Task Identification and Effort Estimation

In order to plan most engineering activities, you must understood all of the work that needs to be done and how much effort will need to be expended to do it. Who can imagine a bridge construction or house construction project being approved without a budget or timetable? Similarly, who can imagine an automobile manufacturer deciding to offer a new model without fairly accurate estimates of the time and cost required to produce the cars?

Despite what many people seem to believe, this information is also important in most software engineering activities, despite the process being used. In a traditional, heavyweight process, this kind of information is needed for almost all aspects of planning and management and is primarily the responsibility of managers or lead technical people. In an agile, lightweight process (like Scrum), functional descriptions (sprintable stories) need to be divided into tasks and team members as a group have to commit to completing them during a sprint.

Thus, two central aspects of the management of a software engineering project are the identification of the tasks that must be completed to build a software product (the scope of the project), and estimating the amount of effort that will be required to complete those tasks. Along the way, it is common to estimate the size of the software product.

Task Identification and Organization

You should recall from our discussion of software processes that a task is an item of work and that a task can be either non-decomposable (in which case we call it an action) or decomposable (in which case we call it an activity or process).

It is usually fairly easy, at a high level of abstraction (which means a low level of detail) to identify the tasks that must be completed to build a software product. What is much more difficult is the identification of tasks at a level of detail that is useful for management purposes. Typically, you do not need to identify all of the actions; tasks can be decomposable but they should not be decomposable into too many different sub-tasks. The ideal level of detail is called a **work package**, which is somewhat tautologically defined as an amount of work that is small and specific enough that it can be estimated reliably.

A good way to both identify tasks at a useful level of abstraction is to use a **work breakdown structure** (**WBS**), which is a deliverable-oriented hierarchical decomposition of work done by the project team to achieve project goals or produce project deliverables. A WBS may be presented in three formats: it may be drawn as a tree, it may be drawn using a hierarchy diagram, or it may be written as a hierarchical list. A tree format has the main item of work at its root. Each level below the root shows a decomposition of the activity at that node into smaller parts. The leaves are work packages. A hierarchy diagram is a tree drawn in a more compact format: at some point child nodes are no longer splayed below their parent but instead are lined up (usually vertically). A hierarchical list uses indentation to show levels of abstraction: the activities or actions decomposing an activity are all listed indented from that activity to the same extent. Trees tend to be impractical for real problems because they use too much space; a hierarchy chart tends to be better for brainstorming (for example, on a whiteboard) and is generally easy to read; hierarchical lists are harder to read but easier to create in documents because they are texts, not images. We will use hierarchy diagrams.

The nodes in the WBS are items of work in the project. The root of a WBS is the project name, representing all the work to be done in the project. The first level is typically (though not necessarily) all the deliverables for the project, including documents, services, hardware, software, etc. Succeeding levels represent increasingly detailed specifications of work, where the leaf nodes are the work packages.

To illustrate, consider the WBS below for a project to develop the software for a home security system.

