

THE ADDISON-WESLEY MICROSOFT TECHNOLOGY SERIES



PROFESSIONAL EXCEL DEVELOPMENT

SECOND EDITION

THE DEFINITIVE GUIDE TO
DEVELOPING APPLICATIONS USING
MICROSOFT® EXCEL, VBA®, AND .NET



ROB BOVEY
DENNIS WALLENTIN
STEPHEN BULLEN
JOHN GREEN

Praise for *Professional Excel Development, Second Edition*

“As Excel applications become more complex and the Windows development platform more powerful, Excel developers need books like this to help them evolve their solutions to the next level of sophistication. *Professional Excel Development* is a book for developers who want to build powerful, state-of-the-art Excel applications using the latest Microsoft technologies.”

—Gabhan Berry, Program Manager, Excel Programmability, Microsoft

“The first edition of *Professional Excel Development* is my most-consulted and most-recommended book on Office development. The second edition expands both the depth and range. It shines because it takes every issue one step further than you expect. The book relies on the authors’ current, real-world experience to cover not only how a feature works, but also the practical implications of using it in professional work.”

—Shauna Kelly, Director, Thendara Green

“This book illustrates techniques that will result in well-designed, robust, and maintainable Excel-based applications. The authors’ advice comes from decades of solid experience of designing and building applications. The practicality of the methods is well illustrated by the example timesheet application that is developed step-by-step through the book. Every serious Excel developer should read this and learn from it. I did.”

—Bill Manville, Application Developer, Bill Manville Associates

“This book explains difficult concepts in detail. The authors provide more than one method for complex development topics, along with the advantages and disadvantages of using the various methods described. They have my applause for the incorporation of development best practices.”

—Beth Melton, Independent Contractor and Microsoft Office MVP

“*Professional Excel Development* is THE book for the serious Excel developer. It reaches far beyond object models and worksheet layouts and code syntax, to the inner workings of a professional developer’s mind. The book covers Excel in great depth, but more important it explores the thought processes and logistics behind successful Excel development.”

—Jon Peltier, Microsoft Excel MVP and President of Peltier Technical Services, Inc.

“The authors have done what I deemed impossible: improve a book that I already considered the best book ever on Excel development!”

—Jan Karel Pieterse, Excel MVP and owner of www.jkp-ads.com

This page intentionally left blank

The background of the cover features a series of gray squares of varying sizes arranged in a stepped, geometric pattern. Some squares are solid gray, while others are white, creating a modern, architectural look. The squares are positioned in the top-left, bottom-left, and bottom-right areas of the cover.

PROFESSIONAL EXCEL DEVELOPMENT SECOND EDITION

This page intentionally left blank



PROFESSIONAL EXCEL DEVELOPMENT SECOND EDITION

THE DEFINITIVE GUIDE TO DEVELOPING
APPLICATIONS USING MICROSOFT®
EXCEL, VBA®, AND .NET

Rob Bovey
Dennis Wallentin
Stephen Bullen
John Green



◆ Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data:

Professional Excel development : the definitive guide to developing applications using Microsoft Excel, VBA, and .NET / Rob Bovey ... [et al.]. — 2nd ed.

p. cm.

Rev. ed. of: Professional Excel development : the definitive guide to developing applications using Microsoft Excel and VBA / Stephen Bullen, Rob Bovey, John Green. 2005.

ISBN 978-0-321-50879-9 (pbk. : alk. paper) 1. Microsoft Excel (Computer file) 2. Microsoft Visual Basic for applications. I. Bovey, Rob. II. Bullen, Stephen. Professional Excel development.

HF5548.4.M523B85 2009

005.54—dc22

2009005855

Copyright © 2009 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671-3447

ISBN-13: 978-0-321-50879-9

ISBN-10: 0-321-50879-3

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.

First printing May 2009

TABLE OF CONTENTS

Acknowledgments	.xiv
About the Authors	.xv

Chapter 1 Introduction

About This Book	.1
Who Should Read This Book	.2
Excel Developer Categories	.2
Excel as an Application Development Platform	.4
Structure	.7
Examples	.8
Supported Versions of Excel	.9
Typefaces	.10
On the CD	.10
Help and Support	.11
The Professional Excel Development Web Site	.12
Feedback	.12

Chapter 2 Application Architectures

Concepts	.13
----------	-----

Chapter 3 Excel and VBA Development Best Practices

Naming Conventions	.27
Best Practices for Application Structure and Organization	.40
General Application Development Best Practices	.45

Chapter 4 Worksheet Design

Principles of Good Worksheet UI Design	.69
Program Rows and Columns: The Fundamental UI Design Technique	.70
Defined Names	.71
Styles	.78

User Interface Drawing Techniques	83
Data Validation	88
Conditional Formatting	92
Using Controls on Worksheets	98
Practical Example	100

Chapter 5 Function, General, and Application-Specific Add-ins

The Four Stages of an Application	107
Function Library Add-ins	110
General Add-ins	117
Application-Specific Add-ins	118
Practical Example	125

Chapter 6 Dictator Applications

Structure of a Dictator Application	141
Practical Example	157

Chapter 7 Using Class Modules to Create Objects

Creating Objects	166
Creating a Collection	170
Trapping Events	177
Raising Events	180
Practical Example	188

Chapter 8 Advanced Command Bar Handling

Command Bar Design	198
Table-Driven Command Bars	199
Putting It All Together	219
Loading Custom Icons from Files	228
Hooking Command Bar Control Events	232
Practical Example	241

Chapter 9 Introduction to XML

XML	249
---------------	-----

Chapter 10 The Office 2007 Ribbon User Interface

The RibbonX Paradigm	273
An Introduction to the Office 2007 Open XML File Format	274

Ribbon Design and Coding Best Practices	278
Table-Driven Ribbon UI Customization	289
Advanced Problem Solving	291
Further Reading	300
Related Portals	300

Chapter 11 Creating Cross-Version Applications

Command Bar and Ribbon User Interfaces in a Single Application	304
Other Excel 2007 Development Issues	319
Windows Vista Security and Folder Structure	326

Chapter 12 Understanding and Using Windows API Calls

Overview	331
Working with the Screen	337
Working with Windows	340
Working with the Keyboard	349
Working with the File System and Network	355
Practical Examples	369

Chapter 13 UserForm Design and Best Practices

Principles	375
Control Fundamentals	384
Visual Effects	392
UserForm Positioning and Sizing	400
Wizards	407
Dynamic UserForms	411
Modeless UserForms	419
Control Specifics	425
Practical Example	432

Chapter 14 Interfaces

What Is an Interface?	433
Code Reuse	435
Defining a Custom Interface	437
Implementing a Custom Interface	438
Using a Custom Interface	440
Polymorphic Classes	443
Improving Robustness	448

Simplifying Development	448
A Plug-in Architecture	460
Practical Example	462

Chapter 15 VBA Error Handling

Error Handling Concepts	465
The Single Exit Point Principle	475
Simple Error Handling	475
Complex Project Error Handler Organization	476
The Central Error Handler	481
Error Handling in Classes and UserForms	488
Putting It All Together	490
Practical Example	496

Chapter 16 VBA Debugging

Basic VBA Debugging Techniques	507
The Immediate Window (Ctrl+G)	517
The Call Stack (Ctrl+L)	521
The Watch Window	522
The Locals Window	532
The Object Browser (F2)	533
Creating and Running a Test Harness	537
Using Assertions	540
Debugging Shortcut Keys That Every Developer Should Know	542

Chapter 17 Optimizing VBA Performance

Measuring Performance	545
The PerfMon Utility	546
Creative Thinking	551
Macro-Optimization	556
Micro-Optimization	567

Chapter 18 Introduction to Database Development

An Introduction to Databases	577
An Introduction to SQL	594
Data Access with ADO	598
Further Reading	613

Chapter 19 Programming with Access and SQL Server

A Note on the Northwind Sample Database615
Designing the Data Access Tier616
Working with Microsoft Access Databases620
Working with Microsoft SQL Server Databases630
Upsizing from Access to SQL Server642
Further Reading647
Practical Example648

