Estimating and Managing Project Scope for Maintenance and Reuse Projects

William Roetzheim
Cost Xpert Group, Inc.

Estimating project scope is considered by many to be the most difficult part of software estimation. Parametric models have been shown to give accurate estimates of cost and duration given accurate inputs of the project scope, but how do you input scope early in the life cycle when the requirements are still vaguely understood? How can scope be estimated, quantified, and documented in a manner that is understandable to management, end users, and estimating tools? This article focuses on scope estimates for maintenance and reuse work, including bug fixes (corrective maintenance); modifications to support changes in the operating system, database management system, compiler, or other aspect of the operating environment (adaptive maintenance); and modifications of existing functionality to improve that functionality (perfective maintenance). Reuse includes any case where you are modifying an existing code base to support enhanced functionality, and includes cases where an existing application is translated to a new language. The effort estimates cover code fixes and enhancements, regression and other testing of those fixes, updates to documentation, and management of those efforts. It does not include requirement/usability efforts or deployment efforts.

At a high level, maintenance projects consist of three types of work:

1. Maintaining an existing, functioning application.
2. Modifying existing code to support changing requirements.
3. Adding new functionality to an existing application.

A team doing a new build for an existing application would only be concerned with item Nos. 2 and 3. A team keeping an existing code base functioning would only do item No. 1, and possibly item No. 2 depending on how new builds are handled. A project manager may be responsible for both areas and might need to estimate the effort required for all three. This article will deal with each individually.

Maintaining Existing Code

Maintenance as we are defining it consists of three types of activities [1]:

- **Corrective Maintenance.** Fixing bugs in the code and documentation. Bugs are areas where the code does not operate in accordance with the requirements used when it was built.
- **Adaptive Maintenance.** Modifying the application to continue functioning after installation of an upgrade to the underlying virtual machine (database management system, operating system, etc.).
- **Perfective Maintenance.** Correcting serious flaws in the way it achieves requirements (e.g., performance problems).

Maintenance effort is a function of the development effort spent on the original project. (Figure 1 shows an example of a commercial tool reuse parameter screen). The larger the original project in terms of effort, the more staff must be assigned to maintain the application. A second factor is Annual Change Traffic (ACT), or the percent of the code base that will be touched each year as a result of maintenance work. Numbers for ACT between 3 percent and 20 percent are reasonable, with 12 percent to 15 percent being fairly typical. The Nominal Annual Maintenance effort while the application is in a maintenance steady state will equal

