A Digest of Research on Software Quality

Christopher Scaffidi

cscaffid /located at} cs.cmu.edu

Institute for Software Research, International
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract
Successfully competing in the software development market requires controlling the quality of the product. This not only involves identifying what software attributes are valued by the target market, but also measuring and managing those attributes over the course of the software development cycle. This represents an emerging point of view in software engineering, as explicit considerations of cost and value over time are largely missing from usual software development methods.

This report attempts to help software engineers achieve those goals. It briefly summarizes the relevant aspects of general economic theory and ties those considerations to the challenges of software development. Then, it discusses research surrounding four evaluation techniques (ATAM, Heuristic Evaluation, COCOMO II, and Code Complexity Metrics) which can help software engineers deliver the right quality attributes through their product. Finally, this report comments on where organizations can expect to head from here, in terms of process optimization and current areas of research.

0.0 Introduction

1.0 Microeconomics 101

1.1 Factor Exchange
1.2 Attributes and Functions
1.3 Economic Choices
1.4 Economic Optimizations

2.0 Application to Software Engineering

2.1 Software as a Valuable Factor
2.2 Software Attributes
2.3 Evaluation Techniques
2.4 Example: An Attribute Evaluation Gone Wrong

3.0 Controlling Software Quality

3.1 Example: Qualitative Techniques Early in the Project – ATAM
3.2 Example: Qualitative Techniques Late in the Project – Heuristic Evaluation
3.3 Example: Quantitative Techniques Early in the Project – COCOMO II Early Design
3.4 Example: Quantitative Techniques Late in the Project – Code Complexity Metrics

4.0 Beyond Today’s Product

4.1 Optimizing the Producer
4.2 Areas of Active Research
4.3 Annotated Bibliography
0.0 Introduction

One writer tells the following fictitious but revealing stories:

A vendor creates a product that helps development teams control their development process. The product is well designed and well made, but fails in its target market of UNIX shops. UNIX shops pride themselves in getting code out without needing a formal process.

Another vendor makes an instrument so straightforward that unskilled operators can run it with ease. Their customer base won't buy it because they consider themselves highly skilled professionals who can run complicated systems.

The writer told these stories to make the point that software designers must take into account the culture of their target market [9]. Failing to do so dooms the product—no matter how perfect its implementation—to failure.

That lesson constitutes but one application within a much larger economic framework with one basic principle: customers only buy products they value.

Too often, programmers implement software simply because it sounds interesting. Then, afterward, they look for a need to meet with the product. This puts the cart before the horse, so to speak, and often leads to software that is sub-optimal for that need.

Even when engineers gather requirements before coding, they sometimes fail to make the connection between the economic realities of the specific market and the specific attributes that this particular software should provide. Indeed, they may not even understand how to think about this connection. As another author writes [31],

We (and the software engineering community in general) do not understand quality attributes well: what it means to be “open” or “interoperable” or “secure” or “high performance” changes from system to system, from stakeholder to stakeholder, and from community to community.

This report attempts to partially address these understanding gaps. Of course, space does not allow for a detailed discussion of every market segment, every quality attribute, and how they all relate to one another. Therefore, this report focuses on two key issues:

- Showing how software development fits into the larger economic framework
- Summarizing existing research which engineers can apply to software development in order to succeed within this framework

Consequently, the first part of this report reviews basic microeconomic principles that will frame later discussions. It summarizes basic terminology and concepts, in order to provide a small but valuable vocabulary. Then, it uses this vocabulary to describe the dynamics of relationships between producers and customers within a competitive market. Finally, it discusses the choices to be made in the process of optimizing position within this market.

The second part of this report ties these principles and vocabulary to software development. The chief result of this discussion is to identify some of the software attributes, such as reliability and low price, which together make software a good value. No attribute is equally valued by all clients, and no client equally values all attributes. Therefore, software must incorporate a customized set of attributes for each target market segment. This leads to tradeoffs, as numerous constraints limit the ways that software developers can adjust these attributes. Together, the first two sections teach how to think about software value.
The third part of this report introduces four techniques intended to help engineers evaluate and control the quality of their products with respect to these attributes. Such techniques fall on a two-dimensional spectrum, with one axis ranging from "qualitative" to "quantitative," and the other ranging from "useful early in the development cycle" to "useful late in the development cycle." These techniques complement one another, as each provides different software quality insights, which engineers can use in a feedback loop to help guide the product's development.

Finally, optimizing a product only goes so far in boosting the producer's competitiveness. Consequently, the last part of this report notes the importance of optimizing the producer (not just the individual product) and the software creation process. This final part also summarizes some active areas of research and provides an annotated bibliography. Engineers can make use of these tools in their quest to learn and improve—and succeed in their markets.

1.0 Microeconomics 101

1.1 Factor Exchange

Actors and Factors
The primary building blocks in microeconomics are economic "actors" and "factors." "Actors" can take a variety of forms: firms, individuals, labor unions, and government agencies. Actors consume "factors of production" and use them to generate products and services. In these terms, microeconomics is the study of how actors attempt to operate profitably in a competitive environment by converting factors into high-value products. Since the market for software products is exceedingly competitive, microeconomics provides several relevant principles. In short, microeconomics is the scaffolding that supports this report's discussion of software quality.

At the simplest level, factors of production fall into three categories: raw materials, labor, and capital. Different actors consume different factors to produce varied products. For example, a steel manufacturer consumes raw materials such as coal and iron ore, as well as labor by steelworkers and other types of staff, plus a great deal of capital including land and money. An insurance company consumes mostly office-related raw materials, such as paper and electricity, plus underwriter labor and capital equipment, including fax machines and computers. Microeconomics views individual humans (not just firms) as actors: they generate labor by combining raw materials like food with labor by barbers or other actors, as well as capital such as cars and homes. Generally, raw materials and labor cover the inanimate and animate inputs that scale with the amount of output, while capital covers the fixed costs that do not vary with the amount of output.

Actors convert factors of production into products and services and send them to other actors, who will treat these products and services as factors of production in order to generate more products and services.

---

1 Since the first part of this report reviews basic introductory microeconomics, the interested reader may refer to a wide range of college textbooks on the subject. These include (in increasing order of mathematical difficulty) [56], [49], [51], and [34]. The business-oriented [52] provides an extremely readable overview of how this material applies to information technology sectors. Because of the basic nature of this microeconomics review, in-text citations will be limited to situations where one of these references does an especially lucid job of explaining the point at hand.

2 An accountant would interject that capital is "depreciated" in the sense that it typically loses a little of its value over time. For instance, according to the US government, a computer printer loses its resale value in five years [2]. Consequently, from an accounting perspective, an actor would "use up" a fifth of its printer each year. Note, however, that this is independent of how much the actor actually uses the printer (and consumes raw materials such as paper and ink), which is the main point here.
Back-Flow
Transactions have two sides: When an actor consumes a factor of production, he typically will be stimulated or obligated to send something back to the supplier. The most obvious item that flows back to the producer is payment. The main goal of a supplier is to maximize the difference between payments flowing back into the supplier, and payments flowing back out of the supplier… in other words, to maximize profit.

However, producers often expect future non-monetary "back-flow" benefits from consumers, as well. These include prestige, new contacts, a bigger network of product users, and the opportunity to deepen customers' lock-in by encouraging entrenchment. Whereas explicit contracts typically govern monetary back-flow, these non-monetary benefits may accrue to the producer without any formal framework. They might flow back immediately when the factor is delivered, or they might be delayed in an unpredictable fashion.

The supplier has the opportunity to utilize these back-flows as a factor of production—parlaying them into additional production and sales. For instance, he can use monetary payments (in excess of production costs) to hire new R&D staff, and he can leverage prestigious customer relationships into marketing campaigns.

When a product demonstrates strong positive network effects (that is, its value increases with the number of other people using it), non-monetary back-flows can be extremely valuable [52]. In cases like these, positive feedback ensues, as more and more customers purchase the product, constantly generating network effects and prestige that facilitate additional sales. This contributes to the value of the firm itself, and many firms will act to maximize long-term value rather than instantaneous profit margins.

Many information-oriented products, such as operating systems and software suites, exhibit positive network effects. Consequently, actors producing products like these will often focus in the short-term on building a large installed base and non-monetary back-flow in addition to payment so that the firm may reap long-term gains.

3 Economists would not actually use “back-flow” as a generalized term for factors like direct payment and prestige which would encourage a supplier to seek a certain customer. Instead, they would write something like “(1) you will generate a large number of additional unit sales to other customers, (2) these sales will be at a high gross margin… (3) these effects will be long-lasting because of lock-in” [52]. A simple term like “back-flow” seems much preferable to such a wordy list.
1.2 Attributes and Functions

Factor Differentiation
Not all factors of production are created equal. For example, steel manufacturers generally attribute more value to ore with higher iron content. Moreover, they prefer raw materials that arrive on a carefully scheduled basis, rather than on a haphazard schedule; that is, the attribute of timeliness is of the essence when valuing iron ore shipments.

Each individual factor may have hundreds or thousands of attributes: purity, timeliness, price per unit, crate size, licensing restrictions, security, efficiency, and so forth. The attributes of labor factors are no less complex: skill set, performance, attitude, expectations for time off or health benefits, etc. Some attributes, such as a laborer's quadricep strength, have continuous values. Others, such as software licensing restrictions, tend to be much more discrete. Yet others, such nominal price per unit, generally take on discrete values (such as cents or hundredths of cents), but the values are so close together that actors can treat them as nearly continuous [46].

Often, multiple suppliers will generate similar products differing only in attributes. If a customer requires such a product, for use as a factor, he then has the pleasure of choosing which best meets his needs. Often, that decision will depend on the attributes of each supplier's product.4

Utility Functions
Of course, not all actors equally value all attributes. As the saying goes, "One man's trash is another's treasure." For example, a hospital may choose one operating system because it rates exceptionally well on the attribute of security, whereas a landscaper may prefer another operating system because it integrates well with the incumbent spreadsheet software. Sometimes, in order to appease such a wide variety of consumers, suppliers will generate a variety of products, differing from one another on attributes that matter. The result is a family of related products.

Each client firm places a limited amount of value on any given factor. This is reflected in the price it is willing to pay for a certain quantity of a particular factor. The graphic representation of this is called the "demand curve." If two factors can substitute for one another, but the client prefers one of the two because of superior attributes, then the superior factor's curve lays somewhat higher.

---

4 Back-flow has attributes, too. Multiple customers generate back-flow differing in attributes. For instance, not all payment plans deliver money at the same time. Likewise, not all prestige carries equal weight. Suppliers may assign varying values to such attributes and select customers accordingly.
Utility can be represented in another way, in terms of how much of one item an actor would willingly trade for another item. For example, perhaps a certain client would willingly trade fifteen apples for ten bananas because he believes bananas are more nutritious than apples. Note, however, that once he has his ten bananas, he might be unwilling to trade another fifteen apples for ten more bananas (he only needs so much potassium, after all). The graphical representation of this phenomenon is called an “indifference curve” [51].

Demand and indifference curves constitute two attempts to represent the complex multidimensional formula called the “utility function.” This conceptual tool expresses the total value (benefits minus price) of a bundle of goods to particular consumer.

If would be nice if utility had a simple functional form, such as a linear combination of “units of A” and “units of B” and so forth. However, as demonstrated by the graphs above, demand and indifference often exhibit curves—that is, non-linearities. Moreover, the utility function depends not just on factors, but also on factor attributes, and varies over time. Plus, every actor has a distinct utility function particular to that firm, laborer, or government agency. Consequently, there does not exist a clean formulaic representation of some perfectly generalized utility function.

Nonetheless, the utility function remains a useful conceptual tool for discussing how much an actor values factors. Groups of similar consumers can be aggregated for analysis purposes, yielding a picture of the target market segment’s utility function. The ATAM technique, discussed later in detail, provides an approach to characterizing this utility function.

Production Functions
An actor’s choice of input factors has a great deal of influence on the resulting product’s attributes. As another saying goes, “Garbage in, garbage out.” For example, if the medical facility mentioned above opted instead for the operating system with the strong spreadsheet integration attribute, hackers might take advantage of its poor security attribute—generating patient angst, lawsuits, and government penalties. The result is inferior service. To optimize their outputs, actors must consciously assess how their factors affect their products, then strive to acquire the most appropriate flows of inputs.
The “production function” captures some of these aspects related to choosing the right factors. If an actor fails to purchase the right input factors, it becomes impossible to produce a significant quantity of output. A firm’s production function is one of the main influencers of the corresponding utility function—that is, an actor tends to value the inputs that facilitate producing outputs.

Often, economists will model the production function with the “Cobb-Douglas” function [51]

\[
\text{Amount of output} = \text{constant} \times (\text{amount of input A})^a \times (\text{amount of input B})^b
\]

Here, 0 < α < 1 and 0 < β < 1 are called the “elasticities” of the two input factors involved. Intuitively, elasticity tells how efficiently a unit of input can be converted into a unit of output. Note that because elasticities are less than 1, the output cannot be doubled by simply doubling the amount of inputs. This represents the “law of declining returns”: after a certain point, adding more of a factor (or factor attribute) produces less and less marginal benefit.

For a given level of output, the actor has a choice: mix a large amount of A with a little of B, or vice versa, or some of each.

The canonical example is capital versus labor. The manufacturer can choose to provide low-tech work conditions and equipment, which means that generating N widgets would require very little in the way of capital but a great deal of labor. Conversely, the same manufacturer can buy automation equipment and generate the same number of widgets with more capital investment but fewer workers. Of course, the actor must make these choices in light of existing technology constraints. This complicates the picture, as does the fact that in the real world, the production function (like the utility function) depends not just on quantities of factors, but on those factors’ attributes.

When a supplier designs his product for use as a factor of production by client firms, the supplier must take into account the elasticity of that factor in the target market’s production function.
1.3 **Economic Choices**

**Choice of Inputs**
In short, as actors combine factors of production to generate products, they have choices about which inputs to utilize. Some choices involve "substitutable" inputs (e.g.: Pennsylvania and West Virginia coal), where two options fill nearly the same niche in the utility and production functions. Other choices involve partially substitutable inputs (e.g.: technology and labor), where two options fit related but not identical niches, and actors trade one input for the other, subject to elasticities.

More often than not, the actual choices selected will depend on the factors’ attributes—most importantly, their price per unit. When item A and item B are substitutes, then a rise in the price of item A will result in an increase in demand for item B. (Conversely, when item A and item B are complements, then a rise in the price of item A will result in a decrease in demand for item B.) Because the price of inputs impacts profitability, an actor will always choose the cheaper of two substitutable inputs if they are of otherwise "equal" value. More generally, an actor will generally select the inputs which best facilitate maximizing overall profit.

**Choice of Outputs**
Moreover, actors usually have some options for which products to generate. For instance, an importer/distributor will convert diesel fuel, labor, ships, and money into a variety of merchandise that he offers to retail stores. He could, for instance, choose to import tons of guns; or, he could import a lot of butter. He could do a combination of these. He has great control over which products to generate from the raw materials, labor, and capital.

He also can control the attributes of products. That is, he may import high-powered rifles, with attributes optimized for killing capability and range, instead of little 22-caliber pistols. Meanwhile, he must decide on his butter's salt/fat/carbohydrate content, number of sticks per box, and so forth. Plus, in all cases, he must decide on how to price, market, and sell his products.

