace technical staff and USC instructors who have been instrumental in the project development including Marilyn Sperka, Mike Lee, Harley Green, and Mathy Pandian of Aerospace, and Vu Nguyen and Anandi Hira of USC. I would also like to thank Jill Dunn, Michal Bohn, and Betsy Legg of the U.S. Government for their direction and guidance. A special thanks to Dr. Barry Boehm of USC and Marilee Wheaton of Aerospace for their leadership in founding this project.

Abstract

The Unified Code Counter (UCC) developed and maintained by the University of Southern California (USC) Center for Systems and Software Engineering (CSSE) has been actively supported by The Aerospace Corporation (Aerospace) under the auspices of a U.S. Government cost analysis group. This software provides a method of automated software size estimation by counting source lines of code and comparing counts between code baselines. This work has benefited the U.S. Government by standardizing software line-counting methodology and facilitating the analysis of contractor performance on software projects. This technical operating report (TOR) is intended to formally document the development, requirements, and design of the UCC software and to outline how to add counting and differencing functions for a new language.

Contents

Acknowledgments iii

Abstract v

1. Introduction 1
   - 1.1 Purpose 1
   - 1.2 Project Sponsorship 1

2. Project Organization 3

3. UCC Software Development Process 5
   - 3.1 Project Teams 5
   - 3.2 Configuration Management 5
   - 3.3 Bug Tracking 5
   - 3.4 Project Documentation 5
   - 3.5 Test Requirements 6
   - 3.6 Release Process 6

4. Software Requirements 7
   - 4.1 Code Counters 7
   - 4.2 Differencing 7
   - 4.3 Reports 8

5. Software Design 9
   - 5.1 MainObject Class 10
     - 5.1.1 Parse Command-Line Parameters 10
     - 5.1.2 Process File List 10
     - 5.1.3 Read Files into Memory 10
     - 5.1.4 Call Appropriate Counter(s) and Process Embedded Languages 10
     - 5.1.5 Determine Duplicate Files 11
     - 5.1.6 Print Counting Results 11
   - 5.2 DiffTool Class 11
     - 5.2.1 Parse Command-Line Parameters 12
     - 5.2.2 Determine File Matches Across Baselines 12
     - 5.2.3 Perform Differencing Comparison Between Baselines 13
     - 5.2.4 Print Differencing Results 13
   - 5.3 CmpMngr Class 13
   - 5.4 CCodeCounter Class 13
     - 5.4.1 Perform Precounting Processing 14
     - 5.4.2 Count Blank Lines 14
     - 5.4.3 Count Whole Line and/or Embedded Comments 14
     - 5.4.4 Count Keywords 14
     - 5.4.5 Perform Language-Specific Processing 14

6. Adding a New Language Counter 15
   - 6.1 Counting Standards Document 15
   - 6.2 Subclass CCodeCounter 15
   - 6.3 Update CWebCounter (if needed) 16
   - 6.4 Add Hooks to New Language Counter 16

7. Summary 19
8. References 21

Figures

Figure 1. Simplified UCC class diagram. 9

1. Introduction

1.1 Purpose

The purpose of this technical operating report (TOR) is to document the development and design of the Unified Code Counter (UCC) software produced and maintained by the University of Southern California (USC) Center for Systems and Software Engineering (CSSE) [1]. The Aerospace Corporation provides project direction, oversight, and validation and verification support. This project is sponsored by a U.S. Government cost analysis and improvement group.

The purpose of the UCC software is to count source lines of code (SLOC) according to a defined set of standards for a variety of programming languages. UCC provides a generic code-counting tool that is designed to provide consistency and impartiality. It also provides counts of new, modified, deleted, and unmodified lines of code between two baselines. UCC replaces the original CodeCount toolset by combining the code counting and differencing functionality of the previous tools.

Typically, software code counts are based on physical source lines of code (PSLOC) or logical source lines of code (LSLOC). A PSLOC corresponds to a single line terminated by a carriage return and/or line feed, typically excluding blank lines and comments. This measure is language independent and simple to count, but it is very sensitive to formatting and coding styles, such as the placement of multiple statements on a single line or the spanning of statements across multiple lines. A more accurate determination of SLOC is an LSLOC. An LSLOC corresponds to a single statement of code logic. This is language dependent, but it is not affected by formatting or coding styles. Both of these SLOC types are included in the UCC software output results. The LSLOC definitions are designed to be compatible with the Software Engineering Institute's (SEI) code-counting framework [2].

