Frederick Brooks, The Mythical Man-Month: Essays on Software Engineering, A. Wesley, 1995

Chapter 1—The Tar Pit

A program becomes a programming product when it becomes a product that can be run, tested, repaired and extended by anybody. It must be written in a generalized fashion and thoroughly tested. It must also be well documented. It costs about 3 times as much as a debugged program with the same function. A program becomes a programming system when it becomes a collection of interacting programs so that it becomes an entire facility for large tasks. Every input and output conforms to some syntax and semantics. It must also be thoroughly tested and debugged. It also costs at least 3 times as much as a standalone program with the same function. A programming product or a programming system can become a programming systems product, which costs 9 times as much. It is the true useful object of programming efforts. The joys of the craft—the joy of making useful and complex things, programming system and the joy of learning. The woes—need perfect performance, fulfill other people's objectives, depend on others, testing and debugging can drag on and then the product becomes obsolete too quick. The technological base on which one builds is always advancing. Programming is both a tar pit and a creative activity.

Chapter2—The Mythical Man-Month (good cooking takes time)

Software is often delayed either because lack of good estimating techniques, confusing effort with progress, uncertainties, poor monitoring of progress, and the natural response to add man power when behind schedule. All programmers are optimists because they build from pure thoughts, concepts and representations, but our ideas are faulty (we have bugs) making our optimism unjustified. The assumption that things will go well in a single task has a probabilistic effect on the schedule. No delay has a finite probability, which gets compounded when the programming effort consists of many tasks. The man-month as a unit of measurement for job size is a dangerous and deceptive myth. It implies that men and months are interchangeable. This is only possible when tasks can be partitioned among workers with no communication among them. Perfectly partitionable and non-partitionable tasks are two extremes, with most software projects falling somewhere in between. Communication effort needs to be added to less than perfectly partitionable tasks, which consists of training and intercommunication (n(n-1)/2). We also expect the number of bugs to be smaller than they actually are, thus testing is the most mis-scheduled part. It is recommended: 1/3 planning, 1/6 coding, 1/4 component testing and 1/4 final testing (1/2 devoted to debugging and testing, coding is the shortest). Most projects are on schedule until testing. When bad news come late it delays the entire project. ***Brook's Law: Adding manpower to a late software project makes it later.*** This is so because of the additional time needed for training and intercommunication. This demythologizes the man-month. The number of months depends on the linear sequential constraints. The number of people depend on the number of independent sub-tasks.

Chapter 3—The Surgical Team

There is a wide variation in programmer productivity, as much as 10 to 1 between best and worst. Thus, if a 200-person project has 25 managers who are the most competent programmers, fire the other 175 and put the managers to program. However, 25 programmers will not be sufficient for large projects. A dilema: for efficiency and conceptual integrity one prefers a few good minds. For large systems one needs considerable manpower. The solution is similar to Mills' surgical team ideas. Have a few surgeons and enough manpower to support the surgeons. Mills calls the surgeon the ***chief programmer***, who defines functional and performance specs, designs the program, codes it, tests it and writes its documentation. The ***copilot*** is the chief programmer's alter ego who shares in the design as a thinker. Other staff include the administrator, the editor, two secretaries, and a program clerk who keeps all technical records, computer input, status notebook, and chronological archives. Mills suggests converting all programming activities from private to public. In addition, there is a toolsmith (constructs, maintains and upgrades special tools), a tester (who develops test cases from functional specs), and a language lawyer. The system is the product of one mind, or two at most, and all differences of judgement are settled by the surgeon. A large programming team can be divided into surgical teams of 7 or so as described here.