Note that a high-powered rifle is a very different product than a pistol. Recall that consumers purchase on the basis of attributes because the attributes determine what is trash and what is treasure—in a sense, the attributes define the product. The product serves as a bundling of those attributes, and can be differentiated into a continuum of products by varying attributes.

Thus, the actor must choose which attributes—and, therefore, which products—to produce. This choice ties into the earlier choice of which input factors to select. For instance, importing rifles instead of pistols might involve dealing with a minefield of government regulations, implying higher labor costs. The attributes and quantities of the inputs, as well as the attributes and quantities of the outputs, must together satisfy certain constraints peculiar to the technology at hand. Indeed, the constraints typically vary from actor to actor within a certain market.

**Choice of Methodology**
Finally, the actor’s choice of factors and his choice of product are strongly interrelated with his choice of how to create the selected product from the selected factors. This last choice can affect the quantities and attributes of the inputs or outputs (e.g.: driving the ships faster may cost more fuel but less labor, and may result in enhanced attributes of timeliness).

More importantly, today's choice of how to produce may affect the constraints that govern future production. Possibly the clearest example is experience: if a firm consistently utilizes the same method of production, succeeding units of production can be generated using less labor. In addition, constraints on future production may "soften" if an actor creates reusable components during current production and those components find significant reuse in future work. (Perhaps staff experience is a special kind of reusable component, a type of functionality stored in the human brain.) By wisely choosing the methodology of today, an actor can decisively improve the internal constraints that govern the future.
To sum up, the actor must make three closely intertwined choices:
1. Selecting which inputs (including their attributes) to consume
2. Selecting which outputs (including their attributes) to produce
3. Selecting a methodology for converting inputs to outputs

Microeconomics is distinguished from macroeconomics in that it focuses on the choices that may be controlled by humans, firms, labor unions, government agencies, and other actors. In contrast, macroeconomics focuses on issues that individual actors cannot control: the current rate of inflation, the current rate of unemployment, the climate this time of year, and so forth. Such topics greatly impact policy choices put into effect by a large government, but controlling them remains well out of an actor's purview.

1.4 Economic Optimizations

Suppose that actor S supplies a certain factor f to an actor C, which typifies or represents a target market. Then, C combines f with a variety of other factors to generate a new product eventually consumed by another actor A.

The main question is, “What is the optimal strategy for S?” Based on a series of fairly sensible assumptions, one reasonable strategy for S is to generate a product yielding maximal utility to C.

**Optimize what?**
Before moving on to strategies, consider first what “optimal” means. Note that certain factors flow into S, though the diagram elides them, and S sends back-flow in response. The most important of these back-flow lines are those that involve money, just as the most important back-flows from C to S involve money.

Economists usually assume that actors attempt to maximize the difference between incoming and outgoing payments—in other words, the profits. It could be that in any particular case, some firm might attempt to optimize the difference between other types of back-flow; for example, it is probable that some actors focus on accumulating prestige, rather than profits. The standard economists’ response usually consists of something like the following [34]:

> The profit-maximisation [sic] assumption is of course much more sweeping than the corresponding assumption of utility maximization in consumer theory. However, though one might want to vary it to explain certain specific phenomena, it has proved remarkably successful in explaining and predicting responses to policy changes and other exogenous [external] changes in the economy. Though many other theories exist of the firm’s objectives (maximizing the utility of managers, etc.), space precludes us from treating them in this book.
In other words, in order to generate a reasonable and tractable problem, economists assume that firms focus on optimizing profits.

**Two roads to profitability**

Oversimplifying somewhat, profit equals incoming payments minus outgoing payments. (There are different approaches to accounting for the time-dimension of money, which the final sections of this report mention briefly.) Therefore, optimization involves either maximizing payments from C or minimizing costs—and preferably both.

There are two reasons why it often makes sense for an actor to focus on boosting incoming payments rather than shrinking costs:

- It is generally the case that S does not hold a complete monopoly. That is, other suppliers probably exist who could quite capably supply a comparable product. If S fails to focus on meeting C’s needs, C may eventually dispose of S and select another. Temporary effects such as lock-in may delay this process, but an actor must always focus on satisfying customers, or else they may slowly drift away.

- It is generally the case that C does not hold a complete monopsony. That is, other customers probably exist who could quite happily consume the actor’s product. In many markets (particularly industries based on selling information), the potential for growth greatly exceeds the current market penetration. In contrast, the potential for cost savings is only as large as the costs; it is essentially bounded. In such situations, it may make sense to try getting an ever-bigger piece of the pie, rather than to try eating a small piece using an ever-smaller fork.

Of course, fending off competition or acquiring new clients would cost money. So a firm must evaluate whether the potential payments from an expanded market share would cover and exceed the anticipated costs of expanding the market share [52]. It sometimes makes sense to let the competition have the customer.

**Two roads to maximizing incoming payments**

When it makes sense to pursue increasing payments, the natural question becomes “How?”

“There are an almost unlimited number of gradations in competitive conditions, ranging from pure competition to pure monopoly…. The vast group of markets, however, lie between these two extremes, and they blend gradually into each other” [56].

- If supplier S belongs to a purely competitive environment, then the goal is to provide a product that is useful for helping C to produce profitable outputs. In other words, f must help maximize C’s utility, production, and profitability functions. Per the arguments given above, boosting profitability involves boosting production of quality goods, which translates into maximizing C’s utility function.

- If supplier S holds a complete monopoly, then it will act to raise barriers to entry for other would-be competitors, perhaps by attempting to boost the lock-in of existing customers. Several strategies exist for doing this, but they are something of a stop-gap measure. “By and large, the key to obtaining superior financial performance in ‘lock-in’ markets is the same as in other markets; by product differentiation, offering something distinctly superior to what your rivals can offer, or by cost leadership, achieving superior efficiency” [52]. In other words, even in a monopolistic scenario, supplier S must still focus on providing a product with distinctively superior attributes (including cost).

On the basis of these considerations, the remainder of this report will generally assume that the most sensible strategy for a supplier S is to optimize its product f in order to maximize the utility function of its client C.
2.0 Application to Software Engineering

2.1 Software as a Valuable Factor

The role of software

One important piece of capital in any modern firm is software. This production input may include commercial off the shelf (COTS) software as well as custom code used to sew the COTS together. It can take the form of systems code, embedded code, application code, and web sites. The software might come as part of a bundle that includes hardware, services, or a combination of the two (such as T1 internet access with support for virtual private networking). The firm can purchase, license, or lease this software, or build it in-house. As a result, the software supplier might be another firm, consultants, the open source community, or a subsidiary or division. In short, firms have several choices along a variety of dimensions when selecting what software to acquire as an input factor. In light of these choices, as discussed above, software suppliers would do well to try optimizing their software products in order to provide maximal utility to the client firm.

In short, firms have several choices along a variety of dimensions when selecting what software to acquire as an input factor. In light of these choices, as discussed above, software suppliers would do well to try optimizing their software products in order to provide maximal utility to the client firm.

![Software Supplier Diagram]

The Software Supplier delivers only one factor to Firm A (who will be attempting to maximize the difference between its incoming back-flow and outgoing back-flow). Firm A may produce software, as well, using the input software as a component, or Firm A might utilize the input software to support production of a wholly different type of product (e.g., an asphalt recycler who relies internally on billing software).

Note, based on the figure above, that maximizing utility involves looking at the entire network of actors surrounding Client Firm A. For instance, if the software requires a great deal of memory or powerful CPUs, then A must purchase hardware from Other Suppliers; this would be reflected by a decrease in the software's contribution to A's utility function. Thus, "when we consider going to market with a product… we still need to consider the necessary relations to other players in the market. This is what is contained in 'the value network.' This network consists of customers, partners, suppliers and competitors. The value network always plays an important role, but for some products it is particularly critical" [20].

The value of software

Throughout the late 1900's, American corporations invested a significant amount of money in software and hardware. Surprisingly, research has shown that these firms did not, on average, reap significant profit increases as a result [25]. In fact, the average profitability of American firms appears to have been inversely correlated with investment in information technology.

Classic economics has a ready explanation for this: competition. Firms typically do not offer unique products or services. Instead, other firms are waiting in the wings, ready to provide a replacement. (Even if no competition currently exists, new competition will enter the market if the

---

5 Some authors, including [36], have argued that the real product of a software supplier is not software per se, but rather the intellectual property incarnated by that software. While this is half true, it neglects the truth that factors of production deliver value only in the context of actual usage. Therefore, the actual embodiment of the intellectual property as software matters.
potential profits appear to exceed the costs of starting up a new business.) As discussed in [52], only two things hold back the firm’s customers from switching to the competition: product attributes (including price) and switching costs:

- If the firm can offer a product that has compelling attributes, such as lower prices or higher quality, then it can differentiate itself from the competition and carve out a market niche.
- Or, if the customers must overcome some hurdles to switch (such as the hassle of finding a new firm, or lock-in to the existing firm), then the firm can delay customers from switching, giving it time to improve the attributes of its products.

Returning to the issue of software, American firms generally have invested in information technology that helped them produce more while consuming less. That is, the gross product margin (the amount of additional product generated per dollar of additional input factor) of firms has been positive, with respect to information technology. In fact, investing in information technology is even more effective at boosting production than is investing in other forms of capital or labor [25]. Put another way, software and hardware suppliers have delivered a great deal of value to firms.

However, firms have been unable to convert that value into profits. Apparently, competing firms have simultaneously made similar investments into technology, and few have succeeded in leveraging information technology investments into competitive differentiation. Thus, competition has forced firms to pass on most of the increased value to the end customers, who have reaped billions of dollars of value each year as a result [25].

As [52] explains in depth, firms’ profits depend on two parts:

- How much value they can efficiently deliver through their products
- What fraction of that value they can cash out

Empirically speaking, it appears that software contributes predominantly to the first of these, but not much to the second. Some firms have bucked this trend by using information technology to improve their product attributes (such as customer service, timeliness of delivery, reliability, and so forth) or to increase lock-in and entry costs (by filing patents, for example). However, on the whole, firms benefit from information technology because it helps them stay abreast of the competition by keeping productivity high and prices low.

This means firms seem to buy software that boosts their production function for a low price. Meanwhile, software suppliers typically face competition of their own. As a result, they must continually seek ways to deliver increased value to firms.

### 2.2 Software Attributes

The notion of “quality”

When a firm searches for an appropriate piece of software, it will typically have some requirements in mind. These range from vague expectations like “supports reporting for management” to precise constraints such as “generates pie charts in JPEG format from SQL statements and an ODBC-compliant database.” In general, the firm’s production function, as well as competitiveness and profitability considerations, drives these requirements.

“Software quality” has met with a variety of definitions over the past few decades, but one reasonable meaning is that attached by ISO: “the degree to which a set of inherent characteristics fulfills requirements” [3]. Researchers often refer to these “characteristics” as “factors” (as in [47]), but they will be referred to here as “attributes” (as in [16]) to avoid confusion with “factor of production” in the economics sense. Therefore, if a software supplier aims to improve software quality, differentiate its product, beat the competition, and thereby retain and grow the customer base, it must start by fulfilling requirements.
Note that this involves “requirements,” not “specifications,” per se. Suppliers of custom software recognize that most firms assign or reassign contracts largely based on whether or not the supplied software actually contributed to the firm’s production and profitability function—that is, not based on whether or not the supplier achieved the software specification. This is why the requirements gathering phase of development is so crucial to the software supplier’s success.

Even suppliers of off-the-shelf software must pay heed to the need for requirements gathering. This can take the form of meeting with existing client firms, who are selected to typify a market segment, and attempting to understand their work processes. The software supplier can then collaborate with those clients to design a better work process (thereby improving their production function) and extract the necessary requirements for software to support redesigned the new work paradigm [9], [36].

Put another way, the goal is not to simply deliver functionality; the goal is to deliver value. That was the main message of the first part of this report. Its application to software takes the form of optimizing software quality attributes.

**Multifarious attributes**

The requisite software attributes vary greatly by client firm. One attempt in 1979 to list the possible attributes included the following (discussed in [47]):

**Product Operation Attributes**
- **Correctness**: The extent to which a program satisfies its specification and fulfills the customer’s mission objectives.
- **Reliability**: The extent to which a program can be expected to perform its intended function with required precision.
- **Efficiency**: The amount of computing resources and code required by a program to perform its function.
- **Integrity**: Extent to which access to software or data by unauthorized persons can be controlled.
- **Usability**: Effort required to learn, operate, prepare input, and interpret output of a program.

**Product Revision Attributes**
- **Maintainability**: Effort required to locate and fix an error in a program.
- **Flexibility**: Effort required to modify an operational program.
- **Testability**: Effort required to test a program to ensure that it performs its intended function.

**Product Transition Attributes**
- **Portability**: Effort required to transfer the program from one hardware and/or software system environment to another.
- **Reusability**: Extent to which a program can be reused in other applications.
- **Interoperability**: Effort required to couple one system to another.

Each attribute presents a different face to each client. For example, as far back as 1968, many practitioners considered “maintainability” to include both repair of defects as well as addition of new functions [4]; in contrast, the list above breaks those into two attributes, “maintainability” and “flexibility”. So the word “maintainability” includes addition of new functionality to some people, but not to others.

Because of this, different organizations will give completely different lists than the one above for how to break down “quality” into attributes. For example, ISO 9001 lists only six attributes:
functionality, reliability, usability, efficiency, maintainability, and portability [47]. Other authors have chosen yet other lists [15].

More to the point, each client firm will think about quality attributes in a different way. For example, where one firm may see software as a collection of functionality, security, and maintainability attributes, another firm may see software as a collection of functionality, performance, attractiveness, and reliability attributes.

Moreover, each client firm will value varying attributes, and varying sub-attributes within those attributes. For example, although two firms may each include “usability” in their utility functions, one firm may value “operability” whereas the other could value “good training.” Each firm chooses software suppliers on the basis of how well their products support the attributes important to that specific firm.

In order to meet the differentiated needs of target markets, software engineers may produce a differentiated array of software products. For example, if one segment of the spreadsheet market demands high performance but less flexibility, but another segment demands high flexibility but can sacrifice performance, then perhaps two flavors of the same product would suffice. One version might omit macros and other advanced flexibility features, whereas the other would contain those features and the appropriate documentation. As noted in the first part of this report, firms have the choice of what to produce. When two closely related markets call for slightly different factors, producers can choose to provide a differentiated array of choices to meet this need. This way of thinking guides the engineer to ask, "Which of my customers value each attribute?"

When tradeoffs arise, the law of declining returns applies: A client firm will only trade so much of one attribute for another before they become unwilling to trade much more. For example, most clients will accept slightly less performance if it boosts the usability of the interface. However, few clients would want sophisticated speech recognition if it meant that the machine would virtually grind to a halt. Software producers must yield to these economic realities if they want to thrive.

To sum up, "software quality" involves more than capabilities. Rather, it covers a variety of non-functional (in addition to functional) concerns. The fundamental question that potential client asks is, “Is this software good enough on all of the important criteria, including cost, to justify a purchase?” Here, “good enough” and “important criteria” involve judgments particular to this specific target market [54].

