**Gathering Software Metrics from Software Version Control Systems and Automated Build Systems**

Mark Cheung, TJHSST 2010

TJHSST Computer Systems Lab, 2009-2010

Mentor: Michael Ihde

Northrop Grumman Information Systems

<http://csse.usc.edu/csse/research/COCOMOII/cocomo_main.html>

**Abstract:** Software metrics are tools that can be used to measure in quantity the software development. They are often used to estimate the cost and resource requirements, the productivity, the data collection, the code quality, and other software performances. The purpose of this project is to develop tools that support the automated production of software metrics from version control systems such as Subversion and build systems like CruiseControl. The project produces code count programs that generate the physical and logical source lines of codes (SLOC) for Java and Midas code. The project will?? also includes the development of tools that assist in generating reports with the COCOMO model and performing analysis of the results. **Results here.**

 *Keywords:* SLOC, Logical SLOC, COCOMO, automated, software metrics, code quality

**Introduction:**

 In 2001, Red Hat Linux 7.1 was found to hold 30,152,114 Physical Source Lines of Code. Using the Basic COCOMO model, this figure implies 7955.75 person-years development effort, making a total of 6.53 years and $1,074,713,481 cost (Wheeler). Although these figures are estimates and assumptions were made in calculation, the resulting Physical SLOC can still give us insights into the Linux 7.1 system.

Estimating software cost requires careful calculation and analysis of the software project. A mistake in evaluation can destabilize the development by allocating either too much or too little resources.

**Background:**

**Source Lines of Code (SLOC)**: Source Lines of Code is a code metrics for counting the number of lines for a set of programs and thereby estimating the amount of effort required. There are various types of counting that can be done, including total lines, commands, compiler directives, executive statements, non-blank lines, physical lines, logical lines, tokens. Tokens are operators or operands such as “while”, and “eof.” SLOC measures are divided into two major types: physical and logical. Physical SLOC measures the code directly; it includes the blank lines, comments lines, and other logical and style conventions. In contrast, Logical SLOC accounts for code conventions such as the closing and ending braces of a for-loop (in this case it would be 1 Logical SLOC and 2 Physical SLOC). Consequently, SLOC for one language should not be used for another due to the syntactical differences among them.

SLOC varies by language. Different programming language has different syntax, structures, and expressions. A simple example is the comment delimiter. C, C++, and Java, uses “//” to symbolize comment until the end of line while Python, Maple, and Perl uses “#” to represent so. Certain type names in languages are also different, especially strings: C uses “char[]” while Java, C# uses “String” to represent String type name. Expression of for loop in Java and C is something like “for(int c=1; c<=10; c++),” but in x-Midas and Ruby, there needs to be an “end” statement after the recursion. The point here is that different language has different productivity and quality. C++, Java, Visual Basic, and Python are credited with higher productivity, reliability, and simplicity than low-level languages such as C. On average, a line of python is as expressive as six lines of C. Programmers can save time and effort when they use a simpler language. Table 2 shows ratios of High-Level-Language statements to equivalent C Code. The higher the ratio, the more each line of code in the language accomplishes.

Besides syntax, software effort is also affected by the programmer’s familiarity with the language. The linguists Sapir and Whorf hypothesize a correlation between a language and the ability to think certain thoughts. According to Sapir and Whorf, in order to think a thought, you first need to understand the words for expressing it (Whorf, 1956, as cited in McConnel, 2004, p. 63). As McConnel explains, the same can be said about programming languages (p.63).The expressions of a programming languages influence the thought process. In summary, using a high-level language can reduce the SLOC overall, but not necessarily the effort.

**COnstructive COst Model (COCOMO’81):** COCOMO, first published by Dr. Barry Boehm in 1981, takes the Code Metrics and computes the size, effort (time), and cost of a program. It allows programmers to estimate the development schedule in Person-Months and plan accordingly. The model classified the classes and projects into three modes: organic, semidetached, and embedded. Organic projects are familiar projects coded by a small team with good experience. They are typically within stable development environment and consist of no more than a hundred thousand SLOCs (100 KSLOC) Embedded project are less comprehensible projects requires operating within “tight” constraints such as hardware and software limitations, regulations (deadline), or operational procedures. There may be high cost on changing parts and projects are usually no on well known fields. Semidetached mode is the intermediate stage of development between the embedded and organic projects. Developers have a reasonable understanding of the system and the team usually comprised of both experienced and inexperienced developers, or even developers specialized for some project aspects.

**Basic COCOMO’81:**

Effort Applied = ab(KSLOC) bb ( mm [man-months](http://en.wikipedia.org/wiki/Man-month) )

Development Time = cb(Effort Applied) db (months)

People required = Effort Applied / Development Time (count)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Software project | aa | bb | cc | dd |
| Organic | 2.4 | 1.05 | 2.5 | 0.38 |
| Semi-detached | 3.0 | 1.12 | 2.5 | 0.35 |
| Embedded | 3.6 | 1.20 | 2.5 | 0.32 |

Effort Applied ACT (Annual Change Traffic)= ACT \* Effort Applied d

In a year, 40 KSLOC project that increased by 2 thousands lines with modified 3 thousands modified lines. ACT= (3000+2000 )/ 40000=0.125

**Intermediate Constructive Cost Model (Intermediate COCOMO’81):** Like Basic COCOMO’81, Intermediate COCOMO calculate the development effort as a function of the code metrics. However, it also considers a set of “cost drivers” concerning assessments of the software, hardware, developers’ attributes, and project attributes. Each driver has been estimated with the following grades: VL, L, N, H, N, H, VH, and EH. See table 1 in appendix for more detail.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Software project | aa | bb | cc | dd |
| Organic | 3.2 | 1.05 | 2.5 | 0.38 |
| Semi-detached | 3.0 | 1.12 | 2.5 | 0.35 |
| Embedded | 2.8 | 1.20 | 2.5 | 0.32 |

C=Effort Adjustment factor (multiplying values of cost drivers) (see table2 for a list of cost drivers)

Effort Applied = ab(KSLOC) bb \*C ( mm [man-months](http://en.wikipedia.org/wiki/Man-month) )

**Example:**

Cost Drivers:

 Software reliability is low  0.88

 Low database size  0.94

 Very high execution time constraint  1.30

 High Programmer capability 0.86

 Other cost drivers assumed nominal 1

C=0.88\*0.94\*1.30\*0.86\*1≈0.93

(Wheeler SLOCCount)

**Constructive Cost Model detailed model:** ?

**Constructive Cost Model II:** COCOMO II is the ?

**Costar 7.0:** Costar is an extension of COCOMO that estimate a project’s duration, staffing levels, effort and cost that provide 20% within the actual results 70% of the time. In addition to providing COCOMO, Costar 7.0 takes into account 32 cost drivers. With Costar 7.0, the user can make exchanges with “what-if” analysis to find the optimal project plan. (more?)

**Eclipse Modeling Framework Project (EMF):** EMF is a modeling framework and code generation facility. Given the definition of the model in XMI (XML Metadata Interchange), EMF can create a working set of Java classes for it. The generated classes and methods are tagged in Javadoc comment with @generated. It is difficult to measure the effort require for EMF generated code because the code falls into two camps: instrumented and generated. The XMI for the EMF may take 30 minutes to a few days to program, and the results may be hundreds or even thousands lines of code.

**Relevant/Similar Projects:**  StatSVN/StatCVS, USC CodeCount, Sonar, Checkstyle 5.0, Hackystat

**StatSVN:** StatSVN is open source software based on StatCVS that retrieves information from Subversion repository (Eclipse) and generates data on the development status. At Northrop Grumman, this is hooked to the Hobgoblin Metrics Collection Server. StatSVN counts the number of files, average file size, code per directory, lines of code (LOC), and Churn per day. StatSVN also produces a repo heatmap, which is an applet that shows all files in a hierarchical manner (see below). The tags of the rectangles are lists of directories and the bigger the directories, the larger the rectangles. The color shows change in LOC ( LGPL).

StatSVN.

Below is an example output StatSVN using SCLC (Source Code Line Counter) for Northrop Grumman “P” Project.

NCSL: Non-Comment-Source-Line

AGSL: Auto-Generate-Source-Line

AESL: Assembly-Equivalent-Source-Lines

Lines Blank Cmnts NCSL AGSL AESL

====== ====== ====== ====== ====== ======

44066 5564 1137 37365 0 822030 ----- HTML ----- (451 files)

 50 0 0 50 0 1100 ----- CSS ----- (2 files)

 188 23 33 132 0 1980 ----- shell ----- (11 files)

**177632 16864 90787 70438 49619 422628 ----- Java ----- (682 files)**

221936 22451 91957 107985 49619 1247738 \*\*\*\*\* TOTAL \*\*\*\*\* (1146 files)

**USC CodeCount** (Ultimate Code Line Accumulator Tool) is a C language toolset that produces software metrics with two possible Source Lines of Code (SLOC)—physical and logical SLOC. Physical SLOCs is the sum of the program’s source code including the commented lines while logical SLOC is the total number of statements are statements that should be counted for less than its number of lines (e.g. “if” and “endif” are redundancy and should be counted as just one logical SLOC). The USC CodeCount supports several programming languages—C/C++, C#, Java, JavaScript, MUL, Pearl, SQL, and XML. It generates a report in .dat format that includes the total lines, total blank lines, total embedded comments, total compiler directives lines, total data declaration lines, total execution instructions lines of each file. It also reports the Physical and Logical SLOC and their ratio.

**Checkstyle 5.0:** Checkstyle is development tool that makes automation of checking Java code to adhere to a coding standard. Its standard checks include AnnotationUseStyle, ModifierOrder, GenericWhitespace, BooleanExpressionComplexity, and many more.

**Sonar:** Sonar is a code quality management platform that can collect, analyze, and report metrics on code. It calculates LOC, total classes, comments, duplications, and violations (a total of 169 rules). Projects at risk (due to duplications, violations, complexities, comments), can be easily detected with Sonar. Sonar currently supports Java and PL/SQL languages, but its algorithms are extensible to cover other languages. See below for a snapshot of a generated report.



**Hackystat:** Hackystat is an open source project for “collection, analysis, visualization, interpretation, annotation, and dissemination” of software metrics. It allows users to attach sensors to their code, and the sensor would collect the data about development and forward it to the repository called Hackystat SensorBase. Hackystat Sensorbase allows developers to share data in the same way as sharing journal entries on blog.

**Midas:** The traditional Midas has three types of programming: primitives, libraries, and scripts/macros. Primitives are stand-alone executable codes in C++ or Fortran. Libraries are sometimes used in place of primitive to add native-code functionality, especially in X-Midas. The program will be focusing on scripts/macros, which are chain commands of macro (scripting) language that can be easily recall and run by a shell. A sample macro is below:

In the above script, SMOOTH, CALC, P\_START, CONSTANT TEMPWORK REAL winlen 1 are abbreviated commands. The characters followed the forward slash (2 and winlen 0) are switches which are used to tell the function to run in a certain way. Sometimes, there are ampersands at the end of a line to signify continuation to the following line. “!’ denotes the comment delimiter. The macro required startmacro and endmacro commands but “startmacro” can be replaced by the name of the macro.

Some other compiler Directives are subroutine/endsubroutine, procedure, startcontrols/endcontrols, and pipe on/pipe off.

Like other languages, Midas also uses control statements to indicate a sequence. This includes if, while, loop, break, goto, label, continue, call, return, and local.

**Project Components:**

**Software:**

**Eclipse:** Eclipse is a multi-language IDE with various components (plugins) that allow for development of Java, C/C++, Python, Perl, web applications etc. It is soft coded; the plug-ins it employed provide all of its functionality including the runtime system. Eclipse open source has more than 60 different projects, which are organized into seven categories: enterprise development, embedded and device development, Rich Client Platform, Rich Internet Applications, Application Frameworks, Application Lifecycle Management (ALM), and Service Oriented Architecture (SOA). I am using Eclipse C/C++ Development Tooling (CDT), RCP/Plug-in Development Environment (PDE), and Rich Ajax Platform (RAP).

**Eclipse Rich Client Platform (Eclipse RCP):** Eclipse RCP is an open source Java-based development platform composed of a minimal set of plug-ins for building a platform application. It is portable in that the components are Java based and widgets have native implementations. By using RCP, developers can make use of the existing codebase for speed purpose. It also allows developers to write in C++ the GUI development in Java. Each of the bundled features can easily be implemented Equinox OSGi standard bunding.

**Eclipse Rich Ajax Platform (Eclipse RAP):** Eclipse RAP is very similar to Eclipse RCP. However, instead of Standard Widget Toolkit, it implements SWT API with RWT. This allows rendering of widgets on a web-enable application from a single code base and reuse of code and development tools.

Eclipse RCP and RAP Comparison:



(Source: <http://www.eclipse.org/rap/introduction.php>)

**Languages:**

 **C -** The language of the modified USC CodeCount uses.

 **Java –**The language of the code of which the program generates a code metrics (SLOC).

 **Midas-** The language of the code of which the program generates a code metrics (SLOC).

**Methods:**

1)     Evaluate the Northrop Grumman Ultimate Code Line Accumulator Tool (based on USC CodeCount)

2)      Extend CodeCount language support as necessary with C: (i.e. X-MIDAS and NextMIDAS scripting languages, Java, Python) (done)

3)      Enhance CodeCount as necessary to support counting of auto-generated source code (done)

3.5) compare StatSVN with Logical SLOC

4)      Select/Develop tool to perform automatic production of metrics and store them in a database (Hackystat)

5)      Develop single-source plugins for Eclipse RAP and Eclipse RCP with Java to support report generation and analysis of metrics database.

6)      Extend metrics collection to include code-quality, code-reuse, code-churn data, build failures, etc.

**Method (in greater detail):**

 Java:

 -Build upon existing USC Java Codecount

-modified methods labeled with @generated NOT or methods without this tag will be ignored during regeneration.

**extern** **void** **generatedcheck**(**char** line[], **int** line\_length, bool\_type \*generated)

 X-Midas Macro Script:

 Macro syntax

 **#define** DIRR\_NAME\_LIST \

 "\* SUBROUTINE "\

 "\* PROCEDURE "\

 "\* STARTMACRO "\

 "\* ENDMACRO "\

 "\* PIPE ON "\

 "\* PIPE OFF "\

 "\* XPIPE ON "\

 "\* XPIPE OFF "\

 "\* local "\

 " "

**#define** CONTROL\_STATEMENTS\_LIST \

 "\* ELSE "\

 "\* LOOP "\

 "\* BREAK "\

 "\* IF "\

 "\* GOTO "\

 "\* LABEL "\

 "\* CALL "\

 "\* RETURN "\

 "\* LOCAL "\

 "\* WHILE "\

 "\* ENDIF "\

 "\* ELSEIF "\

 " "

**extern** **void** **Ampersand**(**char** line[], **int** line\_length, bool\_type \*found\_ptr)

**Results:**

**Logical +Physical SLOC Comparison:**

 Total Blank | Comments | Compiler Cont. Comm. | Number | File SLOC

 Lines Lines | Whole Embedded | Direct. Stat. Instr. | of Files | SLOC Type

-------------------------------------------------------------------------------------------------

 19 1 | 4 0 | 3 3 8 | 1 | 14 CODE Physical

 19 1 | 4 0 | 2 2 8 | 1 | 12 CODE Logical

**Sd350 and Midastest result (Midas):**

 Total Blank | Comments | Compiler Cont. Comm. | Number | File SLOC

 Lines Lines | Whole Embedded | Direct. Stat. Instr. | of Files | SLOC Type

-------------------------------------------------------------------------------------------------

 36 4 | 7 0 | 4 6 15 | 2 | 25 Physical

 36 4 | 7 0 | 3 5 15 | 2 | 23 Logical

Number of files successfully accessed........................ 2 out of 2

Number of files with :

 Commands > 100 = 0

 Data Declarations > 100 = 0

 Percentage of Comments to SLOC < 60.0 % = 2 Ave. Percentage of Comments to Logical SLOC = 30.4

Total occurrences of these Midas Keywords :

 Compiler Directives Data Keywords Commands

 SUBROUTINE......... 0 ELSE............... 1 CALC

 PROCEDURE.......... 0 LOOP............... 0 CONSTANT

 STARTMACRO......... 0 BREAK.............. 0 FASTFILTER

 ENDMACRO........... 0 IF................. 2 MARRAY

 PIPE ON............ 0 GOTO............... 0 MFFT

 PIPE OFF........... 0 LABEL.............. 0 RQFSHIFT

 XPIPE ON........... 0 CALL............... 0 P\_START

 XPIPE OFF.......... 0 RETURN............. 0 SMOOTH

 local.............. 0 LOCAL.............. 0 STATUS

 WHILE.............. 0 WAVEFORM

 ENDIF.............. 2 WAVEFORM

 ELSEIF............. 1 STATISTICS

 XRTDISPLAY

 XRTPLOT

 XRTRASTER

**Java CodeCount : Northrop Grumman Project:**

Total Blank | Comments | Compiler Data Exec. | Number | File SLOC

 Lines Lines | Whole Embedded | Direct. Decl. Instr. | of Files | SLOC Type Definition

-------------------------------------------------------------------------------------------------------------------------------

 170150 16252 | 86210 441 | 7321 12256 48111 | 650 | 67688 CODE Physical

 170150 16252 | 86210 441 | 7321 4512 32513 | 650 | 44346 CODE Logical

 Generated SLOC:

Total Generated Lines: 110865 Total Blank | Comments | Compiler Data Exec. | Number | File SLOC

 Lines Lines | Whole Embedded | Direct. Decl. Instr. | of Files | SLOC Type Definition

------------------------------------------------------------------------------------------------------------------------------------

 110865 9525 | 59469 328 | 0 7980 33954 | 414 | 41934 CODE Physical

 110865 9525 | 59469 328 | 0 2687 21249 | 414 | 23936 CODE Logical

Number of files successfully accessed........................ 650 out of 650

Ratio of Physical to Logical SLOC............................ 1.53

Number of files with :

 Executable Instructions > 100 = 152

 Data Declarations > 100 = 7

 Percentage of Comments to SLOC < 60.0 % = 87 Ave. Percentage of Comments to Physical SLOC = 128.0

Total occurrences of these Java Keywords :

 Compiler Directives Data Keywords Executable Keywords

 import............. 7321 abstract........... 5 goto............... 0

 const.............. 0 if................. 2561

 boolean............ 788 else............... 423

 int................ 1647 for................ 139

 long............... 38 do................. 0

 byte............... 60 while.............. 30

 short.............. 21 continue........... 1

 char............... 0 switch............. 531

 extends............ 636 case............... 2591

 float.............. 67 break.............. 82

 double............. 201 default............ 74

 implements......... 291 return............. 6284

 class.............. 548 super.............. 1376

 interface.......... 144 this............... 3803

 native............. 0 new................ 2791

 void............... 2186 try................ 183

 static............. 1159 throw.............. 100

 package............ 650 throws............. 10

 private............ 1223 catch.............. 178

 public............. 5919

 protected.......... 1369

 operator........... 0

 volatile........... 0

**Discussion:**

**Comparison between SCLC and CodeCount (Physical and Logical):**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | SCLC | CodeCount total | CodeCount generated | CodeCount (not generated) |
|  |   | Physical SLOC | Logical SLOC | Physical SLOC | Logical SLOC | Physical SLOC | Logical SLOC |
| number of files | 650 | 650 | 650 | x | x | x | x |
| total lines | 170150 | 170150 | 146808 | 110928 | 92930 | 59222 | 53878 |
| blank lines | 16252 | 16252 | 16252 | 9525 | 9525 | 6727 | 6727 |
| comments | 86665 | 86210 | 86210 | 59469 | 59469 | 26741 | 26741 |
| embedded | x | 441 | 441 | 328 | 328 | 113 | 113 |
| compiler directives | x | 7321 | 7321 | 0 | 0 | 7321 | 7321 |
| data declarations | x | 12256 | 4512 | 7980 | 2687 | 4276 | 1825 |
| executive instructions | x | 48111 | 32513 | 33954 | 21249 | 14157 | 11264 |
| non-comment source lines | 67669 | 67688 | 44346 | 41934 | 23936 | 25754 | 20410 |
| Auto-Generate-Source-Line | 47114 | x | x | x | x | x | x |
| asembly-equivalent-source-lines | 406014 | x | x | x | x | x | x |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Effort Applied |  |  |  |  |  |  |  |
|  | Physical SLOC | Logical SLOC |  |
|  | Total | Generated | Ungenerated | Total | Generated | Ungenerated |  |
| Lines (in 1000s) | 170.15 | 110.928 | 59.222 | 146.808 | 92.93 | 53.878 |  |
| organic (mm) | 703.9203643 | 449.2035919 | 232.4112796 | 602.8887593 | 373.0041917 | 210.4418047 |  |
| semi-detached (mm) | 945.4815153 | 585.5545764 | 289.9361002 | 801.4583235 | 480.2373173 | 260.7967029 |  |
| embedded (mm) | 1330.933558 | 796.5397451 | 375.0919255 | 1114.95592 | 644.0879229 | 334.8511467 |  |
|  |  |  |  |  |  |  |  |
| Development Time |  |  |  |  |  |  |  |
|  | Physical SLOC |  |  |  | Logical SLOC |
|  | Total | Generated | Ungenerated | Total | Generated | Ungenerated |  |
| Lines (in 1000s) | 170.15 | 110.928 | 59.222 | 146.808 | 92.93 | 53.878 |  |
| organic (months) | 30.19944418 | 25.46058818 | 19.82064328 | 28.47279751 | 23.72418049 | 19.08667046 |  |
| semi-detached (months) | 27.50543854 | 23.25886894 | 18.18636115 | 25.95962048 | 21.69949242 | 17.52450431 |  |
| embedded (months) | 24.98446469 | 21.19953783 | 16.65958112 | 23.60817806 | 19.80625921 | 16.06543841 |  |
|  |  |  |  |  |  |  |  |
| People required  |  |  |  |  |  |  |  |
|  | Physical SLOC |  |  | Logical SLOC |  |  |  |
|  | Total | Generated | Ungenerated | Total | Generated | Ungenerated |  |
| organic (people) | 23.30905033 | 17.64309562 | 11.7257183 | 21.17420176 | 15.7225322 | 11.02559009 |  |
| semi-detached (people) | 34.37434797 | 25.17553961 | 15.94250206 | 30.87326813 | 22.13126962 | 14.88183051 |  |
| embedded (people) | 53.27044524 | 37.57344861 | 22.51508743 | 47.22752928 | 32.51941298 | 20.84295106 |  |

WetResultsSummarySection.java

The results of SCLC and CodeCount are distinctive. The total lines, total comments, and total blank lines are different. I think these are due to of either less LOC for SCLC or difference code format.

For the Codecount, the difference between the Logical and physical lines is 176326-152077= 24249, which is approximately 13.75% of the total LOC. If we input these figures into the intermediate COCOMO’81 and assume the effort adjustment factor to be 1, the Physical SLOC results for an organic project are approximately 730.77 man-months, 30.63 months, and 23.86 people. For Logical SLOC, they are 656.63 man-months, 28.88 months, and 21.67 people. Compared with physical lines, Logical SLOC have an 15% decrease of total lines, an 11% decrease of effort, 6% decrease of time, and 10% decrease of staff.

If we also take into account the generated vs. ingenerated lines of codes, the results would be 61.168 lines of ungenerated Physical SLOC and 55.658 lines of Logical SLOC. Calculation. s

**EMF COCOMO:**

If generated portion is more than 60% of the total lines of codes (115158/176326). If we use this to calculate the COCOMO model, the resulting value would be way off the actual effort. This shows that failing to account for auto-generated code in EMF throws the COCOMO model off:

References

Clayberg, E., & Rubel, D. (2006). *Eclipse: Building Commerical-Quality Plug-ins* (E. Gamma, L. Nackman, & J. Wiegand, Eds.). Boston, MA: Pearson Education.

COCOMO. (2009, November 15). *Wikipedia, The Free Encyclopedia*. Retrieved November 18, 2009, from http://en.wikipedia.org/‌wiki/‌COCOMO

The Eclipse Modeling Framework [Computer software manual]. (n.d.). Retrieved from http://help.eclipse.org/‌ganymede/‌index.jsp?topic=/‌org.eclipse.emf.doc/‌references/‌overview/‌EMF.html

Johnson, P. (n.d.). Source Code Line Counter [Computer software and manual]. Retrieved from http://code.google.com/‌p/‌sclc/‌wiki/‌UserGuide

Johnson, P. (1989). *Hackystat*. Retrieved from Free Software Foundation website: http://code.google.com/‌p/‌hackystat/

Kealey, J., Daigle, J.-P., Mussbacher, G., Xhenseval, B., Jekot, M., & Northrop Grumman. (n.d.). StatSVN [Computer software and manual]. LGPL.

McConnell, S. (2004). Key Construction Decisions. In *Code Complete* (2nd ed., pp. 61-70). Redmond, WA: Microsoft.

*RAP/*‌*BIRT Integration*. (n.d.). Retrieved from http://wiki.eclipse.org/‌RAP/‌BIRT\_Integration

*RAP Project* [Introduction]. (n.d.). Retrieved from http://www.eclipse.org/‌rap/‌introduction.php

*Sonar* [Projects]. (n.d.). Retrieved from http://nemo.sonarsource.org/

Touesnard, B. (2004, February 23). *Software Cost Estimation: SLOC-based Models and the Function Points Model Version 1.1*. Retrieved from UNB website: http://bradt.ca/‌docs/‌SWE4103-report.pdf

Unified CodeCount (Version 2009.10) [Computer program]. (2006-2009). Los Angeles, CA: University of Southern California.

Weathersby, J., Bondur, T., Chatalbasheva, I., & French, D. (2008). *Integrating and Extending BIRT* (2nd ed.). Boston, MA: Actuate Corporation.

What is Eclipse? (n.d.). *Eclipse* [Eclipse Newcomers FAQ]. Retrieved from http://www.eclipse.org/‌home/‌newcomers.php

Wheeler, D. A. (n.d.). SLOCCount (Version 2.26) [Computer software manual]. Retrieved from http://www.dwheeler.com/‌sloccount/‌sloccount.html#cocomo

Wheeler, D. A. (2001). *More than a Gigabuck: Estimating GNU/*‌*Linux’s Size’*. Retrieved from http://www.dwheeler.com/‌sloc/‌redhat71-v1/‌redhat71sloc.html

**Acknowledgements:**

Mr. Michael Ihde, my mentor, Northrop Grumman, Eclipse Community, BIRT-Exchange Community, TJHSST community, etc.

**Appendix**

**Appendix(tables)**

Table 1

|  |  |
| --- | --- |
| Language | Level Relative to C |
| C | 1 |
| C++ | 2.5 |
| Fortran 95 | 2 |
| Java | 2.5 |
| Perl | 6 |
| Python | 6 |
| Smalltalk | 6 |
| Microsoft Visual Basic | 4.5 |

Source: *Estimating Software Costs* (Jones 1998), *Software Cost Estimation with Cocomo II* (Boehm 2000), and “An Empirical Comparison of Seven Programming Languages”, as cited in McConnell,2004, p. 62

Table 2:

|  |  |
| --- | --- |
| **Cost Drivers**  | **Ratings**  |
| **ID**  | **Driver Name**  | **Very Low**  | **Low**  | **Nominal**  | **High**  | **Very High**  | **Extra High**  |
| RELY  | Required software reliability  | 0.75 (effect is slight inconvenience)  | 0.88 (easily recovered losses)  | 1.00 (recoverable losses)  | 1.15 (high financial loss)  | 1.40 (risk to human life)  |    |
| DATA  | Database size  |    | 0.94 (database bytes/SLOC < 10)  | 1.00 (D/S between 10 and 100)  | 1.08 (D/S between 100 and 1000)  | 1.16 (D/S > 1000)  |    |
| CPLX  | Product complexity  | 0.70 (mostly straightline code, simple arrays, simple expressions)  | 0.85  | 1.00  | 1.15  | 1.30  | 1.65 (microcode, multiple resource scheduling, device timing dependent coding)  |
| TIME  | Execution time constraint  |    |    | 1.00 (<50% use of available execution time)  | 1.11 (70% use)  | 1.30 (85% use)  | 1.66 (95% use)  |
| STOR  | Main storage constraint  |    |    | 1.00 (<50% use of available storage) | 1.06 (70% use)  | 1.21 (85% use)  | 1.56 (95% use)  |
| VIRT  | Virtual machine (HW and OS) volatility  |    | 0.87 (major change every 12 months, minor every month)  | 1.00 (major change every 6 months, minor every 2 weeks) | 1.15 (major change every 2 months, minor changes every week)  | 1.30 (major changes every 2 weeks, minor changes every 2 days)  |    |
| TURN  | Computer turnaround time  |    | 0.87 (interactive)  | 1.00 (average turnaround < 4 hours)  | 1.07  | 1.15  |    |
| ACAP  | Analyst capability  | 1.46 (15th percentile)  | 1.19 (35th percentile)  | 1.00 (55th percentile)  | 0.86 (75th percentile)  | 0.71 (90th percentile)  |    |
| AEXP  | Applications experience  | 1.29 (<= 4 months experience)  | 1.13 (1 year)  | 1.00 (3 years)  | 0.91 (6 years)  | 0.82 (12 years)  |    |
| PCAP  | Programmer capability  | 1.42 (15th percentile)  | 1.17 (35th percentile)  | 1.00 (55th percentile)  | 0.86 (75th percentile)  | 0.70 (90th percentile)  |    |
| VEXP  | Virtual machine experience  | 1.21 (<= 1 month experience)  | 1.10 (4 months)  | 1.00 (1 year)  | 0.90 (3 years)  |    |    |
| LEXP  | Programming language experience  | 1.14 (<= 1 month experience)  | 1.07 (4 months)  | 1.00 (1 year)  | 0.95 (3 years)  |    |    |
| MODP  | Use of "modern" programming practices (e.g. structured programming)  | 1.24 (No use)  | 1.10  | 1.00 (some use)  | 0.91  | 0.82 (routine use)  |    |
| TOOL  | Use of software tools  | 1.24  | 1.10  | 1.00 (basic tools)  | 0.91 (test tools)  | 0.83 (requirements, design, management, documentation tools)  |    |
| SCED  | Required development schedule  | 1.23 (75% of nominal)  | 1.08 (85% of nominal)  | 1.00 (nominal)  | 1.04 (130% of nominal)  | 1.10 (160% of nominal)  |    |

**Source: Boehmcite after I receive the book**

**Appendix (Methods):**

**Summaries of Methods:**

**extern** **void** **comment\_processing**(bool\_type \*comment\_flag, **int** line\_length,

 **char** line[], **int** \*comment\_lines, **int** \*e\_comm\_lines)

/\*---------------------------------------------------------------------------\*/

/\* Procedure will scan each line for the comment delimiter ‘!’. When code \*/

/\* appears prior to comment delimiter, it indicates the comment is an \*/

/\* embedded comment. Set 'comment\_flag' to true for whole line comments. \*/

/\*---------------------------------------------------------------------------\*/

**extern** **void** **control\_statements\_processing**(bool\_type \*control\_flag,

 **int** line\_length, **char** line[], **int** \*control\_statements,

 target\_tally\_array\_type local\_control\_tally,

 target\_name\_array\_type control\_names, **int** control\_length,

bool\_type close\_match, bool\_type \*found, **int** \*line\_loc, **char** exclude[], target\_tally\_array\_type control\_tally)

/\*---------------------------------------------------------------------------\*/

/\* Procedure will scan each line for the control statements, which are \*/

/\* defined in the CONTROL\_STATEMENTS\_LIST.Declare SLOC to be a control \*/

/\* statement and set control\_flag accordingly. \*/

/\*---------------------------------------------------------------------------\*/

**extern** **void** **commands\_processing**(**char** line[], **int** line\_length, **int** \*comm\_lines)

/\*---------------------------------------------------------------------------\*/

/\* Procedure will scan each line for the commands, which are defined in \*/

/\* the CONTROL\_STATEMENTS\_LIST. Declare SLOC to be a control statement \*/

/\* and increases comm.\_lines accordingly. \*/

/\*---------------------------------------------------------------------------\*/

**extern** **void** **compiler\_directives\_processing**(bool\_type \*directive\_flag,

 **int** line\_length, **char** line[], **int** \*directive\_lines,

 target\_tally\_array\_type local\_dirr\_tally,

 target\_name\_array\_type dirr\_names, **int** dirr\_length,

 bool\_type close\_match, bool\_type \*found, **int** \*line\_loc, **char** exclude[],

 target\_tally\_array\_type dirr\_tally)

/\*---------------------------------------------------------------------------\*/

/\* Procedure will scan each line for the compiler directives, which are \*/

/\* defined in the DIRR\_NAME\_LIST. Declare SLOC to be a directive line \*/

/\* and set directive\_flag accordingly. \*/

/\*---------------------------------------------------------------------------\*/

**Appendix (files):**

**Sd350:**

STARTMACRO/xpipe/setup\dec=0 N:nample

!

! macro for emulating SD350 spectrum analyzer

!

local SFREQ, CFREQ, FWIDTH, FS

if /c gt 0

 WAVEFORM/t1=1k/ctag=3:SFREQ \_wave CF nsamples SIN

else dsdf

 WAVEFORM/t1=1k/ctag=3:SFREQ \_wave SF nsamples SIN

endif

RQFSHIFT/rs/fft-1k/ctag(2:CFREQ, 3:FWIDTH)/nowarn/wb=3 \_wave \_waves 0e-3 ,,

MFFT/rs/fftp=3 \_waves \_wavep 1k hann 0.0

STATUS/quiet \_wavep ,,,,, FS

XRTPLOT/xs=3/mtag=X:SFREQ/xcnt=2/mark=3 \_wavep(cl=/DEC\*FS) 1e-10 .5 lo ,, 70

XRTRASTER/xs=2/mtag=X:CFREQ/lpb=2/stretch \_wavep 0 0.25 lo mm 20

XRTDISPLAY/xs=4 SFREQ/"Sine Frequency" FWIDTH/"Filter Width"

!

ENDMACRO

**Macrotest:**

SMOOTH A: input A: output N: winlen

!

! smooths data file using rectangular filter

!

CALC/quiet P\_START (1-winlen)/2

CONSTANT TEMPWORK REAL winlen 1/winlen 0

P\_START 1 1

FASTFILTER input TEMPWORK output

STATISTICS file mean

IF mean GT 4.00

 MARRAY file file 0.5

ELSEIF mean LT 1.0

 MARRAY file file 2.0 ENDIF

For

**Appendix (Results):**

|  |  |  |
| --- | --- | --- |
|  | Physical SLOC | Logical SLOC |
|  | Total | Generated | Ungenerated | Total | Generated | Ungenerated |
| Lines (in 1000s) | 176.326 | 115.158 | 61.168 | 152.077 | 96.419 | 55.658 |
| organic (mm) | 730.7724256 | 467.2064155 | 240.4365419 | 625.6287411 | 387.7222572 | 217.7478896 |
| semi-detached (mm) | 984.0010751 | 610.6194898 | 300.6273226 | 833.743413 | 500.4761572 | 270.465679 |
| embedded (mm) | 1389.113263 | 833.1265366 | 389.9304655 | 1163.146117 | 673.214062 | 348.1698393 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Development Time |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  | Physical SLOC | Logical SLOC |
|  | Total | Generated | Ungenerated | Total | Generated | Ungenerated |
| Lines (in 1000s) | 176.326 | 115.158 | 61.168 | 152.077 | 96.419 | 55.658 |
| organic (months) | 30.63213211 | 25.84362031 | 20.07798791 | 28.87622064 | 24.07564256 | 19.3358163 |
| semi-detached (months) | 27.89256535 | 23.60259424 | 18.41831795 | 26.32093752 | 22.01527912 | 17.74921972 |
| embedded (months) | 25.32888424 | 21.50639006 | 16.86770133 | 23.93001533 | 20.08857015 | 16.26721384 |
|  |  |  |  |  |  |  |
| People required  |  |  |  |  |  |  |
|  | Physical SLOC | Logical SLOC |
|  | Total | Generated | Ungenerated | Total | Generated | Ungenerated |
| organic (people) | 23.85640095 | 18.07821079 | 11.97513132 | 21.66588034 | 16.10433683 | 11.26137559 |
| semi-detached (people) | 35.27825651 | 25.87086333 | 16.32219204 | 31.67605305 | 22.73312795 | 15.2381729 |
| embedded (people) | 54.84304993 | 38.7385579 | 23.11698896 | 48.60615848 | 33.51229366 | 21.40316361 |