accounts (e.g., the *Wall Street Journal*), computing-press accounts (e.g., *InformationWeek*), and academic accounts (e.g., a Harvard Case Study). But before we get to those war stories—that will happen in Chapter 2—there are some words of introduction that are needed. We will answer the definitional question "What is a software runaway?" in Section 1.1. We will deal with the claims that software practice is a discipline in crisis in Section 1.2. We will talk about some related but different ideas—what happens when projects find themselves in "crunch mode," and what constitutes a "death march" project, in Section 1.3. And to close this chapter, we will present some fascinating findings of a research study that explored the notion of software runaways, and drew some diverse and important conclusions about them.

from Glass, Robert L.
Software Runaways
Prentice Hall, 1998

1.1 WHAT IS A SOFTWARE RUNAWAY?

Press reports of systems failures caused by software are often spectacular. "Company XYZ," they say, "is M months behind schedule and D dollars over budget in producing system SYS, and it looks inevitable that the project will be canceled, costing the company BUX dollars overall." And when you look at M and D and BUX, they are represented by really large numbers, the kind with lots of zeroes after them.

The main thrust of this book is to present the stories of some of those XYZs and SYSs, including how they screwed up at the M and D level, and how many BUX were really lost. As you will see in that material, XYZs represent some of the better-known companies in the business of producing systems, and the SYSs include some highly-visible projects that appeared more often in the press than their developers could ever have imagined or wished.

But before we get into those stories, there is—as we say in the software business—some initialization to be performed. That is, it is important to put those stories into an overall context. Setting that context is the role of these initializing, introductory sections.

First of all, what is a software runaway? It is

> a project that goes out of control primarily because of the difficulty of building the software needed by the system.

The implication of "out of control" is that the project became unmanageable; it was not possible to manage it to meet its original target goals, or to come even close to them. If those goals are thought of in terms of schedule—and this is the most likely kind of runaway—then the project consumed close to double its allotted estimated time or more. If the goal was cost, and usually projects that go well over budget also exceed cost targets, then the project consumed close to double its estimated cost or more. If the goal was a reliable product that met its functional requirements, then the product could not meet those targets, and in fact, all too often, failed to meet any targets at all because the project had to be canceled.

Our definition differs from a somewhat better-known definition. KPMG's says .

> A runaway project is one which has failed significantly to achieve its objectives and/or has exceeded its original budget by at least 30 percent [KPMG 1995].

As you can see, the KPMG definition of runaway is far more inclusive than ours. That is, if a project is somewhat over cost targets, 30 percent or more, by the KPMG definition that would be a runaway, whereas by ours it would not (it would have to exceed by 100 percent to meet ours). Using the KPMG definition, there would be a lot more runaway projects than we would include. The reason we mention the KPMG definition at all, given our difference with its quantitative representation, is that it comes from the only research studies we are aware of on runaway projects. In 1989 and again in 1995, KPMG performed a survey of runaway projects to determine their frequency, their causes, the remedies tried, and the effect of the runaway on the enterprise where it happened. We will present those research findings in subsequent sections of this chapter.

Why are we so fussy about the definition of runaway? The reason is this: It is now 1998, and we are only 40 something years into the history of the software field. At this primitive, early stage, the most common problem in building software systems is not the construction of them itself, but rather the estimation of the costs of that construction. Why is there such a problem of estimation? Because the software field has not made a conscientious effort to develop histories of past project costs. Because the construction of software is an extremely complex task—some say it is the most complex task ever undertaken by human beings.

Because of the lack of history and the amount of complexity, a barrier was produced that no amount of mathematical techniques and no amount of savvy, individual expertise has been able to overcome. It is all too common for a software project to fail to meet its cost and schedule targets, because those targets themselves were simply (and grossly!) wrong.

It is our belief that a runaway project is one that fails for a reason more profound than poor estimation. That is why we have raised the barrier beyond the KPMG 30 percent figure—we want to make sure that those projects that we call runaways are so called because the development effort itself got out of control, regardless of whether the original estimates were close to the correct figures or not. Often, a project can exceed bad cost targets by more than 30 percent without being a troubled or failed project. It is our intent not to include such projects under our runaway umbrella, even though the KPMG study would.