Chapter 4—Aristocracy, Democracy, and System Design

Conceptual integrity is the most important consideration in system design. It is better to have an imperfect design but one set of ideas, than having several independent and uncoordinated good ideas. The ultimate test of system design is its function to complexity ratio. Neither function alone nor simplicity alone defines good design. That is, for a given level of function, the simplest and most straightforward system is the best. Simplicity alone is not enough, but simplicity and straight forwardness proceed from conceptual integrity. Conceptual integrity in turn dictates that the design must proceed from one mind or from a very small number of agreeing minds. This is accomplished in larger projects by separating architectural effort from implementation. Architecture involves developing complete and detail specifications for the implementers. The aristocracy vested in the architects is necessary in order to achieve conceptual integrity. One concern by implementers is that by allowing architects to do all the specs, the system may end up being too rich in function and thus too costly (this problem is addressed in the next chapter). The benefit is that widespread horizontal division of labor has been replaced by vertical division of labor with simplified communication and improved conceptual integrity.

Chapter 5—The Second System Effect

Architect and builder need to communicate thoroughly. When the architect is confronted with a cost estimate that is too high he needs to work with the builders to bring estimates or design more in line with each other. It is important for the architect to suggest, not dictate ways to implement product, and to be prepared to accept suggestions from the builder that may meet the objectives as well. After the first system is finished, the most dangerous system a person ever designs is the second system because of the tendency to over design the second system. There is also a tendency to refine techniques that may have become obsolete by changes in basic system assumptions. The second system effect can be avoided by being extra careful and disciplined to avoid functional ornamentation and extrapolation that are obviated by changes in assumptions and purposes. If the option is available, hire an architect that has designed at least 2 systems.

Chapter 6—Passing the Word

The manual is the external specification of the product, which describes and prescribes every detail. It is the chief product of the architect. It must describe everything the user sees, but it must refrain from describing what the user doesn't see. These descriptions must be complete and accurate. Formal definitions are precise, but are difficult to understand. Formal definitions need to apply to the externals of the system and these must be carefully delineated. Sometimes implementations are used as definitions (PC compatible, Windows based, etc.). Such definitions are precise but may over-prescribe even the externals. Using implementation as definition may lead to confusion if formal description is the standard. Another technique for disseminating and enforcing definition is to design the declaration of passed parameters or shared storage and to require implementers to include that declaration with %include-like commands. Meetings are also necessary. Two types of meetings are useful: a weekly half-day meeting of all architects and official representatives of the HW and SW implementers and market planners in which the chief architect presides. Proposals are distributed in writing before the meeting. Changes are adopted by consensus or by decision from the chief programmer. These meetings are fruitful because of the information exchange, quick resolution of problems and timely decisions. Backlogs of minor appeals and other issues that pile up are settled at annual supreme court session lasting about 2 weeks, held just before major freeze dates for manuals. This is also a good forum to exchange ideas and getting them accepted.

Chapter 7—Why did the Tower of Babel Fail?

Most projects that fail had communication or coordination problems. Teams can communicate informally, through meetings or via a workbook. The project notebook needs to be started from the beginning. It contains the structure imposed on the documents that the project will produce, including objectives, specifications, standards and administrative memos. It also helps control the distribution of information, not so much to restrict, but to ensure everyone receives the information they need to have. The need for structured workbook increases with project and team size. Workbooks need to be updated frequently with relevant changes highlighted for quick assimilation. With electronic storage, work books should have FIFO change logs and flags. While Parnas recommends hiding interface design details from those who don't need the details, Brooks warn that unless interface is perfect this may lead to problems. He prefers open exposure to interface errors to stimulate their correction. In large projects, the need to communicate to coordinate is reduced through division of labor and specialization of function. Each sub-tree in the structure must have: a mission, a producer, an architect, a schedule, division of labor and interface definitions among the parts. The producer assembles the team, divides the work and establishes the schedule. The architect provides conceptual integrity to the part being designed. The producer and architect can be one person in small projects. In large projects two separate people are needed and one of them must be the boss. The producer is in a best position to be the boss by releasing the architect from having to communicate with the structure and thus concentrate on design issues.

