1

Introduction to COCOMO II

1.1 COCOMO II USER OBJECTIVES

COCOMO II is a model to help you reason about the cost and schedule implications of software decisions you may need to make. In the last eighteen years of answering phone calls and email messages from users of the original COCOMO ([Boehm, 1981]; here called COCOMO 81), we have found many different types of COCOMO 81 users and uses. Perhaps the most unexpected were the occasional calls from Internal Revenue Service auditors. After quickly assuring us that our tax returns were not being audited, they would ask us about how to use COCOMO 81 to validate someone’s software tax write-off claim.

Here is a list of the major decision situations we have determined that you might want to use COCOMO II for in the future:

1. **Making investment or other financial decisions involving a software development effort.** A business case or return-on-investment analysis involving software development needs either an estimate of the software development cost or a life-cycle software expenditure profile.

2. **Setting project budgets and schedules as a basis for planning and control.** For example, how many people should you assign to the early, middle, and late
3. **Deciding on or negotiating tradeoffs among software cost, schedule, functionality, performance or quality factors.** For example, to what extent can one say, “Quality is free?” Is it actually cheaper to build life-critical software than to build, say, inventory control software?

4. **Making software cost and schedule risk management decisions.** You will have unavoidable uncertainties about many of the factors influencing your project’s cost and schedule. Some example uncertainties are the amount of software your project will develop and reuse. Cost models can help you perform sensitivity and risk analyses covering your sources of uncertainty.

5. **Deciding which parts of a software system to develop, reuse, lease, or purchase.** A good cost model can help you understand, for example, when it is cheaper to build a component than to rework an existing component.

6. **Making legacy software inventory decisions: what parts to modify, phase out, outsource, etc.** For example, the model can help you develop and periodically update a multi-year plan indicating how many of the highest-priority capabilities can be developed in each legacy system upgrade.

7. **Setting mixed investment strategies to improve your organization’s software capability, via reuse, tools, process maturity, outsourcing, etc.** Here, the model can help you develop and periodically update a multi-year technology investment plan. For example, COCOMO II has capabilities both to estimate the additional investment required to produce reusable software, and the resulting savings accruing from its reuse.

8. **Deciding how to implement a process improvement strategy, such as that provided in the SEI Capability Maturity Model [Pauk et al. 1995].** For Level 2, how should you combine software cost models such as COCOMO II with other methods of sizing, estimating, planning, and tracking software projects’ cost, schedule, and effort? For Level 3, how do you assess the benefits of training in terms of improvements in such cost drivers as application, platform, language, and tool experience? For Level 4, how can frameworks such as COCOMO II help you set up an effective quantitative software management program? For Level 5, how can you evolve this framework to accommodate such new practices as product lines, rapid application development, and commercial-off-the-shelf (COTS) software integration?

As a bottom line, a software estimation model can be much more than a facility to plug in cost driver ratings and receive budget and schedule estimates. Helping you realize these additional decision support capabilities has determined a number of COCOMO II’s objectives and strategies, as discussed next.
1.2 COCOMO II MODEL OBJECTIVES

As discussed in the Preface, we will relate the COCOMO II model objectives to the feedback we have received over the years on the strong points and capabilities needing improvement in COCOMO 81:

1. Provide accurate cost and schedule estimates for both current and likely future software projects. COCOMO 81 is built on the 1970s’ “waterfall” process framework (sequentially progressing through requirements, design, code, and test). Many organizations are evolving toward mixes of concurrent, iterative, incremental, and cyclic processes. COCOMO II provides a framework for tailoring a version of the model to your desired process, including the waterfall.

2. Enable organizations to easily recalibrate, tailor, or extend COCOMO II to better fit their unique situations. Recalibration enables your organization to fit COCOMO II to your collected project data, with its own special interpretations of project endpoints, product definitions, labor definitions, and organizational practices. Tailoring and extensibility enable you to orchestrate COCOMO II variants to fit your organization’s product, process, and personnel practice variants. COCOMO 81 provided recalibration, but tailoring was available only within the assumptions of the waterfall model.