Chapter 20 Data Manipulation Techniques

Excel's Data Structures661
Data Processing Features667
Advanced Functions678

Chapter 21 Advanced Charting Techniques

Fundamental Techniques687
VBA Techniques702

Chapter 22 Controlling Other Office Applications

Fundamentals709
The Primary Office Application Object Models725
Further Reading739
Practical Example740

Chapter 23 Excel and Visual Basic 6

A Hello World ActiveX DLL742
Why Use VB6 ActiveX DLLs in Excel VBA Projects758
In-Process Versus Out-of-Process774
Automating Excel from a VB6 EXE775
COM Add-ins783
A "Hello World" COM Add-in783
The Add-in Designer788
Installation Considerations790
The AddinInstance Events792
Command Bar Handling795
Why Use a COM Add-in?798
Automation Add-ins799
Practical Examples802

Chapter 24 Excel and VB.NET

.NET Framework Fundamentals	818
Visual Basic.NET	819
Debugging	845
Useful Development Tools	853
Automating Excel	855
Resources in .NET Solutions	863
Retrieving Data with ADO.NET	864
Further Reading	870
Additional Development Tools	871
Q&A Forums	871
Practical Example—PETRAS Report Tool .NET	872

Chapter 25 Writing Managed COM Add-ins with VB.NET

Choosing a Development Toolset	890
Creating a Managed COM Add-in	891
Building the User Interface	908
Creating Managed Automation Add-ins	928
Manually Register and Unregister COM Add-ins	940
Using Classes in VB.NET	940
Using Classic ADO to Export Data to Excel	948
Shimming COM Add-ins	952
Related Blogs	962
Additional Development Tools	962
Practical Example—PETRAS Report Tool.NET	963

Chapter 26 Developing Excel Solutions with Visual Studio Tools for Office System (VSTO)

What Is VSTO?	976
When Should You Use VSTO?	983
Working with VSTO Add-Ins	985
Working with VSTO Templates and Workbook Solutions	1006
Deployment and Security	1016
Further Reading	1026
Related Portal and Blogs	1026
Additional Development Tools	1026

Chapter 27 XLLs and the C API

Why Create an XLL-Based Worksheet Function	1029
Creating an XLL Project in Visual Studio	1030
The Structure of an XLL	1034
The XLOPER and OPER Data Types	1044
The Excel4 Function	1050
Commonly Used C API Functions	1052
XLOPERs and Memory Management	1053
Registering and Unregistering Custom Worksheet Functions	1054
Sample Application Function	1057
Debugging the Worksheet Functions	1060
Miscellaneous Topics	1061
Additional Resources	1062

Chapter 28 Excel and Web Services

Web Services	1065
Practical Example	1072

Chapter 29 Providing Help, Securing, Packaging, and Distributing

Providing Help	1085
Securing	1094
Packaging	1099
Distributing	1104

Index	1107
------------------------	------

ACKNOWLEDGMENTS

First and foremost, this book would never have been written without the support of our partners and families, who have graciously put up with our insatiable computer habits and many late nights over the past year. Neither would it have been done without our dogs, who kept our feet warm while we worked and forced us to get out of the house at least once each day.

We all owe a debt of gratitude to the Excel group at Microsoft, past and present, for making Excel the amazing development platform it is today. It is their dedication and commitment to us that makes Excel application development possible and enjoyable. They have repeatedly demonstrated their willingness to listen to and implement our suggestions over the years.

There are many people we want to thank at Addison-Wesley Professional, particularly our editor Joan Murray for her support while writing the book, Anne Goebel for steering us through the production process, and Curt Johnson for getting it on the shelves.

The quality of a technical book depends as much on the reviewers as the authors, so we want to thank all our technical reviewers. Most of your suggestions were implemented. At the risk of offending the others, we would particularly like to thank Bill Manville, John Peltier, and Gabhan Berry for the quality and rigor of their reviews.

Finally, we want to thank you for buying this book. Please tell us what you think about it, either by e-mail or by writing a review at Amazon.com.

Thank you,

Rob Bovey
Dennis Wallentin
Stephen Bullen
John Green

ABOUT THE AUTHORS

Rob Bovey (robbovey@appspro.com) is president of Application Professionals, a software development company specializing in Microsoft Office, Visual Basic, and SQL Server applications. He brings many years of experience creating financial, accounting, and executive information systems for corporate users to Application Professionals. You can visit the Application Professionals Web site at www.appspro.com.

Rob developed several add-ins shipped by Microsoft for Microsoft Excel, co-authored the *Microsoft Excel 97 Developers Kit* and contributed to the *Excel 2002 and 2007 VBA Programmer's References*. He earned his Bachelor of Science degree from The Rochester Institute of Technology and his MBA from the University of North Carolina at Chapel Hill. Microsoft has awarded him the title of Most Valuable Professional each year since 1995.

Dennis Wallentin (dennis@excelkb.com) is located in Östersund, Sweden, where he lives with his wife and two daughters. Dennis has been developing Excel business solutions since the 1980s and he has a Master's degree in business management and accounting.

He is the founder of XL-Dennis, which delivers solutions for all sizes of companies including the public sector both in Sweden and internationally. He also writes reviews about new Excel versions, books, and other related Excel articles for Swedish magazines. For the last few years he has specialized in creating Excel business solutions based on .NET technologies, including Visual Studio Tools for Office System (VSTO).

Stephen Bullen (stephen@oaltd.co.uk) lives in Woodford Green, London, England, with his partner Clare, daughter Becky, and their dogs, Fluffy and Charlie. A graduate of Oxford University, Stephen has an MA in engineering, economics, and management, providing a unique blend of both business and technical skills.

He is now an employee of Merrill Lynch in London, managing a global spreadsheet development team producing Front Office pricing and risk management tools.

Stephen's Web site, www.oaltd.co.uk, provides a number of helpful and interesting utilities, examples, tips, and techniques to help in your use of Excel and development of Excel applications.

Stephen contributed chapters to John Green's *Excel 2000 VBA Programmer's Reference* and co-authored subsequent editions, published by Wrox Press.

He has been active in various Excel-related online communities for more than 15 years. In recognition of his knowledge, skills, and contributions, Microsoft has awarded him the title of Most Valuable Professional each year since 1996.

John Green (greenj@bigpond.net.au) lives and works in Sydney, Australia, as an independent computer consultant, specializing in integrating Excel, Access, Word, and Outlook using VBA. He has more than 30 years of computing experience, a chemical engineering degree, and an MBA.

He wrote his first programs in FORTRAN, took part in the evolution of specialized planning languages on mainframes, and, in the early 1980s, became interested in spreadsheet systems, including 1-2-3 and Excel.

John established his company, Execuplan Consulting, in 1980, developing computer-based planning applications and training users and developers.

John has had regular columns in a number of Australian magazines and has contributed chapters to a number of books, including *Excel Expert Solutions* and *Using Visual Basic for Applications 5*, published by Que. He is the principal author of *Excel 2000 VBA Programmer's Reference* and its subsequent editions, published by Wrox Press.

Between 1995 and 2006 he was accorded the status of Most Valuable Professional by Microsoft for his contributions to the CompuServe Excel forum and MS Internet newsgroups.

INTRODUCTION

About This Book

Microsoft Excel is much more than just a spreadsheet. With the introduction of the Visual Basic Editor in Excel 97, followed by the significantly improved stability of Excel 2000, Excel became a respected development platform in its own right. Excel applications are now found alongside those based on C++, Java, and the .NET development platform, as part of the core suite of mission-critical corporate applications.

Unfortunately, Excel is still too often thought of as a hobbyist platform, that people only develop Excel applications in their spare time to automate minor tasks. A brief look at many Excel VBA books seems to confirm this opinion. These books focus on the basics of automating Excel tasks using VBA. This book is the first of its kind in providing a detailed explanation of how to use Excel as the platform for developing professional quality applications.

While most other major development platforms seem to have a de facto standard text that explains the commonly agreed best practices for architecting, designing, and developing applications using that platform, until now Excel has not. This book attempts to fill that gap. The authors are professional Excel developers who create Excel-based applications for clients ranging from individuals to the largest multinational corporations. This book explains the approaches we use when designing, developing, distributing, and supporting the applications we write for our clients.

