**Chapter 4. Project Schedules**

**The project schedule is the core of the project plan**. It is used by the project manager to commit people to the project and show the organization how the work will be performed. Schedules are used to communicate final deadlines and, in some cases, to determine resource needs. They are also used as a kind of checklist to make sure that every task necessary is performed. If a task is on the schedule, the team is committed to doing it. In other words, the project schedule is the means by which the project manager brings the team and the project under control.

**Building the Project Schedule**

The *project schedule* is a calendar that links the tasks to be done with the resources that will do them. Before a project schedule can be created, the project manager must have a work breakdown structure (WBS), an effort estimate for each task, and a resource list with availability for each resource. If these are not yet available, it may be possible to create something that looks like a schedule, but it will essentially be a work of fiction. A project manager's time is better spent on working with the team to create a WBS and estimates (using a consensus-driven estimation method like Wideband Delphi—see [Chapter 3](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch03.html)) than on trying to build a project schedule without them. The reason for this is that a schedule itself is an estimate: each date in the schedule is estimated, and if those dates do not have the buy-in of the people who are going to do the work, the schedule will almost certainly be inaccurate.

There are many project scheduling software products that can do much of the tedious work of calculating the schedule automatically, and plenty of books and tutorials dedicated to teaching people how to use them. However, before a project manager can use these tools, he should understand the concepts behind the WBS, dependencies, resource allocation , critical paths, Gantt charts, and earned value. These are the real keys to planning a successful project.

**Allocate Resources to the Tasks**

The first step in building the project schedule is to identify the resources required to perform each of the tasks. A resource is any person, item, tool, or service that is needed by the project that is either scarce or has limited availability.

Many project managers use the terms "resource" and "person" interchangeably, but people are only one kind of resource. The project could include computer resources (like shared computer room, mainframe, or server time), locations (training rooms, temporary office space), services (like time from contractors, trainers, or a support team), and special equipment that will be temporarily acquired for the project. Most project schedules only plan for human resources—the other kinds of resources are listed in the resource list, which is part of the project plan (see [Chapter 2](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch02.html)).

One or more resources must be *allocated* to each task. To do this, the project manager must first assign the task to people who will perform it. For each task, the project manager must identify one or more people on the resource list capable of doing that task and assign it to them. Once a task is assigned, the team member who is performing it is not available for other tasks until the assigned task is completed. While some tasks can be assigned to any team member, most can be performed only by certain people. If those people are not available, the task must wait.

Some tasks may require more than one person to be assigned to them—for example, a programming task may require three programmers. In this case, the *effort* for the task should be divided among those resources. The project manager must keep in mind the difference between effort and *duration* . Duration is the amount of time that elapses between the time the task is started and the time it is completed, measured in hours (or days, weeks, etc.). It does not take into account the number of people performing the task. Effort is measured in *person-hours*(or person-days, person-weeks, etc.), and represents the total number of hours that each person spent working on the task. For example, if 3 people worked on a task together for a total of 2 working days, the duration required to complete the task was 16 hours (at 8 hours per day, with only 5 or 6 of those hours actually devoted to software engineering work). However, since each of the 3 people spent 16 hours on the task, the total effort required was 48 person-hours (keep in mind that some tasks are not divided evenly between resources; the total effort should reflect the actual time worked per resource).

It's possible to allocate one resource to two tasks simultaneously by assigning a percentage of the resource's time to each task. When the task stretches over several days, but the resource is needed only for part of each day or a few days of the task, that resource can be assigned part-time to the task. For example, a resource can be 50% allocated to two tasks, or 30% allocated to one task and 70% to another, etc.

In cases where more than one person is allocated to a task, the project manager must take *overhead* into account. Overhead is any effort that does not go to the core activities of the task but is still required in order for the people to perform it—a sort of "real world" cost of actually doing the work. For example, 2 people performing a task will require more effort than 1 person doing the same task: if the duration of a task is 12 days, it may require 7 days for 2 people to finish it, because they need an additional day to compare and integrate their work. The trade-off is that, while assigning two people to the task requires more effort, the task has a shorter duration.

One useful way to compensate for the extra overhead is to use the range that was generated by the Wideband Delphi estimate (which was for effort, not duration). The project manager can choose an effort estimate from the low end of the range if fewer resources are allocated to the task, whereas an estimate from the higher end can be used for a larger number of resources. The estimation team may have also made assumptions about the number of resources required to perform the task.

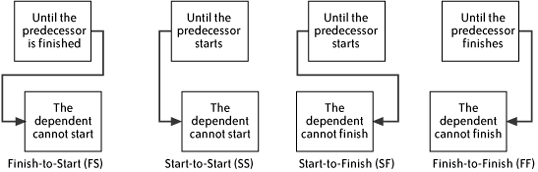
It is important to remember that resources are individual people, and no two people will take exactly the same amount of time to perform a task. The project manager should be familiar with the relative expertise of each team member. A senior programmer can often do a job in a fraction of the time that it would take a junior programmer to do the same work. However, the project manager should also pay attention to professional development. Senior team members are scarce; they can't be assigned every task, and some tasks are too difficult to assign to junior people at all. Assigning a junior programmer to work with a senior one will potentially make that junior programmer more valuable on the next project, but can cost more time from both people for training and overhead.

Resource allocation is often the most difficult and time-consuming part of effective project management, because it requires the project manager to know the team. There is no hard-and-fast rule for deciding who is allocated to which task. This is a decision that requires a great deal of attention to the skill sets of the people on the team and to their personal motivation. Some people prefer working on certain kinds of tasks, and are most productive when they are doing those.

Finally, there are two useful and well-known principles to remember when considering how people work on projects. First, Parkinson's Law (named for C. Northcote Parkinson, who first wrote about it in 1958) states, "Work expands so as to fill the time available for its completion." And second, as Fred Brooks pointed out his 1975 book *The Mythical Man-Month*, "Nine women cannot have a baby in one month"—in other words, some tasks can be done only by one person, no matter how critical that task is.

**Identify Dependencies**

Once resources are allocated, the next step in creating a project schedule is to identify *dependencies* between tasks. A task has a dependency if it involves an activity, resource, or work product that is subsequently required by another task. Dependencies come in many forms: a test plan can't be executed until a build of the software is delivered; code might depend on classes or modules built in earlier stages; a user interface can't be built until the design is reviewed. If Wideband Delphi is used to generate estimates, many of these dependencies will already be represented in the assumptions. It is the project manager's responsibility to work with everyone on the engineering team to identify these dependencies. The project manager should start by taking the WBS and adding dependency information to it: each task in the WBS is given a number, and the number of any task that it is dependent on should be listed next to it as a *predecessor*. [Figure 4-1](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch04.html#appliedprojectmgmt-CHP-4-FIG-1) shows the four ways in which one task can be dependent on another.



*Figure 4-1. Four different types of predecessor*

There are many reasons why one task may be dependent on another. The most common is the causal relationship: the dependent task relies on a work product generated by the predecessor. For example, the reviewers of a document cannot review it until it is completed, so a review task is dependent on the task that generates the document that will be reviewed. One task may also depend on another because they share the same resource: if there is only one programmer who has the knowledge to perform two different programming tasks, he cannot do them both at the same time; one must be dependent on the other. If there is no specific reason that one of those two programming tasks must be done before the other, then the project manager and programmer have discretion to perform them in either order.

The easiest way to maintain the resource allocations and dependencies is to use a project management software package. Most project management software allows the user to maintain a list of tasks and to associate resource information and predecessor information with each task. (It is possible to do this by hand, but very few project managers create schedules this way. Project management software is almost always used for this purpose.)

