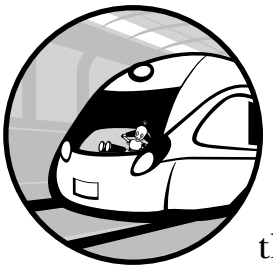
2

PRODUCTIVITY



In the late 1960s, it was clear that training more programmers would not alleviate the software crisis. The only solution was to increase programmer productivity—that is, enable existing programmers to write more code—which is how the software engineering field originated. Therefore, a good place to start studying software engineering is with an understanding of productivity.

2.1 What Is Productivity?

Although the term *productivity* is commonly described as the basis for software engineering, it's amazing how many people have a distorted view of it. Ask any programmer about productivity, and you're bound to hear "lines of code," "function points," "complexity metrics," and so on. The truth is,

there is nothing magical or mysterious about the concept of productivity on a software project. We can define productivity as:

> The number of unit tasks completed in a unit amount of time or completed for a given cost.

The challenge with this definition is specifying a *unit task*. One convenient unit task might be a project; however, projects vary wildly in terms of size and complexity. The fact that programmer A has completed three projects in a given amount of time, whereas programmer B has worked only on a small portion of a large project, tells us nothing about the relative productivity of these two programmers. For this reason, the unit task is usually much smaller than an entire project. Typically, it's something like a function, a single line of code, or an even smaller component of the project. The exact metric is irrelevant as long as the unit task is consistent between various projects and a single programmer would be expected to take the same amount of time to complete a unit task on any project. In general, if we say that programmer A is n times more productive than programmer B, programmer A can complete n times as many (equivalent) projects in the same amount of time as it would take programmer B to complete one of those projects.

2.2 Programmer Productivity vs. Team Productivity

In 1968, Sackman, Erikson, and Grant published an eye-opening article claiming that there was a 10 to 20 times difference in productivity among programmers.[1] Later studies and articles have pushed this difference even higher. This means that certain programmers produce as much as 20 (or more) times as much code as some less capable programmers. Some companies even claim a two-order-of-magnitude difference in productivity between various software teams in their organizations. This is an astounding difference! If it's possible for some programmers to be 20 times more productive than others (so-called Grand Master Programmers [GMPs]), is there some technique or methodology we can use to improve the productivity of a typical (or low-productivity) programmer?

Because it's not possible to train every programmer to raise them to the GMP level, most software engineering methodologies use other techniques, such as better management processes, to improve the productivity of a large team. This book series takes the other approach: rather than attempting to increase the productivity of a team, it teaches individual programmers how to increase their own productivity and work toward becoming a GMP.

Although the productivity of individual programmers has the largest impact on a project's delivery schedule, the real world is more concerned with project cost—how long it takes and how much it costs to complete the

1. Harold Sackman, W. J. Erikson, and E. E. Grant, "Exploratory Experimental Studies Comparing Online and Offline Programming Performance," *Communications of the ACM* 11, no. 1 (1968): 3–11.

project—than with programmer productivity. Except for small projects, the productivity of the *team* takes priority over the productivity of a *team member*.

Team productivity isn't simply the average of the productivities of each member; it's based on complex interactions between team members. Meetings, communications, personal interactions, and other activities can all have a negative impact on team members' productivity, as can bringing new or less knowledgeable team members up to speed and reworking existing code. (The lack of overhead from these activities is the main reason a programmer is far more productive when working on a small project than when working on a medium- or large-sized project.) Teams can improve their productivity by managing overhead for communication and training, resisting the urge to rework existing code unless it's really necessary, and managing the project so code is written correctly the first time (reducing the need to rework it).

2.3 Man-Hours and Real Time

The definition given earlier provides two measures for productivity: one based on time (productivity is the number of unit tasks completed in a unit amount of time) and one based on cost (productivity is the number of unit tasks completed for a given cost). Sometimes cost is more important than time, and vice versa. To measure cost and time, we can use man-hours and real time, respectively.

From a corporation's view, the portion of a project's cost related to programmer productivity is directly proportional to its *man-hours*, or the number of hours each team member spends working on the project. A *man-day* is approximately 8 man-hours, a *man-month* is approximately 176 man-hours, and a *man-year* is approximately 2,000 man-hours. The total cost of a project is the total number of man-hours spent on that project multiplied by the average hourly wage of each team member.

Real time (also known as *calendar time* or *wall clock time*) is just the progression of time during a project. Project schedules and delivery of the final product are usually based on real time.

Man-hours are the product of real time multiplied by the number of team members concurrently working on the project, but optimizing for one of these quantities doesn't always optimize for the other. For example, suppose you're working on an application needed in a municipal election. The most critical quantity in this case is real time; the software must be completely functional and deployed by the election date regardless of the cost. In contrast, a "basement programmer" working on the world's next killer app can spend more time on the project, thus extending the delivery date in real time, but can't afford to hire additional personnel to complete the app sooner.

One of the biggest mistakes project managers make on large projects is to confuse man-hours with real time. If two programmers can complete a project in 2,000 man-hours (and 1,000 real hours), you might conclude that four programmers can complete the project in 500 real hours. In other

words, by doubling the staff on the project, you can get it done in half the time and complete the project on schedule. In reality, this doesn't always work (just like adding a second oven won't bake a cake any faster).

