Auerbach Publications
Taylor & Francis Group

The Debugger's Handbook

J. F. DiMarzio

The Debugger's Handbook

Other Auerbach Publications in Software Development, Software Engineering, and Project Management

The Complete Project Management Office Handbook
Gerard M. Hill
0-8493-2173-5

Complex IT Project Management: 16 Steps to Success
Peter Schulte
0-8493-1932-3

Creating Components: Object Oriented, Concurrent, and Distributed Computing in Java
Charles W. Kann
0-8493-1499-2

The Hands-On Project Office: Guaranteeing ROI and On-Time Delivery
Richard M. Kesner
0-8493-1991-9

Interpreting the CMMI®: A Process Improvement Approach
Margaret Kulpa and Kent Johnson
0-8493-1654-5

ISO 9001:2000 for Software and Systems Providers: An Engineering Approach
Robert Bamford and William John Deibler II
0-8493-2063-1

The Laws of Software Process: A New Model for the Production and Management of Software
Phillip G. Armour
0-8493-1489-5

Real Process Improvement Using the CMMI®
Michael West
0-8493-2109-3

Six Sigma Software Development
Christine Tayntor
0-8493-1193-4

Software Architecture Design Patterns in Java
Partha Kuchana
0-8493-2142-5

Software Configuration Management
Jessica Keyes 0-8493-1976-5

Software Engineering for Image Processing
Phillip A. Laplante 0-8493-1376-7

Software Engineering Handbook
Jessica Keyes 0-8493-1479-8

Software Engineering Measurement
John C. Munson 0-8493-1503-4

Software Metrics: A Guide to Planning, Analysis, and Application
C.R. Pandian
0-8493-1661-8

Software Testing: A Craftsman's Approach, Second Edition
Paul C. Jorgensen
0-8493-0809-7

Software Testing and Continuous Quality Improvement, Second Edition
William E. Lewis
0-8493-2524-2

IS Management Handbook, 8th Edition
Carol V. Brown and Heikki Topi, Editors
0-8493-1595-9

Lightweight Enterprise Architectures
Fenix Theuerkorn
0-8493-2114-X

Outsourcing Software Development Offshore: Making It Work
Tandy Gold
0-8493-1943-9

Maximizing ROI on Software Development
Vijay Sikka
0-8493-2312-6

Implementing the IT Balanced Scorecard
Jessica Keyes
0-8493-2621-4

AUERBACH PUBLICATIONS
www.auerbach-publications.com
To Order Call: 1-800-272-7737 • Fax: 1-800-374-3401
E-mail: orders@crcpress.com

The Debugger's Handbook

Jerome DiMarzio

Auerbach Publications
Taylor & Francis Group
Boca Raton New York

Auerbach Publications is an imprint of the
Taylor & Francis Group, an informa business

Auerbach Publications
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number-10: 0-8493-8034-0 (Hardcover)
International Standard Book Number-13: 978-0-8493-8034-1 (Hardcover)

Library of Congress Cataloging-in-Publication Data

DiMarzio, J.F.
The debugger's handbook / Jerome F. DiMarzio.,
p. cm.
Includes bibliographical references and index.
ISBN 0-8493-8034-0 (alk. paper)
1.Debugging in computer science. 2. Computer software--Quality control. I. Title.

QA76.9.D43D56 2006
004.2'4--dc22 2006044272

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the Auerbach Web site at
http://www.auerbach-publications.com

Dedication

This book is dedicated first and foremost to my family — to my loving wife, Suzannah, for her love, dedication, and work, and to our children, without whom it would be meaningless.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Contents

About the Author

J.F. DiMarzio is an IT manager with 14 years of experience in the technology industry. His other books have been translated into five languages and sold worldwide. He currently works as a management consultant in the southeastern United States.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Acknowledgments

I would also like to thank a number of friends, family, and other important people who each made this possible in their own way — Mom, Dad, Matt, Diana, Laura Lewin and the team at Studio B, John Wyzalek, Kimberly Hackett and the team at Taylor & Francis … Go Red Sox!

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Preface

About This Title

The Debugger's Handbook teaches software programmers and testers how to prevent, identify, and remove everyday bugs from applications. It provides a guide to good code-writing habits and common testing and logical debugging techniques. Written from a language-independent perspective, the book provides code samples in VB.NET, C#, C++, and Java. By using this style the book can focus on general programming concepts that are intended to provide programmers with a mental debugging toolbox no matter what language they use. Following the complete process of writing, testing, and debugging an application from beginning to end, the book begins with an exploration of computer bugs and defines exactly what they are. It then teaches programmers different techniques for identifying and avoiding bugs within their code and producing bug-free code. The book concludes with a number or common real-world scenarios. After working through this practical guide and reference, programmers will be able to think in a way that helps them to catch more bugs before any code is compiled. The book also accomplishes the following:

- Teaches programmers how to recognize, identify, and remove bugs from a language-independent perspective
- Covers topics such as coding habits, the design process, design time debugging, and testing
- Provides simple tips and techniques for avoiding common coding mistakes and making code easier to debug
- Includes exercises at the end of each chapter to test your new debugging skills
- Provides code examples in VB, VB.NET, C++, and Java

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Introduction

Welcome to *The Debugger's Handbook*. The goals of this book are to give you a better understanding of what makes a computer bug, teach you how to avoid bugs in your own applications, and show you the tools and skills needed in removing common bugs once you find them. To achieve these goals you will be introduced to a broad range of knowledge, designed to expose you to as many aspects of the debugging process as possible. In doing so you will have the greatest technical tools set at your disposal, and your applications will be better for it.

There are a myriad of debugging methods and methodologies taught in school. The problem with many of these textbook approaches to debugging is that they treat debugging the same way doctors treat illnesses. A human walks into a hospital and based on a list of symptoms the doctor determines what is wrong with the patient and treats him or her accordingly. However, if there is a set of symptoms that is sporadic or hard to define, the doctor's job is infinitely harder and may even be impossible.

Debugging should not be treated this way. Because of the nature of application development, not every bug or type of bug is going to present itself the same way every single time. Therefore, the best way to debug an application is to avoid bugs in the first place. That is where this book sets itself apart. We will not be subscribing to any of the textbook methods for debugging. Rather, we will focus on making you a natural debugger by broadening your knowledge base and forcing you to think about bugs at all stages of application development.

Hopefully by gaining a greater knowledge of applications, systems, and application structures you can learn to identify and avoid situations where bugs can manifest. By avoiding bugs you can create code that will test better, cost less in revisions, and allow you to focus your energies on other tasks. As technically minded individuals, though, we all know that it is nearly impossible to avoid every bug; therefore, you will also

be exposed to a number of bug-finding skills and techniques. Finding and eliminating bugs is not hard as long as you have the right information.