Lines of code are often used as input to software cost estimation and analysis tools. Historical cost and schedule data are correlated to actual code counts and are used to predict future software project costs and schedules based upon estimated code counts. Thus, consistency of line counting is critical to producing meaningful metrics for accurate software cost estimates and software development productivity metrics.

1.2 Project Sponsorship

Many inexpensive or free code counters exist in the public domain, each with their own specific counting methodology. The U.S. Government needed a way to evaluate various contractor proposals using a consistent methodology. Software projects are often evaluated using SLOC counts for various cost, schedule, and quality metrics. The U.S. Government decided to sponsor a nonbiased, consistent code counter using a university setting as an impartial development environment.

2. Project Organization

The UCC project is developed and maintained through the USC-CSSE, and is led by faculty at USC. The Aerospace Corporation has partnered with USC in order to provide direction and oversight and to ensure the implementation of software engineering best practices. Aerospace provides validation and verification as nonbiased government partners. Aerospace involvement also provides an avenue for input from a U.S. Government cost analysis group to ensure that the UCC satisfies the cost estimation needs of Aerospace's government customers.

The USC-CSSE provides research and development in software design and development processes. It was founded in 1993 by Dr. Barry Boehm, who is world renowned for his contributions in software engineering processes. The UCC project is currently administered through a directed research course (CS590) run by doctoral candidates. USC-CSSE also operates the General Affiliates Program [3] that provides access to beta releases for testing/feedback purposes to affiliate member organizations.

Graduate students of the USC Department of Computer Science act as the primary developers and testers of the UCC software. Participation by students is considered a professional internship. Students commit to at least five hours per week for each registered credit, which are graded credit/noncredit. Each student is also expected to present a briefing on a software engineering research topic. Students are required to document time spent on UCC activities on a weekly basis. Remote students operating through the USC Distance Education Network (DEN) are also involved and typically work on individual assignments such as the development of code-counting standards and documentation.

3. UCC Software Development Process

The UCC project is governed by a well-defined software development process. Project tasks and timelines are defined jointly by the USC instructors and Aerospace personnel. This cooperative environment allows Aerospace to play a critical role in the direction and development of the UCC software. It also ensures that common software engineering processes are followed in order to produce a consistent quality product.

3.1 Project Teams

Each semester, the students are divided into project teams based on their previous experience, software expertise, and time commitment (course units). Within each project team, individuals are assigned as the team lead, developers, or testers. Each team is responsible for their internal management, including regular team meetings, individual tasks, and project milestone schedules. The teams are required to report status informally on a regular basis and formally twice during the semester. All teamwork products and activities are recorded in a common repository throughout the term.

3.2 Configuration Management

Configuration management is a critical tool employed in any software engineering process. It is used to manage and control changes in software development. The primary benefits of configuration management in software development are the ability to identify and control software configurations, track changes, record history, and revert to previous versions. In the UCC project, separate tools were chosen for source code control and management of documentation and student submissions.

Subversion® [4] is used as the software configuration management tool. Being an open source application that is widely used and supported in industry, Subversion was a logical and effective choice. In practice, the instructors typically review the work of each semester's students and integrate changes into the managed version of the code within Subversion. Aerospace also keeps a duplicate Subversion repository for internal use.

3.3 Bug Tracking

Bug tracking tools generally provide more than just problem—a.k.a. "bug"—reporting. They also provide a traceable forum for feature requests, progress tracking, and task assignment. However, due to the regular flow of new students and the typically unchanging initial group task assignments, the UCC model only uses its bug tracking tool to report known bugs for future teams to resolve.

Bugzilla [5] is a commonly used open-source Web-based bug tracking system that is used by the UCC project. This tool allows students to report and describe problems discovered while developing and/or testing UCC code. These bug reports are later used by the instructors to assign tasks to future student teams.

3.4 Project Documentation

A Wikipedia® [6] page was set up for the UCC as a structured software environment to record and present project information to the software community. This wiki is periodically updated by student teams at USC.

3.5 Test Requirements

A common set of general test requirements has been developed and provided by USC and Aerospace in a formal test plan [7]. These requirements cover the entirety of the UCC code and apply to any of the implemented languages. Student teams develop test plans for each language. These language-specific test plans contain each test detailed by its description, expected outcome, and results. These are critical in order to validate the software and preserve the scope of testing for future semester project teams.