Increasing staff to increase the number of man-hours per calendar hour is generally more successful on large projects than on small and medium-sized projects. Small projects are sufficiently limited in scope that a single programmer can track all the details associated with the project; there's no need for the programmer to consult, coordinate with, or train anyone else to work on the project. Generally speaking, adding programmers to a small project eliminates these advantages and increases the costs dramatically without significantly affecting the delivery schedule. On medium-sized projects, the balance is delicate: two programmers may be more productive than three,[2] but adding more programming resources can help get an understaffed project finished sooner (though, perhaps, at a greater cost). On large software projects, increasing the team size reduces the project's schedule accordingly, but once the team grows beyond a certain point, you might have to add two or three people to do the amount of work usually done by one person.

2.4 Conceptual and Scope Complexity

As projects become more complex,[3] programmer productivity decreases, because a more complex project requires deeper (and longer) thought to understand what is going on. In addition, as project complexity increases, there's a greater likelihood that a software engineer will introduce errors into the system, and that defects introduced early in the system will not be caught until later, when the cost of correcting them is much higher.

Complexity comes in a couple of forms. Consider the following two definitions of *complex*:

1. Having a complicated, involved, or intricate arrangement of parts so as to be hard to understand
2. Composed of many interconnected parts

We can call the first definition *conceptual complexity*. For example, consider a single arithmetic expression in a high-level language (HLL), such as C/C++, which can contain intricate function calls, several weird arithmetic/logical operators with varying levels of precedence, and lots of parentheses that make the expression difficult to comprehend. Conceptual complexity can occur in any software project.

We can call the second definition *scope complexity*, which occurs when there is too much information for a human mind to easily digest. Even if the individual components of the project are simple, the sheer size of the

2. Barry W. Boehm, Terence E. Gray, and Thomas Seewaldt, "Prototyping Versus Specifying: A Multiproject Experience," *IEEE Transactions on Software Engineering* 10, no. 3 (1984): 290–303.

3. Generally, this means larger, although conceptual complexity applies as well.

project makes it impossible for one person to understand the whole thing. Scope complexity occurs in medium- and large-scale projects (indeed, it's this form of complexity that differentiates small projects from the others).

Conceptual complexity affects programmer productivity in two ways. First, complex constructs require more thought (and therefore more time) to produce than simple constructs. Second, complex constructs are more likely to contain defects that must be corrected later, producing a corresponding loss in productivity.

Scope complexity introduces different problems. When the project reaches a certain size, a programmer on the project might be completely unaware of what is going on in other parts of the project, and might duplicate code already in the system. Clearly, this reduces programmer productivity, because the programmer wasted time writing that code.[4] Inefficient use of system resources can also occur as a result of scope complexity. When working on a part of the system, a small team of engineers might be testing their piece by itself, but they don't see its interaction with the rest of the system (which might not even be ready yet). As a result, problems with system resource usages (such as CPU cycles or memory) might not be uncovered until later.

With good software engineering practices, it's possible to mitigate some of this complexity. But the general result is the same: as systems become more complex, people must spend more time thinking about them and the opportunity for defects increases dramatically. The end result is reduced productivity.

2.5 Predicting Productivity

Productivity is a project attribute that you can measure and attempt to predict. When a project is complete, it's fairly easy to determine the team's (and its members') productivity, assuming the team kept accurate records of the tasks accomplished during project development. Though success or failure on past projects doesn't guarantee success or failure on future projects, past performance is the best indicator available to predict a software team's future performance. If you want to improve the software development process, you need to track the techniques that work well and those that don't, so you'll know what to do (or *not* to do) on future projects. To track this information, programmers and their support personnel must document all software development activities. This is a good example of *pure overhead* introduced by software engineering: the documentation does almost nothing to help get the current project out the door or improve its quality, but it's an investment in future projects to help predict (and improve) productivity.

---

4. Some large projects appoint a "librarian" whose job is to keep track of reusable code components. Programmers looking for a particular routine can ask the librarian about its availability and spare themselves from having to write that code. The productivity loss is limited to the time the librarian spends to maintain the library and the time the programmer and the librarian spend communicating.

Watts S. Humphrey's *A Discipline for Software Engineering* (Addison-Wesley Professional, 1994) is a great read for those interested in learning about tracking programmer productivity. Humphrey teaches a system of forms, guidelines, and procedures for developing software that he calls the *Personal Software Process (PSP)*. Although the PSP is targeted at individuals, it offers valuable insight into where a programmer's problems lie in the software development process. In turn, this can greatly help them to determine how to attack their next major project.

2.6 Metrics and Why We Need Them

The problem with predicting a team's or an individual's productivity by looking at their past performance on similar projects is that it applies *only to similar projects*. If a new project is significantly different than a team's past projects, past performance might not be a good indicator. Because projects vary greatly in size, measuring productivity across whole projects might not provide sufficient information to predict future performance. Therefore, some system of measurement (a *metric*) at a granularity level below a whole project is needed to better evaluate teams and team members. An ideal metric is independent of the project (team members, programming language chosen, tools used, and other related activities and components); it must be usable across multiple projects to allow for comparison between them. Several metrics do exist, but none is perfect—or even very good. Still, a poor metric is better than no metric, so software engineers will continue to use them until a better measurement comes along. In this section, I'll discuss several of the more common metrics and the problems and benefits of each.

2.6.1 Executable Size Metric

One simple metric that programmers use to specify a software system's complexity is the size of the executables in the final system.[5] The assumption is that complex projects produce large executable files.