### 2.3 Evaluation Techniques

Measuring these quality attributes requires making their definitions more precise. As such, good evaluation techniques each focus on a sub-attribute, rather than trying to capture the whole. These techniques are either qualitative or quantitative.

#### Qualitative versus quantitative evaluation

Qualitative evaluation techniques rely heavily upon subjective interpretation. Methodologies include meetings, code or interface reviews, and documentation. Outputs include text and diagrams, as well as action plans for how to refine the product. All software engineering processes must integrate qualitative evaluations to some degree (e.g.: Contextual Design, Use Case, etc.) during design and testing, simply because human minds must interface to the software at a qualitative level.

Quantitative evaluation techniques rely heavily upon formal measurement, “the process of assigning symbols, usually numbers, to represent an attribute of the entity of interest, by rule” [55]. In reality, quantitative measures (also called “metrics”) capture an aspect of the relevant attribute or sub-attribute. Methodology during design often involves representing the design in a diagram of boxes connected by lines, and then quantitatively characterizing the number of boxes
Methodology during development usually involves automated examination of source code and other artifacts to generate a numbers in scalar, vector, or matrix form.

Qualitative and quantitative evaluation techniques complement one another. For instance, qualitative and quantitative evaluations of the usability attribute focus on different but related issues. A qualitative evaluation might present a user interface to experts, asking them to identify “significant” problems with the application according to heuristic guidelines such as “minimize the user’s memory load” (as in [41]); of course, determining whether a particular guideline transgression is “significant” involves qualitative evaluation using the reviewer’s judgment. In contrast, a quantitative evaluation could present a task list to end users and empirically measure how long the average user requires to complete that list. Some studies, such as [39], combine both qualitative and quantitative approaches in an attempt to avoid missing any important issues.

Whether qualitative or quantitative, each evaluation generates an assessment of some aspect of an attribute or sub-attribute. These assessments can be represented as a vector (with one sub-attribute assessment per vector cell) and weighted against the utility function of the client firm or market segment [46]. Representing “quality” as a vector of attributes, rather than immediately collapsing to a single scalar “quality score,” can help keep different quality concerns separate as long as possible. This, in turn, may help the software supplier reason about how to tweak the software in light of an array of possible clients.

Quantitative evaluation: metrics and scale types
Some metrics are more valuable than others because they support more powerful analysis techniques. Each metric corresponds to a scale that falls into one of these categories [55]:

<table>
<thead>
<tr>
<th>Scale Type</th>
<th>Example</th>
<th>Valid Uses / Tests / Comparisons</th>
<th>Transformations that Result in New Scale of the Same Scale Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nominal</td>
<td>Assigning papers to reviewers (e.g.: labeling papers with reviewer names “Al”, “Bob”, or “Carly”)</td>
<td>Testing for equality (e.g.: determining if two papers should go to the same reviewer)</td>
<td>Permutation (e.g.: redefining the labels “Al” and “Bob” so Dr. Bob gets Dr. Al’s papers, and vice versa)</td>
</tr>
<tr>
<td>Ordinal</td>
<td>Assigning letter grades to exam essays (e.g.: with letters in the ordered set &lt;“F”, “D”, “C”, “B”, “A”&gt;, which is sorted by increasing value)</td>
<td>Testing for less-than-or-equal-to (e.g.: determining whether one essay was better than another)</td>
<td>Monotonic increasing mappings (e.g.: changing all “A” to “A+”, “B” to “B+”, “C” to “C+”, and so on)</td>
</tr>
<tr>
<td>Interval</td>
<td>Assigning temperature in conventional units (e.g.: Fahrenheit, which uses 0 to mean something above absolute zero)</td>
<td>Testing for absolute amount of difference (e.g.: determining that today is ten degrees warmer than yesterday)</td>
<td>General linear group $X_{new}(x) = mx + b$ (e.g.: converting to Celsius)</td>
</tr>
<tr>
<td>Ratio</td>
<td>Assigning a number to objects to indicate weight (e.g.: in pounds)</td>
<td>Testing for ratios between values (e.g.: determining that the keyboard weighs twice the mouse)</td>
<td>Similarity group, $X_{new}(x) = mx$ (e.g.: converting to kilograms)</td>
</tr>
</tbody>
</table>

---

Note that the task list might vary significantly by application, even within the same domain, so this quantitative evaluation is not founded on a particularly generalizable metric. That is, the scope of its validity and usefulness would be fairly circumscribed, as will be discussed shortly.
The “representational approach” to measurement theory views each scale as a set of symbols assigned to real-world phenomena. These symbols have relationships to one another, in terms of what comparisons may be made between symbols from the same scale, as shown above. This means that the scales are mathematical devices with certain internal relationships (similar to rings and Abelian groups) but are not real themselves, per se. As one author puts it [17]

The representational approach to measurement avoids the temptation to define a poorly understood, but intuitively recognizable, attribute in terms of some numerical assignment. This is one of the most common failings in software measurement.

For example, a later section of this report discusses various “complexity metrics.” These evaluate some structural characteristics of software code (such as the number of loops and branches) to generate a numerical assignment called the “complexity.” But the savvy reader will keep in mind that this is just a number: a representation, not a reality. Although counting the number of branches in source code helps characterize its complexity, it omits other issues that contribute to code complexity (such as the number of sub-module dependencies it comprises).

As noted in the table above, not all scales are equally powerful in terms of the number of comparisons they support. Moreover, they differ in the higher-level constructs that they support.

For example, while it might make sense to compute the arithmetic mean of measurements on an interval scale (e.g,: mean daily high temperature in Portland, OR, for 2004), the arithmetic mean is not a sensible construct over an ordinal (or nominal) scale [17]. The reason: the “distance” between points on an ordinal scale might be non-linear. So averaging three essay grades {“F”, “F”, “B”} could yield “F”, “D”, “C”, or “B”, depending on rules peculiar to the scale—it depends on how low an “F” really is relative to a “B”. That is, ordinal scales simply are not powerful enough to support arithmetic mean computations (though they do support median computations), so it is necessary to refer to the underlying reality: How bad does an essay have to be to get an “F”?

Note that each scale type corresponds to an infinite set of measures, and each listed set is a superset of the sets shown above it. So, for example, ratio scales can be used to describe equality relationships, less-than-or-equal-to relationships, amounts of difference, and ratios. Thus, a ratio measure is more powerful than the other types of measures because it can be used to make more types of comparisons. For these reasons, all else being equal, evaluators prefer ratio over interval scales, and interval over ordinal, and so forth.

Quantitative evaluation: metrics and validity

Metrics purport to measure a sub-attribute. Therefore, a metric’s value depends heavily on how well the metric measures that sub-attribute—in other words, the metric’s validity.

The validity of a metric can be characterized at least six ways if the corresponding sub-attribute can be measured precisely on some ratio scale (for which this metric serves as an approximation). For example, if \(a_k\) represents the exact value of an attribute for some \(N\) pieces of software (\(1 \leq k \leq N\)), and \(x_k\) represents the metric’s value for the \(k^{th}\) piece of software, then good metrics possess the following characteristics [50]:

- **Association**
  - The square of the correlation coefficient \(R\) between \(\{a_k\}\) and \(\{x_k\}\) must exceed some threshold.

- **Consistency**
  - Ranking each product according to \(a_k\) must produce nearly the same ordering as would ranking each product according to \(x_k\).

- **Discriminative power**
  - For any “critical value” \(a_c\), there exists some \(x_c\) such that for any product \(k, x_k > x_c\) if and only if \(a_k > a_c\).
Repeatability  Measuring $x_k$ multiple times must generate the same value most of the time (that is, at least a certain fraction of the time).

Tracking  If $a_k$ changes with time, then $x_k$ must also, and the sign of their derivatives (with respect to time) must always match.

Predictability  There must be a function $f$ such that if $x_k$ is measured at time $T$ to predict $a_k$ at some later $T'$, then $\text{abs}[f(x_k(T)) - a_k(T')] / \text{abs}[a_k(T')]$ cannot exceed some positive threshold.

In short, “metric validity” means that the metric measures what it purports to measure (sometimes called “internal validity”), and that it is transferable to situations outside the calibration set (sometimes called “external validity”). In this sense, the tests listed above help to evaluate internal validity. Other means of establishing validity exist, as well. For example, [58] proposes a fairly different set of six evaluation criteria.

Invalid metrics have little value to a software development organization. The real goal of a software firm is to produce valuable software, not collect metrics. Metrics are only useful if they actually track the underlying attribute that has real economic value. Focusing on the goal of the development process—delivering value—demands that engineers look with a critical eye at the validity of proposed metrics.

Of course, as noted earlier, it is generally impossible to quantify an attribute’s “exact value” for every product. Consequently, researchers will generally select some set of products and attempt to validate the proposed metric using that limited set of products.

Quantitative evaluation: metrics and appropriateness
Finally, metrics differ in their practical usefulness due to their appropriateness in specific situations.

One consideration is the point in the development cycle when a metric becomes valid. For example, [53] argues that if architects had a reliable way to evaluate the value of designs, then they could generate a suite of related designs from a set of requirements, then evaluate those designs before ever writing code. That way, the organization could improve its chances of implementing the optimal product (in terms of maximizing value to client firms), potentially preventing a great deal of unnecessary redesign and recoding during development. On this score, design-time evaluation techniques offer a good deal more value than development-time metrics that require actual code.

Another consideration is the metric’s appropriateness to the application domain at hand. The “function point” (FP) family of metrics exemplifies this issue. Specifically, at the most basic level, the total number of function points in an application or specification can be counted by adding up the following [47]:

<table>
<thead>
<tr>
<th>Architectural Element</th>
<th>Simple Instance</th>
<th>Average Instance</th>
<th>Complex Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query tools for user</td>
<td>3 points</td>
<td>4 points</td>
<td>6 points</td>
</tr>
<tr>
<td>Other input screens from user</td>
<td>3 points</td>
<td>4 points</td>
<td>6 points</td>
</tr>
<tr>
<td>Other output screens to user</td>
<td>4 points</td>
<td>5 points</td>
<td>7 points</td>
</tr>
<tr>
<td>External machine interfaces</td>
<td>5 points</td>
<td>7 points</td>
<td>10 points</td>
</tr>
<tr>
<td>Logical groups of files or database tables</td>
<td>7 points</td>
<td>10 points</td>
<td>15 points</td>
</tr>
</tbody>
</table>

Using this metric, a program that presented two simple input screens to gather information and then generate one complex report would contain $2\times3 + 1\times7 = 13$ FP.

This “unnormalized” FP metric belongs to a diversified family of similar but distinct measures of functionality. Each FP innovation over the past twenty years has added or tweaked some detail.
of the metric, such as “normalizing” to take into account other issues such as “Is performance
critical?” and “Are data communications required?” Other FP metrics, such as the “feature point”
metric, take into account algorithmic complexities common in embedded programming—versus
the data-oriented and interface-centric issues represented in the FP metric demonstrated above,
which occur commonly in business programming.

In short, the range of functionality metrics has become specialized to each application domain.
As noted in the first part of the report, all firms—including software development organizations—
have a choice of which methodology to use in producing a product. Choosing an FP metric is
one aspect of choosing a production methodology, so an engineer who needs an FP metric will
put effort into learning about their differences. That way, he can choose the one most appropriate
for the application domain at hand. For surveys of the available FP metrics, consult [47], [29],
and [1].

Quantitative evaluation: No perfect metric

In closing, it should be noted that many researchers have longed for a purely objective way to
characterize each quality attribute, or the product’s quality as a whole. At the same time, most
researchers generally realize that such endeavors are impossible for a variety of reasons.

First, no single simple metric captures all the nuances of any given attribute. For example,
researchers generally believe [47] that the “flexibility” of software owes to several factors: code
complexity, code conciseness, modularity, self-documentation, and so forth. In any real software
development project, these sub-attributes compete against one another. (For instance, using
verbose variable names boosts self-documentation but reduces code conciseness.) Thus,
although it is conceivable that each of these sub-attributes could be individually measured by
independent metrics, no one metric could capture the entire semantics of “flexibility.”

Moreover, at a technical level, combining simple sub-attributes metrics into a composite attribute
metric presents its own challenges. How would these individual metrics combine to form a
composite metric—linearly? Multiplicatively? In addition, how would a researcher validate such a
composite metric in terms of association, consistency, tracking, etc, when the underlying attribute
(e.g.: “flexibility”) is inherently immeasurable?

Even more importantly, metrics generally capture only “internal” aspects, whereas attributes
include “external” nuances sensitive to a particular context [58]. For example, longer programs
generally demonstrate less maintainability than do shorter programs. Counting logical lines of
code has proven to be a fairly useful way to measure program length, despite certain (somewhat
language-variable) ambiguities like whether to include variable declaration lines [29]. Such a
metric captures information that is purely internal to the program’s source code—requiring no
external data to evaluate. In contrast, to discuss a program’s maintainability requires first
agreeing which programmer will perform the maintenance, including external details such as
whether the programmer ever saw the code before, which languages his skill set includes, how
many years he has programmed, and so forth. Since virtually all attributes of interest involve
some external concerns, no metric can ever capture all the nuances of any attribute.

Finally, even if a metric were totally objective, applying it would require some degree of
subjectivity. For example, even the simple “unnormalized” FP metric given above requires
discerning whether each architectural element is “simple,” “average,” or “complex,” based on
guidelines given by the metric’s designer. Indeed, even selecting the right FP metric from the
plethora of available choices requires some consideration based on experience and expertise.
Choosing and using a metric involves subjective considerations. These are unavoidable, since
the real problem exists “out there” in the messy, subjective, real world [27], not in some clean
abstraction, and it must be mapped using analog brains from the analog world into the digital, mathematical world of software.\footnote{Even this mildly pessimistic view glosses over the role of the messy, subjective implementation process! The overarching goal is to deliver maximal value to the customer. Heretofore, this report has focused on using metrics to evaluate product quality attributes. However, the management of each project, as well as the process by which an organization creates software, both help deliver value to the customer (albeit indirectly). As discussed later, metrics also constitute a crucial tool for optimizing the project and process aspects of delivering value. [1050], [61] and [36] consider the messiness of this challenge in detail. }

 Nonetheless, researchers still persist in, and still succeed in, identifying metrics that reasonably approximate attributes in useful ways. This facilitates representing each product’s quality as a vector of attribute scores. When combined with an understanding of how each attribute contributes to a target market’s economic utility function, metrics provide a powerful tool for building the right product.

\textbf{2.4 Example: An Attribute Evaluation Gone Wrong}

The discussion above has introduced the concept of breaking quality into a variety of attributes, and these into sub-attributes, and attempting to evaluate software sub-attributes using qualitative and quantitative techniques. This section introduces one paper, [39], which attempted to develop a single metric capturing the full meaning of a particular quality attribute, “usability.” Any attempt to develop such a metric is fraught with dangers, as exemplified below, owing to the limitations in what can be achieved with metrics.

After looking at what metrics were not meant to do, the following part of this report will turn to four evaluation techniques which, in contrast, have met with some success in helping software suppliers optimize their products to maximize utility to client firms.

