January 2006 Volume II Issue 1

ORAtips

CIO Corner

How Much Is That Oracle® Database in the Window?

By Steve Callan, ORAtips Associate Editor (Database)

Editor's Note: Steve Callan has over 50 Oracle database articles and white papers to his credit. He joins ORAtips as our Oracle database Associate Editor to share lessons learned with our readers. One big lesson learned is to take a hard look at what it really costs you to develop and maintain your Oracle environment. For Steve's first article on Software Lifecycle Development, he discusses counting lines of code, including "how to" approaches, as a factor to consider in your typical software development estimate.

Abstract

For many reasons, accurately estimating the cost of Oracle software development is a difficult and complex problem. Despite the fact that most software developers have degrees in computer science, actual development is more of an art than a science. Attempts to capture or quantify the "art" aspect of development have led to the development of several models or tools, and the chief problem faced by all of these models is how to define the amount of work, or specifically, how much code is involved. Reliance on an unstandardized approach to counting the number of lines of code is a flawed approach to software cost estimation. For Oracle database developers, this problem is compounded by the lack of quantifiable metrics that capture the cost of database development.

Introduction

Much literature has been devoted to the topic of software cost estimation. Along with the literature, many models have been developed whose purpose is to quantify the amount of effort involved. The estimation models typically produce a time-based result, that is, how many person-months are required to produce how many thousands of lines of code. As an alternative approach, some models (primarily object-oriented projects) base their estimation on the number of function points. For the most part, software cost estimation applies to languages, not technologies. One technology of prime importance to many applications is that of the Oracle database management system.

Think of all the code behind every CREATE statement. Designing a good relational model takes as much forethought as does designing a good application, where "good" implies having followed best practices and established methodologies such as the Software Development Lifecycle. Assuming one can equate traditional software cost estimation with database development cost estimation, then what tools or methods can the database developer use to capture the cost of development? Once the tools and methodologies are understood, then how would a designer or developer go about measuring database lines of code (DLOC)?

To begin our investigation, let's take a look at two types of estimation models.

Reliance on an unstandardized approach to counting the number of lines of code is a flawed approach to software cost estimation.

The KLOC Models

KLOC models are based on the number of lines of code, where the number is divided by 1000. The scaling is quite useful when considering projects with millions of lines of code (LOC). "KLOC" is typically used to represent the number of thousands of LOC. The generic form of a KLOC model is represented by

$$E = A + B(e_v)^C$$

where

E = the effort in person-months
A, B and C = empirically derive constants
e_v = an estimation variable (KLOC or function points)

Some representative KLOC models are shown in Table 1.

Model Name	E = A + B(ev)C Format
Walston-Felix	E = 5.2 x (KLOC)0.91
Bailey-Basili	E = 5.5 + 0.73 x (KLOC)1.16
Boehm simple	E = 3.2 x (KLOC)1.05
Doty for KLOC > 9	E = 5.288 x (KLOC)1.047

Table 1: Representative KLOC Models

ORAtips

CIO Corner

Some models are more applicable over a range of KLOC values than others. The effort formulas are based on taking an "x" (the KLOC) and measuring or observing the "y" (the effort in person-months). To develop a predictive model, simple linear regression was used. In simple linear regression (SLR), the computed model (or line) will always have a y-intercept (at X=0, the regression line intercepts the Y-axis). The interpretation of an SLR model is valid over the observed range of X (the independent variable). The Bailey-Basili model reflects an effort estimate of more than five person-months for zero KLOC, which is a nonsensical conclusion.

In comparison, the IBM Federal Systems Division (IBM-FSD) study, resulting in the Walston-Felix model, spanned projects ranging from 4 to 467 KLOC. Although a value of zero for KLOC accurately reflects zero corresponding effort, the model's validity should be judged over its domain or set of observed values. The scatter plot of delivered code versus total effort (in man-months) is shown in Figure 1. The zero KLOC "equaling" five months of effort interpretation shows the danger of extrapolating beyond observed values.

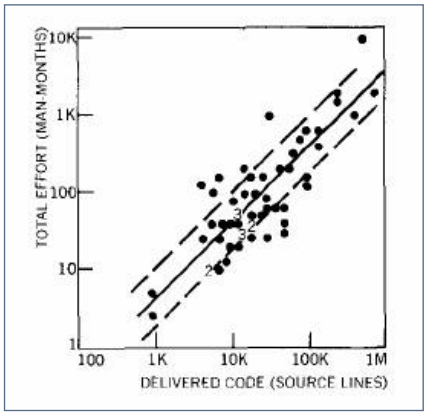

Figure 1: Walston-Felix Data Points

The Function Point Models

A function point (FP) metric falls under function-oriented metrics.