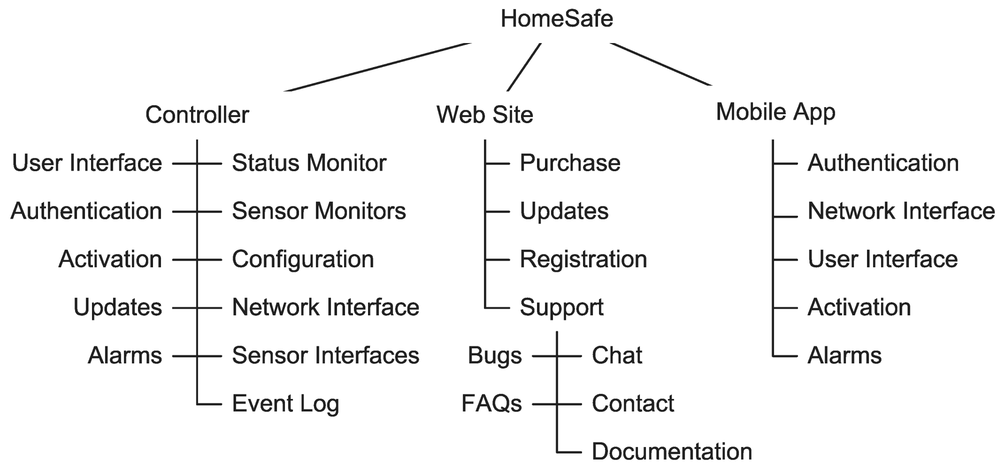


Figure 1: HomeSafe WBS

The nodes at the first level of this diagram are the three software components of the product: the Controller that runs the home system, the Web Site that sells and services the product, and a Mobile App that allows people to contact and control the home system remotely. The nodes at the third level are the main features comprising each component. The fourth level nodes (below Support) are the features of the Support portion of the Web Site. So far this WBS contains only deliverables. Some experts say that a WBS should contain only deliverables, so that this WBS is finished. But using this WBS for effort estimation would be difficult. Consequently other experts advocate adding more detail to make estimation easier. In this case, we could extend the WBS by adding nodes for design, coding, testing, inspection, deployment, and so forth. This would better display all the work that needs to be done in the project, and produce smaller work packages that are easier to estimate.

The WBS above decomposes deliverables and product features and functions to detail project work. Other decomposition strategies are possible, including the following.

- Project phases
- Organizational units
- Physical decomposition
- Geographical location

While mixing decomposition strategies is often frowned upon, decomposition strategies can be mixed in a WBS (even within a level). For example, in Figure 1, the first level is decomposed by deliverables (three programs), the second and third level by features and functions, and we proposed extending the WBS by project phases or activities (design, coding, etc.). In short, the best decomposition strategy is the one that leads to the best set of work packages.

If more information is to be associated with WBS nodes, a *WBS dictionary* can be written. Each node has an entry in the dictionary with additional information about it, such as its estimated effort, party responsible for the work, planned start and end date, and so forth.

There are several heuristics for making good work breakdown structures.

One hundred percent rule—The nodes descended from a parent represent 100% of the work of the parent, but no work outside the project. Obviously, if the WBS is to accurately represent the work in the project and be used for estimation and scheduling, it must not omit any work nor include activities that are not part of the work of the project.

Mutually exclusive siblings—No siblings in the WBS have overlapping work. Violation of this rule leads to inflated effort estimates and confusion about work assignments.

8/80 rule—Work packages (leaf nodes) require between 8 and 80 person-hours of effort. This heuristic helps to determine whether a node should be decomposed; work that can be done by one person in one day to two weeks is considered small enough to estimate reliably.

The best process for creating a WBS is to gather the project team and stakeholders in a meeting in which the WBS is created on a whiteboard, perhaps also using sticky notes. Participants should first make a list of items at level one silently on their own, and then the group can come to consensus on level one. Then each item in level one can be decomposed in the same way. Once deliverables (including product features and functions) are all listed in the WBS, stakeholders can leave the meeting. The lower levels consisting of development tasks and activities can then be filled in by the developers.

In traditional approaches, the entire project is planned as thoroughly as possible at the start of the project, progress is monitored closely, and adjustments are made to activities and plans as necessary as the project progresses. The planning process begins by setting the scope of the project and determining requirements.

Effort Estimation in Traditional Processes

There are several approaches to effort estimation in traditional processes.

Analogy—If the current project is similar to a past project, then the effort to complete it should be roughly the same as the effort required for the past project. *Ad hoc* adjustments can be made to compensate for any differences. This method works well when a project indeed closely ressembles some past project, but this is not often the case.

WBS to Effort—Given a complete work breakdown structure for a project, one can estimate the effort required for each work package and then sum these estimates to generate an effort estimate for the entire project. This can work well if the WBS accurately describes all the tasks needed to complete the project, and if the effort estimates for work packages are accurate. It is often hard to get everything right, however.

Size to Effort—Given an estimate of the size of the software product, along with known relationships between product size and effort, one can generate an estimate of project effort. A great deal of research has been devoted to this approach and we consider it in more detail next.