Who Should Read This Book

This is not a beginner-level book. If you do not already have a clear understanding of the core Excel object model and a basic understanding of Excel VBA development this is not the place to start. We assume that readers of this book have already read and (mostly) understood our *Excel 2002 or 2007 VBA Programmer's Reference*, John Walkenbach's *Excel Power Programming*, or similar titles. This book begins where other Excel VBA books end.

Owners of the first edition of *Professional Excel Development* have a different decision to make. Should you purchase the second edition? We have made numerous corrections and improvements throughout this edition as well as expanding it with over 300 pages of new material that you simply will not find anywhere else.

In the interest of full disclosure, however, we want to be very clear that the bulk of the new material is aimed at Excel developers who are working with Excel 2007 and Visual Studio 2008. If you own the first edition of this book and your primary focus is developing VBA applications in Excel 2003 and earlier, you will see incremental rather than revolutionary improvements in this edition. We don't want to discourage you from upgrading to the second edition and would welcome it if you choose to do so. But most of all we want you to be satisfied with our work, so we state the pros and cons of upgrading honestly to help you make an informed decision.

Excel Developer Categories

Excel developers can be divided into five general categories based on their experience and knowledge of Excel and VBA. This book has something to offer each of them, but with a focus on the more advanced topics. Putting yourself into one of these categories might help you decide whether this is the right book for you.

Basic Excel **users** probably don't think of themselves as developers at all. Excel is no more than a tool to help them get on with their job. They start off using Excel worksheets as a handy place to store lists or perform simple repetitive calculations. As they discover more Excel features their workbooks may begin to include more complex worksheet functions, pivot tables, and charts. There is little in this book for basic Excel users, although Chapter 4, "Worksheet Design," details the best practices to use when designing and laying out a worksheet for data entry; Chapter 20, "Data Manipulation Techniques," explains how to structure a worksheet and

which functions and features to use to manipulate their lists; and Chapter 21, “Advanced Charting Techniques,” explains how to get the most from Excel’s chart engine. The techniques suggested in these chapters should help the basic Excel user avoid some of the pitfalls often encountered as their experience and the complexity of their worksheets increase.

Excel **power users** have a broad understanding of Excel’s functionality and they know which tool or function is best used in a given situation. Power users create complex workbooks for their own use and are often called on to help develop workbooks for their colleagues, or to identify why their colleagues’ workbooks don’t work as intended. Power users occasionally use snippets of VBA, either found on the Internet or created with the macro recorder, but struggle to adapt the code to their needs. As a result, their code tends to be messy, slow, and hard to maintain. While this book is not a VBA tutorial, power users have much to gain from following the best practices we suggest for both worksheets and code modules. Most of the chapters in the book are relevant to power users who have an interest in improving their Excel and VBA development skills.

VBA developers make extensive use of VBA code in their workbooks—often too much. They are typically either power users who started to learn VBA too early or Visual Basic developers who switched to Excel VBA development. While they may be proficient with VBA they believe every problem must have a VBA solution. They tend to lack the experience required to know when a problem is best solved using Excel, when a problem is best solved using VBA, and when the best solution is a combination of the two. Their solutions are often cumbersome, slow, and make poor use of the Excel object model. This book has much to offer VBA developers to improve their use of Excel itself, including best practices for designing worksheets and how to use Excel’s features for data entry, analysis, and presentation. The book also seeks to improve their Excel VBA development skills by introducing advanced coding techniques, detailing VBA best practices, and explaining how to improve VBA code performance.

Excel developers realize that the most efficient and maintainable applications are those that make the most of Excel’s built-in functionality, augmented by VBA where appropriate. They are confident in developing Excel-based applications for their colleagues or as part of an in-house development team. While their knowledge of Excel is put to good use in their applications, their design techniques tend to be limited, and they are reluctant to use other languages and applications to augment their Excel solutions. They have probably read John Walkenbach’s *Excel 2003 or 2007 Power Programming* and/or our own *Excel 2002 or 2007 VBA Programmer’s Reference*. Now they need a book to take them to the highest

level of Excel application development—that of the professional developer. This is the book to do that.

Professional Excel developers design and develop for their clients or employer Excel-based applications and utilities that are robust, fast, easy to use, maintainable, and secure. While Excel forms the core of their solutions, they use other applications and languages where appropriate, including third-party ActiveX controls, Office automation, Windows API calls, external databases, various standalone programming languages, and XML. This book teaches all of those skills. If you are already a professional Excel developer, you will know that learning never stops and will appreciate the knowledge and best practices presented in this book by four of your peers.

Excel as an Application Development Platform

If we look at Excel as a development platform rather than just a spreadsheet, we find that it provides five fundamental components we can use in our applications:

- The worksheets, charts, and other objects used to create a user interface and presentation layer for data entry and reporting
- The worksheets used as simple data stores for lists, tables, and other information required by our application
- VBA code and UserForms for creating business logic and advanced user interfaces
- Worksheet formulas used as a declarative programming language for high-performance numerical processing
- The Excel object model, allowing programmatic control of (nearly) all of Excel's functionality, both from within Excel and from outside it

The Worksheet as a Presentation Layer for Data Entry and Reporting

Most people think about Excel in terms of typing numbers into cells, having some calculations update, and seeing a result displayed in a different cell or on a chart. Without necessarily thinking in such terms, they are using the worksheet as a user interface for their data entry and reporting and are generally comfortable with these tasks. The in-cell editing, validation, and formatting features built in to Excel provide a rich and compelling data entry experience, while the charting, cell formatting, and drawing tools provide a presentation-quality reporting mechanism.

It is hard to imagine the code that would be required if we tried to reproduce this experience using the tools available in most other development environments, yet Excel provides these features right out of the box for use in our Excel-based applications. The biggest problem we face is how to add structure to the free-form worksheet grid to present a simple and easy-to-use interface, while leveraging the rich functionality of Excel. Chapter 4 introduces some techniques and best practices for developing worksheet-based data entry forms, while Chapter 21 covers charting capabilities.

The Worksheet as a Simple Data Store

What is a worksheet when it's never intended to be shown to the end user? At its simplest, it's no more than a large grid of cells in which we can store just about anything we want, including numbers, text, lists, tables, and pictures. Most applications use some amount of static data or graphical resources. Storing that information in a worksheet makes it both easy to access using VBA and simple to maintain. Lists and tables in worksheets can directly feed Excel's data validation feature (as shown in Chapter 4), greatly simplify the creation and maintenance of command bars (Chapter 8, "Advanced Command Bar Handling"), and allow us to construct dynamic UserForms (Chapter 13, "UserForm Design and Best Practices").

VBA Code and UserForms

We expect most readers of this book have at least some familiarity with VBA. If not, we suggest you read one of the resources mentioned at the beginning of this chapter before continuing much further. Many people see the "A" in VBA as meaning the language is somehow less than Visual Basic itself. In fact, both VB6 and Office use exactly the same DLL to provide the keywords, syntax, and statements we program with.

Most beginner and intermediate VBA developers use VBA as a purely procedural language, with nearly all their code residing in standard modules. VBA also allows us to create applications using an object oriented programming (OOP) approach, in which class modules are used to create our own objects. Chapter 7, "Using Class Modules to Create Objects," and Chapter 14, "Interfaces," explain how to use VBA in this manner, while basic OOP concepts (such as encapsulation) are used throughout the book.

Most of this book is dedicated to explaining advanced VBA techniques and a professional approach to application design and development that can put VBA in Excel on par with, and sometimes in front of, VB6 or VB.Net for application development. In Chapters 23 through 26

we show that Excel developers can achieve the best of both worlds by combining Excel with VB6 or VB.Net in a seamless application.

The Worksheet as a Declarative Programming Language

Take the following code:

```
dSales = 1000
dPrice = 10.99
dRevenue = dSales * dPrice
```

That could easily be a few lines of VBA. We give the variable `dSales` a value of 1000, the variable `dPrice` a value of 10.99, and then calculate the revenue as sales times price. If we change the names of the variables and adjust the spacing, the same code could also be written as

```
D1    =1000
D2    =10.99
D3    =D1*D2
```

This looks much more like worksheet cell addresses and formulas than lines of VBA code, showing that worksheet formulas are in fact a programming language of their own if we choose to think of it in those terms. The `IF()` worksheet function is directly equivalent to the `If...Then...Else` VBA statement, while the judicious use of circular references and iteration can be equivalent to either the `For...Next` or `Do...Loop` structures.

