Estimating Software Reuse Equivalent Function Points

Author

William H. Roetzheim
13518 Jamul Drive, Jamul, CA 91935
William@costXpert.com
619.917.4917 voice
530.734.2741 fax

Abstract

A significant part of estimation deals with software reuse/maintenance projects. This includes maintenance build projects and making decisions about reusing existing code versus rewriting new code. This talk presents a quantitative approach to estimating the equivalent function points for a software reuse/maintenance effort. In other words, we show the participants how to modify a function point count in a reuse/maintenance project to reflect the equivalent function point effort for new development. This process involves adapting the reuse work performed by the COCOMO II research team to the function point world. Case studies are used to illustrate the approach in making trade-off decisions.

Introduction

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 lifecycle 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 the case where you are modifying an existing code base to support enhanced functionality. 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 nor 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; and

3. Adding new functionality to an existing application.

A team doing a new build for an existing application would only be concerned with items 2 and 3. A team keeping an existing code base functioning would only do item 1, and possibly item 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. We'll deal with each individually.

Maintaining Existing Code

Maintenance as we're defining it consists of three types of activities:

- 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 (DBMS, operating system, etc.); and

- 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. The larger the original project in terms of effort, the more staff must be assigned to maintain the application. Various models to estimate maintenance are documented in the literature and embedded into commercial cost estimating tools, and these estimates are beyond the scope of this article.

Modifying Existing Code

The basis of code modification is very simple: code already exists that may be utilized in any given project. You begin with a complete function point count for the existing application.

Calculating the Equivalent Function Points

Our goal is to convert from the known value for the existing function points to an equivalent function point volume for the new code. In a simplified sense, 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, re-testing, 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 three factors: Percent Design Modification, Percent Code Modification, Percent Integration and Testing. After we have shown how this process works, we will describe how to adjust the numbers further based on three additional factors: Assessment and Assimilation, Software Understanding, and Unfamiliarity with Software.

- **Percent Design Modification** 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:

  ✓ A value of 0% 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% says that the design is totally wrong and the existing design won't save any time at all.

  ✓ A value of 50% says that the design will require some changes and that the effort involved in making these changes is 50% of the effort of doing the design from scratch.

  For typical software reuse, the Percent Design Modification will vary from 10% to 25%.

- **Percent Code Modification** measures how much we will need to change the physical source code:

  ✓ A value of 0% 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%.

  ✓ Numbers in between imply varying amounts of code reuse.

  The Percent Code Modification should always be at or higher than the Percent Design Modification. As a rule of thumb, we have found the Percent Code Modification is often twice the Percent Design Modification.

- **Percent Integration and Testing** measures how much integration and testing effort the reused code (the entire reused application) will require:

- ✓ A value of 0% would mean that you do not anticipate any integration or integration test effort at all.
- ✓ A value of 100% 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 Percent Integration and Test should always be at or higher than the Percent Code Modification. It is recommended that you set the Percent Integration and Test to at least twice the Percent Code Modification.

It is not unusual for this factor to be 100%, especially for mission critical systems where the risk of failure is significant. For commercial off-the-shelf components (purchased libraries) where the Percent Design Modification and Percent Code Modification are often zero, it is not unusual to see a number of 50% 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 Percent Design Modification (DM), Percent Code Modification (CM), and Percent Integration and Test (I&T), calculate the AAF (where AAF is the Adaptation Adjustment Factor):

$$AAF = .4DM + .3CM + .3I\&T$$

Equation 1: Adaptation Adjustment Factor

where this equation comes from Software Cost Estimation with Cocomo II (Barry Boehm, et. al).

Suppose that we identify a reusable application we can purchase with source code. We count the function points and find that the application has 1,000 function points. Let's assume that the correct value for design modification is 25%, the correct value for code modification is 50%, and the correct value for integration and test is 100%. What would be the equivalent function points?

This is computed as follows:

Equivalent Function Points = AAF x 1,000
= [(0.4 x 0.25) + (0.3 x 0.5) + (0.3 x 1.0)] x 1,000
= 0.55 x 1,000
= 5,500

There are three additional factors that need to be considered to complete the reuse picture: Assessment and Assimilation (AA), Software Understanding (SU), and Unfamiliarity (UNFM).

- 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% to 8%.
- 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? A numeric entry between 10% and 50%, default 30%.
- Unfamiliarity (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 and 100%, default 40%.

These three factors 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%), the following formula applies with adjustments per the above factors:

Equivalent Function Points = ReusedFunctionPoints x [AA + AAF (1+ 2 x SU x UNFM)]

Let's take our earlier example involving 1,000 reused Function Points. Suppose we found that we could get by with 10% design changes, 20% code changes, and 40% integration and test effort. AAF would then be calculated as:

AAF= (0.4 x 0.1) + (0.3 x 0.2) + (0.3 x 0.4) = 0.22

Because AAF is less than or equal to 50% we can use the formula just presented. Now, suppose that AA was 4%, SU was 30%, and UNFM was 40%.

The equivalent function points would now be:

EquivalentFunctionPoints = 1000 [0.04 + 0.22 (1+ 2 x 0.3 x 0.4)] = 261

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

EquivalentFunctionPoints = ReusedFunctionPoints x [AA + AAF + (SU x UNFM)]

Let's work through our same example of 1,000 reusable function points, but let's suppose that the design modification was 50%, the code modification 100%, the integration and test was 100%, and the correct values for AA, SU, and UNFM were 8%, 50%, and 100% respectively.

AAF is now calculated as:

AAF = (0.4 x 0.5) + (0.3 x 1.0) + (0.3 x 1.0) = 0.8

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

EquivalentFunctionPoints = 1000 x [0.08 + 0.8 + 0.5 x 1.0]
= 1,000 x 1.38
= 1,380

In this case, reusing this code actually requires more effort than writing the same code from scratch! In fact, this phenomenon is even more pronounced than shown in the example above. If you need 1,000 function points 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 1,500 function points in size, all to get at those 1,000 function points worth of functionality that you care about. Well, the entire 1,500 function points 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% and 30% design change is the crossing point beyond which you are better off rewriting the code from scratch. The correct value in 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 on-going 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 function points in that module. Set percent design mod, 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 (or 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 DM, CM, etc. Then, look at the modules that are dependent on these modules and put in an appropriate value for Integration and Test for those dependent modules. You can then quickly calculate the resultant equivalent function points and use this to calculate a schedule and effort required. The next build, you 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 function points.

Biography

William Roetzheim is the founder of the Cost Xpert Group and has been involved in software estimation, project management, and metrics for over twenty-five years. He is the author of 15 technical books, over 100 papers, and holds two patents pending. Mr. Roetzheim has an MBA and has completed course work for a Masters degree in Computer Science.