Software size can be measured quantitatively using two kinds of measures, functional and non-functional. **Functional measures of size** are denominated in units that are a direct reflection of the functionality being provided by the program (for example, the number of pages in a WWW application, the number of reports in a database application, the number of windows in a GUI application). Non-functional measures of size are denominated in units that are a direct reflection of the programs structure (for example, lines of code or number of classes).

The obvious shortcoming of all non-functional approaches is that while, at least in principle, it is easy to count lines of code or number of classes after the product has been developed, it is much more difficult to forecast them beforehand from requirements.

Fortunately, software engineers can learn from other disciplines. For example, you can construct a pretty good estimate of the time and cost of constructing a house based on the number of rooms of different kinds, such as kitchens, bathrooms, bedrooms and living rooms. In other words, you can measure the size based on the functionality requested by or provided to the user. You can then improve the estimate by adding details like the finishes and sizes of those rooms. All of this can be done without detailed blueprints. Similarly, in software engineering, you can use the functionality provided (and the the characteristics of the functionality) to estimate time and cost.

In addition to being available from the requirements phase, functional measures have two other advantages over non-functional measures. First, they are largely independent of the technology and platform being used. Second, they are user-focused and, hence, are easier to explain to non-technical stakeholders.

Lines of Code (LOC)

Perhaps the most obvious measure of program size is the number of physical lines of code. The obvious shortcoming of this approach is that some lines are not "of consequence." For example, commentary lines and blank lines are not executed, and hence may not inflate the "real" size of the program. Hence, many people instead argue that the number of logical lines of code, that is, the number of statements "of consequence," is a better measure of size. Interestingly, however, it is hard to agree on which lines are "of consequence." For example, shouldn't comments be included? They certainly take time and effort to write, so much so that many programmers omit them. As another example, should initialization statements be included? To settle these disagreements, some people argue that each statement of consequence should be weighted by its complexity or difficulty. But this only provides more grounds for argument.

As an alternative to lines of code, you can instead count modules (functions, methods, files, or classes). However, you still have the problem of differences in complexity or difficulty. In addition, such measures tend to be very highly correlated to LOC. Hence, LOC is the most popular non-functional measure, and often people opt for the simplest case of counting every non-commentary line of code without weights.

So how can product size in lines of code be estimated? The usual approach is to decompose the software product into smaller and smaller components until the components are so small and simple that estimators feel comfortable estimating their sizes. A work breakdown structure can be used for this decomposition, but some sort of design composition can be used as well. The component estimates are simply summed to produce a product size estimate.

The problem with this approach is that it may not reflect the actual design of the product (because at the start of a traditional project there is no design yet), and hence the components estimated may not coincide with the components ultimately comprising the product, resulting in an inaccurate estimate.

Function Points

One of the first functional measures was proposed by Albrecht (1979), and is called **function point analysis**. In this approach five different kinds of functionality, sometimes called *base functional component* (BFC) types, are measured. Three of these are related to processes or transactions and two are related to data storage.

Processes or Transactions

External Inputs (**EI**) are processes that provide data that will be used or stored by the product. For example, a process that adds new employees to a file or table of employee records is an EI process.

External Queries (**EQ**) are processes that retrieve stored data (without adding value or information but, perhaps, providing formatting). For example, a process that retrieves all of the employees in a file or table of employee records is an EQ process.

External Outputs (**EO**) are processes that provide derived information to a user (such as reports, messages, or prompts) or other system. For example, a process that calculates the average salary of all employees in a particular division is an EO process.

Data Storage

Internal Logical Files (**ILF**) are logical groupings of data maintained by the product. For example, a file or table of employee records would be counted as an ILF.

External Interface Files (**EIF**) are logical groupings of data that are external to the product (not maintained by the product) but are used or referenced by the product. For example, a file or table used by multiple products that maps machine-readable codes to human-readable descriptions is an example of an EIF.

To account for the fact that these different types of components can vary in complexity or difficulty both within a type and between different types, these counts are weighted. Specifically, each component is categorized as being either simple, average, or complex, and the following weights are applied.