3. Provide careful, easy-to-understand definitions of the model’s inputs, outputs, and assumptions. Without such definitions, it is easy to create “million-dollar mistakes” when different people use different interpretations of the model’s estimates. COCOMO 81 was generally satisfactory here; COCOMO II provides some additional definition assistance for the cost driver rating scales.

4. Provide a constructive model, in which the job of cost and schedule estimation helps people better understand the nature of the project being estimated. This was a major objective for COCOMO 81, and continues to be for COCOMO II. The primary improvements in this direction have been to add further cost drivers to help understand the effects of project decisions in the reuse and economy-of-scale areas.

5. Provide a normative model, which allocates the resources necessary for effective software development or maintenance; which highlights unrealistic cost and schedule objectives; and which identifies management controllables for improving current or future projects. Software cost models calibrated to data generally do well on being normative, because very few poorly-managed projects collect data for calibration. As discussed with the previous objective, COCOMO II identifies further management controllables in the reuse and economy-of-scale areas.
6. Provide an evolving model, which adds new capabilities to address new needs (e.g., COTS integration, rapid application development), and maintains relevance to evolving software practices. The COCOMO 81 effort did not plan for model evolution. COCOMO II is a continuing project, both for refining the main project cost-schedule estimation model and for providing extensions (discussed primarily in Chapter 5).

1.3 COCOMO II DEVELOPMENT AND EVOLUTION STRATEGIES

Developing a model to satisfy all of the above objectives would be difficult even in a relatively stable software arena. It is even more difficult in the current and foreseeable future situation, in which new generations of products and processes are changing the way organizations develop software.

Our strategies in developing and evolving COCOMO II have reflected this climate of rapid change and limited foreseeability. The three primary strategies pursued to date have been:

1. **Proceed incrementally**, addressing the estimation issues of most importance and tractability with respect to modeling, data collection and calibration. Our primary initial focus has been on developing a solid model for overall project cost and schedule estimation. We have also initially addressed some further areas where we have had a reasonable combination of experience and data, such as software maintenance. We have proceeded more cautiously in areas where we had less experience and data. These areas are discussed in Chapter 5: applications composition, COTS integration, rapid application development, detailed effort and schedule breakdowns by stage and activity.

2. **Test the models and their concepts to gain first-hand experience**. We use COCOMO II in our annual series of USC Digital Library projects. These involve the architecting of fifteen to twenty potential products, and the selection, development and transition of five to six products annually. We have found the need to modify COCOMO II somewhat to adapt it to our two-semester, fixed-team-size projects. But the experience has given us numerous insights, particularly in the development of the COCOMO II extensions for COTS integration and rapid applications development.

3. **Establish a COCOMO II Affiliates’ program**, enabling us to draw on the prioritized needs, expertise and calibration data of leading software organizations. This has given us access to the large-project experience unavailable in a university setting. We have been fortunate to have a good mix of commercial software developers, government software contractors, and government software acquisition organizations from whom we have acquired a
balanced set of project data points for calibrating COCOMO II. Also, the Affiliate connections have provided us with access to many of the leading software cost estimation and metrics practitioners. A list of the COCOMO II Affiliates is provided in the acknowledgements appearing after the Preface.

The Affiliates’ prioritized needs and available expertise and data have led to six additional strategies:

4. *Provide an externally and internally open model*, enabling users to fully understand and deal with its inputs, outputs, internal models, and assumptions. The only hidden aspect of COCOMO II is its database of Affiliates’ project effort and schedule data. We have developed a set of safeguards and procedures that have prevented any leakage of Affiliates’ sensitive data.

5. *Avoid unnecessary incompatibilities with COCOMO 81*, which most Affiliates have still found largely useful and relevant. There were a few exceptions to this strategy, where subsequent trends and insights had made the original COCOMO 81 approach obsolete. These included eliminating the Turnaround Time cost driver, rebaselining the Software Tools rating scale, eliminating the schedule-stretchout penalty, replacing the linear reuse model with a nonlinear one, and replacing development modes by exponential scale factors.

