Chapter 2. Standing On Principle

The only freedom that has any meaning is the freedom to proceed differently from the way your manager would have proceeded. This is true in a broader sense, too: The right to be right (in your manager's eyes or in your government's eyes) is irrelevant; it's only the right to be wrong that makes you free.

—Tom DeMarco and Timothy Lister, Peopleware

You may be asked to cut corners on the quality of the work you do or the way you do it. Sometimes, the person making the request is a manager who doesn't appreciate the value of the quality activities you practice, or someone who feels budget and schedule pressures you may not. Sometimes, it is a customer who wants you to address only her specific needs, even though you have identified an opportunity to generalize beyond her requirements to provide enhanced value to a broader user community within your own company. Customers and managers alike may press you to skip key engineering steps from time to time. It would be nice to think we are beyond the stage of "don't bother with those specs, start writing code," but this call of the software dinosaur still echoes throughout the land. It's not easy to resist these pressures from the people who pay the bill or your salary. Sometimes you have no choice: Do what is asked of you, or head out the door. But people working in a quality-oriented software engineering culture will do their best to follow established and intelligent practices in times of crisis, as well as in normal times. Developers and managers must adopt personal standards that motivate them to stick to their software process guns in the heat of battle. You might need to temper this idealism with the reality of keeping your job, but start from the position of quality being paramount. Whenever a situation arises that calls for a value judgment, an ethical decision, or a choice between doing what is right and doing what you are asked to do, remember this: Never let your boss or your customer talk you into doing a bad job.

Integrity and Intelligence: With Customers

Perhaps it sounds like heresy, but the customer is not always right; however, the customer always has a point. Too often, software developers incorporate every feature requested by a customer into a requirements specification without regard to how much effort it will take to implement, or whether it is truly necessary to achieve the customer's objectives. While you must respect the attitudes and requests made by every customer, don't be blinded by the notion that you are hearing the "voice of the customer" and therefore must do whatever that voice says. Much of the value that a systems analyst adds to the application development process comes from looking for solutions that address the true user needs with creativity, efficiency, and thoughtfulness. Some years ago, our software group at Kodak developed a PC-based system for controlling some lab equipment. The customers asked us to write a pop-up calculator program the users could invoke while they were running the program (this was before the advent of windowing systems that made such features commonplace). After a little thought, we concluded it would be cheaper simply to buy a basic calculator for each of the hundred-odd anticipated users and slap one on the side of each monitor with a piece of Velcro ® . The customer need was for a calculator function—it didn't really have to be built into the

software. That was just a cute, but unnecessarily expensive, solution the customer envisioned. There are two lessons here:

Look past the customer's request for a feature to find the underlying real need; and, look for simple and cheap solutions for the real need, not just the solution that the customer initially requested or one that looks neat.

In a healthy software engineering culture, developers are empowered to take the actions they feel will best meet the company's objectives, in the context of project goals, customer needs, and resource realities. Sometimes, this means driving a paradigm shift in the way applications are built or delivered: new methods, new tools, new architectures, new platforms.

For example, one Kodak software group was asked to reengineer a large suite of scientific programs, written over a span of many years in Fortran on an IBM mainframe. We wished to migrate this into an object-oriented architecture in C++ running on a UNIX ® platform. However, the current computing infrastructure would not allow the entire user base for this application suite to have direct access to the UNIX server right away. So, we devised an unconventional transitional architecture that helped move us toward the future, while keeping one foot firmly planted in the current reality.

This interim solution was an "inverted" client/server architecture, with in the early stages of migrating toward a more open systems architecture. The other solutions that were proposed were not consistent with the long-term strategic objectives of this organization. They would have resulted in software that few users could access immediately, or they would have further tied us to the older technology of the mainframe. The paradigm shift embodied in the approach we took addressed real user needs creatively.

Everyone is subject to tunnel vision. Customers think in terms of what functions they personally will use in the application, not those that should be included because of their potential value to other users. We ran into this situation on another project, which was requested by one area of the research laboratories at Kodak. Not surprisingly, the primary customer representative focused on the needs of the community she represented. However, the analyst spotted several places where the application could be made more generally useful without a lot of extra effort, thereby meeting the needs of other research departments.

