This paper will analyze chapter six of Frederick P. Brooks' *Mythical Man-Month* while relating the context of this literary work to modern Software Qualities and Principles. The chapter's name, "Passing the Word", refers to the transfer of information between all parties involved in the development and usage of a piece of software (and at times hardware). However, the chapter's main focus is on techniques of communication and production process within the development team. The chapter covers many fine points to consider for team managers, system architects, and system implementors. The goal of these fine points is to minimize confusion between architects and implementors as well as aiding in the delivery of a complete, well documented program. The chapter clearly shows that software documentation and implementation go hand in hand and must both be streamlined if a piece of software is to be functional and comprehensive at the same time.

The chapter opens by identifying the project manual as an important factor in the development process. It is stated that the manual must include all of the things that the user is aware of, but none of the things that he is not aware of. The purpose of this distinction is to safeguard the implementor's design freedom. Also, a manual must be written by one or two people. This ensures consistency among all of the little mini-decisions present in manual writing. These decisions are not big enough for discussion but must be made consistently.

Brooks holds that the manual writer should provide the reader with two definitions of the solution to a given feature. There must be a formal solution focusing on accuracy and precision. However, there must also be a prose definition for comprehensibility. This prose definition is written in spoken language and is more suited at explaining the "why" of a solution. Brooks warns that out of these two definitions, only one must be held as the standard. The other must be considered the derived form of the standard. It does not matter which definition is the standard. The main point of the argument is that there should only be one standard definition.

Brooks states that an existing implementation of a solution can be used as an architectural definition but warns of the large pitfalls associated with this method. Many times there is confusion as to the validity of inputs given to the existing implementation. Test cases that include improper input and artifacts within the output are often recorded as part of the definition. This forces implementors to replicate these original mistakes. Consequently, many times the only way to replicate the mistakes is to use outdated algorithms.

The chapter also suggests that several implementations of the same solution should be constructed in parallel. This method can lessen the temptation to change the manual once a discrepancy between it and the product is found. Instead, the problematic implementations can be adapted to the functional ones without change to what the user expects within the finished product.

Logs of communication between architects and implementors are important. If a question is asked by an implementor and presented an answer by an architect, this answer must be published. By keeping a log of these answers and publishing them to all implementors, overall consistency is increased and question repeats are avoided. Also, publishing these
answers opens them up for critique and makes gaps in reason apparent and susceptible to appeal.

Brooks suggests that product testing should also be done by a third party. The developer's objective is to deliver a functional product, which may conflict with a need to find ways to break it. A third party testing agency's specialty is breaking a product, which makes them more ideal for the field. Such a testing agency is necessary because often times developers will not be as merciless in their testing as the users of whatever is being developed. The testing agency serves as a buffer between development testing and user testing.

Team organization is a huge theme throughout this chapter. Many of the fine points discussed within this chapter focus strongly on minimizing the amount of communication overhead between software architects, and implementors. At the same time emphasis is placed on increasing the quality of the communication that does take place and making sure most of it reduces confusion rather than increasing it. To accomplish this the manual is identified as one of the primary design documents. Being of such importance, it must contain both full, accurate, and precise information while not overwhelming readers' ability to comprehend all that is being documented. This serves to organize the team's goals into a complete and comprehensible form.

The suggestion for well documented question and answer logs also serves to tighten team organization. These logs greatly reduce communication overhead because it eliminates duplicate questions. The logs also serve to freeze an answer to make sure no two implementors get a different answer for the same question. Thirdly, the logs also serve to provide answers in plain view where they may be corrected or appealed if a mistake or misunderstanding is present.

Another resounding theme within chapter six is the subject of implementation strategies. The chapter encourages that several implementations be produced concurrently. As previously stated this saves the project if discrepancies are found between the manual and the implementation without requiring that the manual is changed. Brooks also points out that if a solution to a problem exists but that problem's implementation is not given, the test cases and results can still be used as a design document. Brooks also warns of the faults with this implementation strategy so that an unbiased opinion can be formed.

Although project scheduling is not a huge theme within this chapter, it is still evident that scheduling is greatly valued by Brooks. Almost all of the chapter's tips inevitably lead to an increase in production speed as a result of a streamlined process.

The modern software quality emphasized most within this chapter is Productivity. Almost all the main points of this chapter focus on improving the production process. The various tips on how many people should write a manual and what a manual should contain and represent help greatly reduce team confusion. A team whose members are very sure of what exactly they are building will work faster and create a higher quality product. Some methods for reusing existing solutions are also talked about in the chapter. These recycling methods would help speed up the process and most likely greatly reduce production cost. The chapter also recommends third party testers. This promotes specialization and makes the development process more efficient by eliminating possible conflicts of interest.

Another important modern software quality emphasized in chapter six is Visibility. Brooks pays special attention to documentation, both of team efforts and of incremental implementation towards a finished product. Brooks states that the manual should also contain a prose description of product functionality, explaining why the implementation works. Also,
Brooks' suggestion that an architect have a question and answer phone log creates great documentation of what decisions are being made in the implementation process.

Brooks also puts emphasis on the principle of Separation of Concerns. He points out that an architect's breakdown of the problem should not contain any implementation. This is done so that the implementor can have the design freedom to solve the specific piece of the puzzle that is allotted to him or her. It is suggested manuals contain formal and prose descriptions of the solution but no implementation details for the same reason. Brooks also recommends that developers use a third party testing entity. Their specialization in the matter of breaking software makes them much more efficient at it and allows the developers to develop without the chore of testing.

Chapter 8

M. Coolbeth

In, The Mythical Man-Month, by Frederick Brooks, Chapter Eight, “Calling the Shot,” examines a variety of data sets with the goal of deriving techniques for estimating the effort that the production of a software system will require. Data sets relate to different aspects of programmer productivity, such as the relationship between the number of instructions and the number of man-months required, or the relationship between productivity and the size of the software team. Such information allows software engineers to estimate the time required to produce a system with greater accuracy and precision, as well as to make decisions about things such as the optimal size for their software team more aptly.

The chapter begins by asking questions about how the effort and time required to produce a software product can be estimated. It introduces a study in which production effort for programs was compared to program sizes. The study suggested that the relationship between the two is not linear, but that the effort required to produce a program is proportional to the program size raised to the 1.5th power. This is the first of a number of studies introduced throughout the chapter.

The second study introduced is that conducted by Charles Portman, software manager for ICL. Portman instructed a number of his programming teams to keep detailed logs of their daily time usage after he noticed that they were consistently completing jobs in approximately twice the originally estimated times. The logs they created revealed that only about half of any workweek for a team was accounted for by the projects whose completion time was being measured. This suggested that inaccurate estimates were being made as the result of failures to account for time spent on tasks not related to the project under consideration, and that, had the teams been able to dedicate every man hour to the project, their production periods would not have overrun the initial estimates.

Two more data sets discussed by Brooks are Joel Aron's data and John Harr's data. The studies producing these data both measured the productivity of software teams against the organizational complexity of the systems they produced. Aron, a manager at IBM, conducted a study of large system production, in which he categorized systems according to interactions between programmers and interaction between system components. His data suggest that teams creating systems that require fewer interactions are able to be much more productive.

Harr, another programming manager, made detailed notes of his teams' software projects. His records for four projects are provided by Brooks, and the data thereing are very supportive of
Aron’s observations. The record for each project involves its type, the complexity of the system, the number of team members the completion time and effort, the program size, and the team’s productivity. Brooks notes that, while Harr’s study and Aron’s seem to be in agreement, there are doubts about the relationship between complexity and productivity in Harr’s study. It is unclear, says Brooks, whether the more complex products required more modules and greater effort because they had larger teams, or whether they required larger teams because they were more complex.

They final data mentioned by in chapter eight come from Corbato. These data resemble those in the previously mentioned studies an most ways, but the previous data pertain to assembly programming, while Corbato’s data were produce by an analyses of projects which used higher level programming languages. Since the numbers of statements in the different studies reflect one another nicely, and since, a statement of in a higher level language represents a number of words of assembly code, Brooks concludes that a software developer can become several times as productive through the use of a higher level language.

The studies described in this chapter are informative about scheduling practices and team organization. The scheduling implications of Portman’s data are clear. A software team that is unable to dedicate all of its man-hours to whatever product it is developing should not be subject to a schedule that assumes it can. The schedule for a project should be constructed carefully, and allow for unrelated events to occupy the time of the production team.

A number of the productivity studies make suggestions about effective organizational practices. In Harr’s study, the programs produced least efficiently were those with the largest numbers of programmers. While this information is not conclusive, it suggests that creating a larger than optimal software team could be hazardous to the team’s productivity. Aron’s data shows that an abundance of interactions between programmers impedes productivity. This suggests that, in order to optimize a team’s level of efficiency, one should organize it in a way that minimizes the amount of communication between its members.

This chapter pertains exclusively to the software qualities of productivity and timeliness. Both of these qualities are internal qualities, and as such, are characteristics of a software product’s development process. A product that is timely is completed before its deadline, or by the time it is scheduled for use. A software productivity reflects the efficiency with which a program is produced.

Most of the studies discussed are productivity studies. All of the productivity studies suggest that smaller software projects exhibit greater productivity. There are also data presented in the chapter which suggest that projects with smaller teams and involving less interaction will also be produced more efficiently (require fewer man months per unit of code).

The link between productivity as discussed above and timeliness is quite apparent. However, the part of the chapter dealing specifically with timeliness does not discuss this kind of productivity. Instead it discusses the estimation of completion times, and demonstrates that the placement of deadlines needs to account for more time than needs to be spent on a given project, or the production will likely not be completed on time.

