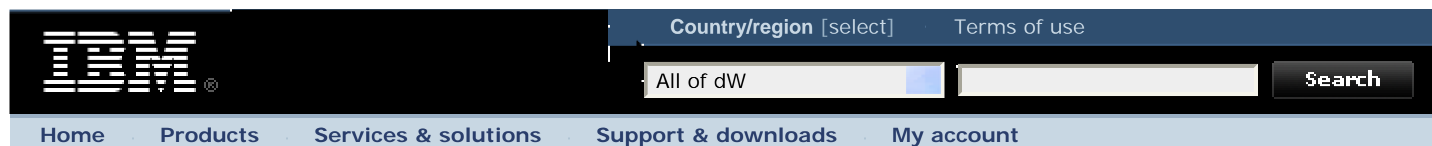

Twelve tips for realistic scheduling in a software development project

Level: Introductory

Laura Rose
QE Manager, IBM Rational software
15 Jul 2005

from The Rational Edge: Software development teams rely on carefully planned schedules. But beyond the use of basic scheduling tools, how can project managers juggle competing demands or allow time for unexpected events that threaten the best-laid plans? This article offers sophisticated scheduling techniques that stress prioritization, clarifying values, and comparing the relative worth of activities.

Contents:

Twelve tips for realistic scheduling
Summary
Notes
About the author
Rate this article

Subscriptions:

dW newsletters
The Rational Edge

Whether you're managing a software development project or coordinating your children's soccer and dance lesson times, schedules are helpful tools for orchestrating a sequence of events. Most schedules involve a start and end date, and include tasks, task duration, and dependencies between tasks. But no matter how well you plan for a series of events, unexpected events will compete for time and threaten deadlines. People you hadn't anticipated will step into your plans and begin to influence, control, and often complicate things. When we do not handle unexpected events and the interpersonal elements well, our schedules fall apart.

Good scheduling is very difficult, a combination of art and science. In this article, I will discuss realistic scheduling, which seeks to cover all the above types of events -- the planned, the possible, and the unimagined. There are some techniques that can help you keep your sanity, which go beyond the notes, checklists, milestones dates, and appointment books. My twelve tips stress prioritization, clarifying values, and comparing the relative worth of each activity. They combine the conventional checklists with preserving and enhancing relationships to accomplish the desired results.

Twelve tips for realistic scheduling

It's common to hear teams complain that "we don't have enough time." We often feel overwhelmed and helpless against an aggressive schedule, and when we work "against" something like time or schedules, there is much struggle and a large chance of failure. When we stop fighting against time[1] and work within our schedules and abilities, we increase our success rate.

For instance, the schedules themselves may not be aggressive; it's just that we've chosen to put too much into that allocated time. Intelligently deciding what to put into the schedule and how to work within those boundaries is within our control.

The following techniques will help restore that control. They illustrate what to prioritize, and how to clarify your vision by comparing the relative worth and value of each activity.

1. Don't allow "being busy" to derail your commitments.
2. Document a detail task list.
3. Identify critical paths and bottlenecks early.
4. Work on what will be of value to the customer.
5. Institute customer-sponsored releases.
6. Master effective and rapid decision making.

7. Rigorously institute reasonable forcing functions.
8. Execute effective meeting management.
9. Accept progressive refinement.
10. Reduce inventory wherever possible.
11. Don't be the source of the chaos.
12. Beware of heroics culture.

Note: I conclude some of my tips with boxed information I call an "additional tip." These boxes contain some related techniques to be used at your discretion. Depending upon the organizational makeup, some of these further suggestions may not always be appreciated.

Tip #1: Don't allow "being busy" to derail your commitments

Do you find that the busier you are, the more interruptions and requests you get? Many of us spend more time switching from task to task than actually accomplishing something. This can cause us to lose patience and actually avoid teammates, since we just want to be left alone!

I call this the paradox of "busy," because what you *want* to do and what you *need* to do are opposite.

In truth, the busier we are, the more patience we need, because everything takes longer.[2] The more in-demand we are, the more important it is to talk to people. The larger the load, the more we need to collaborate, delegate, and work in teams.

Bear in mind that your teammates aren't necessarily aware of your current list of commitments; they are only focused on the things they need. If you can take the time to explain your detail task list and deadlines, then they have a framework and background for your situation. If you plot out when in the schedule you can reasonably accommodate their requests, you will see that they are also reasonable and that your time line is fine with them. All this requires patience and understanding.

We often assume that a new request is about something urgent, important, and needed right away. That's not always the case. Discussing both your situation and their specific needs sets the groundwork to discuss priorities. The other advantage of talking to your teammates about the various tasks and timetables is that they may have already done something similar and can save you the effort. You may find some surprising synergy, collaboration, and networking opportunities.

It helps to interpret the events from different perspectives. Busy isn't synonymous with chaotic. ASAP doesn't mean drop everything, although some people assume "as soon as possible" means "as soon as *humanly* possible." Busy just means actively or fully engaged or occupied, and ASAP usually means as soon as *reasonably* possible. With some patience and communication, it is possible to control and structure a hectic schedule.

The more parallel tasks are required in a schedule, the more lead time and slack is needed. As you take on more tasks, expect more unexpected events associated with each task. Since the unexpected is a part of life in and out of the office, the efficient and realistic schedule anticipates them and their effect on the ideal sequence of accomplishments. Without well-structured safeguards, one incident (or added task) will cause a domino effect. Without well-planned buffers, we squander time switching from task to task without accomplishing much. With strategically placed cushions in your schedule, you position opportunities to accommodate the anomalies without impacting your overall timeline. You can now safely schedule unexpected requests at the next available break.

Sprints and buffers

One way to better ensure that you have a convenient stopping point for an unexpected emergency is to incorporate short sprints and buffers.

Consider this example: We have Tasks A and B, both of which we've estimated at eight days each, taking a total of 16 days in our schedule, as shown in the upper portion of Figure 1. We start off, but at the end of the third day we get an emergency task to accomplish. We spend the next day on the emergency, do some cleanup, and reset to get back to Task A. Because of some additional setup and log review to remember where we exactly left off, we have to re-evaluate how long this will now take. We re-estimate that it will take us about seven more days to complete Task A, because of the overhead of the interruption. After another two days, we get another emergency, and the churn begins again. At the end, we've actually spent over 12 days on the *actual* Task A (2 days spent for Task A + 1 day for the interrupting EmergencyA + 2 days spent to continue Task A + 1 day for the interrupting EmergencyB + 6 re-estimated days to complete Task A), as shown in the lower portion of Figure 1.