Measure	**Simple**	**Average**	**Complex**
External Inputs	3	4	6
External Queries	3	4	6
External Outputs	4	5	7
Internal Logical Files	7	10	15
External Interface Files	5	7	10

Finally, the following fourteen value adjustment questions about the product are answered using a Likert scale that ranges from 0 (Not Important) to 5 (Essential).

1. Does the system require reliable backup and recovery?
2. Are specialized data communications required?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require on-line data entry?
7. Does the on-line data entry require input transactions over multiple screens or operations?
8. Are the ILFs updated on-line?
9. Are the inputs, outputs, files or inquiries complex?
10. Is the internal processing complex?
11. Is the code to be designed to be reusable?
12. Are conversion and installation included in the design?

13. Is the system designed for multiple installations in different organizations?

14. Is the application designed to facilitate change and ease of use by the user?

With this information in hand, the number of **function points** can be calculated. Specifically, letting M_{md} denote measure m of difficulty d, W_{md} denote the weighting factor for measure m of difficulty d, and V_q denote the value adjustment factor for question q, the number of function points, F, is defined as follows.

$$F = \sum_{m=1}^{5} \sum_{d=1}^{3} M_{md} \cdot W_{md} \cdot \left[0.65 + 0.01 \cdot \sum_{q=1}^{14} V_q \right]$$

For example, suppose a small product has the following components.

Measure	Simple	Average	Complex
External Inputs	2	0	1
External Queries	0	3	0
External Outputs	1	1	2
Internal Logical Files	3	0	0
External Interface Files	1	0	0

Then, the weighted EI measure is 2·3+0·4+1·6=12, the weighted EQ measure is 0·3+3·4+0·6=12, the weighted EO measure is 1·4+1·5+2·7=23, the weighted ILF measure is 3·7+0·10+0·15=21, and the weighted EIF measure is 1·5+0·7+0·10=5. Hence, if the answer to all 14 questions is 0 then the number of function points is (12+12+23+21+5)·0.65=73·0.65=47. If, on the other hand, the answer to three of the questions is 1, the answer to 2 questions is 2, and the answer to one question is 5 (it doesn't matter which one), the number of function points is 73·(0.65+0.12)=73·(0.77)=56.

This approach has been formalized and standardized by the International Function Point Users Group (IPFUG) as ISO20926 (2009). In addition, several variants have been developed. Function points are highly correlated with LOC counts, but they do not vary by programming language as LOC counts do, and it is often easier to estimate function points than LOC early in the life cycle.

Other Functional Measures

The Common Software Measurement International Consortium (COSMIC) created a functional measure that is suitable for both application software and system software. Their methodology has been standardized as ISO19761 (2011). The COSMIC methodology counts the *data movements* (movements of data from or to users and from or to persistent storage) associated with functional processes (steps required to complete an action) that are initiated as a result of *triggering events* (indivisible events that occur external to the product). The size of a functional process is the number of data movements associated with it, and the size of a software product is the size of all of its functional processes.

Roetzheim (2000) describes a version of function points that is applicable for Web Apps. When using this measure EI corresponds to input screens or forms, EQ corresponds to externally published interfaces, EO corresponds to HTML pages, ILF corresponds to internal database tables and XML files, and EIF corresponds to internal database tables and XML files.

Boehm (1996) suggests using the notion of **object points** rather than function points. In this variant there are three measures (i.e., screens in the user interface, reports, and components or modules), each of which is characterized as being simple, medium, or difficult, each with appropriate weights. The number of object points is then the sum of the weighted measures times a proportion of reuse (that is between 0 and 1).

Effort Estimation

Given a size estimate, we now need to construct an estimate of the effort that will be required to complete the product. Effort can be measured either in absolute units or in relative units but should be independent (to the extent possible) of the size of the staff that will conduct the work.

Effort estimates in traditional processes invariably use as their unit the **person-month**, which is the amount of effort a typical developer expends in one month of work. For example, if it is estimated that an average developer would need eleven days of uninterrupted work to complete a task, then the task's size is half a person-month (a month is four and a half weeks, and assuming five work days per week, a person-month has about 22 person-days).