REFERENCE

KPMG 1995, "Runaway Projects—Cause and Effects," *Software World* (UK), Vol. 26, No. 3, Andy Cole of KPMG.

1.2 THE CRIES OF SOFTWARE CRISIS

It has been common over the last decade or so to see a mention of the "software crisis" in computing literature and sometimes even in the popular press. Although different writers seem to use the term to mean different things, the most common definition of software crisis is this:

> Software is always over budget, behind schedule, and unreliable.

It is important to me, however, to say here at the outset of this discussion that

> I do not believe in the existence of a software crisis.

It is important that I take that position fairly early in this book, because otherwise it would be easy to conclude that the author of a book on software runaways felt that those runaways were symptoms of, and examples from, the software crisis. And nothing could be further from the truth.

There are lots of runaways, of course. This book will tell those stories. But it is my belief that the incidence of those runaways represents a tiny percentage of all the software projects ever attempted.

It is interesting to note that there have been many estimates provided by those who do believe in a software crisis as to the frequency of such projects. But there is a fundamental problem with those estimates, they differ all over the map! The most famous numbers, derived from a Government Accounting Office study, were that upwards of 98 percent of projects failed ("less than 2 percent of the software contracted for was usable as delivered"). But there was a serious problem with those numbers—the GAO study was of several projects that were in trouble—that's why the GAO studied them—and thus it is not surprising (nor very informative!) that 98 percent of software projects that are in trouble eventually fail. (For a refutation of the misuse of the GAO numbers, see [Blum 1991].)

Many authors and speakers have repeated that GAO finding, without really appreciating that the study was about something different from what they thought it was. But even those who have not repeated that 98 percent number have come up with huge percentages of failed software projects—I have seen numbers such as 60 percent, and 48 percent, and 35 percent, in various places in the literature. It is my personal belief that there is no more reason to believe those numbers than to believe the GAO 98 percent number. The fact of the matter is, no one has really sufficiently performed a survey to tell us what percentage of software projects fail. In fact, hardly anyone has come up with an adequate definition of failure, which of course would be necessary before that survey could have any meaning. Note the difficulty we have already encountered in this book in defining "runaway," surely an easier term to define than "failure" since a runaway is in a sense more spectacular.

I am particularly incensed about these cries of software crisis and their fraudulent quantifications for two reasons:

1. There is an implication in the cries of crisis that software practitioners are the original Mr. Bumble, unable to program their way out of a paper bag.
2. There is an implication also that software practice is, in general, full of failure, and that few successes have been achieved.

My own belief is that those implications are terribly wrong. When I look around, I see a world in which computers and their software are dependable and indispensable. They make my plane reservations, control my banking transactions, and send people into space—with enormous dependability.

Why, then, are there so many crying crisis? Because they have something to gain by doing so. Some vendors cry crisis in order to sell products or services that they claim will offer a cure. Some researchers cry crisis in order to obtain funding for research projects that they claim will also (eventually) offer a cure. Some academics cry crisis in order to motivate the acceptance and reading of their professional papers that suggest a cure. Hardly a disinterested collection of people.

There is, in fact, a funny thing about the cries of crisis (if a problem with such serious implications could ever be funny). Most of those who cry crisis and are trying to sell or promote something are offering a better technology for building software. But most of the case studies of software failure find that poor management technique, not poor technology, is the cause of the problems. Thus even

if there were a software crisis, the offerings of these people would not—and could not—solve the problem.

In fact, the findings of the KPMG study mentioned earlier, and our own findings in this book, will show that, more recently, technology is becoming a more common cause of software failures. But the real irony is this: Most of those technology-based failures are caused, in fact, by the very technologies that others have proposed as solutions. The same tendency toward hype that allows these people to cry crisis in the first place also allows them to make excessive and erroneous claims for the benefits of those technologies. There are stories later in this book where projects became runaways *because of* the use of such technologies as formal methods, expert systems, and fourth generation languages. Far from becoming cures, sometimes those technologies become albatrosses, dragging a project down toward defeat.

