Does Your Internal Management Meet Expectations?

Eli Schragenheim,
ELYAKM Management Systems

H.William Dettmer,
Goal Systems International

Projects that fail to meet expectations often began with an ineffective plan. Estimating task duration in projects is one of the chief culprits. Safety time included in task estimates is nearly always squandered, rendering its value nearly useless. Parkinson's Law, the "student syndrome," and multi-tasking all conspire with natural dependencies between activities to effectively assassinate schedules. A new solution to the planning problem is Critical Chain Project Management (CCPM). CCPM can provide about 90 percent confidence of finishing projects at or before planned times, which are inevitably shorter than traditional critical path schedules as planned originally. Buffer management provides warning of danger to the delivery schedule early enough to apply less extreme corrective measures. Schedule, cost, and performance are all enhanced.

More than 20 years ago in a programming class, the professor for system analysis told us that a delay equal to 100 percent of the planned time is certainly within the normal range for any software project.

Is it much different today? At that time we already knew the professor's statement was true, but the question is "Why?" It is easy to accept the idea that young programmers might be overconfident about their ability to write the code in almost no time. It's also easy to understand why a young programmer might think that debugging time might approach zero. However, the reality of software development during the past 20 years should have taught the industry some lessons. There is no shortage of statistics on the ratio between the time it takes to write the code and the time required to realize a working prototype. A lot is known about software development – after all, the disappointments of the past should have made some kind of impression on developers. So why are we continually disappointed? By disappointment we are referring to *internal management expectations,* not our promises to the clients.

Whenever there is a gap between initial expectations and the real world, it should prompt us to review both the way we set our expectations, and the way we try to meet them. When we constantly fail to meet our expectations, we cannot simply justify it by lack of experience or by the significant amount of uncertainty involved. We *do* have the experience, and we are capable of approximately estimating the impact of the uncertainty. Generally speaking, uncertainty should reflect itself in both delays and early completions. Though we may sometimes be disappointed, we should also have experiences where projects finish earlier than expected. But, is this a common occurrence?

Setting Expectations

Let us first review the process of setting expectations. Suppose you are the CEO of a small software company. You want to add a new module to an existing application that will verify the sensitivity of some processes to certain random variables. This module will rely on your current database, which contains historic data. All you are asking is for the module to apply the appropriate statistical formulas and generate a report. Nothing very complex. The technology is understood and the needs are clear. You present your request to your chief programmer and ask how many people he needs, and for how long. The chief programmer, after consulting with his people, returns with an overall estimate of six man-months, and three months actual project duration before a prototype is ready for beta-testing with clients.

Would you be surprised if the prototype is not ready until *nine* months, instead of three? Would you be surprised if it was finished in exactly three months? Could it take a mere two months?

What does the chief programmer mean by three months of total project time? Certainly, the time required to deliver such a module cannot be predicted precisely. It is a random variable that depends on the level of complexity, which often can only be determined after actual programming begins. It also depends on the skills of the specific people involved, what other jobs those people are required to do simultaneously, and how much pressure they are under to finish on time. These are just a few of the parameters that impact actual duration.

Does an estimated three-month completion time mean that three months is the "expected value" – in other words it will take three months *on average?* Is it an optimistic view, meaning "if we are lucky it will be done in three months?" Or is it a pessimistic view, meaning "it should not take longer than three months?"

To understand the answer, consider the question again. What would you – the reader currently in the shoes of the CEO of that small software company – mean when you asked for an estimate: an *average, optimistic*, or *pessimistic* one? Which of these three would give you the information needed for the decisions you must make: a) Will you proceed to develop that feature? b) When will you promise a first working module to your clients? c) How much of your resources must you dedicate to developing that module?

In most cases, you would probably ask for a pessimistic estimation. A reasonable worst-case scenario tells you the potential full impact of that decision on your operations. This is even more the case when you must promise your client a delivery date. You would like to be able to meet your commitment to the client, so you would probably prefer your chief programmer to commit to a date. Now, if your chief programmer understands that this estimate is a commitment, he or she certainly would not want the project to take more than three months. The chief programmer's estimate will probably be based on the pessimistic view where some "safety" time has been added to the average (expected) time.