Effort estimates generated from product size estimates include all life cycle work writing requirements and designs, doing testing, generating documentation, collecting data, managing the project, and so forth, as well as actually writing the code.

Standard statistical techniques can be used to determine the relationship between program size and effort. These analyses can be conducted either using industry-standard data or organization-specific data. The latter, if collected and used properly, will almost certainly lead to better estimates. However, in practice, both the collection and use of these data requires a level of sophistication that only mature (and large) organizations possess. Hence, we will only consider existing models that have been estimated using industry-standard data. While the same kinds of models can be estimated by individual organizations, we will not discuss the details of collecting such data or of the estimation techniques that are most appropriate.

Albrecht and Gafney (1983) and Kemerer (1987) both posited models of the form

$$E = \alpha + \beta F$$

where E denotes the effort (in person-months) and F denotes the size in function points.

Using several data sets, they used ordinary least squares (a statistical technique for estimating parameters) to determine the following values for α and β.

Study	α	β
Albrecht and Gafney	-91.4	0.255
Kemerer	-37.0	0.960

Obviously, given the negative values of α, neither model is useful for small products (since small values of F will yield negative values of E). Aside from that similarity, the models have very different implications. Albrecht and Gafney found that each function point adds about a quarter of a person-month of effort, whereas Kemerer found that each function point adds almost one person-month of effort. This difference can be explained in several different ways. It may be the result of differences in the calculation of function points. It also may also be the result of using data from different organizations. In either case, the differences tend to argue for the need for organization-specific data except when only coarse estimates are needed.

Several groups have also developed models that relate the size measured in (thousands of) lines of code to the effort. These groups have posited models of the form

