Estimating Software Size

6.1 Estimating Software Size

Size isn't everything in a software project but it does influence most things (*e.g.* resources, cost) so if we don't have an accurate prediction of size it is difficult to plan.

In this lecture we look at:

- Different approaches to size estimation
- Trying to tame the size problem via re-use

6.2 Some Approaches to Size Estimation

We shall discuss three contrasting approaches:

- Through expert consensus (Wideband-Delphi)
- From standard components (Component estimating)
- From a model of function (Function point)

The first of these relies on coordinating the views of experts; the second is based on knowledge of standard forms of design; and the third is driven by an analysis of software function.

6.2.1 Wideband-Delphi Estimating

The Wideband-Delphi method is a way of attempting to get experts in predicting software size to come to a consensus on their predictions - important because experts often disagree. The method is as follows:

1. Group of experts: $[E_1, \ldots, E_i, \ldots, E_n]$
2. Meet to discuss project
3. Each anonymously estimates size: $[X_1, \ldots, X_i, \ldots, X_n]$
4. Each E_i gets to see all the Xs (anonymously)
5. Stop if the estimates are sufficiently close together
6. Otherwise, back to step 2

6.2.2 Standard Component Estimating

The idea of standard component estimating is to guess the size of a software system as a function of the size estimates of its components. These component estimates are obtained from records of the size of previously constructed, similar systems. The method works like this:

- Gather historical data on key components
- Guess how many of each type you will need (M_i)
- Also guess largest (L_i) and smallest (S_i) extremes
- Final estimate (E_i) is a function of M_i, L_i and S_i
- For example, $E_i = (S_i + (4 * M_i) + L_i)/6$

6.2.3 Function Point Estimating

Software size often has a correlation with its functionality - greater functionality typically requiring a larger program. Function point estimating attempts to exploit this correlation by producing a size estimate based on a weighted count of common functions of software.

Commonly used basic functions are:

Inputs : Sets of data supplied by users or other programs

Outputs : Sets of data produced for users or other programs

Inquiries : Means for users to interrogate the system

Data files : Collections of records which the system modifies

Interfaces : Files/databases shared with other systems

A simple function point analysis produces a total estimate based on a count of the number of instances of each basic function, weighted to account for the different level of complexity associated with each in the type of project under consideration. For example:

Function	Count	Weight	Total
Inputs	8	4	32
Outputs	12	5	60
Inquiries	4	4	16
Data files	2	10	20
Interfaces	1	7	7
Total			135

This, of course, is quite a crude way of relating functionality and size so there are various more elaborate forms of function point analysis, for instance we might adjust function point total using "influence factors" to account for the effect of project features not directly related to basic functions of the code.

6.3 Re-Use

With ground-up programming the cost of development rises sharply as software size increases. It also continues to be true that software size tends to be, on the whole, larger in systems year on year. Maybe we can take advantage of earlier effort by re-using its products? Many artifacts of software design processes may be re-used, including:

- Code
- Designs and architectures.
- Documentation.
- Tests.
- anything else which is experience.

There also are a variety of motivations for doing so:

- Saves money
- Cumulative debugging
- Shorter development time
- Encourages modularity

Obtaining benefits from re-use on a large scale has, however, proved more difficult than one might imagine. There are a number of inhibitors, including these:

- Big components are the most tempting and most difficult to re-use
- Re-used components are older so may reach obsolescence sooner
- May not be able to re-use component and documentation
- May be hard to find the original designer if it goes wrong
- Hard to find the right thing
- Tempting to twist project to fit re-usable components
- It costs to design components specifically for re-use
- Need to consider re-use in the *previous* project