3

Wasting Money

It's harder than you might think to squander millions of dollars, but a flawed software-development process is a tool well suited to the job. That's because software development lacks one key element: an understanding of what it means to be "done." Lacking this vital knowledge, we blindly bet on an arbitrary deadline. We waste millions to cross the finish line soonest, only to discover that the finish line was a mirage. In this chapter I'll try to untangle the expensive confusion of deadline management.

Deadline Management

There is a lot of obsessive behavior in Silicon Valley about time to market. It is frequently asserted that shipping a product *right now* is far better than shipping it later. This imperative is used as a justification for setting impossibly ambitious ship dates and for burning out employees, but this is a smoke screen that hides bigger, deeper fears—a red herring. Shipping a product that angers and frustrates users in three months is *not* better than shipping a product that pleases users in six months, as any businessperson knows full well.

Managers are haunted by two closely related fears. They worry about when their programmers will be done building, and they doubt whether the product will be good enough to ultimately succeed in the marketplace. Both of these fears stem from the typical manager's lack of a clear vision of what the finished product actually will consist of, aside from mother-and-apple-pie statements such as "runs on the target computer" and "doesn't crash." And lacking this vision, they cannot assess a product's progress towards completion.

The implication of these two fears is that as long as it "doesn't crash," there isn't much difference between a program that takes three months to code and one that takes six months to code, except for the prodigious cost of three months of unnecessary programming. After the programmers have begun work, money drains swiftly. Therefore, logic tells the development manager that the most important thing to do is to get the coding started as soon as possible and to end it as soon as possible.

The conscientious development manager quickly hires programmers and sets them coding immediately. She boldly establishes a completion date just a few months off, and the team careens madly toward the finish line. But without product design, our manager's two fears remain unquelled. She has not established whether the users will like the product, which indeed leaves its success a mystery. Nor has she established what a "complete" product looks like, which leaves its completion a mystery. Later in the book, I'll show how interaction design can ease these problems. Right now, I'll show how thoroughly the deadline subverts the development process, turning all the manager's insecurities into self-fulfilling prophecies.

What Does "Done" Look Like?

After we have a specific description of what the finished software will be, we can compare our creation with it and really *know* when the product is done.

There are two types of descriptions. We can create a very complete and detailed physical description of the actual product, or we can describe the reaction we'd like the end user to have. In building architecture, for example, blueprints fill the first requirement. When planning a movie or creating a new restaurant, however, we focus our description on the feelings we'd like our clients to experience. For software-based products, we must necessarily use a blend of the two.

Unfortunately, most software products never *have* a description. Instead, all they have is a shopping list of features. A shopping bag filled with flour, sugar, milk, and eggs is not the same thing as a cake. It's only a cake when all the steps of the recipe have been followed, and the result looks, smells, and tastes substantially like the known characteristics of a cake.

Having the proper ingredients but lacking any knowledge of cakes or how to bake, the ersatz cook will putter endlessly in the kitchen with only indeterminate results. If we demand that the cake be ready by 6 o'clock, the conscientious cook will certainly bring us a platter at the appointed hour. But will the concoction be a cake? All we know is that it is on time, but its success will be a mystery.

In most conventional construction jobs, we know we're done because we have a clear understanding of what a "done" job looks like. We know that the building is completed because it looks and works just like the blueprints say it should look and work. If the deadline for construction is June 1, the arrival of June doesn't necessarily mean that the building is done. The relative completeness of the building can only be measured by examining the actual building in comparison to the plans.

Without blueprints, software builders don't really have a firm grasp on what makes the product "done," so they pick a likely date for completion, and when that day arrives they declare it done. It is June 1; therefore, the product is completed. "Ship it!" they say, and the deadline becomes the sole definition of project completion.

The programmers and businesspeople are neither stupid nor foolish, so the product won't be in complete denial of reality. It will have a robust set of features, it will run well, and it won't crash. The product will work reasonably well when operated by people *who care deeply* that it works well. It might even have been subjected to usability testing, in which strangers are asked to operate it under the scrutiny of usability professionals[1]. But, although these precautions are only reasonable, they are insufficient to answer the fundamental question: Will it succeed?