The software principle most important to the chapter is modularity. Modularity presents itself as a solution to many of the problems indicated by the studies that Brooks discusses. The studies demonstrated that software teams lose productivity as the amount of communication required
between team members increases, as well as when there is more interaction required between software components. Teams also seem to lose efficiency as their size grows.

If a software product is modularized, then the problems created by these sorts of circumstances are minimized. A team working on a product that is properly divided into modules can remain productive even if it is a large team working on a large project, since each module constitutes a simpler subproduct and is developed by a smaller, more efficient subteam. In addition to reducing the complexity with which any programmer must deal, modularity minimizes the amount of necessary communication between programmers working on separate modules.

Chapter 8

J. Franco

The core focus of Chapter 8 of the Mythical Man month is the estimation process and its relation to the planning of software projects. This chapter is one of the most critical aspects of the book’s message. This chapter demonstrates most concretely the lack of linear relation between man-months and actual completion time. The very concept of the man-month itself is likely to have evolved out of this estimation process. The chapter is a brief overview of the estimation process and how the complexity of a program can affect its complexity and consumption of time in non-linear ways.

The key concept in this chapter is that of the estimation process for software projects. The most critical point is the non-linear relation between effort as it relates to program size. Although the estimates appear to be for assembly language, the text appears to imply that the estimates apply similarly to higher-level languages. Though due to the higher-level constructs of more advanced languages, the use of high-level programming languages appear to increase the speed of development from a time-per-instruction point of view. Although this is one of the most important concepts in the entire book, it is also likely to be one of the most dated as little to no programming is done at the assembly level. Furthermore, the correlation between LINES per unit of time in a high level language with a specialized runtime such as Java or C# can not even easily be related to INSTRUCTIONS per unit of time. It is puzzling however to note the observation that effort relates exponentially to size even in cases where there is no communication involved. This suggests that the problem of software design arises from more than simple communication or scheduling issues. Incidentally, the author appears to assume a direct correlation between instruction count and overall (qualitative) complexity. If the exponential relationship between instructions and time is any indication, it is more probable that the increase in the number of instructions or lines causes the number of complex programmatic interactions to increase at such a rate as to cause the exponential increase in required effort.

Simply put, the parts of larger programs interact in more complex and intricate ways. This complexity, not programmer communication and scheduling issues, is the root cause of the exponentially increasing work required. It is unfortunate however that although the general concepts in the chapter still hold true today, the specifics are sadly out of date. It is no longer the business of most software companies to be, for example, writing compilers and control systems in assembler language. Programming has advanced in the last 31 years to a much more abstract and conceptual level. Programmers of 1975 could in no way predict things as the Common Language Runtime (of .NET) and its massive set of objects with which to build an application. Such advances in Rapid Application Development languages such as C# must clearly have a serious impact on development time. Sensibly, the estimation processes must be updated to suit
Quite obviously, this chapter falls primarily within the realm of project estimation. It is important to note that although the chapter references outdated methods and practices (procedural paradigm using assembler), the overall message still applies today. Computer programming is a very abstract and idea-oriented process that involves complex factors that cannot be easily measured. It is interesting to note that one technique that the author fails to mention is the possibility of simply using the production time of a previous similar project in order to estimate the development time of a new project. I suspect this omission is due to the fact that in 1975, such information was simply not available, as there were not thousands upon thousands of previous products which one can look upon to make an estimation.

A secondary and related conclusion that is discussed at the conclusion of the chapter is the observations of software efficiency as it relates to higher-level languages. The estimate is that this trend can be followed further with even higher-level languages than those present at the time of publication. New languages such as C# are further saving time by allowing users to write significantly less code than in C++ or even Java. It is probable that as programming languages become more and more advanced and higher levels of abstraction are used, the efficiency trend will continue (that is, development time will drop because single statements will be representing more and more complex constructs).

Two of the major aspects of chapter 8 and, in fact, the entire book are Productivity and Timeliness. No other software quality aspects are more central to the information at hand. Clearly, one cannot deliver a product on time in cases where one cannot make realistic assumptions as to when “on time” is. The time estimation process is a critical aspect of making realistic time-related projections for a customer. Furthermore, without proper prediction techniques, it is difficult to determine the cause of any delays that do manage to arise. An accurate gauge of progress is also needed to properly analyze productivity. Delays that may be attributed to low productivity may actually be caused by poor planning. There are even indications within the book that poor planning may CAUSE low productivity, thus worsening any problems.

Finally, anticipation of change must be regarded as a critical aspect of the planning and estimation process. A static and rigid plan is fairly unrealistic as it is impossible to truly make a time-based prediction without anticipating a change in customer needs. It is critically important because most customers are often incapable of properly expressing their needs on the first attempt. In fact, many do not even know what it is that they actually need. To properly “call the shot” so to speak, a plan requires a realistic assumption that requirements will be subject to change.

**Chapter 9**

**Jim Gedarovich**

Chapter 9’s title is “Ten Pounds in a Five-Pound Stack” which is prefaced with a saying that an author should be like Noah and pack a lot into a small compass. These two contradictory statements are what the chapter is all about: The author writes about how cramming as much functionality as you can into a program is essential in making a good program. However there are constraints to hold so that the size of this incredibly functional program doesn’t run slow and take up way to much disk space. More complex the programs or systems have more tight bounds
on its size. The author goes on to explain the cost of space in a software programming project, what a manager of that project can do to ensure that the software produced is under complete size control. Thirdly the author goes into Techniques of how exactly the size can be kept down and in what manner, in the sense of building the program and getting programmers on the project up to speed with new systems.

One and a Half page summary:
The first segment of this chapter is “Program Space as Cost” starts with an anecdote of how much a program can cost in respect to how much memory it takes up, following a discussion of how memory is an overhead for a software project, and the amount of overhead one can't complain about if the project provides the functionality it claims to encompass, and performs smoothly. The anecdote he used however, I was not sure weather or not he is really talking about companies having to pay to rent memory or not because 1975, the date of this books original publishing, was ten years before I was born. However I am willing to guess that that is true.

The next part of the chapter is “Size Control” where the discussion moves to whose job is it to enforce bounds on size and how this should be done. The project manager has to make sure that everyone working on the project has studied the target user(s) and the application(s) they use so that a definite size of the different components of a software piece can be set. In the case of large scale systems like an operating system, they would have to study what people do with their GUI’s their word processors, etc, and budget how much size for each, as well as hold the responsibilities of budgeting each application that is going to be included at a lower level. So if we are talking about it in general everything that relates to size has to be enforced a budget. In three easy steps he lists setting total size budget and individual component budgets, Defining what each module must do while determining its size (the word processor for this Operating System should have X functionality and take up at most Y space), and finally each member of the project he explains needs to keep a total system awareness, user oriented attitude are the big three essential steps to avoid excessive space using a software project. This is so that, he explains, After that he has another chapter “Space Techniques” that describes how space is a trade off of that needs to be balanced, and how. For one, he states that small successful programs are made with invention and craftsmanship. And that more function means more space. So an essential strategy for the software project manager to keep the program within its budgeted space is to have many optional features. Finally he goes on to explain how the way data is stored is the most important part of programming in the last segment “Representation Is the Essence of Programming” and that the biggest breakthroughs in programming are always not just stumbled upon but precisely constructed through cleverness. For instance storing your data in the correct data structure or substituting a certain sort inside a algorithm were two examples that he gave.

One Page Discussion:
The two most important topics of this chapter are project estimation and implementation strategies. This chapter describes the importance of budgeting or “estimating” on the part of the manager and his team to make sure that each segment of his system weather it be on an overall program level or from a closer standpoint on the modules of a program. Essentially if a program is two memory hogging then it will crash older system and not be able to run on only the best computers which is bad for sale. As well back in the day at least, memory was rented commercially (if that is serious at the beginning of the chapter) so a program that would take up more memory would cost asymptotically more. Even though this is still not true it would be a
good way of looking at the situation nowadays so that the size of memory is never exceeding a certain level. These different levels must me estimated by studying what is actually being done and who your user is and what he sort of application(s) with what sort of functionality of the application(s). Secondly in this chapter implementation strategies were described as incredibly important. The author gave a list of morals to hold when designing and implementing a project for setting the size to stick to and the point of views (strategy) the members of the team must have (user oriented total system). As well many Techniques were described for design and implementation to keep the size reasonable one such was making certain functionality optional so that the user can choose to customize his product, as well saving space. Also brought to light is the cost of performance and space where a given function will perform faster given more space. The author stresses the well educated programmers are key to implementation of a lean program because they will be able to make the breakthroughs that only come from knowledge and innovations. Such as which sort to use in an algorithm or what data structure to place data.

One page Discussion with respect to software Qualities and one software principle:

Out of all the software qualities there are two that are incredibly dear to this chapter and they are Interopitability and Productivity. Productivity is the most important to this chapter because that quality encompasses efficient performance, and after all nobody cares how large a program is PER SAY if its performance is excellent. The efficiency of a program greatly depends on the size. Keeping the size down generally means that you are not only crunching the space but the efficiency of a program. A good software project will have a mix of crunched programs that take little space but don’t run as efficiently as well as larger ones that take up more space but are more efficient. Productivity is a tradeoff as this chapter describes, that must be balanced using certain techniques. The second most important is Interoperability or the ability of the software project to be run on a multitude of systems. For if the size is to large it will run excellent on machines with the size to space, but wont run at all or will run poorly on machines lacking the size because it will require paging files to store segments. If the size and functionality is balanced correctly as discussed before then this chapter suggests that optimized will be the Productivity and Interoperability. In addition to those two Software Qualities being important there is a very important Software Principle that goes along with this chapter quite well. That is Generality; because it is the most corresponding to keeping size reasonable. For if you can generalize the functionality of 4 or 5 programs in an operating system software project then they should share the same functionality then only one thing needs to be stored and can just be made to be specialized in the aspect of how the module will be used in those different programs.
of memory and permanent storage space consumed by a computer program. In most situations there will be a space-time tradeoff: The program can be made to run in less memory by executing more instructions or vice versa. There are also ways to reduce the space consumed without any corresponding increase in other undesirable characteristics: This can be found in the choice of a good form of data representation and good algorithms. By sticking to these guidelines managers and programmers can ensure that their software is as large as necessary, but no larger.