**Create the Schedule**

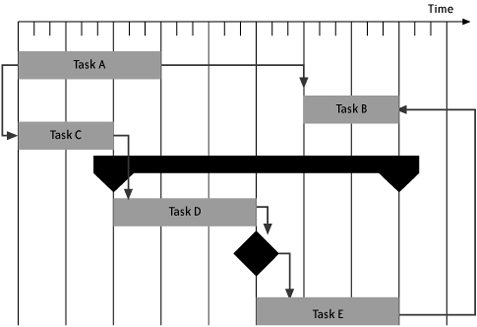
Once the resources and dependencies are assigned, the software will arrange the tasks to reflect the dependencies. The software also allows the project manager to enter effort and duration information for each task; with this information, it can calculate a final date and build the schedule.

The most common form for the schedule to take is a Gantt chart. This is a type of bar chart developed by Henry Laurence Gantt, an American engineer who was prominent during the first two decades of the 20th century. Over the past century, Gantt charts have been used on major civil engineering projects (including the Hoover Dam and the U.S. interstate highway system), and it is now the standard way to document software project schedules.

[Figure 4-2](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch04.html#appliedprojectmgmt-CHP-4-FIG-2) shows an example of a Gantt chart. Each task is represented by a bar, and the dependencies between tasks are represented by arrows. Each arrow either points to the start or the end of the task, depending on the type of predecessor (see [Figure 4-1](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch04.html#appliedprojectmgmt-CHP-4-FIG-1)). The black diamond between tasks D and E is a *milestone*, or a task with no duration. Milestones are used to show important events in the schedule. The black bar above tasks D and E is a summary task, which shows that these tasks are two subtasks of the same parent task. Summary tasks can contain other summary tasks as subtasks. For example, if the team used an extra Wideband Delphi session to decompose a task in the original WBS into subtasks, the original task should be shown as a summary task with the results of the second estimation session as its subtasks.

The Gantt chart in [Figure 4-2](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch04.html#appliedprojectmgmt-CHP-4-FIG-2) demonstrates the following predecessor types:

* Task A is a Finish-to-Start (FS) predecessor of Task B. Task B does not start until Task A is complete. For example, code cannot be reviewed until it is written, so the programming task would be a Finish-to-Start predecessor to the code review task.



*Figure 4-2. Gantt chart*

* Task A is a Start-to-Start (SS) predecessor of Task C. Both tasks start at the same time. If the start time for Task A were delayed, then the start time for Task C would move forward to match it. For example, a team of software testers might all be expected to start their test executions at the same time. One tester's task would depend on the build being delivered; that task would be a Start-to-Start predecessor for the other testers' tasks.
* Task C is a Finish-to-Start (FS) predecessor of Task D. Task D is a Finish-to-Start (FS) predecessor of a milestone, which in turn is a Finish-to-Start predecessor of Task E.
* Task E is a Finish-to-Finish (FF) predecessor of Task B. Note the delay before Task B starts—it does not start until its planned completion time will match up with Task E. This allows the resources required for Task B to be allocated to another task in the meantime. For example, a test plan can be started as soon as the requirements are complete, but it cannot be completed until after the design is done. So the test plan task would have the requirements task as a Finish-to-Start predecessor and the design task as a Finish-to-Finish predecessor.

**Reconcile the Schedule with the Organization's Needs**

Once all of the task durations and predecessors have been determined, the project management software can calculate an expected due date for the project. If this date does not fit with the needs of the organization or the project stakeholders, the project manager should first go back to the resource list to see if the tasks can be reallocated more efficiently. One way to do this is to look for large gaps in the schedule; sometimes a small shift or swap in resources can close those gaps.

Another way to deal with a schedule that runs past a non-negotiable due date is to add or rearrange resources (if available). This is one reason it is important to set up different kinds of predecessors. By making a longer task a Finish-to-Start predecessor of a shorter task, for example, a gap in the allocation level for resources might emerge in front of the shorter one. That gap could be moved or filled with another task.

Sometimes there are technical solutions that can help reduce the schedule. It may be possible to return to the assumptions generated during the estimation session. There may be an implementation approach that can be revisited. For example, the team may have assumed that a user interface would be built for a piece of the software, when it could instead be built to run from the command line. Or it might have assumed it would have to build a feature to address a need, when this component could instead be purchased off the shelf.

As a last resort, however, the project can be released in several phases. This requires the project manager to revisit the project's scope, which will have to be adjusted to allow for a phased release: some features will have to be broken into phases, while others may be cut out entirely. This requires that the project manager revise the vision and scope document and go through its review process all over again (see [Chapter 2](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch02.html)). While this seems severe, it is often the only way to deal with an otherwise unworkable situation in which the organization expects the team to complete a project faster than it is possible for the team to build it.

**Add Review Meetings to the Schedule**

If a schedule is written down, put in a folder, and never looked at again, the project plan may just as well have never been made. There is no reason to plan a project if that plan is never consulted again, nor corrected when it proves to be incorrect. There is no project plan that perfectly estimates every task; the only way the team members can improve their planning skills is by learning from their mistakes. The way to ensure that this happens is to add regular review meetings to the schedule.

*Progress reviews* should be held regularly, both to keep track of whether the schedule is accurate and to plan action if the project goes off course. If the team is already holding weekly, biweekly, or monthly status meetings (see below), then these can also function as progress reviews (as long as the specific details of the schedule are discussed at every meeting). To make sure a status meeting functions as an effective schedule review, the project manager must make sure that the agenda at every meeting includes a discussion of whether the project is still on track. The project schedule serves as the agenda for this part of the meeting. The project manager should go through each task that is currently in progress and work with the team to determine the status of the task.

During the review, if the team discovers that a task is going to be late, the project manager must find a way to deal with it in the schedule. In some cases, a late task may cause other tasks to be delayed, though sometimes the delay can be absorbed in the schedule. For this reason, it is important for the stakeholders to be present: if there are major problems, they will know immediately and can help resolve them. The result of this meeting will usually be an adjusted project schedule. However, sometimes delays will cause serious problems that cannot be dealt with in the schedule. If a delay means that an unmovable deadline will be missed, the team will either have to adjust the schedule to put in overtime, or it will have to go back to the vision and scope document and scale back the scope of the project.

*Milestone reviews* are meetings that the project manager schedules in advance to coincide with project events. The most common way for project managers to handle milestone reviews is to schedule them to occur after the last task in a project phase (such as the end of design or programming). Project schedules are usually broken down into distinct *phases* which correspond to the major software engineering activities: scope, requirements, design, development, and testing. Each of these phases is usually represented on the Gantt chart as a summary task, with a milestone as the final subtask to mark the end of the phase.

Once again, the project manager should make sure that the representatives from the engineering team and stakeholders attend all of those meetings. The difference between a milestone review and a progress one is that the project manager writes up a report after the milestone review. This report should list any schedule delays or changes, any modifications to the scope, and any serious issues that came up since the last milestone review meeting. These reports should be stored with the project plan.

The schedule should always be updated by the project manager. However, the information that is used to update the project schedule should be agreed upon at status meetings. After every status meeting, the schedule should be updated with any changes that were agreed upon in the meeting. If a team member discovers that a problem has occurred between status meetings that could cause a delay, the project manager should be notified immediately. If the project manager feels that the problem is serious enough that it cannot wait until the next scheduled status meeting, an impromptu meeting should be called, and the team should talk about the impact to individual tasks so that the schedule can be kept up to date.

A final milestone should be added to the schedule for a postmortem meeting run by the QA lead (see [Chapter 8](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch08.html)).

**Optimize the Schedule**

Many times, the project manager has options in how the schedule is arranged. There is often flexibility in the order in which the tasks may be performed, or to whom they may be assigned. Most schedules end up with several sequences of interrelated tasks: a requirements document may have a string of elicitation, documentation, and verification tasks; software must be designed, coded, and reviewed; test plans must be written, reviewed, executed, repaired, and regressed.

In many schedules, there is some *slack* in these sequences. In a sequence of tasks, slack is the amount of time that any of the tasks can be delayed without causing the due date of the final task in the sequence to be delayed as well. A tight schedule has very little slack; a delay in any task will cause a delay in the due date. For example, a task may depend on a work product, but the person who will perform that task may not be available until three days after the work product is scheduled to be complete; this creates a three-day gap in the schedule. It may be possible to rearrange the tasks in order to reduce that slack—for example, the task that is occupying the resource could be moved to a point later or earlier in the project, or assigned to another resource. These decisions can make an enormous difference in the final due date.