This customer rep was reluctant to have us spend the extra time building in these generalizations. However, we felt they would enable Kodak to leverage the investment in this custom software over a broader internal user community. So, we went ahead and designed the system to incorporate these generalizations. In this case, we were willing to incur the unhappiness of the primary customer, because we were doing the right thing for the company. If her department had declined to pay for the extra work required for the more general solution, we would have funded it in some other appropriate way. The lesson here is

Look beyond the local interests of the primary customers to see whether you can easily generalize an application to meet a broader need.

This rationale also applies to the identification of potentially reusable components in the course of application design. It will always take more time to design, implement, and verify a reusable component than to code a custom solution for one project. Think of designing for reuse as an investment you can cash in on multiple future projects, for the good of the

company. Don't let resistance by your customer or your manager inhibit you from making smart strategic decisions.

Occasionally, you may encounter a customer who is a self-styled software engineer and who therefore wishes to provide a lot of technical input into how you design his application. This is usually a case of someone having just enough knowledge to be dangerous. The project leader must resist letting such customers become technically involved in design and implementation (unless, of course, the customer really does know what he's doing). Focus his energy on the requirements, or get him involved with testing and writing user aids, but do not let the customer drive the technical aspects of the project. The rule to remember is

Make sure you know where the customer's software expertise stops. Do not allow the customer to intimidate you into making poor design decisions.

Developers must base their relationships with customers on mutual trust and respect. We expect our customers to be forthright with us about their requirements and constraints, and they expect us to be straight with them have to be kept informed, even if they don't want to hear it.

I know of one developer who had originally promised to deliver a new application in October. He fell behind on the project, and as of September, it was clear that several months of work remained to be done. Unfortunately, the developer had not updated his delivery estimate with the customers: They still expected the system to be ready in October. We had to do some quick expectation management. As you might expect, the customers weren't happy about the "sudden" change in schedule. This developer didn't think about this principle:

Be honest with your customers. The project stakeholders are entitled to accurate information, whether or not it is good news.

Do you say you will do something and then never get around to it? I feel this reflects a lack of personal integrity. Follow through on the commitments you have made. However, if something changes—you decided you won't do it, or you are unable to do it on time for whatever reason—talk about it! There are many reasons, some good and others not so good, why you might not be able to fulfill a commitment, but it is irresponsible not to notify those affected by your decision. So long as you are open about a problem, you can work out a solution. The message is this:

If you agree to a commitment, the other people involved expect you to do it. If anything changes, let them know as soon as you can, so you can work out an alternative.

As a manager, I told my team members it was their right and their responsibility to tell me when their backlog pile became too high. Then it was my responsibility to help them deal with the pile. When I asked them to take on a new task, it was completely acceptable for them to reply, "Sure, I can do that, Karl. What would you like me to stop doing to free up the necessary time?" Until they told me the pile was too high, I had no way of knowing that this was the case. Naomi Karten suggests reasons and ways to "just say whoa" [Karten, 1994].

Integrity and Intelligence: With Managers

A colleague was once asked by a manager to estimate the delivery time for a planned large software application. He replied, "Two years." The manager said, "No, that's too long. How about six months?" My colleague's response was, "Okay." Wrong answer! What changed in

the five seconds that elapsed between his first estimate of two years, and his second statement, agreeing to a schedule only one-quarter as long? Nothing, except that he thought the manager wanted to believe that six months was feasible. The project did not suddenly shrink by 75 percent, nor did the available staff increase four-fold or instantly become four times as productive. The project was not reestimated based on an improved algorithm. My colleague simply said what the manager wanted to hear, undermining his own credibility as a software estimator and project leader. To no one's surprise, the project extended beyond two years. Agreeing to unattainable commitments is unprofessional and unfair before caving in.

1. Explain your estimating method to the manager, and ask on what basis the manager's estimate is smaller. The manager may not really have an estimate, but he has a goal. You may not have an estimate, either, only a guess. It's harder to argue with an estimate based on some analytical, quantitative process than with a number pulled out of thin air. Historical metrics data can help build your case.
2. If you can't provide an accurate estimate because there are no written requirements, offer to provide a more precise estimate after some initial exploration of the project scope and general user requirements.
3. Point out that an estimate made very early in a project can be off by 80 percent or more. Present a range of estimates: best case, most likely, and worst case, with the approximate probability of meeting each one. Presenting a single estimate at the beginning of a sizable project sets an expectation that will persist in people's minds long after the original requirements and assumptions have drastically changed.
4. Negotiate for a larger team, fewer features, phased delivery, or reduced quality as ways to achieve an aggressively accelerated schedule. Make sure the stakeholders understand these trade-offs: They will not get something for nothing.
5. Redo your estimate with some different assumptions of project size, resources, or other factors, to see how close you can come to the manager's shorter goal. Make sure the assumptions are clearly communicated to everyone involved in the decision-making.
6. Make a counteroffer, showing the manager what fraction of the system's functionality realistically could be delivered in six months.