ACT x PM

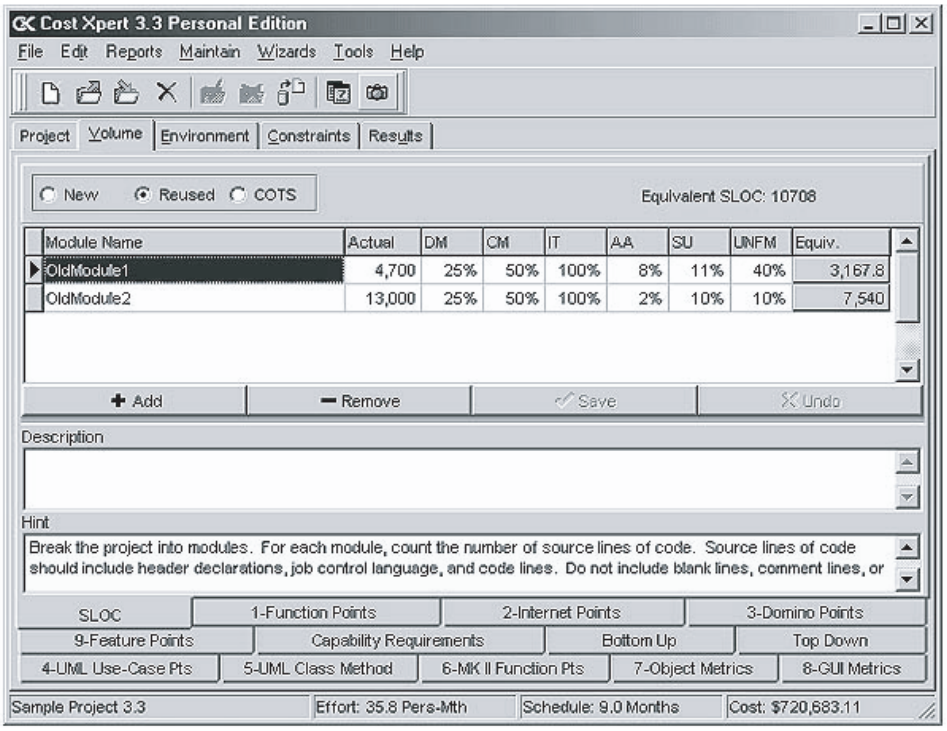
where,

PM is the original person months of development effort.

Maintenance steady state is typically achieved in year four following software delivery. Between software delivery and steady state, the maintenance effort follows a Rayleigh[1] curve starting with 1.5 x ACT x PM in year one and dropping down to the steady state value. Software maintained beyond nine years typically sees maintenance costs begin to climb again. This is due to a combination of increasingly fragile code and an increasing distance between the technology used for development and the current state of the art.

If you know the effort spent on the original development, the above equations may be used as shown. If you do not know the original development effort, you must first estimate that effort, normally using a commercial estimation tool. This will require that you count the physical lines of code (or function points), and then either make educated guesses at values for environmental variables (team capability and so

Figure 1: *Example of Commercial Tool Reuse Parameter Screen*



on) or use the tool's default values.

In some cases, strategic considerations will require that programmers with knowledge of the product be kept available and current to facilitate future planned or potential modifications of the code. These developers would then be available to make required short fixes to the code as well. In this case, factors outside the scope of this article would dictate the number of developers that must be kept available and on the maintenance staff.

Modifying Existing Code

The basis of code modification is very simple: Code already exists that may be utilized in any given project. You begin by actually counting the values measured in the existing application(s). For example, if using lines of code (LOC), employ a code counting utility to physically count the lines of code in the existing program modules (or use the values from your configuration management system with an adjustment to remove the impact of blank lines and comments). If using Function Points, count the existing reports, screens, tables, and so on.

Calculating the Equivalent Volume

Our goal is to convert from the known value for the volume of reusable code to an equivalent volume of new code. Think about it this way:

- If we have 100 function points worth of reusable code but the reusable code is worth nothing to us, then no effort will be saved; the equivalent amount of new code is 100 function points.
- If we have 100 function points worth of reusable code and we can reuse it without any changes, retesting, or integration whatsoever, then using the code is a *freebie* from a developmental perspective. The equivalent amount of new code is 0 function points.
- If we have 100 function points worth of reusable code and this saves us half the effort relative to new code, then the equivalent amount of new code is 50 function points.

We convert from reused volume values to equivalent new volume values by looking at six factors: Percent Design Modification (PDM), Percent Code Modification (PCM), Percent Integration and Testing (PI&T), Assessment and Assimilation (AA), Software Understanding (SU), and Unfamiliarity (UNFM) with software.

Percent Design Modification

The PDM measures how much design effort the reused code will require. Basically, a low percent value indicates high code reuse, whereas a high percent value indicates low code reuse and increases the requirement to develop new code as follows:

- A value of 0 percent says that the reused code is perfectly designed for the new application and no design time will be required at all.
- A value of 100 percent says that the design is totally wrong and the existing design will not save any time at all.
- A value of 50 percent says that the design will require some changes and that the effort involved in making these changes is 50 percent of the effort of doing the design from scratch.

For typical software reuse, the PDM will vary from 10 percent to 25 percent.

Percent Code Modification

The PCM measures how much we will need to change the physical source code as follows:

- A value of 0 percent says that the reused code is perfect for the new application, and the source code can be used without change.
- If the reused code was developed in a different language and you need to port the code to your current language, the value would be 100 percent (ignoring any potential automated translation using automatic translation tools).
- Numbers in between imply varying amounts of code reuse.