It is important to keep in mind that, while there may appear to be slack in the schedule, it may simply be that all of the resources in the project are allocated to another task; there just happens to be some time between a task and its predecessor. Many project management software packages have a feature that summarizes the allocation of each resource per day in the schedule, which can be used to check for slack periods in which resources are unallocated. This is also helpful for ensuring that no resource is *over-allocated*, or more than 100% allocated to multiple tasks simultaneously. If any resource is over-allocated, it means that there is a dependency between two tasks that was not discovered. When this happens, the schedule is guaranteed to be inaccurate.

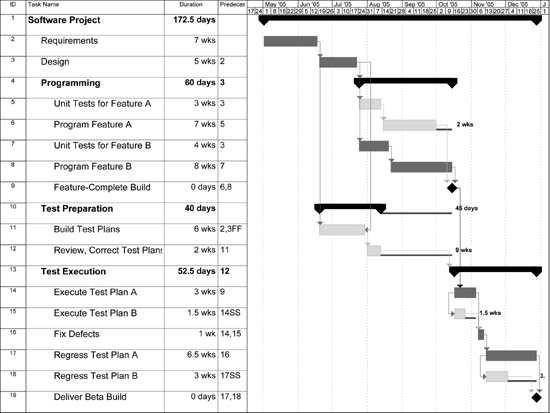
Some project managers fall into the trap of using slack in the schedule as a way to mitigate risk. They think that if there is extra space between two tasks, then the second task will have some protection, in case the first task is late. This is usually a mistake. Most project managers who try to do this find that Parkinson's Law kicks in—the work in the first task expands to fill up the slack. Instead of using slack to mitigate risk, the definition of the task should be expanded to include an activity that may mitigate the risk (see [Chapter 2](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch02.html) about risk mitigation). This new task will take longer, and the schedule should be updated to reflect that. That said, if there is unavoidable slack in the schedule, it means that the schedule can tolerate some delay without affecting the due date of the project. But it is very important not to rely on this; instead, it's often better to plan other activities (such as training) for the resources with slack time.

One important tool for optimizing the schedule is the *critical path* . The critical path is the sequence of tasks that represents the minimum time required to complete the project. It is the sequence that, if delayed, will delay the schedule. The last task on the critical path is always the last task in the schedule—when the critical path is completed, the project is done. Every project schedule has at least one critical path: most have exactly one, but some may have two or more critical paths that complete simultaneously at the end of the project. There is never slack in the critical path. When the schedule is most optimal, the critical path starts near the beginning of the project and the total effort expended on each day of the project is relatively steady.

It is very important to monitor the critical path closely. If a task that is on the critical path is late, the project will be delayed. On the other hand, some tasks that are not on the critical path can suffer delays without jeopardizing the expected due date for the project. Some project management software packages highlight the critical path on the Gantt chart—this is an especially useful feature that allows the project manager to optimize the chart visually.

[Figure 4-3](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch04.html#appliedprojectmgmt-CHP-4-FIG-3) shows an example of how a critical path would be displayed in a project schedule. The darker tasks represent the critical path; the lighter tasks are off of the critical path. (This figure was created with Microsoft Project 2003.)

In this example, the test preparation tasks are not on the critical path. This means that if there is a delay in building or reviewing the test plans, then the project due date will not change unless that delay is long enough to put those tasks back on the critical path. In this case, that would require nine weeks. (This schedule shows the amount of play in the schedule by depicting slack in the schedule as thin bars to the right of each task. The "1.5wks" label next to the "Execute Test Plan B" task shows that task would have to be delayed by 1.5 weeks to impact the due date.)



*Figure 4-3. Example of a project schedule showing a critical path*

This helps the project manager make decisions about the project. For example, if an extra person becomes available for the project, the project manager can assign him to tasks on the critical path, since assigning him as a resource on a noncritical task won't have any noticeable effect on the due date. It also helps the project manager understand the impact of scope creep or changing requirements, by showing whether those changes will make a difference in the time to deliver.

**Don't Abuse Buffers**

Many project managers commonly add *buffers* to their schedules. A buffer is a task added to the schedule with no specific purpose except to account for unexpected delays. This practice involves either adding extra tasks or padding existing tasks at strategic points in the schedule where overruns are "expected."

There are times when buffers are useful. For example, on a year-long project, if every programmer has two weeks of vacation and on average takes one week of sick days, then the project is guaranteed to lose three person-weeks of effort over the course of the year. The project manager could sprinkle three person-weeks of buffers at strategic points in the schedule in order to accommodate for this known loss. The use of buffers in this case is appropriate because the size of the loss is known.

However, there are many times when buffers are abused. The idea that overruns are expected means that there is an implicit assumption that the estimate is incorrect. If this is the case, why not increase the estimate to include the buffer? The danger in the buffer is that it lulls people—especially senior managers under time pressure—into feeling that there is lots of extra "play" built into the schedule. They have an expectation that the scope can be changed without any impact to the schedule, because the buffers will absorb the change.

This is especially bad for the team because each person feels that she can ignore "small" errors in her own estimates. For example, a programmer may find that she underestimated her programming task because the technical solution was more difficult than she originally anticipated. If there is a buffer, she may feel comfortable not reporting that mistake because she assumes that it will be absorbed. Unfortunately, if the designer and tester both made similar estimation mistakes and also failed to report them, all three of them are now counting on using the same buffer: there will be a large overrun. Since nobody reported any individual overruns, the project manager will not know about the problem until after the project has already slipped; this will limit his options much more than if he had known about the estimation problems as soon as they were discovered.

The bottom line is that when buffers are added to the schedule, Parkinson's Law will kick in and the work will expand to fill the time allocated to it. Luckily, the project manager already has a tool to help him plan for the unknown or unexpected: he can work with the team to build a risk plan (see [Chapter 2](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch02.html)). By brainstorming risks and adding mitigation tasks to the project schedule, he can avoid some of the risks. And for the ones that cannot be avoided, there will already be a plan in place to deal with most of them.

Adding a risk plan to a schedule that does not already include risk mitigation tasks requires that the schedule be updated and the project plan reinspected; however, this is not a bad thing. Updating the schedule guarantees that the schedule does not contain any "white lies," and inspecting it effectively communicates the team's true estimate of the work to everyone who will be impacted by it. If the risk plan is thorough, the stakeholders will not be blindsided by the potential problems, because they will see that the team anticipated these possibilities and has a plan to deal with them. This will remove a lot of the pressure usually associated with project overruns.

**Track the Performance of the Project**

After the schedule is completed and optimized, it is ready for review. The schedule should be inspected by the project team, either on its own or as part of the project plan. (See [Chapter 2](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch02.html) for the inspection checklist, which is part of the checklist for the project plan. See [Chapter 5](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch05.html) for more on inspections.) It is important that the people on the project team who will do the work all agree that it represents a realistic plan for completing the project.

A copy of the version of the schedule that has been approved should be set aside and used as the *baseline*. A baseline is a fixed schedule that represents the standard that is used to measure the performance of the project. Every time a change to the scope of the project is approved (see [Chapter 6](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06.html) for information on change control), the schedule should be adjusted and a new revision of the baseline should be used instead. Many project management software packages have a feature that allows the project manager to maintain a baseline schedule and track revisions to it.

This means that there are two versions of the schedule. One is a fixed baseline version that is kept as a reference, and the other is an *actual* version of the schedule that is updated to reflect what actually happened over the course of the project. The baseline schedule never changes over the course of the project. Every time a task is delayed or changed, the actual schedule should be updated to reflect that. Each schedule change should be stored as a separate revision, which shows a snapshot of what the schedule looked like at any time in its history.