Much has been written on the subject of application debugging; however, we will take a slightly different approach to the subject than most. Unlike almost any other book written on the subject, we will take a multifaceted approach to the topic of application debugging. Many people consider a bug to be something that exists in a finished application; therefore, application debugging is an action performed on a completed piece of software. In this book, although we will cover traditional application debugging from the point of view of completed applications, we will also take a step back. We will spend a good part of the book looking at ways to avoid bugs when writing code. This will introduce you to the concept of bug avoidance as a form of application debugging.

The first subject we will tackle in this book is defining exactly what a bug is. The definition of a bug has certainly changed over the years, and before we can become tried-and-true bug hunters, we need to know exactly what we are hunting. There is no doubt that you should already have a preconceived notion of what a bug is, and although your interpretation of a bug is most likely 100% correct, the terms of that definition may differ when compared to a colleague's definition of a bug. Therefore, Chapter 1, "Bugs: Fact or Fiction?," will ensure that we are all on the same page when it comes to identifying system anomalies that turn otherwise good code into a system-destroying mess. We will operationalize our definition of a bug and explore the history of bugs to learn where they are most likely to manifest.

> Operationalization is that concept whereby a definition contains within itself enough objective critical criteria so that an uninformed observer can determine if a thing is the thing defined. In other words, we want to make sure that when we flag something as a bug, we all know what that means.

Once we have defined bugs and learned how to identify them, we will begin tackling the complex subject of avoiding them. Through different techniques and actions we will create programs that are as bug-free as can be expected. By writing the most bug-free code possible, you will save time and money in future tech support costs and you will save on development of code revisions. Admittedly the best form of debugging is to not have bugs in your code to begin with. Therefore, although it may be considered more of an antibug measure rather than a debug

measure, we will spend considerable time looking at how to keep bugs out of our code.

However, not every bug can be foreseen or avoided. Do not get a false sense of security thinking that this book will help you create completely bulletproof code. In fact, it would be nearly impossible to anticipate every bug and malicious interaction that could possibly arise. This book will give you the best toolbox you could have in an effort to protect yourself against a lot of bugs, but you need to be able to write code that will adapt to situations and not crumble when presented with a problem. Success will be measured by helping you achieve a level of programming where bug anticipation, identification, and removal become an extension of your daily work flow.

The remainder of the book will serve more as a debugger's reference guide. The last few chapters will give you many common error codes, descriptions, and code solutions for use in your everyday programming. That is, multiple error codes from the larger software manufacturers will be listed by number and description, accompanied by possible solutions and code samples for those solutions.

Who This Book Is For

The Debugger's Handbook is geared toward programmers and project managers. That is, if your job involves coding, either directly or indirectly, then you stand to gain from this book. Programmers should gain a greater direct knowledge of debugging, techniques for avoiding bugs, and techniques to get the most out of testing. Similarly, project managers should, by getting a look into the processes needed to thoroughly produce bug-free software, be able to strengthen their skills as managers in that they can more accurately account for the time needed to complete a project.

If you are a programmer, you should have at least basic, or entry-level, knowledge of one of the following programming languages:

- Visual Basic®
- Visual Basic .NET
- C#
- Java™

This book tends to teach by example, and in doing so, these languages are featured prominently (some more than others because of their prevalence in the market). However, the topics covered are introduced in a way that any knowledge of basic programming concepts and practices

will help you tremendously in understanding and achieving the goals set herein.

Although more experienced programmers, and those who are actively involved in coding projects on a daily basis, can more easily put into practice what they learn from this book, any level of programmer will be able to strengthen his or her abilities. However, as previously stated, programmers are not the only people who will learn valuable lessons from reading *The Debugger's Handbook*.

Project managers, too, should have some experience in programming to fully understand the concepts contained within. By following along with the examples and taking the time to understand the outlines given in the first three chapters, a project manager will be better prepared to anticipate the needs of the programmers they are working with. Although the more technical aspects of the book are geared toward programmers, the topics are presented and ordered in a way that project managers can easily see how a coding project should be organized.

What This Book Will Not Do

This book is not intended to teach you a specific programming language or operating system programming technique. Rather, this book will cover general programming concepts meant to help you no matter what language you use. As a general rule, most code samples will be provided in four common programming languages: VB, VB.NET, C#, and Java. This will give you a broader understanding of solving general bug problems in most of the popular programming languages.

Getting the Most from This Book

You will get the most from this book if you read it while in a place or situation where you can try the provided code samples on your own. This book is packed with code samples and examples that can be used to further drive home the lessons of each chapter. Code samples are given in VB6, VB.NET, C#, or Java for some of the examples in the book. This will help you understand the concepts we are covering no matter what language you are most comfortable with.

Whether you are directly involved in the programming process as a programmer, tester, or debugger, or you are indirectly involved in the process as a project manager or lead, you will be able to extrapolate from this book a broad range of knowledge. This knowledge will help you in the day-to-day activities of fighting and preventing bugs in applications.

However, to get the most from this situation you should be actively involved in a project or scenario that will allow you to use the skills you are gaining as you progress through the chapters.

Also, to get the greatest impact from the lessons, it is best to fully understand each example and each chapter before moving on to the next. Each chapter builds on the knowledge gained from the last; therefore, if you do not fully comprehend a given chapter, the book will become harder to follow as you go on. The later chapters will make more sense if you have mastered the earlier material. Take all the time needed to review the given material before moving on — it will prove to be beneficial in the end.

One tool provided to help you understand this material is a set of exercises at the end of each chapter. These exercises include questions on the previous chapter, code samples to debug, and descriptions of programs to test your new skills. The answers to all of the exercises will be at the end of the book. It is suggested that you read each chapter, then attempt the exercises at the conclusion of the chapter; check your progress with the provided answer key.

Another tool provided within this book to help you understand the provided lessons is the numerous code samples. Because these code samples could be presented in one or more programming languages, they will be formatted in a very specific way. The following is a code sample from Chapter 1:

Listing 1.1: VB6/VB.NET

```
Private Function AddMe(number1 AS Integer, number2
AS Integer) AS Integer
'*************************************************
'Function used to add two numbers and return the sum
'jfd
'05/05/2005
'*************************************************
'Variable Definitions
'*************************************************
Dim number1 as Integer
Dim number2 as Integer
'************************
'Add number1 and number2
'************************
```

```
AddMe = number1 + number2
End Function
```

The first thing you should notice about the example is that the language it is written in is always listed at the top of the sample. In cases where multiple examples are presented in multiple languages, each sample will be separated by this header, which denotes what language the sample represents.

Finally, each chapter will contain a section entitled "Looking Ahead." This section will provide an overview of the concepts discussed in the following chapter. Having a brief overview will facilitate thought about the coming material, familiarize you with the concepts being covered, and provide an element of preparation for each new stage of the book.

One axiom of teaching stresses that any acquired knowledge will quickly be lost if it is not put to use. The more a new skill is used in the student's daily life, the longer it will be retained. This is also true in computers and computer programming. If you do not use the skills and techniques taught in this book, you will not retain them very long.

The importance of carefully following the chapters and performing all of the exercises contained within cannot be stressed enough. Although this book will explain to you the core knowledge behind application debugging, that knowledge will not sink in as deep if you do not use it. By taking part in the post-chapter exercises, you have a chance to use and gain a better understanding of that chapter's material.

How the Book Is Organized

The Debugger's Handbook is organized in a very deliberate way. From the order of the chapters to the layout of the material within the chapters, I have taken every precaution to ensure that you get the highest impact from the logical progression of the information.

The first chapter of this book provides for you all of the information needed to understand the remainder of the book. The first chapter can be thought of as the prerequisite for the information covered throughout the final chapters.

After you have been given the prerequisite knowledge, the book will progress. The next series of chapters deal with the pre-compile activities of bug avoidance. That is, topics including project organization, coding, and design time debugging will be discussed. These are activities that will take place before the application is compiled.

Once the program has been compiled, the book will move through a series of chapters that include discussions of applications testing and post-compile debugging. The specific order of the chapters is deliberate in that

they follow the natural project progression. The order in which you would need or use the knowledge in the book is the order in which it is presented.

Finally, the last chapters contain a number of the most common real-world application bugs. These bugs are presented by topic, discussed, and corrected by example to show you how each was located and removed.

The layout within each chapter is also very deliberate. Each chapter will contain two major parts. The first, as with most books, will be the presentation of the information. All of the information needed to understand the topics will be presented and discussed in a clear and easy-to-read manner. This will include multiple examples and code samples to help bolster the lesson.

After the information of the chapter has been presented and discussed, review questions will be presented. These questions are designed to help you think about the material in the chapters and use it in a real-world sense. The combination of the two major chapter parts will give you the greatest opportunity to learn the information provided in *The Debugger's Handbook*.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Chapter 1

Bugs: Fact or Fiction?

Know your enemy.

—Sun Tzu, from *The Art of War*

It is fitting that a quote from *The Art of War* starts off this book. At times it can seem that we, as application programmers, truly are at war with elements of design, bugs, and even deadlines. However, the enemy we fight has no face, has no form, and is born of our own doing. The ongoing struggle of every programmer is in keeping bugs out of our systems. Bugs are an enemy of our own creation that we must be ever vigilant of, yet many of us do not do enough during the programming process to keep them at bay.

Thousands of hours of university courses have been devised to teach young programmers the textbook methods for debugging code. However, bugs are not always textbook. With rapid advancements in technology, code is always changing and so are bugs. Although they do a good job of teaching a programmer the basics, the textbook methods of debugging do not fit every situation. Therefore, the purpose of this book is to teach you the skills needed to debug code in a natural way, akin to how you program.

How often is a programming planning session held where one of the objectives is minimizing bugs? Admittedly, it does not happen nearly as often as it should. Most people do not consciously think about writing bug-free code as much as we think about the overall objective of the project at hand.

One of the reasons why we do not think about writing bug-free code is because a common (possibly mistaken) comparison is made between bugs and bad code. That is, many people, both technical and nontechnical, see bugs as being spawned by poorly written, bad code. This comparison, for many people of both technical and nontechnical backgrounds, can be easy to understand. The comparison being that bugs are in applications, applications are written in code, bugs are bad, good applications have no bugs, and so bad code must produce bugs. Therefore, it is believed that if you write syntactically correct code, you will write bug-free code. As solid as that comparison may seem on the surface, it is inherently incorrect. This common misconception is what will be addressed in this chapter.

The problem in thinking that poorly written code is at the root of bug creation, for average business-level programmers, is flawed in that the compiler guards against this very problem. The compiler, or the portion of the programming tool that takes code and converts it to machine language, will identify and alert the programmer to syntactical and structural errors in blocks of code. The compiler, then, is safeguarding the user against poorly written code.

It is true, however, that the compiler may not catch every problem in every line of code. Rare coding errors that may slip through the compiler could create bugs within the application. However, these instances are very uncommon, especially given the accuracy of many modern compilers, making it more likely that the bugs in today's applications are from unanticipated interactions rather than bad code. Given the role of the compiler, we must now look deeper to find the root of a bug.

Not all bugs — in fact very few — are from bad code. The code that generates bugs is actually good code. The code is syntactically correct, and in other scenarios may run correctly, but for reasons to be discovered throughout this chapter, in certain instances, it causes bugs. Therefore, to say that bugs are born of bad code is a generalization that does not correctly sum up the situation.

In fact, bugs can be generated from otherwise good code. The role of the programmer is to recognize and anticipate what code is going to execute in a way that is harmful to the systems under certain conditions. Most programmers, even those with basic experience, have a general idea of the proper execution of code. The more you look at code in its context, the more you will learn to identify the locations and functions of certain objects within an application. Unfortunately, this knowledge alone will only help you to a point. Programmers need to be able to see the oftentimes subtle indications of a bug and quickly identify the block or code that generated it. This can prove to be tricky at times, and it is when this objective is overlooked that bugs manifest themselves in our applications.

> It can be easily argued that bugs are indeed the product of poorly written code in that optimally written code would anticipate any problem and react accordingly. However, a growing school of thought is that not every programmer or program can, in good faith, foresee every problem that may exist or arise in a system. Some processes are better left to the operating system, such as monitoring application interactions, threads, and memory usage. By this, bugs are not necessarily the product of poorly written code so much as a breakdown in the chain of management between the application and the operating system that could not be planned.

At one point, the operating system of a PC was considered to be a simple host. It would reside on the PC and act as a delivery device for applications. However, as the application market exploded, applications began to clash with each other in attempts to access resources such as volatile memory, video memory, and disk space. Because there was no feasible way for one application developer to alert every other application developer as to what its particular application would do on a system, applications would commonly conflict with each other.

It soon became apparent that one of two things needed to happen: either the operating system developers needed to publish detailed descriptions of how their systems functioned, in an attempt to help application developers better understand the platforms their programs would run on, or the operating systems themselves needed to become more like a referee and less like a toll booth. Therefore, to this point operating systems have become a strong element in ensuring applications work well with each other.

