

CHAPTER 8

Getting It Out the Door

I have on occasion claimed that I can build the perfect product. Just don't ask me to ever ship it.

As soon as you require that I ship a product on a given date, I can guarantee you that the product will be imperfect. It will disappoint someone along some dimension. It will lack some feature, exhibit some annoying minor bug, or will lack some piece of documentation. No doubt there will be rough edges in its user interface. If only we had more time...

This is not a phenomenon unique to software products. A shipped product is always a compromise between the product we would ideally like to ship—the one that approaches perfection—and the one we really need to ship because we must begin generating revenue. And sometimes, believe it or not, the product we ship *is* good enough, even though it represents a compromise. The test is whether or not it serves the greatest good for the greatest number.

Consider an *update release* of an existing product, one that will add some new features and fix many annoying defects resulting from the previous compromised release. You can work on this update as long as you like; the longer you take, the more features you can add and the more bugs you can fix. But here's another way to look at it: The longer you wait to ship that update release, the longer your existing customers will have

to live with the bugs in the version they are currently using. So the tradeoff becomes this: Is it better to ship 50 bug fixes today, or 55 in another two weeks? If you have thousands of customers who are suffering with Bug #29 on the list every day, I think I can make a pretty good argument for shipping *yesterday*.

Once you realize that shipping the product is not only part of your job but in fact the critical step—Bob Bond¹ would call it “running it through the tape”—you need to consider exactly what is required to go from some assemblage of working bits to a package that you can put on the loading dock or, alternatively, some set of files that you can stage on your download server. You need to consider testing, installation, documentation, preparing the support organization, and many, many other details. Like the death from a thousand cuts, getting this all right can be extremely painful the first hundred or so times you do it. It is one of those exercises that require method and persistence, and extremely meticulous follow-up.

The purpose of this chapter is not to bludgeon you to death with the obvious. What I focus on in this chapter is a small subset of the problem: How do we “close out” development of the software so that we can ship the product? When we are on “final approach” to shipping the product, what changes? The answer is this: If you have been doing it right, the change is imperceptible. If you have neglected thinking about this problem all along, then you will suffer large, severe, and disruptive change at the end, and your ability to ship will be endangered.

If You Build It, They Will Come²

In the world of software products, there are successes and failures, determined by the free market system. We must, of course, add to the list of failures those projects whose products never see the light of day—the ones that are worked on for various lengths of time but never ship. As obvious as it sounds, you cannot be successful unless you meet the precursor of shipping your product.

As you cannot ship what you cannot build, actually putting together the pieces becomes critical. Intrinsic in this is the concept of a *repeatable* build process. You will build the product over and over again, until one of your *candidate releases* passes muster and you let it out the door. I now confront the issues involved in creating such a repeatable build process for your product.

¹ Bob Bond ran sales and marketing at Rational for many years. He was a very positive mentor for me.

² In fact, the line from the movie “Field of Dreams” was, “If you build it, he will come,” the “he” being either Shoeless Joe Jackson or the principal character’s father. The line has been so frequently misquoted that most people use this one. Of course, at the end of the movie “they” come, as illustrated by the stream of headlights across the Midwestern plain.

In the Beginning, There Was the Sandbox

Products come out of projects, and projects tend to begin in haphazard ways. Organizations with well-defined processes have developers building their components in local work areas, sometimes called *sandboxes*. They provide for mechanisms whereby the sub-products of these sandboxes can be assembled, sometimes in *ad hoc* ways, so that each development team can test its progress in the context of the whole product. Configuration management systems allow for appropriate partitioning such that each developer (or team of developers) has the autonomy and isolation to work on his piece without stepping on the other guy's toes, while at the same time providing for a loose integration context.

This works fine in the early, chaotic days when everything is changing very rapidly, and before architectures are well-defined and interfaces are nailed down. However, before too long even modest projects outgrow this framework. At that point, one of two things happens: Either the organization makes *the build* a priority and adds some structure, or it doesn't. In general, those that do establish a regular “heartbeat” for the project—a periodic, regular, and dependable build cycle—improve their chances for success. Those that don't establish this rhythm find that entropy begins to take over, and that building the product becomes more difficult over time.³

Many organizations vastly underestimate the effort it takes to put a good build process in place. Because of this, projects in their latter stages often have a “new” problem to deal with: In addition to having buggy software, incomplete parts, and so on, they also struggle with something that they have taken for granted—the simple assembly of their product. This is a trap for the unwary. In order to not fall into the trap, you need to understand more about the process of assembling a product.