Concealing a project's scope from management is even more irresponsible than agreeing to a schedule you cannot possibly meet. One project leader was leading a long-term reengineering effort, but she never prepared a detailed project plan. She was afraid if her manager found out how extensive (and expensive) the project really was, the manager would squelch it. Both this project leader and her manager acted inappropriately. The project leader should have been forthright about defining the scope of the project and managing it properly, while the manager should have insisted on seeing a project plan and having some accountability. One purpose of project planning is to identify all the tasks that have to be performed to make the project a success. Pretending those tasks aren't there doesn't make them go away.

The Five Dimensions of a Software Project

There are five dimensions that must be managed on a software project: features, quality, cost, schedule, and staff (shown in Fig. 2.1). These dimensions are not all independent. For example, if you add staff, the schedule may be shortened (although not necessarily), and the cost may increase. A more common trade-off is to shorten the schedule or add features, and

sacrifice quality. The trade-offs among these five dimensions achieve the key project objectives.

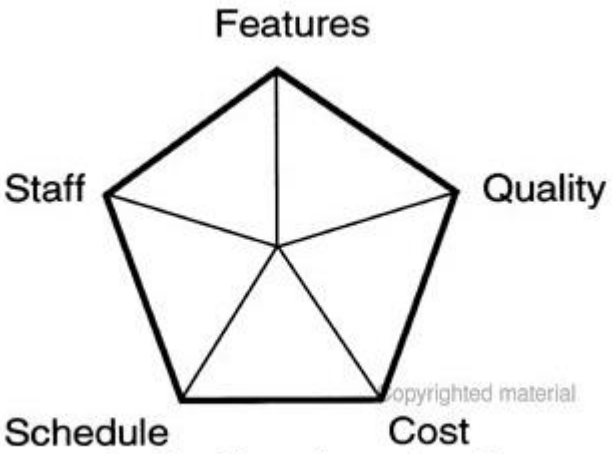

Figure 2.1: The five dimensions of a software project.

Each of these five dimensions can take one of three roles on any given project: a driver, a constraint, or a degree of freedom. A driver is a key objective of the project. For a product that must ship on time to meet a marketing window of opportunity, schedule is a driver. Commercial desktop software, such as word processors and spreadsheets, are often created with features as the driver.

A constraint is a limiting factor that is not within the project leader's control. If a team of immutably fixed size is assigned to a project, staff becomes a constraint. Cost is a constraint on a project being done under a fixed-price contract, while quality will be a constraint for a project to develop software that runs a piece of medical equipment or an airplane's flight control system. Sometimes, you can regard cost as either a constraint or a driver, since it could be both a primary objective and a limiting factor. Similarly, a specified feature set may be the primary driver of the project, but you could view it as a constraint if the feature set is not negotiable. Any project dimension that is neither a driver nor a constraint becomes a degree of freedom. These are factors that the project leader can adjust and balance to some extent, to achieve the overall project objectives. For example, on some internal information system projects, the drivers are features and quality, and staff is a constraint, so the degrees of freedom become schedule and cost. The implication for this profile is that the features demanded by the customers will all be included, but the delivery time for the product may be later than desired. An important aspect of this model is not which of the five dimensions turn out to be drivers or constraints on any given project, but that the relative priorities of the dimensions be negotiated in advance by the project team, the customers, and management. All five cannot be drivers, and all five cannot be constraints. This negotiation process helps to define the rules and bounds of the project. As in most games, we can play according to any set of rules, but all of the players must understand and agree to the rules that are in effect at any particular time. A way to classify each dimension into one of the three categories is to think of the amount of flexibility the project leader has with respect to that dimension. A constraint gives the project leader virtually no flexibility, a driver has low flexibility, and a degree of freedom provides a wider latitude to balance that dimension against the other four. A

graphical way to depict this is to use a Kiviat diagram, which allows us to plot several values (five, in this case) as an irregularly shaped polygon on a set of normalized axes. The position of each point on its axis indicates the relative degree of flexibility of that dimension for a particular project, group's recent applications. This project was constrained to a fixed staff size, so the value plotted on the staff axis is 0. The project was driven to meet a desired schedule, so the point on the schedule axis also has a low value. The project had varying amounts of flexibility around the features that would be incorporated into the initial release, the product quality, and the latitude for cost overruns. Therefore, the values for these degrees of freedom are higher on their axes.