The scope of testing includes basic UCC functions, code counters for individual languages, software version differencing for individual languages, results reporting, and graphical user interfaces (GUI) when applicable. The test requirements also incorporate a list of general program requirements specified by the U.S. Government.

3.6 Release Process

Software releases are controlled through the USC-CSSE General Affiliates Program. An affiliate website [3] provides controlled access to prerelease software. At this stage, affiliate members test software and denote any bugs, offer suggestions, and sometimes provide recommended code modifications. Once the software is determined to be stable, it is moved to a public website [8], where the software is released for public use under the terms of the USC-CSSE limited public license.

Each release of the UCC software typically includes the source code, license agreement, readme file, release notes, and a user's document. The user is only required to have a compatible compiler (see next section).

4. Software Requirements

The specific requirements for the UCC design are documented within the test plan referred to in the previous section [7]. The plan details system-level requirements, code-counting requirements, differencing requirements, and performance requirements. A small subset of basic requirements has been defined by the U.S. Government in order to meet its specific needs. The remaining requirements are derived from experience and customer feedback.

The UCC software is required to be command-line driven with optional parameters to control the handling of inputs and code differencing. The software code also must be able to be compiled on any platform that has a current ANSI standard C/C++ compiler [9], such as those provided through the GNU Compiler Collection [10] (GCC).

4.1 Code Counters

Each language counter must have a “counting standards document” to define the SLOC counting rules based on the language-specific syntax. The information contained in this document is typically based on open-standards language definition documents available on the Internet. The actual counting rules are designed to conform to the UCC “C/C++ CodeCount Counting Standard” [11] document, which is compatible with the SEI code-counting framework [2].

UCC must process files either from a previously generated input file list or a directory including wildcard file specifications. A file should be skipped (and the user notified) if it does not have a supporting code counter or if a physical line in the file exceeds the maximum length allowed (~10 million characters). In addition, duplicate files are to be detected within a baseline and reported independently in order to provide a more accurate picture of actual work completed. This prevents common files that are duplicated within a program from being counted multiple times.

Each code counter is required to tally the PSLOC and LSLOC, including subcounts for blank lines, comments, compiler directives, data declarations, and executable instructions. In addition, the code counters are to include counts of the occurrences of language keywords, mathematical functions, logical statements, and other statistics.

4.2 Differencing

UCC is required to compare the differences between two code baselines and provide counts of added, deleted, modified, and unmodified lines of code. This is a powerful capability that allows code changes to be monitored. Both physical and logical SLOC are included to provide insight into the extent of work completed between baselines.

The differencing feature is required to provide a command line option for specifying the threshold for percent change in a line in order to determine whether a line has been changed or the line was deleted and a new line added. In addition, each baseline must be separately processed for duplicate files, and their corresponding differencing counts must be reported independently.

4.3 Reports

The UCC reporting system is required to be common for all languages to ensure consistency. It provides several useful reports as follows:

- SLOC Counts – PSLOC and LSLOC totals including blank lines, comments, compiler directives, data declarations, and executable instructions. Also includes counts of the occurrences of language keywords.
- LSLOC Differences – Summary of added, deleted, modified, and unmodified LSLOC.
- Duplicate Files – Lists matching file pairs and duplicate files.
- Keyword Counts – Totals of keyword occurrences such as mathematical functions, logical statements, assignments, etc.

5. Software Design

UCC is developed in ANSI standard C/C++ [9], so it executes on virtually any platform that has a valid C/C++ compiler. The source code is provided free under a limited public license and requires only a name, email address, and phone number to gain access to download the software. Users are encouraged to modify the code to meet their needs or distribute the code as needed. However, if a user distributes a modified version, the user is required to provide a copy to USC according to the license terms.

UCC is written with maintainability and reusability in mind. The file handling, language selection, code differencing, and reporting are handled in a language-independent manner using common C++ classes. Each language handler consists of a class derived from the base code counter class and consists of a set of language-specific members and methods. When a new language counter is added, only a language-specific derived class is required.

The differencing engine is designed to be common between languages. Each language-specific counter only needs to determine where the LSLOC exist. Then each LSLOC is compared between baselines. This functionality increases the complexity of the counters since the end point of each LSLOC must be determined. However, this unique capability adds significant value to the UCC software.

