



## Improving Software Development Economics Part I: Current Trends

by [Walker Royce](#)

Vice President and General  
Manager  
Strategic Services  
Rational Software

*Over the past two decades, the software industry has moved unrelentingly toward new methods for managing the ever-increasing complexity of software projects. We have seen evolutions, revolutions, and recurring themes of success and failure. While software technologies, processes, and methods have advanced rapidly, software engineering remains a people-intensive process. Consequently, techniques for managing people, technology, resources, and risks have profound leverage.*



*For more than twenty years, Rational has been working with the world's largest software development organizations across the entire spectrum of software domains. Today, we employ more than 1,000 software engineering professionals who work onsite with organizations that depend on software. We have harvested and synthesized many lessons from this in-the-trenches experience. We have diagnosed the symptoms of many successful and unsuccessful projects, identified root causes of recurring problems, and packaged patterns of software project success into a set of best practices captured in the Rational Unified Process. Rational is also a large-scale software development organization, with more than 750 software developers. We use our own techniques, technologies, tools, and processes internally, with outstanding results, as evidenced by our own business performance and product leadership in the market.*

*One of our primary goals is to apply what we have learned in order to enable software development organizations to make substantial improvements in their software project economics and organizational capabilities. This is the first in a series of four articles that summarize the key approaches that deliver these benefits.*

- ▶ [subscribe](#)
- ▶ [contact us](#)
- ▶ [submit an article](#)
- ▶ [rational.com](#)
- ▶ [issue contents](#)
- ▶ [archives](#)
- ▶ [mission statement](#)
- ▶ [editorial staff](#)

# A Simplified Model of Software Economics

There are several software cost models in use today. The most popular, open, and well-documented model is the COConstructive COst MOdel (COCOMO), which has been widely used by industry for 20 years. The latest version, COCOMO II, is the result of a collaborative effort led by the University of Southern California (USC) Center for Software Engineering, with the financial and technical support of numerous industry affiliates. The objectives of this team are threefold:

1. To develop a software cost and schedule estimation model for the lifecycle practices of the post-2000 era
2. To develop a software project database and tool support for improvement of the cost model
3. To provide a quantitative analytic framework for evaluating software technologies and their economic impacts

The accuracy of COCOMO II allows its users to estimate cost within 30 percent of actuals, 74 percent of the time. This level of unpredictability in the outcome of a software development process should be truly frightening to any software project investor, especially in view of the fact that few projects ever miss their financial objectives by doing better than expected.

The COCOMO II cost model includes numerous parameters and techniques for estimating a wide variety of software development projects. For the purposes of this discussion, we will abstract COCOMO II into a function of four basic parameters:

- **Complexity.** The complexity of the software solution is typically quantified in terms of the size of human-generated components (the number of source instructions or the number of function points) needed to develop the features in a usable product.
- **Process.** This refers to the process used to produce the end product, and in particular its effectiveness in helping developers avoid non-value-adding activities.
- **Team.** This refers to the capabilities of the software engineering team, and particularly their experience with both the computer science issues and the application domain issues for the project at hand.
- **Tools.** This refers to the software tools a team uses for development, that is, the extent of process automation.

The relationships among these parameters in modeling the estimated effort can be expressed as follows:

$$\text{Effort} = (\text{Team}) * (\text{Tools}) * (\text{Complexity}) (\text{Process})$$

Schedule estimates are computed directly from the effort estimate and

process parameters. Reductions in effort generally result in reductions in schedule estimates. To simplify this discussion, we can assume that the "cost" includes both effort and time. The complete COCOMO II model includes several modes, numerous parameters, and several equations. This simplified model enables us to focus the discussion on the more discriminating dimensions of improvement.

What constitutes a good software cost estimate is a very tough question. In our experience, a good estimate can be defined as one that has the following attributes:

- It is conceived and supported by the project manager, the architecture team, the development team, and the test team accountable for performing the work.
- It is accepted by all stakeholders as ambitious but realizable.
- It is based on a well-defined software cost model with a credible basis and a database of relevant project experience that includes similar processes, similar technologies, similar environments, similar quality requirements, and similar people.
- It is defined in enough detail for both developers and managers to objectively assess the probability of success and to understand key risk areas.