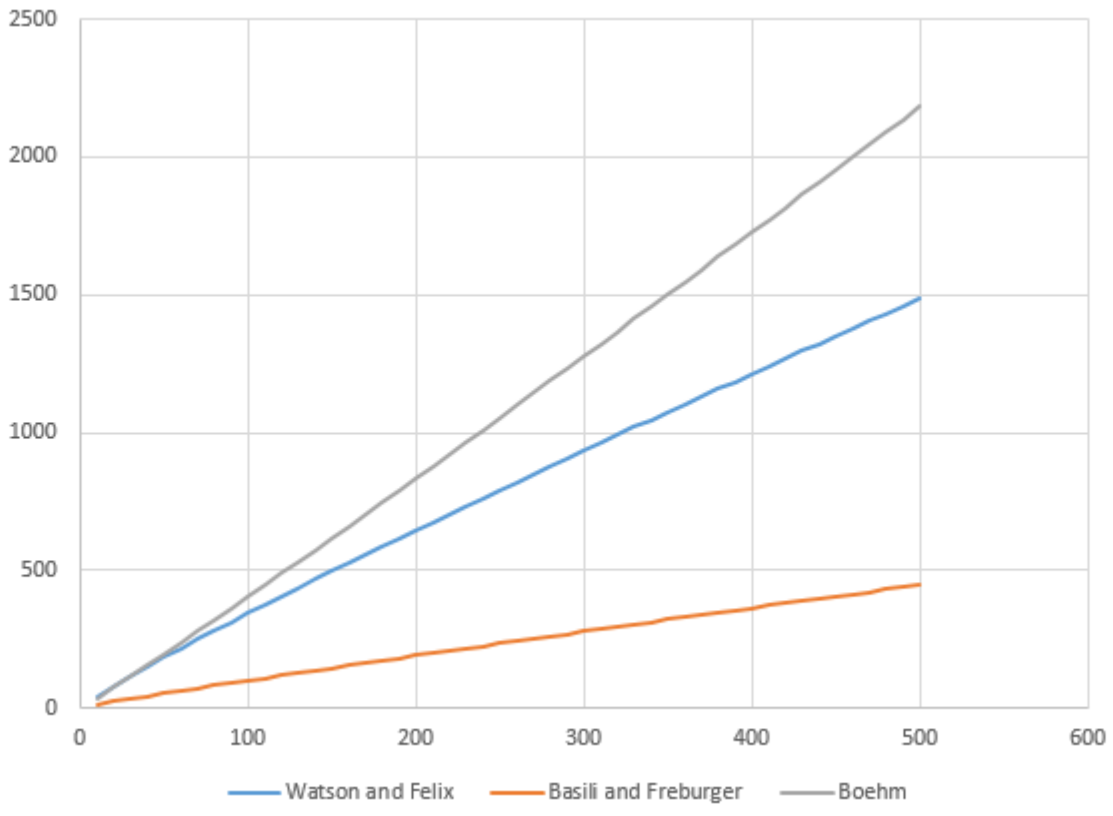
$$E = \alpha \cdot L^{\beta}$$

where L represents the number of thousands of lines of code (KLOC). In models of this form, the parameter β accounts for economies or diseconomies of scale. When $\beta < 1$ the project exhibits economies of scale and when $\beta > 1$ it exhibits diseconomies of scale. Diseconomies of scale result from such things as communications problems and integration problems. Economies of scale are very rare in software engineering and creative work in general.

Early models of this kind were estimated by Watson and Felix (1977), Basili and Freburger (1981), and Boehm (1981). Their results can be summarized as follows.

Study	α	β
Watson and Felix	5.20	0.91
Basili and Freburger	1.38	0.93
Boehm	3.20	1.05

A comparison of these models shows that they are quite different, especially for larger products. Consider the graph below and note how widely the predictions diverge as L grows larger.



Perhaps the most comprehensive set of "simple" models of these kinds have been estimated by the International Software Benchmarking Standards Group. They have estimated the parameters of models for various different platforms (e.g., mainframes, mid-range systems, desktops) and different kinds of languages. The general form of their models is:

$$E = \alpha \cdot F^{\beta} \cdot N^{\gamma}$$

were F is the product size in function points and N denotes the maximum size of the team. One representative instance is the following model.

$$E = 0.512 \cdot F^{0.392} \cdot N^{0.791}$$

It's interesting to consider the impact of the maximum size of the team in this model. Specifically, note that for a product with a given number of function points, increasing the maximum size of the team from five to six actually increases the number of person-months of effort by a factor of about 0.5. This is because larger teams tend to be less efficient than smaller teams.

One of the most complicated models is the **Constructive Cost Model** (COCOMO) developed by Boehm in 1981 and modified in 2000 (and now called COCOMO II). At its core, COCOMO II uses size measured in either lines of code or functional measures. However, it includes models that can be used to convert from functional measures to LOC (in a variety of different languages). COCOMO II actually has three different (but closely related) models used at different stages of the software process. We present the "Post-Architecture" model (which uses the same functional form as the "Early Design" model but a different number of effort multipliers). We will use notation that is consistent with the models presented in this book, rather than that in the documentation of the model.

The nominal-schedule effort (excluding the required development schedule adjustments) is given by the following equation.

$$E = 2.94 \cdot L^{\beta} \cdot \prod_{i=1}^{16} W_i$$

Here W_i denotes effort multiplier i and β is defined as follows.

$$\beta = 0.91 + 0.01 \cdot \sum_{j=1}^{5} \psi_j$$

Here ψ_j denotes scale factor j. This model can be used on a module-by-module basis. After estimating the effort required for each module and summing to find the effort required for the product, COCOMO II also includes an adjustment factor that captures the impact of schedule constraints. Note that while COCOMO II is presented here using thousands of lines of code (L) as the independent variable, one can also use it with function points (F). The model contains conversion factors for a wide variety of programming languages.

In COCOMO II, the parameter that accounts for diseconomies of scale (β) is calculated from the five scale factors in the following table.

Index	Description	Very Low	Low	Nominal	High	Very High	Extra High
1	Precedentedness	6.20	4.96	3.72	2.48	1.24	0.00
2	Flexibility	5.07	4.05	3.04	2.03	1.01	0.00
3	Risk Resolution	7.07	5.65	4.24	2.83	1.41	0.00
4	Team Cohesion	5.48	4.38	3.29	2.19	1.10	0.00
5	Process Maturity	7.80	6.24	4.68	3.12	1.56	0.00

Values in this table increase from right to left because diseconomies of scale increase from right to left. Given the definition of β, anytime the sum of the ψ_j values is greater than 9, the project will exhibit diseconomies of scale.

