Chapter 24 Bug Reporting and Application Feedback

"I'm going to take a bath, Hobson."—Arthur
"I'll alert the media."—Hobson
—Arthur

As I've mentioned before, as much fun as software development is, it's still encapsulated in the object of a business, and a good business wants to get feedback from its customers. Here's how to get feedback—all sorts of feedback—from the users of your application.

Once someone other than the original developer sees the application, defects are going to appear. More accurately, that first user is going to provide feedback. This feedback may identify a defect, but it may also be a question or an enhancement request. And it is possible that the user making the report can't necessarily make the distinction.

Ordinarily, a developer fixes the defects, answers the questions, offers to make the changes requested, and that's that. When I first decided that my company needed to track this feedback, I came face to face with four issues. First, when do you start tracking application feedback? Second, how do you categorize it? Third, what mechanism do you use? And fourth, what information do you track about each feedback incident?

The initial assumption: Do you want feedback?

First, let's talk about that initial assumption—that you want to formally track this feedback at all. Isn't it much easier to just deal with each communication from a user and be done with it? There are three distinct benefits to formally tracking each Application Feedback Incident (AFI).

First, you can provide better customer service. AFIs should be numbered and entered into a database (you've got a spare database laying around somewhere, don't you?). As a result, this feedback doesn't fall through the cracks. One of the steps in the tracking process is to provide resolution to the customer.

Second, you can improve your process. One of the attributes to be tracked for each defect is the type of defect and where it came from. From this, you can determine where your weak spots are, and thus, determine where you need to improve your development processes. The AFIs are a tool from which the developer can learn.

Third, you can help improve user acceptance of your system. Everyone has seen the "perfect" system that was rejected by the user community. Well, "user feedback" might not be about a bug. It may simply be a question or a request for a modification in disguise. It can raise a red flag that end users don't understand what the application is supposed to do. You (or your customer) can adapt end-user training and/or documentation to combat this and head off user rejection of the application.

Finally, as I said, user feedback might not be about a defect. It may simply be a question, or, better, it may be a well-disguised request for a modification or enhancement. If you heard the cash register echo in the background, yep, that's right—more business! You can think of user feedback as a very specific, narrowly focused version of market research. You can show a consumer all the pictures of a car you want, but until they get in and drive the car around the block a few times, they won't really know what they think of it. Same thing here—feedback from the user after the application was delivered can't be gotten any other way.

Bugs, defects—you say tomato…

I'm sure that you've already noticed that I'm using the term "defect" instead of "bug." There are a couple of reasons for this. First, it conveys a more accurate impression of what is being discussed. Manufactured parts don't have "bugs"—they have defects. And a defect is an instance of something that doesn't conform to the specification against which it was manufactured.

A "bug"—when you get right down to it—is really just a programming problem. A misspelled variable name, an improper calculation, an out-of-bounds array, or the use of the wrong function. A (programming) bug is the cause of a specific type of defect. Unfortunately, the use of the term "bug" has grown way out of proportion, to the point that many users tend to run out into the street screaming, "It's a bug! It's a bug!" as soon as they encounter something that doesn't behave just the way they wanted it to. It's hard to talk them down into understanding that it's not a "bug"—since the term "bug" has significant emotional connotations.

My business, in particular, is heavily tilted toward manufacturers, and they all understand the concept of a "defect" because they run into them in their own business as well.

Furthermore, being realistic, they also know that they have to send out their own product even when it's not defect-free—one unit may have a blemish on the paint job, another may be missing an eighth screw on the cover assembly, and yet another may have a gap between two doors that's a sixteenth of an inch too wide. But these aren't "show stoppers" because the product can be shipped and successfully used by the customer without problems.

In a similar vein, we need to help the user realize that software has defects as well, and that while some are critical and must be fixed, others aren't as important, and can be ignored, or at least rectified at a later date.

So I always use the term "defect" because it means something is not operating according to how it's specified. And this makes the rest of the job—both fixing as well as, possibly, charging for that work—easier.

When to start tracking feedback

Assuming you have a formal testing process (see the next couple of chapters), you should start tracking feedback as soon as modules go into internal testing. There are three reasons for this: The first is that you want to fix it, right? If you don't document each bug that your QA department finds, it's going to be really easy to lose track of them, and if you don't keep track of them, some will get lost—and they won't get fixed.