Why Should the Product Build Be Hard, Anyway?

First of all, the product you are going to ship has more pieces to it than the prototypes you have been putting together for internal consumption. Here's a classic example: Developers and testers rarely look at the help system, because they know the product well enough to play with it and test it. Once you are going to have outsiders try to use it, you need a well-elaborated and working help system for people to use. Further, you need instructions for installing the software in different computing environments, as well as various other adjuncts that you can live without when you are only consuming your software

³ *Entropy* is the tendency that all systems have to move from an orderly state to a disordered state when left alone. It is a fundamental physical law. One might say that all attempts at progress, by any civilization, fly in the face of entropy. Another way to say this is that to bring order out of chaos takes work, and that once you stop working, entropy will cause the system to spontaneously move to a more disordered state. I will talk some more about this in Chapter 18, “Bad Analogies.”

internally. So the first problem that comes up is one that might be dismissed as *packaging*. You need more pieces to ship a product than to use it internally; and further, you need to document all the little details that the internal team has always known or taken for granted.⁴ Making the product ready for outside consumers is sometimes called *sanding off the rough edges*. Some of these “rough edges” can be very sharp, and because you don’t catch them all the first time, your first consumers may cut their fingers on them.

Let’s assume, however, that this is just a logistical exercise and that with enough planning you can avoid the “packaging” trap. In some sense, it can be put in the “annoying detail” category: If you ignore it, it will bite you; but if you are aware of it and plan for it, then it is relatively easy to overcome. So, forewarned is forearmed: Treat packaging as a purely technical problem and you will be fine.

In fact, there are three much more fundamental obstacles to success that come up over and over again. They are distinct and interrelated, and all three must be worked on to achieve a successful build process.

Obstacle 1: Organizational Politics

Many software development managers lose sight of the simple fact that controlling the build process is first and foremost a political problem. To put it simply, he who controls the build has an enormous amount of power. After all, the build cycle itself defines the rhythm of the entire development and test organization. Think of the build cycle as the software equivalent of a factory assembly line. The person who gets to define the characteristics of the line and its speed determines, to a very real extent, the output of the factory. Line workers are very aware of their subservience to the line. The cardinal sin in the factory is to slow down, or—Heaven forbid!—shut down the line. The software equivalent is submitting a set of changes that *breaks the build*.⁵

Now the build process is something that everyone must participate in, but only one group can control. By its very nature it is not a democratic enterprise; it requires a certain amount of hierarchical and structural apparatus to work at all. Everyone agrees on this, more or less. The sticky wicket is determining who gets the responsibility and authority to make it work. That group will, from that day forward, wield a lot of power and clout.

Because human beings are, in general, reluctant to give up this sort of power, the build process becomes a political football. Myriad discussions ensue as to who will have

⁴ The standard vehicle for this is called the *release note*. The release note documents the limitations of this version of the software, known bugs, and so on. It is an attempt to characterize the state of the deliverable, as it is better to tell your consumers things you know about rather than have them discover them on their own. Sometimes the release note is called the *readme file*.

⁵ There are legitimate reasons for shutting down the line, and sometimes the person on the factory floor is the most appropriate person to do this. On the other hand, shutting down the line by mistake is definitely not a good idea.

the right to do what to whom in the interest of the build process. All of the negative political tendencies of your organization will be exposed during these discussions.

The purists among you will cry out that political tendencies should be discouraged or even condemned, pointing out that the job is hard enough from a technical point of view and should not be “polluted” by politics. In most organizations, however, *wishing* politics away will not necessarily make them *go* away. Politics is a fact of life that must be dealt with.⁶ However, you must get through this phase, as unpleasant as it first appears. Else, you will be incapable of dealing with the next two hurdles.

Here are some specific suggestions:

- Try to get the group to agree that someone has to be in charge, because a loose confederation approach is doomed to failure.
- Try to reach a reasonable compromise between the autonomy of the constituent teams and the centralized authority that will be required.
- Always make sure that the management team understands the importance of the issue and has the very best people assigned to the build.
- Later in this chapter, we will talk about having a *czar of the build*. Make sure it is a person who is technically competent, firm, fair, and respected by everyone. Install him or her early in the process and have this person guide you through the political shoals.
- Enlist management’s support in crushing “bad politics,” should it rear its ugly head.

Obstacle 2: The Process

Having hacked through all the political jungles that accompany conceding power to the build group, the participants must now agree on the process they will use. Just as form follows function, the process will often be shaped to mirror the political compromises that were made to get to this juncture. There is plenty of interaction between the first and second obstacles. In fact, often the process obstacle presents itself early on, in Phase 1, because it is being used as a surrogate by those who don’t want to openly admit that there are unresolved political issues. In some organizations, we see these two obstacles mashed together into one giant hairball, which in turn gives “process” a bad name. You cannot use “process” to solve what are intrinsically political problems, much in the same way that you cannot “solve” technical problems through political compromise.

⁶ My perspective is that there are “good politics,” akin to the notion of “fighting fair,” and that a healthy political process can and should work toward making good decisions. Then there are “bad politics,” which make organizational objectives subservient to personal agendas and self-aggrandizement; this sort of politics needs to be stamped out wherever it is found. The problem, of course, is the gray zone in between. I treat this subject in more detail in Chapter 13, “Politics.”

The basic tension at this point revolves around the people who want a strict, rigorous process—sometimes called *lots of rules and no mercy*⁷—versus the people who want a looser set of policies. Acknowledging that there is no single, simple, right answer is usually the best place to start here. Your process will have to be tuned to your organization, because all organizations have their peculiarities.⁸

That does not mean that you need to invent new process. I used the word “tune” advisedly in the previous paragraph because I am firmly convinced that the best way to deal with this issue is to start with a base process that has been demonstrated to work before. Rational Software’s Unified Change Management (UCM), for example, has a rich legacy of successful application. We know it works across a broad spectrum of domains, applications, and organizations. Why start over? Do you really think you are going to do better?

There are a few traps you don’t want to fall into at this point. One is the *religious wars* pitfall. In every organization there are process gurus who believe that they, and only they, have the magic formula. And sure enough, every time there are others who resist, quite certain of their own convictions.⁹ Regardless of who is right or wrong, these crusades are totally unproductive, often revolving around obscure details of little importance. The strong manager needs to identify the religious process fanatics and stifle them early. Sometimes the only answer is to tell them to put a cork in it. Remember always that process is not an end in and of itself; it is a means to an end—shipping the product!¹⁰

Another trap is to think that any process, no matter how good, can substitute for thought or judgment. For every ironclad rule, there is bound to be an exception. You will have to watch what is going on and make midcourse corrections, no matter what your

⁷ I believe the world is indebted to James E. Archer for this characterization. Jim is one of the most effective development managers I know, having been the godfather for Rational’s programming environments products from the very beginning. He and I had many interesting discussions on the right amount of process.

⁸ Some people argue at this point that you should endeavor to get your process “right” and then tune your organization to fit the process. While this is a laudable objective and *theoretically* the right approach, I have rarely found it to be successful in practice. You cannot allow a regressive organization the prerogative of rejecting reasonable process; on the other hand, it is difficult to implement any process that is too far out in front of the organization that must carry it off.

⁹ To illustrate how far out of control this can become, the wars are often characterized as struggles between the “process Nazis” and the “anarchists.” With such value-laden labels, it is difficult to have discussions that will get to the right place.

¹⁰ In a like manner, the anarchists will be hard put to demonstrate that they can ship product without any process. As is the case in almost all these debates, neither extreme position is defensible.

process is. As called out previously, you will need to modify and tune your process in real time as you discover what works for you and what doesn't.

Lastly, get on with it. Perfect is the enemy of good.¹¹ You will develop your process iteratively, just the way you develop the software. Get to Iteration 1 quickly. Learn. Change. Improve. Repeat until done.

Obstacle 3: Tools

Just as the first obstacle (politics) and the second obstacle (process) are intimately related, so are the second and the third. The third, of course, is the toolset that you will use to implement the process. Needless to say, choosing the tools first is getting it bass-ackwards, but surprisingly enough, that's the way many organizations go about it. They then wind up with the tool determining the process, which can be loads of fun when the process thus derived is inconsistent with the political philosophy of the organization.

Obviously, you need tools that will automate and enforce the process you have chosen to use. If you have a process that admits mistakes, you will be "backing out" changes from time to time. Does the tool support that easily? Are developers going to be checking in their work to a common baseline from multiple remote sites? If so, then your tool had better support that model. Do you want to build your entire product from top to bottom every night? If so, then I hope your tool has the performance and turnaround characteristics that will permit that. Do you want to automate your regression testing as part of the build? Once again, tool support is crucial.