Model Name	E = A + B(ev) Format
Albrecht and Gaffney	E = -91.4 + 0.355 FP
Kemerer	E = -37 + 0.96 FP
Small project regression	E = -12.88 + 0.405 FP

Table 2: Representative FP Models

This type of metric is based on the functionality delivered by the application as a normalization value. A function point (FP) metric can be used for several purposes, of which estimating cost or effort required to design, code, and test the software is of interest. The following factors are used to develop an empirical relationship or model.

- Number of external inputs
- Number of external outputs
- Number of external queries
- Number of internal logical files
- Number of external interface files

It is interesting to note that once all the effort has gone into calculating the number of FPs, some models convert the end result into KLOC. The validity of how one language compares to another (for example, what takes 77 lines in COBOL can be done in 53 with C++) qualifies as its own separate area of research. Some representative FP models are shown in Table 2.

One characteristic all models possess is the ability for the user to modify or tweak the fixed and variable factors. Adjusting, and even adding factors, is a feature of a widely popular model known as COCOMO II.

The Origin of COCOMO II

COCOMO stands for Constructive Cost Model. The current version, known as COCOMO II, as did its predecessor COCOMO, incorporates scale factors and effort multipliers to adjust or factor how long it will take to code a project of a certain size.

COCOMO was developed by Barry Boehm, currently of the Center for Software Engineering at the University of Southern California. Based on empirical evidence from software projects at TRW, the COCOMO model was published in 1981. The model used input from 63 data points, where a data point represents a software project ranging from 2,000 to 100,000 LOC. Dr. Boehm and others reworked the model and published COCOMO II. Some literature refers to this as COCOMO II.2000 to delineate the difference between the 1981 and 2000 versions.

The COCOMO II Model

Based upon a data set of 161 points (i.e., software projects), COCOMO II demonstrates an accuracy of 30 percent of the actuals 75 percent of time. COCOMO II is also representative of the KLOC models previously mentioned. The main result of interest is the amount of effort as measured in person-months (PM).

COCOMO II incorporates 5 scale factors (SF) and 17 effort multipliers (EM) in the following equations. The "NS" subscript is used to denote a nominal schedule.

$$PMNS = A \times (SIZE)E \times$$

where $E = B + 0.01 \times \sum_{i=1}^{n} SFi$

$$\prod_{i=1}^{n} EMi$$

The calibrated values for A and B are 2.94 and 0.91, respectively. As discussed, SIZE is based on KLOC.

The following tables illustrate the meaning of each "EM" or effort multiplier and "SF" scale factor item. The complete scope, meaning, and derivation of each item is beyond the scope of this article, but the interested reader may reference Boehm, et al's Software Cost Estimation with COCOMO II for more detail and explanation. A basic interpretation follows Tables 3 and 4.

Generic Example of Estimating Effort

For an overall estimate at the project level, all 17 effort multipliers are considered. For a component, the SCED effort multiplier is omitted. As an example, if an average project consists of 100,000 LOC (KLOC=100), each EMi is 1.00, and the product of all 17 is obviously 1.00. If E=1.15 is used (as in Boehm's example in his text, based on "an average large project"), then the estimated effort in person-months can be calculated as

PM = 2.94 x (100)1.15 = 586.61

COCOMO II SCALE FACTORS								
SFi	Driver	Description	Very Low	Low	Nominal	High	Very High	Extra High
1	PREC	Precedentedness	6.20	4.96	3.72	2.48	1.24	
2	FLEX	Development flexibility	5.07	4.05	3.04	2.03	1.01	
3	RESL	Architecture/risk resolution	7.07	5.65	4.24	2.83	1.41	
4	TEAM	Team cohesion	5.48	4.38	3.29	2.19	1.10	
5	PMAT	Process Maturity	7.80	6.24	4.68	3.12	1.56	

