**Mythical man month**

**Chapter 1: The tar pit**

**The programming systems product**

Having a program as programming product, can be run tested repaired and extended by anybody, generalized costs 3 times than just a product

Having a program in a programming system (collection of interacting programs). Every input/output conforms in syntax and semantics with precisely defined interfaces, uses prescribed budget of resources, tested with other system components in all combinations. It is subtle due to component interaction bugs. It costs 3 times more than anormal product

A programming systems product differs from a normal product in all the ways listed above, truly useful and costs 9 times more than a normal program

**The joys of the craft**

Programming is fun because there is the creation of something useful that you can see it and it feels it was built from scratch

**The woes of the craft**

It needs to be perfect to work (against human nature)

One rarely controls the circumstances of his work, or their goal (other people set them)

Debugging has a linear or even quadratic convergence

Better programs can come after and take place (ours becomes obsolete)

**Scale of programmers** from -10 to +10

If you want to go higher in the scale, you should be humble and have good communication skills. If you are intelligent but arrogant you may go quickly to the negative (damaging the project)

**Chapter : The mythical man-month**

**Optimism**

Programmers (young mostly) are too time optimistic.

The inconsistencies and incompleteness of our ideas become clear only during implementation.

In a single task, things might go well without delays. But in chained tasks, the probability of each working becomes vanishingly small

**The man-month**

When a task cannot be partitioned because of sequential constraints, adding more programmers won’t affect the schedule

Tasks that can be partitioned but require communication take less time when there are more people but not inversely proportional due to the effort put into communication

Intercommunication increases effort by n²/2, so if you exceed the number of programmers, things get counterproductive

**Systems test**

Rule of thumb for scheduling a software task: 1/3 planning, 1/6 coding, ¼ component test and early system test, ¼ system test with all components in hands

**Gutless estimating**

Failure to allow enough time for system testing is disastrous, causes delay without warning, unsettling to customers and managers

**Regenerative schedule disaster**

Suppose a task is estimated at 12 man-months, assigned to 3 men (estimate time 4 months). First task takes 2 months instead of 1. Still 9 m-m effort. Solutions

1. Assume task must be done on time, only first part was misestimated. Add 2 men to the 3 assigned
2. Assume task must be done on time, whole estimate was uniformly low. Add 6 men to the 3 assigned
3. Reschedule, allow enough time to ensure that the work can be carefully and thoroughly done, rescheduling won’t have to be done again
4. Trim the task, formally and carefully by hasty design and incomplete testing

First 2 sol°, estimating that the task must be done on time is disastrous. Consider regenerative effects like the time of recruitment and training, + different partitioning. Adding manpower toa late software project makes it later.

This demythologizes the man-month. Nb of months depends upon its sequential constraints. The max nb of men depends upon the nb of independent subtasks

**Chapter 3: The surgical team**

**The problem**

Many want a small sharp team of only good productive programmers which by common consensus should not exceed 10. Otherwise, you need more support in finance

If a 200-man project has 25 competent managers, fire the 175 troops and put managers back to programming.

But these rules do not apply for very big systems because they will need too much time that the product becomes obsolete after that period

How can these two controversial issues be solved?

**Mill’s proposal**

Few minds design but many manpower to bear. Divide teams similar to surgical teams composed of:

The surgeon: a chief programmer, designs codes and documents, should have a great experience and talent

The copilot: able to do any part of the job but is less experienced, shares in the design as a thinker, discussant and evaluator. Represents his team in discussions of function and interface with other teams, should know the code intimately, research alternative design strategies. May write code but not responsible for it

The administrator: The surgeon is still boss, but the administrator handles money, space etc... Can serve 2 teams if possible

The editor: Handles the documentation draft from the surgeon, criticizes, reworks and provide reference for it etc...

Two secretaries: each for editor and administrator

The program clerk: maintaining technical records of the team in a programming-product library. Relieves programmers of clerical chores and enhances team’s most valuable assets

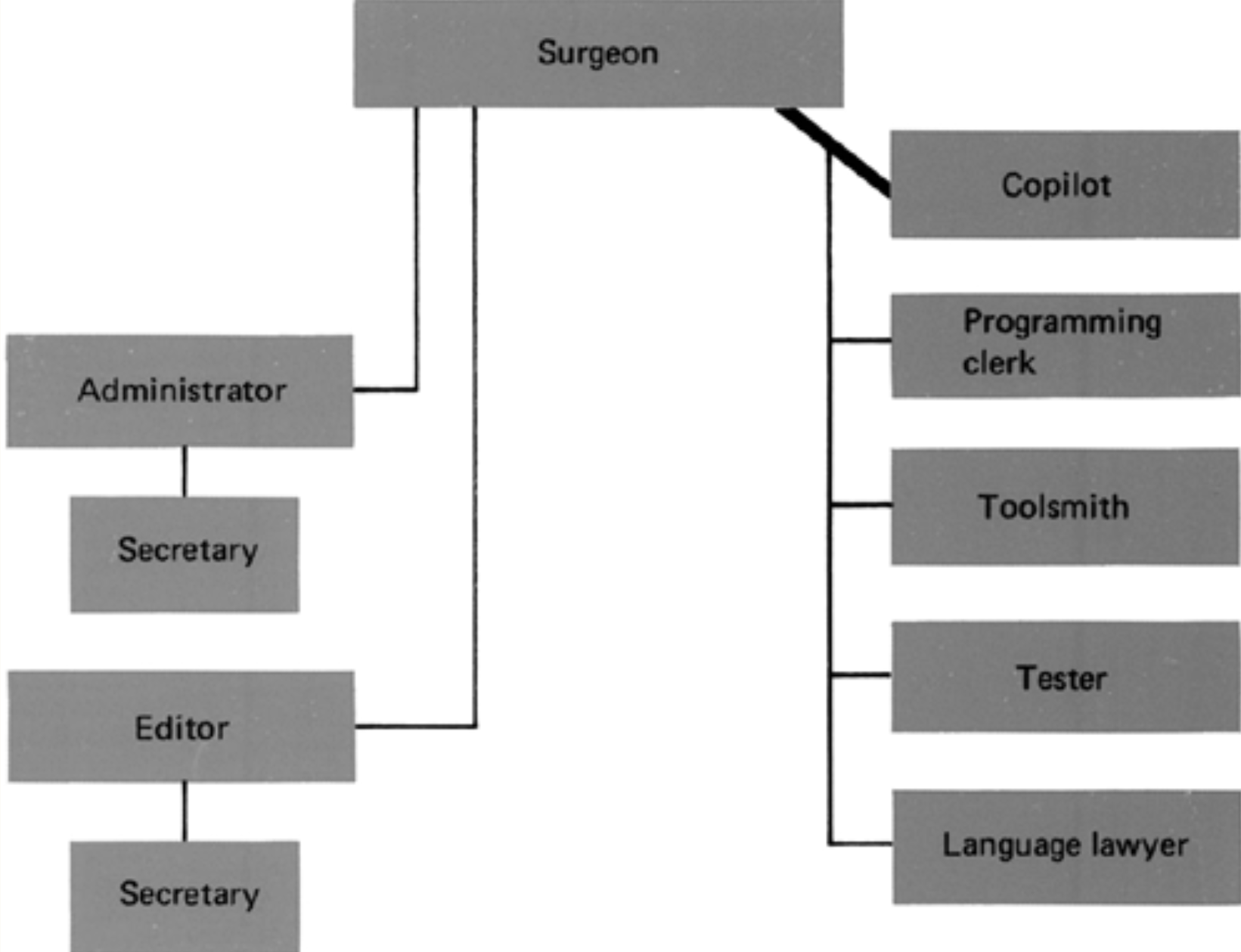
The toolsmith: responsible for ensuring the adequacy of the basic service and for contructing, maintaining and upgrading special tools mostly interactive computer services

The tester:

The language lawyer: can find neat and efficient way to use the language and do difficult things, he is not the brilliant mind of the team as the surgeon but the one who masters well the programming language

**How it works**

Unlike conventional teams where all people are equal, the surgeon decides unilaterally, there are no differences of interest, lack of division of the problem and superior-subordinate relationship makes it possible for the team to act uno animo



**Scaling up**

The conceptual integrity of each piece has been radically improved, the # minds determining the design has been reduced from 200 to 20, those of the surgeons.

But the entire system also must have conceptual integrity, there should be an architect to design it all

**Chapter 16: No silver bullet—Essence and Accident in Software engineering**

**Abstract**

Suggestions for essential parts of the software task:

* Exploit mass market to avoid constructing what can be bought
* Rapid prototyping
* Growing software organically (add more functionality to systems as they are run, used and tested)
* Identifying and developing the great conceptual designers of the rising generation

**Introduction**

Did not understand much of what he said here

**Does it have to be hard? —Essential difficulties**

Software progress is relatively slow to hardware progress (20th century of course)

The hardest part of building software is the specification, design and testing of its conceptual construct, not the labor of representing it and testing the fidelity of representation

Inherent properties of irreducible essence of modern software systems:

* **Complexity:** Software products are more complex than most of the things humans build, they have very large # states. Scaling up is not merely a repetition of the same elements in larger size, it increases the # different elements, and the complexity rises more than linearly. Complexity is an essence for a software not an accident.

Technically, from the complexity rises difficulty of communication between team members, of extending programs w.o flaws and side effects and the unvisualized states that constitute security trapdoors. In management, complexity makes overview hard

In management, complexity makes overview hard and creates the learning and understanding burden

* **Conformity:** Much of the complexity we must master is arbitrary complexity, forces w.o reason by the different human institutions and systems to which interfaces must conform, not only one God
* **Changeability:** Software entity is constantly subject to pressures for change, sometimes from users who like the basic function and invent new uses for it especially if the software is successful. Also, successful software must survive changes in the industry
* **Invisibility:** software is invisible and unrealizable, has no geometric representation, thus depriving the mind of some of its most powerful conceptual tools

**Past breakthroughs solved accidental difficulties**

* **High-level languages:** Gains in reliability, simplicity and comprehensibility. It frees a program from much of its accidental complexity. High level languages offer all the constructs the programmer imagines in the abstract program.

But language approaches are getting closer to the sophistication of users and at some point, the elaboration of high-level language becomes a burden that increases the intellectual task of the user who rarely uses esoteric constructs

* **Time-sharing:** preserves immediacy, hence enables to maintain a complexity-overview. It also shortens the systems response
* **Unified programming environments:** Provide integrated libraries, unified file formats, pipes and filters. As a result, conceptual structures that call and feed each other can do so more easily

**Hopes for the silver**

Considering the tech developments that are often advanced as silver bullets:

* **Ada and other high-level language advances:** Philosophy of modularization, abstract data types and hierarchical structuring
* **Object-oriented programming:** Two important concepts: there are maybe hierarchies w.o hiding and hiding w.o hierarchies. Both allow to express the essence of design w.o having to express large amount of syntactic material that add no new information content
* **AI:** (I guess that this critique is outdated)
* **Expert systems:** are programs containing a generalized inference engine and a rule base.

Inference engine tech is developed in an app independent way , then applied to many users

The changeable parts of the app peculiar materials are encoded in the rule base in a uniform fashion, and tools are provided for dev., changing, test. and doc. the rule base

Most imp advance offered is the separation of the app complexity from the program

The most powerful contribution of expert systems will surely be to put at the service of the inexperienced programmer the experience and accumulated wisdom of the best programers

* **Automatic programming:** The apps have very favorable properties:

The problems are readily characterized by relatively few params

There are many known methods of solution to provide a library of alternatives

Extensive analysis has led to explicit rules for selecting solutions techniques given problem params.

* **Graphical programming:** The writer does not agree with its relevance because the software systems are generally much more complex than we can represent them graphically to give a good and complete overview about the system
* **Program verification:** Program verification might reduce program-testing load but cannot eliminate it. It is a very powerful concept and is very important to secure OS kernels for ex., but the technology does not promise to save labor
* **Environments and tools:** The most they can offer is freedom from Syntactic and small semantic errors. Biggest gain is the use of integrated DB systems to keep track of details that must be recalled accurately by the individual programmer and kept current in a group of collaborators on a single system
* **Workstations:** boost is IPS is always welcome, can offer magical enhancements that we cannot expect

**Promising attacks on the conceptual essence**