Chapter 2 Quality Ensurance

Debugging is just about everybody's least favorite activity. On the other hand, a lot of time and money goes into debugging. Rarely is it possible to estimate accurately how long debugging will take, and generally project management systems fail to capture debugging time. Thus, it bears discussing how debugging fits into the project lifecycle, and to examine whether the task of debugging is necessary.

Debugging may be hard to define formally, but every developer knows it when he sees it. It is the process developers undertake to locate the source of a program's questionable behavior. Debugging code may result in code changes, but that's not necessarily so. That depends on what the source of the problem turns out to be, when it's found, and what the impact of correcting it will be. Problems arise because of inadequate requirements, design defects, coding errors, and documentation errors. Participants can, and should, uncover whatever problems exist no matter what stage a problem is introduced. Once a problem is defined and the cause determined, participants should evaluate alternative solutions, identifying the weaknesses and strengths of each. The best solution might turn out to be leaving the software as-is.

A bug is ultimately a coding error, albeit, not necessarily simple or easy to correct. A defect is any way that a program deviates from the published specifications, user expectations, or correct computation. Since debugging is the process of uncovering the source of problems in software, and since the source may be any number of things, it is misleading to think of debugging as only applicable to bugs. At it's most general, it's an investigation prompted by some event. In this sense, debugging can be applied to any defect at any stage in the project lifecycle from requirements gathering to deployment. In this sense, defects result for many reasons and at all stages of the development process.

Starting an investigation implies two things. One, debugging is done after a project has started. Two, there must be a way to trigger an investigation. It's somewhat obvious, but still bears discussing. Unless a developer knows that the system is behaving in a way that deviates from the expected behavior, there's no way to know it needs debugging: except, of course, in the case of the most obvious program error. A number of questions follow from this, then. When should one start looking for problems in a project? How does one know when the system is not operating as expected? What are effective techniques for uncovering problems? How can one manage the process of finding and fixing problems so that the quality of software improves?

> Debugging is often difficult, time-consuming, and hard to predict. When an unexpected defect shows up, it disrupts planned work, drives everyone crazy, slows down progress. The best way to avoid these problems is to have very few defects. One good way to have few defects is to have good, comprehensive tests.
>
> —Ron Jeffries
> http://www.xprogramming.com/xpmag/PracticesForaReason.htm

To find problems in a process or product, it must be subjected to tests. When can testing begin? It can begin before any code is written. While the initial project phases such as requirements gathering, modeling, and system design are outside of the scope of this book, it's important to note that planning for testing can begin as soon as a project has been kicked off. I have read frequently that test cases should be written before any code for a number of reasons, not least of which is to keep the participants focused on the goals of the project and to encourage thorough planning. Like many FoxPro developers, I work mostly independently on projects. While that advice made sense, I was a bit lost as to how to approach it, especially when I could barely get customers to spend time planning their applications. Then a client and I worked together to learn how to do use cases, and I found they have an unexpected benefit. Not only do use cases pin down the details of a business or system scenario in terms familiar to customers, but use cases also make superb foundations for test cases[1].

Not only can the early phases of a project set the stage for testing, but they also can be tested. Indeed, it is critical that every stage of the software development process include a mechanism to identify and correct problems. It's critical because many, if not most, software defects occur before implementation begins. According to Engin Kirda in "Integrating Design and Automated Test Generation" (http://www.infosys.tuwien.ac.at/Staff/ek/projects/idatg.html), only 30% of software defects occur during implementation, whereas 70% of defects occur during the requirements gathering and design phases.

Not surprisingly, the later a defect is discovered from the time it is introduced, the more costly it is to fix. For example, finding and fixing a defect resulting from a missed or misunderstood requirement can cost 14 times as much to fix during the coding phase as during the requirement phase. Fixing the same defect found in post-release code can cost as much as 100 times more (Cigital Corp., http://www.cigital.com/presentations/testing_objects/sld008.htm). According to W. Wayt Gibbs in "Software's Chronic Crisis" (*Scientific American*, 1994), perhaps as few as one-third of the bugs introduced before release are caught before release.

The process of developing software is an integrated, iterative process in that each stage builds on the prior one. Sometimes one stage even influences the previous one. Regardless of when a defect is introduced, at least some part of the analysis of the problem will likely involve development time debugging. How one gets from one stage to the next with a minimum of defects speaks to the quality of the development process.

Lies, damned lies, and… statistics

The issue of quality is the business of software metrics, which is really just a name for having some way to objectively measure and predict the quality of software. Software development has been the subject of many attempts to quantify and predict quality. For example, a casual search on Amazon for "software metrics" results in 41 hits. The software metrics clearing house at http://user.cs.tu-berlin.de/~fetcke/metrics-sites.html includes a number of links

[1] There are many resources for learning about use cases, which I use, and use case diagrams, which I don't find as useful. The Internet is a free and quick place to start, simply by searching in your favorite engine for "use case." However, a site with several papers on the subject is http://members.aol.com/acockburn, as recommended by Martin Fowler in *UML Distilled, Second Edition*.

