Software Organization Productivity

By Dick B. Simmons
Software Productivity Improvement Laboratory
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112

Email: simmons@cs.tamu.edu
Telephone: (409) 845-2015

Abstract

Software projects often do not turn out as planned. When milestones are missed, frequently the knee jerk reaction of management is to increase effort to assure that a software product is delivered sooner rather than later. One school of thought says that increasing effort helps, another says it does not. This article links compressing delivery schedules to increasing productivity of an organization. We show that if the problem that causes a missed milestone in not related to lack of effort, then increasing effort alone probably will not improve deliver schedule. In fact in some cases, we show that increasing effort actually decreases the overall organization productivity.

Introduction

In the world of software development, the customer would like to pay as little as possible. He would like the highest productivity as possible out of the software contractor. The software contractor would like to make as much money as possible. The software contractor is usually paid either directly or indirectly by how many people are working on a project. The greater the effort (and the lower the productivity) the higher is the profit for the contractor. Whenever a problem arises, the contractor would like to solve the problem by adding people.

For example, most projects do not allow enough time to complete a project. Brooks says that more projects have gone awry for lack of calendar time than for all other causes combined.[1] Plans often do not allow for a realistic amount of time to complete a project. When a project falls behind, contractors often claim that adding more people to a project will speed up project completion. Adding people to a project can actually cause it to take longer. Brooks proposed the following Law:

Adding manpower to a late software project makes it later.

When he first proposed his Law, Brooks admitted that it was oversimplifying outrageously what happens. There have been many careful studies evaluating the truth of Brooks's Law. After 20 years of evaluating the many studies of his Law, Brooks feels that it is the best zeroth-order approximation to the truth, a rule of thumb to warn managers against blindly making the instinctive fix to a late project. If adding manpower to a late project makes it later, the productivity of an organization is actually lowered. Can adding people to a project lower software organization productivity?

We will investigate the effect of adding people to a project to demonstrate when it helps improve delivery schedule and when it does not. If adding people to a late project does not make it later, then adding people must make it finish sooner. If you add people to a project or increase project effort, then the productivity of the project organization must improve for sooner completion. In other words, if you have a project with eight people on a team and you add a ninth person, then the team productivity (or organization productivity) must increase for the project to finish sooner. If it finishes later when effort is increased, then any claim that adding people always helps improve delivery schedule would be false.

We will examine software organization productivity by trying to answer the following questions: What makes up a software project? How do we determine when a project is successful? How do we control a project? What is the relationship between project completion time and sequential factor, communication factor, and organization productivity? Which cost drivers dominate organization productivity causing project failure?

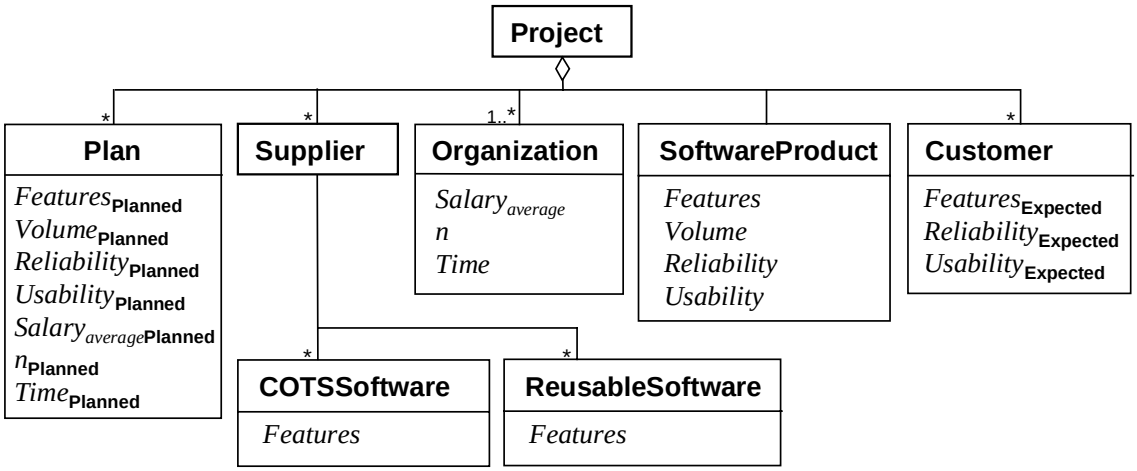
Background

Everyone likes successful software projects. A simple criterion to measure the success of an organization is to determine if it delivers a software product on time, within budget, and to a satisfied customer. In 1995, Capers Jones stated that the failure or cancellation rate of large software products is over 20%.[2] Of the 80% that are completed, approximately two thirds are late and experience cost overruns as much as 100%. Roughly two thirds are also plagued by low reliability and quality problems in the first year of operation.

To create successful large software products, we must have a plan against which to measure success. We will now introduce terminology for discussion. The aggregate parts of a **Project** are represented in Figure 1. A development **Project** starts with a **Plan** that predicts **SoftwareProduct** features ($Features_{Planned}$), size ($Volume_{Planned}$), reliability ($Reliability_{Planned}$), and usability ($Usability_{Planned}$). The **Plan** will also predict the

average salary ($Salary_{average\,\mathbf{Planned}}$), size of the **Organization** ($n_{\mathbf{Planned}}$), and time in it will take to complete development ($Time_{\mathbf{Planned}}$).

The fastest way to developed software is to not develop it, but to reuse software that already works. We can acquire reused software from a **Supplier**. The easiest reusable software to acquire is commercial off the shelf software (**COTSSoftware**). Reusable chunks of software that we obtain from a colleague or from a reuse library we will call reusable software (**ReusableSoftware**). **COTSSoftware** takes no development time, but effort and time may be required to adapt it to the planned **SoftwareProduct**. **ReusableSoftware** obtained from a reuse library usually requires additional time and effort to adapt it for use by the planned **SoftwareProduct**.



Firuge 1. Software **Project** aggregate parts.

Software developers make up the **Project Organization**. Anyone whose salary is charged to the **Project** we define to be part of the **Organization**. **Project** *Cost* can be computed from average monthly salary ($Salary_{average}$), nominal number of persons working on the **Project** (n), and development time in months (*Time*). The status of the **SoftwareProduct** can be monitored by tracking its features (*Features*), reliability (*Reliability*), and usability (*Usability*).

Every effort should be made to satisfy the **Customer**. Experience has shown that **Customer**s are happier if they receive a quality **SoftwareProduct**. Simmons et al.[3] say that **Customer** satisfaction can be predicted if a **SoftwareProduct** has expected features ($Features_{\mathbf{Expected}}$), reliability ($Reliability_{\mathbf{Expected}}$), and usability ($Usability_{\mathbf{Expected}}$). A successful **Project** should plan for

$$Features_{\mathbf{Planned}} \subseteq Features_{\mathbf{Expected}}$$

$$Reliability_{\textbf{Planned}} \geq Reliability_{\textbf{Expected}}$$

$$Usability_{\textbf{Planned}} \geq Usability_{\textbf{Expected}}$$

$$n_{\textbf{Planned}} \leq n_{\textbf{Completed Project}}$$

$$Time_{\textbf{Planned}} \leq Time_{\textbf{Completed Project}}$$

Seldom does a **Project** proceed exactly as planned. When problems emerge they often show up when milestones are not completed on time. **Project** control can be asserted by applying the control triangle shown in Figure 2. Depending on the problem that causes a milestone to be missed, you may be able to decrease *Time* by deleting *Features* or increasing *Effort* or both. When you decrease *Features*, the *Volume* of a **SoftwareProduct** decreases. Thus when you reduce the *Volume* of a **SoftwareProduct**, the *Time* and *Effort* required to develop it should decrease.

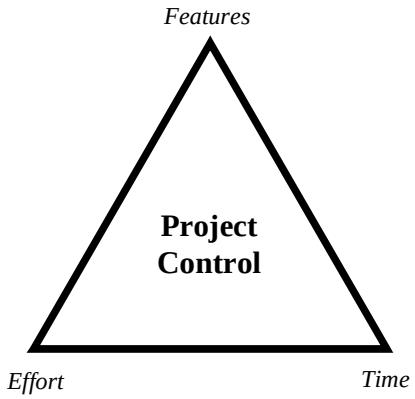


Figure 2. **Project** Control Triangle

For some cases, increasing *Effort* does not decrease *Time*. For *Time* to decrease when you increase *Effort*, **Organization** productivity ($OrganizationProductivity_n$) must increase. We will define $OrganizationProductivity_n$ as

$$OrganizationProductivity_n = n \times Productivity_n \quad (1)$$

where $Productivity_n$ is the nominal productivity of an individual in an **Organization**. Grady and Caswell[4] reported that at Hewlett-Packard they found the probability of 0.300 KNCSS/pm (Thousand Non-Comment Source Statements per person month) ≤ $Productivity_n$ ≤ 0.500 KNCSS/pm was 30% and of 0.250 KNCSS/pm ≤ P_n ≤ 0.600 KNCSS/pm was 60%. They cautioned managers that few **Project**s achieve greater than 0.900 KNCSS/pm. If you

try to decrease *Time* by adding people to a **Project**, inefficiencies cause $Productivity_n$ to decrease as n increases. We will show that $OrganizationPriductivity_n$ sometimes decreases as n increases.

We will concentrate on the control triangle. We will assume that the *Reliability* and *Usability* requirements of the finished **SoftwareProduct** are met. For *Reliability* and *Usability* prediction models see Simmons et al.[3] We will concentrate on the effects on a development **Project** of changing the *Features*, *Time*, and/or *Effort*.

Volume

A **SoftwareProduct Plan** describes the *Features* that the finished **SoftwareProduct** should contain. A requirements document often describes *Features* in terms understandable by the **Customer**s and the software development **Organization**. A design document describes how a **SoftwareProduct** will be created to implement the *Features*. As the number of *Features* grow, the size (or *Volume*) of the **SoftwareProduct** grows. Units for describing the *Volume* of a **SoftwareProduct** include source lines of code, source statements, non-comment source statements, delivered source instructions, function points, object points, chunks, and many more. Once the *Features* of a **SoftwareProduct** have been determined, there are many models for predicting *Volume*.[3]

Time

Simmons, et al.[3] list many empirically derived formulas for predicting the *Time* required to develop a **SoftwareProduct**. The nominal form of the development time estimation equation is

$$Time = a \times Effort^{b} \qquad (2)$$

where a and b are constants in the range $3.04 \geq a \geq 2.15$ and $0.38 \geq b \geq 0.32$.[3] The constant a is the *Volume* of a program that a person can produce in one month. Thus as the *Volume* of a program increases, the *Effort* to produce the program increases and the resulting development *Time* increases.

Observation:	When you add people to any **Project**, the nominal *Time* that it takes to produce the **SoftwareProduct** increases.

By adding people, how much can you actually compress nominal development *Time*? Figure 3 shows the *Effort* required to compress a development schedule for an example **Project**. You have a schedule range from Excessive Time to Impossible. If management were to allow a very long time, then you would reach a point where inefficiencies would set in and you would have excessive staff. Management usually would like for a **Project** to be completed as soon as possible and will not allow excessive time in the schedule. Figure 3 shows a Minimum Cost

region that for the example **Project** ranges from 37 months to 60 months. The nominal completion *Time* falls at the beginning of the inefficient range at about 37 months. Very seldom would a **Project** be allowed to span over three years. People would normally be added to compress the schedule resulting in slight inefficiencies. If you continue to add people, then the costs will begin to rise exponentially as you inter the Crash Project region resulting in major inefficiencies. The point that you continue to add people but do not reduce the development *Time* is the beginning of the Impossible range. This point we will call the point of maximum compression ($Time_{Maximum\ Compression}$). Boehm says his experience has shown that it is virtually impossible to compress the nominal scheduled development *Time* more than 25%.[5, 6] Conte, Dunsmore, and Shen[7] conclude that a good choice of **Project** duration should be about 10% less than nominal development *Time*.

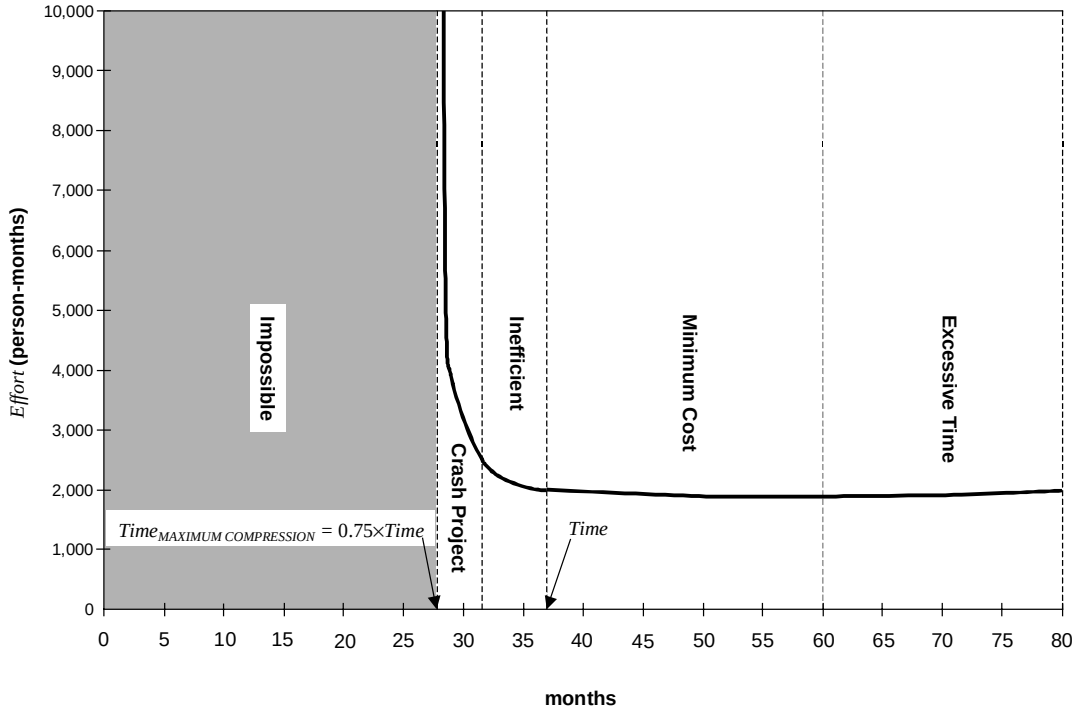


Figure 3. *Effort* Required to Compress a Development Schedule

Observations:

1. When software development contractors are paid by head count, there is no incentive to keep the headcount low.
2. Due to the pressure to complete a software development **Project** as soon as possible, an **Organization** tries to complete a **Project** in less time than the nominal *Time.*
3. Adding people to **Project**s only helps when you are trying to reduce the nominal development *Time* by up to 25%.
4. To avoid excessive waste, the scheduled development $Time_{Scheduled}$ should be in the range of $0.85 \times Time(Effort_{Nominal}) \le Time_{Sheduled} \le Time(Effort_{Nominal})$

McConnell claims that the average business systems **Project** overran its original schedule by 120%.[8] If this is true, then the average business systems **Project** is scheduled to be completed in the Impossible region. Often schedules are set based on budget cycles instead of the known *Time* that a **Project** should take. Actual time scheduled for a **Project** should be around the nominal *Time* predicted using a schedule prediction model calibrated to the business development environment. Once a **Project** is scheduled in the Impossible region, you can add 10 to 100 times the *Effort*, but still the **Project** would not be completed any sooner. The *Effort* expended to meet an impossible schedule would dominate **Project** costs. Simmons, et al.[3] define dominators as **Project** attributes that cause *Effort* (and productivity) to vary by an order of magnitude (or 10 to 1).

A key to effectively adding people to a **Project** is to create a **SoftwareProduct** design that can be partitioned into separate tasks like the tasks related to picking cotton. You can pick as much cotton as you have people to assign to the task. In the world of software development an example of a cotton picking easy **Project** (**Project**$_{\text{CottonPicking}}$) would be simple web site development. With the proper tools and knowledge, you can develop as many simple web sites as you have people available to work on web site development. There would be no need to complete cotton picking easy tasks in sequence. We will define f_S as the ratio of time that must be spent working on tasks in sequence to total time required for a single developer to complete a **Project**. For a **Project**$_{\text{CottonPicking}}$

$$f_S = 0 \tag{3}$$

There are tasks which cannot be partitioned for parallel development. The task of bearing a child takes nine months for a pregnant lady. There is no way to partition the task into subtasks where the subtasks can be developed in parallel. A **Project** that cannot be divided into subtasks for parallel development we will call a **Project**$_{\text{Pregnant}}$. An example of a **Project**$_{\text{Pregnant}}$ was a software system developed by a programmer named Ted that once worked at Bell Telephone Laboratories. He was assigned to an advanced technology group after he had developed a private branch exchange program that ran as a sub-program within a larger software system. His program was a table driven spaghetti code system with no documentation that anyone understood except Ted. We used Ted's **Project**$_{\text{Pregnant}}$ as a task for summer student interns who would try their best to document his system. Even though Bell Telephone Laboratories hired the top students interns from around the country, none of them could document Ted's software. A **Project**$_{\text{Pregnant}}$ like Ted's is one that cannot be partitioned into tasks that can be worked on in parallel. For a **Project**$_{\text{Pregnant}}$

$$f_S = 1 \tag{4}$$

A typical **Project** is neither a **Project**$_{\text{CottonPicking}}$ nor a **Project**$_{\text{Pregnant}}$. The range of f_S is for a typical **Project** is

$$0 < f_S < 1 \tag{5}$$

Many software development tasks are sequential in nature. Often you must wait to use a test laboratory or a conference room, for a bug to be found and fixed, etc. A typical **Project** may have an f_S of 0.3. How much can you improve $OrganizationProductivity_n$ by adding people to a **Project**?

We will answer that by defining $Speedup_n$ as the productivity of a team of n people compared to the productivity of a single person working alone.[3] In addition, we will define a communication factor f_C as the ratio of time spent communicating with other team members compared to the total time available for software development. For the case where there is no communication among team members, the *Speedup* is

$$Speedup_n\left(f_S, f_C = 0\right) = \frac{n}{1 + f_S \times (n-1)} \tag{6}$$

We will define $OrganizationProductivity_n$ for a team with n members as

$$OrganizationPrductivity_n = Productivity_{n=1} \times Speedup_n \tag{7}$$

where $Productivity_{n=1}$ is the average individual productivity for someone working alone. As n increases, $Speedup_n$ approaches a constant

$$Speedup_{n\to\infty} \to \frac{1}{f_S} \tag{8}$$

For a typical **Project** with an f_S of 0.3, the maximum $Speedup_{n\to\infty}$ for the entire team would be 1/0.3 or 3.333. In other words, no matter how many people you add to the team, the team productivity would be no more than 3.333 times the average productivity of a team member working alone.

The f_S is determined when the **Project Organization** structure and the work breakdown structure are determined. The number of tasks that can be worked on in parallel is a function of the system architecture and the modularization of the chunks of software. Once design is complete and work breakdown structure is determined, there is very little that can be done to change the f_S for a given **Project**.

Observations:	1.	$OrganizationProductivity_n$ of a team is limited by the value of $Productivity_{n=1} \div f_S$
	2.	For software development tasks to be as easy as picking cotton, f_S should be as small as possible.
	3.	$Time_{n\to\infty} \geq f_S \times Time_{n=1}$

The assumption we made that f_C is zero is unrealistic. Software development teams spend a considerable amount of time communicating with team members and non team members. In fact, a very hard working member of a software development team spends less than 5 hours of an 8 hour day developing software. The other 3 hours is spent in meetings, training, arriving a few minutes early and possibly leaving a few minutes late, coffee breaks, restroom breaks, , socializing with colleagues, etc. Thus approximately 62.5% of the workday is spent developing software.

Interruptions cause loss in productivity. The average duration of a work interruption is approximately 5 minutes for a typical programmer. The average time to regain a train of thought after an interruption is about 2 minutes. Thus, the average total time spent on a typical interruption of active software development is approximately 7 minutes or 2.33% of a workday (or 0.107% of the work month). For a software development team of eight members, two interruptions a month from each of the other team members would result in an f_C of 0.015. For teams that add members after a **Project** has started, two interactions per month are minimal due to the need to understand the status of the **Project**, current problems, and activities of other team members.

We can examine the effect of f_C on $OrganizationProductivity_n$ by assuming that we have a perfect design where every task can be developed in parallel resulting in an $f_S = 0$. Figure 4 shows what happens to $OrganizationProductivity_n$ as additional team members are added to a team. For a team with a fairly high communication factor of f_C=0.025 when a seventh member is added to a 6 member team, the $OrganizationProductivity_n$ decreases from $3.429 \times Productivity_{n=1}$ to $3.415 \times Productivity_{n=1}$. For each f_C there is a maximum number of team members that can be added to increase the $OrganizationProductivity_n$ of the team. Simmons, et al.,[3] determined that the maximum number of team members for a given f_C is

$$n_{MAXIMUM} = \frac{1}{\sqrt{f_C}} \tag{9}$$

Maximum compression occurs at $n_{MAXIMUM}$.

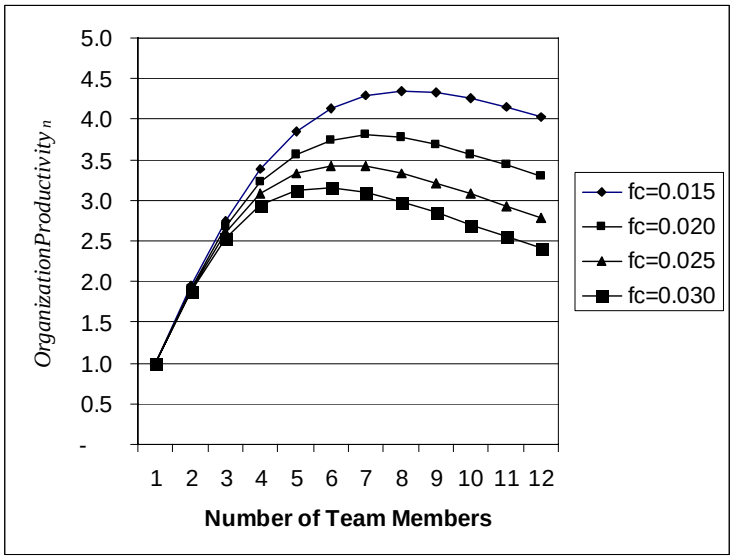


Figure 4. *OrganizationProductivity*$_n$ of teams that have different communication factors where $f_S = 0$.

As the amount of communications increase between team members, the *OrganizationProductivity*$_n$ of the team decreases for n greater than $n_{MAXIMUM}$.

Observations:

1. Adding person-power to a team **Project** makes it later when

$$n > \frac{1}{\sqrt{f_C}}$$

2. $$Time_{MaximumCompression}\left(f_S = 0,\, f_C > 0\right) = \left(2 \times \sqrt{f_C} - f_C\right) \times Time(n = 1)$$

Effort

We saw earlier that the nominal *Time* required to complete a **Project** is based on *Effort*. The nominal *Time* increases when *Effort* increases. We saw that if you increase *Effort*, the maximum compression $Time_{MaximumCompression}$ also increases. If your **Project** is operating near $Time_{MaximumCompression}$, then you cannot improve your schedule without adding large amounts of *Effort*. Adding large amounts of additional *Effort* increases both nominal *Time* and $Time_{MaximumCompression}$.

If a **Project** is not in the Crash Project region, then it can be completed sooner if we increase *Effort*. Prediction of *Effort* required for a successful **Project** is problematic. There are a number of cost drivers that cause required *Effort* to increase and *OrganizationProductivity*$_n$ to decrease. We will define the average productivity of a member of an n member **Organization** to be

$$Productivity_n = \frac{Volume}{Effort} \tag{10}$$

The productivity of the **Organization** would be

$$OrganizationProductivity_n = n \times Prodctivity_n \tag{11}$$

There are a number of attributes called dominators that can have a dominant effect on $OrganizationProductivity_n$ of a **Project.** One of the main causes of large **Project** failure is poor management. Poor management can cause chaos and results in large amounts of wasted *Effort* and very low $OrganizationProductivity_n$. More that 10 to 1 amounts of extra *Effort* could be expended before a **Project** is cancelled. Dominators like poor management usually result in reducing $OrganizationProductivity_n$ instead of improving it. Table 1 lists 19 **Project** attributes that can dominate $OrganizationProductivity_n$. Dominators are not independent of each other. A poor manager will probably also create an inadequate **Plan**. Also, some dominators are difficult to represent in *Effort* prediction model equations. For example, even though Boehm uses 15 cost drivers for his COCOMO model[5] and 23 for his COCOMO 2.0 model,[6] he was unable to represent management quality because of the difficulty of determining management quality ahead of time. The worst case prediction combining the extreme values of his multiplier adjustment factors could cause his estimate to vary by 808 times or almost three orders of magnitude. Thus, let us examine some of the dominators that can cause an *Effort* estimate to be incorrect.

Project Area	Dominator
Overall **Project**	**Plan** Development schedule constraints (*Time*) $K \times Salary_{average}$ Software life cycle (SLC) process
Organization	Management quality Lead designer Individual developers Average number of personnel (n) Personnel turnover Communications (f_C)
SoftwareProduct	*Volume* Amount of documentation Programming Language Complexity Quality (Includes required *Reliability* and *Usability*) Work breakdown structure (f_S)
Suppliers	Reuse
Customers	Interface Requirements volatility

Table 1. Software $OrganizationProductivity_n$ Dominators

The **Plan** is the reference for determining success of a **Project**. The **Plan** establishes **SoftwareProduct** $Features_{\mathbf{Planned}}$, $Volume_{\mathbf{Planned}}$, $Reliability_{\mathbf{Planned}}$, $Usability_{\mathbf{Planned}}$, $Salary_{average\mathbf{Planned}}$, $n_{\mathbf{Planned}}$, $Time_{\mathbf{Planned}}$, and many other items. Adding people to reduce completion *Time* when $Time_{\mathbf{Planned}}$ was incorrectly set to be completed in the Impossible range (See Figure 3) would definitely make the **Project** later than $Time_{\mathbf{Planned}}$. An incorrect **Plan** can cause excessive resources to be used to try to meet an impossible schedule. A **Plan** may cause a **Project** to fail when the $Features_{\mathbf{Planned}}$, $Reliability_{\mathbf{Planned}}$, and $Usability_{\mathbf{Planned}}$ do not meet the **Customer**'s expected $Features_{\mathbf{Expected}}$, $Reliability_{\mathbf{Expected}}$, and $Usability_{\mathbf{Expected}}$. Often large amount of additional *Effort* must be exerted to meet the **Customer**'s expectations which results in later **Project**s and lower $OrganizationProductivity_n$.

Over optimistic schedules are a real problem in software development. Brooks[1] states that more software **Project**s have gone awry for lack of calendar time than for all other causes combined. Attempts to meet an unrealistic schedule results in inefficient crash **Project**s that have no hope of meeting the schedule.

Cost of personnel resources is based on $Salary_{average}$ and the overhead and fringe benefit constant *K*. Software development is a people intensive activity. Cost of overhead, fringe benefits, developer workstations, etc. are all based on the number of people working on a **Project**. Thus we can approximate the cost of a software **Project** as

$$Cost = K \times Salary_{average} \times Effort \tag{12}$$

where *K* is a constant calibrated to the local costs, $Salary_{average}$ is the average salary paid **Project** personnel, and *Effort* is the person-months spent on the **Project**. For a typical **Project** *K* and $Salary_{average}$ are constant and relatively easy to measure. An alternate definition of productivity is

$$Productivity_n = \frac{Volume}{\left(K \times Salary_{average}\right) \times Effort} \tag{13}$$

While large well-established companies have high overhead and fringe benefits, these costs can be reduced by using contract labor that have low overhead and fringe benefits. $Salary_{average}$ can be reduced by requiring unpaid overtime and by subcontracting development tasks to regions of the world where $Salary_{average}$ is very low.

The software life cycle (SLC) can cause large variations in cost. High software $OrganizationProductivity_n$ can be realized when a simple SLC is used to develop software that is cotton picking easy to create. The $\mathbf{Project}_{\text{CottonPicking}}$ can use standard designs, straight forward work breakdown structures, and minimal

communications. Complex SLCs are necessary when developing large complex ultra reliable real time **SoftwareProduct**s like software that controls deep space probes. The SLCs for **Project**s at the Jet Propulsion Laboratory require as many as 10 phases, 38 deliverables, and 13 major milestone reviews. A major milestone review can cost $600,000 or more. Major inefficiencies can occur when a complex SLC is used for a simple **Project** or when a simple SLC is used for a complex **Project**. Selection of an improper SLC often results in failed **Project**s.

Management quality is a major factor leading to successful **Project**s. Boehm[5] says that poor management can increase software costs more rapidly than any other factor. Poor management often leads to excessive consumption of effort, low $OrganizationProductivity_n$, and **Project** failure.

The lead designer is an important factor in **Project**s. Great designers are as important to the success of a **Project** as quality managers. Brooks[1] says that study after study shows that the very best designers produce structures that are faster, smaller, simpler, cleaner, and produced with less effort. He says that the difference between a great designer and the average approaches an order of magnitude.

Individual developers are the foundation of a successful **Project**. Many studies have shown that individual developer productivity varies by more than 10 to 1. To assure success, you should do everything you can to get the best people working on your **Project**.

Increasing the number of **Project** personnel increases **Project** $OrganizationProductivity_n$ up to a point. Past that point, $OrganizationProductivity_n$ declines with each added person. We have shown that a team productivity declines when n exceeds $f_C^{-0.5}$. For large **Organization**s made up of many teams, $OrganizationProductivity_n$ increases up to the Crash Project region. Additional *Effort* applied to a **Project** in the Crash Project region can lead to chaos and reduced $OrganizationProductivity_n$.

Personnel turnover is a factor with which large **Project**s must cope. As long as a **Project** has average turnover, there is no reason for special concern. Only when a **Project** experiences turnover of key people or excessive turnover of other people does turnover begin to dominate $OrganizationProductivity_n$. Excessive turnover results in later **Project**s and lower $OrganizationProductivity_n$.

Communication among **Project** personnel is vital. As the communication factor f_C increases, the maximum $OrganizationProductivity_n$ decreases. From a team $OrganizationProductivity_n$ viewpoint, small f_C is

desirable. Every effort should be made to assure efficient communication among team members. Voice mail, email, electronic conferencing, and other communication tools should be used to improve communication efficiency.

Size or *Volume* of a **SoftwareProduct** is a major factor in determining *Effort*. Adding *Features* to a **SoftwareProduct** increases *Volume*, deleting *Features* decreases *Volume*. Increased *Volume* requires increased *Effort*, reduced *Volume* results in decreased *Effort*. **Project** managers can control **Project** *Effort*, *Time*, *Reliability*, and *Usability* by adding and/or deleting *Features*.

Documentation can be both a help and a burden. Useful documentation, like well written **Customer** requirements, can be helpful. Documentation that is never read is a burden. Government agencies that require as many as 38 different documents for every **Project**, even the $\textbf{Project}_{\text{CottonPicking}}$ easy ones results in unnecessary inefficiencies.

Use of a higher level programming language can help improve productivity. Table 2. lists programming language level for an assembler language, FORTRAN, C++, and graphic icons. If productivity is directly related to language level, then a graphic icon language would result in an 80 times increase in productivity over an assembler language. Realistically, only the coding and unit test phases would gain the entire improvement in productivity. For a sample constant function point **Project** using high level languages, Table 2 shows the *Volume*, *Effort*, *Time*, *n*, and. $OrganizationProductivity_n$ for a COCOMO Basic Semidetached *Effort* estimation model.[5] The SLC is assumed to spend 40% of the *Time* in the coding and unit test phases and 60% of the *Time* in the user requirements, design specification, system test, and V&V (validation and verification) phases. Also, the non coding and unit test phases are assumed to be independent of programming language. Each of the **SoftwareProduct**s developed is assumed to have the same functionality measured in Function Points. While the language level improves by 80 times the $OrganizationProductivity_n$ of the coding and unit test phases, the $OrganizationProductivity_n$ for the entire development SLC only improves by 1.49. As tool sets and **Organization** structures improve, the time spent in the non-coding and unit test phases will decrease and $OrganizationProductivity_n$ improvement will fall somewhere in the range between 1.49 times and 80 times. Use of a higher level programming reduces the nominal *Time* required to develop a **SoftwareProduct** and reduces the number of personnel required as shown in Table 2. Greater improvements than those listed in Table 2 will be realized when we improve software development tool sets to reduce *Effort* for non-coding tasks.

SoftwareProduct complexity is a major cost driver. The *Effort* required to create software for an ultra reliable computer embedded in a space probe is much greater that the *Effort* required for an easy **Project**$_{\text{CottonPicking}}$. Brooks[1] states that software entities are more complex for their size than perhaps any other human construct. He says that many development and management problems derive from complexity and its nonlinear increase with size. Most *Effort* prediction models use some form of complexity cost driver.

	Language Level	NCSS per Function Point	*Volume*		*Effort*			*Time*			*n*	*OrganizationProductivity*$_n$
Source Language			NCSS	Function Points	non coding person-months	coding and unit test person-months	total person-months	non coding months	coding and unit test months	total months		Function Points per project-month
Assembler	1	320	320,000	1,000	1,151	767.3	1,918	21.1	14.1	35.2	54	28.4
FORTRAN	3	107	107,000	1,000	1,151	225.0	1,376	21.1	9.2	30.3	45	33.0
C++	11	29	29,000	1,000	1,151	52.1	1,203	21.1	5.5	26.6	45	37.5
Graphic Icons	80	4	4,000	1,000	1,151	5.7	1,157	21.1	2.5	23.7	49	42.2

Table 2. Productivity for different programming language levels.

A quality **SoftwareProduct** should have a broad spectrum of **Customer** desired *Features*, have few defects, function efficiently, operate easily, and have satisfactory user documentation. A **SoftwareProduct** that functions efficiently, operates easily, and has satisfactory documentation reflects that **SoftwareProduct** is usable. The interaction of *Features*, *Reliability*, and *Usability* can be represented by the Software Quality Triangle in Figure 5. If we add *Features*, *Reliability* and *Usability* may decline. We can improve **SoftwareProduct** *Reliability* by removing defect riddled *Features*. We can improve *Usability* by improving *Reliability* and adding and/or deleting selected *Features*. A large amount of *Effort* must be expended to achieve ultra high **SoftwareProduct** quality.

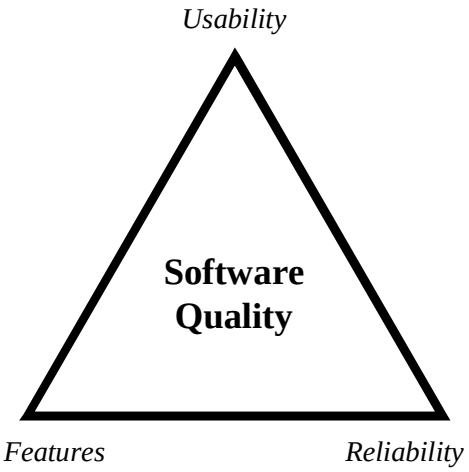


Figure 5. Software Quality Control Triangle

Reuse of software can lead to enormous increases in $OrganizationProductivity_n$. If software is completely reused with zero *Effort* expended on software development, then the $OrganizationProductivity_n$ would be very large. As software development *Effort* approaches zero, the $OrganizationProductivity_n$ approaches infinitive. Reuse is the one software $OrganizationProductivity_n$ dominator that can result in very large improvements in $OrganizationProductivity_n$. Many recent software engineering advances such as object oriented languages, component architectures, architectural patterns, application domain frameworks, and languages that support inheritance all are related to reuse.

Customer interface has proven to be a major productivity cost driver. **Project**s with **Customer** interface complexity much less than normal are many times as productive as ones that are more complex that normal. As the **Customer** interface complexity increases, the $OrganizationProductivity_n$ will decrease.

Requirements volatility can results in major amounts of *Effort* expended to keep up with requirement changes. Ideally, **Customer** requirements should be defined before a **Project** starts and remain stable until a **Project** is completed. In actual practice, **Customer** requirements change continually. While small changes in requirements can be tolerated, highly volatile requirements can lead to very low $OrganizationProductivity_n$ and a failed **Project**.

Conclusion

Completely successful **Project**s are rare. Government **Project**s are especially vulnerable to cost overruns and partially featured **SoftwareProduct**s because of the procurement culture used to select government

contractors. When contractors are paid by the size of the development **Organization** instead of the finished **SoftwareProduct**, this encourages bloated inefficient **Organization**s. Even attempts at process improvement are problematic in this environment. While some **Organization**s have instigated successful software process improvement programs, most have not. Norm Brown, executive director of the Department of Defense (DoD) Software Acquisition Best Practices Initiative, claims that organizational improvement has been occurring at a glacial rate.[9] The DoD Software Engineering Institute (SEI) has developed a five-level software capabilities maturity scale. Brown says that, according to the SEI, of 379 **Organization**s at 99 companies that have process improvement programs in place and have conducted SEI maturity assessments, 73% do not rate higher that level 1. It is not surprising that many **Project** s do not proceed as planned.

What action should be taken once milestones are missed and management realizes that future milestones will also be missed unless effective action is taken. You can remove $Features_{\mathbf{Planned}}$, reduce $Reliability_{\mathbf{Planned}}$, and/or relax $Usability_{\mathbf{Planned}}$ requirements. These type of actions may result in future milestones being met but will not increase the income to a contractor that is paid directly (or indirectly) by increasing *Effort*. Also, these actions may disappoint the **Customer**. Contractors that increase their income by increasing *Effort* may instinctively suggest that the first choice to missed milestones is to increase *Effort* (which of course is often paid for by the **Customer,**. not the contractor)

Arguments against increasing *Effort* to avoid **Project**s falling further behind when milestones are missed are:

1. Brooks's Law suggests that *adding manpower to a late software project makes it later.*
2. Increasing *Effort* for any **Project** that has a high f_C can actually make the **Project** take longer.
3. Increasing *Effort* for **Project**s that have a high f_S may not help reduce completion *Time*.
4. **Project**s where $Time_{\mathbf{Planned}}$ is in the Impossible or Crash Project regions (See Figure3) have very little chance of meeting future milestones by increasing *Effort*.
5. Increasing *Effort* results in longer nominal *Time*.
6. Increasing *Effort* results in longer $Time_{Maximum\ Compression.}$
7. Very few if any **Project**s have a $Time_{\mathbf{Planned}}$ in the Excessive Time or Minimum Cost ranges (See Figure 3). Those in the Inefficient or Crash Project ranges have an estimated maximum 25% possible

compression. If more than 25% compression is necessary for the **Project** not to be late, then increasing *Effort* will not be effective.

8. **Project**s may miss their milestones because of poor planning, impossible schedules, wrong SLCs, incompetent management and lead designers, incorrect *Volume* estimates, incorrect complexity estimates, improper application of reuse, underestimating **Customer** interface problems, and/or excessive requirements volatility. All of these **Project**s will probably be later if only *Effort* is increased.
9. **Project**s that fall behind because of excessive personnel turnover will probably fall further behind even if they replace those who have left.
10. Very few **Project**s rate at the top level 5 on the SEI CMM scale. Even if they do, **Project** milestones can be missed for many of the reasons mentioned above.

Once **Project** planning is complete and resources have been allocated and a **Project** is underway, there is no easily identifiable point where adding person-power to a late software **Project** does not make it later. There are very few cases where only increasing *Effort* will result in greater $OrganizationProductivity_n$ and shorter **Project** completion times. I agree with Brooks[1] that Brooks's Law is still the best zeroth-order approximation to the truth, a rule of thumb to warn managers against blindly making the instinctive fix to a late **Project**. There is no creditable evidence that would lead to repealing Brook's Law any time in the near future.

References

1. F. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition, Addison -Wesley Publishing Co., Reading, MA, 1995.
2. C. Jones, "Patterns of Large Software Systems: Failure and Success," *IEEE Computer*, March 1995, pp. 86-87.
3. D. Simmons, N. Ellis, H. Fujihara, and W. Kuo, *Software Measurement: A Visualization Toolkit for Project Control and Process Improvement*, Prentice Hall, Upper Saddle River, NJ, 1998.
4. R. Grady and D. Caswell, *SoftwareMetrics: Establishing a Company-wide Program*, Prentice Hall, Upper Saddle River, NJ, 1987.
5. B. Boehm, *Software Engineering Economics*, Prentice Hall, Upper Saddle River, NJ, 1981.
6. B. Boehm, E. Horowitz, R. Selby and J. Westland, *COCOMO 2.0 User's Manual*, Version 1.1, University of Southern California, CA, 1995.

7. S. Conte, H. Dunsmore and V. Shen, *Software Engineering Metrics and Models*, Benjamin/Cummings Publishing Co., Menlo Park, CA, 1986.
8. S. McConnell, "Brooks' Law Repealed," *IEEE Software*, November/December 1999, pp. 6-8.
9. N. Brown, "Industrial-Strength Management Strategies," *IEEE Software*, July 1996, pp. 94-103.