Summary

It is important to note that the value of much of the contents of this chapter is of primarily historical interest. Memory costs, then over a thousand dollars per megabyte per month, can be purchased today at costs of roughly one hundred dollars per gigabyte. Disk space, likewise, has dropped in price dramatically, to the point that what once cost millions is now present in every common desktop computer. As a result, memory and space constraints are largely secondary considerations in the design and the implementation of the vast majority of today's software.

However, there are certain subsets of the application of modern computer science that do require attention be paid to the amount of system memory consumed by an application and to the size of the binary residing on disk or in other permanent storage.

As one example, when designing and writing programs for embedded systems, size is of the utmost importance. In devices that are sold for only a few dollars, the cost of both volatile and nonvolatile memory is a significant portion of the variable production costs and must be kept to a minimum.

At the other end of the spectrum, in high performance computing it is often the case that an algorithm will increase its running time by a significant amount when its working set size exceeds the size of the processor cache. Even on state-of-the-art processors, the processor cache size rarely exceeds one to two megabytes and the (faster) level one caches are generally measured in the tens of kilobytes. Keeping the working set size below these bounds can result in measurable performance benefits.

Another reason why keeping the size-on-disk of a binary may still be important today is when it is to be combined with a variety of other data (such as the case of a GNU/Linux distribution) and placed on a fixed-size medium such as a CD or DVD, or sent over a low speed or high cost channel such as a dialup modem or any connection billed on a pay-per-use basis. In many of these situations only a fixed amount of data will fit, or any additional data will come at significant cost (either in time or money, if not both) and a smaller binary may make the difference. This is especially true when there are to be many copies made: If you are to upload a file thousands or millions of times to as many users, a small amount per transfer will in the end save a large amount overall. Likewise, if you are to send some data to many different users, needing one fewer CD or DVD per user may save you a significant overall cost even though each disk costs an insignificant amount individually.

Under these circumstances, working to achieve a small footprint for your software can yield useful benefits. This can be done in part by equating the space consumed with the cost of the storage, memory and/or bandwidth that the increased size will consume. Knowing the cost of the resources allows a manager to make informed decisions: What is the cost of paying a programmer for the time necessary to decrease the program's size, and where is the point of
diminishing returns? How much will the resources cost if the program is not made smaller, or what is the opportunity cost of using those resources in this way instead of another if the available resources cannot be reasonably increased? Are there some features that consume more space than they're worth and that could be removed? Performing this type of analysis allows for size control. Knowing what the tradeoffs are, what each option costs in programming time, space and functionality, allows a reasonable size limit to be set. If constraints are very tight, this process should be applied not only to the program as a whole but to each module individually. Once a reasonable size limit is set, the program size can be reduced in a number of ways. One is through the space-time tradeoff: Calculating a value at runtime or recalculating it each time it is needed means that the value is not consuming resources when it is not in use, but it may increase execution time as it must be recalculated unnecessarily. Another is simply through better use of storage: In many cases the form of representation used for the data is unnecessarily sparse; correcting this can yield appreciable savings.

**Discussion (Project Estimation, Implementation Strategies)**

Size control has a significant role in project estimation and implementation strategies in projects that are size sensitive. In project estimation, it must be determined how large the project as a whole, as well as each constituent part, is permitted to be. Good size targets are required to enable informed design decisions. Additionally, reasonable estimates regarding the environment under which the application will run are necessary to determine the extent to which size optimization is necessary and what tradeoffs must be made against other design considerations in order to reconcile size considerations with the other objectives for the application. Since memory consumption is one of the primary determinants of application performance, and the space-time tradeoff is a powerful way of either reducing memory consumption or improving performance, a reasonable estimate for the expected performance range of the targeted hardware is imperative to making appropriate choices. The optimal choice for the tradeoff between space and time depends heavily on the hardware doing the processing, and these estimates will be necessary in the implementation stage.

The most vital part of an implementation strategy with regard to size control is to have a good plan and follow it. The only way to meet an application-wide size target with any degree of consistency is to meet the size target for the majority, if not the entirety, of the subcomponents. A primary concern here is to see not only that each module is meeting its size target, but that it will continue to do so. It may be a serious problem if a module has already consumed eighty percent of its size target and has only yet implemented thirty percent of its functionality, but this may not be clear if the only check being done is that each module is presently meeting its size limit. Unless issues such as this are caught early in the implementation they can become serious problems late in the project schedule. Another serious concern during the implementation stage is feature creep. Nothing will increase the size of a program faster than the incremental addition of unanticipated functionality. Unfortunately, the decision to add features late in the schedule is often not one the project implementers have any sway over, but writing easily extensible code can mitigate at least some of the damage.

**Discussion (Performance, Understandability, Abstraction)**

The primary software quality with regard to memory utilization is undoubtedly performance. Memory utilization affects performance in two primary ways. The first is the space-time tradeoff. A great many algorithms can be tuned to reduce memory consumption in exchange for
an increase in the number of instructions the processor must execute, or vice versa. The second way in which memory usage affects performance is much less antagonistic. Even the fastest hardware has not only a limited amount of memory, but limited memory bandwidth. The less memory consumed by an application, the less taxing the application is on these resources and, in general, the less time the processor spends stalled and waiting for data.

Other than performance, the software quality most affected by memory optimization is understandability. This can be said of performance optimization in general. Sometimes the most computationally efficient way to perform a task is utterly baffling to anyone inspecting the code who doesn't already know what it does. The primary tools to battle this difficulty are good comments and good documentation. However, there are some situations in which a memory optimization affects readability to such a degree that, unless the optimization is vital to the application, it is of greater importance for the application to be understandable than for it to use slightly less memory.

The software principle that most affects memory usage is probably abstraction. The Java language is an excellent example. In Java, instead of having the programmer explicitly free dynamically allocated memory, the language provides garbage collection. This abstraction allows the programmer to spend less time debugging memory leaks and segmentation faults and more time writing useful code. However, the abstraction comes at a significant cost in memory utilization. Not only does memory stay allocated until the garbage collector runs to clear it, the language must use additional memory to keep track of the number of "handles" to an object to know when it is no longer in use and can be freed. Abstractions are almost universally less efficient than performing the same function manually, but they often provide other highly beneficial advantages to the programmer. The right balance between abstraction and efficiency depends on both the application and the environment.

Chapter 10
J. Henits

Ideas presented in chapter ten of *The Mythical Man Month* by Frederick P. Brooks, Jr. center on the importance of having formal project documentation as part of the formula of a successful project. If a project is worthy of undertaking, it then deserves the basics of identifying the: who, what, when, where, and how much. The chapter focuses on the importance of documentation for project or program management purposes. This is in contrast to high level architectural or detail design documentation, which is intended to serve the detail implementation team, and eventual ongoing support and maintenance requirements well beyond the initial production release, and well into the later phases of a product’s life cycle. The set of documents described are shown to be management tools, which can be applied to not only software development projects, but also any type of project irrespective of discipline from the quick and simple, to the most complex and lengthy.

Project Management Process is Universal

All projects share a basic set of management principals. The *what, when, how much, where* and *who* need to be defined. The “*what*” defines the desired end product or result of the effort expended. If no clear definition exists for a program, it is impossible to continue and further define the remaining elements. Most serious is the fact that the end goal can never be realized, since it is not defined. The lack of having a formal specification may not be as likely as having
one that has shortcomings. The murkiness of a marginal specification can provide a false sense of security, while allowing too wide a latitude for the program team to run unbounded. Projects will benefit from clear and concise formal specifications, and iterative revisions may be useful as the project proceeds.

The “when” of a project refers to defining manpower requirements, and the scheduling of resources. The schedule defines who is working on what, at any point in time. The schedule or project plan incorporates many decisions and reflects many constraints and bounds stipulated in the accompanying project documents. Schedules have to take into account estimates of time and resources to create the end product or result, but schedules are also influenced by external pressures such as a desired target delivery date defined by a customer opportunity or market opportunities. Budgets and schedules have a direct consequence on each other, as human resources usually account for the highest cost component of a project. Lost market opportunities can have an even higher loss consequence.

A projects “how much” attempts to accurately define a projects development cost and the resulting product manufacturing costs. These budget estimates are as important to a project’s success as any of the other elements discussed. Development budgets can drive such decisions as outsourcing development to lower cost groups, in the increasingly leveraged global economy. Likewise, production location maybe determined by target local markets served and market pricing constraints.

“Space allocation” is identified but is given little to no description in chapter ten of The Mythical Man Month. Perhaps deemed to trivial to discuss in the context of the state of development in the 1970’s, space allocation can perhaps take on an expanded definition today. In the 1970’s, the internet did not exist, but today, virtual teams for many projects are the norm. The proliferation of high-speed internet access has made practical, virtual teams working together as if they shared the same office space. Email, instant messaging, collaborative design tools, and collaborative scheduling and meeting tools make a seamless distributed workforce a reality. Costs for virtual private network endpoints and access, incremental IT infrastructure costs for initiatives like security, replace a projects cost estimates for brick and mortar space, at least to some degree.