The effort multipliers (W_i) are given in the following table.

Index	Description	Very Low	Low	Nominal	High	Very High	Extra High
1	Req. Reliability	0.82	0.91	1.00	1.10	1.26	N/A
2	Database Size	N/A	0.90	1.00	1.14	1.28	N/A
3	Complexity	0.73	0.87	1.00	1.17	1.34	1.74
4	Req. Reuse	N/A	0.95	1.00	1.07	1.15	1.24
5	Req. Document.	0.81	0.91	1.00	1.11	1.23	N/A
6	Exec. Time Const.	N/A	N/A	1.00	1.11	1.29	1.63
7	Storage Const.	N/A	N/A	1.00	1.05	1.17	1.46
8	Platofrom Volatility	N/A	0.87	1.00	1.15	1.30	N/A
9	Analyst Capability	1.42	1.19	1.00	0.85	0.71	N/A
10	Programmer Capability	1.34	1.15	1.00	0.88	0.76	N/A
11	Personnel Continuity	1.29	1.12	1.00	0.90	0.81	N/A
12	Applications Experience	1.22	1.10	1.00	0.88	0.81	N/A
13	Platform Experience	1.19	1.09	1.00	0.91	0.85	N/A
14	Lang. & Tool Experience	1.20	1.09	1.00	0.91	0.85	N/A
15	Use of Tools	1.17	1.09	1.00	0.90	078	N/A
16	Multisite Development	1.22	1.09	1.00	0.93	0.86	0.80

The first group consists of characteristics of the product, the next group consists of characteristics of the platform, the next group consists of characteristics of the personnel, and the last group consists of characteristics of the project. Detailed definitions of each of the different levels is available in the model documentation.

As an example, consider the application of this model to a program with 50,000 lines of code ($L = 50$) in which all of the scale factors are nominal except for the team cohesion (which is low) and all of the effort multipliers are nominal except for the required reliability (which is very high) and the language and tool experience (which is very low).

$$\beta = 0.91 + 0.01 \cdot (3.72+3.04+4.24+4.38+4.68) = 0.91 + 0.01 \cdot 20.06 = 1.11$$

Since fourteen of the effort multipliers are 1.00 and the other two are 1.26 and 1.20,

$$\prod_i W_i = 1 \cdot 1 \cdot \cdots 1 \cdot \quad 1 \cdot \quad 1.26 \cdot \quad 1.20 = 1.51.$$

Hence

$$E = 2.94 \cdot \quad 50^{1.11} \cdot \quad 1.51 = 2.94 \cdot \quad 76.89 \cdot \quad 1.51 = 341.35$$

COCOMO II also includes techniques for differentiating new code from *adapted code* (that is, preexisting code that requires some changes) and *re-used code* (code that is used "as is"). These techniques essentially adjust the size of the product (they adjust *L*).

Effort Estimation in Scrum

In Scrum, planners do not estimate software size but instead go directly to effort estimates for stories and tasks. Effort estimates in Scrum can use person-days or person-months (called **ideal days** or **ideal months**) or a unit called a *story point*.

A **story point** or **task point** is a relative unit of size. Story point estimates are made by choosing one or a few small stories or tasks of about the same size and using them as a baseline (by declaring them to be of size one). Then all stories or tasks are estimated relative to this standard. In other words, if story or task X is defined to be the baseline and story or task Y is about four times as big, then the size of Y is four story points. Although it takes a little getting used to, the advantage of story points is that they do not require thinking in detail about the time that is actually required to accomplish a story or task; this abstraction may make estimation easier and hence more accurate. Story points are the most-used effort unit in Scrum projects, so we will restrict our attention to them in the remainder of our discussion.

In Scrum, effort estimations are refined during the project and become more detailed (and presumably more accurate) over time. Epic PBIs tend to have rough qualitative or categorical effort estimates (like small, medium, large, and extra-large) because there is no need to do the work to make more accurate estimates for them. But once PBIs begin to be refined as they rise in the product backlog, more accurate estimates are needed for release and sprint planning. PBIs chosen for sprints must be estimated especially accurately so that the team can ensure a successful sprint.

By not bothering with software size estimates and only making detailed effort estimates when they are needed, Scrum teams avoid much work and rework. The trade-off for this gain is that accurate early effort (and hence time and cost) estimates for a release or a product cannot be made. Of course, given the fluidity and changeability of products developed with agile methods, agilists would argue that accurate early estimates are impossible in any case, so nothing is really lost.

Planning Poker

Planning poker is a technique for an agile team to come to consensus on effort estimates. The team first chooses a sequence of numbers to be used as estimates. Usually this sequence is a modified Fibonacci sequence such as 1, 2, 3, 5, 8, 13, 20, 40, and 100, or the powers of two, that is, 1, 2, 4, 8, 16, and so forth up to some large value, like 128. The point of using such a scale is to try to attain estimation *accuracy* (an estimate that reflects reality) without attempting to achieve unrealistic *precision* (being very close to reality). By leaving out most of the larger numbers, agile estimators acknowledge that estimates for large stories will not be very precise (they will not be very exact) but they will be accurate (a very large story will indeed require a big effort). The numbers from the chosen scale are written on cards, and each team member gets a full set.

The team, PO and SM participate in planning poker, but only team members actually make estimates. The PO presents, explains, and clarifies PBIs when questioned about them during planning poker, and the SM facilitates participation, making sure that all team members contribute to discussion and have a voice in the results. The PO also records story estimates.

The process proceeds in rounds, and a PBI or task is estimated in each round. A round begins when the PO presents and explains a PBI and the team and PO discuss the PBI or task to make clear exactly what it involves. Then all team members simultaneously throw cards (one each) with his or her estimate of the PBI or task. If the cards all agree, the round is over and the thrown value is the estimate. If not, then the team members discuss why they made the estimates they did. It is especially important for the team members with the highest and lowest estimates to explain their reasoning. The PO may be further questioned about the PBI. Once everyone understands the rationales for the estimates and any confusions have been cleared up, the cards are collected and the process repeats from the point where all team members throw cards with his or her estimate.

Usually consensus is reached in two or three rounds. The the estimates are often quite good because they are based on discussion and reasoning. Furthermore, the discussion of PBIs and tasks that results from planning poker helps everyone on the team understand the PBIs and tasks better.

References

1. A.J. Albrecht, "Measuring application development productivity", *Proceedings of the IBM Application Development Symposium*, Monterey, CA, Oct. 14-17, 1979.
2. International Standards Organization, *IFPUG functional size mFunctional Size Measurement*, ISO/IEC 20926, 2009.
3. International Standards Organization, *COSMIC: A functional size measurement mFunctional Size Measurement Method*, ISO/IEC 19761, 2011.
4. W. Roetzheim "Estimating Internet Development", *Software Development*, www.sdmagazine.com/documents/s=741/sdm0008d/0008d.htm, Aug. 2000.
5. B. Boehm, "Anchoring the Software Process", *IEEE Software*, Vol. 13, pp. 73-82, 1996.
6. A.J. Albrecht and J.E. Gafney, "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation", *IEEE Trans.actions on Software Eng.ineering*, SE-9, pp. 639-648, 1983.
7. C.F. Kemerer, "An Empirical Validation of Software Cost Estimation Models", *Communications of the ACM*, Vol. 30, pp. 416-429, 1987.
8. C.E. Watson and C.P. Felix, "A method of program measurement and estimation", *IBM Systems Journal*, Vol. 16, pp. 54-73, 1977.
9. V.R. Basili and K. Freburger, Programming Measurement and Estimation in the Software Engineering Laboratory, *The Journal of Systems and Software*, Vol. 2, pp. 47-57, 1981.
10. B. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981.
11. B. Boehm, C. Abts, A.W. Brown, S. Chulani, B.K. Clark, E. Horowitz, R. Madachy, D.J. Reifer, and B. Steece, *Software Cost Estimation with COCOMO II*, Prentice-Hall, 2000.