The second reason is that, based on what you "fix," you need to determine what testing needs to be redone in order to confirm that the "fix" lives up to its name. You don't want to go through the work to fix something only to find out that there was a deeper-seated problem causing the defect, and that the fix didn't really do the job, or, even worse, to introduce additional defects with the fix. If the fix is to correct a typo on a screen or the order in which items are displayed in a data control, then a simple visual inspection would be enough. But if you changed a calculation in a large invoicing routine, you may have extensive unit/integration/system retesting to do.

The final item has to do with the way you should be trying to improve your development process. "Quality Assurance" has to do with making sure that you're not sending junk out the back door. If you can improve the process you're using up front, you will end up producing fewer defects in the first place.

Thus, you want to track each defect and find out why it occurred. In the case of a programming bug, you want to find out who is putting bugs in the code, and why. Once you find out, you can not only fix the defect, but also take corrective steps to fix the source of the problem. Perhaps you'll provide better training, or develop better coding techniques to make a specific mechanism less fragile, or even learn to avoid certain types of interfaces, processes, or techniques that prove to be more fraught with peril than other types.

Gathering application feedback—the process

When someone encounters a defect in an application, I highly suggest that you require that person to fill out an Application Feedback Incident form and e-mail or fax it in. (I know of some companies that provide a reporting tool over the Web as well.)

Do not—DO NOT—take the report over the phone, or via an informal e-mail or face-to-face discussion. First of all, by making them write down the steps required, they're much more likely to discover that what they perceived as a defect was actually a mistake in their steps. Or, by repeating the steps so they can write them down, they might answer their own question. Or, they might even discover they can't repeat the purported "defect" and thus save you the time and aggravation of trying to reproduce it as well.

The reason for the long name of this form is that frequently the users are not sure what they are reporting. They may think that it's a defect when in actuality it's simply a question about a capability they are unsure of.

Perhaps it's really a request for a new feature or a twist on some existing functionality. They will phrase the request in terms of a defect ("The Framboozle report doesn't break out the prior year and current year values"). But when you review the "defect" report with them, and refer back to the Functional Spec, you can show them that the Framboozle report was never designed to break out the yearly values like they want. Of course, I'm sure that you'd be happy to write up a Change Order and modify the report so that it does so!

Other times, they'll interpret a native behavior in Windows as a defect—such as pressing the Enter key does not advance the cursor from one control to the next, depending on what type of control it is. Or they'll even report behavior that is specified in the Functional Specification as a defect because they're not that familiar with the system and didn't read the specification. For example, we had one customer who, against our advice, wanted the cursor to stay in the field even when it was filled in. However, some of their users reported that behavior as a defect—and when the customer saw the amount of feedback regarding this behavior, they recanted their position.

So, use one form to enter all types of feedback, instead of blanketing the users with a series of forms that they'll either lose or that will just confuse them. The AFI form has spaces for defects, for questions, and for enhancement requests. I'll discuss the contents of the actual form shortly.

Once you receive the application feedback form, your QA department will enter it into the database, and the internal process is started. The developer is notified of the receipt of an AFI form, investigates, produces a fix if necessary, and sends the fix off to QA. Once the fix passes, the fix is given to the customer, and the defect is closed. If the AFI is actually an enhancement request (and the customer wants a Change Order), the developer begins the Change Order process. (See Chapter 27, "Change Orders.")

Each week, QA produces reports for all bugs, questions, and enhancement requests that haven't been closed, and follows up with the developer as to their resolution. Whether the AFI is an actual defect or not, you should report back to the customer as to the proper resolution.

It may be tough to get your customers into the mindset of filling these forms out at first, but often they'll begin to enjoy the challenge of trying to break "your" code. As they become more familiar with the system and spend more time with it, their confidence in the system grows because the rigorous testing and reporting process helps them to realize how sound the system is. And, as an added benefit, the people who test become the "super users" of the system and make your support of the system easier down the road. For some of them, it's a chance to learn new skills, and if they become a system expert, it can enhance their position and standing within their organization.

You'll notice that I don't discuss the testing process itself—and you might be thinking that a well-defined testing process would help alleviate some of the issues raised when getting customers on board with respect to filling out the AFI. The reason I don't is that AFIs can be used at two times—both during formal testing by folks specifically tasked with testing, and during use of the system where issues will undoubtedly crop up. I'll cover testing in more detail in the next two chapters.

Categorizing feedback

I've briefly alluded to the fact that there are three types of feedback: defects, enhancement requests, and questions.