When the due date for the actual schedule is later than that of the baseline, the project has *slipped*. However, the schedule slip does not tell the whole story. A schedule might slip because the team is waiting for a single person to complete a delayed task. It might also slip because there is a general tendency to underestimate the effort required to perform all tasks. It's important for the project manager to track down the source of each slip in order to help improve the team's estimates in the future, and to work with senior management to determine whether action needs to be taken.

The most common way to understand the nature of the schedule slip is to calculate the *variance* . This is a calculation that dates back to the early 1900s, when it was used by factories to track the difference between their budgeted costs and their actual costs. Variance is now the core of a project management system called *earned value management*, which tracks the project by considering effort "earned" against a budget only after it has actually been performed. For software projects, the variance is a measurement in person-hours (or person-days, person-years, etc.) that shows the difference between the effort planned to date on the baseline and the effort completed on the actual schedule.

The *budgeted cost for work scheduled* (BCWS) is the estimated effort of the actual tasks that appear on the schedule to date. The *actual cost of work performed* (ACWP ) is the effort spent on the tasks in the schedule that have actually been completed by the development team members. The data required to calculate this information can be gathered by comparing the effort represented in the baseline schedule (to find the budgeted cost) with the effort in the actual schedule. (Many project management software packages will calculate these numbers automatically.) The variance is the difference between these two numbers (BCWS - ACWP). (BCWS and ACWP are standard acronyms used in most earned value calculations.) If the variance is positive, then the project cost fewer person-hours than were budgeted; if it is negative, then the project overran the budget.

The variance is useful in helping a project manager determine whether a schedule slip is due to an isolated incident or a systematic problem. If there is a large schedule slip but the variance is small (if, for example, it is much smaller than the length of the delay), then the project manager should look for one or two tasks that were delayed. On the other hand, if the variance is large, there may be a problem with the way the team estimated the tasks. The project manager can spend additional time with the team to work on the estimates—for example, extra Wideband Delphi estimation sessions can be performed to decompose the tasks that are more susceptible to delays. This can be used both for future projects and for tasks that have not yet been performed in the current project.

For example, consider a software project where the requirements phase of a project is scheduled to last for six weeks. Over the course of those 6 weeks, 2 software engineers are scheduled to work on the project at 75% allocation, for a total of 360 effort-hours (75% of a 40-hour week is 30 person-hours per week per engineer, multiplied by 6 weeks). At the phase-end review, the project manager finds that the phase actually took eight weeks, and the software engineers put a total of 390 person-hours into the requirements phase. The earned value for this phase is 360 − 390 = −30 person-hours. In other words, the project overran the budget by 30 person-hours. The project manager can even calculate the actual cost (in dollars) of the overrun by multiplying those 30 person-hours by the average cost per hour that the organization pays their software engineers (plus the average overhead cost).

Another way earned value can be used is by generating the *cost performance index* (CPI) for the project. CPI is calculated by dividing BCWS / ACWP and multiplying by 100 to express it as a percentage. A CPI of 100% means that the estimated cost was exactly right and the project came in exactly on budget. If it is under 100%, the work cost less effort than planned; a CPI greater than 100% means that the estimate was not adequate for the work involved. CPI can be used either to compare projects or phases within a project. For example, if the programming tasks took twice as long as estimated but every other type of task in the project took less time than estimated, the total variance for the project might still be low. However, the problem can still be pinpointed by calculating the CPI for each phase of development. If the CPI for the programming phase is well over 100% while the CPI for all other phases is less than 100%, then the project manager should pay extra attention to the estimates for the programming phase in future projects. It is also possible to compare the overall CPI for various projects against one another to see if the CPI goes down over multiple projects. If it does, that shows that the team's estimation skills are improving.

CPI can also be used to find systemic process problems. For example, if, over the course of many projects, the CPI in the coding phase is much lower than the CPI in the other phases, it could mean that there are either problems with the way programming estimates are generated or problems with the execution of the coding tasks. It may also mean that there are uncontrolled changes, which are going unnoticed until the team begins coding. The project manager should take this low CPI as a hint to look more closely at exactly what is causing delays during the programming phase.

In the example given above, the CPI would be (360 / 390) \* 100 = 92.3%. This means that the team is only operating at 92.3% efficiency. Since the CPI is below 100%, the team did not perform as efficiently as expected. Since these numbers were just for the requirements phase, the low CPI could simply mean that the team underestimated the requirements tasks. If the average CPI for the requirements phase in the team's other projects is around 90%, then it means the team systematically underestimates requirements tasks. However, if it is much closer to 100%, then there was a specific problem which caused the team to lose effort in this particular project. The project manager can take appropriate action depending on which of these cases is true.

The CPI calculation is useful when comparing projects to one another, as well as phases within projects. The project manager can create a chart of the CPI for each completed phase of the project to determine the accuracy of the estimates. If there are many phases in which the CPI is far from 100%, this information should be taken into account during future estimation sessions. The total CPI of the entire project can be compared to that of other projects, as well, to determine which teams may need estimation training or better management of resources.

The progress of the project should be tracked at the review meetings in terms of slips, variance, and earned value. The simplest way to track the project's progress is by comparing the due date of the actual schedule with the due date of the baseline to anticipate the expected delay in the due date. The variance data and the individual delays that led to the variance should also be recorded, as well as any viable theories or conclusions drawn about why the schedule slipped. This information will be taken into account in the post-mortem report (see [Chapter 8](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch08.html)).

**SCHEDULING**

https://learning.oreilly.com/library/view/mastering-software-project/9781604276909/images/9781604270341_fig0002.jpg

**INTRODUCTION**

Scheduling is a very important activity in software project management. To say that project planning in many organizations consists of only the scheduling activity would not be an exaggeration. In these organizations, project planning refers only to preparation of a schedule. A schedule during the project planning stage, however, is actually a *calendar* through which the project is envisaged for execution. This practice of schedule-based project planning/managing is apparently seductively simple, but a schedule is only one component of project planning. The schedule is but one of the tools that is available to a software project manager for project monitoring.

Scheduling in its simplest form is the sequencing and setting of calendar dates for the project activities that have been envisaged to accomplish the goals of a project. Yet, scheduling is not merely a rote activity. Good scheduling requires human creativity and ingenuity. When scheduling a project, having an understanding of the aspects of a project is essential:

•A project consists of a number of activities (tasks). Performing these activities/tasks results in execution of the project.

•A project has a number of milestones. Reaching the milestones signifies completion of a certain group of activities.

•A project has a starting point, which is the project’s first milestone or *start* milestone.

•A project has an ending point, which is the project’s last milestone or *end* milestone.

•All activities of a project must be performed between the start and the end milestones.

•Some activities of a project can be performed concurrently with (parallel to) each other.

•Some activities must be performed sequentially (one after the other).

•Some activities can use multiple resources and some activities cannot.

•There is a limit to the number of resources that can be deployed for any given activity.

Let’s now schedule a project using a sample project.

**THE INITIAL WORK BREAKDOWN STRUCTURE**

The list of activities/tasks and milestones needed to execute and complete a project is commonly known as a work breakdown structure or WBS. The first item in a WBS is the *start* milestone, which signifies the beginning of the project. The last item in a WBS is the *end* milestone, which signifies completion of the project. The project “happens” between these two milestones.