Although several parametric models have been developed to estimate software costs, they can all be generally abstracted into the form given above. One very important aspect of software economics (as represented within today's software cost models) is that the relationship between effort and size (see the equation above) exhibits a diseconomy of scale. The software development diseconomy of scale is a result of the "process" exponent in the equation being greater than 1.0. In contrast to the economics for most manufacturing processes, the more software you build, the greater the cost per unit item. It is desirable, therefore, to reduce the size and complexity of a project whenever possible.

## **Trends in Software Economics**

Software engineering is dominated by intellectual activities focused on solving problems with immense complexity and numerous unknowns in competing perspectives. In the early software approaches of the 1960s and 1970s, craftsmanship was the key factor for success; each project used a custom process and custom tools. In the 1980s and 1990s, the software industry matured and transitioned into more of an engineering discipline. However, most software projects in this era were still primarily research-intensive, dominated by human creativity and diseconomies of scale. Today, the next generation of software processes is driving toward a more production-intensive approach, dominated by automation and economies of scale. We can further characterize these three generations of software development as follows:

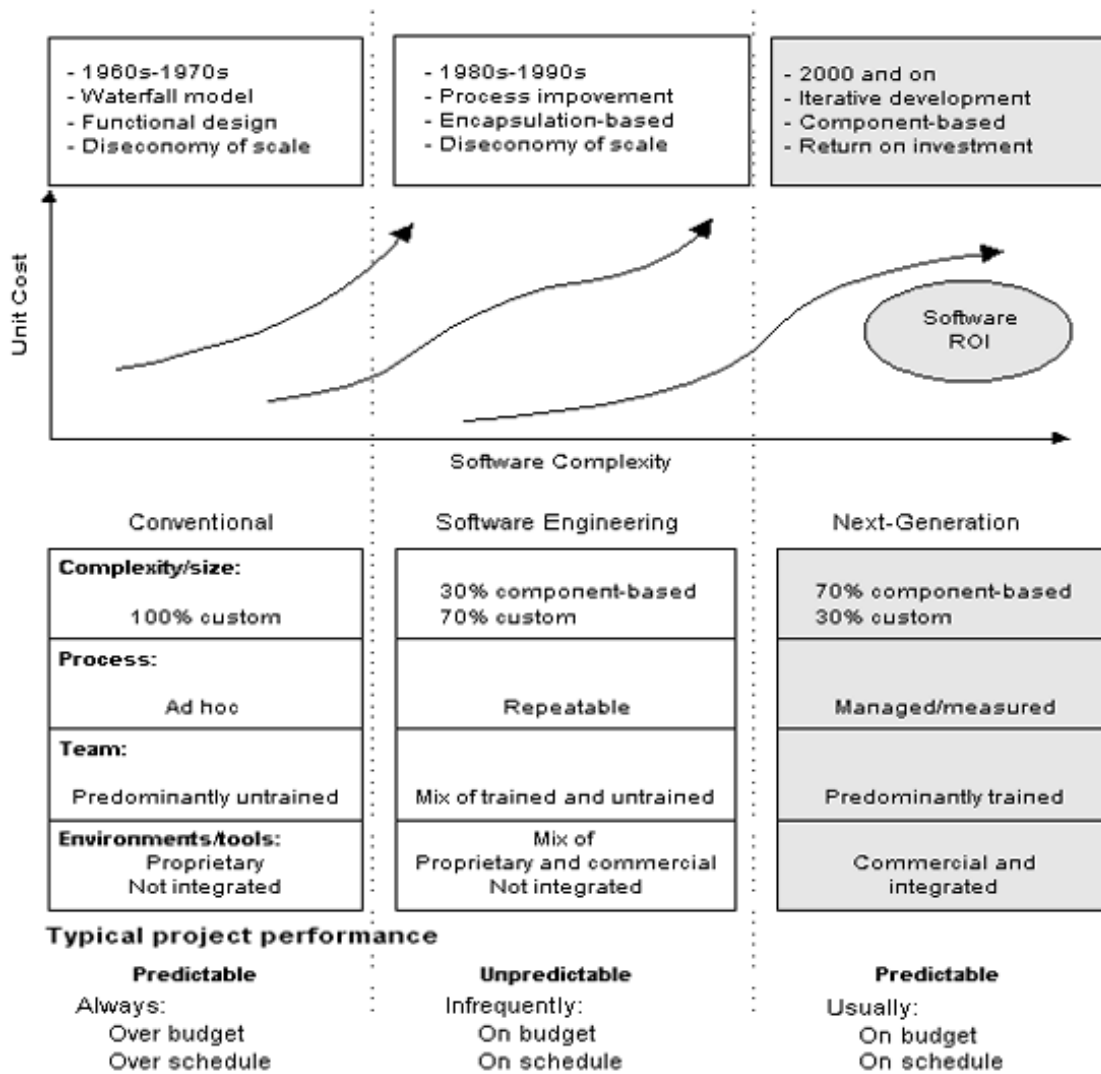
1. **1960s and 1970s: Conventional.** Organizations used virtually all custom tools, custom processes, and custom components built in