The advantages of this metric are:

- It is trivial to compute (typically, you need only look at a directory listing and compute the sum of one or more executable files).
- It doesn't require access to the original source code.

Unfortunately, the executable size metric also has deficiencies that disqualify it for most projects:

- Executable files often contain uninitialized data whose contribution to the file size have little or nothing to do with the complexity of the system.

5. Note that a project might contain multiple executable files. In such a case, the "executable file size" is the sum of all the executable components in the system.

- Library functions add to the executable's size, yet they actually reduce the complexity of the project.[6]
- The executable file size metric is not language-independent. For example, assembly language programs tend to be much more compact than HLL executables, yet most people consider assembly programs much more complex than equivalent HLL programs.
- The executable file size metric is not CPU-independent. For example, an executable for an 80x86 CPU is usually smaller than the same program compiled for an ARM (or other RISC) CPU.

2.6.2 Machine Instructions Metric

A major failing of the executable file size metric is that certain executable file formats include space for uninitialized static variables, which means trivial changes to the input source file can dramatically alter the executable file size. One way to solve this problem is to count only the machine instructions in a source file (either the size, in bytes, of the machine instructions or the total number of machine instructions). While this metric solves the problem of uninitialized static arrays, it still exhibits all the other problems of the executable file size metric: it's CPU-dependent, it counts code (such as library code) that wasn't written by the programmer, and it's language-dependent.

2.6.3 Lines of Code Metric

The lines of code (LOC, or KLOC for thousands of lines of code) metric is the most common software metric in use today. As its name suggests, it's a count of the number of lines of source code in a project. The metric has several good qualities, as well as some bad ones.

Simply counting the number of source lines appears to be the most popular form of using the LOC metric. Writing a line count program is fairly trivial, and most word count programs available for operating systems like Linux will compute the line count for you.

Here are some common claims about the LOC metric:

- It takes about the same amount of time to write a single line of source code regardless of the programming language in use.
- The LOC metric is not affected by the use of library routines (or other code reuse) in a project (assuming, of course, you don't count the number of lines in the prewritten library source code).
- The LOC metric is independent of the CPU.

The LOC metric does have some drawbacks:

- It doesn't provide a good indication of how much work the programmer has accomplished. One hundred lines of code in a VHLL accomplishes more than 100 lines of assembly code.

6. Assuming, of course, that the library routines existed prior to the project and were not part of the project's development.

- It assumes that the cost of each line of source code is the same. However, this isn't the case. Blank lines have a trivial cost, simple data declarations have a low conceptual complexity, and statements with complex Boolean expressions have a very high conceptual complexity.

2.6.4 Statement Count Metric

The statement count metric counts the number of language statements in a source file. It does not count blank lines or comments, nor does it count a single statement spread across multiple lines as separate entities. As a result, it does a better job than LOC of calculating the amount of programmer effort.

Although the statement count metric provides a better view of program complexity than lines of code, it suffers from many of the same problems. It measures effort rather than work accomplished, it isn't as language-independent as we'd like, and it assumes that each statement in the program requires the same amount of effort to produce.

2.6.5 Function Point Analysis

Function point analysis (FPA) was originally devised as a mechanism for predicting the amount of work a project would require before any source code was written. The basic idea was to consider the number of inputs a program requires, the number of outputs it produces, and the basic computations it must perform, and use this information to determine a project schedule.[7]

FPA offers several advantages over simplistic metrics like line or statement count. It is truly language- and system-independent. It depends upon the functionality of the software rather than its implementation.

FPA does have a few serious drawbacks, though. First, unlike line count or even statement count, it's not straightforward to compute the number of "function points" in a program. The analysis is subjective: the person analyzing the program must decide on the relative complexity of each function. Additionally, FPA has never been successfully automated. How would such a program decide where one calculation ends and another begins? How would it apply different complexity values (again, a subjective assignment) to each function point? Because this manual analysis is rather time-consuming and expensive, FPA is not as popular as other metrics. Largely, FPA is a *postmortem* (end-of-project) tool applied at the completion of a project rather than during development.

2.6.6 McCabe's Cyclomatic Complexity Metric

As mentioned earlier, a fundamental failure of the LOC and statement count metrics is that they assume each statement has equivalent complexity. FPA fares a little better but requires an analyst to assign a complexity rating to each statement. Unfortunately, these metrics don't accurately reflect the

7. True function point analysis is based on five components: external inputs, external outputs, external inquiries, internal logical file operations, and external file interfaces. But this basically boils down to tracking the inputs, outputs, and computations.

effort that went into the work being measured, and, therefore fail to document programmer productivity.

Thomas McCabe developed a software metric known as *cyclomatic complexity* to measure the complexity of source code by counting the number of paths through it. It begins with a flowchart of the program. The nodes in the flowchart correspond to statements in the program, and the edges between the nodes correspond to nonsequential control flow in the program. A simple calculation involving the number of nodes, the number of edges, and the number of connected components in the flowchart provides a single cyclomatic complexity rating for the code. Consider a 1,000-line `printf` program (with nothing else); the cyclomatic complexity would be 1, because there is a single path through the program. Now consider a second example, with a large mixture of control structures and other statements; it would have a much higher cyclomatic complexity rating.