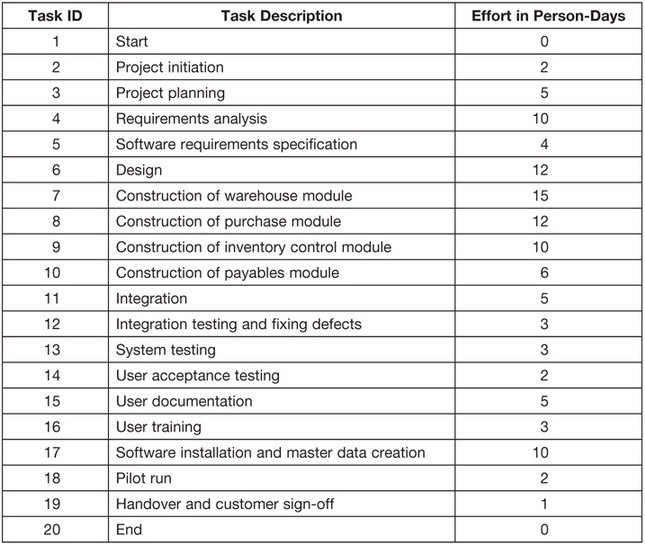
So, the first step in scheduling a project is to prepare a WBS that contains all of the tasks that are to be scheduled. A simple initial WBS for a materials management software development project is illustrated in [Table 9.1](https://learning.oreilly.com/library/view/mastering-software-project/9781604276909/9781604270341_ch_09.xhtml#tbl0014), where all activities are embedded between the *start* and *end* milestones. Of course, a real-life project would have many more activities.

**A WORK BREAKDOWN STRUCTURE WITH PREDECESSORS DEFINED**

Having prepared the initial WBS, the next step is to determine the sequence of the execution of the tasks listed in the WBS in [Table 9.1](https://learning.oreilly.com/library/view/mastering-software-project/9781604276909/9781604270341_ch_09.xhtml#tbl0014). This is achieved by adding a *Predecessor* column to [Table 9.1](https://learning.oreilly.com/library/view/mastering-software-project/9781604276909/9781604270341_ch_09.xhtml#tbl0014) (as shown in [Table 9.2](https://learning.oreilly.com/library/view/mastering-software-project/9781604276909/9781604270341_ch_09.xhtml#tbl0015)). Some organizations use *predecessors* and *successors* (what must come next) as a tool to define sequence. By definition, the *start* milestone does not have any predecessors (even though we know from earlier discussions that significant activities may have taken place to acquire the project before its start). The other project activities, however, should have at least one predecessor (some activities may have more). By definition, the *end* milestone has no successors.

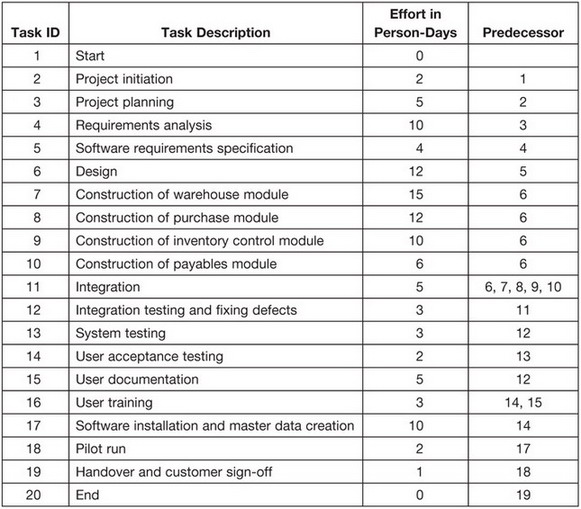
Defining the predecessors consists of a process that includes considering each activity and then answering the question, “What activities should have already been completed before this activity can begin?” The answer is recorded in the *Predecessor* column. The *Predecessor* column indicates all of the activities that need to be completed before the next activity can begin. The scheduler walks through the WBS, iterating the process of asking and answering this “order question” for each activity in the WBS, to ensure that predecessors are identified and recorded for all activities. Some activities will have only one predecessor, while others may have multiple activities as predecessors.

**Table 9.1. Initial Work Breakdown Structure**



A task can also have multiple predecessors or multiple successors. In [Table 9.2](https://learning.oreilly.com/library/view/mastering-software-project/9781604276909/9781604270341_ch_09.xhtml#tbl0015), the predecessor and the successor for each task are shown. For Task 2, the predecessor is Task 1 and the successor is Task 3 (and Task 2 is the predecessor for Task 3). For Task 5, the predecessor is Task 4 and the successor is Task 6. Notice that Task 6 is a predecessor for five tasks: five tasks can start once Task 6 is completed. Notice also that Task 11 has five predecessors: Task 11 cannot start until these five tasks have been completed (think of this as web converging to a single point). (*Note*: Predecessors and successors for each task may also be mapped in a Gantt/PERT chart, which will be briefly described at the end of this chapter.) In summary:

**Table 9.2. Work Breakdown Structure with Predecessors**



•The number of milestones that can be documented between the *start* and *end* milestones have no limit: milestones enhance a schedule’s clarity and understanding.

•Except for the *start* milestone of a schedule, which has no predecessor, and the *end* milestone, which has no successor, every task must have one or more predecessors and one or more successors.

*Note:* There may be multiple tasks as successors to the *start* milestone which are beyond the scope of the present project. Similarly, the *end* milestone may have multiple predecessors which are again beyond the scope of the present project.

Returning to [Table 9.2](https://learning.oreilly.com/library/view/mastering-software-project/9781604276909/9781604270341_ch_09.xhtml#tbl0015), notice that no task lists Task 16 as a predecessor. But doesn’t Task 16 have a successor? This is an anomaly that has to be rectified before we can have a complete schedule. Conversely, if we cannot perceive that another task or activity is a successor to an activity, its successor by definition is the *end* milestone.

Also notice in [Table 9.2](https://learning.oreilly.com/library/view/mastering-software-project/9781604276909/9781604270341_ch_09.xhtml#tbl0015), that an analysis of the schedule will raise questions about the predecessor relationships. Look at Tasks 2 and 3. Task 3 (project planning) cannot be started unless Task 2 (project initiation) has been completed. This relationship is called a *finish-to-start relationship*: Task 2 must be finished before Task 3 can be started. Task 11 (integration) can start once Task 6 (design) has been completed, and other modules can be integrated when any module is completed. Therefore, there is a *finish-to-start relationship* between Task 6 and Task 11. So, in our example, Task 11 can start when Task 6 is finished, but Task 11 cannot be completed until Tasks 7, 8, 9, and 10 are completed. The relationship between Task 11 and Tasks 7, 8, 9, and 10 is called *finish-to-finish relationship*.

Now, look at Task 11 (integration) and Task 12 (integration testing and fixing defects). Should Task 12 wait until all four of the modules are integrated? It could, but waiting is not necessary because when a module is integrated, its integration can be tested. The relationship between Task 11 and Task 12 therefore is a called a *start-to-start relationship*. Task 12 can be started after Task 11 starts, but with the time lag that is necessary to allow finishing the integration of the first module.

To account for all of the possible relationships, one more relationship needs to be defined: the start-to-finish relationship. In the start-to-finish relationship, Task “n” must be started to finish Task “m.” The start-to-finish relationship, however, is atypical in software development and is described here only for the sake of completeness.

Summarizing, there are four types of predecessor relationships:

•Finish (predecessor)-to-start (successor) or **FS-n** (with “n” being the days the successor must wait after finishing the predecessor; if “n” is not mentioned, n = 0)

•Start (predecessor)-to-start (successor) or **SS-n** (with “n” being the days the successor must wait after starting the predecessor; if “n” is not mentioned, n = 0)

•Finish (predecessor)-to-finish (successor) or **FF-n** (with “n” being the days the successor must wait after finishing the predecessor; if “n” is not mentioned, n = 0)

•Start (successor)-to-finish (predecessor) or **SF-n** (with “n” being the days the successor must wait after starting the predecessor to finish successor; if “n” is not mentioned, n = 0)

For each of these relationships, a *lag* (waiting time) may be specified before the successor is started:

•Task 3 can be started 1 day after finishing Task 2. This is depicted as **FS-1:** the relationship of Task 3 to predecessor Task 2 is **f**inish-to-**s**tart with a lag of **1** day.

•Task 12 can be started after 2 days of starting Task 11. This is depicted as **SS-2**: the relationship of Task 12 to Task 11 is **s**tart-to-**s**tart with a lag of**2** days.