The first kind of feedback actually identifies a defect. Again, if I haven't beat on this enough, a defect is a behavior that doesn't perform as it's described in the specification. As the attachment to the Engagement Letter stated:

A defect is defined as an operation that does not perform as specified in the written specifications and/or change notices, or an error that causes the program to stop and display an error message that says, "An application error has occurred." A defect must be reproducible at the developer's site. Non-inclusion of options, behavior not specifically delineated in the written specifications, and operating system and environmental problems are not considered to be defects. If the problem can be resolved without changing application code, if it is not reproducible upon demand, or if it occurs in a module that has been accepted, and has been working for three months and has not been changed, then it is not considered to be a defect. This does not mean

`that Vendor will not resolve these issues; this means that Vendor will not resolve them without charge.`

`There are a multitude of interface behavior, application performance and general system characteristics for which it would be cost-prohibitive to explicitly delineate. Vendor reserves the right to interpret these and other issues that are not specifically described in a written specification as Vendor sees fit.`

The second type of feedback is an enhancement request. This is where they want something added to, changed in, or removed from the application. The AFI does not handle the entire change—it's simply a flag that indicates more work needs to be done. It initiates a Change Order form, which then has its own process, including a specification, a price, a delivery, acceptance, and so on.

The third type of feedback is simply a question. Sure, they could just call up and ask, but that's usually an unsatisfactory process. First, the person who can answer the question may not be available. Second, the question may require an answer more involved than what a simple phone call can provide—the developer may have to investigate something first, or find someone else for more information.

Third, the act of writing down the question often forces the user to think through what they're asking, and they can sometimes answer the question themselves. And, finally, having it written down and formally submitted means, again, it's not going to get lost. It's all too easy for the developer to talk on the phone for a few minutes and promise, "I'll get back to you on that." But, before they have a chance to do anything about it—including even writing it down on their To-do list—all hell breaks loose and it's suddenly forgotten about. Until, of course, the customer, now angry, calls two weeks later, wondering where the answer is.

The Application Feedback Incident form

Remember that your prime motivation should be to avoid getting defect reports at all, and while some of you may think that the best way is to write flawless code, I personally have found that technique to be too hard. Instead, you can just make the defect reporting process a real nuisance—and this starts with a long, intimidating name for what is otherwise known as "The Bug Report."

The header of the AFI has a place for today's date, the name of the customer, the system they're working on, and the personnel involved.

It also has a place for two tracking numbers. The top number is for your own tracking number—and you'll note that it's not pre-numbered, because customers will often just make a bunch of copies and distribute them to users. Even if you gave them pre-numbered forms, they'd lose them, run out of them and then use the backs of envelopes, and so on. Once a form hits your office, have the "Keeper of AFIs" number them while they're entering them into the database.

The Customer Tracking # is for them to keep track of the AFIs that they submit. They may want to track their own reports before submitting them, or otherwise use it for internal tracking. Your tracking number and theirs often don't coincide—while you'll have all of theirs, they won't always have all of yours, because your internal staff will likely be generating AFIs as well.

Of course, if you deploy this type of application electronically, such as on the Web, or even in a separate application you provide to your customers, you can populate the AFI # yourself each time they complete a new AFI.

The rest of the AFI form is dedicated to capturing specific information from them. The fundamental rule about user feedback is that "Users don't tell you what is going on when their software breaks." It's quite a challenge to get this feedback in a manner that is usable. In creating the layout and instructions for this form, we've adopted the format used by Microsoft in its public beta testing.

Type of AFI

First, have them identify whether this AFI is a defect, a question, or an ER. 'Nuf said? Not really. In my experience, I've found that anywhere between 25% and 50% of AFIs classified as "defects" by the customer are actually misunderstandings about the functionality of the system and/or ERs. So don't freak out the first time you implement this type of system and find, of 45 AFIs in your system, 32 of them are marked as "defects"—it may well not be as bad as that!

Is it reproducible?

Second, if it's a defect, ask if it's reproducible. Many times you'll find a user discovers that the problem they are having is not a bug, but that they forgot step five in a seven-step process. We had one customer who installed a new version of a program three times in a row on three different machines, and each time, encountered the same problem. We had provided step-by-step instructions on a single piece of paper—explicit enough that you could have given them to your 10-year-old to follow. They weren't all that happy about having to use the new version in the first place, and so when the install didn't work flawlessly, they got madder and madder with each new attempt.

We finally walked through these steps with them, on the phone, waiting for minutes each time he fumbled to insert a new floppy or for a file to copy, and lo and behold—when he got to step five, he grunts, "Huh. Never saw this step before." Needless to say, the installation went fine after that.