Instead of stating a set of **operations** that are executed line-by-line, we “program” in this language by making a set of **declarations** (by typing formulas and values into worksheet cells), in any order we want:

“D3 is the product of D1 and D2”

“D1 has the value 1000”

“D2 has the value 10.99”

To “run” this program, Excel first examines all the declarations and builds a **precedence tree** to identify which cells depend on the results of which other cells and thereby determine the most efficient order in which the cells must be calculated. The same precedence tree is also used to identify

the minimum set of calculations that must be performed whenever the value in a cell is changed. The result is a calculation engine that is vastly more efficient than an equivalent VBA program, and one that should be used whenever complex numerical computations are required in your application.

Microsoft Excel is unique among application development platforms in providing both a procedural (VBA) and a declarative (worksheet functions) programming language. The most efficient Excel application is one that makes appropriate use of both these languages.

It is assumed the reader of this book has a basic understanding of worksheet functions, so Chapter 20 focuses on using advanced worksheet functions (including best-practice suggestions for handling circular references) and Excel's other data analysis features.

The Excel Object Model

While the other four components of the Excel platform are invaluable in the development of applications, it is probably the rich Excel object model that provides the most compelling reason to base our applications in Excel. Almost everything that can be accomplished through the Excel user interface can also be accomplished programmatically using the objects in the Excel object model. (Accessing the list of number formats and applying a digital signature to a workbook are perhaps the most notable exceptions.)

The vast feature set exposed by these objects makes many complex applications fairly simple to develop. Unlike most other development platforms, there is no need to figure out how to program these features from scratch. Excel provides them ready-made, so all we need to do is determine how to plug them together most effectively. This book does not attempt to explore and document every obscure niche of the Excel object model. Instead, we demonstrate the best way to use the objects we most commonly use in our own application development.

Structure

Over the course of this book we cover both the concepts and details of each topic and apply those concepts to a time sheet reporting and analysis application that we will build in stages as we move along. The chapters are

therefore arranged approximately in the order in which we would design and develop an Excel application:

- **Chapter 2** discusses the different styles of application we might choose to create.
- **Chapter 3** identifies some general best practices for working with Excel and VBA. These are followed throughout the book.
- **Chapter 4** explains how to design and structure a worksheet for data entry and analysis.
- **Chapters 5 and 6** introduce two specific types of application—the add-in and the dictator application, which form the basis of our time sheet reporting and analysis application.
- **Chapter 7** introduces the use of class modules in our Excel applications.
- **Chapters 8 to 11** discuss topics relevant to building command bar and Ribbon user interfaces as well as designing applications that must run in all current Excel versions using a single code base.
- **Chapters 12 to 17** discuss advanced techniques for a range of VBA topics.
- **Chapters 18 and 19** cover database development for Excel developers.
- **Chapters 20 and 21** explain how to efficiently use Excel's features to analyze data and present results.
- **Chapters 22 to 27** look outside Excel, by explaining how to automate other applications and extend Excel with Visual Basic 6, VB.NET, and C.
- **Chapter 28** focuses on how Excel applications can make use of Web Services.
- **Chapter 29** completes the development by explaining how to provide help for, secure, and deploy an Excel application.

Examples

As mentioned previously, throughout the book, we illustrate the concepts and techniques we introduce by building a time sheet data entry, consolidation, analysis, and reporting application. This consists of a data entry template to be completed by each employee, with the data sent to a central location for consolidation, analysis, and reporting. At the end of most chapters we show an updated working example of the application that

incorporates ideas presented in those chapters, so the application grows steadily more complex as the book progresses.

In Chapter 4, we start with a simple data entry workbook and assume that each employee would e-mail the completed file to a manager who would analyze the results manually—a typical situation for a company with just a few employees.

By the end of the book, the data entry workbook will use XML to upload the data to a Web site, where it will be stored in a central database. The reporting application will extract the data from the database, perform various analyses, and present the results as reports in Excel worksheets and charts.

Along the way we rewrite some parts of the application in a number of different ways to show how easy it can be to include other languages and delivery mechanisms in our Excel-based applications. Most chapters also include specific concept examples to illustrate key points that are important to understand but would be too artificial if forced into the architecture of our time sheet application.

Supported Versions of Excel

When we develop an Excel application for a client, that client's upgrade policy usually determines the version of Excel we must use. Few clients agree to upgrade just so we can develop using the latest version of Excel unless there is a compelling business requirement that can only be satisfied by using features the latest version introduces. At the time of this writing, an extremely unscientific poll (based on postings to the Microsoft support newsgroups) seems to indicate the following approximate usage distribution for each current version of Excel:

Excel 2000	10%
Excel 2002	15%
Excel 2003	50%
Excel 2007	25%

There are still a small number of users on Excel 97 and earlier versions, but for various reasons we no longer consider these versions of Excel to be viable development platforms. We therefore decided to use Excel 2000 as our lowest supported version. Many features we discuss, especially when we cover XML and the .NET development platform, are only supported in Excel 2002 or 2003 and higher. Whenever we discuss a feature

that is only supported in a later version of Excel we state which version(s) it applies to.

Typefaces

The following text styles are used in this book:

Menu items and dialog text are shown as *Tools > Options > Calculation > Manual*, where the “>” indicates navigation to a submenu or dialog tab.

```
Sub SomeCode()  
    'Code listings are shown like this  
End Sub
```

Code within the text of a paragraph is shown in a fixed-width font like `Application.Calculation = xlManual`.

Paths on the CD are shown as `\Concepts\Ch14 - Interfaces`.

New terms introduced or defined appear **like this**.

Important points or emphasized words appear *like this*.

On the CD

Most of the code listings shown in the book are also included in example workbooks on the accompanying CD. For clarity, the code shown in the printed examples may use shorter line lengths, reduced indent settings, fewer code comments, and less error handling than the corresponding code in the workbooks. The CD has three main directories, containing the following files:

- **\Tools** contains a number of tools and utilities developed by the authors that we have found to be invaluable during our application development. The `MustHaveTools.htm` file contains details about each of these tools and links to other third-party utilities.
- **\Concepts** has separate subdirectories for each chapter, each one containing example files to support the text of the chapter. For best results, we suggest you have these workbooks open while reading the corresponding chapter.

- **Application** has separate subdirectories for the chapters where we have updated our time sheet example application. These chapters end with a Practical Example section that explains the changes made to implement concepts introduced in that chapter.

Help and Support

By far the best place to go for help with any of your Excel development questions, whether related to this book or not, are the Microsoft support newsgroup archives maintained by Google at <http://groups.google.com>. A quick search of the archives is almost certain to find a question similar to yours, already answered by one of the many professional developers who volunteer their time helping out in the newsgroups, including all the authors of this book. On the rare occasions that the archives fail to answer your question, you're welcome to ask it directly in the newsgroups by connecting a newsreader (such as Outlook Express) to msnews.microsoft.com and selecting an appropriate newsgroup, such as

microsoft.public.excel.misc for general Excel questions

microsoft.public.excel.programming for VBA-related questions

microsoft.public.excel.worksheet.functions for help with worksheet functions

For assistance with Excel and VB.NET integration issues we recommend the MSDN VSTO Web forum located here:

<http://social.msdn.microsoft.com/Forums/en-US/vsto/threads/>

A number of Web sites provide a great deal of information and free downloadable examples and utilities targeted towards the Excel developer, including

www.appspro.com

www.excelkb.com

www.oaltd.co.uk

<http://peltiertech.com>

www.cpearson.com

<http://msdn.microsoft.com/office>

The Professional Excel Development Web Site

As an experiment for the second edition of *Professional Excel Development*, we are introducing a new Web site to accompany the book at www.ProExcelDev.net.

As of this writing the site does not yet exist, so it is difficult to say exactly what you will find there. However, at a minimum you will find the latest corrections, bug fixes, and clarifications related to this book. Our hope is to eventually expand the site to provide more in-depth coverage of popular topics than we were able to fit into our publishing deadline as well as blogs and possibly even interactive technical forums.

