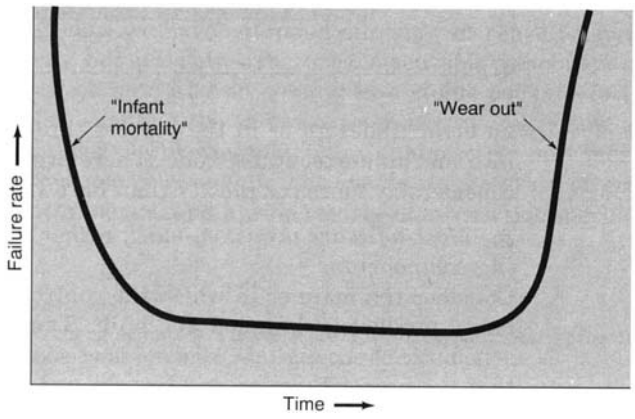
A Comparison of Software Production vs. that of other Engineered Products

2· Software doesn't "wear out. "

Hardware Failure Rates

The illustration below depicts failure rate as a function of time for hardware. The relationship, often called the "*bathtub curve*," indicates the typical failure rate of individual components within a large batch. It shows that in say a batch of 100 products, a relatively large number will fail early on before settling down to a steady rate. Eventually, age and wear and tear get the better of all them and failure rates rise again near the end of the products life. To assist in quality control, many new batches of products are 'soak' tested for maybe 24 hours in a hostile environment (temperature/humidity/variation etc.) to pinpoint those that are likely to fail early on in their life, this also highlights any inherent design/production weaknesses.



These early failure rates can be attributed to two things

- Poor or unrefined initial design. Correcting this, results in much lower failure rates for successive batches of the product.
- Manufacturing defects i.e. defects in the product brought about by poor assembly/materials etc. during production.

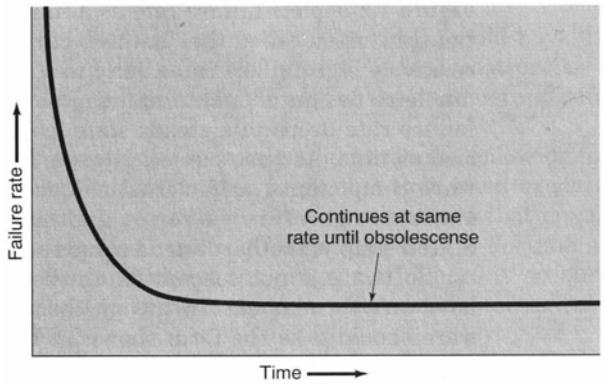
Both types of failure can be corrected (either by refining the design, or by replacing broken components out in the field), which lead to the failure rate dropping to a steady-state level for some period of time.

As time passes, however, the failure rates rise again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes and many other environmental maladies. Stated simply,

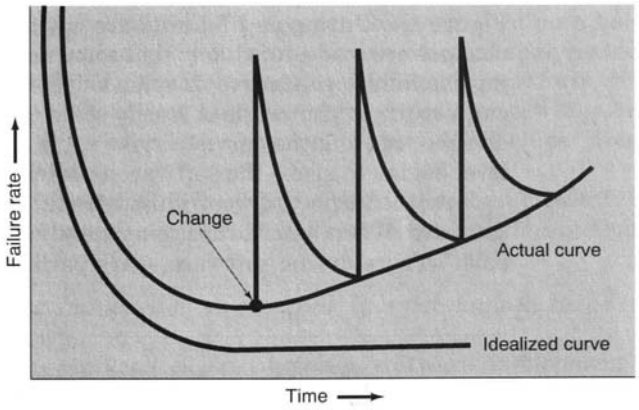
"...The hardware begins to wear out."

Software Failure Rates

Software is not susceptible to the same environmental problems that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form shown below.



Undiscovered defects in the first engineered version of the software will cause high failure rates early in the life of a program. However, these are corrected (hopefully without introducing other errors) and the curve flattens as shown. The implication is clear. *Software doesn't wear out*. However, it does *deteriorate* with maintenance as shown below.



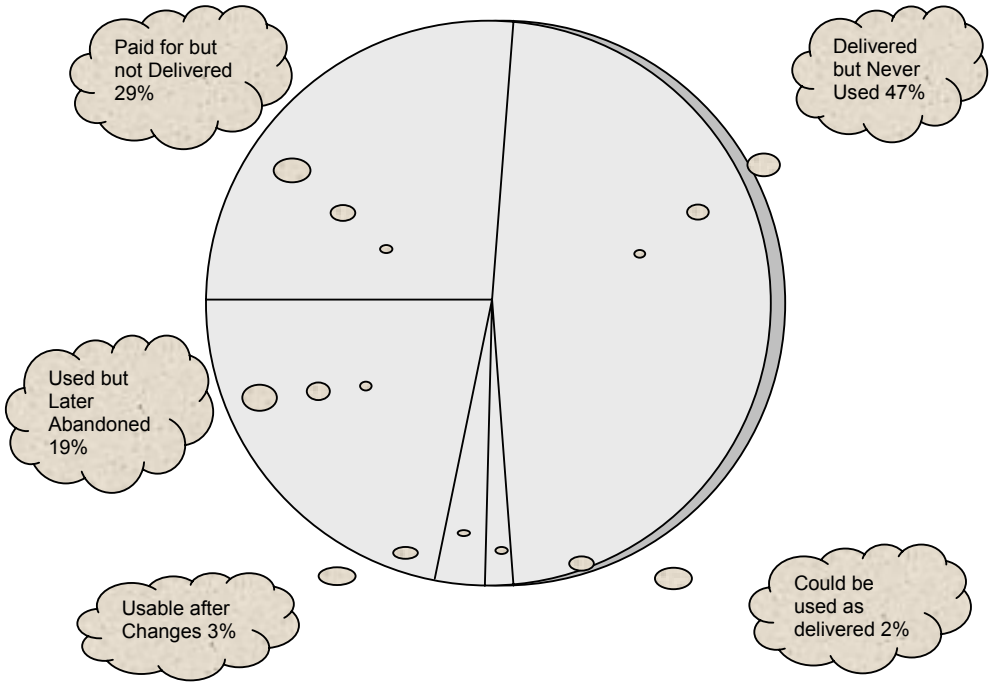
During its life, software will undergo changes and it is likely that some new defects will be introduced as a result of this, causing the failure rate curve to spike as shown above. Before the curve can return to the original steady-state failure rate (i.e. before the new bugs have been removed), another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise-- *the software is deteriorating due to change*.

The Software Crisis/Chronic Affliction

In the late 70's early 80's software development was still in its infancy, having had less than 30 years to develop as an engineering discipline. However, the cracks in the development process were already beginning to be noticed and software development had reached a crisis. Some would describe it as a "*Chronic Affliction*" which is defined as

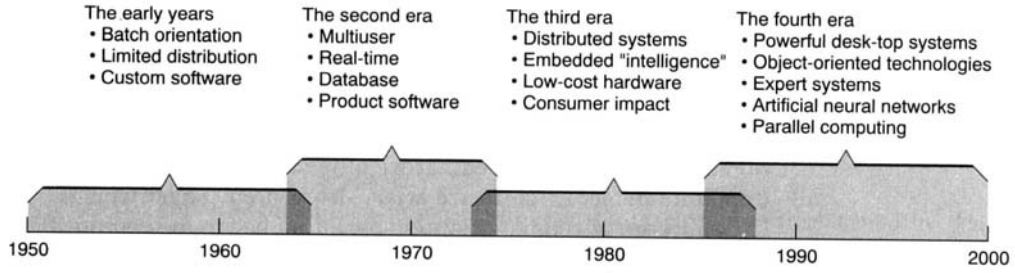
> *"…Something causing repeated and long lasting pain and distress*"

In effect, the use of software had grown beyond the ability of the discipline to develop it. In essence the techniques for producing software that had been sufficient in the 50's, 60's and perhaps 70's were no longer sufficient for the 80's and 90's and there was a need to develop a more "*structured approach*" to software analysis, design, programming, test and maintenance. This can be seen by the graph below (seen earlier), that shows the suitability of software procured by the American Department of defence.



To understand how this problem arose, compare the situation in the 60's with the situation we have today.