The cyclomatic complexity metric is useful because it's an objective measure, and it's possible to write a program to compute this value. Its drawback is that the bulk size of a program is irrelevant; that is, it treats a single `printf` statement the same as 1,000 `printf` statements in a row, even though the second version clearly requires more work (even if that extra work is just a bunch of cut-and-paste operations).

2.6.7 Other Metrics

There's no shortage of metrics we could devise to measure some facet of programmer productivity. One common metric is to count the number of operators in a program. This metric recognizes and adjusts for the fact that some statements (including those that don't involve control paths) are more complex than others, taking more time to write, test, and debug. Another metric is to count the number of tokens (such as identifiers, reserved words, operators, constants, and punctuation) in a program. No matter the metric, though, it will have shortcomings.

Many people attempt to use a combination of metrics (such as line count multiplied by cyclomatic complexity and operator count) to create a more "multidimensional" metric that better measures the amount of work involved in producing a bit of code. Unfortunately, as the complexity of the metric increases, it becomes more difficult to use on a given project. LOC has been successful because you can use the Unix `wc` (word count) utility, which also counts lines, to get a quick idea of program size. Computing a value for one of these other metrics usually requires a specialized, language-dependent application (assuming the metric is automatable). For this reason, although people have proposed a large number of metrics, few have become as universally popular as LOC.

2.6.8 The Problem with Metrics

Metrics that roughly measure the amount of source code for a project provide a good indication of the time spent on a project if we assume that each line or statement in the program takes some average amount of time to write, but only a tenuous relationship exists between lines of code (or

statements) and the work accomplished. Unfortunately, metrics measure some physical attributes of the program but rarely measure what we're really interested in knowing: the intellectual effort needed to write the code in the first place.

Another failure of almost every metric is that they all assume that more work produces more (or more complex) code. This is not always true. For example, a great programmer will often expend effort to refactor their code, making it smaller and less complex. In this case, more work produces less code (and less complex code).

Metrics also fail to consider environmental issues concerning the code. For example, are 10 lines of code written for a bare-metal embedded device equivalent to 10 lines of code written for a SQL database application?

All these metrics fail to consider the learning curve for certain projects. Are 10 lines of Windows device driver code equivalent to 10 lines of Java code in a web applet? The LOC values for these two projects are incomparable.

Ultimately, most metrics fail because they measure the *wrong thing*. They measure the *amount of code* a programmer produces rather than the programmer's overall *contribution to the complete project* (productivity). For example, one programmer could use a single statement to accomplish a task (such as a standard library call), whereas a second programmer could write several hundred lines of code to accomplish the same task. Most metrics would suggest the second programmer is the more productive of the two.

For these very reasons, even the most complex software metrics currently in use have fundamental flaws that prevent them from being completely effective. Therefore, choosing a "better" metric often produces results that are no better than using a "flawed" metric. This is yet another reason the LOC metric continues to be so popular (and why this book uses it). It's an amazingly bad metric, but it's not a whole lot worse than many of the other existing metrics, and it's very easy to compute without writing special software.

2.7 How Do We Beat 10 Lines per Day?

Early texts on software engineering claim that a programmer on a major product produces an average of ten lines of code per day. In a 1977 article, Walston and Felix report about 274 LOC per month per developer.[8] Both numbers describe the production of debugged and documented code *over the lifetime* of that product (that is, LOC divided by the amount of time all the programmers spent on the product from first release to retirement), rather than simply time spent writing code from day to day. Even so, the numbers seem low. Why?

At the start of a project, programmers might quickly crank out 1,000 lines of code per day, then slow down to research a solution to a particular

8. Claude E. Walston and Charles P. Felix, "A Method of Programming Measurement and Estimation," *IBM Systems Journal* 16, no. 1 (1977): 54–73.

portion of the project, test the code, fix bugs, rewrite half their code, and then document their work. By the product's first release, productivity has dropped tenfold since that first day or two: from 1,000 LOC per day to fewer than 100. Once the first release is out the door, work generally begins on the second release, then the third, and so on. Over the product's lifetime, several different developers will probably work on the code. By the time the project is retired, it has been rewritten several times (a tremendous loss in productivity), and several programmers have spent valuable time learning how the code operates (also sapping their productivity). Therefore, over the lifetime of the product, programmer productivity is down to 10 LOC per day.

One of the most important results from software engineering productivity studies is that the best way to improve productivity is not by inventing some scheme that allows programmers to write twice as many lines of code per unit time, but to *reduce the time wasted on debugging, testing, documenting, and rewriting the code, and on educating new programmers about the code once the first version exists.* To reduce that loss, it's much easier to improve the processes that programmers use on the project than it is to train them to write twice as much code per unit time. Software engineering has always recognized this problem and has attempted to solve it by reducing the time spent by all programmers. Personal software engineering's goal is to reduce the time spent by individual programmers on their portion of the project.

2.8 Estimating Development Time

As noted earlier, while productivity is of interest to management for awarding bonuses, pay raises, or verbal praise, the real purpose for tracking it is to predict development times on future projects. Past results don't guarantee future performance, so you also need to know how to estimate a project schedule (or at least the schedule for your portion of a project). As an individual software engineer, you typically don't have the background, education, and experience to determine what goes into a schedule, so you should meet with your project manager, have them explain what needs to be considered in the schedule (which is more than just the time required to write code), and then build the estimate that way. Though all the details needed to properly estimate a project are beyond the scope of this book (see "For More Information" on page 37 for suggested resources), it's worthwhile to briefly describe how development time estimates differ depending on whether you're working on a small, medium, or large project, or just a portion of a project.