Chapter 8—Calling the Shot

Coding represents only 1/6 of the estimation effort. Also, data for small programs are not applicable to more complex programming systems. Planning, documentation, testing, system integration and training need to be added to the estimate. Such figures cannot be extrapolated linearly, unless there is no communication. Effort is a function of some power of program size. Portman's data show that the time beyond the linear relation is spent in non-development activities. Aron's data show that productivity rates fall as the number of expected interactions increase. Harr's data also show declining productivity rates with increasing program size. Aron, Harr and OS/360 data all confirm striking differences in productivity based on task complexity, difficulty and size. Corbato's data on the other hand suggests that productivity may be relatively stable in terms of instruction lines (not words) per person-year (line is a unit of though), suggesting that high-level languages may have an impact on programmer's productivity.

Chapter 9—Ten Pounds in a Five Pound Sack

Because size is a large part of the cost of programming system products, the builder must set size targets, control size and devise size-reduction techniques. Size itself is not bad, unnecessary size is. For the project manager, size control is partly a technical job and partly a managerial job. Systems need to be subdivided and each component given a size target. Setting targets for the core is not enough. One needs to budget all aspects of size. Modules need to be precisely defined before size targets can be established. Care must be exercised so that in an effort to optimize individual pieces the whole system may be negatively affected. In trading function for size a designer must decide how fine-grained the user choice of options will be. Space and speed also trade-off. Optimized subroutines need to be developed and made available for queuing, searching, hashing and sorting. Two versions of each of these programs should be in the notebook, a fast one and a squeezed one.

Chapter 10—The Documentary Hypothesis

Documents for a software project: objectives and specifications (what), schedule (when), budget (how much), organization chart, and space allocations (where). Conway's law: "Organizations that design systems are constrained to produce systems that are copies of the communication structures of these organizations". Writing is important because it uncovers gaps. It also communicates the decisions of others and provides a database and checklist of important items. A handful of critical documents is vital.

Chapter 11—Plan to Throw one Away

In most projects built the first one is almost unusable. One has to build a system to throw away. In fact, you will throw away the first system. The question is whether you plan to throw it away or give your customers a throwaway product. As one plans a throwaway pilot one must accept the fact that things will change. So, systems must be designed for throwaway pilots and for change. Techniques to do this include modularization, subroutine libraries, precise and complete definitions of module interfaces, and complete documentation of these. Quantization of change, version planning and version scheduling are also essential. Reluctance to document often comes from the recognition that things will change. Structuring an organization for change is harder than designing a system for change, but management structures may need to be changed as the system changes. The surgical team approach minimizes the effect of change since it is designed to minimize the number of interfaces. A program doesn't stop changing when it is delivered to the client. It will need to be maintained and updated periodically. It costs about 40% more to maintain a program than to develop it. New errors are discovered as new users start working with the system. But a software fix has a 20%-50% probability of introducing another error somewhere else (2 steps forward, one back). This is why regression testing is important (but costly). Lehman and Belady have found a linear relationship between new version releases and the number of new modules added with potential for more errors and more difficulties for error fixing. They conclude that "things are always better at the beginning".

Chapter 12—Sharp Tools

Tools include a computer facility, a language, utilities, debugging aids, test-case generators and text processing systems. One needs a target machine and a vehicle machine. Target machines need to be scheduled. If the target machine is new one needs a logical simulator for it. Programmers have their respective "playpen" areas where they have no restrictions on what they can do with their programs. When a program is ready for integration it is moved into a system integration library. Once there, even the original programmer cannot change it, except with permission of the integration manager. When a system version is ready for wider use it is then moved to the current version sub-library. This copy is sacred. The idea is to have control and authority over software versions by managers who can authorize change. It also allows for formal separation and progression from playpen to release. Next, it is necessary to have tools, a documentation system and a performance simulator. High level languages are also tools that improve productivity and debugging speed. Use of high languages comes at the cost of less flexibility and less speed, but it trades off with increased functionality and development productivity.

Chapter 13—The Whole and the Parts

The most pernicious and subtle bugs are the ones arising from mismatched assumptions made by authors of various components. Specifications are first handed to an outside testing group to be scrutinized for completeness and clarity. In top-down design (formalized by Wirth) the procedure is to identify design as a sequence of refinement steps. Each refinement becomes a more detailed algorithm. From this process one identifies modules whose further refinement can proceed independent of other work. Good top-down design avoids bugs because of clarity of structure and representation, partitioning and independence of modules, suppression of detail in structure makes flaws more apparent and design can be tested at each step of refinement. Top-down is a very important programming formalization. Another important formalization is structured programming. Component debugging relies on on-machine debugging, memory dumps, memory snapshots and interactive debugging. For each hour spent debugging, 1/2 hour should be spent planning the debugging session and another 1/2 recording results. System debugging will always take longer than expected thus the importance of proper planning. One common approach is to work with debugged components, or at least components whose founds have not been fixed yet, but have been found and documented. Another approach is to build "scaffolding" components (dummy components) that consist of interfaces and perhaps some fake data or small test cases. Miniature files, dummy files and auxiliary programs are examples of scaffolding components. Component changes must be controlled with copies stored away. Components should be added one at a time. Finally, updates must be numbered in versions. Adding new versions of components should go through the same testing as new components.

Chapter 14—Hatching a Catastrophe

Termites do more destruction than tornadoes. Project milestones need to be concrete, specific, measurable events and well defined. Studies have shown that project underestimates don't surface until about 3 weeks before scheduled completion. Good milestones help overcome this problem. Critical path charts help monitor these milestones. Managers need two types of schedule information: exceptions to plan that require action and status picture for education. Line managers may not have an incentive to share this information with the supervisor. Such a conflict can be reduced with open communications or clear scheduling tools that don't lie. Then, only 2 questions need to be asked by the supervisor to line managers: whether milestones have been set or changed, and whether milestones have been met.

Chapter 15—The Other Face

Different levels of documentation are needed for the casual user, the user who depends on the program, and for the user who must adapt a program. To use a program: purpose, environment, domain and range, functions and algorithms used, input-output formats, operating instructions, options, running time, and accuracy and checking. To believe a program, test cases are necessary: mainline cases for commonly encountered data, barely legitimate data that probe the edges of input data domain and all kinds of valid exceptions, and barely illegitimate cases to ensure that invalid inputs raise proper diagnostic messages. To modify a program much more information is necessary. A sharp and clear overview of the internal structure is necessary: flow charts, algorithm descriptions, file layout explanations, overview of pass structure, and a discussion of modifications contemplated. Flow charts are the most oversold piece of program documentation. It breaks down when chart exceeds a page. Detailed blow-by-blow flow charts are a nuisance suitable only to initiate beginners into algorithmic thinking. Most flowcharts are prepared after the fact, often by automated tools, and not a priori to guide code development. Self-documenting programs are very valuable since source program and documentation are in one place. One place to document is when variables are declared by attaching explanatory labels. Heavy commenting and use of format and blank spaces are other useful suggestions. Prose descriptions can be added to procedure calls. If standard algorithms are used it is better to make reference to the original source than to explain it in the code. Declare all variables and mark initialization with labels. Label statements in groups and use indentations to visually spot these groups and to show structure. Flow arrows can be marked at the right edge. Use line comments. Put multiple statements on one line or one statement in multiple lines to match thought-grouping and algorithm description.

Chapter 16—No Silver Bullet—The Essence and Accident in Software Engineering

To address the essential parts of the software task concerned with structures of great complexity, it is suggested: to avoid constructing what can be bought, use rapid prototyping, grow software organically, and identify and develop conceptual designers. There is no single bullet in the horizon in terms of technology or management technique, which by itself promises even one order of magnitude improvement in productivity, reliability and simplicity. There is an anomaly in that software progress is very slow, but hardware process is very fast. There are two types of difficulties: essence and accidents. Essence difficulties involve the correct specification, design and testing the conceptual construct, and not so much the labor of representing it. Essence difficulties include complexity, conformity, changeability and invisibility. Complexity stems from the fact that no two parts are alike. If they are, we make them subroutines. Scaling up is not merely a repetition of the same elements in larger size but an increase in the number of different elements that interact in non-linear fashion. From this complexity comes the difficulty of communication among team members that lead to product flaws, costs overruns and schedule delays. Conformity has to do with the fact that software must "conform" to other interfaces. Changeability makes things difficult because the software entity is constantly under pressure for change. This is partly due to the fact that software can be changed (it is pure thought stuff). Change comes from two sources: people try it in new cases at the edge and software survives the hardware platforms on which they are run. The software product is embedded in a matrix of applications, users, laws and machines, which change continually. Software invisibility is another source of difficulty. There are no geometric abstractions to help visualize software products. As soon as we start diagramming we realize we need more than one diagram. Software technologies have solved mostly accidental difficulties not essential ones. High-level languages embody constructs wanted in the abstract program and avoid lower ones, thus eliminating a whole level of complexity. Language development approaches are approaching user sophistication. Time-sharing preserves immediacy and enables us to maintain an overview of complexity by reducing turnaround time. Unified programming environments help overcome difficulties by providing integrated libraries, unified file formats, pipes and filters. Tool benches further complement this.

Hopes for the silver. (1) Ada and other high level languages with sophisticated features, modular philosophy, abstract data types and hierarchical structuring. (2) OO programming with abstract data types, hierarchical types (classes) and hidden operations. This allows removal of higher order difficulty by allowing high order expression of design. (3) Artificial intelligence—2 definitions: AI-1 is the use of computers to solve problems that previously could only be solved applying human intelligence. AI-2 is the use of a specific set of programming techniques known as heuristics or rule based programming, which has been applied to expert systems. (4) Expert systems is a program containing a generalized inference engine and a rule base constructed form human expert knowledge. The power of these programs comes from the knowledge extracted from experts, but many difficulties remain both to apply the proper rule to a case and to find articulate experts from which knowledge can be extracted (and finding ways to extract that knowledge). (5) Automatic programming using code generators. (6) Graphical programming is somewhat more difficult because it is hard to visualize software. (7) Software verification does not seem to have the magic bullet either. (8) Environments and tools offer some promise with the use of integrated databases, but returns so far have been marginal. (9) Workstations will improve productivity but no magical enhancements. If the conceptual components of the task are the ones taking most of the time, then no amount of improvement on task components that are just expressions of these concepts can give large productivity gains. One possibility is to buy components and not construct at all. This is becoming more feasible and we should expect to see market places for individual modules. This is cheaper and has better documentation, but its applicability to the application is a challenge. Sophisticated applications can now be modeled with simple spreadsheet and similar tools. Rapid prototyping tools may also offer substantial productivity improvements because clients generally are not entirely sure of what they want or need. Also, growing software incrementally rather than building it may also improve productivity. Start with a simple running program and keep adding to it. Great designers (and managers) are the final source of big productivity improvements. The differences between the great and the average are an order of magnitude apart.

Chapter 17—No Silver Bullet Refired—9 years later

NSB argues that much software progress has been the removal of negative factors by accidental (not by accident, but incidental) construction of artifacts and that real productivity improvements will come from getting at the essential aspects of software productivity improvement. Essential difficulties are inherent in the conceptual complexity of software functions to be designed by any method. This complexity exists by levels and it is often related to the complexities of the applications being built. But much of the complexity in a software construct is due to the implementation itself. NSB advocates that progress can be made to attack this complexity by adding the complexity: (1) hierarchically by layered modules or objects and (2) incrementally, so that the system always work. NSB is based on the believe that sources of productivity improvement need to be sharply divided into essence and accident and that this division guides what kinds of attacks to make. Consistent with Harel, the invisibility argument related to the absence of geometrical abstraction means that several diagrams will be needed and that some aspects of software development don't diagram well. Jones has argued that by focusing on quality that productivity will follow and that most of the unproductive activities have to do with defect removal and repairs. Bohem, however suggest that productivity drops again when one pursues extreme quality. One prediction that seems to be coming true is that the development of the mass market is the most profound trend in software engineering. Another promising technology is OO programming in which modularity, encapsulation, inheritance, hierarchical structure of classes and strong abstract data-typing are all provided in one multi-vitamin pill, which is a promising concept. OO has grown slowly for a variety of reasons, from changing paradigms to the use of a variety of complex languages. It seems like developers have taken OO as a tool not as a development paradigm. Brooks believes that its adoption has been slow because of substantial up-front costs, mainly in re-training programmers, but the benefits pay-off during successor building. The promise of easy reuse of classes with easy customization via inheritance is one of the strongest attractions of OO techniques. Barriers to reuse are not on the producer side but on the consumer side. If the perceived cost of finding the right object is higher than building one most developers choose to build. Reuse is very popular in mathematics where the cost to reconstruct a mathematical software component is high but to reuse it is inexpensive. Real reuse is just beginning. One barrier has to do with how to access thousands of components each with 10 or 20 parameters and option variables.

Chapter 18—Propositions of the Mythical Man-Month: True or False?

Summary chapter--see chapter bullets.

Chapter 19—The Mythical Man-Month After 20 Years

What was right when original book was written. *(1) Conceptual Integrity and the Architect (central argument).* Conceptual Integrity—the inherent difficulty of developing large software has to do with needing many minds for the design. The Architect—someone responsible for the conceptual integrity of the project. Separation of Architecture from Implementation—there is a clear boundary between the two. Recursion of Architects—need to partition the system into subsystems with an architect assigned to each. *(2) The Second-System Effect: Featuritis and Frequency Guessing.* Designing for Large User Sets—it is more difficult to build a program for generalized use. Featuritis—tendency to overload the product with too many features of marginal utility to most, but it helps mass market the product. Defining the User Set—the larger and more amorphous the user set the more necessary it is to define it explicitly. Frequencies—the architect needs to estimate attributes of the user set (better to be explicit and wrong than vague). *(3) The Triumph of the WIMP (windows, icons, menus, pointing) Interface. Conceptual Integrity via a Metaphor*—adoption of a familiar desk metaphor. User Power vs. Ease of Use—need to provide both. Incremental Transition from Novice to Power User—smooth transition via short cuts. Device for Enforcing Architecture—uniformity and cross-application conceptual integrity (Mac). *(4) Build One to Throw Away.* This is OK if one assumes the waterfall model. But waterfall concept is wrong because it assumes that the system is built at once. It needs an upstream flow. *(5) An Incremental-Build Model is Better.* Build an End-to-End Skeleton—start with a functional dull system and add modules by having a running system at every pass. Parnas Families—design software as a family of products with features least likely to change at the root of the tree. *(6) Information Hiding is Better.* As suggested by Parnas, programmers are more efficient if shielded from details of modules they don't own. This concept was upgraded into an abstract data type from which many objects could be derived. In addition, the powerful concept of inheritance is also a great contribution. Finally, classes designed and tested for reuse are useful.

Barry Bohem's work—confirms the lack of linear relation between persons and months. He finds a cost-optimum schedule $T=2.5(MM)^{1/3}$. *Brooks Law*—some evidence may suggest that adding people to a late project makes it more costly and not necessarily more expensive. It all depends on the kind of people added. But the warning remains. Boehm's COCOMO finds that the quality of the team is by far the largest factor in its success, much more than the tools. DeMarco and Lister make a similar argument for *Peopleware: Productive Projects and Teams* and team fusion (moving projects). *The Power of Giving up Power*—Schumacher's Principle of Subsidiary Function (Small is Beautiful) that the center will gain in authority and effectiveness if the freedom and responsibility of the lower formations are preserved. Evidence with many small startup firms are confirming this. *The Biggest Surprise*—millions of computers in people's hands. It has brought fluidity to many fields, which can bring order of magnitudes in qualitative improvements. The microcomputer revolution has changed how everyone builds software. Many of the related accidental difficulties have been eliminated. PC's provide the computing engine and the network provides the shared access to files. Client/server make shared access even simpler. *Shrink-Wrapped Software*—from 4GL tools to multi-platform O/S's. The economics of this sector have changed. Schedule and function dominate development cost nowadays. A promising trend is the components shrink-wrapped market and meta-programming. 4 levels of users of shrink-wrapped software: as-is user who uses product as is; meta-programmer who builds templates; external function writer; and meta-programmer who uses several components in a larger system (uses MPI—meta-programming interface).