Table 3: COCOMO II Scale Factors

COCOMO II EFFORT MULTIPLIERS								
			Rating					
	Cost Driver	Description	Very Low	Low	Nominal	High	Very High	Extra High
EMi	Product							
1	RELY	Required software reliability	0.82	0.92	1.00	1.10	1.26	
2	DATA	Database size		0.90	1.00	1.14	1.28	
3	CPLX	Product complexity	0.73	0.87	1.00	1.17	1.34	1.66
4	RUSE	Required reusability		0.95	1.00	1.07	1.15	1.49
5	DOCU	Documentation	0.81	0.91	1.00	1.11	1.23	
	Platform							
6	TIME	Execution time constraint			1.00	1.11	1.29	1.67
7	STOR	Main storage constraint			1.00	1.05	1.17	1.57
8	PVOL	Platform volatility		0.87	1.00	1.15	1.30	
	Personnel							
9	ACAP	Analyst capability	1.42	1.19	1.00	0.85	0.71	
10	PCAP	Programmer capability	1.34	1.15	1.00	0.88	0.76	
11	PCON	Personnel continuity	1.29	1.12	1.00	0.90	0.81	
12	APEX	Applications experience	1.22	1.10	1.00	0.88	0.81	
13	PLEX	Platform experience	1.19	1.09	1.00	0.91	0.85	
14	LTEX	Language & tool experience	1.20	1.09	1.00	0.91	0.84	
	Project							
15	TOOL	Software tools	1.17	1.09	1.00	0.90	0.78	
16	SITE	Multisite development	1.22	1.09	1.00	0.93	0.86	0.78
17	SCED	Development schedule	1.43	1.14	1.00	1.00	1.00	

Table 4: COCOMO II Effort Multipliers

In other words, a 1-person team would need just over 48 years to complete this project, and a 10-person team would need almost 5 years. Looked at another way, if the average salary per person was $50,000 per year, the project would cost 2.4 million dollars.

As another illustrative example of this model, consider Windows XP. Various sources state that XP contains 40 million lines of code. One proposal behind Microsoft's upcoming release of Longhorn was that the Windows operating system code would be re-written so as to remove the spaghetti code nature of this code base. Using the same parameters as the previous example, a 40,000 KLOC project would require

PM = 2.94 x (40000) 1.15 = 576,389 person-months

Using 1,000 developers, the timeline spans just over 48 years. It should be safe to assume that the next release of the Windows operating system is not going to be a complete rewrite of the existing code.

The Metric of Source Lines of Code (SLOC)

The driving factor in calculating a nominal schedule is the number (in thousands) of lines of code. What, exactly, constitutes a line of code, and further, how should lines of code be counted in the first place? One developer may code a block of instruction in 20 lines, while another – and just as skilled developer – may decide to code in only 12. Despite's COCOMO II's consideration of more than 20 factors, its dependence on counting the number of lines of code makes it an inherently flawed tool or model.

COCOMO II addresses the SLOC determination problem by considering the Software Engineering Institute's "Definition Checklist for Source Statement Counts." COCOMO II (and SEI) recognize the problems in determining SLOC and have taken steps to address this issue.

```
Public class Welcome
{
  // main method begins execution of Java application
  Public static void main( String args[] )
  {
   System.out.print( "Welcome to " );
   System.out.print( "Java programming!" );

  } // end method main

} // end class Welcome
```

Figure 2: Outputting a Simple Message

Consider how code is generally presented in introductory programming books. A fairly standard first example has the new programmer outputting a simple message. Figure 2 shows a typical first program (a variation of the "hello world" screen output).

COCOMO II does not count the commented or empty lines, but what about the lines with a single left or right brace ("{" and "}")? Does this class contain four or eight countable lines of code? Or can it be reduced to two lines?

Public class Welcome{Public static void main(String args[]) {System.out.println("Welcome to \nJava programming!");}

The point of this may seem simple, but its implications are far reaching because of programming constructs or options available in many languages. The simple IF-THEN-ELSE statement is typically seen as

```
IF (some_condition) THEN
 Do_this;
ELSE
 Do_that:
END;
```

Just as legitimate is the conditional operator construct seen in Java, JavaScript, and other languages:

(some_condition ? do_this : do_that);

Are there five countable lines of code, or just one? Turning our attention back to database design and development, let's look at the equivalent lines of code we can use to estimate the cost of "coding" a database.

Counting the Lines that Count in a Database

Some may argue that data definition language (DDL) is not real code because all a DDL does is define structure. If that's the case, then why do setters and getters (and even class definitions) in Java count as legitimate lines of code? After all, a line like

SomeCode doProgram = new SomeCode();

does nothing more than set up "doProgram" for references to doProgram.SomeMethod() down the road. To use a database example, let's consider the EMPLOYEES table in Oracle's HR sample schema. The DDL for this table is shown in Figure 3.