(24 as of this writing) to sites with publicly available publications on the subject. I encourage you to make a study of some of the issues in software metrics. In particular, compare different methods. In general, formal metrics are useful when a project or team is facing a particular problem. Applying the correct metric can help a team overcome a problem and measure the success of process changes. However, when metrics become institutionalized and the process is more about the metric than about the product, metrics are less useful and perhaps even cause trouble in the development process. In addition, maintaining the procedures for a formal metric is difficult, if not impractical, for small teams and solo developers. Yet these developers also are concerned with quality and improving quality. There is a lot an individual programmer can do. After all, she owns her code until she incorporates it into the team's work or delivers it to the client.

So, there are situations where formal metrics are impractical, or where a metric has been adopted because of a management trend rather than to meet a specific challenge. Human nature inclines to fudging, slacking, or malicious compliance—a wonderful term I first heard while working for an oversight agency. When the agency enforced reporting requirements, some contractors took the approach of providing an overwhelming amount of information that buried the relevant data under an impossible weight. The goal of a process is always to build a high-quality product and ship it. It is not to have a nice-looking QA document. QA programs do not ship products, but they do help development teams ship high-quality products. That said, QA programs and software metrics arose for a very good reason: The perception is that "software quality" is an oxymoron.

While many quality assurance systems may be effective under different conditions, I believe that what is more important than which system you follow, is that you have one and follow it. When a deficiency becomes apparent (and all systems can be improved), be willing to focus objectively on what is ineffective and make tactical changes. It's likely that deficiencies occurred as a side effect of a process, so identify the reasons for them and institute a mechanism to avoid it in the future. Steve McConnell provides a handy quality checklist in *Code Complete*. He has also put the checklist on his Web site at http://www.construx.com/doc/chk04.htm. McConnell points out the importance of quantifying what exactly the measure of "quality" in a system is. Is quality measured by efficient memory use? Or is it the maximum time to enter a data set, or query speed? In order for the quality measure to be meaningful, it must be objective and not subjective. Contrast the difference between "checking a customer's purchases should be fast" and "scanning three items should be able to be done in under 10 seconds, and the receipt should print within three seconds of posting the purchase." The first is subjective and not quantifiable. There is no way to know when, or whether, you've been successful. The second example is objective. It's harder to come up with objective goals, but it is easier to create tests to measure the goals, and to agree upon the results of the tests. Goals should be unambiguous. In addition to quantifying goals, McConnell recommends identifying the purpose of the goals. This will help develop priorities in case two goals are contradictory. For example, a goal of entering one item in a grid in no more than five seconds combined with a goal of validating every item entry against a complicated set of lookup tables, on a slow LAN, may not both be achievable. If one knows the purpose of the goals—for example, to minimize the customer's wait for a receipt, to free the clerk to talk more with the customer, or to help unfamiliar users enter correct data—then programmers and designers can concentrate on solving the problem with the highest priority.

There are many predictive metrics for defects, including lines-of-code (LOC) measures and complexity measures. Some were developed as long ago as the 1960s, when languages and the applications written were different than a Visual FoxPro developer would probably write today. We are writing more applications with larger data stores that communicate with multiple back ends, perhaps over an intranet or the Internet. The applications have graphical and non-graphical elements, and they support users with accessibility limitations. We're using more third-party tools and frameworks, and, as a community, FoxPro developers are making the transition to object-oriented development.

Measures, or specifically, predictors of bugs per line of code and where bugs occur depend on many variables. Unless an analysis accounts for these variables and clearly states what the variables are, quality statistics should be taken as a guide and not a hard and fast rule. Variables include definition of bug, experience level of the programmers, type of project, language, development environment, process, and so on. What these numbers illustrate, if nothing else, is that the effect of bugs on the process can be significant and certainly feel significant whenever a test run is unsuccessful.

Today

As far as we know, our software has no undetected failures.

—Anonymous newsgroup poster

Whether metrics are useful for your organization is an issue and subject outside the scope of this book, which focuses on what developers can do right away on a modest scale to improve their debugging process. It's a bit of a cliché that the best way to debug programs is to avoid putting bugs in them in the first place. Like defensive driving, which is the idea that the best way to avoid an accident is to anticipate how one can happen at any moment, that's easier to say than do. Testing, debugging, and good coding practices are inseparable when it comes to building quality software. Even when one has worked from the start to avoid writing bugs, some defects are inevitable. The goal should be to find as many as possible as soon as they are introduced.

Testing and debugging during development qualitatively improves the experience of using the applications developers build. This assertion sounds like common sense. Evidence supports the idea that modules with more bugs reported during development have fewer bugs reported after release[2]. The evidence isn't overwhelming—certainly more study could be done—but it is interesting and feels correct. A possible explanation is that those are the modules that are tested the most and, presumably, debugged. It is perhaps less obvious why the experience is better for testers and why that's important. The fewer egregious bugs—fatal errors, corrupted data, or faults—the more time the testers can spend on robust tests of capability, usability, and correctness (of calculations, for example). The goal is to end up with a product that testers verify against a test plan and test for truly extraordinary conditions. It frustrates a test plan if

[2] From Fenton, Norman E., Neil, Martin, Software metrics: successes, failures and new directions, *Journal Of Systems And Software* (47)2-3, pp. 149-157, 1999. http://www.agena.co.uk/new_directions_metrics/HelpFileWhat_you_can_and_cannot_do_with_.htm.

the first fatal bug is reported 10 minutes after tests begin. In this I speak with the voice of hard experience. It's frustrating for everyone, of course, but it is a particular waste of a tester's time, which can be expensive.