The PCM should always be at or higher than the PDM. As a rule of thumb, we have found the PCM is often twice the PDM.

Percent Integration and Testing

The PI&T measures how much integration and testing effort the reused code will require as follows:

- A value of 0 percent would mean that you do not anticipate any integration or integration test effort at all.
- A value of 100 percent says that you plan to spend just as much time integrating and testing the code that you would if it was developed new as part of this project.
- Numbers in between simply refer to differing degrees of integration and testing effort relative to new development.

The PI&T should always be at or higher than the PCM. It is recommended that you set the PI&T to at least twice the PCM.

It is not unusual for this factor to be 100 percent, especially for mission-critical systems where the risk of failure is significant. For commercial off-the-shelf components (purchased libraries) where the PDM and PCM are often zero, it is not unusual to see a number of 50 percent here to allow for the integration effort and time spent testing the application with the commercial component.

Finally, after measuring your existing code's volume and estimating your PDM, PCM, and PI&T, calculate the Adaptation Adjustment Factor (AAF) where AAF is the:

$$AAF = .4DM + .3CM + .3\ I\&T \qquad (1)$$

where,

this equation comes from [2][2].

Suppose that we need to implement a new e-commerce system consisting of 15,000 source lines of code (SLOC). Let us ignore environmental adjustments for the moment.

Assessment and Assimilation

AA indicates how much time and effort will be involved in testing, evaluating, and documenting the screens and other parts of the program to see what can be reused. Values range from 0 percent to 8 percent.

Software Understanding

SU estimates how difficult it will be to understand the code once you are modifying it, and how conducive the software is to being understood. Is the code well structured? Is there good correlation between the program and application? Is the code well commented? The range of possible values is a numeric entry between 10 percent and 50 percent, default 30 percent.

Unfamiliarity With Software

UNFM with software indicates how much your team has worked with this reusable code before. Is this their first exposure to it, or is it very familiar? The range of possible values is between 0 percent and 100 percent, default 40 percent.

Using the Six Factors

Three of the factors – AA, SU, and UNFM with software – add a form of tax to software reuse, compensating for the overhead effort associated with reusing code. For projects where the amount of reuse is small (AAF is less than or equal to 50 percent), the following formula applies with adjustments per the above factors:

$$ESLOC = ASLOC \times [AA + AAF(1 + 2 \times SU \times UNFM)]$$

where,

ESLOC is the equivalent SLOC and

ASLOC is the actual SLOC.

Before discussing how this equation is used to determine the reuse effort, let us take a step back to discuss a simple equation to determine effort. If you are aware of the number of thousand SLOC (KSLOC) your developers must write, and you know the effort required per KSLOC, then you could multiply these two numbers together to arrive at the person months of effort required for your project.

$$\text{Effort} = \text{Productivity} \times \text{KSLOC}$$

where,

KSLOC represents a measure of program scope.

Table 1 shows some common values that Cost Xpert researchers have found for these linear productivity factors. The COCOMO II value comes from research by Barry Boehm at the University of Southern California (USC). The values for embedded, e-commerce, and Web development come from Cost Xpert research working with a variety of organizations, including IBM and Marotz.

You also must consider that researchers have found that productivity varies with project size. In fact, large projects are significantly less productive than small projects. The probable causes are a combination of increased coordination and communication time, plus more rework required due to misunderstandings.

This productivity decrease with increasing project size is factored in by raising the number of KSLOC/thousand software LOC to a power greater than 1.0. This exponential factor then penalizes large projects for decreased efficiency. Table 2 shows some typical size penalty factors for various project types. Again, the COCOMO II value comes from work by Barry Boehm at USC; values for embedded, e-commerce, and Web come from work by Cost Xpert Group and our customers. Note that because the size factor is an exponential factor, rather than linear, it does not change with project size, but rather changes in impact on the end result with project size.