The cries of crisis, in fact, represent something of a "blame the victim" mentality. We have already noted that, when software projects fail to achieve cost and schedule targets, it is often those targets themselves that are at fault. Thus software practitioners, laboring to achieve impossible goals, and all too often putting in tons of voluntary overtime to try to meet them in spite of their impossibility, end up being blamed for a problem that was out of their control from the beginning of the project. Study after study of software estimation in practice has shown that most often cost and schedule targets are set by marketers or customers, next most often by managers, and least often by the technologists who will do the work. Without control of their own destiny, practitioners still get blamed when things go wrong.

Because of all of the above, I would like us all to pause, at this point in the reading of this book, for a moment of appreciation. Here's to all those pioneering software practitioners who have struggled through the barriers in their paths to make our era truly the Computing Era, the half-century or so in which computing has been the dominant force in our society. May they be appreciated for their efforts, and lauded for their successes!

REFERENCE

Blum 1991, "Some Very Famous Statistics," the *Software Practitioner,* March 1991, Bruce I. Blum.

1.3 "CRUNCH MODE" AND THE "DEATH MARCH" PROJECT

Colorful terminology is de rigueur for computing people. We invent new terms and new acronyms at the drop of a hat. It has gotten so bad that we've even invented a term for that profusion of terms—"Computerese" is the word we've coined for the specialized language in which computer people discuss things.

"Runaway" is one such term; "software crisis" is another. In this section, I'd like to talk about two more, related, terms—"Crunch Mode" and "Death March."

Crunch Mode is a term used by John Boddie in his book of the same name [Boddie 1987]. It is a term describing the status of a project. A project that is in crunch mode is there due to a threat to its reaching its original targets (cost, schedule, functionality, . . .), and the project team is working very hard to try to overcome that dilemma. "We're in crunch mode, and I won't be home 'til after midnight," a software specialist might say in a call from the office to his life partner. The life partner probably won't be surprised—crunch mode tends to last for days, weeks, or even months, depending on the duration of the project itself, and the degree to which it is off-target. The term crunch mode itself was not invented by Boddie. It was invented by the software practitioners who found themselves under the gun to finish the project they were working on.

Death March is a term used by Ed Yourdon in his book of the same name [Yourdon 1997]. It is a term that also describes the status of a project. A project is a death march if the parameters for that project exceed the norm by at least 50 percent. (The parameters might include schedule, staffing, budget, and functionality.) "This is a death march project, and I hope to avoid becoming a part of it," a software specialist might say in a call from the office to his life partner. The life partner probably has heard all of this before; unfortunately, all too many software projects these days are death marches, and the life partner probably knows—perhaps more than even the software specialist—that avoiding participation is unlikely. The term death march was not invented by Yourdon. It was invented by people who found themselves involved in projects for which the only hope of

achieving those unreasonable targets was to work far harder and far longer than normal. Unfortunately, as Yourdon points out in the preface of his book, death march projects have become the norm, not the exception.

So what's the difference between all these terms? Obviously, they are all dancing around the same general subject. But there is a difference:

1. Crunch mode is used to describe a project that has an extremely tight schedule. It speaks to the pressures being felt by the project participants.
2. Death march is used to describe a project that has a nearly impossible schedule. It speaks to the oppressive smell of potential failure surrounding the project participants.
3. Runaway is used to describe a project nearing or after its termination. It speaks to the failure of the project to stay within its boundaries. Often it speaks about a project that has either already failed (usually in a spectacular way), or is about to.

A typical project to which these terms might apply could progress in the following way: The project looks from the outset to be crunch mode because someone has promised project results that are too much, too soon. As the project gets underway, project participants all too soon find themselves on a death march, trying to achieve these increasingly unachievable targets. When it becomes obvious that the project probably cannot succeed and will fail in a major way, the project becomes a runaway.

Not all projects that are death marches fail, of course (remember that the death march is the normal way of running a project these days, according to [Yourdon 1997]. Although some of those death marches become runaways, others will become successes). But all of them, successes or failures, will have functioned in crunch mode.