The UCC software is designed around a small set of primary classes. The principal controlling class is the MainObject. This controls the counting and printing functions and serves as a parent class for the DiffTool class used for baseline differencing. The CCodeCounter class contains the general members and methods for counting code. Each programming language is then derived from the CCodeCounter to handle language-specific attributes and processing.

This simplified class diagram illustrates the basic code structure. The details of each primary class are described in the following sections.

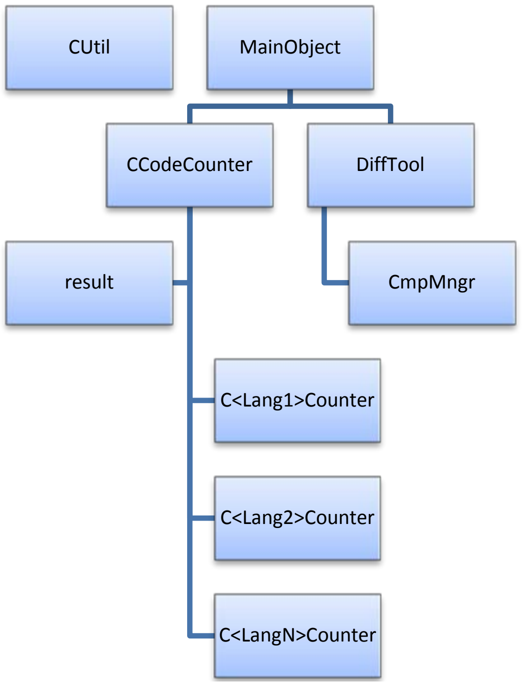

Figure 1. Simplified UCC class diagram.

5.1 MainObject Class

The MainObject class performs the following functions:

- Parse command-line parameters.
- Process file list.
- Read files into memory.
- Call appropriate counter(s) and process embedded languages.
- Determine duplicate files.
- Print counting results.

The MainObject class contains a complete set of the language-specific counter objects to be called as each file type is identified by its file extension. The following is a detailed description of the processing performed by the MainObject.

5.1.1 Parse Command-Line Parameters

The command-line parameters are initially read by the main program function. Command-line parsing is performed in order to determine which optional parameters have been specified. If the [-d] parameter is specified, a DiffTool object will be instantiated in place of the MainObject. Most of the parameters pertain to user preferences such as thresholds and file formats. One important parameter [-dir] is used to specify a directory and file specification to gather a list of files to count. If this parameter is absent, the MainObject will search for a standard file name containing a list of files to count (fileList.txt, fileListA.txt, or fileListB.txt).

5.1.2 Process File List

If the user has provided a directory and a file specification through the [-dir] command-line parameter, the directory will be searched for files that meet the regular expression criteria, such as *.cpp *.h for C++ or *.* for all files. The user also has the option of providing a file containing a list of files to count. In this case, the file list processing is not required.

5.1.3 Read Files into Memory

Each processed or listed file is examined by the list of language-specific code counters (classes derived from the CCodeCounter class). If the file extension is supported by one of the code counters, the file lines are read into a file map for later processing. A file will be skipped if it does not have a supporting code counter or if a physical line in the file exceeds the maximum length allowed (~10 million characters). The user will be notified of any skipped files.

5.1.4 Call Appropriate Counter(s) and Process Embedded Languages

For each file, the appropriate language-specific code counter is called, and the SLOC are processed for counts and statistics. Details of the code counters are described later. For many Web languages such as HTML and PHP, multiple languages may be embedded in one file. These are typically scripting languages such as JavaScript and VBScript. These files are parsed and split into multiple temporary in-memory file maps for later processing by their respective language-specific code counters.

5.1.5 Determine Duplicate Files

Duplicate files are processed separately in order to minimize the effects of counting multiple copies of the same file on the estimation of work effort in SLOC. A duplicate file is one that is determined to be a match to another file in the same baseline. The rules for duplicate matching are as follows:

1. Matching file names
    a. Both files are empty.
    b. Only comments and/or whitespace are different.
    c. If [-tdup #] is specified and the percent difference in LSLOC is less than the specified threshold (#)
2. Different file names
    a. Both files have identical content, including comments and whitespace.

The first file in the set of duplicate files is determined to be the original source of the duplicates and will be not be processed separately with the other duplicate copies. When processing through the DiffTool derived class and multiple duplicate files are found, UCC attempts to select a file with a match in the other baseline as the original source of the duplicates in order to minimize the number of unmatched duplicates processed during differencing. The duplicate file pairs are reported in the DuplicatePairs.txt file.