\textbf{Study overview}

Evaluators should feel great trepidation when combining metrics from varying scales, as when one usability study took a linear combination of six terms in order to create a composite metric purporting to measure the total usability of a diagnosis expert system called “Function Identifier.” (Diagnosis expert systems are a class of software that provide a diagnosis based on a set of symptoms; the user needs to feed in some or all of the symptoms until the system homes in on a diagnosis.) The researcher presented the system, along with a pre-defined set of symptoms, to each user. Each user then fed symptoms to the system and attempted to find the corresponding set of diagnoses.
Here were the six terms in the composite metric:

<table>
<thead>
<tr>
<th>Term</th>
<th>Scale Type</th>
<th>Data Source</th>
<th>Data Source Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confidence</td>
<td>Ordinal</td>
<td>Each user’s rating of his confidence on a 7-point scale that the system generated accurate diagnoses</td>
<td>Qualitative User Preference</td>
</tr>
<tr>
<td>Difficulty</td>
<td>Ordinal</td>
<td>Each user’s rating on a 7-point scale of how hard using the system seemed to be</td>
<td>Qualitative User Preference</td>
</tr>
<tr>
<td>Correctness</td>
<td>Ratio</td>
<td>The proportion of times that the system’s diagnoses matched “correct” diagnoses defined by human experts</td>
<td>Semi-Quantitative User Performance</td>
</tr>
<tr>
<td>Iterations</td>
<td>Ratio or Interval</td>
<td>Average number of symptoms (per diagnosis) that the user had to feed in</td>
<td>Quantitative User Performance</td>
</tr>
<tr>
<td>Inability to Answer</td>
<td>Ratio</td>
<td>The proportion of times that the system was unable to come up with any diagnosis</td>
<td>Quantitative User Performance</td>
</tr>
<tr>
<td>Help Requests</td>
<td>Ratio</td>
<td>The frequency that each user needed to access the help / documentation screens</td>
<td>Quantitative User Performance</td>
</tr>
</tbody>
</table>

The author combined these independent measures using a linear combination, with positive or negative weights chosen by usability experts and normalized so that the resulting metric ranged from 0 through 1. After the system was tested by users, the author plugged those measurements into the formula to generate a mean score of 0.75 for this expert system. The paper concludes, “This score exceeds the scale’s midpoint value. Therefore, with respect to the six usability variables considered here and the normalized usability range [0,1], Function Identifier is a relatively usable system.”

Dangers exemplified
This approach excels at demonstrating some dangers inherent to creating composite metrics.

First, the author’s stated goal in this paper is to create a single usability metric which combines two main attributes: user preference and user performance; the six terms above essentially constitute sub-attributes of these two main attributes. However, a linear combination essentially assumes that the client firm or market segment equally values some X “units” of user preference for Y “units” of user performance. That is, linear combinations completely ignore the issue of declining returns discussed in the first part of this report.

Second, as argued passionately in [46], there is great value in keeping attributes separate (rather than immediately creating a unified index function) any time that attributes cannot be easily traded for one another. This is clearly the case here; there is no means by which the system developers could “convert” all these attributes into one another. For example, how could the system’s inability to answer be traded off against the number of help screen requests? The first is essentially an internal reliability issue, while the second may largely entail user interface concerns. If they are as independent as they appear, then the system developers should independently optimize these metrics, rather than performing a joint optimization as implied by the author’s problem formulation. In economic terms, if two attributes are not related by a tradeoff, and each contributes to the utility function, then they should both be maximized.

In contrast, it might make sense to perform a joint optimization between the “correctness” and “inability to answer” attributes. It is conceivable that the designers could improve the correctness
by making the system only give an answer when it is very “sure” of the diagnosis. However, this might exacerbate its inability to answer. This is a common tradeoff in machine learning [38], and in such a case, the author would have an easier time arguing for immediately combining the attributes, in order to facilitate a joint optimization.

Third, researchers must always take care when combining an ordinal scale measure with any other measure, since it is usually not immediately clear how far apart ordinal “steps” are, in a quantitative sense. So if A represents a numeric ordinal scale and B represents another numeric scale, then the functional form of the “right” combination f(A,B) is not immediately obvious. Unfortunately, although the author does attempt to take pains to give each sub-attribute “equal weight,” each ordinal scale is assumed to have linear steps between adjacent values, and each ordinal scale is assumed to vary linearly with respect to the other scales in terms of contribution to total usability.8

Finally, the author concludes that the usability score of 0.75 indicates a “relatively usable” system. However, it appears that the metric had never been applied to comparable expert systems—perhaps if it had been, all other systems would have scored 0.9!—so it is hard to know whether 0.75 indeed represents a “relatively usable” system. Indeed, the fact that this metric has not been applied to other systems makes it impossible to directly assess its validity at all.

Perhaps one way to establish internal validity would be to collect this metric on a dozen systems, and also to track and categorize post-release bug reports (or source code control changes) on the systems. The author could then attempt to show that the composite metric correlated well with the amount of usability enhancement required after release. This would be worthwhile if usability was a significant contributor to client firms’ production or utility functions. At any rate, establishing external validity requires using the metric on a broad range of interesting expert systems—not just one.

In short, the main weaknesses of this evaluation relate to inappropriate scale usage and a lack of broad-based validation. Together, scale rules and validation requirements provide constraints on what can be reasonably expected from a metric.

3.0 Controlling Software Quality

Economic actors value factors according to their attributes, so software suppliers must control those attributes in order to maximize clients’ utility functions. Evaluation techniques provide a means of tracking and ultimately enhancing those attributes, subject to constraints governing the joint optimization of the target market segment’s utility function.

The previous part of this report looked at one wrong way to evaluate a quality attribute. This part will introduce four superior evaluation techniques. These range from qualitative to quantitative, and their usefulness ranges from early to late in the development cycle.

3.1 Example: Qualitative Techniques Early in the Project – ATAM

Carnegie Mellon University’s prestigious Software Engineering Institute (SEI) has developed the Architecture Tradeoff Analysis Method (ATAM) as a tool to help software engineers understand and manage the tradeoffs inherent in a software product. In particular, the method focuses on performance, availability, security, and modifiability attributes [31].

---

8 There are ways to convert an ordinal scale into an interval or ratio scale, such as plotting out a histogram of ordinal scores, and then converting each ordinal score into a percentile rank. However, this still leaves open the question of how to weight these measures with respect to one another, and what functional form to use for the composite metric.
**Scenarios and utility tree**

One of the first steps to using the ATAM is meeting with stakeholders to understand the business considerations providing the impetus to the software development. The goal is to identify, as precisely as possible, the specific quality attributes essential for success. ATAM’s designers write, “One of the positive consequences of using the ATAM that we have observed is a clarification and concretization of quality attribute requirements. This is achieved in part by eliciting scenarios from the stakeholders that clearly state the quality attribute requirements.”

Scenario descriptions help flesh out the requirements. Engineers abstract these to create use cases, which they then refine further in additional stakeholder meetings. Later, stakeholders cast votes to judge how essential these scenarios are to their business processes; this yields scenario priorities. In short, these use cases provide a clear picture of the customers’ production process—that is, a qualitative understanding of the corresponding economic production function.

Moreover, the stakeholders work together to generate a utility tree. This diagram facilitates breaking down the broad-brush attributes into sub-attributes to an arbitrary level of detail. The customer representatives assign each leaf a utility score (e.g.: “Low”/“Medium”/“High”), and the engineering representatives assign each leaf a difficulty/risk score. (An M,L would mean medium utility and low difficulty, for example.) Optionally, branches may be assigned weights, as well. Through this tree, the customer scores provide an extremely fine-grained picture of how value will be assessed—that is, a qualitative understanding of the corresponding economic utility function.

For example, [31] provides the following tree (which the engineer is half-finished fleshing out):

```
Performance
  Performance
    Data Latency
      (M,L) Minimize storage latency on customer DB to 200 ms.
    Transaction Throughput
      (H,M) Deliver video in real time

Modifiability
  Modifiability
    New product categories
    Change COTS
      (L,H) Add CORBA middleware in < 20 person-months
    Change web user interface in < 4 person weeks

Availability
  Availability
    H/W failure
    COTS S/W failures
      (L,H) Power outage at Site 1 requires traffic re-direct to Site 2 in < 3 secs
    (M,M) Restart after disk failure in < 5 mins
    (H,M) Network failure is detected and recovered in < 1.5 mins

Security
  Security
    Data confidentiality
    Data integrity
      (L,H) Credit card transactions are secure 99.999% of time
      (L,H) Customer database authorization works 99.999% of time
```
Designing
At this point, the architects create multiple designs for the system. Each design may have a different style—which components are involved, and how they interact with one another. The architects “run the scenarios on paper” against each proposed design, assessing how well the design fares. They might perform formal quantitative analyses, such as queuing analysis, to understand how well each design meets specific requirements.

They assign higher qualitative scores to designs which adequately satisfy the highest priority scenarios and the most valued leaves of the utility tree. Moreover, they can use the utility tree to guide iterative refinements of designs until one best design stands out. Often, satisfying a leaf of the tree will crucially depend on some technical feature; when architects encounter such “sensitivity points,” they incorporate the feature into their design.

The utility tree provides crucial guidance when designers face tradeoff points. For example, satisfying the requirement “network failure is detected and recovered in < 1.5 mins” can be facilitated by full-time load balancing: constantly running two servers in separate cities, and dynamically directing end user requests to the server currently experiencing the lowest latency. However, taking this approach may interfere with achieving the requirement “credit card transactions are secure 99.999% of time” because leaving both servers running at all times inherently creates a second point of access that must be secured. (The SEI has written specifically about this particular tradeoff, and many others [16].) According to the utility tree above, the client values recovery more than security (“Medium” versus “Low”), so the architect would probably seriously consider full-time load balancing in this case. When the utility tree fails to provide adequate discernment, the use case priorities may suffice for guiding this decision.

As with all good requirements gathering processes, one of the best features of the ATAM is its facilitation of communication and understanding between the engineers and the customer. So when the utility tree and use case priorities provide inadequate guidance, the designers can fall back on their relationship with the client to collect additional data. This relationship pays other dividends, and not just financial ones: Good customer relationships are key to parlaying existing sales into non-monetary forms of back-flow, including product testimonials, exposure to more potential clients, and partnership opportunities. The value of communication is not to be taken lightly.

As with all good requirements gathering processes, one of the best features of the ATAM is its facilitation of communication and understanding between the engineers and the customer. So when the utility tree and use case priorities provide inadequate guidance, the designers can fall back on their relationship with the client to collect additional data. This relationship pays other dividends, and not just financial ones: Good customer relationships are key to parlaying existing sales into non-monetary forms of back-flow, including product testimonials, exposure to more potential clients, and partnership opportunities. The value of communication is not to be taken lightly.

Gathering final consensus
Now the architects have completed their work. They have identified and understood the sensitivity points of each high-value attribute, and have assessed the tradeoff points among attributes. The sensitivity points’ technical features have become part of various designs. Using the utility tree and scenarios, the architects has judged these designs and selected the best.

At this point, all the stakeholders meet again and discuss the proposed design. This gives the architects a chance to explain what they see as the most salient sensitivity points and how the proposed system meets those needs. New scenarios might be proposed to test and refine the design to further optimize its appropriateness for the client's utility and production functions.

The engineering team must also explain how their perceptions of the implementation risks/difficulties have developed so that the stakeholders can plan how to ameliorate those uncertainties. For example, the project plan might incorporate an early experimental phase to help flesh out any remaining questions about how to structure the software product.
Finally, the designers tweak the design further, if necessary, and complete its documentation. Outputs include the following:

- Diagrams, text, and analysis elucidating the architectural style
- A rationale for why this design deals with sensitivity and tradeoff points as it does
- An identification of the risks and a plan for dealing with those risks
- Schedule information

Related techniques
Researchers have extended and refined ATAM in several ways.

- The Quality Attribute Workshop (QAW) is a step-by-step specification of how to run the scenario-gathering phase [7]. It considers both system as well as software quality attributes and takes into account the fact that one or more stakeholders might already have some ideas about how the system architecture should be structured. The QAW specification’s main value seems to lay in providing guidelines to a manager who feels uncomfortable with the nuts-and-bolts of how to run the meeting.
- The Architecture Based Design method (also called “Attribute-Driven Design” or ADD) “provides structure in producing the conceptual architecture of a system” [6]. The first step is to identify the “architectural drivers,” which are essentially the sensitivity points mentioned earlier. The method then guides designers through the process of decomposing functionality according to the architectural style which best delivers on the most valuable attributes. Design elements may follow templates—something of a cross between prototypes and traditional types—which can conceivably be reused from project to project and within the same project. The method proceeds recursively in a top-down fashion, and it may involve iterations of further requirements gathering through additional QAW sessions [43].
- “The ATAM approach… is representative of a useful but undeniably qualitative approach to design analysis” [47]. The Cost Benefit Analysis Method (CBAM) focuses on better quantifying the return on investment for the customer [30]. Specifically, it advocates gathering more precise requirements in terms of the best, desired, current, expected, and worst cases from the customer’s standpoint. Using these requirements, the stakeholders assign quantitative weights to each scenario (in other words, how many “points” a design will score for satisfying that requirement at the best, desired, and worst case levels). Moreover, the stakeholders look at each possible risk and assign weights to those risks (how many points a design will lose if the risk bears fruit and negative consequences result). These numerical assessments provide designers with quantified criteria for evaluating designs: If a design satisfies a scenario to a certain level with a certain probability, then it scores a certain number of points, weighted by the corresponding probability. If a design runs a certain risk at a certain probability, then it loses the relevant number of points, weighted by the corresponding probability. Choosing a design involves maximizing the number of expected points.
- In essence, the CBAM constitutes a single approach to better quantifying the attribute-driven design process. Other approaches and extensions exist, such as [5], which delineates two conceptual spaces: a problem space dimensioned according to functional and non-functional requirements, and a solution space dimensioned according to structural aspects of designs. The goal is then to identify the potentially mutually exclusive structural aspects, yielding constraints, and then to perform a CBAM-like optimization process to determine the design which yields a maximal score. The authors view the best and worst designs’ scores as delimiting a spectrum of potential solutions; each solution can then be ranked on this spectrum.

To sum up, ATAM provides an essentially qualitative methodology for understanding customers’ production and utility functions. Some refinements achieve a certain level of quantification by estimating weights and probabilities, typically with heavy input by the client firm.
3.2 Example: Qualitative Techniques Late in the Project – Heuristic Evaluation

Qualitative techniques are useful for controlling software quality attributes later in development, as well. For example, qualitative techniques are a powerful tool for identifying usability defects—problems with the user interface's expression of the software's functionality. However, it turns out that, as with all testing, it is essential to trade off the improvement in this attribute with overall development cost (and, therefore, the price attribute of the resulting software), since both influence the clients' utility function.

Heuristic evaluation

Testing usually involves running a system and seeing if it performs properly. Specifications typically do not precisely indicate the exact layout and behavior of specific user interface elements, so precisely defining whether it “performs properly” can be somewhat difficult. Some evaluation is needed during the development process to identify places where the specification and implementation could be enhanced.