The Evolving Nature of Software Development



Early 50's/60's

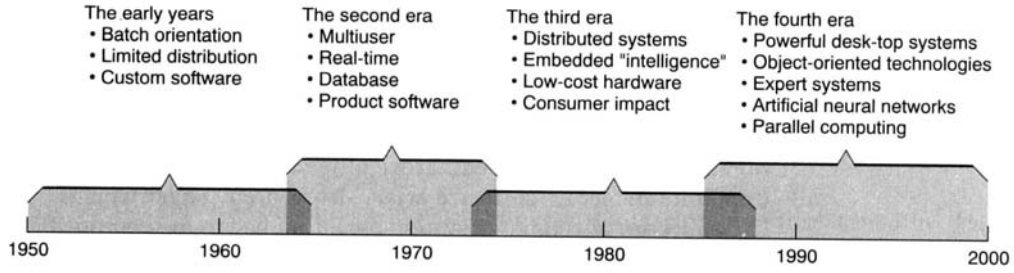
1. In the early days the *customer* was the developer of the application, they wrote the program, fixed it and maintained it when it broke down. Because of low staff mobility in a fledgling market, companies felt secure that they could maintain software using existing staff, thus there was no need for too much design and documentation.

2. Software was often only a few hundred lines of batch processing, perhaps analysing a mathematical problem e.g. statistics. Remember that calculators hadn't evolved yet and the slide rule was the most common method of calculation.

3. The software was *tailor made* and never went outside the company. Because the software was so small and specialised, the need to distribute it never arose.

4. Software of the day ran on large mainframes, whose hardware cost far outweighed the cost of any software development. Large disks and memory storage was virtually unheard of. Programs were loaded from punched paper tape or cards. The operating system was often booted via switches on the front panel, with just enough code being entered to allow the rest of the system to be loaded from paper tape etc. Storage was usually either punched paper tape or, in later years, magnetic tape. Printouts were slow 3 characters per second teletypes if you were lucky.

5. Programming was a "*seat of the pants*" technique without discipline or design techniques where anything could be tried and justified if it got the application to work, this was like trying to build a bridge out of matchsticks, as long as it stayed up, who cared what it looked like. In essence it was *hacking* and practitioners of the day, often academics wearing white overcoats, revelled in the mystique and aura that surrounded any job title relating to computing.



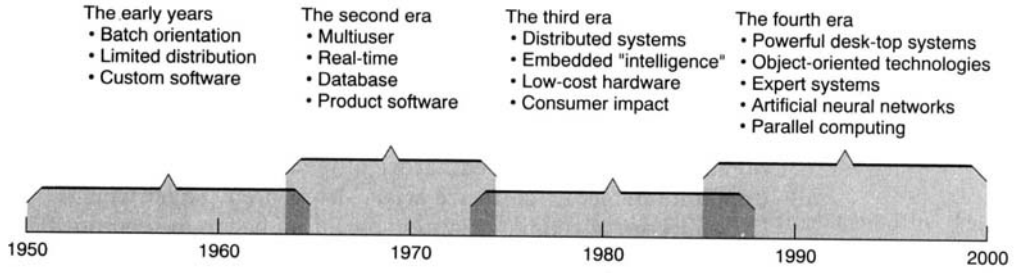
Mid 60's/70's

As the cost of hardware fell and the technology and power of the computer developed to the point where it was able to run more sophisticated programs. New operating systems such as UNIX, amongst others, evolved to take advantage of developments in processing power, memory, disk storage and of course computer science. These systems allowed the processing capabilities of the computer to be *shared* by several users in a multi-user, multi-tasking environment.

More interactive approaches to programming and data processing evolved principally through the use of VDU's, printers and online disk storage. New applications such as databases and mathematical analysis packages evolved. New languages sprang forth to harness the power of the new technology and make software creation more productive.

Real time applications became feasible, where the power of the computer could be used to solve problems requiring an extremely rapid response to changes of data, e.g. process control, rockets etc.

With developments in IC technology increasing at a fantastic pace, the early 70's saw Intel developed the first programmable logic controller '4004' which later became known as the microprocessor. New ram and I/O devices rapidly followed allowing for the first time, small cheap computers to be used in control applications and intelligent instruments. The 1st electronic calculator was born based around an Intel chip.



Mid 70's/80's

Microprocessor technology evolved and their application and use became more wide spread, distributed or networked systems became more common allowing microprocessors to control vastly more sophisticated processes.

Small dedicated microprocessors evolved into complete 'computers on a chip' offering single chip solutions to many data processing control applications.

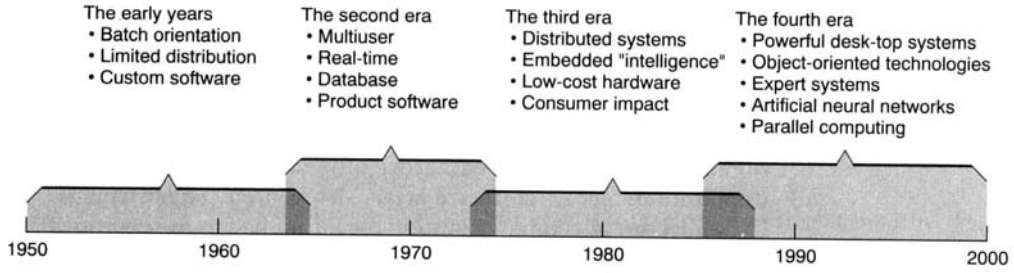
It is estimated that today, such single chip microprocessors out number processors such as the Pentium by at least 100:1. For example, the typical modern day luxury Motor Car typically has 12 or more microprocessors. Here are some examples

- ABS,
- Traction Control,
- Engine Management/Emission Control,
- Automatic Gearbox Control,
- Air conditioning/Climate Control,
- Radio/CD,
- Satellite navigation,
- Engine/vehicle condition monitoring,
- Active suspension,
- Intelligent instrumentation,
- Seat memory,
- Alarm and Immobiliser

New intelligent instruments and controller were designed to take advantage of the small compact microprocessor, e.g. lifts, traffic lights, machinery etc and the effect on the consumer society was just beginning to take off. Today microprocessors are commonplace in all the following systems.

Burglar alarms, Television, Video recorders, Camcorders, Calculators, Telephones, Clocks, Watches, Washing Machines, Central Heating Controllers, Hi-Fi.

Even the PC has at least half a dozen microprocessors (beside the Pentium) to control things like keyboard, monitor, printer, scanner, disk drives, modem, tape streamers, force feedback joystick/steering wheel, sound/midi/games ports etc.



Current situation - early 2000

Today the situation is very different to the early 50's/60'. In summary:

1. The customer is almost **never** the person who develops the software. More and more software is "*outsourced*" to other companies to develop, which are often composed of teams of developers. The need for discipline, communication between developer and customer, documentation and a thorough understanding of the specification is essential.

2. Software complexity has escalated and is now frequently *multi-tasking*, often *real-time* and sometimes *embedded* into dedicated systems e.g. fuel injection in car systems etc. Programs with 100,000+ lines of code are the "norm" rather than the exception. User interfaces are more sophisticated with graphics and mouse based "windows" application replacing simple dumb terminal.

3. Software breadth has grown, as there has been a trend to develop more *specialised* **and** more *general* software. The availability of new hardware has facilitated the creation of completely new kinds of applications e.g. Distributed systems, Web programming, e-mail, reusable libraries and components.

4. Software is now written with mass appeal, so that millions of copies can be sold, e.g. spreadsheets, databases, wordprocessors etc.

5. The cost of software development now vastly exceeds the cost of hardware.

6. Software evolves more and is used for longer than before. Maintenance of software is now far more important. Witness the Millennium bug introduced 30-40 years ago.

All of these developments have meant that the techniques of the 50/60's **no longer** work. Software development times cannot simply be scaled up and there is not a linear relationship between the number of lines of code and the time scale of the project.

Problems with Software Development

What factors have contributed to the crisis?

- Manager's estimates of productivity, cost and time scales for software development have proven time and time again to be wildly inaccurate. Cost overruns and slipped schedules of a factor of x10 are not uncommon. Furthermore, Middle or Upper managers with no experience of software were often given the job of managing it and adopted project management techniques that were wholly inappropriate for large-scale software.
- In the early days, project management costing estimates all too often centred on the cost of the hardware, as this easily dwarfed the cost of software development. Inaccurate software development estimation could easily be swept under the carpet and offset against an expensive piece of hardware. These days this is almost impossible and a more accurate analysis of software development costs is required.
- There was no accurate way to gauge the productivity of a programmer. Managers used the crude "*count the number of lines produced per day*" formulas as a guide to how productive their programmers were, regardless of the complexity of the software or how many bugs that code had. In essence programmers were forced to hack code rather than design/engineer a complete system.
- Quality of software was difficult to judge against a specification? Such metrics are only just beginning to be developed using mathematical specifications such as 'Z' to allow comparisons between specification and implementation.
- The productivity of software developers has failed to keep pace with the demands for their services, leading to software being rushed and delivered early, often exacerbated by fierce competition and the need to be first to market with a new product. Short cuts may have been taken in the development and the product was probably inadequately specified, designed and tested. The customer was then left to highlight (or debug) the software's inadequacies, leading to a lack of confidence in the product and customer dissatisfaction.
- Customers often assumed that the programmer knew what was required, while programmers assumed that customers would be happy with whatever was delivered. In essence, customer and programmer did not *talk* to each other to thrash out the specification accurately enough. In *essence "Communication between Customer and developer was poor*".
- Existing software could be extremely difficult to maintain, devouring millions of pounds and months/years of effort. The maintenance of software was often not considered (or given a high enough priority) when embarking on any new software development.
- Many programmers often adopted "*The structure is in my head*" approach to disguise the fact that they are making it up as they go along and that the structure was changing daily. When they left, that structure went with them.

Software Myths and the Software Crisis

Many of these software development problems can be traced to a number of mythologies that arose during the early history of software development, which propagated misinformation and confusion.

Software myths appear to contain *reasonable* statements of fact and have an almost intuitive feel to them and which experienced practitioners who "*knew the score*" often promulgated.

Today, most knowledgeable professionals recognise myths for what they are - misleading attitudes that have caused serious problems for managers and technical people alike. However, old attitudes and habits are difficult to modify, and remnants of software myths are still believed as we move toward the fifth decade of software.

Customer Myths

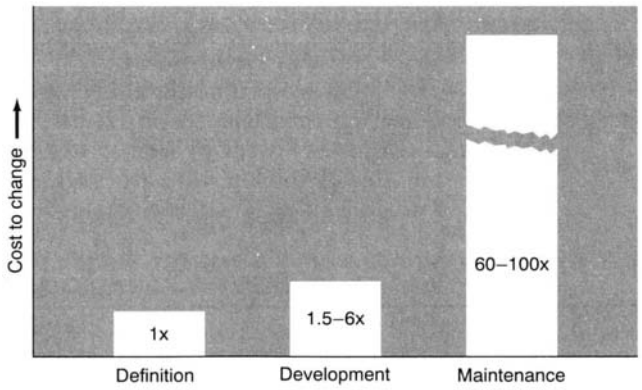
In many cases, a customer believes myths about software because software responsible managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and, ultimately, to dissatisfaction with the developer.

Customer Myth 1: *A general statement of objectives is sufficient to begin writing programs--we can fill in the details later.*

Reality: Poor up-front definition is the major cause of failed software efforts. A formal and detailed description of information domain, function, performance, interfaces, design constraints, and validation criteria is essential. These characteristics can be determined only after thorough communication between customer and developer.

Customer Myth 2: *Project requirements continually change, but change can be easily accommodated because software is flexible.*

Reality: It is true that software requirements do change, but the impact of change varies with the time at which it is introduced. The illustration below demonstrates the impact of change. If serious attention is given to up-front definition, early requests for change can be accommodated easily. The customer can review requirements and recommend modifications with relatively little impact on cost.



When changes are requested during software design, cost impact grows rapidly. Resources have been committed and a design framework has been established.

- Change can cause upheaval that requires additional resources and major design modification, i.e., additional cost.
- Changes in function, performance, interfaces, or other characteristics during implementation (code and test) have a severe impact on cost.
- Change, when requested late in a project, can be more than an order-of-magnitude more expensive than the same change requested early.

Programmers Myths

Myths that are still believed by software programmers have been fostered by four decades of programming culture. As we noted earlier, during the early days of software, programming was viewed as an art form surrounded by *mystique*. Old ways and attitudes, die-hard.

Programmer Myth 1: *Once we write the program and get it to work, our job is done.*

Reality: Someone once said "*the sooner you begin 'writing code,' the longer it'll take you to get it finished.*" Industry data indicates that between 50 and 70 percent of all effort expended on a program will be expended **after** it is delivered to the customer for the first time.