Of course, it's a rare AFI where the user will circle No in response to this question. They always circle Yes, even if they didn't try to reproduce it. Of course, this doesn't make the defect reproducible. What it does do is give you ammunition when you try to reproduce it and can't. Our position is that we should get paid for spending our time on dead-ends like anomalous behavior that can't be reproduced. When they tell you they can reproduce the behavior, but then can't, it's substantiation for charging for the time spent on that AFI.

What are the steps?

Next, if it is reproducible, describe the steps to reproduce the bug. This is the hardest part for the customer to fill out.

Unfortunately, a common example of "steps to reproduce" looks like so:

```
"Add some data with a number in it to the system."
```

An ideal response to this question is along the lines of:

```
1. Select File, Parts to bring the Parts form forward.
2. Press the Add button.
3. The cursor is in the First Name field.
4. Enter a name with some digits in it, such as "herman444".
5. Press the Tab key twice.
```

I've found that providing a couple of examples of "good" and "bad" sets of steps to your customers or users will help them create better reports for you.

What happens?

After the user has described what they did, ask them what the result of those steps was. It's amazing the number of times that the user will fill out the steps to reproduce and the last line will provide absolutely no useful clue about the results.

There are typically two similarly useless responses here.

The first that you'll often see is "And see what happens?" This is an unsatisfactory answer because the user is assuming that the same thing is going to happen on your system. But what if it doesn't? What if the behavior is environmentally related, or perhaps has to do with specific data they entered? (Why would they have entered a negative invoice number?)

The other reason this answer isn't acceptable is because while the results may be the same on both the user's system and your system, they may be the correct results, and the user doesn't understand this. So, in this case, simply saying, "See what happens?" would evoke a "Yeah, I see what happens. That's good—that's what's supposed to happen," from the developer or tester.

The second response is along the lines of "Then the system died." What do they mean by "died"—did the current form disappear and return them to the main menu, did the system lock up, did they get the blue screen of death, did the power go out or what? More information would be very helpful here as well.

What did you think was supposed to happen?

The fourth part of this form is "What did you expect to happen?" Occasionally, they will indicate that they expected something to happen which reveals that it is not a defect, but a misunderstanding of the way the system actually works. For instance, we received this AFI report once:

```
"Add a record without pressing 'Add' button. Record not added."
```

We were not particularly surprised at the result. The problem was that the user was in "live edit" mode, but thought that simply entering new data would add a new record instead of changing the current record. So while we still charted this as a defect, we marked it down as attributable to training.

Of course, when we received four more AFIs from this same user over the next month about this same issue, we realized that it wasn't a training error after all.

The more specific the user is about what they expected to have happen, the better off you're going to be when it comes time to hunt this one down. Even the most dutiful of users will miss a step now and then, but if they're explicit about the results, you'll still be able to read their mind.

What do you want us to do about it?

No, this isn't the smart-ass question it seems to be.

There are several different avenues of action available to you once you've received an AFI, depending on what type of AFI it is. If it's a defect report, they may want it fixed right away—particularly if it's a serious issue. However, if the application is a large one, and the defect is trivial, or if the application has been delivered to a large population, they may just be notifying you for repair in a future version.

If it's an ER, they may want you to submit a Change Order immediately, but they also might just want to put it in the queue for discussion for the next version's specification.

And, finally, they may just be providing information for you, but don't expect any action.

In any case, it's important for you to understand what the customer expects of you. If you're thinking, "Wow! That's a huge amount of work…" and that you'll get to it in the next release, but they're perceiving the defect as minor, you're going to have a problem resulting in "differing expectations" when they ask, "Why haven't you fixed this yet?"

Determining what to do can be a point of conflict, and how you handle this can have a substantial effect on customer satisfaction and confidence in the system—and the developer. Some give and take is often wise.

Other information

Finally, you'll want to get any other information you can. There's no telling what type of information you can dredge up with this question, and sometimes that's the one piece of the puzzle that will help you figure out what's going on. After all, getting a properly filled out AFI is just the first step in what could be a very complex puzzle.

The AFI database: What information do you track?

Obviously, you'll enter your AFIs into a database.

The complexity of your AFI database can vary greatly. Some people find it sufficient to enter them into a simple spreadsheet, just tracking a few items, such as the name of the customer and the system, the date the AFI was reported, a description of the AFI, and when it was handled. If one person is in charge of maintaining this entire system, you probably don't need much more than this.

However, with bigger systems, in companies with more than a couple of developers, and in order to be able to run the occasional statistical analysis, you'll need more information. Here are some suggestions—you can pick and choose as you desire.