Most people, given their choice, would not want to participate in a death march, would not want to find themselves in crunch mode, and would never, ever, want to be on a runaway. But project managers have found a way to entice people into participation. They use a process called "signing up" (described in [Kidder 1981]), that dangles so many benefits[1] in front of the prospective participant that they simply can't say no.

Why do we have such terms and such projects? The fact that these terms are needed by the computing field speaks of the intense pressure in our era put on project completion. Usually systems projects are managed by schedule—that is, the responsible manager examines project progress against a predetermined schedule of events,[2] and all too often that schedule is unreasonably short, and those events are thus late in occurring. Because we know so little about accurate schedule estimation, and because estimates are usually made by the people who are least able to make accurate estimates (e.g., marketers and customers), it is simply the norm that schedule targets, and thus cost targets, are unreasonably short. Thus project achievement of them is at best problematic. This is, of course, not a problem unique to the systems and software field. Given the intense competitive pressures of the last half of the 20th century, workers in all fields find themselves under the gun to do more than they can in too short a time. The problem for systems and software is worsened, however, by the fact that the field is so young, and because we know so little about it compared to other fields, we are really never quite sure that an unreasonable schedule is, in fact, THAT unreasonable.

REFERENCES

Boddie 1987, *Crunch Mode*, Yourdon Press, 1987, John Boddie.

Kidder 1981, *The Soul of a New Machine*, Little-Brown, 1981, Tracy Kidder.

Yourdon 1997, *Death March*, Prentice Hall, 1997, Ed Yourdon.

1. Software personnel are motivated by things such as challenging projects and/or a chance to use new technology rather than the more traditional ones of power or money.
2. The schedule is often made up of "milestones." If these milestones are extremely detailed, they are sometimes called "inch-pebbles."

1.4 SOME RELEVANT RESEARCH FINDINGS

Research in the computing field is all too often focused on theory to the exclusion of practice. That is, computing researchers are very interested in developing new algorithms, new data representations, or new formal methods, but very seldom are they interested in the formalization of best practices or the learning experiences that can be derived from worst practices.

That is an important failure of the computing research field. It is well known in other fields that practice sometimes leads (is more advanced than) theory; I will not belabor the point here (it is belabored elsewhere, such as in [Glass 1989] and [Glass 1990]!), except to point out that the invention of the steam engine preceded the development of the theory of thermodynamics, and the invention of the airplane preceded the development of the theory of aerodynamics. In a newly-emerging field—and what field in our time is more "newly-emerging" than software?—there is a great deal that theory can learn by studying practice (some theorists even say that "theory is the formalization of practice"), and computing theorists are not taking advantage of that possibility.

All of that is prelude to this good news and bad news:

1. The bad news is that there are few research studies of runaway projects. There are case studies of individual runaways (that is what this book is primarily about), usually appearing in the popular computing press rather than the theoretic literature, but there has been little organized research attempting to study runaways in more breadth in order to grapple with the lessons that might be learned from doing so.
2. The good news is that there is one such recent study, and it is an excellent one. It is published in what to most of us jingoistic Americans is an obscure journal, but nevertheless it is a superb study of the trends in, reasons for, remedies attempted, and aftermath of software runaway projects [KPMG 1995].

In fact, that research study is actually about two such studies. Not only do we have the findings of the study published in 1995, but that study also reflects on the findings of the same study performed by the same organization in 1989. In other words, we not only can learn some things about contemporary software runaway projects, but we can learn some things about the trends in such projects.

In reporting on the findings of these studies, I would like to divide their discoveries into three categories. The first is *predictable* findings. The software engineering literature contributes considerable insight into what to do, and what not to do, during a software project. The predictable findings reflect what we already know from that literature. (It is important, of course, for research to study those predictions in order to determine what can be supported by empirical findings and what cannot. Those that cannot should be quickly relegated to the category of "old wives' (or husbands') tales," and not passed on through the literature any more.

The second category I would call *surprising* findings. There are usually fewer surprising findings than predictable ones, but in a sense surprising findings are more important than predictable ones, since they provide us with new insight into our field. And, in the case of the KPMG study, there are actually *more* surprising findings than predictable ones.