5.1.6 Print Counting Results

Print functions are common for each of the languages since these are performed within the MainObject class. There are methods for printing out both SLOC counts and keyword statistics. The SLOC counts report both physical and logical SLOC, SLOC types (compiler directives, data declarations, executable statements), blank lines and comments, and language-specific keyword counts. Keyword reports contain summaries of key statistics such as mathematical functions, logical statements, assignments, etc. For Web languages where multiple languages are contained in the same file, the output is specialized to delineate the individual languages counted.

Results are normally reported in comma-separated variable (CSV) files. Results can also be reported in plain text files using the [-ascii] option. There is also a [-legacy] reporting option that produces results in static, formatted, plain text files. The purpose of this option is to support the U.S. Government automated database import and processing of UCC results, which currently requires the results files to be unchanging.

5.2 DiffTool Class

The DiffTool class is a class derived from the MainObject class. It includes additional methods to process multiple code baselines and perform LSLOC comparisons. The functions are an extension of those for the MainObject class. Unique functions that are not contained in the MainObject class are marked with an asterisk (*).

- Parse command-line parameters.
- Process file lists for each baseline.
- Read files into memory.
- *Determine file matches across baselines.

- Call appropriate counter(s) and process embedded languages for each baseline.
- Determine duplicate files for each baseline.
- Print counting results for each baseline.
- *Perform differencing comparison between baselines.
- *Print differencing results.

Since the DiffTool class inherits the MainObject class, much of the processing is the same except that two baselines of code are processed at each step. Differences are addressed below.

5.2.1 Parse Command-Line Parameters

There are some additional command-line options pertaining to the DiffTool class that become available when the [-d] parameter is specified (this invokes the DiffTool object). The first parameter [-t #] allows a user to specify a threshold percentage for a modified line across the two code baselines. If the parameter is specified as –t 60 (the default), then any logical line that has changed by more than 60% will be considered replaced (one line added and one line deleted) rather than modified. Also available are the [-i1] and [-i2] parameters that allow a user to specify alternate file list files for each baseline (overrides default fileListA.txt and fileListB.txt).

5.2.2 Determine File Matches Across Baselines

In order to compare two code baselines, UCC attempts to match all of the files in baseline A to baseline B. Files in baseline A that do not have a match in baseline B have their total number of lines reported as deleted. Files in baseline B that do not have a match in baseline A have their total number of lines reported as added. Matching is simple if all files have a match where the name is identical and the relative path is similar. However, some files may have been moved, requiring an algorithm to determine the best possible match (file names must be identical). This process is critical since the failure to correctly match files would greatly overstate the degree of change between the baselines because unmatched files would all be counted as deleted and added.

UCC uses an implementation of the Gale-Shapley stable matching algorithm [12]. Each possible file pairing is given a weighted preference based on the similarity of the file path. Then the Gale-Shapley algorithm is executed to determine a stable matching set based on optimal preference alignment. This algorithm assures that given the two file baselines, where each file has ranked all files with the same file name of the other baseline with a unique number in order of preference, files are matched such that there are no two files that would “prefer” each other than their current match. If there are no such pairs, all the matches are considered stable. The list of matching file pairs is reported in the MatchingPairs.txt file.

This process is executed before duplicate file lists are compiled since unmatched files are not preferred to be declared the original sources of the duplicates in order to minimize the number of unmatched duplicates processed during differencing. The process is also executed a second time after each baseline is counted only for the temporary in-memory file maps of Web languages that were generated for later processing by their respective language-specific code counters. This second process also includes duplicate files.

5.2.3 Perform Differencing Comparison Between Baselines

Each of the matching file pairs is processed through a CmpMngr object. The CmpMngr compares the file LSLOC lists and returns the counts of added, deleted, modified, and unmodified lines. For Web languages, each of the component language results are summed to provide a complete differencing account for the file pair. Details of the comparison process are provided in the CmpMngr class description.

5.2.4 Print Differencing Results

The differencing results report the tally of added, deleted, modified, and unmodified logical lines. As noted in the MainObject class description, results are normally reported in comma-separated variable (CSV) files but may also be produced as plain text files using the [-ascii] or [-legacy] options.