As seen in the tables, the productivity and penalty constants vary by project and organization. Let us take an example involving 15,000 reused SLOC. Using the following formula, as well as the productivity and size penalty factors for e-commerce development, the predicted effort will be:

Project Type	Linear Productivity Factor (PM/KSLOC)
COCOMO II Default	3.13
Embedded Development	3.60
E-Commerce Development	3.08
Web Development	2.51

Table 1: *Common Values for Linear Productivity Factor*

$$\begin{aligned}\text{Effort} &= \text{Productivity} \times \text{KSLOC}^{\text{Penalty}} \\ &= 3.08 \times 15^{1.030} \\ &= 3.08 \times 16.27 \\ &= 50 \text{ Person Months}\end{aligned}$$

Suppose we found that we could get by with 10 percent design modifications, 20 percent code modifications, and 40 percent integration and test effort. AAF would then be calculated as:

$$\text{AAF} = (0.4 \times 0.1) + (0.3 \times 0.2) + (0.3 \times 0.4) = 0.22$$

Because AAF is less than or equal to 50 percent we can use the formula just presented. Now, suppose that AA was 4 percent, SU was 30 percent, and UNFM was 40 percent. The equivalent source lines of code (ESLOC) would now be:

$$\text{ESLOC} = 15{,}000 \left[0.04 + 0.22(1 + 2 \times 0.3 \times 0.4)\right] = 4{,}692$$

Using our earlier assumptions, the effort required to build this software would be:

$$\begin{aligned}\text{Effort} &= \text{Productivity} \times \text{ESLOC}^{\text{Penalty}} \\ &= 3.08 \times 4.692^{1.030} \\ &= 3.08 \times 4.915 \\ &= 15.14 \text{ Person Months}\end{aligned}$$

The formula when reuse is low and AAF is less than 50 percent changes. The formula in this situation is:

$$\text{ESLOC} = \text{ASLOC} \times \left[\text{AA} + \text{AAF} + (\text{SU} \times \text{UNFM})\right]$$

Let us work through our same example of 15,000 lines of reused code, but let us now suppose that the design modification is 50 percent, the code modification 100 percent, the integration and test are 100 percent, and the correct values for AA, SU, and UNFM are 8 percent, 50 percent, and 100 percent respectively.

AAF is now calculated as:

$$\text{AAF} = (0.4 \times 0.5) + (0.3 \times 1.0) + (0.3 \times 1.0) = 0.8$$

Because AAF is over 50 percent, we use the formula as follows:

$$\begin{aligned}\text{ESLOC} &= 15{,}000 \times [0.08 + 0.8 + 0.5 \times 1.0] \\ &= 15{,}000 \times 1.38 \\ &= 20{,}700\end{aligned}$$

Effort is now calculated as:

$$\begin{aligned}\text{Effort} &= \text{Productivity} \times \text{ESLOC}^{\text{Penalty}} \\ &= 3.08 \times 20.7^{1.030} \\ &= 3.08 \times 22.67 \\ &= 69.82 \text{ Person Months}\end{aligned}$$

In this case, as seen by comparing the person months in the first example of this article with the person months in the final example, reusing those 15,000 LOC actually takes 19.82 person months more effort than writing the same code from scratch! In fact, this phenomenon is even more pronounced than shown in the preceding example. If you need 15,000 lines of new functionality, you will seldom find a reusable block of code that exactly matches the functionality you are looking for. More often, the reused code will be significantly larger than the new code because it will do many functions that you are not interested in.

Perhaps you will be reusing a piece of code that is 25,000 LOC in size, all to get at those 15,000 lines of code worth of functionality that you care about. Well, the entire 25,000 LOC will typically need to be assessed, understood, and tested to some degree. The end result is that in general, you will find that somewhere between 15 percent and 30 percent design change is the crossing point beyond which you are typically better off rewriting the code from scratch. The correct value in

Table 2: *Penalty Factors for Various Project Types*

Project Type	Exponential Size Penalty Factor
COCOMO II Default	1.072
Embedded Development	1.111
E-Commerce Development	1.030
Web Development	1.030

Coming Events

December 2-3
The 6th IEEE Workshop on Mobile Computing Systems and Applications
Lake Windermere, United Kingdom
http://wmcsa2004.lancs.ac.uk/