The purpose of this chapter is to help you understand exactly what an application bug is. We all know a bug when we see or experience one, but like most intangibles, it can be very hard to define. The goal of this book is not to teach you the textbook definitions or methods of debugging code. Rather, this book will help you formulate your own methods and best practices that work in your specific situations. Think of defining a bug as trying to define an emotion such as happiness or anger. We can all list examples of things that exude happiness, but how do you define the feeling of happiness? While if multiple people listed the items that make them happy there may be some common items between them, what those items mean to each of those people may be different.

The same is true of an application bug. We have all experienced at least one bug, and we can all give examples of common application bugs,

but do we each have the same definition of what a bug is? That is, if you had to explain what a bug is without using an example, would your definition match that of anyone else's? Chances are it would not. Without having a working definition to use in our daily programming, finding bugs before they externalize themselves can be extremely difficult. It is a common definition that will give us all a step onto equal footing. Therefore, no matter your experience or background, we will all be starting at the same place with a shared common idea of what we are looking for, and from this point we will better be able to identify and remove bugs.

To achieve this unified definition of what a bug is, we will be looking at the history of bugs and following how they have changed and manifested over the years. We will trace the roots of the modern bug to see how it evolved into the system-crashing menace it is today. This definition will give us something tangible to look for when producing our own code. Having a working definition of what constitutes a bug and where they are most likely to exist will help you spot them as you read through your code, and even prevent them in your writing. That is, even years after you have finished this book, you will be able to examine practices, error messages, and even blocks of code and determine if they are prone to bugs by comparing them with our definition.

Having an operationalized definition of a bug is important because you need to know exactly what you are trying to prevent before you can attempt preventing it. Admittedly, attempting to find something without knowing exactly what it is you are looking for would not be productive. Let us look at this as a scenario. For example, if you were asked to go through all of your code and pull out all of the operators, chances are you would know exactly what you were looking for. As programmers, we know that operators are generally characters such as <, >, or =. It is easy for you to separate these characters when asked, because everyone knows the definition of an operator, regardless of the language. Therefore, extracting the operators from the following block of VB6 code would not be a very laborious task.

VB6/VB.NET

```
Private Function AddMe(number1 AS Integer, number2
AS Integer) AS Integer
'*************************************************
'Function used to add two numbers and return the sum
'jfd
'05/05/2005
'*************************************************
```

```
'Variable Definitions
'*************************************************
Dim number1 as Integer
Dim number2 as Integer
'************************
'Add number1 and number2
'************************
AddMe = number1 + number2
End Function
```

Look through the code sample provided and identify the operators. Obviously, the operators appear in the line

```
AddMe = number1 + number2
```

This scenario was easy to complete because we all know what an operator is. Programmers already have a common, unified definition of what an operator is. Therefore, as soon as we see one, we can immediately identify it as an operator.

When we are talking about bugs however, the task is a bit harder. To this point we still have not defined what a bug is; this fact makes finding one quite difficult. There is no character or object identifier to look for when going through code looking for bugs. Anyone can say, "Go through that code and remove all the bugs," but if it were that easy, there would be no bugs in the code to begin with.

Let us look at a new example. For the purposes of the scenario, this example will have to be a bit hypothetical. In this scenario, you are asked to look through a block code and pull out all the bugs. What do you look for? There is no common object or delimiter to identify bugs by, as in this fictitious block of C++ code.

C++

```
//this is my sample program, it contains a bug
//jfd
//05/05/2005
int main()
{
 cout<<"Hello World"<<endl;
return 0;
//insert bug here
//*****************************
```

```
(This is a bug) //if it were this easy, all bugs
would be in parentheses
//*****************************
}
//end sample program
```

Most of the time, bugs do not present themselves in such an obvious manner; they are often more abstract and can span multiple lines of code. For this reason, we must know exactly what we are looking for when we start to debug code, or else we could be on a wild goose chase.

Here is a block of code that actually contains a bug. Can you find it? It definitely does not present itself in the way our fictitious bug did.

VB.NET

```
Public Function LogErrors(ByRef colProcessErrors As
Collection, fileName as Variant) As Boolean
'*********************************************
'Function for logging error
'jfd
'05/05/2005
'*********************************************
     Dim objFso As Scripting.FileSystemObject
     Dim ts As Scripting.TextStream
     Dim strFileName As Integer
     Dim objProcessError As ProcessError
     Dim strLogLine As String
     Dim hshParams As Hashtable
     Dim strKey As String
     Dim strLogDir As String
     Dim objCommonMethod As New CommonLib.Method()
     Dim strIniDir As String
'*********************************************
     strIniDir = objCommonMethod.AppPath
     Dim objTextIniReader As New
CommonLib.TextIniReader(strIniDir & "\Enviroment.ini")
     objFso = New Scripting.FileSystemObject()
     strLogDir = strIniDir & "\ERRORS\"
```

```
    If Not Directory.Exists(strLogDir) Then
        Directory.CreateDirectory(strLogDir)
    End If
    'Only one log file exists for each month
    'Set file name to the beginning of the month
    strFileName = fileName

    If File.Exists(strLogDir & strFileName) Then
        'If file exists, append
        ts = objFso.OpenTextFile(strLogDir &
strFileName, Scripting.IOMode.ForAppending)
    Else
        'If file does not exists, create it
        ts = objFso.OpenTextFile(strLogDir &
strFileName, Scripting.IOMode.ForWriting, True)

ts.WriteLine("Date|Number|Location|Description|
Parameters")
    End If
    'Loop for each error
    For Each objProcessError In colProcessErrors
        strLogLine = ""
        strLogLine += objProcessError.ErrorDate & "|"
        strLogLine += objProcessError.Number & "|"
        strLogLine += Chr(34) &
objProcessError.Location & Chr(34) & "|"
        strLogLine += Chr(34) &
objProcessError.Description & Chr(34) & "|"

        'Add the being " for the params field
        strLogLine += Chr(34)

        'Add all the params to the LogLine string
        For Each strKey In objProcessError.Data.Keys
            strLogLine += strKey & "~" &
objProcessError.Data.Item(strKey) & "^"
        Next
        'Remove the extra ^ in the params field
```

```
            If objProcessError.Data.Count > 0 Then
                strLogLine = Left(strLogLine,
Len(strLogLine) - 1)
            End If
            'Add the ending " for the params field
            strLogLine += Chr(34)
            ts.WriteLine(strLogLine)
        Next
        LogErrors = True
GarbageDump:
        objCommonMethod = Nothing
        ts.Close()
        ts = Nothing
        objFso = Nothing
End Function
```

The bug in this function appears in one of the opening lines:

```
Dim strFileName As Integer
```

Here the variable `strFileName` is being fed by a `Variant` parameter to the function. Therefore, the code will compile correctly, and in case where an integer value is being passed to the function, it will execute properly as well. The problem occurs as soon as an alpha value is passed to the function. The function will return an error when it attempts to assign this alpha value to the variable `strFileName`.