One way to deal with this situation is to have a usability expert evaluate a system against a list of general rules commonly believed to capture the essence of what makes software useful. Examples include [41]

- Speak the user’s language
- Minimize user memory load
- Be consistent
- Provide feedback
- Provide clearly marked exits
- Provide shortcuts
- Good error messages
- Simple and natural dialog
- Prevent errors

A usability expert identifies instances of these generalized problem types. For example, a popup labeled “Error 154F7: i-node referent does not exist” exemplifies at least three of the rules above and would hopefully be flagged by a usability expert: Error messages like these provide nearly zero utility to the user. Because these qualitative rules are known as “heuristics,” the process is called “heuristic evaluation.”

Even the experts make mistakes

Interestingly, each expert typically misses most usability problems. Indeed, across a wide spectrum of studies, researchers find that each expert only finds 19 to 51 percent of all usability problems [60], [41], [42].

That is, if a software development organization took a user interface implementation and hired an infinite number of evaluators to look at it, they would together find \( N_{\text{total}} \) problems with the user interface. Some of these problems would be find by multiple evaluators, but each individual interviewer would only find a fraction \( \lambda \) of the total number of problems (where \( \lambda \) is somewhere between 0.19 and 0.51 or so).

It turns out empirically that if \( N_{\text{found}}(k) \) represents the total number of problems found by evaluators 1 through \( k \), then \( N_{\text{found}}(k) \) roughly tracks the curve

\[
N_{\text{found}}(k) = N_{\text{total}} \times (1 - (1 - \lambda)^k)
\]

\( N_{\text{total}} \) and \( \lambda \) are parameters that vary by project.
Projecting the discovery rate

The software development organization can estimate $N_{\text{total}}$ and $\lambda$ by letting three or four experts independently perform heuristic evaluation, and then fitting $N_{\text{total}}$ and $\lambda$.

The fit should be performed independently of the particular order in which these heuristic evaluations actually occurred, but instead should average over the possible permutations of testers. For example, suppose that the following defect discovery pattern occurred with three testers:

<table>
<thead>
<tr>
<th></th>
<th>Problem A</th>
<th>Problem B</th>
<th>Problem C</th>
<th>Problem D</th>
<th>Problem E</th>
<th># Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tester 1</td>
<td>Found</td>
<td>Found</td>
<td>Found</td>
<td>Found</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Tester 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Found</td>
<td>4</td>
</tr>
<tr>
<td>Tester 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Found</td>
<td>1</td>
</tr>
<tr>
<td>Estimate of $N_{\text{found}}(1)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.67</td>
</tr>
</tbody>
</table>

Here, the approximate number of errors that one tester would find is 2.67.

Next, consider what would happen if two testers had participated, one right after the other:

<table>
<thead>
<tr>
<th></th>
<th>Problem A</th>
<th>Problem B</th>
<th>Problem C</th>
<th>Problem D</th>
<th>Problem E</th>
<th># Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testers 1+2</td>
<td>Found</td>
<td>Found</td>
<td>Found</td>
<td>Found</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Testers 2+3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Found</td>
<td>4</td>
</tr>
<tr>
<td>Testers 3+1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Found</td>
<td>5</td>
</tr>
<tr>
<td>Estimate of $N_{\text{found}}(2)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.33</td>
</tr>
</tbody>
</table>

So using two testers would have probably found approximately 4.33 problems, or $N_{\text{found}}(2) = 4.33$. 
Finally, all three testers of course found a total of 5 problems, yielding \( N_{\text{found}}(3) = 5 \). These three estimates of \( N_{\text{found}}(k) \) could be plotted on a graph and fitted using nonlinear least squares to generate \( N_{\text{total}} \) and \( \lambda \).

This functional form represents a Poisson distribution, which can be derived from a few basic assumptions. The most important assumption is that “the probability of finding any given usability problem in any given test is independent of the outcome of previous tests” [42]. This important assumption will play a further role in the discussion later.

How valid is this? Numerous studies have confirmed that this curve accurately represents the discovery rate of usability problems by expert reviewers. Indeed, although the exact \( N_{\text{total}} \) and \( \lambda \) vary (because not all user interfaces are identical, and not all pools of expert evaluators are equally skilled), the statistical correlation coefficient \( R^2 \) for the function fit is consistently above 0.9, even for sets with large numbers of testers.

When is enough “enough”? Enter economics. Since hiring expert evaluators costs a good deal of money, one study argues that a certain point, the probability of finding future problems becomes too low to justify additional investment in heuristic evaluation [42]. For example, if performing a battery of heuristic evaluations presents a fixed cost of \( h_f \) plus a sliding cost of \( h_c \) for each of \( k \) evaluations, then the total cost of the battery is:

\[
\text{Total cost } C = h_f + h_c \cdot k
\]

These \( k \) evaluators would be expected to find \( N_{\text{total}} \cdot (1 - (1 - \lambda)^k) \) problems. If solving each problem translates into a value \( V \) for each problem found, then the total benefit of the battery is:

\[
\text{Total benefit } B = N_{\text{total}} \cdot (1 - (1 - \lambda)^k) \cdot V
\]

For low \( k \), \( B \) rises more steeply than \( C \). However, at some point, the growth rate of \( B \) declines, and \( C \) soon exceeds \( B \). [42]. However, there are several oversimplifications involved in this analysis, mainly due to problems with converting numbers of usability problems to dollars.

- The conversion rate \( V \) between problems and money can be viewed in two ways: from the client firm’s viewpoint, or from the software supplier’s viewpoint. If the goal is to maximize the client firm’s utility function, then the engineers must take care not to underestimate \( V \).
- The analysis neglects the cost \( r_c \) of repairing each problem found. Moreover, the most straightforward way to model this cost, by subtracting \( r_c \cdot N_{\text{total}} \) from \( V \) (or adding it to \( C \)), neglects the fact that fixing a problem costs more later in the development or maintenance cycle. Put another way, the benefit \( V \) of finding a usability problem is higher earlier on (since the fix cost is lower). In fact, by some estimates, the cost to fix a defect (not necessarily a usability defect) increases by more than two orders of magnitude over the course of development and maintenance [26].
- One practical problem with this whole \( B \) versus \( C \) approach is that it requires each software development organization to quantify a list of parameters: \( h_f, h_c, \) and \( V \). These vary greatly from organization to organization, and possibly from project to project [42]. However, a firm might be able to perform heuristic evaluation batteries on several projects and discern a rough range for these values.

\[9\] The careful reader will take some conclusions in [42] with a grain of salt. Specifically, the author aggregates a variety of usability test studies to estimate various parameters and costs. Unfortunately, these various studies had a wide variety of methodologies—indeed, some even relied on end users rather than usability experts—so it is anything but obvious that an “average” over them is applicable to a real software development organization’s situation. The author notes these limitations and properly caveats his conclusions, then suggests that each organization would do well to calibrate their own parameters on a per-project basis.
Finally, from the purely microeconomic perspective, V is not a fixed constant. This is because some usability problems are more serious than others, with a correspondingly higher V. Per the law of diminishing returns, as the quantity of a factor or factor attribute increases, its marginal contribution to production and utility functions decreases. So getting rid of those first few bugs is really valuable, but the marginal benefit goes down from there. Thus, the B estimate shown above is actually an overestimate, and the error of the overestimate increases as k rises.

In short, as with so many other quality attributes, there is no trivial quantitative way to convert between usability and price. But the overall forces at work should be clear: (1) Heuristic evaluation uncovers progressively fewer usability problems for every dollar spent. (2) The marginal benefit of finding a problem goes down as the big problems are discovered. These forces reinforce one another (both driving down B-C for higher k), and so software development organizations are faced with the important choice of when to call it quits.

Related techniques
Many organizations rely on usability testing by end users rather than heuristic evaluation experts. In cases like these, the end users are typically given a list of tasks to perform, and an observer records how long the tasks take. In addition, he observes how often the users struggle (and may videotape the session to facilitate future review), or he may require users to talk aloud while using the application, describing problems as they arise. Later, experts may examine the data and subjectively determine the problems’ seriousness based on the users’ comments and behavior.

A large family of usability testing methodologies has grown up in the past two decades, with each methodology blending these various techniques for eliciting and evaluating user feedback. Researchers have attempted to test these methodologies by pitting them against one another, but the research has struggled with validity to a certain degree [23]—mainly because of the difficulty in applying these varying methodologies in a consistent way, and also because researchers sometimes forget that there is no such thing as “one best” methodology.

In terms of modeling, one advantage of heuristic evaluation over end user testing is that heuristic evaluation better satisfies the independence assumptions of the Poisson distribution. In user testing, an observer watches a series of end users, and his interaction with later users can be influenced by their interactions with earlier users. Consequently, he must take great care not to introduce a bias, or else the probability of finding a certain problem in a later test will become dependent on the outcome of earlier user tests [42].

In terms of practical application, heuristic evaluation and end user testing costs vary a great deal by organization [42]. Moreover, neither methodology seems to be uniformly more effective than the other (in terms of having a consistently higher fitted \( \lambda \) parameter). It appears that researchers have only begun to understand when one methodology is preferable to the other.

One distinctive of end user testing is that it provides an empirical insight into how the client firm would value the software—assuming, of course, that users are wisely chosen to properly represent the target market. It may be a challenge to “invert” their perceptions into specific software changes to boost quality (whereas heuristic evaluators may try to provide specific instructions about how to improve the interface or implementation). However, there is much to be said for actually trying software out on the target population because this, in a sense, provides one direct measure of the usability term in the client’s utility and production functions.

Finally, with respect to both user testing and heuristic evaluation, the statistical fits discussed earlier constitute a patina covering up what remains an inherently subjective process: Using a human brain to decide what is important to test and to design a test, using a human brain to
process perceptions of the software while performing those tests, and using a human brain to assess the outcomes of the test.\textsuperscript{10}

3.3 Example: Quantitative Techniques Early in the Project – COCOMO II Early Design

Along the spectrum from qualitative to quantitative techniques, the COCOMO II models constitute more quantitative approaches [10]. The main goal of the Early Design variant is to predict man hours and schedule based on three sets of factors: (1) the size of the software as inferred at design time, (2) the other attributes of the software to build, and (3) the characteristics of the development organization and project. COCOMO II represents one of the best known, and most refined, attempts to provide an algorithm for relating quality attributes (including cost) to one another.

Basic functional form

Based on researchers’ analyses of dozens of projects, COCOMO II predicts the exponential equation...

\[ PM = M \times \text{Size}^5 \]

Here,

\[
\begin{align*}
PM &= \text{person-months to build} \\
\text{Size} &= \text{software size in thousands of logical lines of code (KSLOC)} \\
S &= 0.91 + 0.01 \times \sum s_i \\
s_i &= \text{scale factor } i \text{ discussed below} \\
M &= 2.94 \times \prod m_i \\
m_i &= \text{effort modifier } i \text{ discussed below}
\end{align*}
\]

In other words, the work to build an application rises somewhat exponentially with the software’s size. At design time, the engineers determine the projected size by computing the unnormalized FP metric, discussed earlier, and convert it to KSLOC by multiplying by a programming-language-driven conversion ratio. These conversions have been extensively studied [29]. Some samples follow.

<table>
<thead>
<tr>
<th>Language</th>
<th>Logical Lines of Source Code (LOC) per Function Point (FP)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Low</td>
</tr>
<tr>
<td>Basic Assembly</td>
<td>200</td>
</tr>
<tr>
<td>C</td>
<td>60</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>75</td>
</tr>
<tr>
<td>COBOL</td>
<td>65</td>
</tr>
<tr>
<td>C++</td>
<td>30</td>
</tr>
<tr>
<td>Visual Basic</td>
<td>20</td>
</tr>
</tbody>
</table>

\textsuperscript{10} In contrast, researchers have attempted for decades to apply formal methods to usability analysis, with mixed success, possibly due to adoption overhead. For an overview and analysis of commonly used techniques, refer to [28].
Each $s_i$ and $m_i$ parameter depends on the characteristics of the project, and the organization performing the work. They are as follow:

<table>
<thead>
<tr>
<th>Var.</th>
<th>Characteristic</th>
<th>Value to use when the characteristic is…</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>* = product quality attr.</td>
<td>Extra Low</td>
</tr>
<tr>
<td>$s_1$</td>
<td>Precedentedness</td>
<td>6.20</td>
</tr>
<tr>
<td>$s_2$</td>
<td>Development flexibility</td>
<td>5.07</td>
</tr>
<tr>
<td>$s_3$</td>
<td>Architecture resolution</td>
<td>7.07</td>
</tr>
<tr>
<td>$s_4$</td>
<td>Team cohesion</td>
<td>5.48</td>
</tr>
<tr>
<td>$s_5$</td>
<td>Process maturity</td>
<td>7.80</td>
</tr>
<tr>
<td>$m_1$</td>
<td>Personnel capability</td>
<td>2.12</td>
</tr>
<tr>
<td>$m_2$</td>
<td>Personnel experience</td>
<td>1.59</td>
</tr>
<tr>
<td>$m_3$</td>
<td>Facilities quality</td>
<td>1.43</td>
</tr>
<tr>
<td>$m_4$</td>
<td>Schedule stretch-out*</td>
<td>1.43</td>
</tr>
<tr>
<td>$m_5$</td>
<td>Reliability and complexity*</td>
<td>0.49</td>
</tr>
<tr>
<td>$m_6$</td>
<td>Reusability*</td>
<td>0.95</td>
</tr>
<tr>
<td>$m_7$</td>
<td>Platform difficulty*</td>
<td>0.87</td>
</tr>
</tbody>
</table>

For example, if the precedentedness of a product is very high, that means this team is familiar with this code (or code very like it), and consequently $s_1$ is fairly low: 1.24. Having a low $s_i$ helps keep the $S$ exponent small, thereby helping to keep the person-months PM low.

COCOMO's designers provide fairly precise qualitative explanations of what each characteristic refers to and how to evaluate it on the scale from “extra low” to “extra high”. (Refer to [10] for detailed instructions.) A quick summary of the criteria follow:

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Meaning and Evaluation Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>* = product quality attr.</td>
<td>Precedentedness: Similarity to previously developed products</td>
</tr>
<tr>
<td></td>
<td>Development flexibility: Freedom to not support legacy system requirements or interfaces</td>
</tr>
<tr>
<td></td>
<td>Architecture resolution: Amount of the architectural design that is now known to be risk-free (ie: “do-able”)</td>
</tr>
<tr>
<td></td>
<td>Team cohesion: Ability of the team to work together toward a common goal</td>
</tr>
<tr>
<td></td>
<td>Process maturity: How well the organization collects, analyzes, and optimizes process metrics</td>
</tr>
<tr>
<td></td>
<td>Personnel capability: Analyst and programmer ability, efficiency, thoroughness, and communication skills</td>
</tr>
<tr>
<td></td>
<td>Personnel experience: Months to years of experience using the development tools, platform, and language</td>
</tr>
<tr>
<td></td>
<td>Facilities quality: Supportiveness of the tools and the development site</td>
</tr>
<tr>
<td></td>
<td>Schedule stretch-out*: Amount that the project can be stretched out as needed</td>
</tr>
<tr>
<td></td>
<td>Reliability and complexity*: Extent it functions right; also control, computational, data, and user interface complexity</td>
</tr>
<tr>
<td></td>
<td>Reusability*: Additional effort required to carve out and properly package as reusable subcomponents</td>
</tr>
<tr>
<td></td>
<td>Platform difficulty*: How close the software comes to using all available CPU and memory on the platform; also the complexity of the platform used to support the application</td>
</tr>
</tbody>
</table>

Predicting person-months requires assessing the organization and product on each characteristic, determining the right $m_i$ or $s_i$ value, and then plugging into the formula.