There is no firm analysis of the impact of bugs on software, and what analysis has been done has been done against ever changing processes and tools. A survey of the literature finds references to anywhere from 3 to 5 bugs per 100 lines of released code for good programmers, which seems atrocious. However, that also equates to a 95-97% accuracy rate. Can the accuracy be 100%? Even if code *is* 100% accurate, it is impossible to certify that it is. Even if one could certify it so, what does 3 bugs per 100 lines of code mean? Are they "big" bugs or inconsequential? Is it a bug only if someone other than the programmer sees it? Do the bugs we fix in the normal course of writing code count? Were the mistakes I made writing this sentence typos? These are just some obvious criticisms of LOC. According to Fenton and Neil, neither the number of lines of code nor the complexity of code is an accurate predictor of defects. In any case, it's impossible to certify code as 100% correct because of the variables involved between all components both internal and external to the software. It is impractical to do so because of the pressure of releasing software in a suitable—as in, profitable or useful—timeframe.

One conclusion I've drawn is that some bugs are inevitable and that debugging is simply a part of the job of programming. This approach has freed me to look at debugging as a normal part of the process of programming, which means it's a process that can be improved upon and that is deserving of quality support tools. Why, then, does time spent debugging feel like a waste of time? Why doesn't it feel like something we should be doing? After all, debugging, especially during implementation, is great. It means that the product is on the way to being one step up in quality, and that's a good thing. Right?

I contend that debugging often seems unproductive because the amount of effort put into tracking down the source of a defect is often far out of proportion to how long it takes to fix. If the cause, or the solution, is complex, schedules and budgets may be seriously impacted, other programmers' modules affected, or customers made unhappy. A simple cause is rarely a comfort since one knows it would have been better to avoid the bug in the first place. Debugging unfamiliar, complex, or undocumented code is particularly frustrating. Everyone is familiar with the "build it from scratch" syndrome. This seems to be an occupational hazard. Programmers want to rewrite code: others' and their own. It seems easier to write it anew than to trace through and understand the existing code, or one simply knows more today than yesterday.

One's own recently written code is easier to understand than either unfamiliar code or old code. How much easier is it? An interesting experiment would be to ask developers to describe the functionality and logic of a recently completed routine. The experiment would have the programmer do the same thing after 6 and then 18 months. Perhaps one study could let the programmer review the code for a short period of time before writing the description, and another would ask them to write the description based on rerunning the module from the user's point of view. It would be an interesting result set, and it would be instructive to include a variety of skill sets, languages, and development styles.

Debugging also feels unproductive when one has to track down and fix the same bug, or the same sort of bug, repeatedly. Sometimes even in the same code. Finally, debugging is frustrating when it interrupts the flow of work. This is probably one reason bugs introduced and fixed during development are faster and cheaper to find and fix. The interruption to the flow of programming is minimized.

Test early, test often

It's heard often. Test early, and test often. A not-so-pithy corollary to this maxim might be test early, test often, and fix what's found. Testing alone does not improve quality. Debugging alone only documents defects, which may be the best short-term solution available. However, quality is ultimately assured by fixing defects.

> Computer software is the brightest of bright spots on the American economic landscape, a consumer product evolving in a floodtide of innovation and ingenuity, an industry that has barely noticed the recession or seen any challenge from overseas. Bugs are its special curse. They are an ancient devil—the product defect—in a peculiarly exasperating modern dress. As software grows more complex and we come to rely on it more, the industry is discovering that bugs are more pervasive and more expensive than ever before.
>
> —James Gleick
> "Chasing Bugs in the Electronic Village"
> http://www.around.com/bugs.html

Why test early? Why not wait until the end and test an application all at once? As I mentioned previously, Fenton and Neil present evidence that the more a module is tested before release, the less likely it is to have bugs reported post-release. Finally, defects are cheaper to fix the sooner they are caught. Period. Any defect that makes it into production code will have escalated in criticality simply because a user found it or will have to upgrade her software when a patch is issued. Any disruption to a user is serious.

How often should you test? Test new code always. It is possible to argue that quality in software *has* improved over the years—the applications I use do more and crash less than ever before. Yet the perception is that quality is still the same awful mess. Using Visual FoxPro, an object-oriented language, facilitates writing encapsulated code, thereby achieving one of the long promised benefits of reuse. Reusing tried-and-tested code improves software quality. The trouble with reuse comes when code is reused inappropriately. This can lead to the temptation to sneak in special conditions. Instead, step back and reconsider the design. If a routine or class can't be reused as-is, then perhaps it shouldn't be reused at all. There are usually other solutions. Object-oriented design principles and design patterns are a worthwhile study for this problem. In any case, the risk of changing existing, encapsulated code needs to be weighed as a risk. It is no longer strictly code reuse but modification that has a cost associated with it of retesting the module—thoroughly.

Once code has been tested and is correct, however, will it always be correct? Not necessarily. Retest modules whenever any code has changed. In addition, retest whenever reusing code in a new environment, and it's helpful to retest legacy code whenever a fundamental change is made. An example of a fundamental change is when a form base class has a save method that is called before the form closes. Any change in such fundamental code should prompt a smoke test and an integration test. I go into detail on these tests later in the chapter.

But… I have a test team

Your job, I presume, is to program—not test. You may work in a shop that has a formal test team, or a shop that measures productivity in lines-of-code written. I hope I've convinced you that testing and debugging during the implementation phase is cheaper and more efficient than debugging later in the process. If you do not have any explicit project responsibility for schedule or budget, then the benefit to you may seem tangential. If a shop is so ruthless and short-sighted that taking time to test and thoroughly debug your code really does constitute a risk to your job, then it's possible the job is already threatened either because of the whims of an unreasonable management team or because the company will be out of business through poor customer support. This is an extreme case, however.

There are several arguments to be made for persisting with rounding out your immediate (programming) responsibility by verifying your product. All professionals have a responsibility to do what's right for quality. It's not enough for a bridge engineer to use substandard materials because a client demanded it. But, really, whom does it hurt? Surely, the testers will find the bugs, won't they? The bugs will be reported and then there will be time to fix them. An operating procedure that relies on these notions is a good indicator of a project and a team in trouble.

Joel Spolsky tells a perhaps apocryphal story of a Microsoft Word 1.0 developer who coded a function that was supposed to calculate line height. He didn't have time to write it properly because the project was far behind schedule. Since debugging wasn't part of the schedule, however, programmers were writing buggy code, counting on time to fix bugs after testing. So, the programmer in question just wrote a function that returned the value of 12. Since this was correct most of the time, the code made it quite far into production before it was caught. As James Gleick points out in "Chasing Bugs in the Electronic Village," the tactic of delaying testing and debugging wasn't successful—customers were more than a little inconvenienced. According to Spolsky, Microsoft changed its policy to emphasize fixing a bug before going on to write the next bit of code. In other words, they embraced developer testing and debugging as part of the development process.

If developers should test their code, what kinds of tests are appropriate? John Robbins discusses the "smoke test" in *Debugging Applications*. A smoke test is a list of minimal features an application must support. A developer performs smoke tests by building a private copy of the whole application and testing his new code in context before moving on to the next task, in the simplest case. A development team conducts smoke tests on the same set of features, or perhaps a more extensive set that includes specifically the new work that's been done, before deciding the build is ready to go to testing.

A smoke test compares an application's functionality against the basic points of what it is supposed to support. Any changes to code should be checked against these points. It's surprising how often a core piece suddenly stops working. If you don't remember the last time it was working, you have a hard time knowing where to look for what broke it, and the more changes you're likely to lose by rolling back to an earlier version. Smoke tests fit into the software project process during implementation or prototyping whenever a developer has written a unit of code that he wants to check against the specification.

Unit testing is like a smoke test, except that it should test the code against the complete specifications for the code. Unit testing can be incremental ("test often"). For example, when you create a new class, test each method as you write it. You can save your tests to retest the class as a whole before turning it over for project integration. You can also reuse these tests when retesting the class after making changes.

Is one kind of test enough? Probably not, according to Steve McConnell in *Code Complete*. It is likely that adding a second kind of test to your protocol significantly and effectively improves the quality of your product. In particular, McConnell mentions studies that have found simple code reviews are 70% more effective than unit testing and less expensive. If two tests are better than one, code reviews can be a great addition to your repertoire. They have the additional benefit of letting colleagues share tips and tricks.

Unit testing includes glass box and black box testing. Glass box testing, not surprising, is testing that is done while specifically examining the code that is being tested. You check whether all the lines of code are being tested during glass box testing. You also monitor the effects of tests, such as examining variables, for example. With glass box testing you might even change the state of the routine being tested. Glass box testing uses the traditional debugger tools: the trace window, debug output window, call stack, and locals window.

Black box testing examines the output of code without looking into the details of what happens internally to the routine. This means that you have to have some way to input data and get results. This is one of the reasons that it's important to consider how you will test your code while designing and implementing it. In short, to make testing (and debugging) easier, always include some kind of a return value in every procedure or method. Even if the return is a generic success flag, and it seems that success is guaranteed, there is nearly always some way that a routine can fail. If you have a return value, it's sometimes easier to set breakpoints for specific methods or procedures.

Programmers should also perform integration testing, which tests whether the code works correctly with the larger code base into which it fits. Integration testing fits into the software project process during the implementation phase and subsequent to successful unit testing. Integration testing checks whether the new code breaks the existing code base. An example of how code might pass a unit test, but fail an integration test is common in modifying an existing module. For example, let's say a routine has been written that has different branches depending on what parameter has been passed in. If you have a requirement to add a branch to the code (not a great example of a well-designed module, perhaps), then it might be that you'll inadvertently change the return value to a data type that calling code will not expect. Code should never be incorporated into the code base until it's passed its integration tests. Of course, this means that there has to be a clear idea of how the new code will be used in the system.

Robbins writes that he has participated in teams that used the convention that the person who checks in code that "breaks the build" becomes responsible for managing each subsequent day's build… until the next person breaks a build. A build is the term used for creating the application—commonly the EXE or APP. On some teams, each night's builds are smoke-tested by the test team to make sure the application is still working with regard to at least its basic functionality. I've been guilty of sending an update to a client, customer, or colleague that I thought was working but that broke as soon as the recipient tried to use it because I'd forgotten to update a related piece of the application. This is not only embarrassing, but it's a waste of effort, budget, and schedule.

System and beta testing is done by the test team or individual—a party other than the developer should do this testing.

When are you done testing? When your code compiles without any errors, and when it has passed smoke tests, unit tests, and integration tests, you're ready to move on. If any test fails, then that is when debugging starts (or continues).

"Houston, we have a problem."

Tests by themselves don't remove defects. First, it's necessary to understand what the results of the tests show well enough to be able to examine, diagnose, and correct discovered bugs.

> "I don't know if I would use that word," a Microsoft support engineer said.
>
> "What word?" I replied innocently.
>
> "You know—that three-letter word you just used."
>
> Of course he wouldn't use it. He's under strict instructions never to say "bug" to a customer. In the official parlance of the world's most powerful software company, when a product is defective, one may speak delicately of an "issue." This could be a "known issue" or an "intermittent issue." Then again, it could be a "design side effect" or "undocumented behavior" or perhaps a "technical glitch." Excuse me while I go powder my nose.
>
> —James Gleick
> "A Bug By Any Other Name"
> http://www.around.com/microspeak.html

Debugging during development

In the opening paragraph of this chapter, I say that there is a lot of development time spent debugging. How much time is "a lot"? Unfortunately, but perhaps not surprisingly, there is no easy answer to that question, and most answers interrelate testing with debugging. Answers, however, range from "any time is too much time" to "60-70% of development time" (ParaSoft, http://www.parasoft.com/papers/bugfree.html). According to Steve McConnell, debugging can represent as much as 50% of development.

Debugging during development means less debugging later. At the very least, code you are working on currently is easier to debug than code you wrote two months or a year in the past. As I write code, I think about situations that can result in errors in my code. Am I making an assumption about a parameter type? Am I assuming a table will be open? Those vulnerabilities point to potential test cases and preventive code.

The Task List that Visual FoxPro 7.0 introduced is a good tool to make notes of test cases you think of as you're coding.

"Bug" is used casually to mean a problem with program code—usually still undetermined as to the exact cause or fix. The way a program behaves (or doesn't behave) that is in contradiction to the way it is supposed to behave (or not behave) is a bug. Bugs can exist in documentation so that a program doesn't operate the way the documentation says it should. Bugs can exist in the way a design choice results in users consistently making incorrect choices or being confused about the intent of a prompt, for example.

There are defects that are faults, errors, or, sometimes, features. Dieter Kranzlmüller says, "A failure is the inability of a system (or one of its components) to perform a given function within given limits. Additionally, a failure is often defined as a loss of some service to the user" (http://www.gup.uni-linz.ac.at/~dk/thesis/html/problema3.html#828911). Errors (incorrect calculations, for example) and failures are the result of *faults*. A fault is a mistake introduced into the application by a programmer with incorrect logic or a poor implementation of a design feature. Faults may be dormant or unobtrusive, or just rare for a long time, even years. This is, at least partly, why the literature refers to the bug potential, rather than the bug rate.

Effective debugging starts by recognizing that defects are errors in the original requirements or design, in implementing the design, or in coding. Defects in code can be introduced by modifying code or as side effects to changes to other code, as when a new use exposes a hidden bug. Defects can result from running the same code base in a new environment. In short, *every defect is the result of some change to the system*. When searching for the cause of a defect, it is critical that only one thing be changed at a time so that 1) you know whether the change had the effect you expected and was solely responsible for the defect, and 2) you can roll back any changes that are unsuccessful.

Debugging test versions

Every so often, a build of all the code the team is working on will be shipped to the testing department. That might be nightly, monthly, or at the end of a project. Even though it's a test build, treat it as if it's going to the user. The testers will be looking at it from that point of view.

What testers will expect is robust code with whatever functionality the development team has said it should have. What you can expect in return is an extensive and knowledgeable report of any problems or questions. Where a user might not be able to tell you how much RAM they have, your tester should be able to tell you operating system, version, service packs applied, and specifically what steps they took to get the error including any sample data if that's applicable.

Debugging in post-release or maintenance

Debugging production code is more difficult than debugging during development. It's also politically more sensitive since somebody besides the programmer has probably witnessed the problem. Debugging now has a significant psychological component.

How clearly the program informs the user about what has occurred and recovers from an error affects how difficult the problem is to debug. If you don't include custom error messages, the user will see the default error message—no line number, either, unless you ship your EXE with the debug information compiled in, which is not the recommendation for Visual FoxPro applications. The user will be, by definition, frustrated and probably impatient when an error occurs. Many users do not bother to write down messages, especially since many program errors are cryptic.

Designing with diagnostics in mind

One part of a program's responsibility is the obvious functionality. Another part is maintainability, and part of maintenance is how accessible the logic is for debugging and enhancing. Later chapters discuss using Visual FoxPro features to aid debugging. Any useful technique ultimately involves getting information out of the system when it's needed in a reasonable format.

If debugging becomes an integrated part of the software development lifecycle, then elements can be designed in from the beginning that support debugging and therefore minimize the cost of debugging. This can mean designing in a message path for objects to communicate with the system. Consider whether messaging can use an interface or not. For example, is the code running in a server-side component? The University of California, Riverside, Computer Science Department recommends "including a dump member function for each object class" (http://www.cs.ucr.edu/content/documentation/docs/programming/design.htm). For example, error methods could call this dump method and get back a string representing the current state of the object. The string can be displayed to a user, written to disk, or passed to an error handler.

When are you done debugging?

How confident are you that you found the source of the defect? If this one item is changed, does the problem change or go away? Have you identified at least one solution? If you have implemented a solution, have you tested it? If you aren't able to implement a solution right away, have you identified a workaround or the steps to recover from the problem? Have you documented the problem, source, status, solution, and action taken? If the effort was frustrating, consider ways it might have been better.

It's important to test that a solution is correct. If I've done much experimenting to find the problem or solution, even though I'm careful to roll back changes, I often revert to my source control version. Then I run the error test, make the planned change, and rerun the error test. This assures me I've fixed the problem. According to the University of California, Riverside, more than half the bugs introduced by professional programmers are introduced during the debugging process (http://www.cs.ucr.edu/content/documentation/docs/programming/debugging.htm). Whether this is the result of forgetting to remove test code or of coding errors, it shows the importance of thoroughly retesting after making debugging changes.

Finally, have you considered whether the problem occurs anywhere else in your application? For example, the following code sample has a latent bug in it that is not immediately apparent from casual reading.

```
USE (HOME() + "SAMPLES\TASTRADE\DATA\CUSTOMER.DBF")
SET ORDER TO Customer_I
LOCAL lcSetOrder
lcSetOrder = SET("ORDER")
IF !EMPTY( lcSetOrder)
  SET ORDER TO &lcSetOrder
ENDIF
```

The bug occurs in the combination of `SET('ORDER')` with a space in the path of the table's file name, as is the case with the default Visual FoxPro installation directory, which in this case is in \Program Files\. I would find this out by examining the value of **`lcSetOrder`** when the **`SET ORDER`** command causes the error "Command contains unrecognized phrase/keyword." As a part of debugging this single bug, I would ask myself whether I use this convention in other parts of my application and, if so, evaluate the cost of fixing the problem against the cost of not fixing all instances. I don't start fixing multiple occurrences of one problem until I've tested one thoroughly. Only after you've confirmed that a defect has been fixed (or carefully documented) should you consider moving on to write the next new bit of code.

Risk

Risk management is the process by which the cost of an action is weighed against the risk of taking no action. Risk is measurable with a number of criteria, and risk management is only as good as the criteria and metrics for measuring it. Today's feature may be tomorrow's bug. For example, the millennium bug resulted from trying to save memory and disk space, and time doing data entry. When faced with a clear diagnosis of a bug and a likely solution, come up with at least two alternatives, even if there seems to be only one. The choice to do nothing is frequently overlooked, but can always be weighed. The reason to have a handle on your options is to manage the risk inherent with making any changes compared with making no changes.

Some elements that go into managing the risk of code changes include the following questions. What is the effect of doing nothing? This means, what's the impact of the bug? Is it something that only 5% of your users will ever find but will put the project 25% over budget? Is it something that will cause 90% of your customer's monthly accounts receivable processing to be out of balance? Every time you find and fix a bug, you can make this assessment. From the obvious, a syntax error the first day you're coding on the project, to changing the name of a base table in a database container an hour before FedEx is supposed to leave with the CD for headquarters. Every bug has a cost, both to fix and to leave alone. Whichever is the lesser cost is, usually, the correct action to take. However, every bug, even ones too costly to fix, should be documented, along with the reason for not fixing it. In the case of the infamous Pentium floating-point bug, Intel decided too few people would ever encounter the bug and refused to acknowledge the problem. Then, when they acknowledged it, they refused to acknowledge that it was serious, and then that it needed to be fixed. Intel's initial assessment was, arguably, correct, but the approach they took with their customers was not, as they eventually decided.

The previous **`SET ORDER`** example is from a legacy application I've worked on that predates long file names—an example of yesterday's feature leading to today's bug. Even though the solution is simple in this case—I can use **`ORDER()`** instead of **`SET("ORDER")`**—the team had to weigh the risk of making extensive changes in established areas of the code base during final testing. It was decided that the risk of introducing more bugs or delaying the schedule for more testing was higher than for not making the change at that time. The situation and solution are documented and planned for a future release. Whether the choice is to implement a solution or not, there will also be a cost to the customer. A change that means an entire application's code base must be searched for a specific, common condition will necessitate retesting the entire application, and testing is expensive.

Measure twice, cut once

I once worked in a cabinet shop as a detailer. The detailer translates a layout and design into a cut list and construction instructions for the shop to build the cabinets. I worked with an experienced craftsman a few years from retirement—a curmudgeon if ever there was one. Bob built some of the custom cabinetry. My boss frequently dashed through my office demanding that Bob speed things up (Bob wasn't in my office). I learned two things from this particular relationship. First, Bob took however long he was going to take. Second, even though he took longer to build a cabinet than other craftsmen, he built things *once*, only. None of the other craftsmen could claim this. One or two cabinets were returned to the shop from nearly every job because other people cut corners or rushed through their tasks to meet schedules. Bob told me once when I asked him about this that carpenters have a saying, "Measure twice, and cut once."

The analogy fits debugging and programmers in a couple of ways. Debug a problem until you've identified the real cause, corrected it, and tested the correction. It's especially hard for junior programmers to resist being bullied by external pressures while not resorting to obstinacy that can be equally as frustrating as bugs. Experience helps balance these extremes. Unless one's ambition is to always work with other people's designs, a programmer will have to eventually step into a critical role. My experience is also instructive in that Bob had to be firm repeatedly. Once wasn't enough. No matter how many times my boss acknowledged that Bob's work was better and in the long run more cost-effective, the next project went the same way.

"Measure twice, and cut once" is the equivalent of taking the time to identify the real cause of a defect and the best solution instead of writing code around the symptoms of a problem. This approach might *feel* like it takes longer in the short term, but it is often more effective in the long run. Otherwise, it is likely you'll be debugging that same problem again, or it may be even harder to find the real source when it starts causing different behaviors. Another problem with this approach is apparent when you find that the solution is difficult to change later because other work has subsequently been built that depends on the quick fix. I've never mastered the art of the quick fix even though I have tried to cultivate a bit more of the fighter pilot mentality. I'm more of a Bob. In my experience it's usually just as easy to solve a problem correctly than to kludge a temporary solution. It is probably wise, in this regard, to know what your particular strengths are as a programmer.

In some cases, it won't be possible to identify the real cause of a problem with complete assurance. In this case, weigh the option of taking no action. If the error doesn't cause fatal errors, and if there is a reasonable workaround, patching over a problem without finding the cause may introduce even more bugs, which makes finding the real cause harder, and misleads the team into thinking the problem was solved. If the action is to do nothing, then document the debugging process: What was tried, what were the results, and what hunches do you have about where to look next? This saves having to retrace old ground when the issue next arises. Sometimes, too, in writing notes, I've seen where I hadn't done what I thought I had and was able to find the problem. I had a bug in my debugging process, as it were.

Bug tracking

I mention that documentation is as important during debugging as it is elsewhere. It's probably just as unlikely to be done, especially with pressure to find and fix a bug.

Documenting is important for several reasons, and it can save money—if not on the current project, on a future one.

Document code you write to test hypotheses about bugs. This will help you find and remove code that should just be temporary. Failure to do so can introduce new bugs or cause embarrassing messages. Who hasn't left a message like "I have no (*#(@ idea if this will work" in a production application? Document code that is written to solve a problem: what it does and why it was needed. Most programmers have revisited mysterious code and wondered why it's there. It may not appear to do anything, but the fear is that it's some bit of special glue holding a bug fix in place. The time spent wondering about the code will exceed the time it takes to document the code. Document the bug in project documentation so that the rest of the team can learn from it, and so that you can remember how you fixed it when you see it again in a few months or a year.

Source code control

Source code control is a general term for a process that saves versions of a code base. This can be as informal as a careful use of WinZip or as formal as third-party tools such as Concurrent Versions Systems (CVS) or Microsoft Visual SourceSafe. The former (http://www.cvshome.org/) is a free source code system. The latter (http://msdn.microsoft.com/ssafe/) is included with Visual Studio Enterprise edition[3].

Whatever tool is used, a careful procedure is critical to using source code control successfully. Source code control aids debugging. I recommend using a formal source code control system since they can be used effectively by solo programmers as well as by teams, and since source code control systems offer versioning specific capabilities such as labeling, version comparisons, and histories.

Use Visual SourceSafe either from within the Visual FoxPro IDE or from the Visual SourceSafe manager. Any type of file can be tracked in it, and it can be used over the Internet through virtual private networking or in combination with Source OffSite (http://www.sourcegear.com/sos/), which optimizes remote access to Visual SourceSafe.

If you don't use source code control, it might be for a variety of reasons. My reason was that I didn't want to bother installing and learning it. My projects had only one or two developers, and WinZip was adequate. Once a third programmer was added to a particular project, our team decided to move to Visual SourceSafe and we have never looked back.

Visual SourceSafe specifically supports debugging by being a repository of versions. Any file can be rolled back to an earlier version. If the developer who checks in a file is careful to comment each check-in, a history of a file will give the team a good idea of what may have broken in a particular module, if you're in Cancun on a well-deserved vacation. My technique for Visual SourceSafe is to check in known good builds of my code so that each comment will be specific to what has changed in the file.

Irreverent evangelizing

No, no, not to *you*. After all, you are reading this. It's my experience that bugs rarely occur because programmers don't care about them. Programmers generally loathe bugs. And almost

[3] As of this writing, it's not clear what purchase options will be available for Visual Studio .NET.

to a person developers are stubborn enough that there's no way they'd let a bug defeat them. Bugs exist in software for any number of reasons other than apathy. Programmers can work within the system to insinuate, encourage, spread, and otherwise evangelize good processes.

Bugs happen, even if your users haven't found one in a long time. There is no way to certify a program of any useful complexity as bug-free, and if you could, no one would buy it because it would have cost an infinite amount of money to develop and would never be ready. In addition, there's a fine line between a bug in code and a user's perception of a bug. If a user's program crashes because of a printer driver, and other programs don't crash using that printer driver, you can quarrel with her until night falls that the bug is in the driver, and it won't matter. Browbeating users is rarely effective at anything but ensuring they will avoid calling you back, which is not optimal for job security.

Bugs exist because of schedule, budget, and process pressures. Bugs exist because the tools we use have bugs: the operating system, the language, and the video drivers. Bugs exist because systems are complex and interactions between components are sometimes unpredictable. Bugs exist because a scientific process for software development is notoriously difficult to pin down. So bugs exist in production software because customers want something, *anything*, in their hands. However, customers also dislike buggy software. So, there seem to be divergent but concurrent goals.

Bugs also exist because the time-to-market pressures in the computer industry are intense. The pressure to ship is measured in months rather than years, and yet Joel Spolsky makes the case that good software takes 10 years to build (http://www.joelonsoftware.com/stories/storyReader$368). No one in his or her right mind expects to have 10 years to complete software, however. So, logically, successful software evolves over time. It may even be necessary for it to be in use, real use, in order for it to develop robust characteristics.

Given all of that, what can be done other than wringing hands and hiding in cubicles or snarling at users when they report bugs? Other than keeping a sense of humor and a dash of humility, programmers can evangelize to their bosses and colleagues about the process. Learn how software projects work and develop ways to work the process into the corporate, or customer's, culture. Let people know, gently, what debugging and testing is, what it accomplishes, and what it costs. Then adopt and hone skills that avoid bugs in the first place, and help find and squash bugs effectively when they do appear. Evangelize the importance of requirements gathering and design. Not once, not twice, but repeatedly. It's a cultural shift and will require the patience of a stream wearing a sharp stone round. It's a slow process but effective. Finally, you can rate your team according to Spolsky's own metric for software team quality at http://joel.editthispage.com/stories/storyReader$180.

The best offense is a strong defense

If the majority of software defects is in a product by the time coding starts, then it is reasonable to presume that debugging is a beneficial process throughout a product's lifecycle—the earlier the better. Good debugging skills are good problem solving skills, and, thus, can be applied even to the early stages of software projects—requirements gathering and design.

Testing and good coding practices are complementary to debugging. Early effective testing finds defects before customers see them, and the earlier defects are uncovered, the cheaper they are to fix. Given that developers will continually improve their programming skills, and since it's cheaper and more cost-effective to fix bugs as soon as possible after they are introduced, it makes sense for programmers to test and debug their own code throughout the implementation phase. Even where the work environment isn't conducive to formal testing and debugging, there are ways to introduce good habits if only to improve one's skills enough to seek out that dream job.

Even if I achieve my goal and convince you that debugging isn't an annoyance or tangent, but is, rather, an integral part of programming, debugging is often frustrating nevertheless. Anything that minimizes the impact of debugging is a good thing. However, whistling past the graveyard, pretending no bugs could have possibly crept into *my* code does not qualify as a quality assurance program.

If debugging is a full partner in the process of developing software, it should be part of the project planning, but it is hard to quantify. It's hard, but not impossible. If you use a timer to log project time, set up a debugging task for significant efforts. Use the research that shows that debugging can be 50% of development time to estimate how long you will need for debugging, even if only for your personal information. Review difficult debugging sessions to consider how the debugging approach you chose worked, or didn't work. Is there a process or technique that would avoid similar bugs in the future? Even a programmer who is not a designer or project manager can have an impact on the quality of her product if she considers her coding techniques and how they can be improved.

Clearly and calmly explain the different aspects of programming—coding, testing, debugging, documenting—and put each part of the process in business or layperson's terms. Prepare to be able to explain why each is important. I'll wager most programmers have had the experience of a customer saying, "You're the expert, you should know all this, I'm not paying you to learn it and I pay you too much for you to make mistakes." Analogies can be helpful, especially if your customer's process has a natural affinity with the software development process. Doctors? Try an analogy to diagnosing what's wrong with a patient. Counselors? Investigating the source of a conflict between two people. Lawyers? Both programmers and lawyers are experts but the domain is too large for any one person to have it all in mind at once. All three disciplines require research, strategies, and specialists.

Managers and clients can be anxious when a developer seems to be sitting and staring at his monitor, muttering, not making any visible progress, or, conversely, tearing his hair out and complaining about all the problems he sees in his code. Better, I think, is the image of an emergency room doctor with a professional and reassuring manner. An ER doctor remains focused on the highest priority issue until a patient is stabilized, at which time other problems can be addressed. In our case, the immediate priority is diagnosing the cause of a defect, or bug, in an application, so that the defect can be corrected or mitigated. Debugging is a matter of remaining calm while examining the patient, our software, exhibiting a problem, and identifying the most effective approach to correcting the problem.