This scenario gives you a good example of what debugging involves. The VB.NET code involved is somewhat long, and the bug only involved one line that would otherwise be fine. In fact, this particular code VB.NET function should compile without issue and should work under certain situations. The only time this function would throw an error is if you attempted to pass an alpha filename to it. If you only pass integer-based filenames, that function will not throw an error.

Did you recognize the bug? Would you have recognized the error as a bug? Let us now take a look at the history of bugs so we can begin to understand and begin to form our definition of a bug.

The History of Bugs

When most of us think of bugs, we think more of glitches in software and rarely hardware. However, bugs are found in hardware-based systems as well, though admittedly not as much now as perhaps 20 to 30 years

ago. Although the focus of this book is in fact software bugs, the genesis of the bug is actually found in hardware. Therefore, it is in the foundations of hardware that we must look first if we are to trace the history of and define bugs.

In the beginning, the focus of all computing systems was hardware. And although hardware is still very important today, we as programmers would not have a job, nor would there be a reason for this book, if applications were not the foundation of the modern computing platform. Whether you took computing classes in school or have just researched the subject of computer history on your own, you no doubt have heard of the early relay calculators, vacuum tubes, and punch cards. These old hardware systems are where we will begin our search for the origin of the computer bug.

Some of the earliest computers ever built were the Harvard Mark series of relay calculators created by Howard Aiken and IBM. These hulking masses of metal and early electronics were mechanical marvels. Basically a giant math processor, the Mark I was over 50 feet long, weighed 5 tons, and cost over $300,000 to produce. According to About.com and the University of Limerick Computer Society, the early Mark I consisted of 78 individual adding machines sequenced together by a rotating shaft, over 700,000 moving parts, and could perform an addition of a 23-digit integer in 1/3 second. Although by today's standards this is unbelievably slow and large, for its time, it was truly a wonder of modern science.

Mark I vs. Today's PCs

For comparison, the current Pentium 4-based hyper-threading processors measure but a few inches long, weigh ounces, and can perform upward of 3 billion additions per second, as opposed to the Mark I's three per second. Proving that the progression of technology over the past 60 years has been truly amazing, computing hardware has grown in speed by a power of nearly 20 and are about 1/3000th the size.

It is amazing to look at how far we have come in the first half century of computing. At its current pace, how far will we go in the last half century? Computing hardware has been evolving at an astronomical pace; there is really no telling how far we will go, but it is sure to be an amazing ride.

In the 1940s a young female Navy officer stood on the verge of making history with the fledgling successor

to the Mark I, the Mark II. Grace Hopper, one of the country's first computer programmers, joined the naval reserves in 1944. In 1945 she was assigned to work with Howard Aiken at Harvard University. With Aiken, she began testing the Mark II relay calculator. According to the Naval Surface Warfare Center Computer Museum in Dahlgren, Virginia, at 3:45 P.M. on September 9, 1945, Grace Hopper made a historic entry in her manual logbook. (Courtesy of the Naval Surface Warfare Center, Dahlgren, VA, 1988.)

Relay Calculators

The relay was first invented to help extend the distance Morse code signals could be transmitted. We have all seen a Morse code tapper; they are basically two pieces of metal that complete an electric circuit when they are touched together. A relay was very similar in that two pieces of metal were separated by a third piece of metal attached to a magnet. This relay was then placed between two distant Morse points. When the magnet intercepted the weakened electric signal, it would pull one side of the relay toward it, thus recreating the signal and sending the now recreated strengthened signal to the distant point.

It did not take very long for people to realize that this type of relay signified something that is still used in computing to this day, the binary system. A relay is inherently in one of two states, open or closed. These two states could then be used to represent the binary digits of 0 and 1. Therefore, connecting enough relays could allow for the computation of large numbers.

Early computers or relay calculators leveraged exactly this kind of hardware. They were series of relays connected in a way that they could be continually switched on and off, representing the addition, subtraction, division, and multiplication of large numbers.

While testing the Mark II, a problem had been recorded with some of the values produced by the giant calculator. Something did not look correct with the output, and Grace Hopper decided to investigate. The input, which consisted of a combination of hard switch settings and punch tape, seemed to be correct; however, the output was still puzzling. In looking through the giant apparatus, it was discovered that a moth had become lodged in the computer and was blocking one of the relays, keeping it in the open position. The offending moth was removed from the machine and taped to the logbook adjacent to the following message:

First actual case of bug being found.

(http://www.history.navy. mil/photos/pers-us/uspers-h/g-hoppr. htm; Photo NH 96566-KN)

With these seven words Grace Hopper landed herself a place in computer history as the first debugger, literally. Thus, on the surface, we have our first example of a computer bug, albeit primitive. However, a close observer would notice something about the implied syntax of her message. She mentions the problem as being the first *actual* case of a bug being found. This implies that the term *bug* existed before Grace Hopper used it to describe the problem between the moth and the relay. We now must look further into the history of computing and electronics to find the basis for using the term *bug* in describing the problem Grace Hopper experienced.

The term *bug* itself had been used to describe problems in electrical equipment long before computers. In 1896, the Hawkins' *New Catechism of Electricity* described a bug as follows:

> To a limited extent to designate any fault or trouble in the connections or working of electric apparatus.

Now we have two pieces of our puzzle: we have found where the term *bug* was coined, and we have found our first written case of it being used to describe a computing problem. Now we can relate the two and somehow develop our definition of a bug, because as simplistic as it would be, we cannot define a bug as a moth the gets lodged within a relay.

So the term *bug* precedes the first computer by almost 50 years and the modern computer by almost 80. Therefore, for the first years of computing a bug was only something that affected hardware; at some point this had to change. Our view of the process had to be altered by

some event to move the focus of what a bug was from something that affected hardware to something that resided with our software.

This is important to understand because although the term *bug* itself is meant to describe any fault in a piece of electrical equipment, admittedly this is not the best definition for our purposes. We need to come up with a more specific definition for applications and application programming. We must now take what we have learned about the origin of the bug and apply it to modern programming. Let us now investigate how the current computing environment has changed both how we look at programming and how we describe bugs.

The Rise of the Modern Programmer

Computers, although driven by software, were seen as a dominant piece of hardware through the 1970s. That is, like their relay calculator predecessors, the mainframes of the 1960s and 1970s were still considered to be primarily ruled by hardware. When you think back to the computing environment of the 1970s, the first thought is large clunky boxes, reel-to-reel tapes, and monochromatic screens. As evidenced by the Mark II, and those that followed it, hardware was king and it drove the computer industry. The software was seen as portable programs that could be run on anything; it was the hardware that was going to make the real difference in the quality and speed of the output.

> The modern computer age began in the late 1970s with the advent of the personal computer, the refinement of the silicon chip, and the greater acceptance of computer science as a legitimate field of study. Before that time, the costs of training, hardware, and support, kept the computing field in the dark ages. It was viewed as a great unknown that only a select few people understood.

Companies like IBM made their fortunes in selling hardware. As with all emerging technologies (even today), new, cutting-edge products are exponentially more expensive when they are first released. In the 1970s, computers were still extremely expensive, so much so that unless you were a big business, owning one was very much cost-prohibitive. Given the expense involved in owning an early computer, training programmers was not an everyday occurrence. Not only could very few people afford to be trained on real equipment, but very few people could understand

the complex languages and methods used at the time. All of these factors combined to create an environment where hardware was at the forefront of the computing market and applications were but an afterthought. The applications were seen as something that could be used anywhere, and your big choice was which hardware platform you thought would give you the best results.

This changed when the computing environment was turned on its head with the release of the first mass-marketed personal computers. While some lesser known personal computing devices came and went over the years, the PC market exploded in 1981 with the releases of both the IBM PC and Apple II. All of a sudden anyone could own a computer, and garage development projects sprang up all over the country. Using languages such as BASIC, everyone could develop their own software.

For the first time, computers were being marketed as educational tools, personal and small business management tools, and to a minor extent entertainment devices. It did not take long for the general public to see the new, smaller, personal computers as more than a business machine and as something that could positively impact their lives.

Though still expensive, more people could now own, and better yet experiment on, computers. This put the tools for developing and implementing all kinds of software into the hands of the average person. The development of homegrown applications flourished as more people began to try coding with the more user-friendly programming languages that were introduced with the personal computers. Through the 1980s and 1990s the world experienced an unprecedented rate of new developments, and before long software was the new king of the computer industry.

The hardware of a computer was still important, but users now looked for specific applications to help them with their everyday lives. It would not be long before the forward progress and trends in computing were governed by the development of software and not the capacity of hardware. By the late 1990s users would buy hardware based on its ability to run specific hardware, which is a complete turnaround from how computing started.

In the early 1980s, almost any preteen Generation X (Atari age) individual could make a decent argument for needing a home computer; more, though admittedly not many, schools accepted printed homework rather than handwritten, educational software promised to teach us the mysteries of the world, and computers were just supposed to make you smarter (all according to the media at the time). Therefore, more computers kept popping up in homes across the country. These early home computers really did not do very much on their own, and software support was fairly limited. However, one thing most new personal computers had was a built-in development environment.

The early PCs could be thought of more as application development kits than modern computers. My first computer, a Tandy TRS-80 Color Computer II, did little more than run a few cartridge-based games, but it did come with its own small version of BASIC to program with. Given the small amount of memory registers, the nongraphic environment, and the storage media (at the time the most common was a standard audio-cassette), the average homegrown application was not very large. However, they represented the earliest forms of an industry that was about to explode. The following is an example of a 14-line program written in 1982 on the TRS-80 CCII.

BASIC

```
10 Print "Welcome to the ninja battle."
20 Print "You are fighting 3 ninjas."
30 Let a = RND(10)
40 Input "Please select a number from 1 to 10" b
50 If a > b then Print "The red ninja won with a ", a
60 Print "You are fighting the black ninja."
70 Let a = RND(15)
80 Input "Please select a number from 1 to 15" b
90 If a > b then Print "The black ninja won with a ", a
100 Print "You are fighting the white ninja."
110 Let a = RND(20)
120 Input "Please select a number from 1 to 20" b
130 If a > b then Print "The red ninja won with a ", a
140 Run
```

This program, however simplistic, was written by a nine-year-old in 1982. A generation before, something on this level would have been unheard of. However, here we are on the verge of an application revolution. Children are controlling the computers, and software is about to become king.

Anyone with a PC and a desire to be creative could now be a programmer. By today's standards very little could be done on these machines programmatically. There were a few registers that could be written to and read from, and if you were lucky you had a cassette drive or tape-style printer to interact with. The really brave programmers attempted plotting pixels and crude geometric shapes on the screen. However, from early games to business applications, mom-and-pop software development houses were rising up all over the country. The average

citizen now had the tools needed to turn the hulking Mark II's of the world into the sleek necessities we are used to now.

Today software is the core of the computing world. There are more software developers now than ever before, and our job could not be more complicated. Many of the world's programmers use one of three popular platforms for their applications: Microsoft® Windows®, UNIX, or Linux. In an effort to keep up with the growing needs and demands of programmers, operating system developers have had to increase efforts in producing more robust systems and platforms for running applications.

Whereas the hardware industry once dictated the direction of the computing market as a whole, the hardware vendors now find themselves keeping up with the demands of the software developers. As soon as new hardware can be developed, software developers will already have applications that will take advantage of its power. This leads to rapid development of hardware and the development of software on fledgling systems that may or may not be fully tested.

The combination of these factors may be the leading cause in the development of bugs, the main cause being the speed at which new advancements in technology are made, and a lesser factor being the "every man" aspect to the programming community. Examining how the two have combined to produce modern bugs will give us the definition we are looking for.

Killing Bugs Is Just a Game

Since the advent of the personal computer, advances in software development have been quick and consistent. That is, as the hobbyist programmers of the 1980s became the computer professionals of the late 1990s they realized how much more these machines were capable of and pushed them to their limits. However, this quest to push the limits of the current computing environment may have been what led to the modern bug. Let us examine how these factors came together to create the bug.

> The following generalization does not apply to any one software developer, but rather the overall climate in the software industry in the late 1980s and early 1990s.

In an effort to be the first out with a revolutionary new product, some software development houses may not thoroughly check every aspect of every program for potential conflicts with other products. The fact is that to check every product against every other product would take too much time and require too many man-hours. The added cost in testing new software

against all existing applications on the market would leave the price of most new products out of the budget of consumers. Yet, this was almost the situation the software industry was in as the early 1990s rolled around.

With most consumer operating systems only treating applications as items to be served up when called and not mitigating communications between them, applications commonly stepped on each other within the operating environment. Developers did not know who else was programming for the environment, or how many other applications might be fighting for the same resources, nor should they.

The smaller software developers were left to fend for themselves when testing their applications for conflict problems between their products and those of other. Many products were released that, when installed in conjunction with other applications, would cause crashes, corruptions in data, and other unanticipated behaviors.

As consumers we came to expect a certain level of "bugginess" in the applications released at the time. It was known, and sometimes expressed directly in the manuals, that software product A could not be used with software product Z. Nowhere was this more evident than in the world of PC gaming.

PC games had been around since the advent of the PC. From simple text-based games to the earliest crude graphics, people have always enjoyed a little entertainment with their business or education. However, when PCs were first released, they were not initially meant solely for gaming. Therefore, the types of games that could be run on a PC were primitive at best, and nothing near in quality of that which could be seen in the now thriving arcades. At this time, PC game developers (what few existed) were not considered legitimate PC developers in the application inner circles. This viewpoint would quickly change.

In the early 1990s PC gaming was just coming out of its infancy. By this time people had been playing games on PCs for a few years, and game makers were now legitimate PC application developers. However, the platform that PC game makers had to work with was not stable. The operating system at the time was MS-DOS (Microsoft's disk operating system — also referred to simply as DOS), and a major software package was on the market that wanted to use all of DOS's available resources, Microsoft Windows. At this time Windows was not yet an operating system unto itself; it still ran on top of DOS. This situation created havoc in the world of game design. Games had to be designed within a DOS environment, but must work with Windows. Many times the solution was to shut down Windows and run the game in DOS.

Games, which even today test the limits of desktop computing, historically require nearly every resource a PC can offer up. It was not uncommon to have bugs in games that would cause memory leaks,

graphics problems, and sound issues when used with Windows. Because debugging Windows was almost unheard of at the time, the fix offered by many developers was to create a customized boot disk that would load the user's PC clean of Windows, thus eliminating most of the bugs.

> Although Windows itself could just be shut down, leaving you in the DOS operating environment, this would not solve the problems or bug. Windows would often use resources and not release them, even upon its closure. Therefore, a disk that would allow you to boot clean of Windows was often the only choice.
>
> Telling consumers that to use an application bug-free they need to create a boot disk with custom memory settings, and leave behind Windows, would spell doom for a developer in today's market. Modern consumers do not accept products unless they work perfectly out of the box. Software programmers began to develop a better sense of what to look for when creating applications. Several characteristics of buggy software were assembled and used as a guide of what to avoid when developing.

The most common bugs found at the time were:

- Memory bugs
- Graphic errors
- System crashes
- Resource locking

From this list we can now define what a bug is, then look at how one company has worked to eliminate them (or at least make the task of finding them easier).

Reading through the last few sections, one theme should be clear: many bugs are created by adverse reactions or conflicts with other software products. That is to say that when two or more products fight over resources, the resulting conflict can be called a bug. A resource can be memory, a particular function, a printer, or anything that can be accessed by more than one object.

In a broad sense this will be our working definition of a bug. Analyzing this definition, it is visible that one common element blamed for creating bugs may seem to be absent — bad code (going back to our discussion for earlier in this book about the fact that bad code does not necessarily equal bad code, and vice versa).

Most common compilers do a better job than ever before of finding and labeling bad code. Bad code, as in code with major syntactical errors, accounts for very few of the bugs in today's software. However, even the best programmers in the world still miss a variable call or a parameter in a function. Therefore, our definition of bug is really twofold.

We can start our definition of a bug by describing what we now know about bugs. Looking back over the last section, we can start our definition as follows: A bug is an unanticipated error or reaction created by application-level conflicts with other objects in the process of gaining, releasing, or locking resources. Bugs are also evident in code that has not been fully realized, traced, or is syntactically incorrect.

This definition states that a bug is any part of an application that causes a conflict when attempting to use any of the computer's resources. We go on to add that bugs can also be found in code that has not been fully realized, traced, or is syntactically incorrect. Together these represent parts 1 and 2 of our bug definition.

Having the definition is really only half the battle; we must now interpret what this definition means and how it affects us as programmers in our daily lives. We need to answer the following questions:

- What does this definition mean?
- How does it affect our attitude toward debugging?

Now that we have our definition, let us discuss what it means to today's programmers.

Dissecting a Bug: Definition

Let us take a look at the first half of our definition of a bug:

> An unanticipated error or reaction created by conflicts with other objects in the process of gaining, releasing, or locking resources.

If you have been programming for any amount of time, especially with an object-oriented language such as C++, the first thought that might have come to your mind here may have been memory conflicts. One of the most common bug conditions is in fact some form of memory conflict. Memory conflicts or bugs can be caused by either of the following:

- Not allocating enough memory for a given object
- Not releasing memory when it is no longer needed

Let us discuss the first point: not allocating enough memory for a given object.

> In some languages such as VB, memory allocation and de-allocation problems are mostly handled by the runtime environment. VB programmers have very little control over how or when memory is allocated for specific objects. However, this has changed in VB.NET. The user now has more say over processes such as garbage collection and memory de-allocation.

When a programmer does not allocate enough memory for a given variable or object, it can easily cause an application crash. A simple example of this would be if a programmer creates a 16-bit integer variable and then, during runtime, attempts to place a 32-bit value into the variable. This situation would cause the application to immediately halt or throw an error to the operating system. Such situations are more localized examples of memory bugs because they are contained to, and generally affect only, the application to which they are confined.

This is true for most object-oriented languages, such as Java and C++. Memory allocation bugs can also be found in smaller support languages, such as T-SQL. That is, especially in some languages such as T-SQL, the amount of memory to be used by each parameter, column, and various other objects must be specified with the object's definition. Subsequently, any further calls to that object must be prepared to handle the correct amount of data.

These kinds of local memory bugs can be fairly easy to find. While the compiler may or may not pick up on such a situation as that previously described, a programmer would definitely see it upon running the applications, either for the first time or in debug mode. However, given the ease of detecting such a local memory bug, they are also fairly common.

> Debug mode and other debugging environments and techniques will be discussed later in this book.
>
> Conversely, the second type of memory bug — not releasing memory when it is no longer needed — can be harder to find and more destructive. Knowing when to release resources is a bit trickier than creating the object in the first place. Every programmer can easily find the point at which memory should be allocated to an object: the first time the object is defined, called, or otherwise used. However, tracking the last time an object is needed, so that its memory can be released, can be a bit harder, especially if the object is shared across multiple blocks of code.

As in the previous example, a programmer should allocate memory for the objects needed throughout the application. This memory must also be de-allocated when it is no longer needed. By de-allocating the once used portions of memory, you are telling the operating system it is okay to use that memory for other functions. When this memory does not get de-allocated, the operating system never realizes those segments can be reused. Thus, the memory is dead and cannot be accessed. Such dead memory segments can be very harmful to a system if not monitored.

This kind of memory bug, also known as a memory leak, can manifest itself in a few different ways. First, the overall performance of the PC will begin to slow. This is due to the fact that the PC now has less memory to run on. The operating system may now have to juggle segments of data to work around the dead memory that was not de-allocated by the application. Over time, and possibly after several uses of the offending application, the memory leak may build to a point that the PC has no operating memory and crashes.

Think of the results this way. Run an application that uses a lot of memory at one time without releasing it, such as a small game. Now attempt to run another application; you should notice that there is a little bit of lag, but nothing that is unbearable. Close the application but leave the game open. If you can, open a second instance of that game. Now reopen the application you opened before. Does it open slower? Does it open at all? Chances are that the system is running pretty slow at this point. This is the same effect that memory leaks have on a system.

However, system freezing and crashes are not the only hazard of memory leaks. Another symptom of memory leaks, and the one that makes them fairly difficult to find and diagnose, is that in a weakened state these bugs may affect applications other than those creating them. That is, one program may cause the memory leak, and although this leak may not be large enough to crash the system, another program could attempt accessing the dead memory and fail. This failure could cause the second application to crash or become corrupt. The slowness caused by the buildup of dead memory can also make it difficult, if not impossible, to open diagnostic tools, making troubleshooting such problems a hard project.

Over the years, the major operating system manufacturers have made great strides in creating programmer-friendly environments. Windows in particular has changed greatly over the past 15 years. Where once programmers shuddered at the thought of producing code on Windows, opting to shut Windows down and run application in DOS, they now flock to Windows, making it one of the most dominant programming platforms.

After the release of Windows 95, when Windows moved from a DOS application to a full-fledged operating system, Microsoft began to realize

that it needed to create an environment that would not only be easier to program on, but also allow multiple developers' products to work together without fear of adverse interactions.

Consequently, with the release of Visual Studio .NET, Microsoft has made even more strides and taken even greater precautions in avoiding exactly these kind of memory errors. Features such as dynamic memory allocation and garbage removal will help programmers avoid some memory bugs; however, you must still be aware of their existence and the steps needed to find and eliminate them.

So, we have concluded that the first part of our definition of a bug (an unanticipated error or reaction created by conflicts with other objects in the process of gaining, releasing, or locking resources) not only is the most common type of bug you will experience, but also can be the most destructive. Throughout this book we will focus on comparing this definition against our code to help find and eliminate bugs. Let us now look at the second half of our bug definition:

> Bugs are also evident in code that has not been fully realized, followed through, or is syntactically incorrect.

This, the second part of our definition, is by far the more complex of the two halves of the bug definition, and it covers much more ground. Where the first part of our definition focused on one specific type of bug, a memory bug, the second half of the definition covers the more general bugs you are likely to come across.

Whereas memory bugs are quite specific in their symptoms, yet can be hard to track down, these more general code bugs can have symptoms that are best described as quirky in nature but are relatively easy to fix. The majority of the bugs that fall under the second half of our definition are also considered bad code or sloppy work by some, but for our purposes they will be called code bugs, as opposed to memory bugs.

> Through the remainder of this book bugs will be referred to in one of two ways, memory bugs or code bugs. Although memory bugs are ultimately caused by an error in coding, for the purposes of separating the bugs' techniques and symptoms, they will be referred to thusly.

According to the definition of a bug, code bugs can be separated into three different categories:

1. Code that has not been fully realized
2. Code that has not been followed through
3. Code that is syntactically incorrect

Let us briefly look at how each of these can materialize in your code.

Fully Realized Code

Most projects begin on a white board, or in some other type of planning session. Here, all of the different components are laid out and connected in a way that makes the target application seem more achievable. Different ideas are put down, everyone's input is gathered, and eventually a working model is created that can be used as a reference during the programming stage of the project.

However, many things can change when it comes to the actual coding of the application. Some features may turn out to be a little ambitious for the level of coding being done, the timeline may not be realistic, or the fact that everyone has his or her own way of interpreting ideas and writing code may turn the earlier plans on their side. These situations are a sample of what may lead to code not being fully realized.

Fully realized code is code that is complete in every aspect. That is, everything that was set to be done has been done. For example, a basic representation of this concept would be the omnipresent "Hello World" application. Our goal is to create a VB6 application that will display a dialog box with the phrase "Hello World!"

VB6

```
Private Sub Form_Load()
**********************
'Hello World
'5/1/2005
'-jfd
'**********************
msgbox "Hello World!" 'Displays –Hello World dialog box
'**********************
End Sub
```

However simple it is, this small VB6 application is fully realized. Everything that we set out to do has been done. Although this may seem to state to obvious — finish what you set out to do when you are dealing

with multiple programmers and thousands of lines of code — things can get overlooked. Whether it is an error handler in an obscure function or a modal Windows form in place of a nonmodal one, code that is not fully realized can cause bugs.

The problem with bugs that are caused from code that is not fully realized is that they may only materialize in very specific conditions. That is, the use of the region of code that has not been fully realized will generate the bug. The following VB6 function accepts two numeric input values, adds them, and returns the sum; however, the code is not fully realized and contains a bug.

VB6

```
Public Function AddMe(iVal1 as Variant, iVal2 as
Variant) as Integer
'****************************************
'AddME
'function to add values and return sum
'5/1/2005
'-jfd
'****************************************
AddMe = iVal1 + iVal2
'****************************************
End Function
```

> Even if you are not proficient in Visual Basic 6 code, just follow along. As we move through the book, all examples will be in VB6, VB.NET, C++, or Java. However, where these are very basic samples to further the discussion, they are just in VB6.

If you execute this code with parameters of 2 and 3, the function will return 5, and supplying values of 130 and 6 will yield 136. This function, as written, definitely does do what we wanted it to do; we feed it two numbers and it will add them and supply us a sum. Look at the code again and see if you can spot where this code has not been fully realized; where is the bug?

What if the user supplies an A and a 4, what will the function yield? An error. The parameters `iVal1` and `iVal2` are dimensioned as variants. However, the function is fully expecting to add two integers. This is a

bug that will materialize in a fairly specific situation, but it is a bug nonetheless. To be fully realized, this VB6 function needs some form of error trapping or integer validating. There are a few ways to fully realize this code. One would be to use the internal VB6 function `IsNumeric` to test the parameters before adding them.

VB6

```
Public Function AddMe(iVal1 as Variant, iVal2 as
Variant) as Integer
'*****************************************
'AddME v.2 - using IsNumeric
'function to add values and return sum
'5/1/2005
'-jfd
'*****************************************
'*****************************************
'if the supplied parameters are numeric, add them
'*****************************************
If IsNumeric(iVal1) and IsNumeric(iVal2) Then
AddMe = iVal1 + iVal2
End If
'*****************************************
End Function
```

Another way to fully realize this code would be to throw in an error handler. Just to be safe, we will use both the numeric test and the error handler; this will give us a good, fully realized VB6 function.

VB6

```
Public Function AddMe(iVal1 as Variant, iVal2 as
Variant) as Integer
'*****************************************
'AddMe v.3 using an error handler
'function to add values and return sum
'5/1/2005
'-jfd
'*****************************************
```