Feedback

We have tried very hard to present the information in this book in a clear and concise manner, explaining both the concepts and details needed to get things working as well as providing working examples of everything we cover. We have tried to provide sufficient information to enable you to apply these techniques in your own applications without getting bogged down in line-by-line explanations of entire code listings.

We'd like to think we've been successful in our attempt, but we encourage you to let us know what you think. Constructive criticism is always welcomed, as are suggestions for topics you think we may have overlooked. Please send feedback to the following authors:

Rob Bovey: robbovey@appspro.com

Dennis Wallentin: dennis@excelkb.com

APPLICATION ARCHITECTURES

One of the first decisions to be made when starting a new project is how to structure the application. This chapter explains the various architectures we can use, the situations where each is most applicable, and the pros and cons of each choice.

Concepts

The choice of where to put the code for an Excel application is rarely straightforward. In anything but the simplest of situations there is a trade-off among numerous factors, including

- **Complexity**—How easy will the chosen architecture be to create?
- **Clarity**—How easy will it be for someone other than the author to understand the application?
- **Development**—How easy will it be to modify the code, particularly in a team environment?
- **Extensibility**—How easy is it to add new features?
- **Reliability**—Can the results be relied on? How easily can calculation errors be introduced into the application?
- **Robustness**—How well will the application be able to handle application errors, invalid data, and other problems?
- **Security**—How easy will it be to prevent unauthorized changes to the application?
- **Deployment**—How easy will it be to distribute the application to the end user?
- **Maintainability**—How easy will it be to modify the application once it has been distributed to the end user?

Codeless Applications

The most basic application architecture is one that only uses Excel's built-in functionality. Everyone creates this type of application without knowing it, simply by using Excel. Codeless applications are typically created by beginning to intermediate Excel users who have not yet learned to use VBA. All the custom formatting, validation, formulas, and so on are placed directly on the same worksheet where data entry will be performed. There are some major problems with this approach when it is applied to non-trivial Excel applications, so totally codeless applications are rarely a good choice.

To avoid VBA, the worksheet functions and data validation criteria tend to become convoluted and hard to follow. The equivalent VBA often is easier to understand and maintain. The same worksheet is normally used for data entry, analysis, and presentation. This tends to result in a cluttered appearance that is difficult to understand, unintuitive to use, and almost impossible for anyone except the author to modify reliably.

Codeless applications have to rely on Excel's worksheet protection to prevent users from making unauthorized changes. Worksheet passwords are notoriously easy to break, and a simple copy and paste will wipe out any cell data validation. Codeless applications are therefore neither secure nor robust.

Without code, we are unable to provide much assistance to users; we have to rely on them to do everything themselves—and do it correctly—instead of providing reliable helper routines that automate some of their tasks. The more complex the application, the less likely it is that all the tasks will be performed correctly.

If we consider the definition of a “program” to be “anything that isn't the data,” we see that all the conditional formatting, data validation, worksheet functions, and so on are really part of the “program,” so codeless applications break the basic tenet of keeping the program and data physically separate. Once users have started to enter data it is difficult to distribute an updated workbook to them without losing the data they've already entered. You have to either hope the user can copy the existing data to the new workbook correctly or write a conversion program to copy the data from the old workbook to the new workbook for them.

Codeless applications can work well in the following situations:

- There will only be one copy of the application workbook, so any changes can be made directly to that workbook.
- Each copy of the workbook will have a short lifetime. In this case, the assumption is that the workbooks will not need updating after they have been distributed.

- The end users will maintain the workbook themselves or the workbook will not require any maintenance at all.
- There is a small number of relatively sophisticated end users who can be trained well enough to ensure the application is used correctly and not inadvertently broken.

A good example of a codeless application would be a simple survey or data collection form that requires the end user to fill in the details and e-mail the completed workbook to a central address for consolidation and analysis. The main benefit of a codeless application in such a situation is the avoidance of Excel's macro security warnings and the corresponding assurance that there is nothing malicious in the file.

Self-Automated Workbooks

A self-automated workbook is one in which the VBA code is physically contained within the workbook it acts upon. The automation code can be as simple as ensuring the workbook always opens with Sheet1 active or as complex as an entire application. This is usually the first type of application a beginning VBA developer produces, by adding helper routines to a workbook that get progressively numerous and more complex over time.

Once we introduce VBA into the workbook we acquire much more flexibility in how we provide the features required by the application. We can make a considered choice whether to use Excel's built-in functions or write our own equivalents to avoid some of Excel's pitfalls. For example, Excel's data validation feature may not operate correctly when entries are made in multiple cells simultaneously, and data validation is usually cleared when data is pasted onto a range that uses it. We can work around these limitations by trapping the `Worksheet_Change` event and performing our own validation in code, making the application more robust, reliable, and secure.

The Workbook and Worksheet code modules provided by Excel allow us to trap the events we want to use. Any ActiveX controls we add to a worksheet are automatically exposed in that worksheet's code module. This is the simplest application architecture to create and probably the simplest to understand—most VBA developers have written an application of this type and therefore understand, for example, how the code within a worksheet code module is triggered.

The biggest advantage of the self-automated workbook application architecture is its ease of deployment. There is only one file to distribute. There is no need to install or configure anything, and because the code is physically stored within the workbook, it is available and working as soon as the workbook is opened.

Unfortunately, the self-automated workbook's clearest advantage is also its biggest problem. When the code is physically inside the workbook, how do you update the code without affecting the data that has been entered on the worksheets? While it is possible to write VBA that modifies the code within another workbook, the user has to make a specific macro security setting to allow that to happen (in Excel 2002 and above). Also, it is only possible to unprotect and reprotect the VBA project using SendKeys, which cannot be relied on to work in foreign-language versions of Excel or if Excel doesn't have the focus. Even if the project could be unprotected and reprotected, saving the updated project would remove any digital signature that had been applied, resulting in macro virus warnings every time the workbook was subsequently opened. The only reliable way self-automated workbooks can be updated is to provide a completely new workbook with VBA code (or instructions) to copy the data from the old workbook. Self-automated workbooks are a good choice if the following conditions apply:

- The VBA code contained within the workbook provides functionality specific to that workbook (as opposed to general purpose utilities).
- There will only be one copy of the application workbook, so any changes can be made directly to that workbook.
- The workbook will have a short lifetime or will be distributed to a large audience, in which case ease of deployment becomes a significant consideration.
- The workbook does not contain any data that will need to be retained during an update, such as one that obtains its data from an external data source or saves the data entered into it to an external data repository.

General Purpose Add-ins

An add-in is a specific type of application, usually used to add features to Excel. The worksheets in an add-in are hidden from the user, so the user never interacts directly with the workbook. Instead, the add-in exposes its features by adding items to Excel's menus and toolbars or Ribbon, hooking key combinations, trapping Excel events, and/or exposing functions to be used from worksheets in other workbooks. VBA procedures in an add-in can also be executed by typing their fully qualified name (for example, `MyAddin.xla!MyProcedure`) in the *Tools > Macro > Macros* dialog, even though they do not appear in the list of available macros.

The procedures in a general purpose add-in will always be available to the Excel user, so this application architecture is most appropriate for utility

functions that are designed to work with any file, typically using the `ActiveWorkbook`, `ActiveSheet`, or `Selection` objects to identify the items to operate on.

Care should be taken to handle potential user errors, where procedures in the add-in may be called from a context in which they won't work. For example, if your add-in changes the case of the text in the selected cell, you must verify that a cell is selected, isn't locked, and doesn't contain the result of a formula. Similarly, if your code applies custom formatting to the active worksheet, you must verify that there is an active sheet (there may be no workbooks open), it's a worksheet (not a chart or macro sheet, for example), and it's not protected.

An add-in is just a much hidden workbook, so it doesn't appear in the list of workbooks or the `VBA Workbooks` collection. It is, however, just like any other workbook in almost every other respect and should therefore be easy for an intermediate Excel/VBA developer to understand and maintain. In fact you can toggle between having the add-in workbook behave like an add-in or a normal workbook by simply changing the `IsAddin` property of its `ThisWorkbook` object in the VBE Properties window between `True` and `False`.