2.8.1 Estimating Small Project Development Time

By definition, a small project is one that a single engineer works on. The major influence on the project schedule will be the ability and productivity of that software engineer.

Estimating development time for small projects is much easier and more accurate than for larger projects. Small projects won't involve

parallel development, and the schedule only has to consider a single developer's productivity.

Without question, the first step in estimating the development time for a small project is to identify and understand all the work that needs to be done. If some parts of the project are undefined at that point, you introduce considerable error in the schedule when the undefined components inevitably take far more time than you imagined.

For estimating a project's completion time, the design documentation is the most important part of the project. Without a detailed design, it's impossible to know what subtasks make up the project and how much time each will take to accomplish. Once you've broken down the project into suitably sized subtasks (a suitable size is where it's clear how long it will take to complete), all you need to do is add the times for all the subtasks to produce a decent first estimate.

One of the biggest mistakes people make when estimating small projects, however, is that they add the times for the subtasks and call that their schedule, forgetting to include time for meetings, phone calls, emails, and other administrative tasks. They also forget to add in testing time, plus time to correct (and retest) the software when defects are found. Because it's difficult to estimate how many defects will be in the software, and thus how much time it will take to resolve them, most managers scale a schedule's first approximation by a factor of 2 to 4. Assuming the programmer (team) maintains reasonable productivity on the project, this formula produces a good estimate for a small project.

2.8.2 Estimating Medium and Large Project Development Time

Conceptually, medium and large projects consist of many small projects (assigned to individual team members) that combine to form the final result. So a first approximation on a large project schedule is to break it down into a bunch of smaller projects, develop estimates for each of those subprojects, and then combine (add) the estimates. It's sort of a bigger version of the small project estimate. Unfortunately, in real life, this form of estimate is fraught with error.

The first problem is that medium and large projects introduce problems that don't exist in small projects. A small project typically has one engineer, and, as noted previously, the schedule completely depends upon that person's productivity and availability. In a larger project, multiple people (including many nonengineers) affect the estimated schedule. One software engineer who has a key piece of knowledge might be on vacation or sick for several days, holding up a second engineer who needs that information to make progress. Engineers on larger projects usually have several meetings a week (unaccounted for in most schedules) that take them offline—that is, they're not programming—for several hours. The team composition can change on large projects; some experienced programmers leave and someone else has to pick up and learn the subtasks, and new programmers join the project and need time to get up to speed. Sometimes even getting a computer workstation for a new hire can take weeks (for

example, in a large company with a bureaucratic IT department). Waiting for software tools to be purchased, hardware to be developed, and support from other parts of the organization also creates scheduling problems. The list goes on and on. Few schedule estimates can accurately predict how the time will be consumed in these myriad ways.

Ultimately, creating medium and large project schedule estimates involves four tasks: breaking down the project into smaller projects, running the small project estimations on those, adding in time for integration testing and debugging (that is, combining the small tasks and getting them to work properly together), and then applying a multiplicative factor to that sum. They're not precise, but they're about as good as it gets today.

2.8.3 Problems with Estimating Development Time

Because project schedule estimates involve predicting a development team's future performance, few people believe that a projected schedule will be totally accurate. However, typical software development schedule projections are especially bad. Here are some of the reasons why:

They're research and development projects. R&D projects involve doing something you've never done before. They require a research phase during which the development team analyzes the problem and tries to determine solutions. Usually, there's no way to predict how long the research phase will take.

Management has preconceived schedules. Typically, the marketing department decides that it wants to have a product to sell by a certain date, and management creates project schedules by working backward from that date. Before asking the programming team for their time estimates of the subtasks, management already has some preconceived notions about how long each task should take.

The team's done this before. It's common for management to assume that if you've done something before, it will be easier the second time around (and therefore will take less time). In certain cases, there's an element of truth to this: if a team works on an R&D project, it will be easier to do a second time because they only have to do the development and can skip (at least most of) the research. However, the assumption that a project is always easier the second time is rarely correct.

There isn't enough time or money. In many cases, management sets some sort of monetary or time limit within with a project must be completed or else it will be canceled. That's the *wrong* thing to say to someone whose paycheck depends on the project moving forward. If given a choice between saying, "Yes, we can meet that schedule," or looking for a new job, most people—even knowing the odds are against them—will opt for the first.

Programmers overstate their efficiency. Sometimes when a software engineer is asked if they can complete a project within a certain timeframe, they don't lie about how long it will take, but instead make optimistic estimates of their performance—which rarely hold up during

the actual work. When asked how much they can produce when *really pushed*, most software engineers give a figure that represents their maximum output ever achieved over a short period of time (for example, while working in "crisis mode" and putting in 60–70 hours per week) and don't consider unexpected hindrances (such as a really nasty bug that comes along).

Schedules rely on extra hours. Management (and engineers) often assume that programmers can always put in "a few extra hours" when the schedule starts to slip. As a result, schedules tend to be more aggressive than they should be (ignoring the negative repercussions of having engineers put in massive overtime).

Engineers are like building blocks. A common problem with project schedules is that management assumes it can add programmers to a project to achieve an earlier release date. However, as mentioned earlier, this isn't necessarily true. You can't add or remove engineers from a project and expect a proportional change in the project schedule.