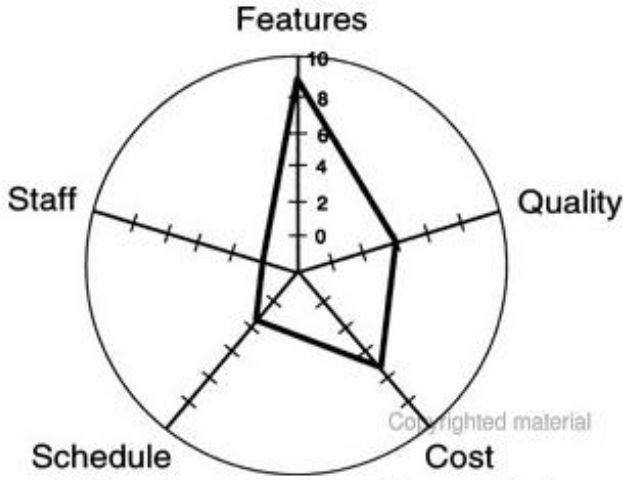

Figure 2.2: Flexibility diagram for an internal information system.

Figure 2.3 illustrates a flexibility diagram for a hypothetical project for which quality is a driver and the schedule shows the greatest latitude. The profile for a highly competitive commercial software product might look like Fig. 2.4 , in which a specified feature set must be included (a constraint), the schedule is constrained to a specified ship date, and the quality is just whatever it turns out to be.

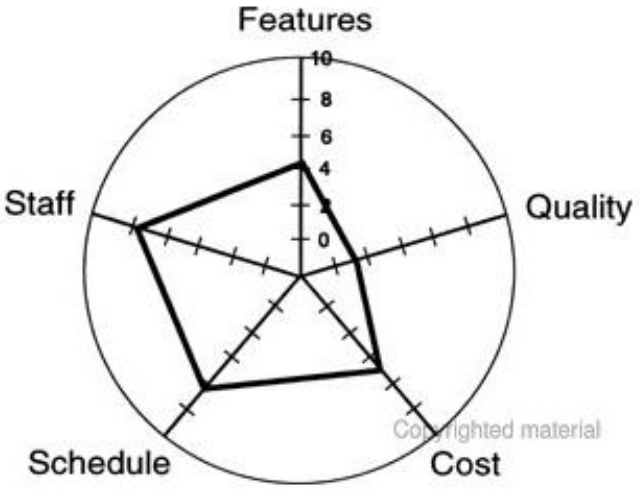

Figure 2.3: Flexibility diagram for a quality-driven application.

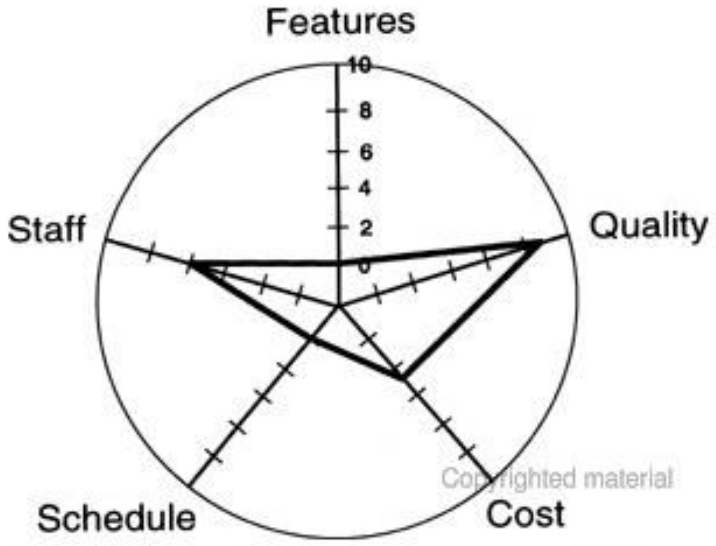

Figure 2.4: Flexibility diagram for a competitive commercial application.

The shapes of the polygons in these examples provide a visual indication of the important aspects of each project. If you push the point on one of the axes inward to reduce the amount of latitude the project leader has in that dimension, you'll generally have to adjust the other dimensions to compensate: Nothing comes without a price. You can apply diagram was shown in Fig. 2.2 .