Because add-ins never expose their worksheets to the user, all user interaction is done with `UserForms` (although the `VBA InputBox` and `MsgBox` functions can be used in simple situations). This gives us a high level of control over user inputs, allowing us to create applications that are robust and reliable—assuming we include data validation code and good error handling.

If the add-in needs to persist any information, such as the most recent selections made by the user in a `UserForm`, that information should be kept separate from the add-in file, either by storing it in the registry (using `SaveSetting/GetSetting`) or in a separate file such as an INI file. By following this practice you ensure the add-in will never need to be saved by the end user and can simply be replaced by a new version if an update is required.

If you are willing to trust the end user to install the add-in correctly, it is also easy to deploy—just send the XLA file with instructions to either copy it into their Library folder or to use the Browse button in the *Tools > Add-Ins* dialog to locate the file. The alternative is to use an installation routine to write the registry entries Excel uses to maintain its add-ins list, such that the add-in is automatically opened and installed when the client next starts Excel. These registry entries are covered in detail in Chapter 29, “Providing Help, Securing, Packaging, and Distributing.”

Structure of a General Purpose Add-in

Most general purpose add-ins use the same basic structure:

- Code in an `Auto_Open` or `Workbook_Open` procedure that creates the add-in's menu items and sets up the keyboard hooks. Each menu item has its `OnAction` property set to call the appropriate procedure in the add-in file.
- Procedures associated with each menu item that are located in a standard code module.
- (Optionally) Public functions located in a standard code module that are exposed for use in worksheet formulas.
- Code in an `Auto_Close` or `Workbook_Close` procedure that removes the add-in's menu items and clears its keyboard hooks.

Application-Specific Add-ins

As mentioned previously, the main problem with both codeless and self-automated workbooks is that the program is physically stored in the same file as the data it works with. It is difficult to reliably update the program part of those workbooks without affecting or destroying the data.

The alternative is to structure the application such that all the code is contained within one workbook, while a separate workbook is used for data entry, analysis, and so on. One such architecture is that of an application-specific add-in. These are similar to general purpose add-ins, but instead of immediately setting up their menu items, keyboard hooks, and so on, they stay invisible until the user opens a workbook the add-in can identify as one that it should make itself available for.

In a typical application-specific add-in architecture, the user would be supplied with at least two workbooks: the XLA workbook containing the program and a template workbook used for data entry. The template workbook(s) contains some kind of indicator the add-in can use to identify it, usually either a hidden defined name or a custom document property.

The key benefit of using an application-specific add-in is that we can safely distribute updates to the code, knowing we will not cause any harm to the user's data. There is, however, a small price to pay for this convenience:

- Splitting the application into two (or more) workbooks makes it slightly harder to manage, because we have to keep the correct versions of both workbooks synchronized during the development process. Simple version control is discussed in more detail in Chapter 3, "Excel and VBA Development Best Practices."

- The application is slightly harder for other developers to understand, particularly if they are used to single-workbook applications or do not understand the technique of using class modules to hook application-level events, as explained in Chapter 7, “Using Class Modules to Create Objects.”
- Deployment is more complicated, because we need to distribute multiple files. Deployment strategies are covered in Chapter 29.

Structure of an Application-Specific Add-in

Application-specific add-ins are similar in structure to general purpose add-ins, but with extra code to identify when to enable or disable the menu items:

- A class module used to trap the application-level events.
- Code in an `Auto_Open` or `Workbook_Open` procedure adds the add-in's menu items. Each menu item has its `OnAction` property set to call the appropriate procedure in the add-in file, but these menu items are all initially either disabled or hidden. It then creates an instance of the class module and initializes application event hooks.
- Procedures associated with each menu item that are located in a standard code module.
- (Optionally) Public functions located in a standard code module that are exposed for use in worksheet formulas.
- Code in the class module that hooks the application-level `WorkbookActivate` event, checks whether the workbook “belongs” to the add-in and if so enables the menu items and sets up the keyboard hooks.
- Code in the class module hooks the application-level `WorkbookDeactivate` event, to disable the menu items and remove the keyboard hooks when no application workbook is active.
- Code in an `Auto_Close` or `Workbook_Close` procedure removes the add-in's menu items.

General purpose and application-specific add-ins are discussed in more detail in Chapter 5, “Function, General, and Application-Specific Add-ins.”

Dictator Applications

All the architectures considered so far have sought to enhance Excel in some way to improve the end user's experience when they're using our application. In contrast, dictator applications attempt to take over the Excel user interface completely, replacing Excel's menus with their own and exercising a high level of control over the user interface. In the ideal dictator application, users will not be able to tell they are working inside Excel.

These applications are created in Excel to use the features Excel provides, but those features are entirely controlled by the application. The user interface is made up of tightly controlled data entry worksheets and/or UserForms designed to appear like any other Windows application. These applications require large amounts of code to provide that degree of control, but that control allows us to write large-scale, fully functional Windows applications on par with any that can be written in Visual Basic or other "mainstream" application development platforms. In fact, by building our application within Excel, we have a head start over other development platforms because we are immediately able to utilize the incredible amount of functionality Excel provides.

As dictator applications become more complex, they will often start to use functionality that only exists in the most recent versions of Excel (such as the XML import/export introduced in Excel 2003), so we need to decide what should happen if the application is opened in an older version of Excel. If the functionality being used is a core part of the application, it is unlikely the application will be usable at all in older versions of Excel. If the use of the new features can be limited to a small part of the application, it may make more sense to just disable user interface access to those features when running in older versions of Excel or provide separate procedures for older versions to use.

Making use of new Excel features often results in compile errors if the application workbook is opened in an older version of Excel, so many dictator applications use a "front-loader" workbook to do an initial version check, verify that all external dependencies are available, and then open and run the main application workbook if all the checks are okay. If the checks fail, we can provide meaningful error messages to the end user (such as "This application requires Excel 2003 or higher and will not work in Excel 2000").

There's no escaping the fact that dictator applications are much more complicated than either self-automated workbooks or application-specific add-ins and will require an intermediate to advanced level Excel/VBA developer to create and maintain them. The complexity of dictator

applications can be mitigated by following the best practices advice discussed in Chapter 3 (general advice) and Chapter 6, “Dictator Applications” (specific advice for dictator applications).

Once the decision to build a dictator application has been made, we have an incredible amount of flexibility in terms of physically creating the application. The data can be stored in one or more separate workbooks, local databases such as Access, or a central database such as SQL Server. We can put all the code into a single add-in workbook or have a small core add-in with numerous applets that plug into the core, each performing a specific task. The decision will probably be a trade-off between (at least) the following considerations:

- A single-workbook structure tends to be easier for a single developer to maintain, because everything is in the one place.
- A multiworkbook structure is easier for a team of developers to create, because each developer can work on her own applet without conflicting with another team member.
- If a multiworkbook structure is built so each plug-in applet is not loaded until it is first used, the initial opening of the “core” add-in will be faster than loading the full application of the single-workbook structure—though modern PCs may make that difference appear immaterial.
- A single-workbook structure must be updated in its entirety, but the applets of a multiworkbook structure can be updated and deployed independently.
- The code required to implement a multiworkbook plug-in architecture is complex, and may be too complex for the intermediate VBA developer to fully understand—though we explain it in Chapter 14, “Interfaces.”

Requirements of a Dictator Application

To look and operate like a standalone Windows application, a dictator application needs to modify many Excel application properties, from turning on `IgnoreOtherApplications` (so double-clicking an XLS file in Explorer will not use our instance of Excel) to turning off `ShowWindowsInTaskBar` (because we may have multiple workbooks open and do not want each of them to spawn new TaskBar buttons), as well as hiding all the built-in command bars. Unfortunately, Excel will remember

many of these settings the next time it is started, so every dictator application must first record the existing state of all the settings it changes and restore them all when it closes. If the code to do this is written as two separate procedures that are assigned shortcut keys, they also provide an easy way to switch between the application user interface and the Excel user interface during development.