Subproject estimates are inaccurate. Realistic project schedules are developed in a top-down fashion. The whole project is divided into smaller subprojects. Then those subprojects are divided into sets of sub-subprojects, and so on until the subproject size is so small that someone can accurately predict the time needed for each tiny part. However, there are three challenges with this approach:

- Being willing to put in the effort to create a schedule this way (that is, to provide a correct and accurate top-down analysis of the project)
- Obtaining accurate estimates for the tiny subprojects (particularly from software engineers who may not have the appropriate management training to understand what must go into their schedule estimates)
- Accepting the results the schedule predicts

2.9 Crisis Mode Project Management

Despite the best intentions of everyone involved, many projects fall significantly behind schedule and management must accelerate development to meet some important milestone. To achieve the deadline, engineers often are expected to put in more time each week to reduce the (real time) delivery date. When this occurs, the project is said to be in "crisis mode."

Crisis mode engineering can be effective for short bursts to handle (rapidly) approaching deadlines, but in general, crisis mode is never that effective, and results in lower productivity, because most people have things to take care of outside of work, and need time off to rest, decompress, and allow their brains to sort out all the problems they've been collecting while putting in long hours. Working while you're tired leads to mistakes that often take far more time to correct later on. It's more efficient in the long run to forgo the crisis mode and stick to 40-hour weeks.

The best way to handle crisis mode schedules is to add milestones throughout the project to generate a series of "small crises" rather than one big crisis at the end. Putting in an extra day or a couple of long days once a month is infinitely better than having to put in several seven-day weeks at the end of the project. Working one or two 16-hour days to meet a deadline won't adversely affect the quality of your life or lead you to the point of exhaustion.

Beyond the health and productivity issues, operating in crisis mode can cause scheduling, ethical, and legal problems:

- A poor schedule can affect future projects as well. If you work 60-hour weeks, management will assume that future projects can also be done in the same amount of (real) time, expecting this pace from you in the future without any additional compensation.
- Technical staff turnover is high on projects that operate for lengthy periods of time in crisis mode, further reducing team productivity.
- There is also the legal issue of putting in lots of extra hours without being paid overtime. Several high-profile lawsuits in the video game industry have shown that engineers are entitled to overtime pay (they are not *salary exempt* employees). Even if your company can survive such lawsuits, the rules for time reporting, administrative overhead, and work schedules will become much more restrictive, leading to productivity drops.

Again, operating in crisis mode can help you meet certain deadlines if managed properly. But the best solution is to work out better schedules to avoid crisis mode altogether.

2.10 How to Be More Productive

This chapter has spent considerable time defining productivity and metrics for measuring it. But it hasn't devoted much time to describing how a programmer can increase their productivity to become a great programmer. Whole books can be (and have been) written on this subject. This section provides an overview of techniques you can use to improve your productivity on individual and team projects.

2.10.1 Choose Software Development Tools Wisely

As a software developer, you'll spend most of your time working with software development tools, and the quality of your tools can have a huge impact on your productivity. Sadly, the main criterion for selecting development tools seems to be familiarity with a tool rather than the applicability of the tool to the current project.

Keep in mind when choosing your tools at the start of the project that you'll probably have to live with them for the life of the project (and maybe beyond that). For example, once you start using a defect tracking system, it might be very difficult to switch to a different one because of

incompatible database file formats; the same goes for source code control systems. Fortunately, software development tools (especially IDEs) are relatively mature these days, and a large number of them are interoperable, so it's hard to make a bad choice. Still, careful thought at the beginning of a project can spare you a lot of problems down the road.

The most significant tool choice for a software development project is which programming language and which compilers/interpreters/translators to use. Optimal language choice is a difficult problem to solve. It's easy to justify some programming language because you're familiar with it and you won't lose productivity learning it; however, future engineers new to the product might be far less productive because they're learning the programming language while trying to maintain the code. Furthermore, some language choices could streamline the development process, sufficiently improving productivity to make up for lost time learning the language. As noted earlier, a poor language choice could result in wasted development time using that language until it becomes clear that it is unsuitable for the project and you have to start over.

Compiler performance (how many lines per second it takes to process a common source file) can have a huge impact on your productivity. If your compiler takes two seconds to compile an average source file rather than two minutes, you'll probably be far more productive using the faster compiler (though the faster compiler might be missing some features that completely kill your productivity in other ways). The less time your tools take to process your code, the more time you'll have for designing, testing, debugging, and polishing your code.

It's also important to use a set of tools that work well together. Today, we take for granted *integrated development environments (IDEs)*, which combine an editor, compiler, debugger, source code browser, and other tools into a single program. Being able to quickly make small changes in an editor, recompile a source code module, and run the result in a debugger all within the same window onscreen provides a phenomenal boost in productivity.

However, you'll often have to work on parts of your project outside the IDE. For example, some IDEs don't support source code control facilities or defect tracking directly in the IDE (though many do). Most IDEs don't provide a word processor for writing documentation, nor do they provide simple database or spreadsheet capabilities to maintain requirements lists, design documentation, or user documentation. Most likely, you'll have to use a few programs outside your IDE—word processing, spreadsheet, drawing/graphics, web design, and database programs, to name a few—to do all the work needed on your project.

Running programs outside an IDE isn't a problem. Just make sure the applications you choose are compatible with your development process and the files your IDE produces (and vice versa). Your productivity will decrease if you must constantly run a translator program when moving files between your IDE and an external application.