**A WORK BREAKDOWN SCHEDULE WITH INITIAL DATES**

Once the structure of the WBS has been completed by defining the predecessors and the predecessor relationships, and by ensuring that all tasks have predecessors and successors, the next step is to start assigning dates to the tasks. Note the following points from the schedule depicted in [Table 9.3](https://learning.oreilly.com/library/view/mastering-software-project/9781604276909/9781604270341_ch_09.xhtml#tbl0016):

•The start date for the *start* milestone is the project’s starting date.

•The end date for the *end* milestone is the project’s completion date.

•Weekends (Saturday and Sunday) are not counted as working days. (Also exclude holidays, e.g., notice that July 4, which is Independence Day in the United States, is excluded in Task 18.)

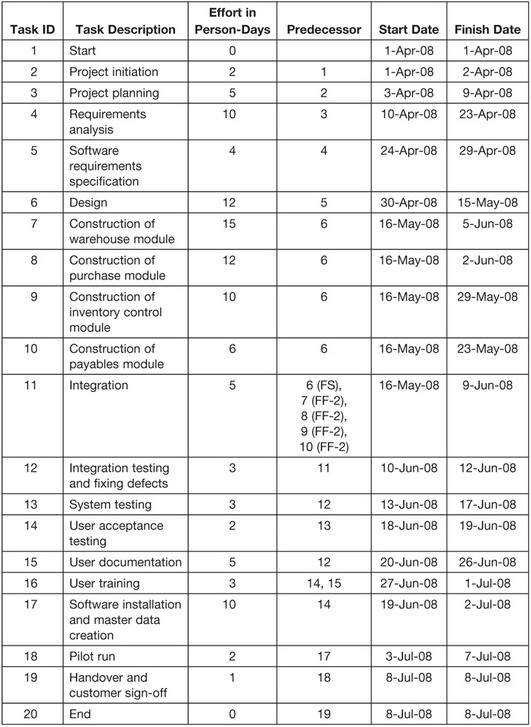
•Task 3 starts April 3 and Task 2 is completed on April 2 (the day before). Why? Because when a task is to be completed on April 2, typically the task will be completed by the end of the working day on April 2. Therefore, the successor can only start the next day.

•Task 11 (with five predecessors) starts on May 16, the day after the completion of Task 6. Task 11 has a finish-to-start (FS) relationship with Task 6. Task 11 also has a finish-to-finish with 2 days lag (FF-2) relationship with the rest of its predecessors. Therefore, Task 11 completes on June 9, 2 working days after the completion of Task 7. Task 7 is the predecessor that finishes last (on June 5) of all the predecessors of Task 11. Because the lag is 2 days, Task 11 completes on June 9, which is 2 working days after the completion of its last predecessor.

•No relationship for Task 16 is given. When no relationship is explicitly given, the relationship is a finish-to-start relationship (FS) with no lag. Task 16 has two predecessors: Task 14 (completes on June 19) and Task 15 (completes on June 26). Therefore, Task 16 can start 1 day after Task 15, which is the last of Task 16’s predecessors.

•Look at the *end* milestone, which has two predecessors. Both of these predecessors must be completed for the end milestone to be reached. Therefore, the start date (as well as the end date) is July 7, the day on which Task 19 (the last task) is completed.

**Table 9.3. Work Breakdown Structure with Initial Dates**



Some inferences for future use may be drawn from this description of the relationships:

https://learning.oreilly.com/library/view/mastering-software-project/9781604276909/images/9781604270341_chr0001.jpgThe *start* date of an activity depends on its relationship with its predecessors:

•In a finish-to-start relationship, the start date depends on the predecessor that finishes last.

•In a start-to-start relationship, the start date depends on the predecessor that starts first.

•Other relationships have no impact.

https://learning.oreilly.com/library/view/mastering-software-project/9781604276909/images/9781604270341_chr0001.jpgThe *end* date of an activity depends on its duration *and* on the relationship with its predecessors:

•In a finish-to-finish relationship, the end date depends on its predecessor finishing last.

•Other relationships have no impact.

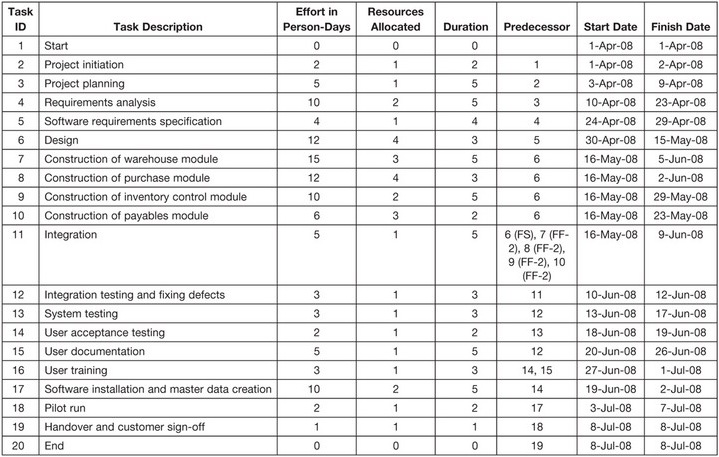
**A WORK BREAKDOWN STRUCTURE WITH RESOURCE ALLOCATION**

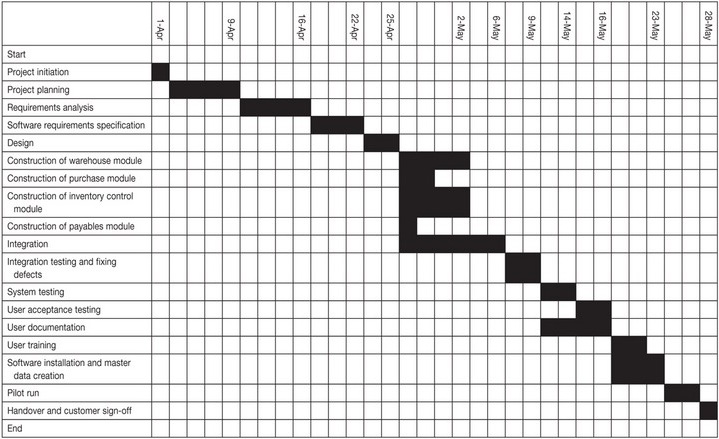
In [Table 9.3](https://learning.oreilly.com/library/view/mastering-software-project/9781604276909/9781604270341_ch_09.xhtml#tbl0016), the term *effort* is used synonymously for *duration*. In our example, this synonymous use of the term allows us to assume that only one resource has been allocated to the project. In most real-life projects, however, multiple resources are allocated to a project and the resources have different skill sets. Naturally, multiple resources and the various skill sets of these resources result in differences between the effort and the duration for specific activities.

For example, say that coding takes 100 person-days to complete. So, if one programmer is allocated to the task, the duration will be 100 workdays; if two programmers are allocated, the duration will be 50 workdays; and if four programmers are allocated, the duration will be 25 workdays (assuming that all programmers are equal). So, to get a realistic schedule, we need to add a *Resource Allocated* column and a *Duration* column to [Table 9.3](https://learning.oreilly.com/library/view/mastering-software-project/9781604276909/9781604270341_ch_09.xhtml#tbl0016) and adjust duration.

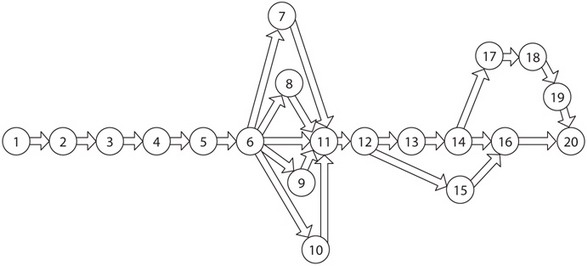
Now, look at [Table 9.4](https://learning.oreilly.com/library/view/mastering-software-project/9781604276909/9781604270341_ch_09.xhtml#tbl0017). Notice that *Duration* (effort ÷ number of resources) has been adjusted for each task by taking into consideration the number of resources allocated for each task. Duration depends on the effort in person-days and the number of resources allocated for the activity. The dates in the schedule have been set based on the duration and predecessor relationships. [Table 9.4](https://learning.oreilly.com/library/view/mastering-software-project/9781604276909/9781604270341_ch_09.xhtml#tbl0017) now reflects all of the components needed to develop a useable schedule.