Once a snapshot of the user's settings has been taken, the dictator application can set the application properties it requires. It then needs to lock down Excel to prevent the user from doing things we don't want them to do. This includes

- Hiding and disabling all built-in command bars or Ribbon tabs (including shortcut command bars), and then setting up our own.
- Protecting our command bars and disabling access to the command bar customization dialog.
- Disabling all the shortcut key combinations that Excel provides, and then optionally reenabling the few we want to be exposed to the user.
- Setting `Application.EnableCancelKey` to `xlDisabled` at the start of every entry point to prevent users from stopping the code.
- When using worksheets as data entry forms, we don't want the user to be able to copy and paste entire cells, since that would include all formatting, data validation, and so on, so we need to turn off drag-and-drop (which does a cut and paste), redirect both `Ctrl+X` and `Shift+Delete` to do a Copy instead of a Cut, and redirect `Ctrl+V` and `Shift+Insert` to paste only values.

Having locked down the Excel environment while our application is running, we need to provide a mechanism to access the code so that we can debug the application. One method is to set a global `IsDevMode` Boolean variable to `True` if a particular file exists in the application directory or (more securely) depending on the Windows username. This Boolean can then be used throughout the application to provide access points, such as enabling the `Alt+F11` shortcut to switch to the VBE, adding a Reset menu item and/or shortcut key to switch back to the Excel environment, and not setting the `EnableCancelKey` property, to allow the developer to break into the code. This variable can also be used within error handlers, to control whether to display a user- or developer-oriented error message.

Structure of a Dictator Application

A typical dictator application uses the following logical structure:

- A front-loader/startup procedure to perform version and dependency checks as well as any other validation required to ensure the application can run successfully.
- A core set of procedures to
 - Take a snapshot of the Excel environment settings and to restore those settings.
 - Configure and lock down the Excel application.
 - Create and remove the application's command bars.
 - Handle copying and pasting data within the worksheet templates.
 - Provide a library of common helper procedures and classes.
 - (Optionally) Implement a plug-in architecture using class modules, as described in Chapter 14.
- A backdrop worksheet, to display within the Excel window while UserForms are being shown, usually with some form of application-specific logo (if we're primarily using forms for the user interface).
- Multiple independent applets that provide the application's functionality.
- Multiple template worksheets used by the applets, such as data entry forms or preformatted report templates.

Physically, all the elements that make up a typical dictator application can reside in a single workbook or can be distributed across multiple workbooks. Dictator applications are discussed in more detail in Chapter 6.

Technical Implementations

In our discussion of the main types of application architecture there has been an underlying assumption that the application will be written using VBA. That need not be the case, as we discuss in Chapters 23 through 27, where we examine how we can use the C API to create XLL add-ins and use Visual Basic 6 and/or VB.Net to support our VBA procedures and create COM add-ins.

Any of these architectures can be implemented using either a traditional procedural design, where most of the functionality is implemented using helper procedures in standard code modules, or an object-oriented approach, where the functionality is implemented as properties and methods of class modules, as discussed in Chapter 7.

Summary

The five main types of application architecture each have their pros and cons and each is the most applicable to certain situations. The choice of architecture should be made carefully, with appropriate consideration given to ongoing maintenance (probably by a different person than the original author) as well as just the ease with which the application can be created initially. Table 2-1 lists each architecture and the advantages and disadvantages of each.

Table 2-1 Summary of Application Architectures

Architecture	Pros	Cons	Applicable To
Codeless Workbook	No VBA requirement. No macro security issues. Easy to deploy.	Usually cluttered and hard to use. Neither robust nor reliable. Doesn't provide much assistance to the user. Difficult to update.	Simple data entry forms, surveys, etc.
Self-automated workbook	Simple application, easy for a beginner VBA developer to understand. VBA can be used to improve robustness and reliability. Provides a lot of extra functionality for the user. Easy to deploy.	If the VBA needs to be updated, it will be difficult or impossible to do so once deployed.	More complex data-entry forms, where the VBA can be used to improve the quality of the data being entered, but there is little data stored in the workbook long-term.

Table 2-1 Summary of Application Architectures

Architecture	Pros	Cons	Applicable To
General purpose add-in	<p>Designed to extend Excel's functionality.</p> <p>Simple application, only slightly more complex than an automated workbook.</p> <p>Easy to deploy (though not as simple as a workbook).</p>	<p>Must include robust context checks and error handling.</p> <p>Harder to deploy if it should be automatically ready for use.</p>	<p>Ideal for adding custom functionality to Excel, designed for use with any workbook.</p>
Application-specific add-in	<p>Separates the code from the data, so the code can be updated without affecting the user's work.</p> <p>Removing the code from the data workbooks makes them smaller and avoids the macro security warning.</p>	<p>Slightly more technically complex than the general add-in, requires an intermediate level VBA developer.</p> <p>Slightly harder to deploy, as it requires at least two workbooks to be installed, sometimes to separate locations.</p>	<p>Suitable for applications of any size and complexity.</p>
Dictator application	<p>Can write fully functional applications that appear to be applications in their own right.</p> <p>High degree of control over the user interaction allows you to write very robust and reliable applications.</p> <p>Functionality can be split over multiple workbooks, making them easier for a team to develop and easier to deploy updates.</p>	<p>Much more complex than other architectures.</p> <p>Care must be taken to restore the user's Excel environment.</p> <p>Harder to deploy, typically requiring an installation routine.</p>	<p>Best suited to complex applications or those that require a high degree of control over user interaction.</p>

This page intentionally left blank

EXCEL AND VBA DEVELOPMENT BEST PRACTICES

This chapter appears early in the book because we want you to understand why we do certain things the way we do in later chapters. Unfortunately, this also means we'll have to cover a few topics in this chapter that don't get full coverage until later. For best results, you may want to review this chapter after you've read the rest of the book.

As you read this chapter, you should also keep in mind that even though the practices described here are generally accepted best practices, there will always be certain cases where the best thing to do is not follow the best practice. We try to point out the most common examples of this here and in the best practices discussions in the chapters that follow.

Naming Conventions

The term “naming convention” refers to the system you use to name the various parts of your application. Whenever you declare a variable or create a UserForm, you give it a name. You implicitly name objects even when you don't give them a name directly by accepting the default name provided when you create a UserForm, for example. One of the hallmarks of good programming practice is the consistent use of a clearly defined naming convention for all parts of your VBA application.

Let's look at an example that may help demonstrate why naming conventions matter. In the following line of code:

```
x = wsDataSheet.Range("A1").Value
```

What do you know about `x`? From its usage you can reasonably assume it's a variable. But what data type is it designed to hold? Is its scope public,

module-level, or private? What is its purpose in the program? As it stands, you can't answer any of these questions without searching through the rest of the code. A good naming convention conveys the answers to these questions with a simple visual inspection of the variable name. Here's a revised example (we cover the specifics in detail in the next section):

```
glListCount = wksDataSheet.Range("A1").Value
```

Now you know the scope of the variable (*g* stands for global or public scope), what data type it was designed to hold (*l* stands for the Long data type), as well as having a rough idea of the purpose of the variable (it holds the number of items in a list).

A naming convention helps you to immediately recognize the type and purpose of the building blocks used in an application. This allows you to concentrate on what the code is doing rather than having to figure out how the code is structured. Naming conventions also help make your code self-documenting, which reduces the number of comments required to make the purpose of your code clear.

We present one example of a well-structured naming convention in the following section. This is the naming convention we use throughout the book. You may or may not decide to use the naming convention we present here; this is not important. What is important is that you do pick some naming convention and use it consistently. As long as everyone involved in a project understands the naming convention, it doesn't matter exactly what prefixes it uses or what its conventions are for capitalization in variable names. When it comes to the use of a naming convention, consistency rules, both across projects and over time.

A Sample Naming Convention

A good naming convention applies not just to variables, but to all the elements of your application. The sample naming convention we present here covers all the elements in a typical Excel application. We begin with a discussion of variables, constants, and related elements, since these are the most common elements in any application. The general format of the naming convention is shown in Table 3-1. The specific elements of the naming convention and their purposes are described afterwards.