Project Name: New Information System

Assessed By: _____

Date Assessed: _____

Dimension	Driver (State Objectives)	Constraint (State Limits)	Degree of Freedom (State Range)
Cost			up to 20% overrun from initial estimate is acceptable
Features			60-90% of priority 1 features must be in release 1.0
Quality			release 1.0 can contain up to 5 known major defects
Schedule	release 1.0 must be delivered within 4 months		
Staff		4.5 full-time staff available for duration of the project	

Figure 2.5: Sample form for documenting the negotiated dimensions for a project.

Too often, we focus only on one driver aspect (usually features or schedule), and we overlook the impact on the other dimensions. This is the sort of behavior that leads to those software engineering surprises nobody likes to hear about. Customers, managers, and the marketing department have to accept that they cannot have all the features they want, with no defects, delivered very quickly and at low cost by a downsized development staff. Another way to apply the five-dimension model is to renegotiate when the world changes. If new requirements come along that simply must be included in the upcoming release, ask management what you should adjust to accommodate this request:

- Should other features be deferred?
- Can the schedule be allowed to slip? by how much?
- Can you add staff or pay for overtime to meet the new schedule?
- Or, as usual, does quality slip because sound processes and quality control practices are neglected in the press to ship something—anything—out the door?

The specific answer is less important than the discussion that is triggered when project leaders and developers push back against unexpected changes in any of these five dimensions. Customers and managers have to understand the impact of such changes on the other project dimensions so they can make the right decisions. One characteristic of both a software engineering culture and a mature software development process is that project expectations and commitments are established through negotiation. To avoid unpleasant surprises late in a project's life cycle, the stakeholders all have to understand and agree on their objectives and priorities. Not everyone may A group with a culture in which people are afraid to say no or to discuss problem areas openly will always fall short of expectations. Tools like the flexibility diagram can facilitate these frank, and often difficult, negotiations.

Summary

✓ Never let your boss or your customer talk you into doing a bad job.

✓ The customer isn't always right, but the customer always has a point.

✓ Identify and creatively address the true customer needs, which may not be the same as doing whatever your customers ask you to do.

✓ Look for opportunities to generalize from a specific customer's request to a solution that offers broad potential for use in other applications or other customer domains.

✓ Do not let customers with a bit of software experience dictate how you should build their applications.

✓ Be forthright when discussing project status with your customers and managers, even if you are the bearer of bad news. The ostrich approach does not make the problem go away.

✓ If you agree to do something, either do it or explain to the people affected that you are unable (or unwilling) to do it, so they can make alternate arrangements.

✓ Do not commit to preposterous project schedules that are based on goals and dreams, rather than on a realistic estimating procedure.

✓ The software project stakeholders should establish the relative degrees of flexibility of five project dimensions—features, quality, cost, schedule, and staff—when the project is launched.

Culture Builders and Killers

Culture Builder: Strive to provide an environment in which you shelter your team members from political machinations, useless meetings, and unwarranted finger pointing. Your team members need to know that you will help fight their battles so they can work productively on the right projects. As a manager, take your share of the blame when things go awry, and take only your share of the credit when projects succeed. No one wants to work for a manager he neither trusts nor respects.

Culture Killer: Give your subordinates the responsibility for delivering a project, but do not give them the authority to take the actions and acquire the resources they need to be successful. Delegating any of your authority to project leaders will be perceived as a sign of weakness. Some may grumble that decisions are being made at too high a level on the organization chart, by people who are not close enough to the project to make good decisions, but you know you have enough information to make the right decisions.

References and Further Reading

Karten, Naomi. Managing Expectations. New York: Dorset House Publishing, 1994. Karten provides suggestions for working with “people who want on how to say “No” so it sounds like “Yes.”

Maguire, Steve. Debugging the Development Process. Redmond, Wash.: Microsoft Press, 1994. In Chapter 3 , Maguire describes the importance of using goals to drive your decisions, rather than the desire to please everyone who asks for something. He presents several anecdotes that illustrate the importance of saying “No” and the need to rely on one’s personal integrity when asked to do something inappropriate. Maguire also discusses the importance of focusing on features that are of strategic importance to a product to keep the project on track.

McConnell, Steve. Code Complete. Redmond, Wash.: Microsoft Press, 1993. Chapter 31 discusses “personal character” issues that affect an individual’s effectiveness as a software engineer, including communication and cooperation, creativity and discipline, and intellectual honesty.