**Table 9.4. Work Breakdown Structure with Resource Allocation**





**Figure 9.1.** Gantt chart.



**Figure 9.2.** Network diagram.

**SCHEDULING IN PRACTICE**

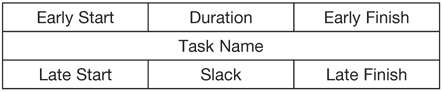
In actual practice, manual iteration is not required as many times as we have done in our example. Tools such as spreadsheets (e.g., Microsoft Excel) may be used and information can be filled in column by column. Excel’s capability for date arithmetic can then be used to our advantage for assigning dates to tasks. Specialized software tools, such as Primavera, Microsoft Project, and PMPal, can also assist in scheduling. These tools take weekends and holidays into account when assigning dates to tasks. Using an automated spreadsheet or specialized scheduling software also makes scheduling easier when a project start date is shifted or if a change in any of the tasks subsequently requires recalculation of the schedule.

**GRAPHIC REPRESENTATION OF A SCHEDULE**

Graphic representations of schedules are frequently used. Two popular graphic representations are bar charts (also called Gantt charts) and network diagrams:

***Bar charts.*** A Gantt chart is illustrated in [Figure 9.1](https://learning.oreilly.com/library/view/mastering-software-project/9781604276909/9781604270341_ch_09.xhtml#fig0030). A Gantt chart is a type of bar chart that illustrates a project schedule. Gantt charts can be produced using Microsoft Excel spreadsheets or scheduling packages such as Primavera and Microsoft Project.

***Network diagram.*** Network diagrams may take various forms. In [Figure 9.2](https://learning.oreilly.com/library/view/mastering-software-project/9781604276909/9781604270341_ch_09.xhtml#fig0031), each task is depicted in a circle and is identified by only its task IDs. In more traditional network diagrams, a task is depicted in the arrow of the network and the circle depicts the milestone. In network diagrams currently being used in the software development industry, the arrow represents only a predecessor relationship.



**Figure 9.3.** Network node diagram.

***Network node diagram.*** The most frequently used depiction of an activity, however, is illustrated in [Figure 9.3](https://learning.oreilly.com/library/view/mastering-software-project/9781604276909/9781604270341_ch_09.xhtml#fig0032). This graphic representation of a task is in a rectangular shape that is divided into seven sections. Variations of this representation are found in effective scheduling software tool packages, such as Microsoft Project, Primavera, PMPal, EstimatorPal, etc.

# Managing Multiple Projects

Many project managers are responsible for multiple projects . If each project is planned well, managing a set of them should not be difficult. When projects don't share dependencies, managing them is straightforward—just manage each project individually, with a separate project schedule for each one.

When projects share dependencies, they are more challenging to manage. There are two ways that one project might depend on another. In the first type, two projects rely on the same resources; in the second type, a work product generated by one project is needed by the other. Getting a handle on these dependencies is the first important step in managing multiple projects.

## Understand Dependencies Between Projects

The most common way for projects to be interdependent is through shared resources. One instance of this happening is "pipelined" projects. In many software organizations, software projects go through a set of sequential phases: requirements, design, development, programming, and testing. In each phase, most of the work is done by a small subset of the team, leaving the rest of the team available to work on other projects. To allow the team to work at full capacity, they might be working on several projects at once. Project A is in the requirements phase, while at the same time, project B is in design, project C is in development, and project D is being tested.

The trouble with this system is that no two projects take exactly the same time, and the phases don't always require the same percentage of the total effort. For example, programming is typically 30% to 40% of the total effort of a project. But during that time, a lead tester might be working on a test plan, while programmers might have to stop work for half a day to review the requirements or design documents on another project. When there are more than two or three projects being worked on simultaneously, things can get very chaotic.

Usually, a programmer is fully allocated to a programming task. This means that the programmer is spending all of his time on that task (minus lunch, bathroom breaks, etc.) But in other cases, a resource will be only partially allocated to a task. For example, a conference room may be a scarce resource; it might be reserved only in the morning, but free in the afternoon. Or a designer might work on two projects at once, meeting with stakeholders for one project in the morning while analyzing usability lab data for another in the afternoon.

The project manager's first goal is to make sure that the shared resources are not over-allocated. If one project's schedule has a resource allocated 50% for the entire week, while another has that resource allocated 100% during the same week, that resource has a 150% allocation. Over-allocation problems often do not show up on the schedule.

What's more, if the estimates do not include overhead (going to meetings, reading email, helping customers, talking to senior managers, or other interruptions), then a person can be over-allocated even if the schedule says that she isn't. When taking effort into account on a project schedule, it's important to remember that even though people may be in the office for eight hours each day, they might only be available for project work for five of those hours. Also, the project manager must make sure that changes are controlled properly. If the scope of the project changes but the team is not given a chance to create new estimates, team members will almost certainly end up over-allocated.

Under-allocation is also a danger. If an engineer does not have any scheduled tasks for a week, she can easily get bored. That engineer may not be unhappy about the situation, but if the rest of the team is crunched for time, their morale will be impacted when they see their teammate take off early every day. One way to prevent this is to have low-priority projects where tasks can be assigned to under-allocated team members. However, enough time must be given to allow each person to ramp up on the task and finish enough to feel like the time was not wasted; otherwise, it just feels like busywork.

Modern project management software will often have a "resource pool" feature that allows a project manager to set up a single set of resources available to multiple projects. When a project schedule draws a resource from that pool, the allocation level for that resource is increased accordingly in the pool, so that allocation shows up on all of the other project schedules. Allocation reports can be run to verify that no resource is over- or under-allocated. Alternately, multiple projects can be included on a single schedule. This is slightly harder to maintain, but very easy to understand at a glance.

Another common way for two projects to be interdependent is when one project has a task that has a predecessor in another project. Identifying these predecessors is generally straightforward. Any time a task relies on a piece of software, a document, or another work product that is scheduled to be built in another project, a dependency is created in the schedule. The easiest way to handle this situation is to require that the dependent task does not begin until its predecessor ends. Modern project management software tools have features that help to automate this process by grouping multiple projects together into one master schedule.

The one important pitfall is that each cross-project dependency increases the risk of delay on the dependent project. To mitigate this risk, every predecessor must be reviewed at the project meetings for the dependent project.

## Prioritize Projects Realistically

Prioritizing projects is similar to prioritizing tasks—it requires tough decisions, and will almost always make someone unhappy. Priorities are always relative: each project's stakeholder feels that his project is the most important one. And in a way, he's right—it's the most important one to him, but not necessarily to the organization. That doesn't change the fact that if there is a programmer available to work and there are two projects that need to get done, a decision must be made to assign her to one project or the other.

While prioritizing projects seems to require the same actions as prioritizing tasks, it is much more politically charged, and the project manager is under much more pressure to throw away the prioritization entirely and pretend that all of the projects can be done at the same time. This is usually a mistake—unless two projects do not share any resources at all, there will come a time when a resource must be assigned to either one project or the other. If there is no clear priority, this can create confusion and chaos.

Prioritizing projects is about making decisions. Someone has to put his foot down and say that project A is more important than project B. In some organizations, this is the project manager; more often, it is one or more senior managers (often a steering committee) who have the authority to make the decision. One reliable way to figure out who should be making that decision is to follow the money: the person who has the authority to allocate funds to pay engineers and buy computers is generally the person who has the authority to decide which project should be worked on first. Ideally, this responsibility should fall on a single person (often the chair of the steering committee). If no decision-maker can be found, it becomes the project manager's most important responsibility to find that decision-maker.