6. *Experiment with a number of model extensions*, prioritizing their pursuit based on Affiliates’ needs and available data. These have included some extensions of the main COCOMO II project cost and schedule model, such as adding multiplicative cost drivers addressing development for reuse and personnel continuity, and adding exponential scale factors for process maturity and team cohesion. They also include the experimental extensions discussed in Chapter 5, covering estimators for COTS integration and rapid development effects, and for delivered software quality (defect density).

7. *Balance expert-determined and data-determined modeling*, most effectively via Bayesian analysis techniques. The Bayesian calibration approach has been our most significant methodological improvement. Section 1.6 provides an overview, and Chapter 4 provides the full approach and results.

8. *Develop a sequence of increasingly accurate models*, based on the increasingly detailed and accurate input data available to estimators as they progress through succeeding stages of the software life cycle. We have done this via the Applications Composition, Early Design, and Post-Architecture sequence of models discussed in the next two sections.

9. *Key the COCOMO II models to projections of future software life-cycle practices*.

Our projection of future software practices and associated set of COCOMO II models is discussed next.
1.4 FUTURE SOFTWARE PRACTICES MARKETPLACE MODEL

Figure 1.1 summarizes the model of the future software practices marketplace that we developed in 1994 to guide the development of COCOMO II [Boehm et al. 1995]. It includes a large upper “end-user programming” sector with an estimated size of roughly fifty-five million practitioners in the United States by the year 2005; a lower “infrastructure” sector with roughly 750,000 practitioners; and three intermediate sectors, involving the development of application generators and composition aids (600,000 practitioners), the development of systems by applications composition (700,000), and system integration of large-scale and/or embedded software systems (700,000).

End-user programming will be driven by increasing computer literacy and competitive pressures for rapid, flexible, and user-driven information processing solutions. These trends will push the software marketplace toward having users develop most information processing applications themselves via application generators. Some example application generators are spreadsheets, extended query systems, and simple, specialized planning or inventory systems. They en-

---

1These figures are judgment-based extensions of the Bureau of Labor Statistics moderate-growth labor distribution scenario for the year 2005 [CSTB, 1993; Silvestri and Lukasiewicz, 1991]. The fifty-five million End-User programming feature was obtained by applying judgment-based extrapolations of the 1989 Bureau of the Census data on computer usage fractions by occupation [Kominski, 1991] to generate end-user programming fractions by occupation category. These were then applied to the 2005 occupation-category populations (e.g., 10% of the 25M people in “Service Occupations”; 40% of the 17M people in “Marketing and Sales Occupations”). The 2005 total of 2.75M software practitioners was obtained by applying a factor of 1.6 to the number of people traditionally identified as “Systems Analysts and Computer Scientists” (0.829M in 2005) and “Computer Programmers” (0.882M). The expansion factor of 1.6 to cover software personnel with other job titles is based on the results of a 1983 survey on this topic [Boehm 1983]. The 2005 distribution of the 2.75M software developers is a judgment-based extrapolation of current trends.
able users to determine their desired information processing application via
domain-familiar options, parameters, or simple rules. A major new source of
such applications could be called "Webmaster" applications. Every enterprise
from Fortune 100 companies to small businesses and the U.S. government will be
involved in this sector.

Typical infrastructure sector products will be in the areas of operating sys-
tems, database management systems, user interface management systems, and
networking systems. Increasingly, the infrastructure sector will address "middle-
ware" solutions for such generic problems as distributed processing and tran-
saction processing. Representative firms in the Infrastructure sector are Microsoft,
Netscape, Oracle, Sybase, 3Com, Novell, and the major computer vendors.

In contrast to end-user programmers, who will generally know a good deal
about their applications domain and relatively little about computer science, the
infrastructure developers will generally know a good deal about computer sci-
ence and relatively little about applications. Their product lines will have many
reusable components, but the pace of technology (new processor, memory, com-
munications, display, and multimedia technology) will require them to build
many components and capabilities from scratch.

1.4.1 Intermediate Sectors

Performers in the three intermediate sectors in Figure 1.1 will need to know a good
deal about computer science-intensive infrastructure software and also one or
more application domains. Creating this talent pool is a major national challenge.