Can I recommend tools for you to use? No way. There are too many projects with different needs to even consider such suggestions here. My recommendation is to simply be aware of the issues at the start of the project.

But one recommendation I *can* make is to avoid the "Gee whiz, why don't we try this new technology" approach when choosing a development tool. Discovering that a development tool can't do the job after spending six months working with it (and basing your source code on it) can be disastrous. Evaluate your tools apart from your product development, and work in new tools only after you're confident that they'll work for you. A classic example of this is Apple's Swift programming language. Until Swift v5.0 was released (about four years after Swift was first introduced), using Swift was an exercise in frustration. Every year Apple would release a new version that was source code–incompatible with earlier releases, forcing you to go back and change old programs. In addition, many features were missing in early versions of the language, and several features weren't quite ready for "prime time." By version 5.0 (released as this book was being written), the language seems relatively stable. However, the poor souls who jumped on the Swift bandwagon early on paid the price for the immature development of the language.[9]

Sadly, you don't get to choose the development tools on many projects. That decision is an edict from on high, or you inherit tools from earlier products. Complaining about it wastes time and energy, and reduces your productivity. Instead, make the best of the tool set you have, and become an expert at using it.

2.10.2 Manage Overhead

On any project, we can divide the work into two categories: work that is directly associated with the project (such as writing lines of code or documentation for the project) and work that is indirectly related to the project. Indirect activities include meetings, reading and replying to emails, filling out time cards, and updating schedules. These are *overhead* activities: they add time and money to a project's cost but don't directly contribute to getting the work done.

By following Watts S. Humphrey's *Personal Software Engineering* guidelines, you can track where you spend your time during a project and easily see how much is spent directly on the project versus on overhead activities. If your overhead climbs above 10 percent of your total time, reconsider your daily activities. Try to decrease or combine those activities to reduce their impact on your productivity. If you don't track your time outside the project, you'll miss the opportunity to improve your productivity by managing overhead.

2.10.3 Set Clear Goals and Milestones

It's a natural human tendency to relax when no deadlines are looming, and then go into "hypermode" as one approaches. Without goals to achieve, very little productive work ever gets done. Without deadlines to meet, rarely is there any motivation to achieve those goals in a timely manner.

---

9. Today, I don't have a problem recommending Swift. It's a great language, and version 5.0 and later seem relatively stable and reliable. It's moved beyond the "Gee whiz, ain't this a great new language" stage and is now a valid software development tool for real projects.

Therefore, to improve your productivity, be sure to have clear goals and subgoals, and attach hard *milestones* to them.

From a project management viewpoint, a milestone is a marker in a project that determines how far work has progressed. A good manager always sets goals and milestones in the project schedule. However, few schedules provide useful goals for individual programmers. This is where personal software engineering comes into play. To become a superproductive programmer, micromanage your own goals and milestones on your (portion of the) project. Simple goals, such as "I'll finish this function before I take lunch" or "I'll find the source of this error before going home today" can keep you focused. Larger goals, such as "I'll finish testing this module by next Tuesday" or "I'll run at least 20 test procedures today" help you gauge your productivity and determine if you're achieving what you want.

2.10.4 Practice Self-Motivation

Improving your productivity is all about attitude. Although others can help you manage your time better and aid you when you're stuck, the bottom line is that you must have the initiative to better yourself. Always be conscious of your pace and constantly strive to improve your performance. By keeping track of your goals, efforts, and progress, you'll know when you need to "psych yourself up" and work harder to improve your productivity.

A lack of motivation can be one of the greatest impediments to your productivity. If your attitude is "Ugh, I have to work on *that* today," it will probably take you longer to complete the task than if your attitude is "Oh! This is the best part! This will be fun!"

Of course, not every task you work on will be interesting and fun. This is one area where *personal* software engineering kicks in. If you want to maintain higher-than-average productivity, you need to have considerable self-motivation when a project makes you feel "less than motivated." Try to create reasons to make the work appealing. For example, create mini-challenges for yourself and reward yourself for achieving them. A productive software engineer constantly practices self-motivation: the longer you remain motivated to do a project, the more productive you'll be.

2.10.5 Focus and Eliminate Distractions

Staying focused on a task and eliminating distractions is another way to dramatically improve your productivity. Be "in the zone." Software engineers operating this way are more productive than those who are mentally multitasking. To increase your productivity, concentrate on a single task for as long as possible.

Focusing on a task is easiest in a quiet environment without any visual stimulation (other than your display screen). Sometimes, work environments aren't conducive to an extreme focus. In such cases, putting on headphones and playing background music might help remove the distractions. If music is too distracting, try listening to white noise; there are several white noise apps available online.

Whenever you're interrupted in the middle of a task, it will take time to get back in the zone. In fact, it could take as long as half an hour to become fully refocused on your work. When you need to focus and complete a task, put up a sign saying that you should only be interrupted for urgent business, or post "office hours"—times when you can be interrupted—near your workstation; for example, you could allow interruptions at the top of the hour for five minutes. Saving your coworkers 10 minutes by answering a question they could figure out themselves could cost you half an hour of productivity. You do have to work as part of the team and be a good teammate; however, it's just as important to ensure that excessive team interactions don't impair your (and others') productivity.

During a typical workday, there will be many scheduled interruptions: meal breaks, rest breaks, meetings, administrative sessions (for example, handling emails and time accounting), and more. If possible, try to schedule other interruptions around these events. For example, turn off any email alerts; answering emails within a few seconds is *rarely* imperative, and someone can find you in person or call you if it's an emergency. Set an alarm to remind you to check email at fixed times if people do expect quick responses from you (ditto with text messages and other interruptions). If you can get away with it, consider silencing your phone if you get a lot of nonurgent phone calls, checking your messages every hour or so during your breaks. What works for you depends on your personal and professional life. But the fewer interruptions you have, the more productive you'll become.

2.10.6 If You're Bored, Work on Something Else

Sometimes, no matter how self-motivated you are, you'll be bored with what you're working on and have trouble focusing; your productivity will plummet. If you can't get into the zone and focus on the task, take a break from it and work on something else. Don't use boredom as an excuse to flitter from task to task without accomplishing much. But when you're really stuck and can't move forward, switch to something you can be productive doing.

2.10.7 Be as Self-Sufficient as Possible

As much as possible, you should try to handle all tasks assigned to you. This won't improve your productivity; however, if you're constantly seeking help from other engineers, you might be damaging their productivity (remember, they need to stay focused and avoid interruptions, too).

If you're working on a task that requires more knowledge than you currently possess, and you don't want to constantly interrupt other engineers, you have a few options:

- Spend time educating yourself so you can do the task. Although you might hurt your short-term productivity, the knowledge you gain will help you with similar future tasks.
- Meet with your manager and explain the problems you're having. Discuss the possibility of their reassigning the task to someone more experienced and assigning you a task you're better able to handle.

- Arrange with your manager to schedule a meeting to get help from other engineers at a time that won't impact their productivity as much (for example, at the beginning of the workday).

2.10.8 Recognize When You Need Help

You can take the self-supporting attitude a little too far. You can spend an inordinate amount of time working on a problem that a teammate could solve in just a few minutes. One aspect of being a great programmer is recognizing when you're stuck and need help to move forward. When you're stuck, the best approach is to set a timer alarm. After some number of minutes, hours, or even days being stuck on the problem, seek help. If you know who to ask for help, seek that help directly. If you're not sure, talk to your manager. Most likely, your manager can direct you to the right person so you don't interrupt others who wouldn't be able to help you anyway.

Team meetings (daily or weekly) are a good place to seek help from team members. If you have several tasks on your plate and you're stuck on one particular task, set it aside, work on other tasks (if possible), and save your questions for a team meeting. If you run out of work before a meeting, ask your manager to keep you busy so you don't have to interrupt anyone. Further, while working on other tasks, the solution just might come to you.

2.10.9 Overcome Poor Morale

Nothing can kill a project faster than an infestation of bad morale among team members. Here are some suggestions to help you overcome poor morale:

- Understand the business value of your project. By learning about, or reminding yourself of, the real-world practical applications of your project, you'll become more invested and interested in the project.
- Take ownership and responsibility for (your portion of) a project. When you own the project, your pride and reputation are on the line. Regardless of what else might happen, ensure that you can always talk about the contributions you made to the project.
- Avoid becoming emotionally invested in those project components over which you have no control. For example, if management has made some poor decisions that affect the project's schedule or design, work as best as you can within those confines. Don't just sit around thinking bad thoughts about management when you could be putting that effort into solving problems.
- If you're faced with personality differences that are creating morale problems, discuss those issues with your manager and other affected personnel. Communication is key. Allowing problems to continue will only lead to larger morale problems down the road.
- Always be on the lookout for situations and attitudes that could damage morale. Once morale on a project begins to decline, it's often very difficult to restore what was lost. The sooner you deal with morale issues, the easier it will be to resolve them.

Sometimes, financial, resource, or personnel issues decrease morale among the project's participants. Your job as a great programmer is to step in, rise above the issues, and continue writing great code—and encourage those on the project to do the same. This isn't always easy, but no one ever said that becoming a great programmer was easy.

2.11 For More Information

Bellinger, Gene. "Project Systems." Systems Thinking, 2004. *http://systems-thinking.org/prjsys/prjsys.htm.*

Heller, Robert, and Tim Hindle. *Essential Managers: Managing Meetings.* New York: DK Publishing, 1998.

Humphrey, Watts S. *A Discipline for Software Engineering.* Upper Saddle River, NJ: Addison-Wesley Professional, 1994.

Kerzner, Harold. *Project Management: A Systems Approach to Planning, Scheduling, and Controlling.* Hoboken, NJ: Wiley, 2003.

Lencioni, Patrick. *Death by Meeting: A Leadership Fable . . . About Solving the Most Painful Problem in Business.* San Francisco: Jossey-Bass, 2004.

Levasseur, Robert E. *Breakthrough Business Meetings: Shared Leadership in Action.* Lincoln, NE: iUniverse.com, Inc., 2000.

Lewis, James P. *Project Planning, Scheduling, and Control.* New York: McGraw-Hill, 2000.

McConnell, Steve. *Software Project Survival Guide.* Redmond, WA: Microsoft Press, 1997.

Mochal, Tom. "Get Creative to Motivate Project Teams When Morale Is Low." TechRepublic, September, 21, 2001. *http://www.techrepublic.com/article/get-creative-to-motivate-project-teams-when-morale-is-low/.*

Wysocki, Robert K., and Rudd McGary. *Effective Project Management.* Indianapolis: Wiley, 2003.