Even organizations that have done a good job with the first two problems sometimes flounder with the third. And sometimes it is not the tools' fault either. Once again, using our factory analogy, you need someone to monitor the line and to do quality control for the product coming off the line. Without constant vigilance, it is easy to automate a process that produces a low-quality result. Every successful build process requires a foreman or the equivalent thereof; sometimes he or she is called the *czar* (or *czarina*) of the build or, more simply, the *buildmeister*.¹² The buildmeister monitors the health of the line and makes sure that a steady stream of good-quality product is produced.

¹¹ I first heard this from Mikhail Drabkin of Riga, Latvia, and assumed it was Russian folk wisdom. He may have in fact been (mis?)quoting Soviet Admiral Sergei Georgievich Gorshkov, although that citation turns out to be only an approximation: Gorshkov's ended with "Good enough." Dr. Stephen Franklin of U.C. Irvine points out that very similar sentiments have been attributed to both Clauswitz and Voltaire. Clauswitz is much more verbose, and there seems to be evidence that Voltaire "borrowed" from an Italian proverb. Though these pithy sayings sometimes have a provenance that is difficult to pin down, one can deny neither their truth nor their wisdom.

¹² Here's a cautionary, funny, and politically incorrect tale: I once made a big deal about having a czar of the build, and then appointed a fellow who was, shall we say, altitudinally challenged. He unfortunately became known as the czardine of the build. Ouch!

One last semi-technical note: Beware of the old saying “We can always write a script that can do that.” The problem is that these scripts always start out small and simple and then grow in ways that are random and unsupervised. Scripts, unlike programs, are rarely designed; they just grow. They become compendia of special cases and are inadequate to respond to the ever-increasing demands of the organization; they are brittle. They are a maintenance nightmare, especially if the original author moves on. And they are very, very difficult to debug. Just as the road to Hell is paved with good intentions, the road to “build Hell” is paved with the out-of-control products of general-purpose scripting languages.¹³

What About Iterative Development?

Iterative development sidesteps one of the great dangers of the waterfall approach: leaving system integration to the last minute. One of the reasons so many waterfall projects fail is that, very late in the game, developers are trying to assemble their product for the very first time. In addition to finding many bugs, mostly in the interfaces, they grapple with the normal logistical and organizational problems of putting together a build chain for the first time. Often, things that pass for bugs are nothing more than the artifacts of broken builds. But the organization is in such chaos at this point—running out of time, nothing working, people frazzled—that it is hard to separate the sugar from the salt. It is also a very bad time to be trying to solve political and process problems.

By contrast, iterative development requires that you construct your build chain to accomplish the deliverable for Iteration 1—a working program. So you begin to debug this process early in the project, not at the end. By the time you get to Iteration 3 or 4, the build process is actually starting to work pretty well. For the last iteration, the one that will deliver the final bits, the build should be working like a finely lubricated Swiss watch.

As with pretty much everything else in software development, there are a small number of ways to get this right and almost an infinite number of ways to get it wrong. If you view the build as a detail that will “just happen,” then the odds are against you. Make sure you attack the build process as a conscious effort that is critical to your success, and devote the time, energy, and resources to it that it demands. To do any less is sheer folly.

¹³ Some scripting languages have been favorably likened to duct tape; would you want to begin the final assembly of your jet aircraft by having some guy yell out, “Time to get out the duct tape!”?

Recap

I've frequently been called in late in the software development cycle when projects are in trouble. Often, it is difficult at first to judge the depth of the yogurt. Usually the developers are focusing on how much they are "behind," as measured in things that are coded but just don't work, and things that are not coded at all but should be.

While this is one important aspect, I always begin to also look at the health of the build process. If the build process is non-existent or badly broken, it needs attention *immediately*. The reason for this should be obvious; at this stage of the proceedings, the lack of a reliable, repeatable build process will impede all further progress. You can't test what you can't build, and repeated testing is a necessity at this point; else, how can the developers know what they've fixed and what remains problematic?

Sadly, in many organizations the build process is something that is relegated to the B Team. This is a huge mistake. You must have A-list players in this part of the organization. As soon as people understand how much you as the senior manager appreciate the contribution of this team, you will have no problems getting volunteers.

The other thing that proves very slippery is answering the question, "How do we know when we are done?" Getting agreement well in advance on some clear criteria is incredibly useful; such agreement reduces the odds that the bar will move radically up or down as the ship date looms. One of the major objectives when moving into the Transition Phase of iterative development is to define reasonable criteria for shipment. Without a clear "exit plan," the project risks a series of never-ending last-minute slips.

Thus ends Part 2 on the basics. In Part 3, I will look at software development from a project management perspective.