primitive languages. Project performance was highly predictable: Cost, schedule, and quality objectives were almost never met.

2. **1980s and 1990s: Software engineering.** Organizations used more repeatable processes, off-the-shelf tools, and about 70 percent of their components were built in higher level languages. About 30 percent of these components were available as commercial products, including the operating system, database management system, networking, and graphical user interface. During the 1980s, some organizations began achieving economies of scale, but with the growth in applications complexity (primarily in the move to distributed systems), the existing languages, techniques, and technologies were just not enough.
3. **2000 and later: Next generation.** Modern practice is rooted in the use of managed and measured processes, integrated automation environments, and mostly (70 percent) off-the-shelf components. Typically, only about 30 percent of components need to be custom built.

Figure 1 illustrates the economics associated with these three generations of software development. The ordinate of the graph refers to software unit costs (per Source Line of Code [SLOC], per function point, per component - take your pick) realized by an organization. The abscissa represents the life-cycle growth in the complexity of software applications developed by the organization.

Technologies for achieving reductions in complexity/size, process improvements, improvements in team proficiency, and tool automation are not independent of one another. In each new generation, the key is complementary growth in all technologies. For example, the process advances could not be used successfully without new component technologies and increased tool automation.



**Figure 1: Trends in Software Economics**

## Keys to Improvement: A Balanced Approach

Improvements in the economics of software development have been not only difficult to achieve, but also difficult to measure and substantiate. In software textbooks, trade journals, and market literature, the topic of software economics is plagued by inconsistent jargon, inconsistent units of measure, disagreement among experts, and unending hyperbole. If we examine only one aspect of improving software economics, we are able to draw only narrow conclusions. Likewise, if an organization focuses on improving only one aspect of its software development process, it will not realize any significant economic improvement -- even though it may make spectacular improvements in this single aspect of the process.

The key to substantial improvement in business performance is a balanced attack across the four basic parameters of the simplified software cost model: Complexity, Process, Team, and Tools. These parameters are in priority order for most software domains. In Rational's experience, the

following discriminating approaches have made a difference in improving the economics of software development and integration:

1. Reduce the size or complexity of what needs to be developed.
  - Manage scope.
  - Reduce the amount of human-generated code through component-based technology.
  - Raise the level of abstraction, and use visual modeling to manage complexity.
2. Improve the development process.
  - Reduce scrap and rework by transitioning from a waterfall process to a modern, iterative development process.
  - Attack significant risks early through an architecture-first focus.
  - Use software best practices.
3. Create more proficient teams.
  - Improve individual skills.
  - Improve project teamwork.
  - Improve organizational capability.
4. Use integrated tools that exploit more automation.
  - Improve human productivity through advanced levels of automation.
  - Eliminate sources of human error.
  - Enable process improvements.

Most software experts would also stress the significant dependencies among these trends. For example, new tools enable complexity reduction and process improvements; size-reduction approaches lead to process changes; and process improvements drive tool advances.

In the subsequent issues of *The Rational Edge*, we will elaborate on the approaches listed above for achieving improvements in each of the four dimensions. These approaches represent patterns of success we have observed among Rational's most successful customers who have made quantum-leaps in improving the economics of their software development efforts.



**For more information on the products or services discussed in this article, please click [here](#) and follow the instructions provided. Thank you!**