The application generators sector will create largely prepackaged capabilities
for user programming. Typical firms operating in this sector are Microsoft,
Netscape, Lotus, Novell, Borland, and vendors of computer-aided planning, en-
gineering, manufacturing, and financial analysis systems. Their product lines will
have many reusable components but will also require a good deal of new-
capability development from scratch. Application composition aids will be de-
veloped both by the above firms and by software product-line investments of firms
in the applications composition sector.

The applications composition sector deals with applications that are too large
or diversified to be fully handled by prepackaged solutions, but which are suffi-
ciently simple or mature to be rapidly composable from interoperable compo-
nents. Typical components will be graphic user interface (GUI) builders, database
or object managers, middleware for distributed processing or transaction pro-
cessing, hypermedia handlers, smart data finders, and domain-specific compo-
nents such as financial, medical, or industrial process control packages.

Most large firms will have groups to compose such applications, but a great
many specialized software firms will provide composed applications on contract.
These range from large, versatile firms such as Andersen Consulting and EDS, to
small firms specializing in such specialty areas as decision support or transaction
processing, or in such application domains as finance or manufacturing.
The *system integration* sector deals with large-scale, highly embedded, or unprecedented systems. Portions of these systems can be developed with application composition capabilities, but their demands generally require a significant amount of up-front systems engineering and custom software development. Aerospace firms operate within this sector, as do major system integration firms such as EDS and Andersen Consulting, large firms developing software-intensive products and services (telecommunications, automotive, financial, and electronic products firms), and firms developing very large-scale corporate information systems or manufacturing support systems.

### 1.4.2 1999 Model Assessment

In a 1999 assessment of the model, it appeared that its predictions were reasonably on track. The fifty-five million figure for end-user programming performers in 2005 may be somewhat high. But the trend is toward a very high number, particularly with the emergence of large numbers of practitioners in the “Webmaster” category—although such categories are blurring the distinction between what might or might not be called end-user programming.

The 1998 U.S. population of software developers has been estimated by [Jones 1998] and [Rubin 1999] to be roughly two million. Given a rough trend of a 5 percent cumulative annual growth rate in software developers, this would yield a 2005 population of roughly 2.8 million, fairly close to the 2.75 million estimated in the 1994 model.

The distribution of software developers across the sectors is more difficult to compare, given the blurring of distinctions among the sectors. As another example, it is hard to distinguish which Web application support capabilities should be called “application generators” and which “infrastructure.”

### 1.5 RESULTING FAMILY OF COCOMO II MODELS

To support the software practices marketplace sectors above, COCOMO II provides a family of increasingly detailed software cost estimation models, each tuned to the sectors’ needs and type of information available to support software cost estimation.

#### 1.5.1 COCOMO II Models for the Software Marketplace Sectors

*The user programming sector does not need a COCOMO II model.* Its applications are typically developed in hours to days, so a simple activity-based estimate will generally be sufficient.

*The COCOMO II model for the application composition sector is based on object points.* Object points are a count of the screens, reports and third-generation language modules developed in the application, each weighted by a three-level...
(simple, medium, difficult) complexity factor [Banker et al. 1994; Kauffman-Kumar 1993]. This is commensurate with the level of information generally known about an application composition product during its planning stages, and the corresponding level of accuracy needed for its software estimates (such applications are generally developed by a small team in a few weeks to months).

The COCOMO II capability for estimation of application generator, system integration, or infrastructure developments is based on a tailor-able mix of the application composition model (for early prototype efforts) and two increasingly detailed estimation models for subsequent portions of the life cycle.

The rationale for providing this tailor-able mix of models rests on three primary premises:

First, unlike the initial COCOMO 81 situation in the late 1970s, in which there was a single, preferred software life-cycle model (the waterfall model), current and future software projects will be tailoring their processes to their particular process drivers. These process drivers include COTS or reusable software availability; degree of understanding of architectures and requirements; market window or other schedule constraints; and required reliability (see [Boehm 1989, pp. 436-437] for an example of such tailoring guidelines).

Second, the granularity of the software cost estimation model used needs to be consistent with the granularity of the information available to support software cost estimation. In the early stages of a software project, very little may be known about the size of the project to be developed, the nature of the target platform, the nature of the personnel to be involved in the project, or the detailed specifics of the process to be used.