December 2-4
International Conference on Intelligent Technologies (InTech) '04
Houston, TX
http://csc.csudh.edu/intech04/index.htm

December 4-8
IEEE/ACM International Symposium on Microarchitecture
Portland, OR
www.microarch.org/micro37/

December 6-9
Interservice Industry Training, Simulation, and Education Conference
Orlando, FL
www.iitsec.org

January 6-9, 2005
Internet, Processing, Systems, and Interdisciplinary Research (IPSI) 2005
Oahu, HI
www.internetconferences.net/industrie/hawaii2005.html

January 9-12
International Conference on Intelligent User Interfaces
San Diego, CA
www.iuiconf.org/

January 31-February 3
16th Annual Government Technology Conference
Austin, TX
www.govtech.net/gtc/?pg=conference&confid=182

April 18-21
2005 Systems and Software Technology Conference
Salt Lake City, UT
www.stc-online.org

this range will depend largely on how well matched the reused code is to your requirements and the quality of that code and documentation.

If you are doing an ongoing series of maintenance builds with a large, relatively stable application, there are some tricks to simplify your planning. Create a spreadsheet containing all of the modules and for each module, the LOC in that module. Set percent design mode, code mode, and so on to zero for each module in the spreadsheet. It is also useful in the spreadsheet to include an area where you identify the dependent relationships between modules (this can sometimes be done using a tool like Microsoft Project, where you treat each module as a task in the dependency diagram). Save this as your master template for planning a new build.

When you are planning a build, analyze each requirement for change to identify the modules that must be modified and fill in the appropriate value for design modification, code modification, etc. Then, look at the modules that are dependent on these modules and put in an appropriate value for IP&T for those dependent modules. You can then quickly calculate the resultant equivalent scope and use this to calculate a schedule and the effort required. For the next build, go back to the template you started with and repeat the process. Some commercial estimating tools support this approach, as well.

Adding New Functionality

Finally, when preparing a new software build there are normally some areas where completely new functionality is added to the system. This functionality is defined and estimated as new development using the standard approaches suitable for estimating new software development.

Conclusions

This article presents quantitative approaches to estimating scope and effort for maintenance, enhancement, and reuse projects. Following these techniques will produce reasonable and justifiable estimates and budgets for maintenance projects, and help with build release planning.◆

References

1. Cost Xpert Group. Cost Xpert Vers. 3.3 User Manual. Rancho San Diego, CA: Cost Xpert Group, Inc., 19 Nov. 2003.
2. Boehm, Barry W., et al. Software Cost Estimation With COCOMO II. 1st ed. Prentice Hall PTR, 15 Jan. 2000.

Notes

1. A Rayleigh curve yields a good approximation to the actual labor curves on software projects.
2. The work in this paper is heavily dependent on work by Barry W. Boehm and others, as documented in this latest book [2].

About the Author

William Roetzheim has 25 years experience in the software industry and is the author of 15 software related books and over 100 technical articles. He is the founder of the Cost Xpert Group, Inc., a Jamul-based organization specializing in software cost estimation tools, training, processes, and consulting.

Cost Xpert Group
2990 Jamacha RD STE 250
Rancho San Diego, CA 92019
E-mail: william@costxpert.com

Web Sites

Reusable Software Research Group

www.cse.ohio-state.edu/rsrg

The Reusable Software Research Group (RSRG) began at Ohio State University in the mid 1980's and is currently active at Ohio State, at Clemson University, and at Virginia Tech. The RSRG deals with the disciplined engineering of component-based software systems and the software components (aka reusable software components) from which they can be built. The RSRG's work involves a framework for component-based software engineering, a research language, and a component-design discipline called RESOLVE. The group is concerned with creating software that is at once reusable, efficient, verifiable, and comprehensible.

RESOLVE

http://people.cs.vt.edu/~edwards/resolve

RESOLVE is a comprehensive and robust framework, discipline, and language for the construction of highly reusable component-based software. The RESOLVE Web site contains links to online readings and an annotated bibliography for those interested in this work.