ORAtips

CIO Corner

```
REM ***********************************************************************
REM Create the EMPLOYEES table to hold the employee personnel
REM information for the company.
REM HR.EMPLOYEES has a self referencing foreign key to this table.

Prompt ****** Creating EMPLOYEES table ....

CREATE TABLE employees
  ( employee_id  NUMBER(6)
  , first_name   VARCHAR2(20)
  , last_name    VARCHAR2(25)
             CONSTRAINT   emp_last_name_nn NOT NULL
  , email     VARCHAR2(25)
            CONSTRAINT   emp_email_nn NOT NULL
  , phone_number  VARCHAR2(20)
  , hire_date   DATE
            CONSTRAINT   emp_hire_date_nn NOT NULL
  , job_id     VARCHAR2(10)
            CONSTRAINT   emp_job_nn NOT NULL
  , salary     NUMBER(8,2)
  , commission_pct NUMBER(2,2)
  , manager_id   NUMBER(6)
  , department_id NUMBER(4)
  , CONSTRAINT   emp_salary_min
         CHECK (salary > 0)
  , CONSTRAINT   emp_email_uk
         UNIQUE (email)
  ) ;

CREATE UNIQUE INDEX emp_emp_id_pk
ON employees (employee_id) ;


ALTER TABLE employees
ADD ( CONSTRAINT   emp_emp_id_pk
         PRIMARY KEY (employee_id)
  , CONSTRAINT   emp_dept_fk
         FOREIGN KEY (department_id)
         REFERENCES departments
  , CONSTRAINT   emp_job_fk
         FOREIGN KEY (job_id)
         REFERENCES jobs (job_id)
  , CONSTRAINT   emp_manager_fk
         FOREIGN KEY (manager_id)
         REFERENCES employees
  ) ;

ALTER TABLE departments
ADD ( CONSTRAINT dept_mgr_fk
                      FOREIGN KEY (manager_id)
                      REFERENCES employees (employee_id)
  ) ;
```

Figure 3: DDL for the HR.EMPLOYEES Table

How many lines of code did it take to create this table? Not counting the remark lines, and moving the dangling closing parens up one line (and by convention, blank lines do not count), there are 39 lines. The DDL statements and clauses look just as complex as the way code does in many other languages. And as another consideration, what if a schema has default data loaded into it ahead of time? Those INSERT statements should count too.

Other countable lines of code for a schema include lines used for creating packages, package bodies, procedures, function, triggers, views, indexes, sequences, and jobs. In other words, the lines used to create every object in a schema count.

Yet another area of consideration includes GUI objects. For Oracle, we can include Forms, Reports, and anything related to enhancing Application Server. A moderately sized form can have thousands of lines of code buried within elements such as data blocks (all the triggers behind block elements), list of values, record groups, and stored procedures and functions.

Putting the DLOC to Use

So with all these thousands, if not millions, of lines of code being "coded" into a database, doesn't counting the lines themselves add some additional burden to the effort? Yes it does, but this effort isn't any different than what "real" languages face. Fortunately for database developers and administrators, there is a built-in counting mechanism in the form of "COUNT(whatever)" in SQL.

There are two approaches to counting database lines of code (DLOC), and they can be divided into before and after the fact. The "before the fact" camp has it a bit easier if all DDL and DML statements are contained in scripts. Coding conventions help standardize what does and does not count (e.g., all CREATE SEQUENCE statements count as two lines). This approach parallels traditional software engineering cost estimation efforts.

For those in the "after the fact" camp wanting to obtain an estimate of their DLOC, well, start counting what's already in the database. Once a DLOC number is obtained, some extrapolation can be used to provide an estimate for similarly sized development projects in the future. Interestingly enough, this approach also parallels that of traditional software engineering cost estimation. How can that be? How do you think the KLOC and FP models were developed in the first place? By counting the lines of code in completed projects and then using linear regression, that's how.

In either case, once the DLOC number has been obtained, COCOMO II can be used to determine effort estimates. Once the time is computed, cost estimates can be calculated, and with the cost estimate in hand, you'll know how much that database in the window costs.

Steve Callan – Steve is an Oracle DBA and developer. His Oracle experience includes several versions of the RDBMS, Forms & Reports, and Application Server. In addition to working with Oracle, Steve also spends time on researching other database systems such as SQL Server and DB2 and would someday like to start his own software company. Steve may be contacted at Steve.Callan@ERPtips.com.

References

Boehm, Barry W. et al. Software Cost Estimation with COCOMO II. Upper Saddle River, NJ: Prentice Hall PTR, 2000.

"COCOMO 81." Center for Software Engineering, University of Southern California. 2 Oct. 2005. <http://sunset.usc.edu/research/COCOMOII>

Montgomery, Stephen. Building Object-Oriented Software. New York: McGraw-Hill, 1998.

Pressman, Roger S. Software Engineering: A Practitioner's Approach. Boston, MA: McGraw-Hill, 2005.

Watson, C.E., and Felix, C.P. "A Method of Programming Measurement and Estimation." IBM Systems Journal 16.1 (1977): 54pp. 2 Oct. 2005

<http://domino.research.ibm.com/tchjr/journalindex.nsf/0/aa74f16b2732c9ee85256bfa00685add?OpenDocument>

January 2006 Volume II Issue 1

ORAtips Journal