We're not only four days late in delivering Task A to those who need it, we're also holding up the people working and waiting on Task B.

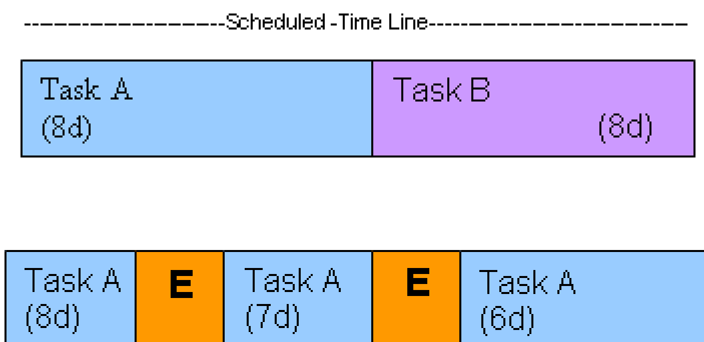

Figure 1: Emergency Sample A includes an initial 8 day estimate for Task A. After 2 days of work on Task A, an emergency occurs. After the emergency is dealt with, we re-estimate that it will now take 7 days to actually complete Task A (to cover interruption overhead). After another interruption is dealt with, we re-estimate that it will take 6 days to finally complete Task A.

A better way to approach this is to break the Task A and subsequent Task B into smaller, self-contained activities, or sprints, as shown in Figure 2. We schedule some buffer time between each sprint. The total schedule timeline for Task A has now increased from the original eight days to eleven. Let's see what happens with this same example.

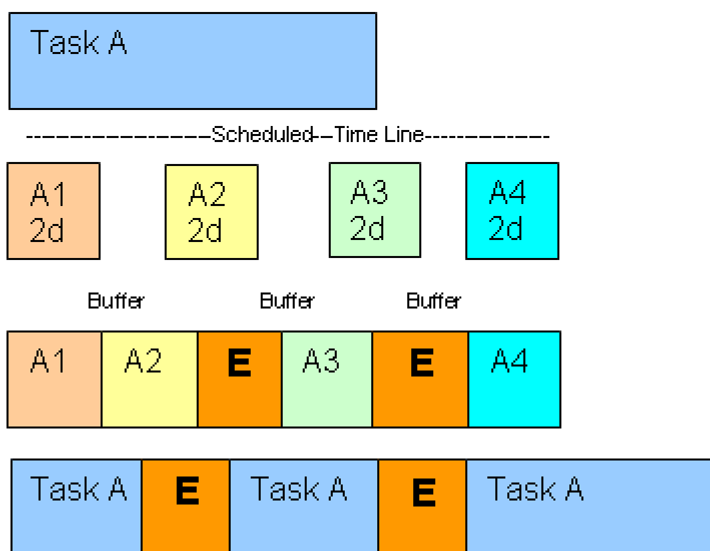

Figure 2: By incorporating sprints and buffers in our schedules, we can see that the actual timeline for the sprint strategy (A1, A2, A3, A4) accommodated both the emergencies and the schedule obligations. The Task A method took much longer and missed the schedule targets.

We start with Task A1. After two days, no emergencies have come up. We start on Task A2 without delay. At the end of the first day of A2 (third day into the exercise) the emergency arises, but because we can explain that we will be at a good stopping point at the end of tomorrow, the emergency is scheduled at that time.[3] When we've completed A2, we spend time on the emergency. Once again, we don't automatically stop what we are doing and absorb the overhead of switching tasks at this point. This example continues in this manner as another emergency appears after the start of A3.

Although the initial timeline based on a sprints-and-buffers method is longer than the original Task A, the expectations are more realistic and the results are closer to the actuals. People depending on the total Task A (A1, A2, A3, A4) are delivered those pieces on time and sometimes even ahead of schedule, and Task B items are not impacted.

If a request comes in that's more urgent and important than what you are currently doing (i.e., the requester cannot wait until your next available break in your schedule), go to your manager to make sure everyone is aware of the priorities, impact, and consequence regarding the schedule change. People will usually understand and accept this approach, because it is based on priorities that have consequences for the business, and it compares the relative worth and interdependency of each activity. By doing this prioritization and comparison with your manager, you clarify the value of each activity regarding the overall project schedule.

Time boxing

Another hazard that topples schedules is flattery. Now that you're known for your expertise in one area, you are the "go to" person in other similar but peripheral areas. It's difficult to say "no" to a coworker or another manager, especially when they preface the request with "It should only take you five minutes." That little voice in our heads says, "Sure, you can spare just five minutes for your friends and other managers." But five minutes often turn into half-a-day, and your manager is still waiting for your daily progress report, and

that was due last night.

A good technique is to time box these "extra requests."[4] Schedule a convenient five-minute meeting with your friend in which he or she explains the issue to the best of his or her ability. Use that information to estimate how long it might take you. Check your calendar or schedule to see if you can fit the appropriate time and explain, "I can spend nnn minutes on this at 10:00 on day XXX. If we haven't discovered or fixed the problem by then, we will need to re-evaluate the level of effort required, the priority of this, and get my manager involved." After that appropriate time limit has expired then ***STOP***[5] and re-evaluate. Time boxing is a great way to say "yes" without introducing chaos.

With the above strategies you are still a team player without derailing your other commitments. But these strategies depend upon a detail task list of what you are doing, by when, for whom and why (priority).

Additional tip: Many time-management books suggest the "Just Say No" technique. But sometimes it's wiser to say "yes" the right way, on your own terms and when it fits with your priorities and values.

Tip #2: Document a detail task list

Once again, the previous tip doesn't work without a detail task list with deadlines. You can't effectively and credibly explain to your partners everything that you need to accomplish if you don't know these things yourself. Detail task lists also help center you and keep you focused on the importance of each activity.

Effective time managers actually keep a list (either on their white boards or bulletin boards). When someone arrives with a new task, they have a visible template in which to start their negotiations.

Detail task lists are also important for level of effort determinations. When your manager asks you how long will it take to complete Task A, make sure you understand his or her definition of "complete." What if "complete" to your manager means feature design, design review with all stakeholders, code inspections against coding standards and system house standards, use of code profiling and status analysis tools to collect and report coding metrics on the features prior to code submission, unit testing, automation of those unit tests into the automated build verification tests, and complete functional verification testing with 95 percent pass rate? Fine. But if YOU interpret "complete" to mean simply "code the component and submit," you have a problem. And it's likely that the discrepancy won't be detected until much later.

When estimating your level of effort, start off with a detail task list template. This list template should include every major[6] activity that's needed to accomplish this general task. Depending upon the specific objective, some action items may not be required, but at least you have considered each of them (versus a forgotten task for which no time has been allotted). When we don't take the time to write the tasks down and we just review them in our heads, the resulting estimates are lower and less accurate than they are when we take the time to write down all the steps. My organization found that by making mental estimates and then doubling those mental estimates, we still required 30 percent more time to complete those tasks. In other words, even though we arbitrarily doubled our mental estimates, we had consistently under-estimated the actual required time. But when we wrote down all the steps suggested in a template, we improved our estimating accuracy; we were less likely to be surprised by a hidden or unknown task; and we had a documented reference on where all the unanticipated time actually went.

A detail task list also allows you and your manager to effectively rescope the project when you are in danger of missing a deadline. That is, if this is a risk, there are several alternatives to contain the problem: 1) move out the schedule, 2) intelligently add resources, 3) reduce the quality, 4) reduce the scope (remove the number of things we are going to do). If you don't have a detail list of what things you are planning to do, then it's difficult to intelligently remove activities to meet the deadline. By "intelligently" removing, I mean removing things that do not affect the quality of the code or their dependencies along the way.

Unambiguous activities also allow you to see if you're on track to complete on time. To illustrate this, consider a vaguely conceived Schedule A, as shown in Figure 3.

Task Name	Duration	Start	Finish
Comp1	10 days	Fri 3/11/05	Thu 3/24/05

Figure 3: Schedule A

In Schedule A, we show a high-level estimation of ten days to complete component Comp1. The schedule commences, and on day nine the developer announces that he has submitted his code. The team thinks we're on target.

Task Name	Duration	Start	Finish
Comp1	**9.5 days**	**Fri 3/11/05**	**Thu 3/24/05**
Design Inspection	0.5 days	Fri 3/11/05	Fri 3/11/05
code	5 days	Fri 3/11/05	Fri 3/18/05
Code Review/R2A	**1.5 days**	**Fri 3/18/05**	**Mon 3/21/05**
Run tool	0.5 days	Fri 3/18/05	Fri 3/18/05
Fix problems	1 day	Mon 3/21/05	Mon 3/21/05
Write/Automate Unit Test:	1 day	Fri 3/18/05	Mon 3/21/05
Run Unit Tests	0.5 days	Mon 3/21/05	Mon 3/21/05
Collect Test Results/Metri	0.5 days	Tue 3/22/05	Tue 3/22/05
Debug/Fix for 100% pass	2 days	Tue 3/22/05	Thu 3/24/05

Figure 4: Schedule B

In Schedule B, we see that only five days are really allocated for "coding." The rest of the time is allotted for inspections, testing, and bug fixing. So when the developer announces on day nine that "coding is complete," we're actually four days off schedule. Although these tasks weren't specifically visible on Schedule A, these activities that are not specified in Schedule A still need to be completed to achieve our iteration exit criteria.

So, agreeing ahead of time on the meaning of "complete" improves communication and allows us to better track our status.

Tip #3: Identify your critical paths and bottlenecks early

Risk management has always been highly publicized as an important project management tool. Yet, we don't really take the time to model or study our workflow to identify the risks, critical paths, or bottlenecks early on. Like defining our level of effort, we often rely on a quick and ad hoc approach, depending on our mental review of past experiences. Very rarely does the risk management exercise involve as much as a peer review of all the tasks and workflows of the project. If we don't understand all the tasks and timing, we don't realize the majority of the risks. If you don't realize the risks, you can't manage them.

A very effective and easy way to quickly and visibly identify risks in a project is to outline the process workflow in a visual flowchart. The workflow method can be used to analyze anything: Workflow analysis is effective on component dependencies, process step dependencies, even resource conflicts. Consider the diagram of a process in Figure 5, which outlines the different components under development. Specific colors represent the different human resources required to complete the activity, while the estimated duration is in ( ).

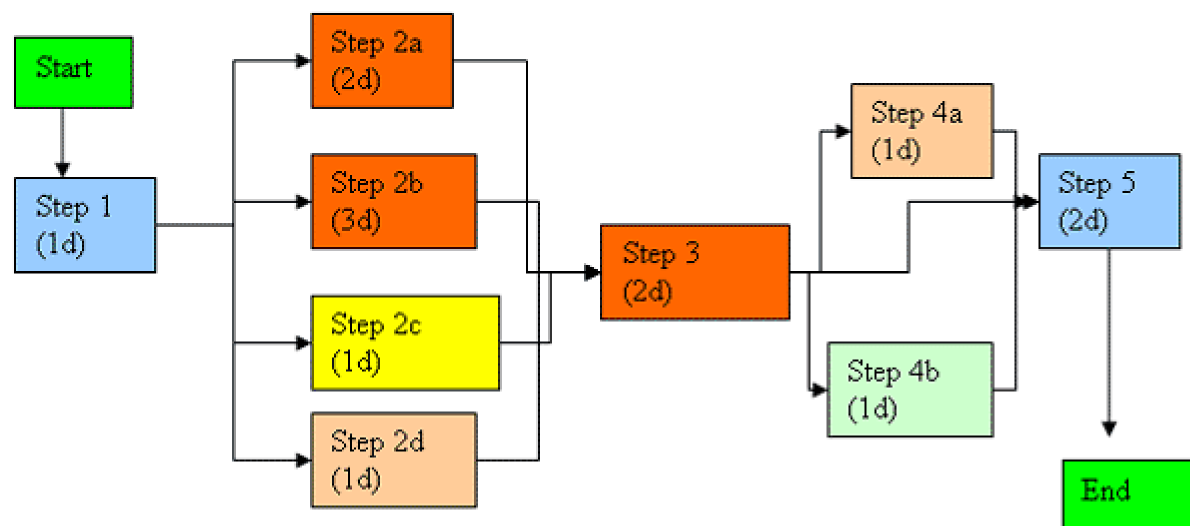

Figure 5: Process WorkFlow A: Mapping the workflow makes it clear that Step 3 and resource "orange" are the bottlenecks.

The diagram reveals some significant implications regarding the resource allocation proposed. Note that without resource considerations, the length of time through the critical path is nine days (longest time through the various sequential steps). But because we've used the same resource to do Step 2a, Step 2b, and Step 3, we need to add an additional two days. Why? Because

even though the steps are not dependent on each other, the resources performing those steps are. We are now up to 11 days.

Continuing with our analysis, if there are several input lines going into and out of a step or resource, you have visually identified an architectural or structural bottleneck. In this simple example, there are multiple items dependent on that Step 3; therefore, we have a real bottleneck not only in the resource but in the architecture. Unless Steps 2a, 2b, 2c, and 2d are all completed at the right time, Step 3 can't be done. If the resource on Step 3 is stuck on Step 2b, progress is completely blocked. No other steps can be started. This places the "orange" resource on the critical path. If we wait until the teams have started coding, and we actually hit the bottleneck, there is little we can do about it, because the orange resource is already deeply committed. He is the only one who knows the code in Step 2a and Step 2b, and he's probably coded additional assumptions into all three steps because he's the single owner of that code. He has complicated the interdependencies to make it faster to complete (once again, because he is the single owner). He is pretty much entangled, such that we can't efficiently add a resource to help him.

Mapping the workflow makes this problem visually clear and provides us a way to avoid it before the project even starts. Consider the improvements illustrated in Figure 6.

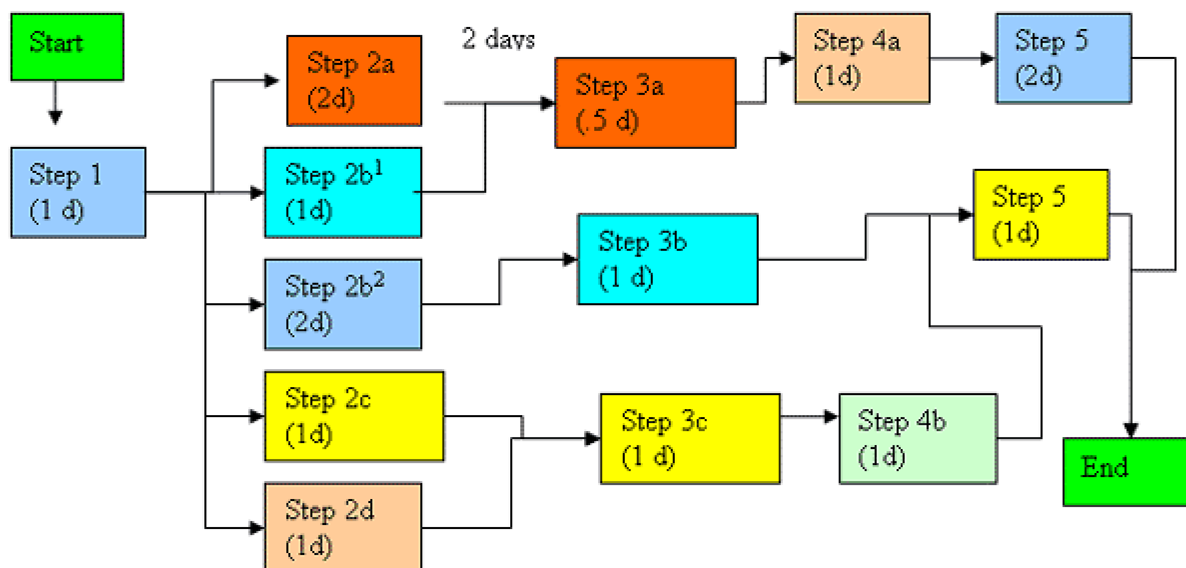

Figure 6: Workflow B: Once you've diagrammed your initial flow, you optimize to correct the risks and bottlenecks around the dependent components and resources.

Once you've diagrammed your initial flow, you optimize to correct the risks and bottlenecks around the dependent components and resources. In this example, although I've split my tasks into additional steps, my critical path is just seven days (shorter than my original scheme). I still feel that the orange resource -- Step 3a -- is a potential bottleneck, so I schedule a two-day buffer before the potential bottleneck. This allows all the sub-steps (Steps 2a, 2b, 2c, and 2d) to accumulate in a slight holding pattern. This stabilization period is a great way to incorporate mid-cycle validations, defect fixing, and quality audit checkpoints.[7] Although I have reduced the risk of bottlenecks and provided some additional lead time to the critical path, I haven't added any time to the overall project plan.

I also acknowledge that skill level of the resources is not 100 percent interchangeable. But the fact remains, if we haven't done this level of workflow analysis, we don't know that we can't redistribute, reorder, or restructure to take better use of the resources and skill level that we have. In this example, the orange resource was required to do Step 3 in Workflow A, only because there was a portion of Step 3 that needed an advanced level of multi-threaded Java design. When we take the time to split that piece away from the rest of the component, we find that several other resources could do the rest of Step 3. If we had otherwise identified that the orange resource had critical skills no one else had, we could reposition the orange resource into designing and architecting so that others could take his well-designed specifications and easily code from those artifacts.

Another advantage of this project management technique is that it avoids the over-padding (or sand-bagging) that we might experience when individuals pad for each of their tasks. We allow an additional time buffer associated with the critical path, and not the other tasks. The other non-critical path tasks already have an inherent buffer with the critical long pole.

If you do not map alternative workflows from the onset, you can easily get trapped. After you are already in the middle of the project and have the programmers entrenched in the code, many of these alternatives are closed to you. So the essence of this tip is that mapping or modeling your workflow early provides multiple options and alternatives.

Additional tip: Don't use your favorite scheduling tool (like Microsoft Project, Modeling tool, or Gantt charts) to do this initial workflow. These tools entice you to model in a vacuum. Use Post-it® notes and a large white board to make it easy for teams and groups to participate in the organization and remodeling. Multiple eyes see things that an individual will miss. This also helps build interpersonal relationships and group accountability at the same time.

Only after your team is happy with the optimization should you create the path in your favorite scheduling or modeling tool for

periodic and iterative reviews and updates.

Tip #4: Work on what will be of value to the customer

A large study made by James Johnson from the Standish Group (XP 2002) shows that 45 percent[8] of the features coded into most applications are never used (see Figure 7). It seems absurd to spend time on things no one will use, so where do all these features come from?

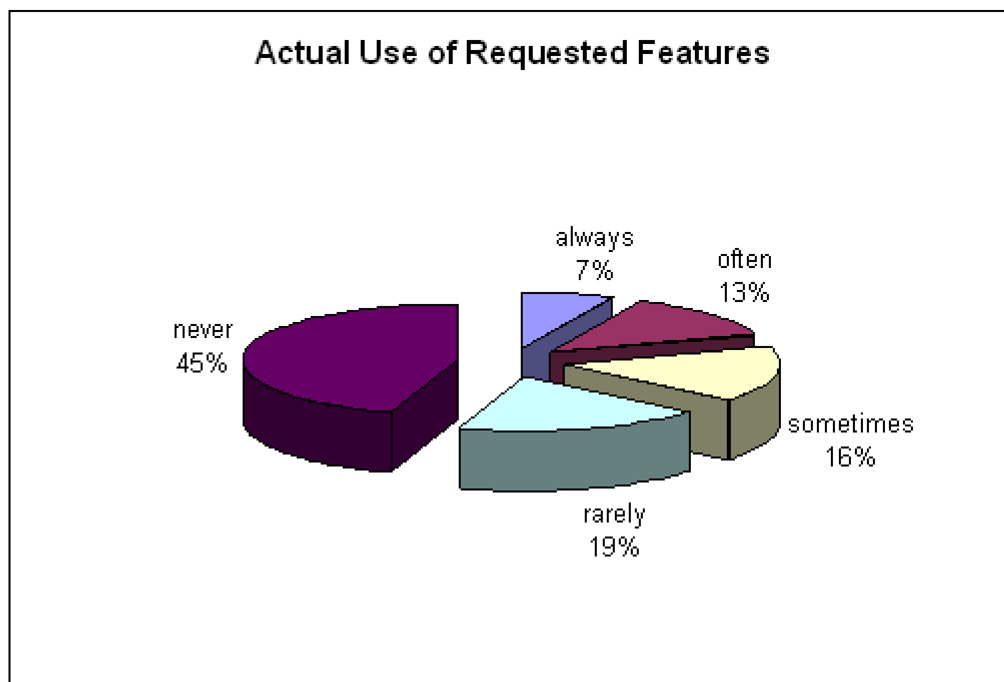

Figure 7: Usage of coded features, presented by Standish Group at the XP 2002 conference

Feature lists come from many places. Some come from our business analyst, who wants a visible representation for how the business is staying competitive. This sounds both reasonable and important. We want competitive differentiation. But the way we go about it may not be adding value for the customer. Consider, for instance, all the *Consumer Reports* analyses, or even your own analysis based on comparisons from technical sources. Typically those charts list more than 100 features compared across the various brands of a certain tool, with each item given a checkmark across the columns if that brand has that feature. At a superficial glance, it's assumed that the brand with the most checkmarks is the best solution.

What isn't as obvious is that, of those 100 features, a total of 64 percent are rarely or never used. But to stay seemingly competitive to the other brands, we incorporate superficial solutions to the unneeded 64 features, allowing us to "checkmark" those rows (making our product more complicated and difficult to use).

Rely on customer review

Granted, it may not be possible for many project managers to eliminate 64 percent of the required features a customer lists at the beginning of a project. Rather, the approach is to designate a customer who reviews each iteration's deliverable (requirements, prototypes, demos) throughout the development lifecycle. By doing this, your resulting product will have more of the features that actually get used, and you learn which features to spend your time on. Even if you have 100 features in your product, by constantly communicating and interfacing with your customers, you know which 36 features to focus on, and which 64 features to place lower in your priorities. So, even though you may not have influence on how the business folks list their required features, you at least know what's going to make your customer most satisfied by the end of the project.

The other side of this customer-value-added feature review is the removal of customer value features (more typically known as "scoping to fit the schedule"). Many times, it's the items that customers most value (like ease of use or user documentation) that are placed at a lower priority. Serviceability utilities (utilities that can help customers determine what's wrong and get up and running immediately) and troubleshooting database catalogs are also typically deferred. These aren't necessarily the "neat technical" items we enjoy working on, but are items that our customers value. By recognizing the possibility that some of the features we think less valuable may in fact be higher on the customer's list, we can discuss these items with the customer to improve overall satisfaction.

Ask your team

If you don't have a customer sponsor, then when reviewing requirements for a release (whether we're adding or removing features), have your team explicitly answer and sign off on the following:

- Will this feature really help the customer?

- Can you quantify the value added?
- What is the goal?
- Is it a realistic goal?
- How would you test it?

If you can effectively answer and quantify the above, you will be able to come up with an accurate level of effort and schedule that is realistic and valuable to your customer.

Beware of other causes of the long feature list

Another common contributor to software feature-bloat is our corporate system requirements and security functions. Many of our companies' system requirements come from a very general template that covers a huge range of applications and products. For instance, some networking and security concerns are non-existent in a stand-alone, isolated, single-use module and they are not required by the customer. But, in order to be sold with the company seal, the application needs to comply with certain company regulations (especially if it's intended to be exported to the international market).[9]

Another common contributor is the notion of "we, the development team." We love our work. We love creating neat and technically advanced gadgets. Sometimes we design something because it's fun to work on. Or we take a feature and over-design it to include everything we (not the customer) could "possibly imagine." It is fun, but we don't really have these creative extensions accounted for in our project schedule.

Tip #5: Institute customer-sponsored releases

Another way to increase the customer value to your feature list is to have a customer sponsor each release. This is a more formal approach to customer involvement than the customer review suggested in Tip #4.

In the past, our product managers collected various requirements from various inputs, and they would prioritize these according to competitive advantage, level of effort, and the needs of product delivery cycles. Today, product management needs to broaden the reach of this requirements process by selecting a customer sponsor who serves as the focus group for the functionality we are trying to accomplish. The customer sponsor needs to be involved in the requirements gathering, review, and design of this particular release. They evaluate the product at each deliverable iteration. Their test cases and use flows that illustrate how they accomplish their goals become our top "Must Pass" test cases. Their obstacles and defects become our "High Priority" defects. And their final evaluations become part of our "Go / No Go" decisions.

Having customer sponsor releases provide a tangible team goal (make this customer successful) that everyone can understand. Priorities and focus automatically follow. And at the end, we're not looking for a success story, because we've been creating it all along.

Tip #6: Master effective and rapid decision making

One of the hardest aspects of creating realistic schedules is making decisions.

Often the reason for a delayed decision is the fear of making the wrong one. Ironically, choosing not to make a decision is itself a decision to delay action, which doesn't bring you any closer to the right answer. Even the wrong decision brings you closer to a workable solution, because you can immediately deal with the consequences of a "made" decision. The quicker you make the decision, the quicker you can move forward and take action on both the positive and negative results.

I'm not suggesting reckless or snap decision making. I recommend moving fast on the reversible ones, and more slowly on the non-reversible, high-risk items. If the projected consequences of the wrong choice are minor, you should make the best decision based on what you know now and move on. In our iterative and constantly changing technical environments, we cannot know with 100 percent certainty that any decision is correct because it is implemented in the future. So make it and don't worry about it. Once the decision is made, stop discussing it. Execute, learn from any ensuing mistake, and just move on.

Of the items you cannot decide today, define the specific action items that will close the gap between where you are and where you need to be to make that decision. Make sure you have explicit owners and deadlines for each action (see Tip #7: Rigorously institute reasonable forcing functions). In my product group, we often attend meetings to exhaustively discuss a problem. After much struggle we are dismissed (often because we need to attend another meeting concerning another issue). Therefore, not only have we delayed the decision, we haven't put in place a chain of events to get us any closer (see Tips #7 and 8).[10]

> Additional tip: If no one takes ownership or commits to a deadline on one of the program's "critical" needs, then it's not really a problem. Remove it from the agenda, explicitly acknowledge that nothing will be done and remodel your program without that decision.

Tip #7: Rigorously institute reasonable forcing functions

As you can see from the previous tips, I've mentioned identifying explicit action items, tasks, owners, and deadline dates. This is essentially what "reasonable forcing functions" are all about. To assure success in any endeavor, you need not only a successful plan, but explicit, time-boxed task lists with responsible, accountable owners and acknowledged consequences.

You also need to define success criteria for various actions or programs. Steven Covey would describe this as "Starting with the End in Mind." If an activity is to succeed, we need to understand the definition of success.[11]

Using an earlier example: Does "successfully completing a feature" mean: a) the developer submitted the code for the component; or b) the stakeholders reviewed and approved the design, code review took place to assure more than one developer understands the mechanism, the component has undergone 100 percent unit and feature testing with 100 percent pass rate, and there are no major defects outstanding on this piece of code?

Either success criteria can work, as long as everyone agrees up front what "success" means.

Additional tip: Use the term "reasonable forcing functions" in every meeting you attend. Don't reserve it to remind the team to assign reasonable forcing functions, but also to recognize the team when they have set them. Repetition is the key to making reasonable forcing functions a habit.

Tip #8: Execute effective meeting management

As described in Tip #1, the busier we are, the more we find ourselves communicating and working in teams. This often leads to more meetings. Since we spend a majority of our communication time in meetings, they need to be purposeful and meaningful. There is much written on effective meeting management, which I don't intend to cover. We all understand that it's best to:

1. Have a purpose and success criteria for this meeting.
2. Have an agenda with timetables.
3. Stick to your timetable and meeting ground rules.
4. Do not adjourn before verifying that you've met your meeting's success criteria with a summary of your action items, owners, and deadlines (review Tip #7).

We all know the attributes of a successful meeting, but ensuring that these attributes describe the meetings we conduct is still a goal for most of us.

Additional tip: If you felt you just came out of an ineffective meeting, it's your fault. It doesn't matter whether you facilitated it or just participated in it. You are ultimately responsible for the way you spend your time. If you attended a meeting without understanding the purpose, agenda, or reasonable forcing success criteria, let that be the last you time you failed to get these important answers.

Tip #9: Accept progressive refinement

We've all heard the groan: "We have an aggressive schedule to meet." The fact is, it's not the "schedule" timeline that is aggressive; rather, it's what we choose to fit into it.[12] As I noted earlier, we cannot know with 100 percent certainty that any decision is correct because it's implemented in the future. By accepting that time changes our environment and sometimes even our purpose, we also need to accept progressive refinement. The concept is to define a very conservative stance in the early inception of the project schedule. Commit to only a conservative few (3-5) features and provide a "best effort" on the rest of the feature sets. And as time progresses, and we learn more, we refine our estimates and schedules accordingly. This allows us to change our energies from complaints about "an aggressive schedule" to a realistic but aggressive best-effort feature list.

Progressive refinement isn't delaying a decision on the schedule. We're actually making a series of decisions, consistently reviewing and refining throughout the project.

I am also not condoning excessive padding. Instead, you should incorporate a reasonably sized stabilization period between each of the project's critical paths and bottlenecks (see Tip #3: Identify critical paths and bottlenecks early), and institute a confidence percentage (e.g., "We are 70 percent certain that...") as part of your schedule assessment and review criteria.

Oftentimes we feel pressure to sign up to a schedule and commit too early in the development phase.[13] The schedule might look fine as far as you know today, so you can't exactly state that you disagree with it. It's just that you don't know what you're really committing to. Incorporating the confidence percentage strategy allows you to comfortably agree and also illustrates your areas of concern.

For instance, at the start of a "next generation version 1 product" -- your confidence on your original schedule and level of estimate might be only 40 percent. This means that you are only 40 percent confident in your estimates, and that your estimates may be as much as 60 percent off. Share with your teammates that confidence level, the action items, timeline, and plan you will use to increase you confidence level to 80 percent. Include what you don't know -- i.e., the activities, owners, and deadlines you will be using to close that gap -- in your schedule assessment.

Additional tip: Don't give a "confidence percentage" without a plan of action to close the gap between where you are and where you want to be in confidence.

Tip #10: Reduce inventory wherever possible

The retail industry -- e.g., clothing stores, grocers, and hardware shops -- understands the cost of keeping a large inventory. Their goal is to stock the shelves just enough to keep the merchandize moving. Large inventories bear uncertain costs, because the dress might go out of style, the food might spoil, the mechanize might not sell, and you're stuck with the entire lot.

In the software industry, the concept is similar, except our inventory consists of a backlog of defects, a list of features that haven't been delivered, and a series of unmade decisions. The wait cycle on these items are costly because technology and customers are always changing. Ways to reduce these wait times are:

- Use periodic and frequent stabilization periods to fix and reduce the backlog of defects.
- Plan and schedule so that the exit criteria for each of your stabilization periods is delivered to a specific customer (in some way like an alpha, beta, customer trip, residency program, etc.)
- Time-box and make decisions early as you go along. For instance, use the frequent exit criteria for each stability period to iron out and execute on your end-game deployment packaging decisions instead of waiting until the final release of your software application.
- Make your changes to the design document and test plans as you go. Backlogging your changes until you have time to update them causes confusion and time delays. Remember that the audience for these documents are other interested parties, not you, and if they need the document, they're going to access it. So you can't wait until you have more time. If the document is out-dated, your audience will be making inaccurate decisions. Constantly update and review the needed documents on a regular basis. Use your documentation reviews as forcing functions to the decision-making timetables provided in Figure 4.

Additional tip: If no one is responsible for keeping a design document accurate and consistently reviewed, don't write one in the first place. If it's already been written, but no one claims ownership, delete it.

Another way to reduce inventory is to declutter your email, your calendar, machine maintenance, and your to-do list from unimportant items that add no value to you or your customers.

Steven Covey, author of *The Seven Habits of Highly Effective People*, talks about the four categories of normal activities: Urgent and Important (Quadrant I), Important but Not Urgent (Quadrant II), Urgent but Not Important (Quadrant III), Not Urgent and Not Important (Quadrant IV). The danger of always operating in the seemingly Urgent Quadrants (I and III) is that you don't spend time on the non-urgent items, which can help you avoid the urgency later. Quadrant II items are important but not urgent. Therefore, we often postpone, defer, and delay those activities that will help eliminate the fires of tomorrow.

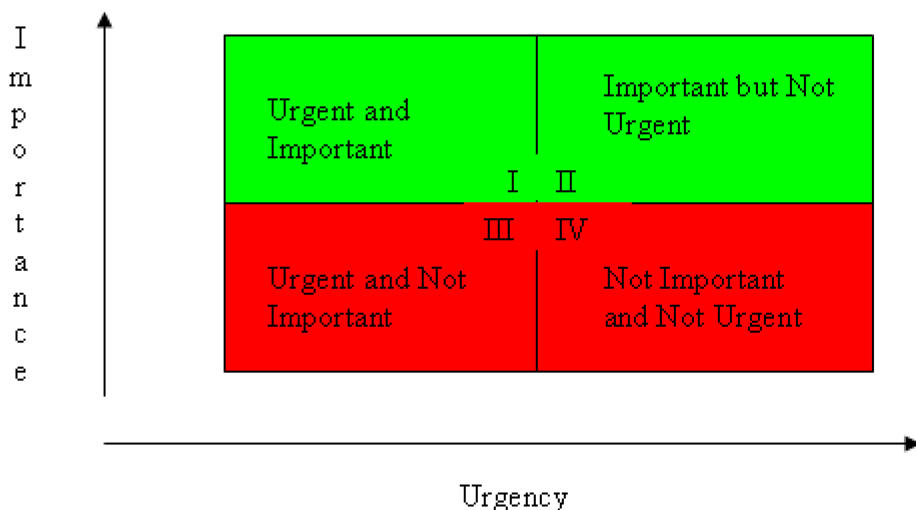

Figure 8: Covey's urgency quadrant chart

When you are first starting to declutter and reduce inventory, the only place to get time for Quadrant II is eliminating Quadrants III and IV activities. But before we can start eliminating Quadrants III and IV activities, we need to recognize them. Let's consider some telltale signs of seemingly urgent items that frequently trip software development organizations.

Defect control

I recommend that you be relentless in the elimination of the backlog of defects and feature enhancements. Constantly review your defect lists. If the defect or enhancement request isn't scheduled for the next two releases of your product line, then close it. The reason is that our industry and technology are constantly changing. If you haven't started work on those defects or features within the next two releases, several other things will have happened by then:

1. New technologies or methods have been introduced, which makes the original defect obsolete.

2. Current customers have either found a workaround or moved onto a different product.

3. If you couldn't get to it within the next two releases of the product, then it pales by comparison to the importance of resolving incoming defects.

Additional tip: If your Technical Support staff or Product Management staff disagrees with closing a particular defect without fixing it, then challenge them by placing this "urgent and important item" on the MUST FIX plate for this release and delay the release until it's completed.

Machine maintenance and security activities

Many companies have various urgent fire drills on machine security and maintenance issues. If fire drills are planned, they are at a level that is invisible to the practitioner and those who need to do the work. Often it's a "drop everything and get this done" predicament. And because these cost-of-doing-business moments aren't explicitly associated with any one project, program, or release, there's usually no place to schedule the effort and resource required. Everyone recognizes security is a serious issue, and a virus can stop production and be very costly for everyone. So, I'm not suggesting bypassing any of these important steps; just make them more efficient through good practices. Try the following:

1. Automate the update process.

2. Make the schedule well-publicized in advance (see Tip #1).

3. Consolidate and eliminate machines. If no one is willing to be responsible for its upkeep, dispose of the machine.

4. Find a sister group in your corporation that's doing similar things, and share machines and cross-train resources.

Don't waste your time making decisions that do not have to be made

About 80 percent of the decisions you make are inconsequential. Many of them, someone else can handle. Many of them don't matter in the long run. Before spending any time on making a decision, determine if it's an important one for you to make. If not, eliminate it or hand-off to someone else. In some cases, you can ask yourself: "Should I be the one holding up the team to make that decision?"

For example, if the team is spending an hour talking about how to implement a "temporary install" (the real install will be delivered next week), the team is wasting time. That particular decision doesn't matter. You should let the individual developer decide how to temporarily patch it and make sure he doesn't spend much time deciding. Eliminating decisions is appropriate at times as well, especially if the decision is not focusing attention on the right problem. If a decision is not important to anyone (i.e., no one wants to sign up to own it, with a deadline or actions), then eliminate the need for it. Here's a simple example: The question "What's for lunch?" is irrelevant if no one is hungry. Just skip lunch, thus eliminating the need for a decision.

Other backlogs: Reconsider your automated email lists

Automated email lists can be a great help or a hindrance to your individual time. If you are getting daily confirmation of builds or minutes of details that you aren't individually concerned about, it takes time to filter and review those items. Get off those lists, have those notices placed on a Website or central location (so you can review when you want), or automatically have them sent to separate folders and out of your inbox. Any email that stays in your inbox unread for more than one week, archive it out of your inbox.

Tip #11: Don't be the source of the chaos

One busy second-level manager I know sent several teams an email at 1:00 a.m. announcing a group meeting for 8:00 a.m. later that same morning. Since he had done this a few times in the past, it was time to talk to him about the messages he was sending and his expectations.

I pointed out that not many people actually arrive at the office by 8:00 a.m.; therefore, it would be unlikely that they have seen the "urgent" email announcement by 8:00. Because they would not have any time to prepare for the meeting, they would not be able to provide him with any valuable input or feedback he was seeking. The second-level manager's response was: "But my schedule is so chaotic that this was my first opportunity to call the meeting. I'll be in solid meetings for the next few days, so 8:00 a.m. was the only free spot in my calendar to get this done."

His desire to keep everyone well-informed and get their input was genuine, but he was not aware of the unintended message people were receiving. The teams and first-line managers were already working extra hours to achieve an aggressive schedule. They were trying to structure, filter, and control their own workload schedules around the previously planned meetings to still get their work done on time. A last-minute 1:00 a.m. meeting announcement for an 8:00 a.m. meeting produced a false sense of urgency and importance. There was no agenda sent, so teams assumed they were expected to drop everything and rearrange their work schedules to accommodate this no-notice meeting. This required an unplanned series of task-switching. And since there were no details or agenda items, they questioned how important could their attendance and input be if they aren't allowed the time to properly prepare for the meeting. At the same time, they wondered if there's an unwritten expectation that everyone needs to be working past 1:00 a.m. every night and at their desks by 8:00 a.m. every morning in order to catch these urgent invitations. The ill-planned meeting notice caused an unintended change in climate and morale.

These were not the messages that the second-line manager meant to convey. After reviewing the actual goal of the meeting, the second-line manager was able to identify an agenda and required participants, reducing the attendance list. Once the goal and agenda was outlined, he realized that he didn't have to be the one giving the presentation; therefore, this particular meeting wasn't

dependent upon his hectic calendar. Someone else was able to facilitate this meeting at a more appropriate time. Once the goal, agenda, and new attendance list were redistributed, teams were able to judge for themselves the urgency and importance of this particular meeting among the other things they were currently responsible for.

There will always be a handful of urgent and important issues that arise, causing our schedules to be chaotic for short periods. Even so, don't compound the situation by introducing more chaos into your schedule or your teams.

Tip #12: Beware of heroics culture

Another form of chaos erupts when heroics becomes the norm, not the exception. Organizations commonly expect additional hours and heroics to get their products out the door. While very common, it doesn't make it right. Depending upon heroics to get your product out the door is simply bad program management. If you built a realistic schedule in the first place, then heroics would not be required. Failing to properly identify the risks and delays is bad program management. Assigning single resources on parallel projects without the proper buffers and supporting cast is also bad program management. The Capability Maturity Model states it simply as a Level 1 mentality.

Save heroics for the additional, unexpected tasks that fall outside the regular daily needs of developing a product and satisfying a customer. Some examples are white papers on successful deployment of the products, hints and tips for the customer, presenting at conferences, etc.

Summary

Although this is not an exhaustive list of ways to improve scheduling, I'm confident that they will provide some ease to organizations that are growing and taking on increasing workload. I'd be happy to hear from you regarding any of these ideas, especially if any of them help you and your team.

Notes

[1] Since we can neither create nor destroy time, it's pointless to fight it.

[2] Idle hands can easily drop everything to accommodate. But we don't want idle hands. We're attracted to lean organizations proficient in doing more with less. Therefore, we need to create an empathic, supportive business to support "busyness."

[3] Although the emergency may not be able to wait until the entire Task A is completed (originally 8 days in length), it may be able to wait 1 day, enabling you to complete your sprint on Task A2.

[4] By "extra requests," I mean those requests that aren't already in your detail task list for today. Extra requests are the tasks that aren't necessarily requested by your manager or test lead. Even though these extra tasks are not explicitly approved, they are the grease that makes our interdependent teams and organizations function. It's the stuff that makes effective networking and harmony, and because we all know this, it's difficult to say "no."

[5] It's important to follow-through on the STOP and re-evaluate. We normally start with the mind-set of only spending 30 minutes on something, but we don't actually STOP after that time limit. Executing the STOP principle is the key to this technique.

[6] Deciding what is a "major activity" is a judgment call. We don't want to overwhelm with trivia; but to some degree of useful division of time and resources.

[7] Use these strategically placed stabilization points to reduce the backlog of defects, documentation reviews, and overall house cleaning (see Tip #10: Reduce Inventory Wherever Possible).

[8] See The Standish Group paper, *What Are Your Requirements?* Standish Group International, Inc., 2003, Standish Group (based on 2002 CHAOS Report).

[9] The reason system requirements are often a scheduling hiccup is because they aren't incorporated and calculated in the inception of the program schedules. When elaborating and putting our project schedule estimates together, we focus on only customer features and not with all the required supporting tasks. (See Tip #2: Document a Detail Task List).

[10] For more tips on rapid decision making, see http://www.liraz.com/tdecision.htm

[11] See Stephen R. Covey, *The Seven Habits Of Highly Effective People*, Simon and Schuster, New York: 1989.

[12] For instance, if we try to put ten pounds of stuff in a five-pound bag, is it the bag's fault?

[13] Many high-level project managers like to ceremonially go around the room getting verbal commitment on an early skeleton schedule. The intent is to get the individual practitioners to verbally commit and be held accountable for the work. Even though the intention is good, the timing may not be appropriate. We may not know and understand enough about the program to comfortably "commit" to the schedule at this point.

About the author

Laura Rose is the quality assurance manager responsible for automated performance test tools at IBM Rational. In addition to leading projects in both software programming and testing environments, she has thirteen years of programming experience and ten in test management. She has been a member of the American Society for Quality, the Triangle Quality Council, and the Triangle information Systems Quality Association, and has published and presented at various test and quality conferences. You can reach her at llrose@us.ibm.com.

Rate this article

This content was helpful to me:

Strongly disagree (1) Disagree (2) Neutral (3) Agree (4) Strongly agree (5)

Comments?

Submit feedback