The third category I will call *trends*. Because the study was conducted in both 1989 and 1995, the author of [KPMG 1995] had a unique opportunity to present us with observations that take into account the lapse of six years between the studies.

The predictable findings are:

1. Many of the runaway projects are (or were) "overly ambitious." It is well known in the field that large projects are problematic.
2. Most of the projects failed from a multiplicity of causes. There may or may not have been a dominant cause, but there were several problems contributing to many of the runaways.
3. Management problems were more frequently a dominant cause than technical problems. But see the list of surprising findings below.
4. Schedule overruns were more common (89 percent) than cost overruns (62 percent).

The surprising findings are:

1. Survey respondents thought that there would be more runaways in the government and financial sectors, and fewer in service and manufacturing. But the survey findings found all such sectors equally susceptible.
2. Respondents were optimistic about the trend in runaways; 42 percent believed they would decrease in number, while only 8 percent felt they would increase.
3. The use of packaged software did not help in reducing the incidence of runaways. Of the runaway projects studied, 47 percent consisted of mixed custom and packaged software; 24 percent were custom software; and 22 percent were packaged software.
4. Runaway projects showed their true colors early in the project history. More than half started showing symptoms during system development, and 25 percent showed those symptoms during initial planning.
5. In spite of the above, visibility into the existence of a runaway came first of all from the project team (72 percent); only 19 percent were spotted initially at the senior management level.
6. Technology is dramatically increasing as a cause of runaways. "Technology new to the organization" was the fourth most common problem in the runaway projects. See this topic discussed also under trends, below.
7. Risk management appears more and more frequently in the software management literature. But 55 percent of the runaway projects had not performed any risk management, and of those 38 percent who did (some respondees did not know whether it was used or not), half of them did not use the risk findings once the project was underway.

The trends are:

1. Companies were much more reluctant to discuss runaway projects in 1995 than they were in 1989. The number of respondees in the newer study was half that of the earlier one, in spite of the fact that the survey population (about 250 major organizations) was roughly the same.
2. Technology is a rapidly increasing *cause* of runaway projects. Whereas in 1989 only 7 percent reported it as a cause, in 1995 the figure was 45 percent. (Interestingly, only 16 percent of respondents felt the technology was "wrong" for the job.) [KPMG 1995] concludes "Technology is developing faster than the skills of the developers." (This conclusion will be questioned in what follows.)

It is important to say several things about this study as we consider these predictable and surprising findings and trends:

1. The definition of runaway used in this study is different from that used in this book (this was discussed earlier in Section 1.1). Far more projects would be considered runaways by the definition used in the KPMG study ("a project that has failed significantly to achieve its objectives and/or has exceeded its original budget by at least 30 percent") than by the more restrictive definition we use ("a project that goes out of control primarily because of the difficulty of building the software needed by the system," where "out of control" is taken to mean "schedule, cost, or functionality that was twice as bad as that sought").
2. The survey was conducted only in the UK (there were 250 major enterprises contacted). In this book, on the other hand, most of the projects discussed are American. We are not saying that we know of any differences in runaway projects caused by country of origin; we are only saying that we would prefer to put that fact on the table in case it is discovered to be important by later studies.
3. The surprising trend in technology as a problem may be the single most important finding of the survey. As noted above, the KPMG study concludes that the problem usually lay not with the technology, but rather with the ability of the software practitioners to utilize it. Our analysis of the runaways in this book suggests a different conclusion. Although our sample is

considerably smaller than that of the KPMG study, it is clear from the projects we examined that (a) new technology was also a frequent cause of problems, and (b) the reason was that the technology was used prematurely or inappropriately. For example, one project used a 4GL that was clearly inappropriate for the large project on which it was attempted (performance was inadequate for the large number of users projected), and another made a point of using several different advanced technologies (formal methods, expert systems, etc.) and failed because those technologies were not yet ready for the kind of large project use on which they were attempted. Our conclusion is that it is often the technologies themselves (particularly either their inability to scale up or the lack of a track record of scaling them up) rather than the technologists, who are primarily at fault when technology fails and a runaway ensues.