If this is the case, then why do the chances of finishing the project in a mere two months become so remote? If the chief programmer's estimates typically add substantial safety time to average durations, we should frequently see many projects finish early. Yet this is almost never the case in the reality of the software world. More commonly, it might take six months or more to deliver the module. And if that is true, what confidence should we have in the chief programmer's "pessimistic" estimates in the future?

A Hidden Cause

This phenomenon can be explained by recognizing a vicious cycle. Any adjustment to the estimate (making it even more pessimistic than before) does not improve the chances of meeting the estimated/expected time. This is because the estimate itself, which includes a large degree of safety time embedded in it, serves to prolong the project. The effect at work here is known as Parkinson's Law: Work will expand to consume the time allotted for it.

Applied to projects, Parkinson's Law suggests that the duration of any project stretches – at least – to the full duration of the time planned for it. In other words, once the project is planned for three months, human behavior will not let it finish early. By not allowing the project to finish earlier than estimated, the chances of finishing late actually increase. Here is how this happens.

There are three major human motivations behind Parkinson's Law. The first is called the "student syndrome." The term originates from the academic world where students typically wait until just before an assignment is due before beginning it. As long as we think we have enough time to finish our part in a project, we do not feel any real pressure to get started. Consequently, almost all the efforts are concentrated near the end of the time allowed for the activity, and only very little is done at the beginning. In essence, the built-in safety time is squandered. By the time work is actually begun, a timely task completion depends on mere luck with no safety time to accommodate the unexpected.

The second motivation is embedded in organizational politics. If you estimate the necessary "pessimistic" time to be three months, you certainly are not going to admit it actually took only two months. Even if the estimate was imposed on you, it is not likely that you would admit to finishing early, as you could expect even tighter times to be imposed on you in the future. Here again, safety time is frittered away. So there are two human behavioral issues at work. One says, "I don't have to start right now – there's plenty of time." The other says, "Even if I finish early, I'm not going to tell anyone about it."

Lastly the third motivation is even stronger than the preceding two: The code can *always* be improved. We can add features that were not requested by the client, but are nice to have all the same. We can improve the screen layout or use 3D graphics – even though 2D would do fine. (It looks *so* much more sophisticated in three dimensions.) We can also write the code in a more clever or elegant way.

Giving programmers this latitude to define what any function or module should actually do serves to consume the safety time prior to the delivery date. We do not think we are dragging the task. Actually, we feel completely the opposite: There are *so many* more things to do in the specified time! After all, no one really expects early completion. But any additional code – bells and whistles – tends to complicate the truly necessary part of the code and may cause huge quality problems. Even one more day devoted to additions that are not truly required can result in weeks of searching for and fixing bugs thus causing even more bugs. The critical balance between the perceived time to write the code and the amount of time needed for effective debugging is shattered. Put another way, "The road to hell is paved with good intentions."

Even if the safety time added to the average estimate were relatively small, the impact of Parkinson's Law is such that most of the time the best we could ever hope for is to meet the original schedule. Jobs would almost never finish early. Unfortunately, it is usually never even this good. Over many tasks, the nature of the distribution of time for developing a module makes the full impact of Parkinson's Law quite deadly.

What does this distribution look like? It is not a nice, symmetrical normal distribution. Think about your own experiences. Sometimes working on a one-month job – a true average estimate, not the estimate given to the boss for public consumption – may take three months. This may not happen frequently, but sometimes it does occur. On the other hand the one-month job will never be finished in three days, or probably in a week either. But it might take two weeks instead of a month. What we find is that such a distribution of time is not symmetrical. This occurs more so in software development because of the tendency to add the niceties that complicate the code. The resulting distribution of task completion times is more likely to be skewed, as indicated in Figure 1.

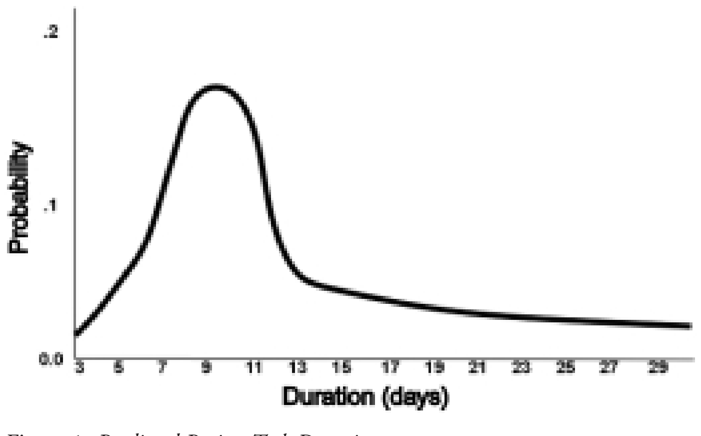


Figure 1. *Predicted Project Task Duration*

This figure shows a typical probability distribution over many projects of the number of days it takes to do similar tasks – or even similar whole projects. The area under the curve equals a probability of 100 percent. The median (a 50 percent chance of being less or equal to that number) is 10 days. The possible outcome may be any number from three on, but above 30 the probability is too small to be of practical value. Yet in some infrequent cases – but not absolutely rare – such a job might take 30 days, though most likely it would require 9-10 days.

Notice something very critical in this chart. If you would like to add enough safety time to your median estimate, you need to *double* your estimate of the expected time to be adequately protected.[1] And even doing so, you will not be perfectly protected. In such a case, estimating 20 days seems about right, though it might actually take longer than that. But it might be more reliable all the same, were Parkinson's Law not involved.

So, updating expectations does not prevent new disappointments. The expectations set the deadline, and the people on the job see that deadline and try to meet it – not beat or exceed it, just *meet* it. However, enough unexpected incidents and problems occur to delay completion even more. Toward the end of the project, pressure often builds to sacrifice some of the original features that were planned.

Dependencies Impact Outcomes

The example cited above dealt with a simple, straightforward, single module. In most software projects, such a module is just one part of the overall project. Between the various modules in the project, and between the tasks within each module, there are certain dependencies. Dependencies make a significant difference in the outcome, because their sensitivity to delays has a greater impact than not merely finishing early.

Figure 2 shows a typical project activity network in which three different modules, each done by a different programmer (or a team) are integrated into one larger module, resulting in a requirement to test and modify the original code. Integration cannot start until all the three inputs are done – in other words, after the last one finishes.

If all modules behave according to a skewed distribution

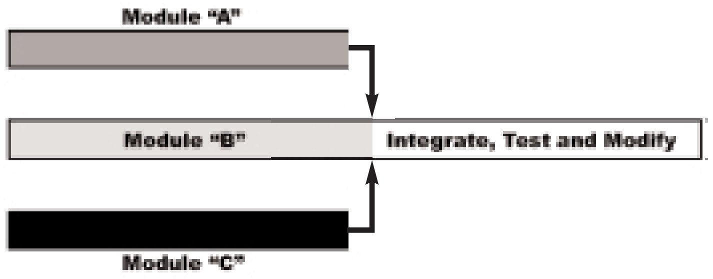


Figure 2. *Software Project Activity Network (Three Modules)*

(Figure 1), there is a fair chance that the overall effort will finish much later than the linear sum of the longest path (critical path). In other words the average of the sum is more than the sum of the averages. Being late in just one of the inputs is fully transferred to the last task, regardless of how early the other two are completed. (Variation accumulates at the end of the process.)

Combine the addition of safety time to every task, wasting that safety, adding unnecessary bells and whistles, compound the effect of all these with dependencies, and the result is a late project. Updating the original expected delivery date does not help. In fact, it creates the aforementioned vicious cycle. Expectations fail to be met on a regular basis - a huge problem.

Expressing the Problem as a Conflict

The Theory of Constraints suggests that a fully recognized problem that has not been overcome is the result of an unresolved conflict. A problem seems unresolvable because what appears to be the solution on one hand seems to intensify the problem from a different aspect.

Let us consider the vicious cycle described above. On one hand, safety time must be added to the various tasks to fairly estimate how long the whole project might take. On the other hand, we *should not* add safety to tasks because that safety is eventually wasted and we inevitably find ourselves facing worse delays.

By expressing the conflict clearly, we can surface some underlying assumptions that might be challenged. Figure 3 illustrates our conflict. The objective (A) is to manage our software company well. In order to manage our company well, we must plan software projects realistically (B). We must also complete our software projects as early as possible (C). In order to plan software projects realistically, we must plan for adequate safety time in each task (D). In order to complete our software projects as early as possible, we must *not* incorporate safety time for each task (D'). On one hand we must add safety time to each task; on the other hand we must not. But we cannot do both.