Programmer Myth 2: *Until I get the program "running" I really have no way of assessing its quality.*

Reality: One of the most effective quality assurance mechanisms can be applied from the inception of a project -- the formal technical review. Software reviews are a "*quality filter*" that have been found to be more effective than testing for finding certain classes of software defects.

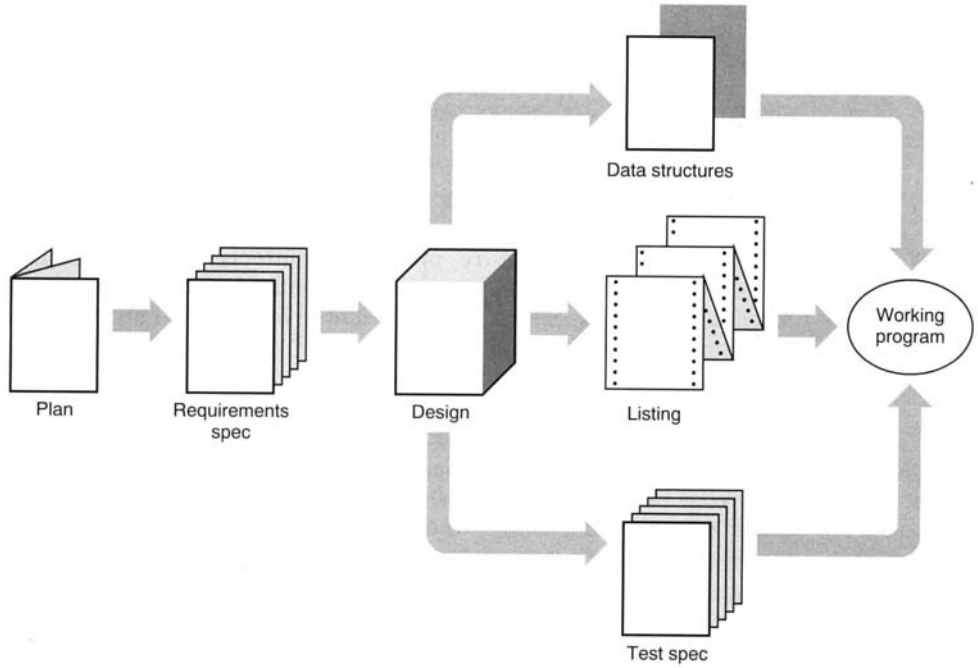
Programmer Myth 3: *I can't justify spending 20-30% of the project time scale doing analysis, I'll need that time at the end to sort out all the bugs.*

Reality: If there isn't time to do it right, there certainly isn't time to do it wrong. Experience has shown that developers often rush the analysis because they fear that they will need the time at the end to sort out the bugs which, had they done the analysis in the first place would not have occurred. A case of catch-22.

Programmer Myth 4: *The only deliverable for a successful project is the executable program.*

Reality: An executable program is only one part of a software configuration that includes all the elements illustrated below. Documentation forms the foundation for successful development and, more importantly, provides guidance for the software maintenance task.

As Page-Jones says, "*trying to analyse the structure of a program when all you have is the source code listing, is akin to analysing the structure of an elephant while stood 2 inches away from it, all you can tell is that it is grey*". The illustration below gives some indication of the documentation that will be required for successful maintenance.



Software Management Myths

Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

Management Myth 1: *We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?*

Reality: The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software development practice? Is it complete? In many cases, the answer to all of these questions is "no."

Management Myth 2: *My people do have state-of-the-art software development tools; after all, we buy them the newest computers.*

Reality: It takes much more than the latest model mainframe, workstation, or PC to do high-quality software development. Computer-aided software engineering (CASE) tools are more important than hardware for achieving good quality and productivity, yet the majority of software developers still do not use them.

Management Myth 3: *If we get behind schedule, we can add more programmers and catch up*

Reality: Software development is not a mechanistic process like manufacturing. In the words of Brooks:

> *"... Adding people to a late software project makes it later."*

At first, this statement may seem counter intuitive. However, as new people are added, people who were working on the project must now stop what they are doing and spend time educating the newcomers and bringing them up to speed so that they can be of assistance, thereby reducing the amount of time spent on productive development effort. People can be added, but only in a planned and well co-ordinated manner.