The “who” of a project refers to the team members and organizational structure defines each team member’s role. Clear responsibilities and compatible personalities are just as important as competency.

In summary, no single element is any more important or crucial then another. A fine tuned team is “running on all cylinders” given each of the components required for success.

**Formal Writing Has Its Benefits**

The process of committing each project management element to paper in a formal document process has many benefits. The act of writing involves greater effort and thought, which only helps the ability to communicate ideas, objectives, and direction with clarity and conciseness. Inconsistencies may appear by virtue of writing and reviewing the primary core project documents. The sharing of written documents is the most efficient means for communicating decisions to others.

**The Managers Tools**

The project objectives, specification, schedule, budget, organizational structure both physical and definition of individual roles and responsibilities form the core documents are used by a
Manager to gauge conformance to plan. Events unforeseen can be assessed, evaluated in the context of the original plan to determine effect, and changes to the plan can recalibrate the team’s activities. These core project management tools do not necessarily guarantee a successful program, but the team will know exactly where they are every step of the way in their journey.

**Risk Analysis**

Projects may encounter many risk factors. Some examples are:

- personnel shortfalls
- unrealistic schedules and budgets
- developing the wrong functions and properties
- developing the wrong user interface
- continuing stream of requirements changes
- shortfalls in externally performed tasks
- real-time performance shortfalls

Examples of only the first identified risk of personnel shortfalls can be further elaborated as:

- educational level
- years of experience with the company, indicating a knowledge of company standards as well as loyalty and dedication
- years of software development experience
- years of experience in the domain
- years of experience in the language
- years of experience on similar projects
- years of relevant specialty training

The formal documents explored so far track cost, schedule, effort, and quality. A formal method of identifying risks with metrics is a prudent addition to the Manager’s core document set for project management. Risks are too important to simply be absorbed into schedules and team organizations, based on one-shot implied assumptions. A risk management plan that identifies risk factors, measured with metrics with risk mitigation contingencies explicitly defined can only strengthen a comprehensive management plan for success.

**Project Estimation**

Much of our examination of the methods of project scheduling and budgeting follow a conventional process where requirements are well defined, project teams are in-place and jointly team members contribute to a working plan of schedule, budget and resulting expectations. Even with all these pieces in-place, we focus on the difficulties of achieving success with our attempts at predicting the “bumps-in-the-road” and our plans to circumvent the obstructions we might encounter. This scenario is the perfect ideal scenario, even with its challenges, that our processes are formally grappling with to predict and control.

On many occasions, it is necessary to estimate project costs and timelines without having the benefit of formal requirements or the staff on-hand to provide estimates for vague requirements.
University research initiatives for example must estimate project costs and timelines well before resources are in place. The process to compete for project funding begins well in advance of having all of the “knowns” defined. This scenario is not all different in industry where Request for Proposals (RFP’s) are solicited to companies, where competitive responses determine the outcome of who gets awarded the contract and who loses. These competitive responses are difficult expeditions in estimating, that perhaps can benefit from additional research into the methods and discoveries of how to better characterize the tasks involved in creating accurate plans for projects that are narrowly defined, but still contain much work to be performed.

Software Qualities and Principles

Two important aspects of software engineering are maintainability and visibility whose success depends upon the level of documentation supporting them. Maintainability is a broad term involving any activity that modifies an existing software system. In this case, these modifications are most likely not going to come from the original designers or creators of the system and thus they must be provided with robust documentation and specifications for the system in which they are to perform maintenance. This documentation will provide the necessary insight into the original goals of the system including but not limited to areas such as performance, reliability, robustness, portability, and usability. It is important that the formal documentation be clear and concise so as to maximize the understanding of the current system and to ensure that any changes made do not undermine it. Visibility is the set of documentation, which depicts the steps and status of all aspects of the development and design processes. A level of transparency is created where all team members now have the ability to track and measure qualities such as progress and timeliness so as to ensure that the software team is working together towards the common project objective. From both a managerial role and as an engineer this aids in the decision making process where insight can be given as to the possible implications that can occur. Another area where this documentation can be further seen as invaluable is when the team losses or gains members. In the event that a member is removed from the team their work will be easier to comprehend and continue with if, all of their progress has been documented whereas without this information progress will be limited to information through word of mouth. Likewise, if a team member is added it will be a more efficient and timelier process to have the individual read through concise formal documentation to get up to speed with the rest of the team members rather than time being taken away from others to bring the individual up to speed.

Anticipation of change is closely related to maintainability where formal documents can be drawn to allow for a level of flexibility and to weigh its impact on the design architecture and scheduling set forth. In this case, it may be necessary to modify the original specifications as to minimize the impact of these modifications to the original system in the future and to ease their integration. If the design requirements and specifications are too rigid then not only will difficulty arise in future maintenance but also market and client demands may significantly increase costs and time to delivery. For this reason having, a built in flexibility will allow changes to be evaluated minimizing their impact to scheduling and costs and to measure the effects to the design and architecture of the product allowing time for these implementations.

Chapter 10

D. Keener
Introduction:

In Chapter 10 of *The Mythical Man-Month* the author, Frederick P. Brooks, explains the importance of documentation throughout the process of developing software. Brooks’ stance on documentation is that it is not only crucial to the design process but is extremely helpful, and helps a manager perform the majority of his managerial work by forcing a reflection upon his own thoughts and the thoughts other people involved in the design process. Among the standard documentation that is required for a program there are the Objectives, Specifications, Schedule, Budget, Organization Chart, Space Allocation, Estimate, Forecast, and Prices. All of this documentation is a must in any project.

Key Ideas and Points:

Eighty percent of a manager’s duties are free of paper and computer. A managers duty is to communicate with team members, make decisions, and consult people inside and outside of the project. The other twenty percent is documentation, albeit extremely tedious and most managers dread it. Documentation stands as a reference throughout a project, to the people working within the project and people trying to use or edit the software further down the line, and often it is the place where ideas on design are set in stone to help the project move ahead without second thoughts. In the writing and examination of documentation is where many times continuity errors may be caught or problems will be realized before being implemented.

Objectives is the place where goals are set, constraints are realized and stated, and the priorities are set regarding functionality, interface, security, interchangeability, and data storage. These measures set at the beginning of a project will act as a guide throughout the rest of the project as to how to implement certain items and functionality.

Specifications is the major documentation throughout a project, it is to be started early in the design process and will not be completed until the project is complete. It is a constant running documentation of the program’s functionality and implementation, which will act as an operating manual among other things. The documentation is to be enormous and very detailed in its descriptions.

Schedule is an outline of deadlines as to the functionality of the software. This documentation is one of the more difficult to construct as it is impossible to accurately estimate the time of certain components, let alone the entire project. Budget serves an important but not obvious role in the process. Technical decisions are forced where they may have otherwise been avoided due to the existence of a budget. It has the same effect on policy decisions.

Organization Chart organizes a manager’s team into specific roles to help clarify responsibilities. This document sets project roles to paper to avoid confusion.

Space Allocation determines where data is to be stored. This is essential, it determines who has what versions of the code written, is it in one place, is it split among different members, these questions have to be addressed. Basically, where’s your code?

Estimate, Forecast, and Prices are all in a constant loop, forecast what it should cost, get an estimate, then compare with the market. These documents determine whether a project is headed towards economic success. A low estimate is desirable.
Project Estimation:

In Brooks’ discussion of the man-month, he elaborates on the difficulties of making such predictions on how to utilize people in a project. This comes into great use when deciding on the Schedule and, ultimately, the project estimate. When a deadline is not met two assumptions could be made, either it was just a miscalculation for that particular piece or the entire project was understated and all pieces will take longer. It is critical to correctly determine which case it is. Without a schedule and deadlines these lags could not be detected.

Implementation Strategies:

Implementation strategies are resolved in both the Objectives documentation and the Specifications documentation. In part the Budget also influences the implementation. A general decision of how to implement the program is decided when writing up the Objectives. Implementation from that point on adheres to the constraints, goals, and priorities of the project. The Specification describes the implementation of the program. This is useful during development and for later use by operators. Implementation must adhere to what the Budget allows, the cheapest implementation would more often be ideal.

Team Organization:

Team organization is completely dependent on the Organization Chart that is put together by the manager. That chart tells everyone they’re role and responsibilities to clear up any confusion. The chart is critical for this reason. A manager doesn’t want some role to be overlooked and end up with some gap in the development process.

Correctness:

Documentation is strongly linked with correctness, as a software quality. Documentation ensures that a project is implemented correctly. In Specifications the capabilities of the program are clearly stated and so as long as the program adheres to those stated in the Specifications the program is functionally correct. Performance and scalability that is needed for a program are vaguely included in the Objectives and determines if the program meets nonfunctional requirements. It is crucial that the specifications are written carefully so that the program can be called correct.

Maintainability:

In order for a program to be maintained efficiently the person doing the maintenance has to be familiar with the program, in its implementation and functionality. For this the person needs access to a Specification of the program. This document explains the ins and outs of the program. The person will also need to be able to access the code directly and understand it. Documentation inside the code will assist in this.