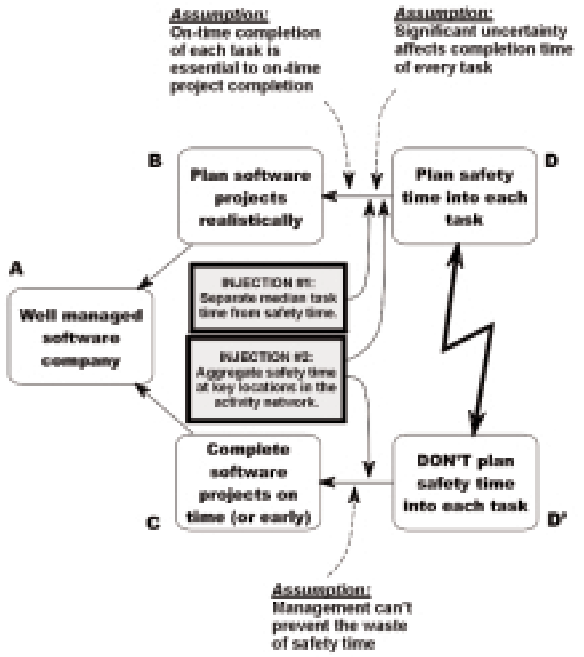

Assumption: On-time completion of each task is essential to on-time project completion
Assumption: Significant uncertainty affects completion time of every task
B
Plan software projects realistically
D
Plan safety time into each task
A
Well managed software company
INJECTION #1: Separate median task time from safety time.
INJECTION #2: Aggregate safety time at key locations in the activity network.
C
Complete software projects on time (or early)
DON'T plan safety time into each task
D'
Assumption: Management can't prevent the waste of safety time

Figure 3. The Project Planning Conflict

The typical management solution is to compromise between the two seemingly contradictory actions. In our software case, senior management typically does this by arguing with the estimator until agreement is achieved, reluctantly. For instance, you could argue with your chief programmer that instead of three months you want it to be ready in only two months, with only four total man-months invested. The chief programmer might resist, and you might eventually agree to 10 weeks.

But a compromise approach still leaves the problem in place: You still cannot realistically estimate when the project will be completed while time is wasted, delaying project completion without giving any real added value.

The importance of structuring the conflict as it appears in Figure 3 is that it allows us to articulate underlying (and probably unstated) assumptions. If any of these assumptions are invalid – or if they can be *made* invalid – then one or both of the conflicting prerequisites (D or D') can be replaced with some other alternative that satisfies the requirements of B and C, yet does not pose a conflict. Three key assumptions are shown in Figure 3.

One of these assumptions says that the completion time of each task is affected by much uncertainty. Can we challenge that assumption? Up to a point, we can. There are certainly ways to reduce uncertainty. For instance, knowing who is going to do the job and being familiar with his or her capabilities can help reduce the range of expected time for delivering the module. But can we reduce the amount of uncertainty low enough to dissolve the conflict? In many cases this is not possible. So this is probably a valid assumption.

What about the other two assumptions? Let us start with the one at the bottom (between C and D'): "Management can't prevent the waste of safety time." If we could create a situation where we can continually monitor the use of the safety time, we might be able to discourage people from wasting it. Can we do this?

First, we must clearly differentiate between the average estimation and any additional safety time. This would allow us to monitor the use of the safety time, which would reduce people's inclination to waste it. It would certainly allow management to ask whether the consumption of the safety time is caused by including features that were not required by the clients. However, this does not fully invalidate the assumption. It isn't practical to track *every* task and inquire why it used its safety time. Note that the median 50 percent estimation for every task means that half the time the task would use the safety time, even without any conscious intent to waste it. Ineffectiveness in tracing excessive use of safety time might still encourage people to behave according to Parkinson's Law.

The final assumption is even more important to invalidate. Even in the software world, most projects involve several different resources and task dependencies. The ultimate objective is that the overall project should not be delayed. On-time completion of individual tasks may be irrelevant. It might take longer to finish a particular task, but if we can still complete the whole project on time, why should the delay of an individual task matter? Moreover, there is no point including safety time for every task if that time is nearly always squandered anyway. Accumulating safety time for the project as a whole (at key points and at the end) is statistically superior; knowing that individual task safety time has been eliminated neutralizes people's tendency to waste time in return for no real value.