Figure 1.2, extended from [Boehm 1981, p. 311], indicates the effect of project uncertainties on the accuracy of software size and cost estimates. In the very early stages, one may not know the specific nature of the product to be developed to better than a factor of 4. As the life cycle proceeds, and product decisions are made, the nature of the product and its consequent size are better known, and the nature of the process and its consequent cost drivers are better known. The earlier “completed programs” size and effort data points in Figure 1.2 are the actual sizes and efforts of seven software products built to a partially-defined specification [Boehm et al. 1984]. The later “USAF/ESD proposals” data points are from five proposals submitted to the U.S. Air Force Electronic Systems Division in response to a fairly thorough specification [Deveny 1976].

Third, given the situation in the first and second premises, COCOMO II enables projects to furnish coarse-grained cost driver information in the early project stages, and increasingly fine-grained information in later stages. Consequently, COCOMO II does not produce point estimates of software cost and ef-

---

2These seven projects implemented the same algorithmic version of the Intermediate COCOMO 81 cost model, but with the use of different interpretations of the other product specifications: produce a “friendly user interface” with a “single-user file system.”
Figure 1.2  Software costing and sizing accuracy versus phase

fort, but rather range estimates tied to the degree of definition of the estimation inputs. The uncertainty ranges in Figure 1.2 are used as starting points for these estimation ranges.

1.5.2 Tailoring COCOMO II Estimation Models to Process Strategies

With respect to process strategy, application generator, system integration, and infrastructure software projects will involve a mix of three major process models. The appropriate sequencing of these models will involve a mix of three major process models. The appropriate sequencing of these models will depend on the project’s marketplace drivers and degree of product understanding.

The early prototyping stage involves prototyping efforts to resolve potential high-risk issues such as user interfaces, software/system interaction, performance, or technology maturity. The costs of this type of effort are best estimated by the application composition model.

The early design stage involves exploring alternative software/system architectures and concepts of operation. At this stage, not enough is generally known to support fine-grain cost estimation. The corresponding COCOMO II capability
involves the use of function points (or lines of code where available) and a small number of additional cost drivers.

The post-architecture stage involves the actual development and maintenance of a software product. This stage proceeds most cost-effectively if a software life-cycle architecture has been developed; validated with respect to the system’s mission, concept of operation, life-cycle plan and risk; and established as the framework for the product. The corresponding COCOMO II model has about the same granularity as the previous COCOMO 81 and Ada COCOMO models. It uses source instructions and/or function points for sizing, with modifiers for reuse and software breakage; a set of seventeen multiplicative cost drivers; and a set of five factors determining the project’s scaling exponent. These factors replace the development modes (Organic, Semidetached, or Embedded) in the original COCOMO 81 model, and refine the four exponent-scaling factors in Ada COCOMO.

To summarize, COCOMO II provides the following three-model series for estimation of Application Generator, System Integration and Infrastructure software projects contained in our marketplace model:

1. The earliest phases or spiral cycles will generally involve prototyping, using Application Composition capabilities. The COCOMO II Application Composition model supports these phases, and any other prototyping activities occurring later in the life cycle.

2. The next phases or spiral cycles will generally involve exploring architectural alternatives or incremental development strategies. To support these activities, COCOMO II provides an early estimation model. This uses function points (or lines of code where available) for sizing, and a coarse-grained set of seven cost drivers (e.g., two cost drivers for Personnel Capability and Personnel Experience in place of the six current Post-Architecture model cost drivers covering various aspects of personnel capability, continuity and experience). Again, this level of detail is consistent with the general level of information available and the general level of estimation accuracy needed at this stage.

3. Once the project is ready to develop and sustain a fielded system, it should have a life-cycle architecture, which provides more accurate information on cost driver inputs and enables more accurate cost estimates. To support this stage of development COCOMO II provides a model whose granularity is roughly equivalent to the current COCOMO 81 and Ada COCOMO models. It can use either source lines of code or function points for a sizing parameter, five exponential scale factors as a refinement of the COCOMO 81 development modes, and seventeen multiplicative cost drivers.