- Who submitted the AFI. This could be an internal person or a customer.
- The type of AFI—defect, ER, or question—and, if defect, whether it was reproducible.
- The specific Module and Task within a Project. You may not be able to pinpoint every AFI to this extent. For example, a toolbar may exhibit undesirable behavior when focus is shifted from one form to another. Is that to be attached to the first form? The second form? Or the toolbar? At times, the user may not be able to identify which Module, depending on the behavior exhibited. This is particularly

true for questions and ERs, since they may encompass a process, not a specific form or report.

- The initials of the person responsible for handling the AFI. In other words, the developer who is going to fix a defect, answer a question, or issue a Change Order for an enhancement request.
- The date the AFI was reported closed (fixed, answered, and so on), and when its closure was reported back to the customer. Also, in the case of a defect, the date and the initials of the QA person involved in signing off on the repair.
- The severity and priority for handling the AFI. In the surgical field, this is known as "triage." When there is more to do than can be done in a given amount of time, someone has to decide who gets taken care of first. Same thing here—which are the high priorities, and which can wait? You will probably have to work with the customer to assign these—since they can tell you what's most important to them, and you can tell them the tradeoffs, given your not unlimited resources.
- Who was responsible for the development of the Module or Task that ended up generating the AFI if it was a defect.
- One piece of data that we don't collect but that my technical editor does is determining where in the process a defect was introduced. Was it a missed requirement, poor design or improper coding, or something else altogether? This is actually done during reviews that are performed at each step in the process, and the purpose is to see how long it takes to find a defect and then to see whether they need to do something different earlier in a project to have better deliverables.
- Resolution. It's often handy to document, particularly for the customer, what the resolution of a specific issue was. Humans aren't perfect—they'll submit the same AFI more than once, they'll change their mind again and again, and they'll even withdraw AFIs that were submitted earlier. Standard responses, such as "Fixed," will be entered in specific date and "Fixed By Who" fields. You can document each of these resolutions most easily in a text field.
- Finally, as described earlier, you should define what testing needs to be re-executed in order to ensure that the fix is indeed a fix.

One tricky situation is when a customer puts multiple items on the same AFI. If they're all going to be handled as a group, you can simply enter one record in the database, but if different pieces will be handled separately, you may consider breaking up their AFI into multiple pieces, and assigning a separate AFI number of yours to each piece.

For example, if they find three typographic errors on the same screen, that can all be lumped under one AFI, but if they include a typo on a control's caption, data not being saved for that control, and a request for the field to be made bigger, those become three separate AFIs. The first is easy and trivial to fix, but at the same time, not critical, while the second is critical, and may or may not be easy to fix, and the third lies somewhere in between—easy to deal with, but may or may not be critical.

Categorizing defects

I think it makes sense for all but the smallest development shops to be very exacting about categorizing defects. Most people think that there are two kinds of bugs: those that the customer finds and those that they don't. You can get much more precise. As an example, here is how we categorize them.

Analysis

This means that you made a mistake in the analysis phase of the project. Examples of an analysis bug would be the situations where the way a customer process worked, or how they recorded data was misunderstood.

I've mentioned earlier about the application where I understood the term "part" to mean a component of the product that the customer was shipping, while they used the term to refer to a component of a machine that was used to make those products. We spent several weeks of iteration before realizing this—if we hadn't realized it at all, and had implemented a system based on our poor understanding of the term, we would have recorded it as a bug in analysis.

True, it's pretty hard to document something like this, but you need to have a mechanism in place. It may not be that important to track if you've only got one or two people in the shop doing the analysis, but once you have 38 analysts at the shop, you'll want to track these bugs just like any other.

Design

Two obvious examples of design bugs would be mistakes in data structure design and screw-ups in designing forms. A data structure design bug would be the incorrect normalization of a set of tables, the use of an overloaded table when it was inappropriate, or the use of data attributes as keys when the instances of that data weren't going to be unique. (Of course, if you didn't realize that the data wasn't going to be unique, that might be an analysis bug.)

An example of a bug in form design would be the creation of a form that required the user to hit four keys to save a record, and then three more keys to start the Add process for a new record—when the form in question is specifically a heads-down data entry tool.

Again, if you heard this in analysis and didn't design the form to be able to do so, then it's a design bug. If you didn't catch on to the customer's requirements, then it would have been an analysis bug. If a customer calls and asks you the same question four times, it could be a design bug. If they have to ask you incessantly, then there was probably a better way to have designed the interface or process that the user is asking about. But it could also be an end-user training bug—that you need to do a better job explaining something to a user, or perhaps find a different way altogether.

Coding

The third type of bug is a coding bug. We're all familiar with these. Syntax errors, mistakes in algorithms, incorrect usage of commands; a coding bug is defined as "We knew what to do (analysis and design were correct) but didn't properly create the code to do the function properly."

Environmental

This category flags a situation when the application runs fine on your system, and runs fine on their system, but when they add another machine to the network the app comes crashing down. Turns out they didn't configure memory properly, or the network card is conflicting with something, or they installed an old set of DLLs, or whatever. In any case, the bug is resolved either by changes to the environment, or by adding features to the code that make checks for the environment problem at hand. The key is that you didn't have to change your code, and, in fact, may not have had to do anything at all.

Installation

Installation issues are the fifth type of defect we track. These obviously only happen at one point during the cycle, but it's important to note them. This bug is flagged whenever an application runs fine at your site but somehow it's not installed to the point of proper operation at the customer's site. This could range from technical issues like bad drivers, lack of disk space, or bad media to human problems like forgetting the media when they visited the customer or overwriting files by mistake.

Training

This next type of bug is a bit of a gray area (perhaps not as gray if you're using computer-based training), but it's worth paying attention to anyway. If a customer has to ask a question about the application that they should have known after you were finished installing the system, then it gets marked down as a training bug. For example, a question like "Why don't I see all of the transactions in the list box?" indicates that we didn't do a good enough job documenting or explaining how the list box is populated. The reason that it's a gray area is that we might have documented it and felt that we did a good enough job, but sometimes users need a bit more clarification than we need.

The important issue here is to make sure that you're taking care of the customer's learning curve—the best application in the world isn't going to do any good if they don't know how to use it.

Data

The next category is my most favorite and least favorite at the same time. I lump these issues under the heading "Data." Suppose the user imports a file and suddenly every list box in the system has garbage in it. Or they turn the system on and are getting intermittently screwy results and a lot of processing errors. Perhaps the records in one table are missing most of their children. In each case, the error is not fixed by changing code, because there are problems associated with the actual data in the tables.

What might have happened? The import file was not in the correct format. Or the user went into the file manually and deleted all the fields with surrogate keys. Or it could just be that a "helpful" administrator from another department restored a backup and overwrote the lookup table with an old lookup table without the most current values.

On the one hand, "it's not our fault"—but on the other, these are issues that we should consider when enhancing the system and making it "idiot proof."

Irreproducible

Finally, the most famous category of them all: irreproducible. "It's just one of those things." This could be traced to a flaky network card, an errant video driver, or just plain magic. I know of one installation where the notebook of a certain executive would flake out and just die every once in a while. It turned out that his office backed onto the freight elevator, and every time he had his modem on and the elevator went by, some sort of interference crashed the system.

Reporting back to the customer

In any but the most trivial project, the list of AFIs you receive from a customer will be sizable enough that you won't be able to keep track of them in your head. And neither will your customer—so they're going to want a report of AFIs that they've submitted. Most likely, they're going to want several reports, or at least one report broken into multiple sections.

These sections are 1) outstanding AFIs (those that haven't been fixed), 2) AFIs that have been taken care of in the most recent build, and 3) all other AFIs. Depending on personalities and needs, the first section may be broken down further—organized by priority. Some AFIs (both defects as well as ERs) might be scheduled for a future version, while others may be serious enough to be scheduled to be handled in the next build.

We've found that we need two types of reports—those that are produced for the customer, and those that are produced for use internally.

The customer reports include our AFI number, the customer's AFI number, the date reported, the priority, the date fixed or dealt with, the description of the AFI, and the resolution of the AFI (if any).

The internal reports also include who's responsible for handling the AFI, the date that the AFI fix was reported back to the customer, and the module the AFI belongs to.

Where next?

Now that you have a mechanism to capture feedback from users, and a process to do so reliably, it's time to examine testing. But before I do so, I'd like to say one last thing about application feedback. Requiring users to fill out these forms and send them in has been a significant benefit to our operations. It enforces discipline on the part of the user, allows us to batch process defects instead of getting interrupted every few minutes with the odd phone call, allows us to charge for more of the odd time we spend with customers, and even reduces time as the users solve some of their own problems. Sure, it can be politically tricky to get users to do so, but if you introduce the requirement as part of the initial specification, it becomes a lot easier to train the users to do so regularly.