The main idea is to add safety time, but not to every individual task. We should use discrete safety time at critical locations within the project. Each task should be planned to consume the median estimate of the time required. When needed, the common safety time would be available to use. Making such safety time common to many tasks would make it much more difficult to waste.

The Nature of the Solution

To break the vicious cycle depicted in this conflict (Figure 3), we need to get rid of at least one invalid assumption. Invalidating multiple assumptions would be better. The last two assumptions cited above, very common today in managing software development, can be invalidated with the application of a new approach based on several new policies:

•Replace critical path concept with the notion of a "critical chain" as the constraint to earlier completion of any project.

The critical chain[2] is defined as the longest chain of operations linked by either finish-start connection or by resource availability. Resource availability is typically compromised by contention: We cannot complete some tasks concurrently, because they must be done by the same resource unit. The critical chain concept replaces the critical path concept, which usually ignores resource dependencies. The total length of the critical chain dictates the length of the project as a whole.

•Strip safety time from individual tasks. What should remain is the median. By doing so, we make it clear to every programmer that we expect them to concentrate only on the necessary features, avoid the student syndrome, and strive to complete their work early.

• Establish a project buffer This concept requires aggregating individual task safety time, while scheduling individual tasks based on average estimated completion time without safety time. The buffer is a specified length of time – usually considerably less than the sum of the individual task safety times – that is placed at the very end of the project, immediately after the last functional task. The actual completion time of the project is assumed to be at the end of the project buffer. This buffer compensates for the safety time eliminated from the individual tasks along the critical chain by re-inserting part of that safety time as a buffer for the whole chain. If we avoid the effects of Parkinson's Law and take advantage of the fact that some tasks would normally finish early, we can potentially realize a substantial time saving. Some of this aggregated safety time can be used to buffer other tasks; some of it can actually be eliminated, resulting in early delivery of the entire project. For planning purposes, the buffer looks like a task, with predicted start and finish time, but it has no resource assigned to it.

•Establish feeding buffers. The project buffer is the primary protection mechanism. It directly protects the project's due date from any delay along the critical chain. Delays in tasks that are *not* on the critical chain can delay the project completion *only* if they delay a critical chain task. To preclude this from happening, time buffers are inserted wherever a task not on the critical chain merges with a critical chain task. These are called "feeding buffers." They represent safety time to neutralize delays that otherwise might pass all the way through to the completion of the last task in the project.

•Manage the buffers. Once the critical chain schedule is constructed, with appropriate buffers inserted at key points, the project manager must then monitor the actual state of the buffers as the project is executed. At any point in time we can look at the current buffer consumption, meaning the accumulated delay along the chain that ends with the project buffer.

"Buffer penetration" constitutes the total time consumed, for all tasks to date, over and above expected average task time.

For example, let us assume that the project is a single sequence of 20 discrete tasks, each of approximately the same duration. Let us also assume that we have planned a 40-day project buffer. By the completion of the fifth task (roughly one-quarter of the way through the project), we have had one task finish on time; one finish two days early; and three tasks finish three, five and seven days late, respectively. Because one task finished two days early, the 40-day buffer is penetrated by 13 days (3+5+7-2). One quarter of the way through the project, we've used 32.5 percent of the project buffer. This is a disturbing indication because the protection (buffer) is being consumed at a faster pace than the progress of the project. At the moment, this might not be enough to warrant any actions to expedite, but it is enough of an indication to keep us watching the next few activities very closely. If any of those activities turn out to overrun their expected times (i.e., if more of the project buffer is consumed), we would probably consider options to accelerate progress.

The comforting aspect of this for the project manager is that the amount of the buffer consumed gives a clue to how much is left to protect the remaining tasks. It provides warning of a possible late delivery very early in the schedule, while there is still time to make small corrections versus crashing the project. The project buffer status, relative to where we are on the critical chain, gives the project manager invaluable information about the status of the project as a whole.