Parkinson's Law

Managers know that software development follows Parkinson's Law: Work will expand to fill the time allotted to it. If you are in the software business, perhaps you are familiar with a corollary to Parkinson called the Ninety-Ninety Rule, attributed to Tom Cargill of Bell Labs: "The first 90% of the code accounts for the first 90% of the development time. The remaining 10% of the code accounts for the other 90% of the development time." This self-deprecating rule says that when

[1] *Usability professionals are not interaction designers. I discuss this difference in detail in Chapter 12, "Desperately Seeking Usability."*

the engineers have written 90% of the code, they *still* don't know where they are! Management knows full well that the programmers won't hit their stated ship dates, regardless of what dates it specifies. The developers work best under pressure, and management uses the delivery date as the pressure-delivery vehicle.

In the 1980s and 1990s, Royal Farros was the vice president of development for T/Maker, a small but influential software company. He says, "A lot of us set deadlines that we *knew* were impossible, enough so to qualify for one of those Parkinson's Law corollaries. 'The time it will take to finish a programming project is twice as long as the time you've allotted for it.' I had a *strong* belief that if you set a deadline for, say, six months, it would take a year. So, if you had to have something in two years, set the deadline for one year. Bonehead sandbagging, but it always worked."

When software entrepreneur Ridgely Evers was with Intuit, working on the creation of QuickBooks, he experienced the same problem. "The first release of QuickBooks was supposed to be a nine-month project. We were correct in estimating that the development period would be the same as a gestation period, but we picked the wrong species: It took almost two-and-a-half years, the gestation period for the elephant."

Software architect Scott McGregor points out that Gresham's Law—that bad currency drives out good—is also relevant here. If there are two currencies, people will hoard the good one and try to spend the bad one. Eventually, only the bad currency circulates. Similarly, bad schedule estimates drive out good ones. If everybody makes bogus but rosy predictions, the one manager giving realistic but longer estimates will appear to be a heel-dragger and will be pressured to revise his estimates downward.

Some development projects have deadlines that are unreasonable by virtue of their arbitrariness. Most rational managers still choose deadlines that, while reachable, are only reachable by virtue of extreme sacrifice. Sort of like the pilot saying, "We're gonna make Chicago on time, but only if we jettison all our baggage!" I've seen product managers sacrifice not only design, but testing, function, features, integration, documentation, and reality. *Most product managers that I have worked with would rather ship a failure on time than risk going late.*

The Product That Never Ships

This preference is often due to every software development manager's deepest fear: that after having become late, the product will never ship at all. Stories of products never shipping are not apocryphal. The project goes late, first by one year, then two years, then is euthanized in its third year by a vengeful upper management or board of directors. This explains the rabid adherence to deadlines, even at the expense of a viable product.

For example, in the late 1990s, at the much-publicized start-up company Worlds, Inc., many intelligent, capable people worked on the creation of a virtual, online world where people's avatars could wander about and engage other avatars in real-time conversation. The product was never fully defined or described, and after tens of millions of investment capital was spent, the directors mercifully pulled the plug.

In the early 1990s, another start-up company, Nomadic Computing, spent about $15 million creating a new product for mobile businesspeople. Unfortunately, no one at the company was quite sure what its product was. They knew their market, and most of the program's functions, but weren't clear on their users' goals. Like mad sculptors chipping away at a huge block of marble hoping to discover a statue inside, the developers wrote immense quantities of useless code that was all eventually thrown away, along with money, time, reputations, and careers. The saddest waste, though, was the lost opportunity for creating software that really was wanted.

Even Microsoft isn't immune from such wild goose chases. Its first attempt at creating a database product in the late 1980s consumed many person-years of effort before Bill Gates mercifully shut it down. Its premature death sent a shock wave through the development community. Its successor, Access, was a completely new effort, staffed and managed by all new people.

Shipping Late Doesn't Hurt

Ironically, shipping late generally isn't fatal to a product. A third-rate product that ships late often fails, but if your product delivers value to its users, arriving behind schedule won't necessarily have lasting bad effects. If a product is a hit, it's not a big deal that it ships a month—or even a year—late. Microsoft Access shipped several years late, yet it has enjoyed formidable success in the market. Conversely, if a product stinks, who cares that it shipped on time?

Certainly, some consumer products that depend on the Christmas season for the bulk of their sales have frighteningly important due dates. But most software-based products, even consumer products, aren't that sensitive to any particular date.

For example, in 1990 the PenPoint computer from GO was supposed to be the progenitor of a handheld-computer revolution. In 1992, when the PenPoint crashed and burned, the Apple Newton inherited the promise of the handheld revolution. When the Newton failed to excite people, General Magic's Magic Link computer became the new hope for handhelds. That was in 1994. When the Magic Link failed to sell, the handheld market appeared dead. Venture capitalists declared it a dry hole. Then, out of nowhere, in 1996, the PalmPilot arrived to universal acclaim. It seized the handheld no-man's-land *six years late.* Markets are always ready for good products that deliver value and satisfy users.

Of course, companies with a long history of making hardware-only products now make hybrid versions containing chips and software. They tend to underestimate the influence of software and subordinate it to the already-established completion cycles of hardware. This is wrong because as Chapter 1, "Riddles for the Information Age," showed, these companies are now in the software business, whether or not they know it.

Feature-List Bargaining

One consequence of deadline management is a phenomenon that I call "feature-list bargaining."

Years ago programmers got burned by the vague product-definition process consisting of cocktail-napkin sketches, because they were blamed for the unsuccessful software that so often resulted. In self-defense, programmers demanded that managers and marketers be more precise. Computer programs are procedural, and procedures map closely to features, so it was only natural that programmers would define "precision" as a list of features. These feature lists allowed programmers to shift the blame to management when the product failed to live up to expectations. They could say, "It wasn't my fault. I put in all the features management wanted."

Thus, most products begin life with a document variably called a "marketing specification," "technical specification," or "marketing requirements document." It is really just a list of desired features, like the list of ingredients in the recipe for cake. It is usually the result of several long brainstorming sessions in which managers, marketers, and developers imagine what features would be cool and jot them down. Spreadsheet programs are a favorite tool for creating these lists, and a typical one can be dozens of pages long. (Invariably, at least one of the line items will specify a "good user interface.") Feature suggestions can also come from focus groups, market research, and competitive analysis.

The managers then hand the feature list to the programmers and say, "The product must ship by June 1." The programmers—of course—agree, but they have some stipulations. There are far too many features to create in the time allotted, they claim, and many of them will have to be cut to meet the deadline. Thus begins the time-honored bargaining.

The programmers draw a dividing line midway through the list. Items above it will be implemented, they declare, while those below the "line of death" are postponed or eliminated. Management then has two choices: to allow more time or to cut features. Although the project will inevitably take more time, management is loath to play that trump so early in the round, so it negotiates over features. Considerable arguing and histrionics occur. Features are traded for time; time is traded for features. This primitive capitalist negotiation is so human and natural that both parties are instantly comfortable with it. Sophisticated parallel strategies develop. As T/Maker's Royal Farros points out, when one "critical-path feature was blamed for delaying a deadline, it would let a dozen other tardy features sneak onto the list without repercussion." Lost in the battle is the perspective needed for success.

Farros described T/Maker's flagship product, a word processor named WriteNow, as "a perfect product for the university marketplace. In 1987, we actually shipped more copies of WriteNow to the university market than Microsoft shipped Word. However, we couldn't hold our lead because we angered our very loyal, core fans in this market by not delivering the one word-processor feature needed in a university setting: *endnotes*. Because of trying to make the deadline, we could never slip this feature into the specification. We met our deadline but lost an entire market segment."

Programmers Are in Control

Despite appearances, programmers are completely in control of this bottom-up decision-making process. They are the ones who establish how long it will take to implement each item, so they can force things to the bottom of the list merely by estimating them long. The programmers will—in self-defense—assign longer duration to the more nebulously defined items, typically those concerned with substantive user-interface issues. This inevitably causes them to migrate to the bottom of the list. More familiar idioms and easy-to-code items, such as menus, wizards, and dialog boxes, bubble to the top of the list. All of the analysis and careful thinking done by high-powered and high-priced executives is made moot by the unilateral cherry picking of a programmer following his own muse or defending his turf.

Like someone only able to set the volume of a speaker that isn't within hearing distance, managers find themselves in the unenviable position of only having tools that control ineffective parameters of the development process. It is certainly true that management needs to control the process of creating and shipping successful software, but, unfortunately, our cult of deadline ignores the "successful" part to concentrate only on the "creating" part. We give the creators of the product the reins to the process, thus relegating management to the role of passenger and observer.

Features Are Not Necessarily Good

Appearances to the contrary, users aren't really compelled by features. Product successes and failures have shown repeatedly that users don't care that much about features. Users only care about achieving their goals. Sometimes features are needed to reach goals, but more often than not, they merely confuse users and get in the way of allowing them to get their work done. Ineffective features make users feel stupid. Borrowing from a previous example, the successful PalmPilot has far fewer features than did General Magic's failed Magic Link computer, Apple's failed Newton, or the failed PenPoint computer. The PalmPilot owes its success to its designers' single-minded focus on its target user and the objectives that user wanted to achieve.

About the only good thing I can say about features is that they are quantifiable. And that quality of being countable imbues them with an aura of value that they simply don't have. Features have negative qualities every bit as strong as their positive ones. The biggest design problem they cause is that every well-meant feature that *might possibly* be useful obfuscates the few features that *will probably* be useful. Of course, features cost money to implement. They add complexity to the product. They require an increase in the size and complexity of the documentation and online help system. Above all, cost-wise, they require additional trained telephone tech-support personnel to answer users' questions about them.

It might be counterintuitive in our feature-conscious world, but you simply cannot achieve your goals by using feature lists as a problem-solving tool. It's quite possible to satisfy every feature item on the list and still hatch a catastrophe. Interaction designer Scott McGregor uses a delightful test in his classes to prove this point. He describes a product with a list of its features, asking his class to write down what the product is as soon as they can guess. He begins with 1) internal combustion engine; 2) four wheels with rubber tires; 3) a transmission connecting the engine to the drive wheels; 4) engine and transmission mounted on metal chassis; 5) a steering wheel. By this time, every student will have written down his or her positive identification of the product as an automobile, whereupon Scott ceases using features to describe the product and instead mentions a couple of user goals: 6) cuts grass quickly and easily; 7) comfortable to sit on. From the five feature clues, not one student will have written down "riding lawnmower." You can see how much more descriptive goals are than features.

Iteration and the Myth of the Unpredictable Market

In an industry that is so filled with money and opportunities to earn it, it is often just easier to move right along to another venture and chalk up a previous failure to happenstance, rather than to any *real* reason.

I was a party to one of these failures in the early 1990s. I helped to start a venture-funded company whose stated goal was to make it absurdly simple to network PCs together.[2] The product worked well and was easy to use, but a tragic series of

[2] *Actually, we said that we wanted to make it "as easy to network Intel/Windows computers as it was to network Macintosh computers." At the time, it was ridiculously simple to network Macs together with AppleTalk. Then, as now, it was quite difficult to network Wintel PCs together.*

self-inflicted marketing blunders caused it to fail dismally. I recently attended a conference where I ran into one of the investors who sat on the doomed company's board of directors. We hadn't talked since the failure of the company, and—like veterans of a battlefield defeat meeting years later—we consoled each other as sadder but wiser men. To my unbridled surprise, however, this otherwise extremely successful and intelligent man claimed that in hindsight he had learned a fundamental lesson: Although the marketing, management, and technical efforts had been flawless, the buying public "just wasn't interested in easy-to-install local area networks." I was flabbergasted that he would make such an obviously ridiculous claim and countered that surely it wasn't lack of desire, but rather our failure to satisfy the desire properly. He restated his position, arguing forcefully that we had demonstrated that easy networking just wasn't something that people wanted.

Later that evening, as I related this story to my wife, I realized that his rationalization of the failure was certainly convenient for all the parties involved in the effort. By blaming the failure on the random fickleness of the market, my colleague had exonerated the investors, the managers, the marketers, and the developers of any blame. And, in fact, each of the members of that start-up has gone on to other successful endeavors in Silicon Valley. The venture capitalist has a robust portfolio of other successful companies.

During development, the company had all the features itemized on the feature list. It stayed within budget. It shipped on schedule. (Well, actually, we kept extending the schedule, but it shipped on *a* schedule.) All the quantitatively measurable aspects of the product-development effort were within acceptable parameters. The only conclusion this management-savvy investor could make was the existence of an unexpected discontinuity in the marketplace. How could *we* have failed when all the meters were in the green?

The fact that these measures are objective is reassuring to everyone. Objective and quantitative measure is highly respected by both programmers and businesspeople. The fact that these measures are usually ineffective in producing successful products tends to get lost in the shuffle. If the product succeeds, its progenitors will take the credit, attributing the victory to their savvy understanding of technology and marketing.

On the other hand, if the product fails, nobody will have the slightest motivation to exhume the carcass and analyze the failure. Almost any excuse will do, as long as the players—both management and technical—can move along to the next high-tech opportunity, of which there is an embarrassment of riches. Thus, there is no reason to weep over the occasional failure. The unfortunate side effect of not understanding failure is the silent admission that success is not predictable—that luck and happenstance rule the high-tech world. In turn, this gives rise to what the venture capitalists call the "spray and pray" method of funding: Put a little bit of money into a lot of investments and then hope that one of them gets lucky.

⌘

Rapid-development environments such as the World Wide Web—and Visual Basic before it—have also promoted this idea of simply iterating until something works. Because the Web is a new advertising medium, it has attracted a multitude of marketing experts who are particularly receptive to the myth of the unpredictable market and its imperative to iterate. Marketers are familiar with the harsh and arbitrary world of advertising and media. After all, much of advertising *really is* random guesswork. For example, in advertising, "new" is the single most effective marketing concept, yet when Coca-Cola introduced "New Coke" in the mid-1980s, it failed utterly. Nobody could have predicted this result. People's tastes and styles change randomly, and the effectiveness of marketing can appear to be random.

On the Web, the problem arises when a Web site matures from the online-catalog stage into the online-store stage. It changes from a one-way presentation of data to an interactive software application. The advertising and media people who had such great success with the first-generation site now try their same iteration methods on the interactive site and run into trouble, often without realizing it. Marketing results may be random, but interaction is not. The cognitive friction generated by the software's interactivity is what gives the impression of randomness to those untrained in interaction design.

The remarkably easy-to-change nature of the World Wide Web plays into this because an advertisement or marketing campaign can be aired for a tiny fraction of the cost (and time) of print or TV advertising. The savvy Web marketer can get almost instantaneous feedback on the effectiveness of an ad, so the speed of the iteration increases dramatically, and things are hacked together overnight. In practice, it boils down to "throw it against the wall and see what sticks." Many managers of Web start-ups use this embarrassingly simple doctrine of design by guesswork. They write any old program that can be built in the least time and then put it before their users. They then listen to the complaints and feedback, measure the patterns of the user's navigation clicks, change the weak parts, and then ship it again.

Generally, programmers aren't thrilled about the iterative method because it means extra work for them. Typically, it's managers new to technology who like the iterative process because it relieves them of having to perform rigorous planning, thinking, and product due diligence (in other words, interaction design). Of course, it's the users who pay the dearest price. They have to suffer through one halfhearted attempt after another before they get a program that isn't too painful.

Just because customer feedback improves your understanding of your product or service, you cannot then deduce that it is efficient, cheap, or even effective to toss random features at your customers and see which ones are liked and which are disliked. In a world of dancing bears, this can be a marginally viable strategy, but in any market in which there is the least hint of competition, it is suicidal. Even when you are all alone in a market, it is a very wasteful method.

Many otherwise sensitive and skilled managers are unashamedly proud of this method. One mature, experienced executive (a former marketing man) asked me, in self-effacing rhetoric, "How could anyone presume to know what the users want?" This is a staggering question. Every businessperson presumes. The value that most businesspeople bring to their market is precisely their "presumption" of what the customer wants. Yes, that presumption will miss the mark with *some* users, but not to presume at all means that *every* user won't like it. This foolish man believed that his customers didn't mind plowing through his guesses to do his design work for him. Today, in Silicon Valley, there might be lots of enthusiastic Web-surfing apologists who are willing to help this lazy executive figure out his business, but how many struggling survivors did he alienate with that haughty attitude? As he posted sketchy version after sketchy version of his site, reacting only to those people with the stamina to return to it, how many customers did he lose permanently? What did *they* want? It has been said that the way Stalin cleared a minefield was to march a regiment through it. Effective? Yes. Efficient, humanitarian, viable, desirable? No.

The biggest drawback, of course, is that you immediately scare away all survivors, and your only remaining users will be apologists. This seriously skews the nature and quality of your feedback, condemning you to a clientele of technoid apologists, which is a relatively small segment. This is one reason why so few personal-computer software-product makers have successfully crossed over into mass markets.

I am not saying that you cannot learn from trial and error, but those trials should be informed by something more than random chance and should begin from a well-thought-out solution, not an overnight hack. Otherwise, it's just giving lazy or ignorant businesspeople license to abuse consumers.

The Hidden Costs of Bad Software

When software is frustrating and difficult to use, people will avoid using it. That is unremarkable until you realize that many people's jobs are dependent on using software. The corporate cost of software avoided is impossible to quantify, but it is real. Generally, the costs are not monetary ones, anyway, but are exacted in far more expensive currencies, such as time, order, reputation, and customer loyalty.

People who use business software might despise it, but they are paid to tolerate it. This changes the way people think about software. Getting paid for using software makes users far more tolerant of its shortcomings because they have no choice, but it doesn't make it any less expensive. Instead—while the costs remain high—they become very difficult to see and account for.

Badly designed business software makes people dislike their jobs. Their productivity suffers, errors creep into their work, they try to cheat the software, and they don't stay in the job very long. Losing employees is very expensive, not just in money but in disruption to the business, and the time lost can never be made up. Most people who are paid to use a tool feel constrained not to complain about that tool, but it doesn't stop them from feeling frustrated and unhappy about it.

One of the most expensive items associated with hard-to-use software is technical support. Microsoft spends $800 million annually on technical support. And this is a company that spends many hundreds of millions of dollars on usability testing and research, too. Microsoft is apparently convinced that support of this magnitude is just an unavoidable cost of doing business. I am not. Imagine the advantage it would give your company if you didn't make the same assumption that Microsoft did. Imagine how much more effective your development efforts would be if you could avoid spending over five percent of your net revenue on technical support.

Ask any person who has ever worked at any desktop-software company in technical support, and he will tell you that the one thing he spends most of his time and effort on is the file system. Just like Jane in Chapter 1, users don't understand the recursive hierarchy of the file system—the Finder or Explorer—on Windows, the Mac, or Unix. Surprisingly, very few companies will spend the money to design and implement a more human-friendly alternative to the file system. Instead, they accept the far more expensive option of answering phone calls about it in perpetuity.

You can blame the "stupid user" all you want, but you still have to staff those phones with expensive tech-support people if you want to sell or distribute within your company software that hasn't been designed.

The Only Thing More Expensive Than Writing Software Is Writing Bad Software

Programmers cost a lot, and programmers sitting on their hands waiting for design to be completed gall managers in the extreme. It seems foolish to have programmers sit and wait, when they could be programming, thinks the manager. It is false economy, though, to put programmers to work before the design is completed. After the coding process begins, the momentum of programming becomes unstoppable, and the design process must now respond to the needs of programmers, instead of vice versa. Indeed, it is foolish to have programmers wait, and by the simple expedient of having interaction designers plan your next product or release concurrently with the construction of this product or release, your programmers will never have to idly wait.

It is more costly in the long run to have programmers write the wrong thing than to write nothing at all. This truth is so counterintuitive that most managers balk at the very idea. After code is written, it is very difficult to throw it out. Like writers in love with their prose, programmers tend to have emotional attachments to their algorithms. Altering programs in midstride upsets the development process and wounds the code, too. It's hard on the manager to discard code because she is the one who paid dearly for it, and she knows she will have to spend even more to replace it.

If design isn't done before programming starts, it will never have much effect. One manager told me, "We've already got people writing code and I'm not gonna stop." The attitude of these cowboys is, "By the time you are ready to hit the ground, I'll have stitched together a parachute." It's a bold sentiment, but I've never seen it work.

Lacking a solid design, programmers continually experiment with their programs to find the best solutions. Like a carpenter cutting boards by eye until he gets one that fits the gap in the wall, this method causes abundant waste.

The immeasurability and intangibility of software conspires to make it nearly impossible to estimate its size and assess its state of completion. Add in the programmer's joy in her craft, and you can see that software development always grows in scope and time and never shrinks. We will always be surprised during its construction, unless we can accurately establish milestones and reliably measure our progress against them.

Opportunity Cost

In the information age, the most expensive commodity is not the cost of building something, but the lost opportunity of what you are *not* building. Building a failure means that you didn't build a success. Taking three annual releases to get a good product means that you didn't create three good products in one release each.

Novell's core business is networking, but it attempted to fight Microsoft toe-to-toe in the office-applications arena. Although its failed efforts in the new market were expensive, the true cost was its loss of leadership in the networking market. The money is nothing compared to the singular potential of the moment. Netscape lost its leadership in the browser market in the same way when it decided to compete against Microsoft in the operating-system business.

Any developer of silicon-based products has to evaluate what the most important goals of its users are and steadfastly focus on achieving them. It is far too easy to be beguiled by the myriad of opportunities in high tech and to gamble away the main chance. Programmers—regardless of their intelligence, business acumen, loyalty, and good intentions—march to a slightly different drummer and can easily drag a business away from its proper area of focus.

The Cost of Prototyping

Prototyping is programming, and it has the momentum and cost of programming, but the result lacks the resiliency of real code. Software prototypes are scaffolds and have little relation to permanent, maintainable, expandable code—the equivalent of stone walls. Managers, in particular, are loath to discard code that works, even if it is just a prototype. They can't tell the difference between scaffolding and stone walls.

You can write a prototype much faster than a real program. This makes it attractive because it seems so inexpensive, but real programming gives you a reliable program, and prototyping gives you a shaky foundation. Prototypes are experiments made to be thrown out, but few of them ever are. Managers look at the running prototype and ask, "Why can't we just use this?" The answer is too technically complex and too fraught with uncertainty to have sufficient force to dissuade the manager who sees what looks like a way to avoid months of expensive effort.

The essence of good programming is deferred gratification. You put in all of the work up front, and then you reap the rewards later. There are very few tasks that aren't cheaper to do manually. Once written, however, programs can be run a million times with no extra cost. The most expensive program is one that runs once. The cheapest program is the one that runs ten billion times. However, any inappropriate behavior will also be magnified ten billion times. Once out of the realm of little programs, such as the ones you wrote in school, the economics of software take on a strange reversal in which the cheapest programs to own are the ones that are most expensive to write, and the most expensive programs to own are the cheapest to write.

Writing a big program is like making a pile of bricks. The pile is one brick wide and 1,000 bricks tall, with each brick laid right on top of the one preceding it. The tower can reach its full height only if the bricks are placed with great precision on top of one another. Any deviation will cause the bricks above to wobble, topple, and fall. If the 998th brick deviates by a quarter of an inch, the tower can still probably achieve 1,000 bricks, but if the deviation is in the fifth brick, the tower will never get above a couple dozen.

This is very characteristic of software, whose foundations are more sensitive to hacking than the upper levels of code. As any program is constructed, the programmer makes false starts and changes as she goes. Consequently, the program is filled with the scar tissue of changed code. Every program has vestigial functions and stubbed-out facilities. Every program has features and tools whose need was discovered sometime after construction began grafted onto it as afterthoughts. Each one of these scars is like a small deviation in the stack of bricks. Moving a button from one side of a dialog box to the other is like joggling the 998th brick, but changing the code that draws all button-like objects is like joggling the 5th brick. Object-oriented programming and the principles of encapsulation are defensive techniques whose sole purpose is to immunize the program from the effects of scar tissue. In essence, object orientation divides the 1,000-brick tower into 10 100-brick towers.

Good programmers spend enormous amounts of time and energy setting up to write a big program. It can take days just to set up the programming environment, before a line of product code is written. The proper libraries must be selected. The data must be defined. The storage and retrieval subsystems must be analyzed, defined, coded, and tested.

As the programmers proceed into the meat of the construction, they invariably discover mistakes in their planning and flaws in their assumptions. They are then faced with Hobson's choice of whether to spend the time and effort to back up and fix things from the start, or to patch over the problem wherever they are and accept the burden of the new scar tissue—the deviation. Backing up is always very expensive, but that scar tissue ultimately limits the size of the program—the height of the bricks.

Each time a program is modified for a new revision to fix bugs or to add features, scar tissue is added. This is why software must be thrown out and completely rewritten every couple of decades. After a while, the scar tissue becomes too thick to work well anymore.

Prototypes—by their very nature—are programs that are slapped together in a hurry so that the results can be assayed. What the programmer exchanges in order to build the prototype so speedily is the perfect squaring of the bricks. Instead of using the "right" data structures, information is thrown in helter-skelter. Instead of using the "right" algorithms, whatever code fragments happen to be lying around are drafted for service. Prototypes *begin* life as masses of scar tissue. They can never grow very large.

Some software developers have arrived at the unfortunate conclusion that modern rapid-prototyping tools—such as Visual Basic—are effective design tools. Rather than designing the product, they just whip out an extremely anemic version of it with a visual programming tool. This prototype typically becomes the foundation for the product. This trades away the robustness and life span of the product for an illusory benefit. You can get a better design with pencil and paper and a good methodology than you can with any amount of prototyping.

For those who are not designers, visualizing the form and behavior of software that doesn't yet exist is difficult, if not impossible. Prototypes have been drafted into the role of a visualization tool for these businesspeople. Because a prototype is a rough model created with whatever prebuilt facilities are most readily available, prototypes are by nature filled with expedient compromises. But software that actually works—regardless of how badly—exerts a powerful pull on those who must pay for its development. A running—limping—prototype has an uncanny momentum out of proportion to its real value.

It is all too compelling for the manager to say, "Don't throw out the prototype. Let's use it as the foundation for the *real* product." This decision can often lead to a situation in which the product never ships. The programmers are condemned to a role of perpetually resuscitating the program from life-threatening failures as it grows. Like the stack in which the first 25 bricks were placed haphazardly, no matter how precisely the bricks above them are placed, no matter how diligently the mason works, no matter how sticky and smooth the mortar, the force of gravity inevitably pulls it down somewhere around the 50th level of bricks.

The value of a prototype is in the education it gives you, not in the code itself. Developer sage Frederick Brooks says, "Plan to throw one away." You will anyway, so you might as well do it under controlled circumstances.

In 1988, I sold a program called Ruby to Bill Gates. Ruby was a visual programming language that, when combined with Bill's QuickBasic product, became Visual Basic. What Gates saw was just a prototype, but it demonstrated some significant advances both in design and technology. (When he first saw it, he asked, "How did you *do* that?") The Microsoft executive in charge of then-under-construction Windows 3.0, Russ Werner, was also assigned to Ruby. The subsequent deal we struck included having me write the actual program to completion. The first thing I did was to throw Ruby-the-prototype away and start over from scratch with nothing but the wisdom and experience. When Russ found out, he was astonished, angry, and indignant. He had never heard of such an outrageous thing, and was convinced that discarding the prototype would delay the product's release. It was a fait accompli, though, and despite Russ's fears we delivered the completed program on schedule. After Basic was grafted on, VB was one of Microsoft's more successful initial releases. In contrast, Windows 3.0 shipped more than a year late, and ever since it has been notoriously handicapped by its profuse quantities of vestigial prototype code.

In general, nontechnical managers erroneously value completed code—regardless of its robustness—much higher than design, or even the advice of those who wrote the code. A colleague, Clay Collier, who creates software for in-car navigation systems, told me this story about one system that he worked on for a large Japanese automotive electronics company. Clay developed—at his client's behest—a prototype of a consumer navigation system. As a good prototype should, it proved that the system would work, but beyond that the program barely functioned. One day the president of the Japanese electronics company came to the United States and wanted to see the program demonstrated. Clay's colleague—we'll call him Ralph—knew that he could not deny the Japanese president; he would have to put on a demo. So Ralph picked the president up at LAX in a car specially equipped with the prototype navigation system. Ralph knew that the prototype would give them directions to their offices in Los Angeles, but nothing else had been tested. To Ralph's chagrin, the president asked instead to go to a specific restaurant for lunch. Ralph was unfamiliar with the restaurant and wasn't at all confident that the prototype could get them there. He crossed his fingers and entered the restaurant's name, and to his surprise, the computer began to issue driving instructions: "Turn right on Lincoln," "Move into the left lane," and so on. Ralph dutifully followed as the president ruminated silently, but Ralph began to grow more uneasy as the instructions took them into increasingly unsavory parts of town. Ralph's anxiety peaked when he stopped the car on the computer's command and the passenger door was yanked open. To Ralph's eternal relief, the valet at the desired restaurant had opened it. A smile broke across the president's face.

However, the very success of this prototype demonstration backfired on Ralph. The president was so impressed by the system's functioning that he commanded that Ralph turn it into a product. Ralph protested that it was just a feasibility proof and not robust enough to use as the foundation for millions of consumer products. The president wouldn't hear of it. He had seen it work. Ralph did as he was told, and eight long years later his company finally shipped the first working version of the product. It was slow and buggy, and it fell short of newer, younger competitors. The *New York Times* called it "clearly inferior."

The expertise and knowledge that Ralph and his team gained by building the prototype *incorrectly* was far more valuable than the code itself. The president misunderstood that and, by putting greater value on the code, made the entire company suffer for it.

⌘

If you define the boundaries of a development project only in terms of deadlines and feature lists, the product might be delivered on time, but it won't be desired. If, instead, you define the project in terms of quality and user satisfaction, you will get a product that users want, and it won't take any longer. There's an old Silicon Valley joke that asks, "How do you make a small fortune in software?" The answer, of course, is, "Start with a large fortune!" The hidden costs of even well-managed software-development projects are large enough to give Donald Trump pause. Yacht racing and drug habits are cheaper in the long run than writing software without the proper controls.