(Many of these parameters capture product characteristics. However, in passing, it should be noted that from an economic perspective, other parameters capture production methodology issues. As discussed in this report's first section, all firms get to choose their production methodology. Here, software development firms have the option of investing in team cohesion, process maturity, personnel capability, personnel experience, and facilities quality. Doing so will reduce the person-months PM of a project, but will require significant up-front investment. For commentaries on the relevant management choices, consider [61] and [19].)
Validity
How valid is COCOMO II's estimate? According to the most frequently-cited work, the Early Design PM estimate is usually within 60% of the actual (empirically measured) person-month effort to design the system [10]. Of course, part of the inaccuracy comes from the fact that this estimate derives from the fact that it is made so early in the project, before most of the "gotchas" appear (more on that later). Moreover, any attempt to boil down a complex process like software development into a single simple formula like COCOMO's is sure to run into some accuracy trouble.

Finally, another important source of error is that the values shown in the LOC/FP conversion chart are based on IFPUG FP metrics [1], which differ somewhat from the unnormalized FP metric used by COCOMO II. Nonetheless, researchers use these conversion ratios because they are the best currently available. In general, the various types of FP metrics for a given project generally can differ by a few percent [32], though some writers note a somewhat wider variance between FP metrics, perhaps up to 50% [29]. This difference, of course, is amplified by the exponent S, potentially leading to significant inaccuracies.\footnote{On the other hand, it's not clear this accuracy is so bad, given that "expert judgments" can be off by quite a bit more, perhaps a factor of three.}

Nonetheless, despite these cautions regarding validity, COCOMO II has proven accurate enough that researchers have begun to use it as a framework for discussing the economic tradeoffs inherent in the software development process.

Tradeoffs
Most COCOMO II parameters, such as "personnel capability" have little to do with the product at hand. In contrast, the four asterisked \( m_i \) above may have some relationship to the resulting project's overall quality attributes—that is, they probably impact the utility function of client firms. In addition, of course, the customer will probably care about PM, which likely feeds directly into the product's price.

Specifically, suppose that the software development organization identifies a 500 FP design that would meet a certain market segment's needs. The organization has "nominal" values for all \( s_i \) and \( m_1 \) through \( m_3 \), assuming that the application could be implemented in their language of choice, C++. Using the conversion factor listed above, this results in \( \text{Size} = 26.5 \text{ KSLOC} \). This yields:

\[
S = 0.91 + 0.01 \sum s_i = 0.91 + 0.01 \cdot 18.97 = 1.0997 \\
M = 2.94 \cdot \Pi m_i = 2.94 \cdot m_4 \cdot m_5 \cdot m_6 \cdot m_7 \\
\text{PM} = M \cdot \text{Size}^S = 2.94 \cdot m_4 \cdot m_5 \cdot m_6 \cdot m_7 \cdot 26.5 \cdot 1.0997 = 36.74 \cdot m_4 \cdot m_5 \cdot m_6 \cdot m_7
\]

Now, if each person-year costs $125,000 (a reasonable number for many development organizations, including overhead costs),

\[
\text{Labor (\$)} = 382700 \cdot m_4 \cdot m_5 \cdot m_6 \cdot m_7
\]

For example, with a nominal schedule, reliability/complexity, reusability, and platform difficulty, the engineer would expect to incur development costs of $382700, which would be passed on to the client firm. Boosting the reusability to "very high" would raise the price to $440105. Alternatively, boosting the reusability to "very high" but cutting the reliability/complexity to "low" would leave the cost at $365287.15. The designer faces a five-dimensional tradeoff on labor ($), \( m_4, m_5, m_6, \) and \( m_7 \).
The software organization must now consider certain strategies.

<table>
<thead>
<tr>
<th>Option</th>
<th>Attribute Affected</th>
<th>Impact on Client Firm Utility Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimize $m_4$</td>
<td>Timeliness</td>
<td>Shortening the time to market allows the client firm to benefit from it sooner (and helps cement the supplier’s market share).</td>
</tr>
<tr>
<td>Minimize $m_5$</td>
<td>Reliability</td>
<td>Fewer problems using the software typically boosts the client firm’s production function, thus encouraging a rise in the utility function.</td>
</tr>
<tr>
<td>Minimize $m_6$</td>
<td>Reusability</td>
<td>Proper packaging may allow the client firm to perform some maintenance and extension without help from the software development organization. Alternatively, higher reusability might help the software developers build future products with a lower PM.</td>
</tr>
<tr>
<td>Minimize $m_7$</td>
<td>Compatibility</td>
<td>For a piece of software to work well on a wide variety of machines, it must make efficient use of CPU and memory (otherwise it will run slowly on older machines). This boosts the $m_7$ variable. Likewise, supporting a wider range of platform versions can introduce additional platform volatility, also contributing to $m_7$.</td>
</tr>
<tr>
<td>Minimize PM</td>
<td>Price</td>
<td>Reducing total person-hours generates cost savings, which the software development organization may pass on in the form of a lower price attribute.</td>
</tr>
</tbody>
</table>

Of course, in any real software development scenario, it is infeasible to achieve all five goals simultaneously. Consequently, the shrewd organization will consider what combination of attributes will maximize the utility function of the target market. Accomplishing this requires significant insight into the structure of that utility function; perhaps the best way to achieve this understanding is to utilize ATAM [31], Contextual Design [9], or a similar interview-driven process which puts the requirements engineers in direct contact with the potential client firms.

Related techniques
COCOMO II builds on the very similar COCOMO model, which became outdated as a result of modern tools and processes, such as rapid application development. Likewise, over time, COCOMO II faces the danger of becoming outdated. To keep pace, researchers have refined COCOMO II to address alternate development contexts.

- COCOMO II provides a “post-architecture” version of the formulas and attribute variables. Whereas the “early design” version discussed above has been calibrated for use after requirements gathering but before architecting, the post-architecture version has been calibrated for use after architecting—in other words, right before coding begins. The seven $m_i$ effort multipliers are broken out and specialized into seventeen specialized $m_i$ parameters in the post-architecture version. This additional specificity, as well as the better understanding of the “project gotchas” that arise during architecture design, results in a more precise estimate of PM—typically within 30% of the actual PM about 80% of the time [10]. This is impressive, considering the warning mentioned earlier that FP estimates can vary by up to 50%.
- COCOMO II provides a formula for converting person-months into months of calendar time. (This conversion applies to both the early design version and the post architecture version.) The formula ties into the “schedule stretch-out” parameter in a non-linear fashion.
- Researchers have performed early work on a COCOMO-like model called COQUALMO, which predicts the total number of software defects as a function of other post-
architecture quality attributes \[13\]. It models defects as being created by three processes—requirements-gathering, design, and coding.

\[ N = N_{req} + N_{design} + N_{coding} \text{ = total number of defects} \]
\[ N_x = M_x \times \text{Size} \text{ = # of defects created by } x \{ \text{requirements-gathering, design, coding} \} \]
\[ \text{Size} = \text{software size in thousands of logical lines of code (KSLOC)} \]
\[ S_x = 1.00 \]
\[ M_x = A_x \times \Pi m_{i,x} \]
\[ A_{req} = 1.8 \quad A_{design} = 0.77 \quad A_{coding} = 2.21 \]

As with COCOMO II, the \( m_i \) range from a little below 1.00 to a little above 1.00, depending on product, project, and organizational characteristics: precededness, development flexibility, architecture resolution, and so forth. In contrast to COCOMO II, however, evaluating each characteristic generates not just one but three \( m_i \)—one for the number of defects introduced during requirements-gathering (\( m_{i,req} \)), a second for the number introduced during design (\( m_{i,design} \)), and a third for the number introduced during coding (\( m_{i,coding} \)). This reflects the fact that some characteristics are more likely to create certain types of defects.

COQUALMO not only predicts the number of defects introduced, for each type of defect-introduction process, but it also predicts the number of defects detected and repaired via three testing methodologies (\( y \)): automated analysis, peer reviews, and execution testing.

\[ R = R_{req} + R_{design} + R_{coding} \text{ = total number of defects remaining} \]
\[ R_x = N_x \times P_{x,y} \text{ = # of defects of type } x \text{ remaining} \{ \text{requirements-gathering, design, coding} \} \]
\[ P_{x,y} = \Pi (1-p_{x,y}) \text{ = proportion of defects of type } x \text{ repaired by method } y \{ \text{automated analysis, peer reviews, execution testing} \} \]

\( p_{x,y} \) reflects how good an organization is at using method \( y \) to remove defects of type \( x \). Each parameter ranges from 0.00 (terrible) to nearly 1.00 (outstanding).

COQUALMO provides some baseline estimates for the \( m_{i,x} \) and \( p_{x,y} \) parameters, based on the opinions of experts in this field. However, unlike with COCOMO II, researchers have done very little calibration of COQUALMO against real project data. For this reason, software development organizations may be hesitant to use COQUALMO to assess tradeoffs among the different quality attributes \( i \) and the total number of residual defects.

Nonetheless, in theory, it may someday be possible to tie COQUALMO’s 21 product, project, and organization characteristics to a projected defect count. COQUALMO could then help engineers who have interviewed customers (and understood their utility function) consciously assess how much of each attribute must be sacrificed to boost reliability, or vice versa.

- Finally, with the advent of the web, many firms have moved toward extremely rapid-cycle development methodologies and heavy reuse of components [48]. These organizations often eschew the typical requirements-gathering phase of software development; instead, they immediately start building and evaluating prototypes by combining components with the electronic equivalent of duct tape.

This change of application domain, coupled with shifts in methodology, has inspired a web-targeted version of COCOMO II called WEBMO [48]. The first innovation is to replace the FP with a size metric more suitable for web applications. This new size metric resembles the traditional unnormalized FP function in that it is a linear combination weighting various functional forms; however, it counts web-like artifacts like multimedia
files, scripts, and graphics files instead of generic files and screens. As with FP, the weights depend on the complexity of the artifacts involved.

WEBMO predicts that the person-months PM grow exponentially with this new size metric. The exponent takes into account the type of product (such as e-commerce versus simple business-to-business application). As in COCOMO II, WEBMO’s PM will be multiplied by nine additional factors that take into various product, project, and organizational characteristics.

It is important to note that approaches like WEBMO remain in their infancy, having been applied by only a few researchers to only a few dozen projects with somewhat uninspiring association and predictive power. Within the next decade, workers may attempt to apply this model to other projects—where, of course, the model might not fit reality very well. In response, they may add additional parameters to the model or tweak existing parameter values.

To summarize, COCOMO II provides an early design and a post-architecture version. Each relates the total development effort to a variety of product and organizational characteristics. This facilitates having a discussion of how much boosting an attribute will hurt other attributes. Combining this with an understanding of the target market's production and utility functions has the potential to improve the product's overall value.

Other researchers have attempted to extend the use of these characteristics to predict additional phenomena, such the total number of residual defects, and as the development cost for a web application. For a fuller list of innovative approaches, consult [10]. In the future, researchers will doubtless extend the framework further, yielding additional tools to facilitate intelligently trading off quality attributes against one another.

3.4 Example: Quantitative Techniques Late in the Project – Code Complexity Metrics

Code complexity metrics exemplify the use of quantitative measures later in the project to control quality attributes. In software engineering, “complexity” usually refers to the “convolutedness” of source code (rather than to asymptotic performance characteristics, which is perhaps the more common sense of the word in computer science).

Since the late 1970’s, researchers have hoped that complexity metrics would serve well to predict some aspects of quality attributes such as cost, maintainability, flexibility, testability, understandability, and reliability, since it was believed that these attributes contributed very strongly to a wide variety of target markets' utility functions. To a large degree, this line of thought has borne fruit, leading to increased interest in the area of research and a proliferation of complexity metrics. These serve as a complement to the oldest complexity measure, lines of code, which had been used to characterize software size as far back as the 1960’s [4].

“Software Science”

One variety of complexity-related metrics appeared in the mid-1970’s under the banner of “Software Science” [24]. This work, and papers leading up to it, attempted to justify various metrics by considering psychological and information-theoretic arguments (whose specifics are irrelevant to the topic at hand). The resulting quantitative measures could be evaluated through an automated inspection of software code.
This approach defines the following:

\[ \eta_1 \quad \text{number of unique operators appearing in a module's implementation} \]

\[ \eta_2 \quad \text{number of unique operands appearing in a module's implementation} \]

\[ N_1 \quad \text{total usage count of all the operators appearing in a module's implementation} \]

\[ N_2 \quad \text{total usage count of all the operands appearing in a module's implementation} \]

\[ \dot{N} = \eta_1 \times \log_2 \eta_1 + \eta_2 \times \log_2 \eta_2 \] (called “calculated length”)

\[ V = \dot{N} \times \log_2 (\eta_1 + \eta_2) \] (called “program volume”)

Here, the program volume corresponds roughly with the minimum number of bits required to represent this implementation of the algorithm. However, this implementation is presumably not optimal in the sense of being minimally complex / convoluted. The least complex implementation would simply call a sub-function that directly implements the algorithm, resulting in a lower value \( V_{\text{opt}} \). The model’s creator argues that the abstraction level of a program can be expressed as the ratio between \( V \) and \( V_{\text{opt}} \). This “program level” can be estimated using

\[ L = \frac{2^*\eta_2}{(\eta_1 \times N_2)} \]

And

\[ D = \frac{1}{L} = \frac{\eta_1 \times N_2}{(2^*\eta_2)} \] (called “difficulty,” a complexity metric)

How valid is this? In the second half of the book, the author attempts to parameterize production costs and reliability in terms of \( D \). With respect to the validity characteristics discussed earlier, he has some success in demonstrating that \( D \) is associated with higher error rates (lower reliability) and higher production costs. This was a "double-whammy" of good news, since most software development firms would certainly love to produce reliable and cheap software (both of which obviously contribute to any rational client’s utility function). \( D \) and \( L \) seemed to be perfect metrics.

However, when later researchers attempted to test the validity of this metric on other sets of products, they found good association and predictability—but not necessarily any better than simply using physical or logical lines of source code (LOC).

For example, [37] attempted to correlate \( L \) with number of errors in NASA software modules; no strong correlation (association validity) existed. So the researchers then tried other analysis techniques (including an approach based on machine learning) which showed that almost all error free modules had \( L > \text{constant}_1 \) and \( V^D < \text{constant}_2 \). So it would appear that having a high \( L \) and a low \( V^D \) successfully predicts reliability. Unfortunately, and conversely, the authors also found that the modules containing errors generally had \( \text{LOC} > \text{constant}_3 \) and \( v(G) > \text{constant}_4 \) (discussed later). In fact, this paper ends up lauding the ancient LOC metric just as much as the \( D \) metric, meaning it cannot be concluded in an absolute sense that \( L \) and \( D \) are “superior” to simple LOC. On the other hand, other authors have reported greater satisfaction with the “Software Science” metrics [15], [47], [18].