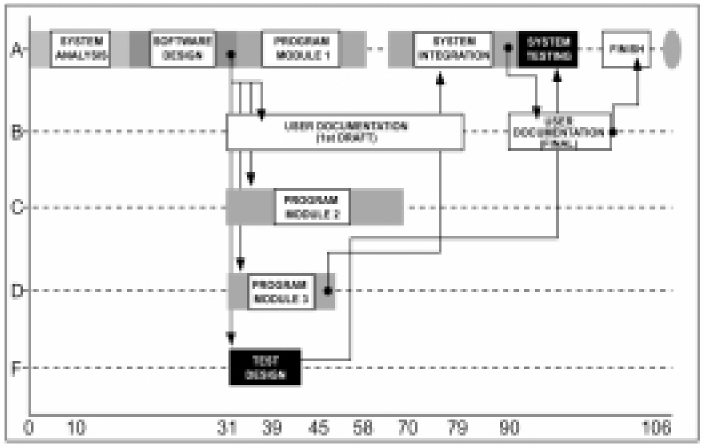
Other actions are required to apply this approach in multi-project environments, but that is beyond the scope of this article.

An Example

Figure 4 shows a basic structure for a simplified software project. The tasks outlined here are: system analysis, software design, program module 1, program module 2, program module 3, test design, user documentation (first draft), system integration, system testing, final user documentation, and finishing (packaging and shipping).

Task durations include individual safety time for every task providing a 90-percent probability of finishing on time or early. The estimated time to complete the whole project is 106 days.

Figure 4. *Three Module Software Development Project*



There is one problem in this plan. Only two programmers are assigned to the project. We have three modules to program that could theoretically be done in parallel, but because of the lack of one programmer, two of the modules will need to be completed consecutively. Also, system integration and system testing cannot begin until all the modules are completed and both programmers are available to support these activities.

The first step to achieving a realistic schedule is to resolve the resource contention. We can also identify the critical chain – the chain of tasks that dictate the length of the project. Figure 5 shows the original schedule replanned with the critical chain in mind. The figure highlights those tasks (heavy arrow). According to the adjusted plan, the project should be completed on day 118.

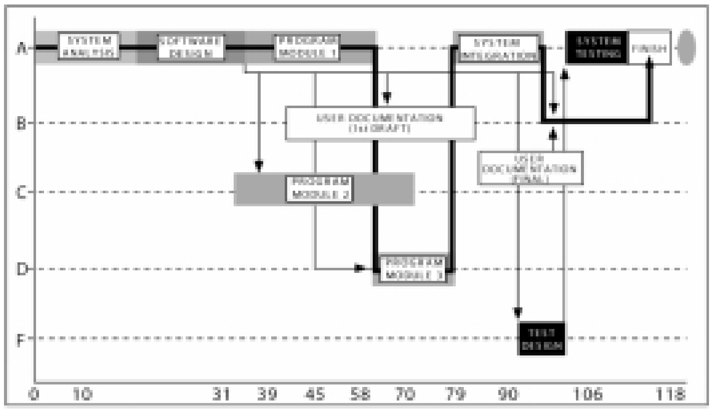


Figure 5. *Critical Chain Activity Network*

We have simulated this example to evaluate the impact of Parkinson's Law. For our computerized simulation, we have assumed that Parkinson's Law applies only 75 percent of the time. In other words when a task seems to have enough time, in 75 percent of the cases the performer will fill the excess time so that delivery to the next step in the process would happen exactly at the scheduled time. The other 25 percent of the time, when the task should take less than the planned duration, it actually does take less time. In Figure 5 we have simulated the case characterized 1,000 times. The project finished on time or early in only 34 percent of the runs (340 times out of 1,000). Remember that every task estimate was inflated so that in 90 percent of the cases it would finish on time or earlier, yet the project as a whole finished on schedule just one-third of the time. So adding all that individual task safety time did not do much good.

Following the concepts described earlier, the next step is to strip the safety time from each task and establish a project buffer and feeding buffers. That arrangement looks like Figure 6 (See page 24). The blocks with the beveled ends indicate the buffers.

Notice that all the tasks are now planned to take only about half of the time they were assigned previously. But we are not fooling ourselves – not all of them will meet those schedules. Actually, we expect that about half of the tasks will actually take longer than planned, so we place a buffer of 30 days at the very end of the project. While the last task appears to finish on day 59, we actually expect it to finish at/or before day 89. Notice also the feeding buffers at the right side of rows A, B, and C.

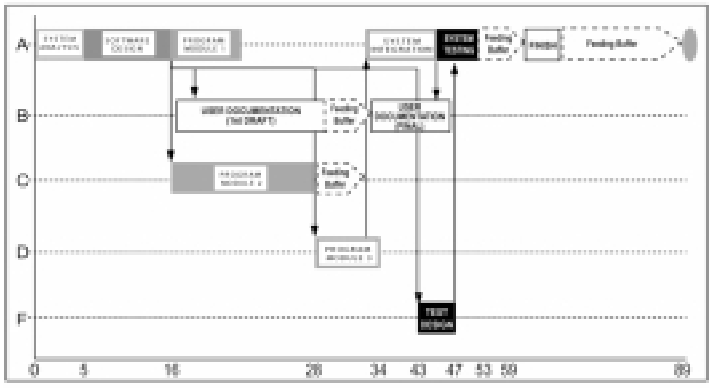


Figure 6. Project Buffers and Feeding Buffers

They protect critical chain tasks from delays induced by non-critical chain tasks.

Finishing a project in 89 days rather than 118 days seems very favorable. But we know that our original project plan was not too realistic. Do we really stand a chance of completing in only 89 days?

Even though we assume that half of the tasks would need more than the expected time to complete, the shorter planned time for all tasks helps to drastically reduce the impact of Parkinson's Law. After running the simulation in this configuration (with project and feeding buffers in place) 1,000 times, the project finished on time in 92 percent of the runs; 8 percent of the time it took more than 89 days. Buffers cannot offer 100-percent protection. But which would you rather have: 34 percent probability or 92 percent?

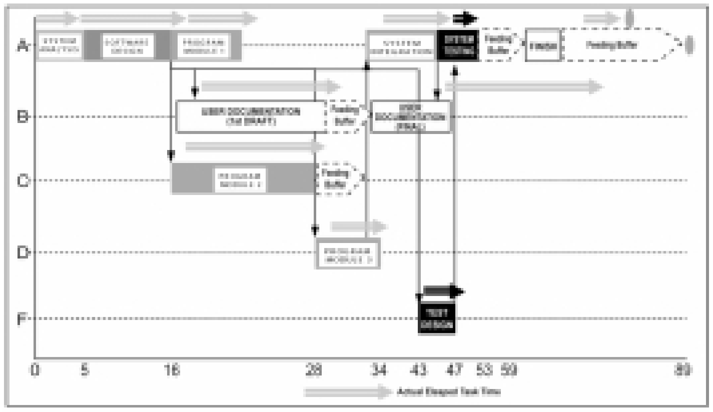


Figure 7. *Critical Chain Activity Network (Simulation Run #77)*

Figure 7 shows one discrete run out of the 1,000. The critical chain tasks are indicated by the heavy black arrow in Figure 5. The actual run is reflected in the hollow arrows above the "plan" bars. The dark stripes in each buffer symbol show how much of the buffer was consumed. Only one of the three feeding buffers was fully consumed. The consumption of the project buffer resulted from delays in the critical chain tasks, especially the next-to-last one. All in all, the project finished in 77 days in this run, well within our expectations.

Conclusion

Due to the vicious cycle inherent in software projects, updating our expectations only worsens a bad situation.Two central ideas to solve the vicious cycle emerge from verbalizing the problem as a conflict between two required actions: Add safety time to every task on one hand, but refrain from adding safety time to every task on the other.

The solution is developed from challenging two basic assumptions. First we cannot prevent wasting safety time; second we must strive to ensure that every individual task is finished at the planned time.

Accumulating the buffers where safety is truly needed enables us to set and achieve realistic expectations. Monitoring these buffers is crucial to successful project execution. Though not all tasks can be expected to go exactly as planned, effective buffer management assures a much higher probability that the overall project will be delivered on time.◆

Notes

1. There are three common values that represent the "center" of a statistical distribution: *mean, median* and *most likely.* For project task estimation, we consider the median an easier measure to estimate intuitively. We also believe it to be more relevant for decision making. The mean value is affected by extreme values that should not, in our opinion, be part of routine decision making. Note that while the three values are different for skewed distributions, for the human intuitively doing the estimating, the differences can be ignored for practical purposes.
2. For more information about Critical Chain Project Management see *Project Scheduling According to Dr. Goldratt* by Timothy K. Perkins in January 2001 CrossTalk. Also, visit the Goldratt Institute Web site at www.goldratt.com, the Goal Systems International Web site at www.goalsys;com, Goldratt's site at www.eligoldratt.com, and the Web site maintained "as a hobby" by David Shucavage at www.rogo.com/cac/index.htm

Did this article pique your interest?

Want to learn more about Critical Chain Scheduling, Theory of Constraints Thinking Processes, or the Theory of Constraints (TOC) in general? Then attend the Thirteenth Annual Software Technology Conference 2001 to be held April 29 – May 4 in Salt Lake City.

Dr. Eliyahu M. Goldratt, father of the Theory of Constraints, and author of several books on the Theory of Constraints will be the plenary speaker on May 3.

H. William Dettmer and Eli Schragenheim will address the Theory of Constraints Thinking Processes and Critical Chain Scheduling in the Software Technology Support Center Sponsored Track (Track 3) on May 3 following Dr. Goldratt's speech. In their presentation titled "Software Development Without the High Blood Pressure and Premature Grey Hair," Dettmer and Schragenhieim will describe how the principles of TOC can be applied to software development and project management.◆

" Megahertz – a very large car rental organization. "

FRANK J. ROMANO

About the Author

Eli Schragenheim has been associated with the Theory of Constraints for 16 years. He is active both as consultant and educator, and as software developer of simulations used for management education. He works closely with Dr. Eli Goldratt, the father of the Theory of Constraints, especially on issues of software development. Schragenheim has a master's degree from Tel Aviv University, Israel, and a bachelor's degree in mathematics from Hebrew University, Israel. He is the author of "Management Dilemmas: The Theory of Constraints Approach to Problem Identification and Solutions," co-author with H. William Dettmer of "Manufacturing at Warp Speed: Optimizing Supply Chain Financial Performance," and co-author with Dr. Eli Golidratt and Carol Ptak of "Necessary But Not Sufficient." Schragenheim can be reached at:

E-mail: elyakim@netvision.net.il

H. William Dettmer has more than 20 years of leadership and management experience in operations and logistics environments. He was an adjunct faculty member at the University of Southern California and has taught extensively at the graduate level on the Theory of Constraints, total quality management, project management, systems analysis, management control systems, and organizational behavior and development. Dettmer has a bachelor's degree from Rutgers University and a master's degree from the University of Southern California. He is the author of "Goldratt's Theory of Constraints: A Systems Approach to Continuous Improvement," and "Breaking the Constraints to World-Class Performance," Dettmer can be reached at:

Goal Systems International
E-mail: gsi@goalsys.com
Voice: (360) 565-8300

"If computers get too powerful, we can organize them into a committee -- that will do them in."

BRADLEY'S BROMIDE

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

C.A.R. HOARE

Coming Events

April 29-May 4

Software Technology Conference (STC 2001)
www.stc-online.org

May 1-3
2001 IEEE Radar Conference
www.atlaessgrss.org/radarcon2001

May 12-19
23rd International Conference on Software Engineering, and *International Workshop on Program Comprehension*
www.csr.uvic.ca/icse2001

May 8-14
Software Testing Analysis and Review
www.sqe.com/stareast

May 22
7th Annual Montgomery Area IT Partnership Day
web1.ssg.gunter.af.mil/partnership

June 5-7
AFCEA's 55th International Convention and Exposition
www.technet2001.org

June 11-13
E-Business Quality Applications Conference
qaiusa.com/conferences/june2001/index.html

June 18-22
ACM/IEEE Design Automation Conference
www.dac.com

June 25-27
2001 American Control Conference
www.ece.cmu.edu/~acc2001

Letter to the Editor

Dear Editor:

I just received the December 2000 issue of CROSSTALK. How could you leave out the Project Management Institute's (PMI®) Web site www.pmi.org ? You have shortchanged your readers on this one. PMI is one of the leading project management advocates in the United States. Since its founding in 1969, PMI has grown to be the organization of choice for project management professionalism. With nearly 70,000 members worldwide, PMI is the leading nonprofit professional association for project management. PMI establishes project management standards, provides seminars and educational programs, and professional certification that more and more organizations desire for their project leaders.

David Cottengim PMP,CSTE,CGFM,MBA
Business Manager, Project Integration
Defense Joint Accounting System
Defense Finance and Accounting Service