There are two other conclusions that might be drawn from our own runaway reports in this book that are not found in the KPMG study. They are:

1. Early on, those responsible for our runaway projects often bragged about the "breakthrough" nature of the project, either in terms of its business relevance or its technical (computing) advances. There seemed to be little or no understanding by those making such claims of the perilous course they were undertaking. (In fact, for one of our runaways, a report appeared in the computing literature bragging about the project *after* it was known to have become a runaway!) In some ways this contradicts one of the conclusions drawn in the KPMG study: that many runaways are "misconceived to start with." Or perhaps it is not a contradiction—the projects were, in fact, misconceived; it was just that certain key players didn't know it yet! (This, in fact, matches the KPMG finding that runaway projects were spotted more often at the team level than the management level.)
2. Of all the technology problems noted earlier, the most dominant one in our own findings in this book is that performance is a frequent cause of failure. (This was not noted at all in the KPMG study.) A fairly large number of our runaway projects were real-time in nature, and it was not uncommon to find that the project could not achieve the response times and/or functional performance times demanded by the original requirements. This is an important finding; it has become popular now to say that with the rise in speed of marketplace computers, there is no longer a need for software people to take responsibility for product performance. This saying is manifested in several ways, the most important of which is that there is little in computer science or information systems education having to do with improving software performance, or providing for it in the first place. Since poor performance may be difficult to overcome once it has been built into a project (by using an interpretive 4GL or 3GL, for example, instead of a 3GL with an optimizing compiler), this lack could be contributing to the incidence of runaways.
3. There was one other problem in our runaways that was not mentioned in the KPMG study. A significant number of the runaway projects were in an application domain that might be called "movement of goods." As you will see in the stories of Chapter 2, there were two warehousing applications and one baggage handling application among our collection. The KPMG study, by contrast, felt that there was no one business sector that was more likely to have runaways. (It may be that the difference here is one of semantics; KPMG was talking about what might be called "industries," and we are discussing something below that level that we would call "domains." It may well be that there is no difference in likelihood across sectors (industries), but there is across domains.) It will be interesting to watch for this difference in future such studies (we hope there will be lots of them!).

There is one more, perhaps the most important, idea to utilize from the KPMG study. One of the primary findings of the study was that of identifying the "six top problems" that arose during runaway

projects. Those six top problems become the organizational, outlining device we use in Chapter 2 of this book. Using those six findings as section numbers in the chapter (2.1 through 2.6), we slot the runaway war stories that we have found into those causes. (You will note that we have also identified a Section 2.7 to cover causes "Other" than those identified by KPMG. This is largely an opportunity to list the runaways that failed from performance problems.)

And now, enough of this initialization, this introductory material. The context for our war stories has been set. Let us turn to those stories themselves for the primary lessons, and the most fascinating material of this book.

REFERENCES

Glass 1989, "The Temporal Relationship Between Theory and Practice," *Journal of Systems and Software*, July 1989, Robert L. Glass.

Glass 1990, "Theory vs. Practice—Revisited," *Journal of Systems and Software*, May 1990, Robert L. Glass.

KPMG 1995, "Runaway Projects—Causes and Effects," *Software World* (UK), Vol. 26, No. 3, 1995, Andy Cole of KPMG.

PART

2

Software Runaway War Stories

Frankly, one of the challenges facing Microsoft is that many of its employees have not suffered much failure yet. Quite a few have never been involved with a project that didn't succeed. As a result, success may be taken for granted, which is dangerous . . . When you're failing, you're forced to be creative, to dig deep and think hard, night and day. Every company needs people who have been through that.

—Bill Gates, in "The Importance of Making Mistakes," USAir Magazine, *July 1995.*

Here they are, the war stories we have been promising you in this book about software runaway projects. As an organizational scheme, we divide these war stories into several categories identified in the research study discussed in the previous section. In that study, it was found that runaway projects could be categorized by their primary cause. These causes, in order of decreasing importance, are each preceded by the section number assigned to them in this chapter; they are: