THREE

# Implementing Effective Error Coding

In the first chapter, we established that effective error coding is critical to the larger goals of higher productivity and quality. In the second chapter, we identified some of the barriers to implementing effective error coding. This chapter offers a discussion of techniques to achieve an improved standard of error coding.

## 3.1 Raise Your Expectations

The first step in standards improvement is to enhance your expectations. You must no longer accept old standards of error coding as satisfactory. You must demand more effective, more complete error coding from yourself and from your staff. The following are some of the raised expectations you should set for your development process:

- *Handle errors early.* When errors are dealt with early, less work is required to handle them. Early error management also results in a better user experience.
- *Error code as you go.* You must code each feature completely as it is written. Going back later to error code usually never happens. If it does actually happen, it takes more time and yields poorer results.
- *Avoid assumptions.* Assumptions are the bane of good error coding. Later in this chapter, we look at specific assumptions to avoid.

- *Design all routines to be reused.* By viewing all code as reusable, you automatically apply a higher standard of error coding and consistency. Your mind-set must be that every routine may be reused. You cannot apply one standard for reusable code, and another for "one-time only" code.
- *Never code for the same error twice.* If you don't reuse your error coding, it will not be feasible to apply it universally. You will simply have too much error coding to complete the project.
- *Develop a systematic error-coding methodology.* If error coding is not implemented in a systematic, standardized fashion, it will not be easily reused and maintained.

## 3.2 Manage Errors Early

To create the most satisfying user experience, it is critical to manage errors as early as possible. From the user's perspective, the earlier an error is corrected, the better. Let's imagine, for example, a typical data entry form with a number of text entry fields. We will examine several possible points at which to apply error coding in this typical situation.

1. The user enters all values and presses a "Save" button. The application attempts to save the values, and errors are returned from the database. The user is informed of a database error.
2. The user enters all values and presses a "Save" button. At that point, the application validates all values and informs the user that a field entry was incorrect. The user corrects the error, then clicks "Save" again. The application now informs him or her that a second field has an invalid value.
3. The user enters one field and presses tab to move to the next field. The application validates that field and informs the user of an error. The application does not allow the user to leave the field until a valid entry is made.
4. The user presses a key and the application detects that the key press is invalid. It cancels the key press and the user is not allowed to enter the invalid character.

When we examine these error-coding strategies, we see that they are obviously ordered by increasing effectiveness. However, the least effective of these examples is the most likely to actually be implemented in most applications. We will examine the reasons for this shortly.

The earlier the error is handled, the better. Errors handled early are less aggravating for the user because they can be handled more precisely and

effectively. If the error is not handled immediately, then every step the user is allowed to make after the error is input results in more wasted effort for him or her.

Imagine pressing an invalid key but not being informed of it immediately. You continue working, thinking everything is fine. Then suddenly the program tells you that you made a mistake much earlier. All your subsequent effort was a waste of time. How would you feel? Wouldn't you much rather be told immediately that you pressed the wrong key so you don't go on needlessly? Even better, wouldn't you much rather be simply prevented from pressing that wrong key in the first place?

The secret of good error coding is simple: handle errors early. The following Golden Rule encapsulates this practical philosophy:

> ***The Golden Rule of Error Coding***
> *Prevent all errors that can be anticipated and prevented.*
> *Handle all errors that can be anticipated but not prevented.*
> *Trap all errors that cannot be anticipated.*

Errors fall into two general types: those that can be anticipated, and those that cannot be anticipated. The response of an invalid key can normally be anticipated. The application can recognize ahead of time what is valid and what is not. This error should be *prevented* from happening in the first place.

An example of an error that can be anticipated but not prevented is the reading of a file. All sorts of potential errors—such as a file being corrupted—can be anticipated, but these cannot normally be prevented. Therefore, the best we can do is handle them. To handle errors means taking corrective action, with or without user intervention. Prompting the user to repair the corrupted file and try again is an example of *handling* that particular error.

Some errors cannot be anticipated. Since the developer does not know the nature of the error ahead of time, these errors cannot be prevented, nor can they be handled. The best that can be done is to trap the errors. *Trapping* simply means making sure that the error does no damage, then reporting it to the user or the system administrator.

Note that error suppression is not error trapping. Suppressing an error is a form of error handling. We discuss this in greater depth later.

> *The Golden Rule of Error Coding prevents errors as early as possible.*

## 3.3 Code Errors as You Go

The concept of early error management has important implications for the development process as well as for the user experience. The earlier error coding is put into place in the development cycle, the better. Let's examine several common scenarios for error coding during development.

1. The user entry form goes to the testing department. As their team attempts to save the fields, a database error occurs. It goes back to the development team. Rather than fix many input controls and risk introducing errors, it is more expedient to trap the database error and display a message, clearing the form of any potentially erroneous entries.
2. While unit testing a data entry form, the developer realizes that when the “Save” button is pressed, errors can occur because of invalid entries. He or she places code in the Save Click event to check for these errors and to alert the user of the first one encountered.
3. The programmer adds a large number of new text boxes to a form. He or she realizes later that he must protect these text boxes from invalid entries. He or she adds code to the LostFocus event to alert the user if the entry is invalid.
4. The programmer puts a new text box on a data entry form. He or she immediately adds code to the KeyPress event to filter user key presses so that no invalid keys can be entered. (Note that pasting text into the text field would not fire a KeyPress event. This must be handled separately.)

Which of these items is the most effective? Clearly, they are presented in order of increasing effectiveness, so the last approach, utilizing earlier error prevention, is certainly the most preferable. By preventing errors rather than responding to them, both the user experience and the coding efficiency are greatly enhanced.

Notice that these examples parallel the examples shown in the previous section. This is no coincidence. The development examples in this section could have resulted in the different user experiences from the previous section. These scenarios illustrate the general rule that the later at which errors are coded in the development process, the later they will also appear in the user experience. If you think about this a bit, you can see why it is natural and inevitable. The later errors are coded, the less likely that the solution will be a well-implemented and satisfying one.

Postponing error management in development has many other ramifications as well. An error is really more like a weed than a bug (see Figure 3.1). It starts with one seed but quickly winds through the ground. If you catch it early, it is easy to yank it out. But if you wait, its tendrils take hold in every part of the application. It pokes above ground in many places. The task of removing it can be formidable. It can be very difficult to find all the underground tendrils and even more difficult to remove them without causing damage.

One error can send many vines up into the user interface, giving the impression of an extremely buggy application. As another side effect, branching errors are caught late in the development process. The solution to them is likely to be a snipping at the buds, not a removal of the roots. This kind of error-snipping fragments the project and makes it more difficult to maintain. It

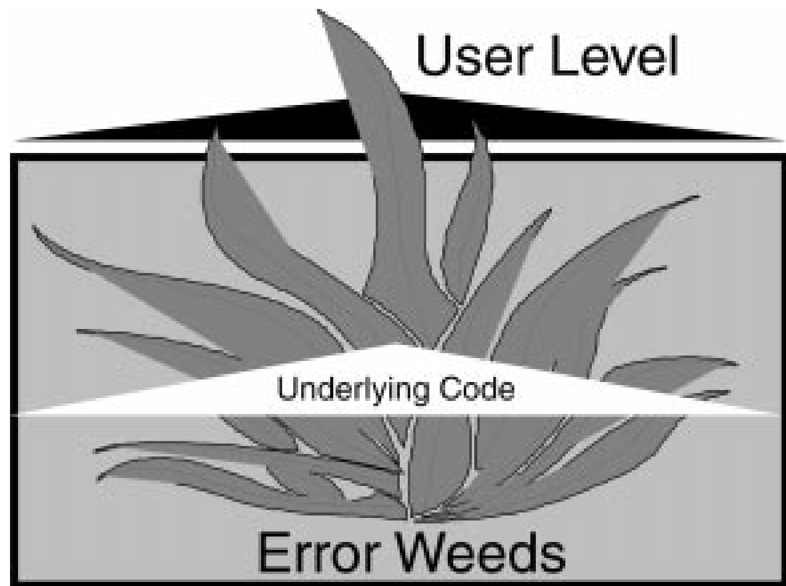


FIGURE 3.1 Bugs are more like weeds

requires much more testing and development time to find and snip each one, and it demoralizes the staff late in the project development cycle.

Another problem exists with thinking about errors late in the development cycle: if you don't code for errors as you code features, this procedure will probably never get done. You can be honest here—it's just us. Have you ever really gone back and added good error coding to your application? It almost never happens. Projects always run out of time, money, or both before error coding ever gets done. When the deadline has passed and the budget is dry, the manager will always decide to ship. When that happens, what should have been error coding gets pushed into the testing and debugging cycles. Worse yet, the errors force users to request support. This is not a profitable situation.

Even in those rare cases when enough time is given to go back for that "error-coding pass," it cannot be well implemented in a second pass. By the time you return to error code those features, you will have forgotten all the subtle error-coding requirements and the effective responses to them. Worse yet, a different programmer who never coded the features is often asked to error code them. The likelihood of introducing errors in a second error-coding pass is sometimes greater than the errors that would have been fixed.

For greatest effectiveness and efficiency, errors must be dealt with early in both the user experience and in the development cycle.

*As each line of feature is coded, it must be 100 percent error coded before moving on.*

Now, if you are wearing your management hat as you read this, you may be thinking that this sounds like ivory tower programming. In a perfect world maybe this could be done, but in reality, we could not meet our deadlines if we spend a lifetime error coding.

That would be true if you continued to program as usual. However, this book presents error-coding techniques in conjunction with standardization guidelines and reusability strategies that make it possible. The Smart Coding Triangle makes it possible to fully error code every feature and still meet deadlines.

A program that is feature complete with partial error coding is far less desirable than an application that is partly feature complete but completely error coded (see Figure 3.2). No one can predict how long it will take to make the first program stable. You have a very good idea how long it will take to complete the second one. Additionally, you can ship the second program at any time, with its completed functionality 100 percent solid. You cannot test or ship the first.

*Your goal should be that your applications are 100 percent solid and shippable after each new feature is completed.*

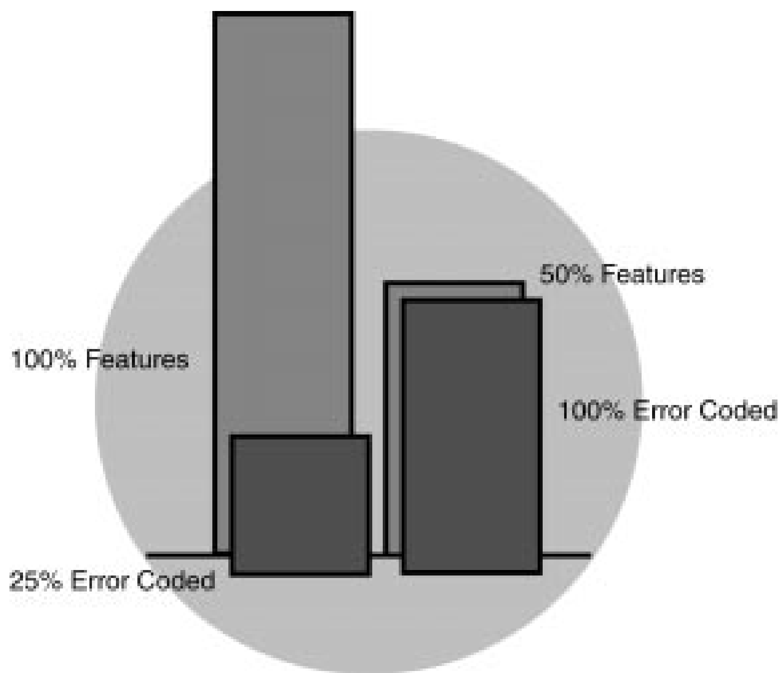


FIGURE 3.2 Complete error coding is more important than having complete features

## 3.4 Anticipate Errors

*Anticipating errors is one of the most important keys to effective error coding.*

Errors that can be anticipated can be coded much earlier in the development process and resolved earlier in the user experience. But how do we anticipate errors? Which errors can be anticipated and which cannot?

In truth, there is no such thing as unanticipated errors, only errors that you have not yet learned to anticipate. Some talented programmers have a "sixth error sense" that allows them to anticipate errors they have never encountered personally. The rest of us generally learn to anticipate errors only from experience. We have to be burned by them at least once to learn to avoid them. At best, we don't have to be burned too many times before we learn to avoid them.

Therefore, I would argue that a programmer's ability to anticipate errors is at least as important as his or her ability to implement new features. Here experience counts. A computer science degree does not guarantee a keen error sense, nor does it provide a great deal of experience. As I said earlier, most training emphasizes features and technology, not error coding. Experienced, battle-worn developers have probably seen it all and have been bitten by bugs often enough to know how to anticipate them. Better yet, they know how to prevent or handle them.

Some companies focus on new technologies in their hiring practices, so why not hire a newly trained programmer for a fraction of the cost of that battle-scarred veteran? Experience with error-coding situations is usually not factored into this evaluation. Rather than discussing the long-term economic impact of hiring practices, I am providing techniques that allow junior programmers to utilize the talent and experience of the veterans without having to learn the same hard lessons.

*Anticipating errors is the result of experience. That experience can be effectively shared and reused.*

## 3.5 Prevent Errors

*The ability to anticipate errors is extremely important, but it is merely a necessary prerequisite to our real goal of preventing them.*

Error prevention is the most important part of error coding. It eliminates much of the need for error handling and trapping. Additionally, it dramatically reduces the amount of code required for a robust application, resulting in the most maintainable applications. Finally, it is the most satisfying form of error coding from the user's perspective.

Why, then, is it almost universally overlooked? It is practically never discussed in technical literature. Instructional courses don't normally emphasize it. Discussions of error coding neglect error prevention completely. They discuss error trapping and some error handling, but never include error prevention in the discussion. It is like writing a book on lung cancer that talks about care and treatment without mentioning smoking habits. It focuses on how to deal with horses after they have left the barn, instead of discussing how to keep them from escaping in the first place.

*Error prevention is the first and strongest line of defense in the war against bugs.*

## 3.6 Handle Errors

Error prevention is the diplomatic arm of error coding, nipping problems in the bud. Error handling is the main fighting force, resolving errors when they occur. It comprises the bulk of error coding. In fact, without effective prevention and reuse, complete error handling could comprise 90 percent of all code in a project. Since it is so time consuming and difficult to produce, error handling forces are typically mobilized quite inadequately.

Error handling should be used to correct errors that can be anticipated but not prevented. Obviously, if you cannot anticipate an error, you cannot handle it. Handling implies that something intelligent is done specifically to correct a particular error situation as soon as it occurs. Simply recovering from an unanticipated error situation is not handling the error. That is error trapping. Error handling, like error prevention, is most effectively implemented as each new feature is being added.

*Error handling is the second and most code-intensive line of defense against bugs.*

## 3.7 Trap Errors

Error Trapping is the most commonly practiced form of error coding. It is the only defense against unanticipated bugs. Without error trapping, unanticipated bugs can result in program crashes, loss of data, and system failures.

Although important, error trapping is both overused and misused. It does not prevent errors, so error trapping effectively locks the barn door after all the horses have run out. This is not a very satisfying response for the user. Also, error trapping does not handle errors. Since it only traps unanticipated errors, it cannot do anything intelligent with each specific error. It merely takes a safe, generic action, such as reinitializing the program. Last, it does not really fix the specific problem. It merely recovers from the problem.

This is not to minimize the importance of error trapping. It is essential for handling unanticipated errors, but when error trapping is used to manage errors that could be prevented or handled, then it is being misused.

Error trapping is over-emphasized for many reasons. To name one of the biggest, it is extremely easy to do. This form of error coding can be added without any knowledge of the specific errors trapped or the correct responses to them. It adds a relatively small amount of code to the application. It also requires only modest programming skill to implement. In fact, a number of third party products that will automatically add error trapping code to your application.

Error trapping does not need to be implemented with each feature, as do error prevention and handling. It can be added at any time, even just prior to shipping. This means that development groups who failed to code error prevention or error handling as they created any features can tack error trapping on as an afterthought. They tell everyone that they did complete error coding. This is not the case, however. Again, the later in the development cycle that error coding is added, the less effective and useful it is. The attractiveness of error trapping is also its weakness.

If you have included error trapping in your application during development, you can use it to diagnose error conditions. If an error is trapped during development or testing, prevent or handle that error so it is never trapped again. An error should never be trapped twice.

As more errors are prevented and handled, the need for error trapping decreases. Again, theoretically no errors exist that cannot be anticipated, only errors that we have not yet learned to anticipate. As we anticipate and prevent or handle more errors, the need for trapping decreases. That is not to recommend completely dispensing with error trapping. Instead, this book will present a Safe Programming Framework that will allow you to code a majority of your application safely without any error trapping. The Safe Programming Framework will encourage you to use error trapping when it is legitimately required in localized circumstances, not as a catch-all approach to error coding.

*Error trapping is the last, impenetrable line of defense against bugs.*

## 3.8 Report Errors

Another essential component of error coding is reporting. Though not often treated as a separate topic, it really does deserve independent consideration. How will errors be presented to the user? How will diagnostic information be communicated to the developers?

As a user, how often have you seen programs display cryptic error details that make sense to no one except perhaps one or two developers? As

a developer, how often have you been asked to respond to bug reports with only the sketchiest of high-level user information?

Clearly, error reporting has two important goals. One is to report errors to the user. The second is to provide diagnostic error information to the developer. The two should not be confused. It is probably best not to accomplish both in a single message.

**User error reporting should:**

- Be presented in language that the user can understand.
- Not confuse the user with unnecessary details.
- Be as specific as possible regarding the particular error that occurred.
- Clearly communicate the reason for the error.
- Clearly communicate how the user can correct the error, or the response the application is taking to correct it.
- Reassure, rather than frighten, the user.
- Tutor and mentor the user.
- Be presented to the user as soon as possible after the error occurs.

**Diagnostic error reporting should:**

- Provide the developer or support staff with all information necessary to reproduce the error.
- Not be shown to the user.
- Be easily accessible by support staff.

What do you think of the following typical error message (Figure 3.3)? What would you do if this error message suddenly popped up on the page as you were reading and did not allow you to continue?

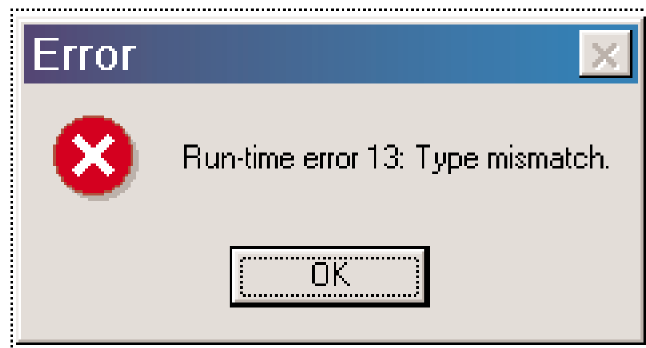


FIGURE 3.3 Typical user error reporting

What does this message mean, especially to a non-programmer? What was the cause of the error? What do you need to do to correct it? Figure 3.4 shows a better example of error reporting.

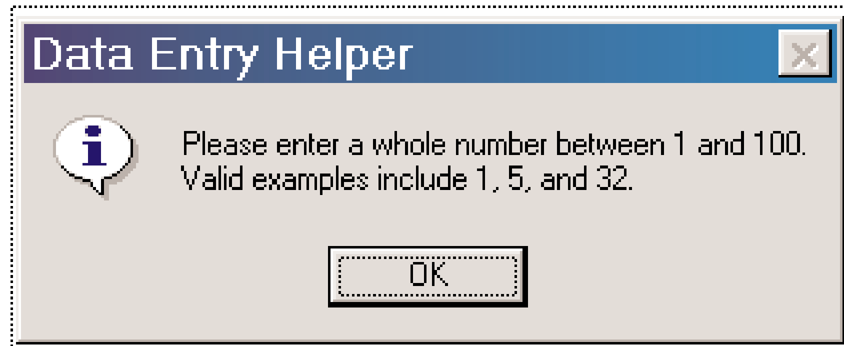


FIGURE 3.4 Helpful user error reporting

The message box displayed in Figure 3.4 is a great improvement over the one in Figure 3.3. It clearly communicates, in nontechnical terms, the cause and nature of the error. It also tells the user clearly what needs to be done to correct the error. As an even more subtle improvement, the second message box does not even appear to be an error. The first message is clearly an error, signaling a problem to the user and making the program look buggy. The second message is presented as more informational. It makes the users feel that the program is intelligently helping them rather than frustrating them. The attitude shown in your error coding can have a powerful effect on the perceived friendliness of your application.

The internal mechanism by which errors are generated, diagnosed, and communicated is a determining factor in achieving the goals of good error reporting. The error reporting mechanism not only determines the content but also the timing of error reporting.

*A good error-coding scheme must integrate reporting into the coding standard.*

## 3.9 Avoid Assumptions

Anticipating errors may be the means of achieving good error coding, but avoiding assumptions is the essential quality and technique vital to anticipating errors. Assumptions are the breeding ground of bugs. Ask any programmer how or why any particular bug was introduced, and he or she will almost always begin the answer with, "I assumed that . . ."

An assumption is a condition or fact that will or may be invalid in the future. Sometimes it manifests as one that the programmer feels is or will be

obvious. Assumptions come in many forms, but most boil down to wishful thinking. One of the most common types of wishful thinking are short-term thinking. The following are some typical assumptions made by programmers and their managers. Look for underlying themes of wishful thinking.

### *3.9.1 I Won't Ever Need to Use This Code Again*

This is the main assumption that undermines reuse. It results in the belief that disposable code does not require the same standard of error coding. One standard is applied to "one-time" coding and another to code designed specifically for reuse. It is usually only an excuse for not taking the time and effort to make it fully error coded. It takes the pressure off the programmer to attain a "reuse" level of quality. This point is discussed in the next section.