In the end, [12], published in 1990, gives a well-balanced summary of this approach: it may be empirically useful under some circumstances, but the main contribution back in 1977 was the concomitant shift in how researchers viewed algorithms. That is, the “Software Science” helped develop the notion that algorithms possess attributes (such as \( V_{\text{opt}} \)) independent of any particular implementation, and the implementations’ attributes (such as complexity) can be profitably judged with respect to the characteristics of the underlying algorithm.
“Cyclomatic complexity”
Other researchers soon developed their own complexity metrics, including the “cyclomatic complexity.” Essentially, cyclomatic complexity measures the number of unique paths through a module’s implementation, as governed by the branching and looping constructs in the source code [15]. Chunks of code can be represented as nodes connected by edges representing control statements; the usual symbol for cyclomatic complexity on a code graph G is v(G). (This is the same v(G) mentioned above.)

In Java, one convenient way to calculate a method’s complexity is as follows [40]:

- Start with 1 for the straight-line path through the routine.
- Add 1 for each of the following keywords: if, while, and for or the conditional operator ?:
- Add 1 for each of the Boolean operators && and ||.
- Add 1 for each case in a switch statement, plus 1 more if the switch statement does not have a default case.

How valid is this? As with the D metric, researchers have hoped that v(G) would be a powerful predictor of reliability and other quality attributes. They have been somewhat pleased.

As noted above, NASA noted good predictivity of v(G), along with LOC, with respect to reliability [37]. Moreover, Hewlett-Packard found v(G) to be highly associated with the number of times that programmers had to modify a module during the course of development: So managers began tracking v(G) in order to predict when modules might need to be split up to reduce complexity [22]. In general, most authors seem satisfied that v(G) is generally useful for quantifying understandability, with attendant gains in reliability or maintainability [22], [47], [18] (though [12] expresses some reservations with respect to predicting error rates). Thus, if a development firm believes that reliability is a strong contributor to clients’ utility functions, then it may make sense to put effort into minimizing v(G), even if this implies occasionally costly code rewrites.

Some researchers, including [18], recommending combining the elements of D and v(G) to generate a unified “maintainability index”:

\[ MI = 171 - 5.2 \cdot \ln(V) - 0.23 \cdot v(G) - 16.2 \cdot \ln(\text{LOC}) + 50 \cdot \sin(\sqrt{2.4 \cdot \text{perCM}}) \]

Here, V, v(G), and LOC represent their respective metrics’ values averaged over all modules. The perCM variable indicate the percent of commented lines, averaged over all modules. Hewlett-Packard and the United States Air Force have informally validated this metric’s maintainability predictions against anecdotal evidence and the experiences of software developers [14]. More formal validation will hopefully be forthcoming from researchers.

Design complexity metrics
The complexity metrics discussed above have a significant weakness: they rely on code. While this makes it relatively easy to automate the metric calculation process, it means that engineers must expend significant time creating code before that code can be tested for complexity. In contrast, design-time metrics facilitate evaluation of multiple designs prior to large investments.

One way to attack the problem of quantifying the complexity of a design is to examine the proposed modules and their interconnections. Note, however, that the appropriate approach is not necessarily the most obvious. For example, as noted earlier, many studies found that modules with higher LOC contained more errors; so it might be inferred that complexity grows with average LOC per module. On the other hand, breaking long modules apart can result in more convoluted interconnections between modules. If complexity is supposed to include inter-module convolutedness, then simply minimizing LOC per module is not completely right, either.
In other words, successful architects seem to intuitively understand the need to pick modules of “just the right size” in order to maximize maintainability and other quality attributes, thereby boosting overall economic value. One metric suite that captures such tradeoffs was proposed over a decade ago [12].

\[
\begin{align*}
    f_k &= \text{fanout of module } k \\
    v_k &= \text{number of I/O variables for module } k \\
    s_k &= f_k \cdot v_k \\
    d_k &= \frac{v_k}{f_k + 1} \\
    c_k &= s_k + d_k \\
    C &= \text{average}[c_k]
\end{align*}
\]

(“structural complexity” of module k), (“data complexity” of module k), (“complexity” of module k), ("design complexity")

How valid is this? The original researcher claimed that these metrics adequately predicted error rate, man-hours to implement, and subjective code quality (as rated by programmers) [12].

Since then, other researchers attempted to validate the metrics by applying them to legacy systems. They did this by diagramming the system’s design, calculating the \(c_k\) of each module, counting the number of defects \(e_k\) in each module, and then attempting to correlate \(e_k\) against \(c_k\) [59]. To their surprise, the resulting coefficient was small, even when the number of defects was expressed as a rate per lines of code \((R^2 < 0.1)\).

The authors discussed this with developers who had worked on the project; these people reported that after a module became complex, programmers avoided adding to it, or behaved very carefully when modifying the module. In other words, somewhere between the design of the modules and the completion of the modules, project dynamics affected the defect rate.

This study highlights the fact that the project’s management and the organization’s process affect software quality—and, ultimately, the economic value of the product. Had the reliability of the legacy system been purely governed by the design metrics, then its high complexity modules would have contained more errors. Instead, at least in the case of this particular project, the developers made compensations in their practices to facilitate higher scores on this key attribute.

This, in closing, brings to mind the tradeoffs inherent in quality attributes. Anecdotally, the developers claimed that “they become more careful when modifying” complex modules. Such behavior affects the project’s schedule, which might have been acceptable and even laudable in the eyes of this client firm. On the other hand, other client firms value different quality attributes and might balk at increased development time, even if reliability suffered as a result. Economically savvy development firms will not only choose metrics appropriate to the situation at hand, but will also respond to those metrics according to the utility function of the customer at hand.

---

12 The metric shown above constitutes one of the best studied and most debated approaches to characterizing complexity at design-time. Others have been proposed, ranging from a design-time cyclomatic complexity metric (where each node in the graph corresponds to a module, and the directed edges represent dependencies) [35], and other, fairly recent attempts [62]. It is not yet clear how generalizable these will turn out to be, in part because of project-dependent variations like those mentioned above. Another hurdle to validation had been a lack of a widely-adopted, consistent architectural representation scheme; the rise of CASE tools, UML, and architectural description languages like Wright and UniCon might ameliorate this problem [62].
**Related techniques**

Code complexity metrics serve as a powerful tool for controlling quality late in the development cycle because they (1) are automatable and (2) can highlight high-risk areas of code. [7] recommends that metrics like the maintainability index “be integrated into a development effort to screen code quality as it is being built and modified; this could yield potentially significant life cycle cost savings.” This exemplifies using a metric as a “project measure.” [47] explains,

> Software project measures are tactical. That is, project metrics and the indicators derived from them are used by a project manager and a software team to adapt project work flow and technical activities.... The intent of project metrics is twofold. First, these metrics are used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks. Second, project metrics are used to assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality.

In other words, product attribute measures can be used dynamically to guide the project that produces the product.

But note that code complexity metrics are not the only product quality tool that can be used to guide project managers. As the product nears completion, almost any technique will do, as long as applying it makes sense late in the project. For instance, as discussed earlier, a manager can observe how many usability errors are uncovered by heuristic evaluators, and he can use this information to decide whether to keep testing or to release the software.

In addition to tracking product metrics, managers can measure project metrics. These include the project's person hours consumed, the current position relative to schedule, the number of modules successfully coded and tested, and so forth. Some measures, such as tracking the code coverage of testing, blend product and project aspects. For a fuller list, consult [47].

Organizations seem to pick project metrics in an arbitrary fashion and “try them out.” This helps them gain confidence in metrics’ ability to track the economic attribute of interest. These organizations usually dispense with a full-blown validity analysis. Of course, this may impact the generalizability and trustworthiness of the resulting measures. However, in many cases, the organizations are simply looking for a way to give their managers a handle on their projects.

Tracking a product’s current condition, whether qualitatively or quantitatively, is a major key to managing a successful project. One writer has advised, “Think of developing a new release as just a continuous process of adding many single features to a known product and shipping it out the door.... The key is always to keep the product in a known, shippable state” [36]. The lesson is simple: If the manager uses code complexity metrics or other tools to track the product’s status as it nears completion, then he knows when it is ready to ship, and he can do everything in his power to keep it there as last-minute features are implemented.

When measured throughout implementation, metrics provide a dynamic picture of what’s currently right or wrong about the product’s economic value. Used this way, they are a measure not only of “How good is it?” but also “Is it getting better?”

What project manager would ever want to be without tools like these?
4.0 Beyond Today’s Product

4.1 Optimizing the Producer

From product to producer

This report has introduced an economic framework, and then used it to frame the discussion of four families of quality attribute evaluation techniques. In other words, this report has summarized (1) how to think about software’s value in economic terms and (2) how to measure software attributes which contribute to a software product’s value.

Suppose that an engineer learns those principles, applies them to one particular project, and produces one product with attributes highly-tuned to one market. Suppose that one product is released, the customers love it, and the producer sells a million copies.

What comes next?

Flush with success, the organization may want to repeat that one success on another product, employing all the evaluation techniques, which proved so valuable in the first project, to the second. And they may succeed again. And again. And, hopefully, again. By then, the evaluation techniques, now highly valued, may become a mandatory part of the organization’s development process on every project, to be applied in a standardized way over and over.

But not all projects will be equally successful. Some will stand out. Others will be marginal.

At some point, the organization may begin to ask, “How can we ensure that we always succeed? How can we continually do better? What do we have to do so our successes grow in frequency and magnitude?” In response to this strategic thinking, the organization may tweak their development process slightly, perhaps calling for more requirements analysis here and a bit more testing there. They can track the project’s metrics against historical projects (with a similar product developed using a slightly different process) and see whether or not the tweaks increased the margin of success.

The organization has now moved beyond optimizing products and begun to optimize its process. In a sense, it is optimizing itself.  

Raising the level of planning, introspection, and optimization from the product to the process has immense power, since it helps the organization to generate products with high economic value every single time. The increased scope provides both a better understanding of the organization’s relationship to the market (since it looks at a wide range of data, spanning multiple projects, rather than just one), as well as a wider range of products that it can affect. That is, it has increased effectiveness and wider applicability than the product-oriented evaluation techniques covered in this report.

Or at least that is the promise of process optimization. The reality is that tweaking a process incorrectly can quickly result in significant monetary loss [33] [19]. Moreover, getting an organization to think introspectively about its process, let alone actually beginning to optimize it, can take years of effort. It is not something that a new software engineering graduate can realistically expect to accomplish quickly except in the smallest firms.

---

13 It should be noted that many authors, such as [33] and [36], spend a good deal of time arguing that the main key to software development success is to hire the right people. In microeconomic terms, this means inputting high-quality factors into the software development production function. In this view, metrics and other optimization tools are secondary. This is just another way of saying that metrics are not a cure-all.
For these reasons, this report has focused on product value optimization, rather than process optimization. Nonetheless, because process optimization is so valuable, and because product value optimization can naturally lead over time to process optimization, this report concludes by very briefly skimming over a bit of research on the subject. That way, the foresighted engineer will be aware that the tool exists, should a propitious situation for applying it arise.

**Capability Maturity Model**

The Capability Maturity Model (CMM) refers to the progression from product measurement to process discipline as "becoming a mature organization" [44].

In a mature organization, managers monitor the quality of the software products and customer satisfaction. There is an objective, quantitative basis for judging product quality and analyzing problems with the product and process. Schedules and budgets are based on historical performance and are realistic; the expected results for cost, schedule, functionality, and quality of the product are usually achieved. In general, a disciplined process is consistently followed because all of the participants understand the value of doing so, and the necessary infrastructure exists to support the process.

The various CMM levels have been defined as follow:

- **Level 1: Initial**
  The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort.

- **Level 2: Repeatable**
  Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

- **Level 3: Defined**
  The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software.

- **Level 4: Managed**
  Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.

- **Level 5: Optimizing**
  Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.

(This model corresponds roughly to the $s_5$ variable of COCOMO II, “Process Maturity.” An organization at CMM level 1 is at “Very Low” or “Low” process maturity. CMM level 2 is “Nominal,” level 3 is “High”, level 4 is “Very High”, and level 5 is “Extra High.”)

Organizations start out at level 1 and ideally move toward level 5—assuming adequate availability of funds, time, and management commitment to increasing maturity. The first few steps involve taking hold of the development processes and gradually imposing standardization on how the organization produces software. Once the organization has settled on, and is actually following, some semblance of a disciplined, repeatable software development process, then it makes sense to begin collecting detailed metrics. (Putting a defined process in place helps ensure that metrics measure the same thing from project to project; that way, the metrics have a consistent meaning and level of validity, which facilitates inter-project comparisons.) Finally, the organization can tweak the process and see how it affects these metrics in a never-ending quest to improve the process further.
Closing thoughts
Although moving a large organization from one CMM level to the next can save an organization millions of dollars [19], achieving this rise in process maturity requires an upfront investment of millions of dollars and months of hard work. For this reason, as well as institutional inertia, virtually all organizations remain at level 1, but this may be changing as vendors develop tools which facilitate the standardization of process and collection of metrics [33].

The bottom line is that each organization must decide how important it is to deliver high quality software. Various organizations will answer this question differently, of course, and will take varying approaches to improving their development process. Some will focus on attempting to standardize the software development process, and leave it at that. Others will integrate early-cycle qualitative evaluation techniques (like ATAM) and benefit from their ability to deliver requirements tuned to the client firm’s needs. Others will rely on full-featured cost estimation models like COCOMO II or WEBMO.\(^{14}\)

Yet one lesson remains: in today’s competitive software environment, any organization that neglects quality does so it at its own peril.

---

\(^{14}\) A wise organization should approach any process enhancement (including metrics tracking) with tentative steps. Not all enhancements will work for all organizations, and many organizations are simply not yet ready to move up the CMM scale. Above all else, take care not to introduce fancy new process enhancements into an already-struggling project, or at least not without a good deal of thought [61]. Process enhancements add overhead, and therefore represent an investment, so they demand consideration concerning how to best direct the investment.
4.2 Areas of Active Research

Quality attribute research still remains somewhere between infancy and adolescence. Current areas of investigation include...

- Run-time optimization of quality attributes. One application is handheld computing, where competing attributes such as battery life and application responsiveness (that is, CPU usage) both contribute to the user’s utility function yet cannot be independently optimized due to technological constraints. Moreover, the user’s utility function might vary over the course of an hour. Consequently, researchers are looking into ways of dynamically reconfiguring the behavior of the hardware-software system in order to re-optimize the total utility on a real-time basis [45].

- Optimization of potentially emotion-laden attributes. These include notions such as the risk that a system security breach will do a great deal of property damage or kill somebody. Security sub-attributes like these can be hard to weigh in a rational manner against one another (including, as they do, the possibility of human deaths). Yet, as with other quality attributes and sub-attributes, these risks must be traded off against one another. Consequently, the challenge is to find robust ways of characterizing the additive value of these sub-attributes (while avoiding touchy conversions to a base concept like dollars) and utilizing a decision process for weighing sub-attributes against one another [11].

- Design-time optimization of usability. It had been realized several years ago that proper optimization of usability begins well before the final stretches of a software development process (though some organizations persist in bringing usability experts into the project only near the end). Researchers continue to push usability concerns even earlier in the process and have identified over two dozen scenarios in which early architectural commitments are necessary (but not necessarily sufficient) to ensure the usability of the resulting system [8]. Emerging research, including case studies and quantitative experiments, strongly suggest that designers are far more likely to address these usability issues early if they are provided with examples of how to do so. (The pattern community will surely laud this result when the work reaches fruition!)

- Optimization of software construction as a real option. In economics, an “option” is an opportunity to invest; the holder of this opportunity has the right, but not the requirement, to exercise this option. A “real option” is an option to invest in a “real asset” such as real estate, buildings, and intellectual property. Once the purchase is made, and the building or software is built, the decision to exercise the option cannot be reversed. Although economists have already developed a well-rounded theory of real option issues, software engineering researchers have only begun to explore how this economic theory applies to software-build decisions. One early outcome of this research was to identify certain limitations to the time-worn methodology of evaluating software developments (the “net present value” calculation) [57].

This obviously does not constitute an exhaustive list. Indeed, researchers are constantly inventing new ways of examining the interplay among quality attributes and evaluating the overall value of a software system.
4.3 **Annotated Bibliography**

The first section of this bibliography groups references according to their topic's relationship to software quality. The second section lists the references alphabetically.

**Grouped by Topic**

*ATAM and Other Semi-Qualitative Design-Time Evaluation Techniques*

{31 43 5 9 27 7 6 30 16}

*COCOMO II and Cousins*

{13 10 48}

*Complexity Metrics and Related*

{22 21 37 59 24 14 18 62 12 35}

*Economics and Business*

{34 52 20 49 51 56}

*FP Metrics*

{1 29 32}

*Introductions, Overviews, and General Reference*

{47 15 3 4 40}

*Measurement Theory and Metric Validity*

{58 17 50 55 23}

*Nuts and Bolts of Project Management*

{61 36}

*Process Improvement and Related Return on Investment*

{26 44 19}

*Software Valuation*

{54 53 25 46 57 33 11 45}

*Usability*

{41 60 39 8 42 28}
Listed Alphabetically

   As FP metrics continue to evolve, this site provides pointers to the latest approaches.

   According to widely-accepted accounting practices, and the federal government, capital investments lose value over time. As a capital factor of production, software depreciates too. It turns out that the rules for depreciating software have some odd wrinkles (essentially depending how custom it is). Hire an accountant.

   This handy reference contains definitions of commonly used software engineering terms, as well as (through the organization of this document) a structure showing how the corresponding engineering concepts relate to one another. Because hundreds of engineers contributed to this document, and because it has received the imprimatur of IEEE, it can be useful for arbitrating among competing term definitions in the literature.

   In the current context, this conference, which essentially invented the term “software engineering,” is perhaps interesting for what it does not cover: Specifically, the participants’ conception of “quality” seems to be somewhat embryonic with respect to certain attributes such as usability and portability. It provides an interesting historical balance to realize that some attribute conceptions remained relatively unformalized until well into the 1970’s.

   Software designs are intended to solve problems. The authors argue that the problem space can be dimensioned according to requirements, and the solution space can be dimensioned according to architectural features. The two spaces are coupled through constraints and utility. This paper represents another way of looking at the problem of choosing a “high quality” software design.

   This report introduces Attribute-Driven Design (known by its previous name, "Architecture Based Design"), a methodology for using quality attribute requirements to drive architectural decisions.

   This report introduces Quality Attribute Workshops and gives specific hints for how to run them, with the goal of acquiring and precisely documenting customer expectations for software.

   This paper provides an insight into ongoing research into specific quality attribute issues—specifically, usability. It inventories over two dozen situations where usability directly impacts architectural decisions, thus tying into early quality attribute tradeoffs.

   The authors argue for the collection of customer data in the context of real work, and the integration of this data into a work and software design process. Viewed through the lens of software quality, this book’s main contribution is to outline a methodology by which a software development organization can understand what to build in order to maximize the client firm’s utility and production functions.

    The early portions of this book introduce COCOMO II, including the Early Design and Post-Architectural flavors. It contains some discussion of validation attempts.

This thesis brings the tools of decision science to bear on the problem of weighing security sub-attributes against one another. Perhaps its most interesting contribution to quality attribute control is a framework for balancing emotionally touchy sub-attributes without converting to a base unit like dollars.


The authors review several complexity metrics as well as various attempts to demonstrate that these metrics correlate with interesting quality attributes (such as modifiability and reliability). They conclude that although the metrics hold some promise in certain situations, they are not a panacea.


In the final sections of this thesis, the author explains the inner workings of COQUALMO and explains how to use it to predict defect introduction and removal rates. As of the publication date, COQUALMO was not fully calibrated, and it appears that only limited recalibration has occurred since then.


The authors report on various experiences applying project metrics to Hewlett-Packard and US Air Force projects. The main contribution is to add weight to the claim that complexity metrics constitute a valuable tool for identifying modules that might be somewhat unmaintainable.


The author reviews quality and complexity metric research through 1981. It contains a particularly readable explanation of the cyclomatic complexity, at a conceptual level.


This technical report discusses several dozen techniques for enhancing security, performance, dependability, and modifiability quality attributes. Some techniques focus on directly promoting these attributes through architecture, whereas others focus on detecting when the attribute is flagging at run-time and/or dynamically reconfiguring the system to enhance the attribute. The report’s main contribution is to provide numerous examples of how optimizing one attribute can have negative effects on another.


This mathematically-oriented paper considers what metric validity means in the context of the representation theory of measurement. It points out a number of thought-provoking conclusions concerning the limitations of metrics due to the fact that each is confined to a scale with limited powers.


This wide-ranging review of software technologies is interesting because it covers the Maintainability Index on pp 231 ff, including a discussion of how organizations might use this metric in practice. Moreover, it contains a list of citations useful for future follow-up.


The authors note that introducing process enhancements costs money and ask whether the investment is paying off. They note possible selection bias in their anecdotal data (meaning that firms whose enhancement plans failed were probably less likely to trumpet their results than successful firms). They then recount several instances where process improvement saved millions of dollars to various organizations.

This paper is only tangentially related to software quality. It discusses the differences among a business idea, a business plan, and a business model. In doing so, it outlines several of the key issues that a software engineer should consider when proposing a new business or new product line.


This paper summarizes experiences applying project metrics to small US Army projects. The main contribution is to add weight to the claim that complexity metrics constitute a valuable tool for identifying error-prone modules.


The author reports on his experiences applying project metrics to Hewlett-Packard projects. The main contribution is to add weight to the claim that complexity metrics constitute a valuable tool for identifying error-prone modules.


The authors give an overview of what constitutes validity in experiments (as well as potential threats to validity), and then apply these principles to several studies which had attempted to compare usability evaluation methods. This paper is valuable mainly because it covers the essence of comparison studies in an extremely readable, practical manner.


This book begins by arguing for a number of complexity metrics; the second half focuses on trying to correlate these metrics with maintainability, reliability, cost, and other software attributes. The result is only half convincing: Although some of the claims of correlation seem reasonable, later researchers have generally concluded that the theoretical underpinnings of the “Software Science” are questionable.


For years, writers have claimed that information technology has not contributed, on net, to the economy. This paper considers this claim in the light of nearly a decade’s corporate spending data. It shows that although productivity has benefited, businesses have been unable to convert these benefits into profits, meaning that much of the information technology’s value has trickled through to the final consumers.


The author spends a great deal of time arguing that each programmer should develop a personal software development process. While some of this is generalizable to organizational development processes, the gem in this book lies within chapter 9, where the author briefly reviews quantitative estimates of defect repair. The punch-line is that quality is not something that can be cost-effectively added at the end of a project.


The author argues that certain problems appear over and over and that by categorizing the types of problems in the world, software engineers can take a great leap toward improved quality. Perhaps the best insight in this book is the argument (beginning in chapter 1, but recurring throughout the book) that the real problem to be solved exists in the world, not at the interface between the world and the system. That is, clients are looking for real solutions to real problems. This realization is the starting point for understanding that different customers value different quality attributes.

GOMS represented an early approach to formalizing usability testing by attempting to quantify the difficulty of using a given interface’s elements to achieve a certain set of tasks. Since its introduction over ten years ago, researchers have created variants. This paper compares these GOMS variants on the basis of the assumptions they make about the architecture and the outputs they are capable of generating. In the context of quality attributes, this paper provides a tidy overview of four related evaluation techniques dealing with a particular attribute, usability.


The author draws on extensive experience as a software development consultant to characterize the many types of software development projects and draw key insights about each. The main value of this book, in the context of software metrics, is the discussion of commercial size metrics (such as FP) in chapters 3 and 10.


This report introduces the Cost Benefit Analysis Method, which adds a semi-quantitative finish to ATAM. In particular, it connects risks and rewards with the potential return on investment.


This paper introduces ATAM, a qualitative methodology that helps software engineers tailor their architectural designs to the particular quality attribute needs of the current project.


This paper, more than three years in the making, assigned over two dozen real system requirement documents to FP evaluation experts. These experts then read through each assigned documents and estimated the FP metric for the corresponding software project. Multiple reviewers were assigned to each project, with varying FP metric methodologies, which allowed the paper author to assess the inter-reviewer FP reliability and intra-FP reliability. The results were encouraging: by most criteria, the correlations between reviewers, and the correlations between distinct FP metrics, were fairly high.


The author reviews recent developments in CASE tools and process methodology. The discussion centers on two key issues: the promise of software metrics, and the importance of hiring good people.


This relatively mathematical text can be referenced after the other economics texts listed here, if it becomes desirable to treat an introductory economic principle at a higher level of rigor and formalism.


The authors attempt to generalize their code-centric cyclomatic complexity metric to the design-time context. This involves introducing a mildly complicated set of rules for calculating the design complexity. Although this application appears to have been ground-breaking, this paper does not seem to have provoked significant validation studies by other authors, perhaps because the approach felt unnecessarily complex.


The author offers 54 pithy rules for how to successfully manage the software development process. Most of these insights tie into quality attribute control in some fashion or another. Many deal with the challenges of tracking a project’s completion through the various tradeoffs and challenges a real project encounters.

The authors report on their experiences applying project metrics to NASA projects. The main contribution is to guardedly add some weight to the claim that complexity metrics constitute a valuable tool for identifying error-prone modules.


It can be shown rigorously that any general machine learning system (general in the sense of incorporating no bias toward any particular distribution of problem instances) would be completely incapable of solving any problems. Consequently, machine learning systems must be biased toward solving particular problems; they cannot successfully solve all possible problems. This truth and its corollaries are a recurring theme throughout the book. It is an excellent demonstration, in a particular solution domain, of how no computer system can optimize all possible quality attributes (such as precision and recall).


This paper combines six usability measures into one composite metric and then applies it to a single diagnosis expert system. In the process, it highlights the challenges of creating a valid quality metric.


Although most of this book focuses on Java-specific issues, the last chapter provides an interesting and useful overview of complexity metrics, CMM, and other software engineering innovations. The author leverages the reader’s familiarity with Java to help make these innovations more understandable.


The authors give an overview of heuristic evaluation and then apply it to four projects. Their main accomplishment is to document the fact that each usability expert typically identifies only 20-50% of the total number of defects but that each expert identifies different problems, so that their independent efforts could be combined to generate much better defect discovery.


The authors consider eleven papers that applied end-user usability testing or heuristic evaluation; they draw out some general lessons (such as the applicability of the Poisson distribution for defect detection). Then they argue, somewhat convincingly, that because of the declining returns of usability testing, organizations should attempt to assess when testing costs threaten to exceed the benefits of additional testing.


This technical report examines how a software development organization can combine QAW and ADD in the context of ATAM. The resulting methodology exhibits QAW’s effectiveness at eliciting and documenting the customer’s quality requirements, yet also exhibits ADD’s effectiveness at incorporating these requirements into architectural decisions in a semi-quantitative fashion.


This report introduces the Capability Maturity Model, which is useful for rating an organization’s progression toward a metrics-collecting and optimizing state. Since its introduction over ten years ago, this scale has been widely adopted for characterizing process maturity.


The authors outline the problem of dynamically optimizing mobile hardware-software systems as the users’ utility function rapidly changes with time. In a sense, this represents an attempt to optimize software quality in real-time and promises to be an area of active research in the future.

This paper’s main contribution is to point out that it is dangerous to automatically valuate all quality attributes in a single scalar unit, such as dollars. The reason is that these attributes (and, more generally, resources and costs) vary in terms of divisibility, fungibility, measurement scale, and other important properties that make straightforward conversions impossible.


This book provides an excellent overview of what software is, from a practical viewpoint, though the material sometimes seems somewhat out of order. Chapters 4, 19, and 24 deal with project, product, and object-oriented metrics (respectively).


The author contends that existing cost estimation approaches are inadequate for the rapid-fire environment of web application development. Consequently, he introduces a new approach tailored to estimating project size and person-months in the web context. Further validation studies are required.


This economics text introduces basic principles with only a little formalism and mathematics.


The author outlines six metric validity criteria and then demonstrates them on several simple metrics.


This economics text covers basic microeconomics with enough formalism and mathematics to meet the needs of most software engineers. [49] and [56] are listed here in case a more qualitative “ramp-up” is needed.


The authors pick out a number of time-worn economic principles and show how they apply to software engineering problems. In the context of software quality, chapters 3 and 7 deliver the most value, as they focus on market segmentation (according to what qualities each segment values) and network effects (essentially non-monetary back-flow), respectively.


This paper aims to translate the formalism of economic utility optimization into a form usable for supporting software engineering decisions. It starts by introducing the necessary formalism, then uses it to discuss ATAM and COCOMO II. This paper essentially served as intellectual yeast for the current report.


The main gist of this article is that software does not need to be optimized on all its attributes in order to be successful. The paper then reviews current Carnegie Mellon University research concerning how to sensibly manage the inter-attribute tradeoffs, and how to compensate users when software quality flags.


Chapters 1 and 3 provide an excellent overview of what measurements are and how they relate to software quality.


This economics text provides the “entry level” qualitative introduction to basic economic principles.

The author argues that software, like buildings, represent real options. He proceeds to examine how this important fact impacts software valuation. The most important outgrowth is that traditional “net present value” calculations are inadequate when exercising options is irreversible and future conditions are uncertain.


These authors propose half a dozen evaluation criteria for metric validity. Interestingly, many of these criteria appeal as much as possible to the internal mathematical structure of the metric at hand (versus direct comparison to the underlying, measured attribute), which may be valuable when proxies for the underlying attribute are in short supply. The authors mainly focus on complexity metrics, but the methodology appears to be somewhat more general.


The authors report on their experiences applying design complexity metrics to modules of a legacy IBM project. The main contribution is to highlight the fact that these metrics only capture half the defect-prediction story: Project dynamics can have a significant effect on where defects appear.


The author applies three somewhat varied usability tests to three different systems in order to support the argument that, empirically speaking, a Poisson distribution properly models the defect discovery process. The paper is a fairly convincing example of how to demonstrate the generalizability of a result.


No discussion of the software development process would be complete without incorporating the relevant points from this iconoclastic yet realistic book. In the case of software process metrics, the relevant chapter is number 5, which contains numerous warnings and hints about enhancing an organization’s development process.


This paper pauses to speculate that perhaps older design-time complexity metrics did not take off because there had been no unified languages for expressing architectural design. The authors then discuss recent research in the area, taking advantage of the architectural description languages which have arisen in the past ten years.