Incrementality:

As part of a Schedule, a project is divided into different steps, increments. Certain increments have to be finished at certain deadlines. Incrementality allows a project to be tested at different points during the development which makes convenient deadline goals. Have mock versions working by some date or have a certain piece working by a certain point. The more incremental a project is, the easier it is to produce a schedule and measure up to the schedule.
Chapter 11  
M. Kosa

The quote referenced from Swift adequately depicts the main idea of this chapter. “There is nothing in this world constant but inconstancy.” In a perfect world there would be an exact and obvious solution to every problem addressed. However, this is not the case. In fact there are many problems that still do not have answers, or have been answered and are now inadequate. This is where Planning to throw one away comes into play. Planning to make a pilot version and allowing for changes to be made in the future paves the way to easy modifications later on.

Chapter 11 gives the example of building a pilot water plant that processes 10000 gallons of water a day before attempting to create one that uses 2 million gallons a day. This idea of getting your program to work for some before it can work for many is very important when designing software. This is especially true today with the widespread use of internet based applications. The important thing is to distinguish from producing a pilot version where it is planned to be just that, or when a pilot version is created and passed as the actual application. The concept this chapter enforces is, of course, to plan to throw this pilot version away and learn from that, hence the title “Throw one away.”

Planning for change is a very important concept behind the main idea of throwing one away. Since change in any application is a necessary attribute that links all programs that will survive into the future, one must prepare for this change rather than assume there design to be perfect as is and that it will not need to be changed. The chapter lists the key features of planning for change as careful modularization, extensive subroutining, precise and complete definition of intermodule interfaces, and complete documentation of all listed. The most important feature of any program is the use of accurate and clear documentation so that less error will occur when changing the program in the future. In addition, the version of the program itself should be clear to the users and those working on it so that miscommunication and misunderstandings do not occur. An example of a poor use of this technique, in my opinion, is how java’s latest version 1.5 is also referred to as 5.0. The constant back and forth interchangeability of the name of the version is misleading to those not entirely familiar to it.

Organization is extremely important to the success of any project, regardless of the type of problem, software or otherwise. In software design, organization is important at every level. Each part of the programs design must be managed and documented in order to allow for the overall success of the final product. Just as an application being produced must be equipped for changes, the structure of manages must allow for changes as well. Chapter 11 gives the example of IBM’s dual ladder of advancement which has a managerial ladder, and a technical ladder with 4 subdivisions each. Also the book stresses that this ladder system and the way it interacts does not belittle those farther down on the ladder.

“Two steps forward and one step back” really focuses on the way software develops. More time and money goes into debugging and patching a program than actually creating it in the first place for most applications. The book also addresses an evil property of programming all computer scientists are familiar with. This is that when one bug is debugged another one will frequently arise. Chapter 11 points out the interesting theory that eventually, in some cases, the program can reach a point of atrophy. At this point the program is producing more bugs fixing the problem at hand than there was when the program had the original bugs. This is usually the point at which the software process is stopped, and then restarted.
An implementation strategy that this chapter enforces is the strategy of planning for change, and planning to throw the pilot version out in favor of a better final product. Many companies employ this tactic with alpha/beta releases of their software. This allows them to see what a common user, using common hardware, will experience when using their software. The information, problems, and input they receive from this implementation are invaluable. This allows them to address these things directly and prepare their final product in such a way that changes in the future will hopefully be kept to a minimum.

Team organization is obviously fundamental to any group project. The software side of the world is not an exception. Chapter 11 suggests a dual ladder advancement structure for its team organization. This structure promises very limited barriers due to sociological breakdown. Having a managerial ladder and a technical ladder, the dual ladder has structure within structure. This type of organization also allows for changes and restructuring which is important to the success of the team as a whole.

Performance is the goal of the throw one away method. The whole idea of the pilot version is obtain a level of performance that will cater to the overall performance of the final product. Creating a version of your program that is, from the beginning, planned to be thrown away clearly shows that performance is more of an issue than any other principle. In addition, correction can be seen as an important mission of the throw away technique. This is because the program must work accurately and adequately for its intended audience. When the pilot version performs as expected, correctly, there is a much better chance that the finished product will also work in this manner than if the final product was produced without this pilot version. Lastly, but of course not lest important, would have to be the principle of maintainability. As previously stated, the target of design the pilot version to throw it away is to produce an application that is not only planning for change, but is also ready. This will ensure that in the future, changes will be easier, less time consuming, and more effective.

Chapter 11

D. Lanier

Since the age of computers, software engineers have attempted to deal with the problem of users that constantly change their expectations for software. In Fredrick Brooks’ book *The Mythical Man-Month*, he wrote a chapter entitled, “Plan to Throw One Away” which outlined the problems and ways to overcome the prospect of unexpected change. Software engineers have problems with trying to complete a program without adequately testing it, or excluding some the features that the users desire in their programs. However, deadlines influence decisions made by the software engineers on whether or not to submit software to the public before it is finished. Brooks puts this ideal into perspective, “The management question, therefore, is not whether to build a pilot system and throw it away. You will do that. The only question is whether to plan in advance to build a throwaway, or promise to deliver the throwaway to customers” (Brooks, 116). By inquiring further into the issue of change, software engineering can improve to create more effective process models, implementation strategies, and evolvement.

The idea to create a more robust program can start by creating a pilot plant. Brooks explains that many other areas of engineering already use this method. For example, “Chemical engineers learned long ago that a process that works in the laboratory cannot be implemented in a factory in only one step” (Brooks, 116). Like chemical engineering, software engineering contains the problem that once the process or creation grows too big, it reacts in different ways
than expected. It can also become awkward to use. A way to overcome this problem is to build a “pilot program” or a prototype that will be expected to be thrown away after it has been created. By doing this, the software engineer can see which problems will occur and how to make their program different so these problems will not appear again. If a program is launched without use of a pilot program, users may find changes to be made to the program that are unfixable. There is a need for a new program to be made from the ground up anyway. Therefore, it may be useful to plan to start over from scratch from the very beginning.

Change can affect every design aspect for a software project. Thus, good documentation is required to understand what exactly needs to be fixed. “Every product should have numbered versions, and each version must have its own schedule and a freeze date, after which changes go into the next version” (Brooks, 118). This practice of good documentation will lead to adaptability to different issues that appear. However, a lot of software engineers opt to not document their designs. This, Brooks says is because they cannot create a defensible decision that will not be criticized. Software engineers must overcome their personal issues of pride to create an efficient workplace.

Change will not become as big of an issue if the workers are trained briefly in opposing fields, and attempt to work a small amount in different areas of the project.

Once an adequate program has been submitted to the public, users will find more and more bugs in the program. There is no way to have a program that is one hundred percent correct without any errors. As soon as bugs have been reported and fixed, the system becomes more and more complicated. Also, the people who are fixing the code may not have been the people who implemented the code in the first place. This causes a problem because if the direction of the original program was different than the direction of the current program, eventually the program will cease to be fixable. The modules also may not correspond enough. In addition, testing might be an issue. If the program’s bugs are fixed, and not retested from the beginning, there could be another bug from the very beginning that appears or recurs which was overlooked. “Any attempt to fix [the bug] with minimum effort will repair the local and obvious, but unless the structure is pure or the documentation very fine, the far-reaching effects of the repair will be overlooked” (Brooks, 122). This problem returns to the issue that the software engineer must plan to throw their program away and start from scratch with documented designs that will ensure the same problems will not happen again.

The chapter in Mythical Man-Month entitled, “Plan to Throw One Away” discusses how software process models can be created, implementation strategies, and team organization. Creating a good software process model implies a multitude of different approaches and making a decision to make one that fits closely to the project that the engineer is working on. The software engineer has to understand that there may be new implementations that come up in the
middle of building the project. With this in mind, the software engineer must plan to make a prototype to throw away afterwards instead of making the finished product right off the bat. By doing this, the engineer can clearly map out the problems and create a more accurate software process model. The teams should have more experience with how the code will fit together, thus, making a much more efficient program with less room for errors. In addition to this, after the prototype is discarded, the software engineer may also decide to use a new implementation strategy. For example, if the original code was being implemented with the object-oriented paradigm, and some of the system’s functions were becoming spread out or contained poorly created hierarchies, a new implementation strategy may be employed. The engineer may want to keep with the object-oriented paradigm, but create better hierarchies that contain similar functions on each level. Another approach could be that the engineer starts over in a new paradigm, like aspect-oriented programming. This will allow the program to have more versatility and have less room for errors. Along with the program’s versatility, the software engineers should consider the team’s versatility. “Each man must be assigned to jobs that broaden him, so that the whole force is technically flexible” (Brooks, 118). If the team takes on these jobs that are just out of their fields, they may acquire new knowledge that will help them in their next assignment, or add insight from looking at things from a different point a view. By organizing teams like this, there should be a more robust program as a finished product.

The chapter, “Plan to Throw One Away” also can be related to software qualities and principles. Some qualities include “Reparability/Evolvement,” and “Robustness/Performance.” An important principal related to the chapter is “Anticipation of Change.” Reparability and evolvement are extremely important to any kind of program. As many veterans of software engineering say, over 40% of time is spent maintaining and repairing the program. Therefore, the program should be written so that the programmers can go back and change what needs to be updated. If this concept is incorrectly implemented, it can lead to dead ends. Brooks states, “All repairs tend to destroy the structure, to increase the entropy and disorder of the system… As time passes, the system becomes less and less well-ordered. Sooner or later the fixing ceases to gain any ground” (Brooks, 122 - 123). As time goes on, the program is still usable, but not repairable. If this is in fact the case, the program’s performance will decrease unless the engineer decides to start from scratch. So the software engineer must make the program robust, and foresee as many possible changes before they happen to prolong the life of the program. Since it is impossible to predict every little error, there should be alpha and beta tests for the prototype and the actual product as well. The performance will also be affected by how diverse the planning and testing is. If the software engineer has anticipation for these changes, the prospect of forced change won’t be as much of a shock to the system of organization. Therefore, management should map out possibilities of what might happen, and tell all the programmers about some of the possible paths, allowing the code to be mutated into something else later. The idea to “Plan to Throw One Away” becomes extremely valid when dealing with software. Software is ever-evolving and prototyping is one way engineers can anticipate the change.

Chapter 12

R. Norbeau

Chapter 12 is entitled “Sharp Tools”. It discusses the tools a programmer needs to use in order to complete a project. These tools do not refer to physical tools necessarily, but to the concept of a tool. A tool is something that aids someone in performing a task. He breaks up the tools required into computer facility, operating system, language, utilities, debugging aids, test-
case generators, and text-processing system. He is arguing that common tools should be set in place by the company so that communication between programmers is easier since they will all be familiar with the same tools, and the tools will be able to communicate with each other more easily than if everyone used their own tools. He also argues though that there are times when individualized tools are needed, and would bog down the usability if they were attached to the common tool. He explains through experience, although not in theory, what tools and techniques for applying the tools have been the most productive and suggestions for the future of programming in teams.

There is a separation of target machines and vehicle machines. Vehicle machines are used for creating the software and target machines are the machines that the software will be used on, and test with. Scheduling time on a simulation machine is a tricky matter, since it can be a new machine and there are limited numbers of them. From experience he has found that it is best to schedule a team to use the simulators in large blocks of time, because the fact that they are working continuously increases productivity since “sustained concentration reduces thinking time”. When a program does not work as expectedly there are 2 possible causes. It could be a hardware problem or it could be a software problem. He suggests that too often we assume that the hardware is trustworthy and that the software must be the problem and we should use logical simulators that we run on a trusted machine instead from the start. Without this it causes a problem of uncertainty of where the error occurred and causes the programmer to be less motivated in debugging his code because the problem may not even resolve there. Compilation should also be done on dependable vehicles. The advantage of high level languages is that we can test the program out on the vehicle machine because the high level code is trusted equivalent versions of machine code on the vehicle and target computer. He discusses a library format of version management of software, which gave permissions to certain folders of the library to certain programmers and the project manager to have access to all the programmers’ folders for integrating them with each other. This is very similar to what businesses use nowadays and call version trackers. Documentation systems are very important too. A lot of the information he talks about in this essay has already been implemented better than described by him, but he has formulated the ideas that it should go in this direction. For example, Integrated Development Environments have been a huge progression to his ideals. Combined with high-level languages many of his ideals have been reached. For example, Java IDE’s are capable of generating documentation based on the source code. This is extremely useful for programs that are meant to be used by other programmers. IDE’s also provide excellent debugging through use of the language that they implement. Java for example helps you with debugging as it runs by pointing out where exceptions occurred and handling memory for you. Debugging is a key utility needed to minimize project time. It helps the programmer in spotting errors. He believed strongly in high-level languages and interactive programming, because they increase performance substantially, not only in ease of use to code but ease of readability to debug and built in debuggers. He argues against claims that high-level languages reduce speed, lengthen code, and limit abilities. Compilers have improved as much as he expected in optimizing code length and runtime abilities to points that if programs of large size were coded in assembly, they likely would be larger and slower than compiler optimized code. Assembly though, should be used in small portions of code that have a long execution time to try to focus on that and speed it up if execution time is a key goal.

He suggests a standardized software model, to the extent that can be done without encapsulating too much in the model that isn’t needed by everyone. For example, with software
tools, he suggests that there should be common tools that everybody uses. But there should also be individualized tools that not everybody needs, because when everybody uses a tool they will need to communicate each other about the tool and communication will be facilitated if the tool is standardized. If the tool encapsulates too much specific functionality though, performance will go down because the tool will be harder to master. This analysis of software model also applies to the concept of high-level languages, testing machines, etc. The key is to make the project as productive as possible in a certain amount of time. Team Organization and Project Breakdown would be a lot easier to manage with the library that he discusses. It allows the managers of projects to see all the development of code by each programmer on their team, and decide when to integrate them together. Through this process of breaking a project into separate parts and being organized and watched over by the manager, the manager will gain a sense of how long projects take, by noticing how hard certain sections are and how long it takes certain programmers on his team to do certain tasks. Then when a manager decides that a code is ready to be released to the current version library, he can authorize it and the code cannot be changed there except for bug fixes. The separation of programmers from accessing other programmer’s code is a key separation of tasks goal. This allows programmers to not depend on other programmers to finish a section before they start their own section. The programmers just worry about their code and how it will work assuming that all other sections of code it needs to correspond with have been completed correctly. Then the manager is the one who gets to decide when it is time for integration of these parts. This concept is now widely used as an excellent version management and isolation of parts system in the working world.

Portability is a quality that is well achieved through the idea of using a high-level language as he suggests. High-level programming languages have the advantage of being much more portable than lower level languages. For example, if one was to write a program in assembly, it would only run on systems that support the same architecture of assembly commands. When it is written in Java though, which is very high-level, it has the advantage of being extremely portable. Java compiles programs to byte-code by the machine that coded it which is then compiled to assembly code specific to the machine it is running on. This enables any computer that has a byte-code translation to assembly code generator to run Java programs. This is why Java has now become so popular of a language. The productivity software quality is also very relevant to this chapter. He discusses in length how to maximize productivity. Standardization of tools is necessary to maximize productivity. It has been proven that this is true because it improves communication between the team members. If everyone was using different tools the communication overhead would be much higher. Also he discusses using large blocks of time for debugging instead of small ones, because it keeps people in the same train of thought and therefore causes the computers needed for debugging to be less needed and speeds up productivity. Separation of concerns is a key software principle that is mentioned in this chapter. The idea of using the library folder structure for coding causes separation of concerns. The programmers do not need to understand how the whole program works and wait for each other to be done with each others section to start their own. It is broken down to the individual parts that can be coded on their own, and each programmer works on them in their separate folder. The manager then is able to integrate these together as needed when each programmer is finished coding.

Chapter 12

P. Rago
My assigned chapter, “Sharp Tools”, deals with the ever important topic of the different tools involved and employed in a programming project. These tools include, but are certainly not limited to, the computer facility, operating system, language, utilities, debugging aids, test-case generators, and a text-processing system (now known as a word processor). Frederick Brooks makes it known that unlike a mechanic, a programmer mustn’t be constrained to a set of tools which he has collected. In the chapter, Brooks seeks to prove that treating the programmers’ tools in a similar manner to the mechanics’ is an imprudent practice to exercise. This is because individualized tools hinder communication, an indispensable component of constructing software systems. Also, and of equal importance, in this field as much as any other the technology changes extremely rapidly. As such, the tools must change with it.

Language is stressed one of the tools of most importance. Brooks talks specifically about the value high-level language. He points out that productivity and speed are the foremost reasons to use high-level language. “The debugging improvement comes from the fact that there are fewer bugs, and they are easier to find” (Brooks, 135). This improvement to which he refers is actually a result of not only the high-level language, but also the compiler. For example, when programming in a high-level language such as Java 5.0 (also known as 1.5) and using a high-level compiler such as Eclipse, literally all of the syntactical debugging is done before compile-time! This means the instant a syntax error is made by a programmer, he or she will know about it, and it turns out this is beneficial and saves time (thereby increasing productivity) for two reasons. First, it saves time by notifying the programmer of their error immediately, so the programmer doesn’t go on programming similar errors without even knowing about it. Secondly, the instant error-notification serves to help teach the programmer as they are programming, which in turn saves them much grief and time. Assembly language, on the contrary, is much harder to find such bugs. Brooks also asserts that such classical objections to the tool high-level language are no longer valid. The objections are that the programmer cannot do exactly what he or she wants, the object code is too big, and the object code is too slow. With such advances in the high-level languages and vehicle machines, the code is no longer big and compiling it is now longer slow. Additionally, if there is some function that the programmer feels he or she cannot do, this isn’t so – he or she must merely do some work to find out how.

Brooks also emphasizes the importance of the documentation system which he calls the “text-processing system”. He claims of all the tools, this one saves the most labor. This of course is counter-intuitive because most people would wonder what word processing has to do with programming. However, Brooks maintains that the word processor is still the biggest time-saver. He states that without a word processor, manuals would be later and far more cryptic. Such manuals created are used as a reference to the system being built. The reasons Brooks believes the word processor is so important for documentation are obvious: one may write, delete, and revise all very quickly and without any physical wastage. Today, word processing systems have advanced even further allowing even better collaborative documentation. Nowadays, a team can work on the same document all at once, allowing for exceptional efficiency. This also leads into another productivity-increasing tool – interactive programming. This is similar to the mentioned collaborative word processing and for the exact same reasons increases efficiency in building the system.

Evolvability is not only a representative quality of software products, but also it characterizes the tools used by programmers. As we have already seen, the word processing and programming environment have evolved a great deal, but this is only the beginning. The language, test-case
generators, as well as all other software engineering tools other must evolve too. But as with software products, modifications for these software engineering tools start at the design level and then proceed to the implementation of the product. “For example, if one decides to add a second story to a house, one must first do a feasibility study to check whether the addition can be done safely. Then one is required to do a design, based on the original design of the house. Then the design must be approved, after making sure that it does not violate the existing regulations. Finally the construction of the new part may be commissioned” (Ghezzi, 25). This applies to the tools as well, because in order, for example, to build a new language (the quintessential tool of software engineering), all previous languages and their implementation must first be considered. Separation of concerns is a software engineering principal that also applies to the software engineering tools in a similar way. For example, when conceptualizing the problem, a tool such as a test-case generator must be designed and implemented in such a manner that it is concentrated on individually, apart from all other concerns.

Chapter 13
C. Revel

Brooks divides the task of producing a working application into two major tasks relevant to preventing programming errors in applications: Minimizing errors introduced in design, and minimizing the errors introduced in assembling disparate components into a single application. Brooks applies lessons learned personally and through colleagues to illustrate pitfalls of modularized programming and failure to properly plan for application functionality.

Summary

The most obvious element of creating a bug-free application is to create a bug-free specification. Brooks makes the claim that while human error in writing designed code is indeed a critical opportunity to introduce code error, so too is the application of incomplete or poorly communicated specifications. According to Bell’s V.A. Vyssotsky: “Many, many failures concern exactly those aspects that were never quite specified” He goes on to clarify that while the lack of specificity is not a problem for the individual teams, it becomes a significant problem when integrating multiple components, for “They won’t tell you they don’t understand it; they will happily invent their way through the gaps and obscurities.” For this reason, having an external group test the specification for clarity and completeness is essential.

To that end, a top-down design stance should be adopted. A popular one, outlined in 1971 by Niklaus Wirth, begins with the creation of a rough solution, followed by a series of “refinement steps” which serve to break the tasks encountered into successively smaller steps that are more easily handled by individual groups. Brooks argues that this approach helps reduce the frequency of design flaws by more clearly defining the requirements and functions of each module. Further, by having independent modules, each module will be simpler in function and requirement, resulting in fewer errors. This has the added benefit of allowing for more rapid module development, which lends itself to earlier testing of the individual modules. Finally, this methodology allows for easier and earlier identification of excessively complex code, allowing developers to replace the offending module with simpler module or set of modules.

The final element Brooks details for designing a bug-free application is the replacement of branch statements with control structures. In short, the elimination of GOTO statements in favor of bracketed IF…THEN…ELSE, DO…WHILE and FOR statements. According to E.W.
Dijkstra the use of branch statements lend themselves to logical errors. It should be noted that
this principle has been adopted by many if not all the Computer Science faculty at The
University of Connecticut.

In addition to proper design, Brooks argues that proper debugging practices will reduce the time
spent on eliminating errors from programs. In addition to a brief history of the process of
debugging applications, explaining the origins of many deprecated debug procedures that are
now seeing a renewed relevance in module verification, Brooks argues that there are certain
procedures on the system level that will minimize the introduction of errors.

Brooks’ first suggestion smacks of common sense, if not common practice: use debugged
components. By using modules that are [as] bug-free [as possible] any application errors that
arise from the combination of said modules can be more easily traced to their source.

Brooks also recommends the use of application ‘scaffolding’, that is, the creation of a bare
structure designed specifically for debugging, into which can be placed individual modules for
the purpose of testing, rather than execution. This method can also be used for file access,
through the use of “dummy files”, or “miniature files”, that is the use of data files containing no
or few records, to ensure that they will be properly read by the application.

Brooks also talks of change control. In my own experiences at ING last summer, change of
production applications was strictly controlled through the use of a complex set of administrative
and peer controls, ensuring that only one person was able to work on a specific component of an
application at a given time. When finished, the changed code was re-integrated onto a test
system, run through a battery of tests to ensure proper functionality, and the module was checked
back into the ‘available’ library, along with any change comments and the documented change
control record. This rigidity was necessary as the applications dealt with customers’ financial
data, however the system worked remarkably well, considering the size of the department
(approximately 500 members, arranged in 12-15 teams) and the size of the applications in
question.

The method employed by ING also followed Brooks’ last recommendation: Add one component
at a time. By minimizing the change to an application between tests, one can get a much more
accurate idea of where anomalous code lies. It follows logically, that if a new or modified
module is added to a previously correctly functioning application, the problem will likely lie in
the module added.

Minimizing Errors

A topical examination.

Implementation Strategies

As alluded to above, Brooks advocates a stepwise implementation strategy, the overreaching
factor being the incorporation of small changes followed by thorough testing. In his article,
Brooks comments on the common practice of assembling the entire project and “flipping the
switch”. Brooks argues that the time and cost (in man-months and in capital) of implementing
this methodology will be far higher than the cost of testing each individual change and addition.
For one, creating a monolithic application and attempting to track down each bug and
abnormality in it increases the available code for each specific error to be hiding in, thus
stretching out the amount of time required to locate each bug. Secondly, by testing each module
independently, testers are able to work in parallel, reducing the overall time required to complete
testing.

Team Organization

Brooks is a strong advocate for the surgical team organizational structure, with a tight collaborative effort among the heads of each programming unit. Within this construct, there lies a convenient method for controlling flow and structure of the program. Minimizing the number of voices heard in designing the individual modules of an application will minimize the risk of a breakdown of communication among the program architects, and increase the likelihood of a well defined application specification.

Error Free

As it relates to…

Correctness

“[Software] Correctness is a mathematical property that establishes the equivalence between the software and its specification.” The ease of characterizing correctness of a particular application or module is largely dependant on the completeness and specificity of its specification. As Brooks argues, the creation and verification of a formal, complete specification is essential in reducing the bugs present in the assembled system.

Verifiability

Verifiability of software is essential to its being capable of performing the task for which it was created. Using Brooks’ methods of software design and error-checking, software teams can more ably check to make sure that software is performing as it should, by splitting major components into smaller modules that serve a specific purpose. The reduction of size and complexity of these modules allows for faster testing of each module and reduces the likelihood of errors being introduced at all.

Modularity

Brooks is a firm advocate of the separation of tasks into discrete, compartmentalized sub-programs, or modules, that can be later assembled and run as a single application. He argues that the ability to break down tasks into smaller, simpler components minimizes the introduction of programming errors, while at the same time, having multiple, small components increases productivity by allowing testing to be completed on some components before the entire application is assembled. This application of stepwise refinement allows not only allows work to progress in a nonlinear fashion, but also allows for the reuse of code. Yet another benefit of this approach results in the ease of testing from the inherently simpler design of each module reduces the frequency of introduction of code-errors.

Chapter 13

J. Schindler

The Whole and the Part, this chapter had a few different parts that it touched on, but they were all about debugging. The first part called Designing the Bugs Out is about how to code so that you program does not have any bugs to begin with. The second part called Component Debugging is about how debugging has evolved over time starting from the old machines that used punch cards to what was modern day at the time it was written. The last part called System Debugging is about how to go about debugging. All three of these parts are
essential to the debugging process of writing a program. Without one of these 3 parts the other two would take a lot longer or not be able to function at all.

Designing the Bugs Out, the first part that he touches on, is about the bugs that could be encountered way up at the design level. How assumptions that the authors made could be the hardest to detect. That is why he says “Careful function definition, careful specification, and the disciplined exorcism of frills of functions and flights of technique all reduce the number of system bugs that have to be found.”(142) Next he gives a good example of how to test for a good specification, by handing it off to an outside testing group for them to look over with a fine tooth comb. The next topic he touches on is the top-down design and how he believes that it is the best way to build a program in order to get the least amount of bugs, which is from a 1971 paper by Niklaus Wirth. He explains the process of how Niklaus Wirth explains top-down design and how it uses refinement steps. From these refinement steps used in top-down design he identifies modules, and how they are used independently other parts. He then explains why he believes top-down design avoids bugs with four reasons. 1) The clarity of structure and representation makes it easier to spot. 2) Partitioning and independence avoids bugs. 3) Since there is no details the flaws in the structure are easier to spot. 4) The project can be tested at every refinement. Then he reminds us that just because were building it this way doesn’t mean we don’t have to go back to the drawing board sometimes. He then explains Structured Programming, and how we should design programs using loops defined such as Do While or If Then Else and not “unrestrained branching”(144) like Go To. He then finishes this section with explaining what he means by saying “think about the control structures of a system as control structures, not as individual branch statements.”(144)

Component Debugging, the next section he talks about, goes through the history of debugging. He starts off with the beginning, On-machine debugging, which is when the computers used magnetic tape or punch cards. “The programmer carefully designed his debugging procedure planning where to stop, what memory locations to examine, what to find there, and what to do if he didn’t.”(145) This was a very meticulous and long. This was effective but this was way back in the day before PC’s and computer time was valuable. So they came up with memory dumps where the programmer would write the program and run it, but when the checks failed it would then dump all the memory on to paper for the programmer to take back to his desk in order to work on it. Therefore instead of sitting at the computer debugging the program the programmer would sit at his disk with the outputs and memory on paper debugging. This would free up the computer for another programmer to use. With technology advancing and memory growing with leaps and bounds, dumping all of the memory started to become huge and became cumbersome; therefore they came up with snapshots. Snapshots are “techniques for selective dumping, selective tracing, and for inserting snapshots into programs.”(145-6) Then Codd and Strachey were able to come up with a “time-shared debugging” that would achieve the best of all forms. It would have a supervisory program that would decide what program to run. Therefore allowing the programmer to sit there and debug on-machine while allowing the computer to run other programs while he thinks. Now because of the technology that we have a programmer can “program and debug in a high-level language.”(146) This allows the programmer to utilize all the former techniques. In the end he came to the conclusion that as much time is still needed at the desk as well as at the computer, or in other words sufficient planning is still needed.

System Debugging, the final part of this chapter, talks about the different was to approach
system debugging. The first approach he calls bolt-it-together-and-try, which holds the idea that
the sooner its all together the sooner the bugs will start showing themselves. Also, there is a lot
less “test scaffolding” used this way. This he says might look easier and faster at first but out of
experience isn’t. The next one that also looks good is called “documented bug” where a
component is put into a system when all the bugs are known but not fixed. However its is hard
to know all the bugs of something and its even worse to try and guess all the effects that the bugs
will have on the whole system. The next part he calls Build Plenty of Scaffolding, which are
programs made to test but never to release. The first kind of scaffolding is called dummy
component, which only interfaces and maybe some faked data, used to see if it works. The next
kind is called miniature file, where a file is created with only a few records but everything the is
associated with the records is included. A smaller version of miniature file is called a dummy
file. The final kind of scaffolding is auxiliary programs. “Generators for test data, special
analysis printouts, cross-reference table analyzers, are all examples of the special-purpose jigs
and fixtures one may want to build.”(149) Updating or changing the software happens often but
someone needs to be in charge of it. There also needs to be controls on the versions. There
needs to be a “locked-up” copy, a test copy, and a copy that can be played with by others. He
then goes to talk about an old electrical system and how all the wires used were yellow, but
when a bug was found a quick fix would be implemented with purple wire so that it would stick
out. This he says is what we need for programming. Also, he emphasizes adding one
component at a time. This may seem tedious but when bugs are found at least you know that it
is from the component just added because the other components added before all tested without
bugs. Finally he talks about quantizing updates. “The replacement of a working component by a
new version requires the same systematic testing procedure that adding a new component
does.”(150) The most recent version of the software should always be used by anyone making
another component. These quanta should ether be very big and few or small and often, but is
found to be more stable with the first one.

This chapter focuses on three main parts; project estimation, project scheduling and
implementation strategies. They are all emphasized in different ways implying but never
outright said. They are all heavily dependent on debugging.

Project estimations are always done at the beginning and are hardly ever right when it
comes to software. He goes over this in earlier chapters. This is true because usually debugging
takes longer then expected, which he explains greatly with saying, “From all of that, one should
be convinced of two things: system debugging will take longer then one expects, and its
difficulty justifies a thoroughly systematic and planned approach.”(147) This just reinforces
what he said in that when you start the project the programmers are optimists. The estimates are
always made with the idea that everything will work as expected but it never does. However, the
techniques that we use can minimize the amount of time that is needed to debug.

Project scheduling should be taken into consideration all the time. By using the
techniques that he explained in this chapter about how we should be building our projects a more
accurate schedule can be made. For example, say I was building a project top-down like he
suggests in the beginning of the chapter. Then I can accurately schedule how much time I think
that I’m going to take to program each refinement. Then I’m able to see if I’m on schedule with
each refinement, and because each refinement has its own debugging already ingrained in it, I’m
debugging the whole time instead of starting when I’ve finished the whole thing.

Implementation strategies are directly involved with the beginning and the end of this
In the beginning he strongly encourages us to use top-down design. This design promotes the software principle of modularity. Since it uses modularity it also promotes two software qualities by the way its built, reusability and reparability.

Modularity is used very often in top-down design. With each refinement we are taking each of the parameters and breaking them down even more. Making more and more modules to be used in the classes. This is why top-down design is so effective. Modules are a great way to use a lot of software qualities. It’s the easiest way to make the code you write reusable and they are all easily repairable.

If there is a function that needs to be used many times, writing the function once and then reusing it over and over is one of the basic forms of modularity. It takes the code and buts it in a place that can be called again and again. This is reusability. The more a program has reusable code the better it will be because there will be less chances for bugs to get in. If you know that a certain bit of code has no bugs and it works very well then reusing it will mean that anytime you use that code it will work how you want it to work.

Modules are also easily repairable. If you have a program modulated and you use this bit of code many times but every time you call this function it crashes, then you know that the problem is most likely with that module. If you know this then you can build test scaffolding in order to debug this one module. This is an easy way to figure out exactly where it is going wrong and why. This allows for easier repair and in all likely hood faster repair. This is how software principles and qualities can directly affect how one builds and debugs a program.

Chapter 14
J. Stevens

Chapter 14 of The Mythical Man-Month is about delays in the development process. The chapter describes how a small delay of one or two days can cause a project to fall significantly behind schedule creating a disaster for the manager to sort out. The chapter gives solutions for keeping projects on schedule in a realistic manner by using such methods as the PERT technique, the planning of a schedule with realistic time frames and controlled and monitored by devoted staff. The chapter also warns of the danger of poor relations between bosses and managers. The boss may be overbearing and interfere, and the manager may feel as though reporting problems will result in loss off authority and loss of control over their own agenda. This may be prevented if the boss learns to trust his managers, and allows his managers a review period where they may submit honest status reports to be reviewed without the boss taking any action. These steps may help to prevent small problems from becoming huge disasters and delays.

The first step in preventing day-by-day slippage into complete disaster is to create milestones and a PERT chart. The idea of the milestone is that the manager must be specific about what should be completed when, and avoid fuzzy milestones (such as the notion of percentage complete) at all cost. Specific and concrete milestones are undecieving in that if it
says component 'x' will be completed by some date 'y', then everyone can see if that milestone
has been met or not. Fuzzy milestones, or millstones, lower morale and hide the critical nature of
and the completion time of components and hide problems until they become unsolvable
disasters. Helpful milestones can be implemented in a PERT chart.

The idea of the PERT chart is to predict an event three times, and to create a range of
dates in which lateness for a component can be tolerated until the situation becomes critical. The
PERT chart also defines what components are dependent on other components, therefor the
completion time of the dependent component can be re-evaluated easily depending on the status
of the component it is dependent upon. This prevents what the authors calls “the other piece is
late anyhow” demoralization by creating a critical time frames and maintaining programmer
hustle.

The other step is to create an environment that prevents role conflict between managers
and bosses. Many managers refuse to report problems to their bosses because they believe that
their boss will interfere with project and its schedule. This belittles the manager's authority and
ruin the manager's own plans. The boss may prevent this by inspiring the manager to share their
status honestly and openly without threat of interference. For example, the boss may hold status-
review meetings where no action will be taken by the boss, as apposed to problem-action
meetings. The boss must understand the capabilities of the manager and restrain themselves from
interfering unless they are sure the problem is beyond the capacity of the manager in order to
keep the communication open and honest.

On top of that, the boss may select a small team, a Plans and Controls group, to be
dedicated to maintaining the PERT chart and obtaining honest production feedback from the
manager in order to check and modify the PERT chart. This group, being an extension of the
boss, is almost like a buffer, allowing managers to be honest to the Plans and Controls group
without reporting directly to the boss and fearing some overbearing action. This reinforces the
notion that an unbiased status report is more helpful than reports manufactured to be palatable.

This chapter is a good example of project scheduling. Project planning is where the
manager sets the goals for the project, and begins to organize the project and estimate the
completion times for its components according to a time-line. Here the time-line is a special type
called a PERT chart. When creating this chart, many methods of estimation are used to create a
realistic and probable time frame for understanding when a component should be completed.
This time frame shows clearly how long before a late component becomes critical. These
predications may be based on the complexity of the project, the amount of functions in the
project, how many hours per function, and the dependency between components.

Estimating such things in the scheduling process is heavily based on experience in the
field, and can never be truly precise. However, as with the PERT chart, as indicated by the
mention of a dedicated Plans and Controls team, plans and goals may need to be revisited and re-
assessed during the production of the project. There will be problems, and the important concept
here is catching those problems early, and allowing the managers to deal with those problems if
they can handle.

This chapter also demonstrates team organization. It is clear from the chapter than a well
suited team is important in maintaining the project's goals and deadlines. First, the managers and
the bosses must be able to communicate freely and honestly. And the managers should be able to
trust the bosses not to interfere when the slightest hiccup in production is reported (as long as the
manager can handle the problem). Also, it is highly recommended that the boss select about three employees to be a Plans and Controls team that is dedicated to maintaining a PERT chart for large, complex projects. Chapter 14 gives an example where the boss had hired an outside professional organization, A.M. Pietrasanta, to handle this function. Usually such a group could be seen as an irritant, but in this case, the group's talented and unobtrusive. It is part of the team organization to decide if members will be on the Plans and Controls team rather than working directly on the project, or if an outside team should be hired. Being a part of such a team is highly rewarding in that it is the first line of defense against disastrous delays.

One software quality that is evident in chapter 14 is the idea of “visibility”. Visibility is having the development process and the status of all its steps clearly documented and visible for all to see and analyze. Having the quality of visibility allows the impact of development problems and delays to be quickly analyzed and solved. Chapter 14 meets the challenge of creating a visible development environment by suggesting the idea of using a PERT chart, and having a dedicated team maintain it. The PERT chart clearly defines the dependencies of components and gives a range of expected completion times and indicates when a component will become critically late. As development progresses, the PERT chart will need to be kept up to date accurately or it will become useless, this is why the Plans and Controls team is vital to maintaining the PERT chart.

Another software quality seen in this chapter is the concept of timeliness. Timeliness is the ability to deliver components of a project on-time and getting things done to avoid a “software crisis”. The goal of timeliness can be met with careful scheduling, accurate estimation of time-frames, and clear milestones. Chapter 14 meets this with the PERT chart and the idea of sharp milestones versus fuzzy milestones (or millstones). Also, the idea of “hustle” is brought up, suggesting that these techniques keep up morale the team's willingness to want to work on the project. The more hustle a team or individual has, the more on schedule they will be.

The software principle of rigor and formality is found in this chapter. Rigor and formality is the idea of having clearly defined specifications and clearly defined goals for a project. Having a rigid and formal specification and set of goals with well defined time frames (visualized by a PERT chart with sharp milestones) sets the ground work for creating an on-time project. Once rigid and formal documentation is created, the status of the project can be easily compared to it.