### *3.9.2 I'm the Only One Who Will Touch This Code*

This belief is a variant of the previous assumption. Both are based on short-term thinking. This one rationalizes and attempts to excuse inadequate error coding and rationalizes poor coding in general. For example, if no one else will use this code, then why bother with detailed internal documentation? The programmer may say, "I know what I did," which is almost always a bad assumption. In the first place, if code is successful, others will almost certainly need to modify it later. Second, even when the original programmer returns to the code, it will be difficult to recall what was done and why without the proper level of commenting.

### *3.9.3 I Only Designed It for a Particular Situation*

Here is yet another standard programmer disclaimer. By not attempting to make the code robust, the programmer accepts no responsibility for future maintenance. It is short-term thinking to develop any code for a particular purpose only. All code should be potentially reusable in a wide variety of situations.

### *3.9.4 General Coding Assumptions*

The following are some general coding assumptions based on wishful thinking:

- *External variables*. When writing a function, it is dangerous to assume that external variables will be provided as globals. It is much better to make each routine self-contained by passing variables as arguments or by setting class properties. Avoid using globals.
- *Arguments*. When passing arguments, it is a poor assumption to expect that the calling routine will provide a valid argument. Each routine should validate any argument passed to it rather than place the burden of responsibility on the calling routine.

- *I can trust this function.* This is a system function, so can we assume it is robust? Don't trust built-in functions. Even though built-in Visual Basic methods are intended to be optimized for speed, they are not robust with regard to errors. Almost all built-in Visual Basic functions, statements, and methods must be protected with error coding.
- *Implicit behavior*. This is the type of assumption in which we believe that a certain behavior will remain the same. The next chapter looks at assumptions of this type in detail.
- *User behavior*. Assuming that the user will behave in a certain way is usually a recipe for a bug report.

We will refer to these general assumptions throughout the rest of the book with specific examples.

## 3.10 Design Functions for Reuse

This point was made during the discussion of assumptions, but it is important enough to deserve its own section. In order to accomplish effective error coding, each function must be designed to be reused *by someone else.* Without this mind-set, assumptions and compromises are justified that undermine effective coding practices. Consistent standards are not imposed on the code.

Implicit in the argument that code will not be reused is the assumption that you can predict future needs. In truth, none of us can really predict how or when some code might need to be reused, no matter how specific it may seem at the time. It is wishful thinking to assume that no situation will arise in the future in which this code will be needed again.

Further, it is wasteful for your business to write "one-time only" code. The higher the percentage of code that is designed for reuse, the more effective your future programming efforts will be.

## 3.11 Reuse Error Coding

In order to write good error coding, we must take the position that all our code might be reused in the future. There is another side to the relationship between code reuse and error coding. In order for error coding to be economically and consistently applied, the error coding itself must be designed for reuse—and then actually be reused.

When programmers are asked to improve their error coding, their first tendency is to add a great deal more error coding. They tend to cut and paste error handling code liberally and end up tremendously swelling the amount of code in their projects. This sometimes results in a situation opposite to the

desired effect. With all the additional code, the likelihood of errors increases and the amount of code becomes more difficult to maintain.

This does not need to be the case. By coding smarter, you can add complete error protection to every line of feature code without bloating your application. To do so requires highly reusable error-coding routines that can be easily added and maintained, without adding a significant amount of code or code maintenance, and without introducing new errors into the error coding itself.

## 3.12 Systematic Error Coding

In order to reuse your error coding efficiently, it must be written in adherence to clear standards and conventions. This is not usually the case. Error coding, like feature coding, is typically written with a bit of this and a dash of that. When it is implemented, error coding is typically redesigned specifically for each situation. And no two developers on a team follow the same approach to error coding. Individual developers do not normally even follow consistent standards from one situation to another.

Without a coding standard, it is unlikely that error coding will be reused and easily maintained, or that it will mesh seamlessly into a cohesive application. The Safe Programming Framework presented later in this book provides one good systematic approach to integrated application and error coding.

One of the goals of the Safe Programming Framework is to avoid assumptions through good, explicit coding practices. Before describing the Safe Programming Framework, we discuss the importance of explicit coding and provide some specific recommendations to eliminate implicit assumptions from your code.

The next chapter describes implicit coding practices and suggests ways to eliminate them from your Visual Basic applications.