Once the decision-maker is found, the process of prioritization is straightforward. It is similar to the way risks are prioritized, with the exception that no two projects are allowed to receive the same priority. Without this restriction, the team will end up with only top-priority projects. If the decision-maker has trouble choosing a priority, he can use the same technique used for risk planning. If there are 20 projects to prioritize, he should identify top-priority project and assign it priority 1, then the lowest-priority project and assign it priority 20. Each additional project should then be prioritized in relation to those. (If it is absolutely impossible to decide the relative priority of two projects, he can flip a coin; if the coin's answer seems wrong, he should reverse it. This is a helpful way to "force" him to consider the alternative.)

The final result of the prioritization process is a list of projects that are arranged in order of priority, with a unique priority must be assigned to each project. This list of projects should also indicate which projects are dependent on other ones. This inclusion means that when a team member is identifying the next task to work on, its dependencies can be taken into consideration.

It is often tempting to assign the same priority to two different projects that seem to be equally important to the organization. It is sometimes difficult to make priority decisions; basically, they amount to deciding that one project may be delayed at the expense of another. But if these decisions are not made, that can lead to serious project problems. The most common problem is that the team ends up with several projects that are top priority, and it's not clear which one should be worked on next. This is bad because it's almost always the case that one of those projects really is more important that the others, so there is a good chance the team will work on a less important project. When many projects have a priority of 1 assigned, then the prioritization is essentially useless.

Once priorities are assigned to all projects, it's time to assign resources to the tasks. Each resource should be assigned to the next task on the project with the highest priority that does not yet have work being done. This is not necessarily a hard-and-fast rule: if a task on a lower-priority project must be done in order to make the schedules work out properly and to avoid over-allocation, the schedules should reflect that. And in many cases, there is only one person on the team who has the expertise to perform a specific task. But, wherever possible, the priorities should be honored.

Finally, a periodic priority meeting should be held to reevaluate the project priorities. Some organizations do this weekly, while others do it every two weeks or even monthly. This meeting can be piggybacked on the status meeting, as long as all of the project's decision-makers are there. The important part is that everyone is kept in the loop: the project manager, the lead decision-maker, and the stakeholders for the projects.

# Use the Schedule to Manage Commitments

A project schedule represents a commitment by the team to perform a set of tasks. When the project manager adds a task to the schedule and it's agreed upon by the team, the person who is assigned to that task now has a commitment to complete it by the task's due date. Senior managers feel that they can depend on the schedule as an accurate forecast of how the project is going to go—when the schedule slips, it's treated as an exception, and an explanation is required. For this reason, the schedule is a powerful tool for commitment management .

One common complaint among project managers attempting to improve the way their organizations build software is that the changes they make don't take root. Typically, the project manager will call a meeting to announce a new tool or technique—he may ask the team to start performing code reviews, for example—only to find that the team does not actually perform the reviews when building the software. Things that seem like a good idea in a meeting often fail to "stick" in practice.

This is where the schedule is a very valuable tool. By adding tasks to the schedule that represent the actual improvements that need to be made—for example, by scheduling all of the review meetings—the project manager has a much better chance of gaining a real commitment from the team.

If the team does not feel comfortable making a commitment to the new practice, the disagreement will come up during the schedule review. Typically, when a project team member disagrees with implementing a new tool or technique, he does not bring it up during the meeting where it's introduced. Instead, he will simply fail to use it, and build the software as he has on past projects. This is usually justified with an explanation that there isn't enough time, and that implementing the change will make the task late.

By explicitly adding a task to the schedule, the project manager ensures that enough time is built in to account for the change. This cements the change into the project plan, and makes it clear up front that the team is expected to adopt the practice. More importantly, it is a good consensus-building tool because it allows team members to bring up the new practice when they review the project plan. By putting the change out in the open, the project manager encourages real discussion of it, and is given a chance to explain the reason for the practice during the review meetings. If the practice makes it past the review, then the project manager ends up with a real commitment from the team to adopt the new practice.

# Diagnosing Scheduling Problems

When a project manager doesn't create a schedule, the organization is given an unrealistic view of how the project will progress. When schedules are not correct, the project manager usually has to resort to drastic measures in order to try to bring the project in line with the organization's expectations, and those measures often don't work. Even when they do, they hurt the morale of the team, and they frequently hurt the quality of the software produced as well.

## Working Backward From a Deadline

One of the most common problems that affects a project is that the deadline, which seemed perfectly reasonable when the project started, begins to seem completely unrealistic as the date starts getting closer. This is often caused by a project manager facing a deadline that cannot be changed. Usually, the date comes from marketing or customer relations needs. Instead of being based on estimates of actual effort from the team, expectations are based on agreements between project managers, senior managers, and stakeholders. (For example, a consulting company may take on an overly aggressive deadline in order to satisfy a client seen as important for the company's growth.)

When faced with a non-negotiable deadline for a project, many project managers will work backward from the deadline to determine what work needs to be performed. One misguided way of doing this is to divide the project into phases and assign each phase a certain percent of the schedule. The project manager may decide, for example, that programming should take 60% of the time, testing should take 25%, etc. These numbers don't come from any specific knowledge of the work required; rather, they come from the need to fit that work into a predetermined tomfooleries.

What results is deadline-driven work. If milestones look like they will be missed, key activities like reviews, inspections, and even testing are often abandoned in order to meet unrealistic expectations. The people working on the project are treated unfairly because they are asked to perform an impossible task. They may be told to work overtime or spend weekends in the office to make up for poor planning.

This is also unfair to the stakeholders—especially if they are clients of a business. Many businesses will see certain clients as important and will promise things they can't deliver. A project manager in this situation, who creates an unrealistic schedule to meet those commitments, does not necessarily recognize that the project's deadline is unrealistic. More often, the client is blamed for expecting too much or for being too picky about the deliverable. The project manager may also blame a marketing or sales department for over-promising. But, truthfully, it is the project manager's fault: he agreed to an unrealistic schedule rather than being honest about the likelihood of failure and presenting alternatives like adding resources, reducing scope, proposing a phased release, bringing on consultants, or using different technologies. Had the project manager been up front with the stakeholders from the beginning, they might have avoided this mess.

## Misunderstood Predecessors

Sometimes, a deadline that seemed reasonable based on the effort estimates can still go awry, if the project manager has not taken the time to understand how the tasks depend on each other. If a dependency is discovered halfway through the project, it can send the entire team into chaos.

This situation is most common when the team does not sit down to create a work breakdown structure. When a WBS has not been created, it is not uncommon to discover important tasks required to complete the project well after the work has started. By the time the extent of the poor estimate is known, it may be too late to change expectations within the organization.

For example, it may seem like all of the work for a particular project will get done on time. Suddenly, halfway through the project, a programmer discovers that he needs a component that isn't scheduled to be built by another team member for another two weeks. The code that he is building is in turn necessary for another programmer, who will be using it as soon as he is done. Now, instead of being done on time, he will be stuck for the next two weeks; another programmer will be stuck for even longer, waiting for him to complete his work. Unfortunately, the project manager already committed to a deadline. As a result, some team members will have to work overtime to make up for the lost time—while others are left sitting idle.

When predecessors are not discovered until the project is underway, there are usually few opportunities for correction. Critical team members are already working on other tasks, and end dates may have already been agreed upon. What's more, in a tight schedule, predecessor problems often cascade. When one task has to wait, all of the tasks that depend on it will also have to wait, as will the tasks that depend on those, and so forth.

Often, these cascading delays aren't fully recognized by the team until late in the project. Since each person performs each task in the amount of time estimated, the project manager might not realize the problem throughout most of the project. To the project manager, it seems that things are moving along at a steady pace; it is not until the project nears completion that it becomes apparent that the deadline is in jeopardy.

This problem is especially hard on software testers, simply because they are responsible for the tasks at the tail end of the software project. Since the project is in its final phase when the problem is discovered, the testers are responsible for the bulk of the remaining work. This is especially unfair when the root cause of the delay is in an earlier phase of the project: the testers did not create the problem, yet they bear the brunt of the pressure!