5.3 CmpMngr Class

During the SLOC counting process, each logical line is stored in an array to be used in differencing. The CmpMngr class compares two lists of LSLOC and reports the number of added, delete, modified, and unmodified logical lines. Harley Green of Aerospace developed the GDiff07 algorithm for comparing the LSLOC. The GDiff07 algorithm breaks the counting process for finding the differences between the two baseline files into several steps.

1. Find and remove unmodified lines (exact matches).
2. Find and remove modified lines (within threshold [-t #]).
3. Count remaining lines as added (new baseline) or deleted (original baseline).

All lines are compared for an exact match and counted. During this process, lines that are not matched are placed into a separate list for each of the two files. Then the remaining lines are searched again for modified matches based on the default or user-specified percentage threshold [-d #]. Once the unmodified and modified lines are removed, the remaining lines either will be added or deleted, and no further action is necessary.

5.4 CCodeCounter Class

The CCodeCounter class is the base class for each of the language-specific counters. It contains members and methods for supporting all of the counting functions. The counting process consists of a number of common steps, some of which are optional based on the specific language syntax requirements.

1. Perform precount processing.
2. Count blank lines.
3. Count whole line and/or embedded comments.
4. Count keywords (language keywords, operators, structures, etc.).
5. Count compiler directive SLOC.
6. Perform language-specific processing (contained in language subclass).

The logic and basic methods for performing these actions is contained in the CCodeCounter class. However, the CCodeCounter class should never be instantiated by itself. An object is created for each of the available language-specific classes that inherit the CCodeCounter class. This allows for all of the required language-specific file extensions, delimiters, and keywords to be available, as well as a language-specific process that handles the particular details of each language-syntax requirements.

5.4.1 Perform Precounting Processing

Some languages may require special processing before executing counting activities. For example, the Perl language counter replaces some difficult characters with “$” characters to avoid collision with other parsing logic. This method is typically overridden by the derived-language class as needed.

5.4.2 Count Blank Lines

The processing of blank lines is simple and common among any software language. Each line is checked for any nonwhitespace characters and counted as blank if none are found.

5.4.3 Count Whole Line and/or Embedded Comments

Each language counter has specified character(s) to denote the beginning of a line comment and the beginning and end of a block comment (if applicable). Common processing parses each file, counts each comment by type, and removes the comments from the file line map.

5.4.4 Count Keywords

Since standard complexity analysis is currently beyond the scope of UCC, the software collects statistics on the occurrences of significant keywords, operators, structures, etc. in order to provide some information on the contents of each file. This requires a list of these items for each language counter. Common processing simply searches each file for the count of each keyword. The statistics are reported for each file as type totals, with a summary of the totals for each specific keyword at the end.

5.4.5 Perform Language-Specific Processing

The heart of the code counters is contained within the derived-language counter classes. These contain language-specific methods used to process physical and logical SLOC. The logical SLOC are stored in an array for use by the differencing functions if needed. Any new language requires these methods to be contained in a class derived from the CCodeCounter.

6. Adding a New Language Counter

UCC is structured so that the controlling classes of UCC are mostly language independent. This includes functions such as file handling, language selection, line comparison (differencing), and results reporting. The only requirement for an individual language counter, besides a few hooks in the controlling classes, is the subclassing of the CCodeCounter class. This design provides uniform processing and reporting leading to consistent results and reports.

6.1 Counting Standards Document

The first step in adding a new language counter is not actually a software programming task. A "counting standards document" is required that defines the SLOC counting rules based on the language-specific syntax. If this document does not already exist for the new language, the counter development team must create one based on the established format. A common method is to copy an existing counting standards document and modify it. The purpose of this document is to describe a set of predefined rules for counting physical and logical SLOC. This provides a basis for designing and implementing a language code counter.

This document should contain basic rules for counting physical and logical SLOC. It should provide definitions of the items counted, including physical SLOC, logical SLOC, data declarations, compiler directives, blank lines, comment lines, and executable statements. The most important feature (from a programming standpoint) is the illustration of example SLOC counting that details specific language constructs such as selection statements, iteration statements, jump statements, expressions, blocks, data declarations, and compiler directives.

6.2 Subclass CCodeCounter

As discussed in the previous section, the CCodeCounter class is the base class for each of the language-specific counters. It contains the members and methods for supporting all of the counting functions. The class derived from this for the new language counter will only override methods that contain unique features to the new language. Specific steps are as follows where "NewLang" is the name of the language to be added:

1. Create the new subclass header and code files.
    a. Create the CNewLangCounter.h and CNewLangCounter.cpp files.
    b. These may be copied from a similar language counter.
2. Declare the class constructor variables and lists.
    a. Set the class type and name (NEW_LANG, "NewLanguage").
    b. Set the quote, escape, and line continuation characters (if any).
    c. Set the block and/or line comment delimiters (/* */ or //).
    d. Set the file extensions (.c, .cc, .cpp, .h, …).
    e. Set the language keywords (directives, data, executable).
    f. Set the operation keywords (math, trigonometric, logarithmic, …).
3. Create a CountDirectiveSLOC method (if applicable).
    a. This method parses out all precompiler directives defined in the constructor.

4. Create a LanguageSpecificProcess method.
    a. This method processes each line for physical and logical lines according to language-specific syntax rules.
    b. This method should call an LSLOC method to parse the logical lines.
5. Create an LSLOC method.
    a. This method extracts logical lines of code and stores them by calling the results::addSLOC method.
    b. The logical SLOC stored by this method are used in the differencing process.
6. Create any additional new or override methods required by the new language.
    a. This may include PreCountProcess or additional parsing methods.

6.3 Update CWebCounter (if needed)

If adding a Web language capable of being embedded within another Web language, the CWebCounter.h and CWebCounter.cpp files may need to be modified. For example, JavaScript may be embedded within HTML code. The CWebCounter class separates multiple languages based on tags and creates temporary in-memory files to be parsed by associated individual language counters. This may also require modification of MainObject.cpp in order to process new basic Web language types (HTML, ASP, JSP. PHP).

6.4 Add Hooks to New Language Counter

The new language counter must be made available to the main process. This is completed through inclusion in several of the main class files:

1. Add the new language class type to the list of define statements in cc_main.h.

```
#define C_CPP        1    // C/C++
#define JAVA         2    // Java
#define JAVA_JSP     3    // Java in JSP
#define NEW_LANG     4    // New Language
```

2. Add the subclass include file to MainObject.h.

```
#include “CCCounter.h”
#include “CJavaCounter.h”
#include “CCsharpCounter.h”
#include “CNewLangCounter.h”
```

3. Create an instance of the new language counter in MainObject.cpp.

```
tmp = new CCCounter;
CounterForEachLanguage.insert(
    map<int,CCodeCounter*>::value_type(C_CPP, tmp));
tmp = new CJavaCounter;
```

```
CounterForEachLanguage.insert(
    map<int,CCodeCounter*>::value_type(JAVA, tmp));
tmp = new CNewLangCounter;
CounterForEachLanguage.insert(
    map<int,CNewLangCounter*>::value_type(NEW_LANG, tmp));
```

7. Summary

The UCC provides a standard solution to a problem that previously had no industry standard. The underlying logical source lines of code-counting rules are designed to be compatible with the Software Engineering Institute’s (SEI) Code Counting Standard. This solution is supported by a U.S. Government cost analysis group to provide a common software line-counting method to be applied to national space software programs. The collaboration between USC-CSSE and The Aerospace Corporation has produced a nonproprietary, open-source solution proposed as an industry standard for software costing based on source lines of code.

8. References

[1] USC-CSSE web site, http://csse.usc.edu.

[2] Park, Robert E., "Software Size Measurement: A Framework for Counting Source Statements," CMU/SEI-92-TR-020, September 1992.

[3] USC-CSSE Affiliates website, http://csse.usc.edu/csse/affiliate.

[4] Subversion website, http://subversion.tigris.org.

[5] Bugzilla website, http://www.bugzilla.org.

[6] Wikipedia website, http://en.wikipedia.org/wiki/Unified_Code_Count(UCC).

[7] Nguyen, V., and M. Sperka, "Test Plan: Unified Code Counter with Differencing Tool Functionality," January 2010.

[8] USC-CSSE CodeCount website, http://sunset.usc.edu/research/CODECOUNT.

[9] ANSI/ISO C++ Draft Standard, http://www.ansi.org.

[10] GNU Complier Collection website, http://gcc.gnu.org.

[11] USC-CSSE, "C/C++ CodeCount Counting Standard," June 2007.

[12] Gale, David, and Lloyd. S. Shapley, "College Admissions and the Stability of Marriage," American Mathematical Monthly, 69, pp. 9-14, 1962.