Table 3-1 A Naming Convention for Variables, Constants, UDTs, and Enumerations

Element	Naming Convention
Variables	<scope><array><data type>DescriptiveName
Constants	<scope><data type>DESCRIPTIVE_NAME
User-defined types	Type DESCRIPTIVE_NAME <data type>DescriptiveName End Type
Enumeration types	Enum <project prefix>GeneralDescr <project prefix>GeneralDescrSpecificName1 <project prefix>GeneralDescrSpecificName2 End Enum

The Scope Specifier (<scope>)

g—Public
 m—Module-level
 (nothing)—Procedure-level

The Array Specifier (<array>)

a—Array
 (nothing)—Not an array

The Data Type Specifier (<data type>)

There are so many data types that it's difficult to provide a comprehensive list of prefixes to represent them. The built-in data types are easy. The most frequently used built-in data types get the shortest prefixes. Problems arise when naming object variables that refer to objects from various applications. Some programmers use the prefix “obj” for all object names. This is

not acceptable. However, devising consistent, unique, and reasonably short prefixes for every object type you will ever use is also too much to ask. Try to find reasonably meaningful one- to three-letter prefixes for the object variables you use most frequently and reserve the “obj” prefix for objects that appear infrequently in your code.

Make your code clear, and above all, be consistent. Keep data type prefixes to three or fewer characters. Longer prefixes, in combination with scope and array specifiers, make for unwieldy variable names. Table 3-2 shows some suggested prefixes for the most commonly used data types.

Table 3-2 Suggested Naming Convention Prefixes

Prefix	Data Type	Prefix	Data Type	Prefix	Data Type
b	Boolean	cm	ADODB.Command	cbo	MSForms.ComboBox*
byt	Byte	cn	ADODB.Connection	chk	MSForms.CheckBox
cur	Currency	rs	ADODB.Recordset	cmd	MSForms. CommandButton
dte	Date			ddn	MSForms.ComboBox**
dec	Decimal	cht	Excel.Chart	fra	MSForms.Frame
d	Double	rng	Excel.Range	lbl	MSForms.Label
i	Integer	wkb	Excel.Workbook	lst	MSForms.ListBox
l	Long	wks	Excel.Worksheet	mpg	MSForms.MultiPage
obj	Object			opt	MSForms.OptionButton
sng	Single	cbr	Office.CommandBar	spn	MSForms.SpinButton
s	String	ctl	Office. CommandBarControl	txt	MSForms.TextBox
u	User- Defined Type				

Table 3-2 Suggested Naming Convention Prefixes

Prefix	Data Type	Prefix	Data Type	Prefix	Data Type
v	Variant	cls	User-Defined Class Variable	ref	RefEdit Control
		frm	UserForm Variable	col	VBA.Collection

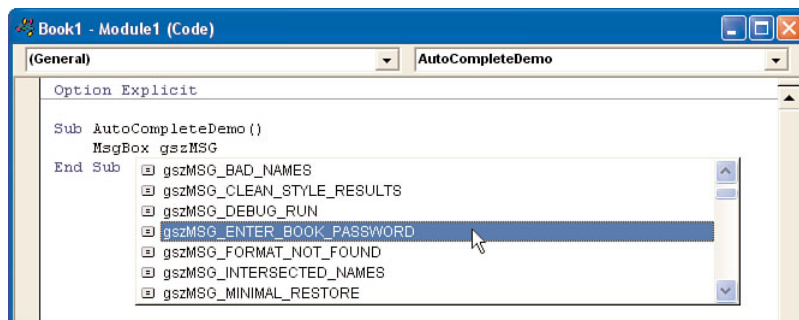
*Used for ComboBox controls with a DropDownCombo Style setting.

**Used for ComboBox controls with a DropDownList Style setting.

Using Descriptive Names

VBA gives you up to 255 characters for each of your variable names. Use a few of them. Don't try to save yourself a little effort by making your variable names very short. Doing so will make your code difficult to understand in the long run, both for you and for anyone else who has to work on it.

The Visual Basic IDE provides an auto-complete feature for identifiers (all the names used in your application). You typically need to type only the first few characters to get the name you want. Enter the first few characters of the name and press Ctrl+Spacebar to activate an auto-complete list of all names that begin with those characters. As you type additional characters, the list continues to narrow down. In Figure 3-1 the Ctrl+Spacebar shortcut has been used to display a list of message string constants available to add to a message box.

**FIGURE 3-1** Using the Ctrl+Spacebar shortcut to auto-complete long names

A Few Words about Enumeration Types

Enumerations are a special type of constant available in Excel 2000 and higher. They allow you to group a list of related values together using similar, logical friendly names. VBA and the Excel object model make extensive use of enumerations. You can see these in the auto-complete list that VBA provides for the values of many properties. For example if you type

```
Sheet1.PageSetup.PaperSize =
```

into a VBA module, you'll be prompted with a long list of `xlPaperSize` enumeration members that represent the paper sizes available to print on. Figure 3-2 shows this in action.

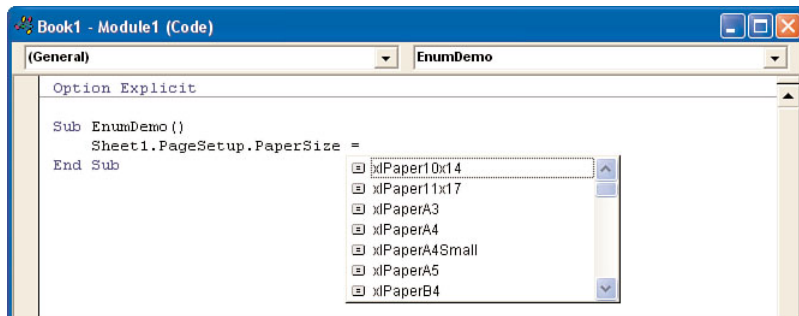


FIGURE 3-2 The Excel paper size enumeration list

These names actually represent numeric constants whose values you can examine by looking them up in the Object Browser, discussed in Chapter 16, "VBA Debugging." Notice the structure of these enumeration names. First, they all begin with a prefix identifying the application they are associated with, in this case "xl," which obviously stands for Excel. Next, the first part of their name is a descriptive term that ties them together visually as belonging to the same enumerated type, in this case "Paper." The last part of each enumeration name is a unique string describing the specific value. For example, `xlPaper11x17` represents 11x17 paper and `xlPaperA4` represents A4 paper. This system for naming enumerated constants is common and is the one we use in this book.

Naming Convention Examples

Naming convention descriptions are difficult to connect to real-world names, so we show some real-world examples of our naming convention in

this section. All these examples are taken directly from commercial-quality applications written by the authors.

Variables

- **gsErrMsg**—A public variable with the data type `String` used to store an error message.
- **mauSettings()**—A module-level array of user-defined type that holds a list of settings.
- **cbrMenu**—A local variable with the data type `CommandBar` that holds a reference to a menu bar.

Constants

- **gbDEBUG_MODE**—A public constant of type `Boolean` that indicates whether the project is in debug mode.
- **msCAPTION_FILE_OPEN**—A module-level constant of data type `String` that holds the caption for a customized file open dialog (`Application.GetOpenFilename` in this instance).
- **lOFFSET_START**—A local constant of data type `Long` holding the point at which we begin offsetting from some `Range` object.

User-Defined Types

The following is a public user-defined type used to store the dimensions and location of an object. It consists of four variables of data type `Double` that store the top, left, width, and height of the object and a variable of data type `Boolean` used to indicate whether the settings have been saved.

```
Public Type DIMENSION_SETTINGS
    bSettingsSaved As Boolean
    dValTop As Double
    dValLeft As Double
    dValHeight As Double
    dValWidth As Double
End Type
```

The variables within a user-defined type definition are called member variables. These can be declared in any order. However, our naming

convention suggests you sort them alphabetically by data type unless there is a strong reason to group them in some other fashion.

Enumeration Types

The following is a module-level enumeration type used to describe various types of days. The “sch” prefix in the name of the enumeration stands for the application name. This enumeration happens to come from an application called Scheduler. DayType in the enumeration name indicates the purpose of the enumeration, and each of the individual enumeration elements has a unique suffix that describes what it means.

```
Private Enum schDayType
    schDayTypeUnscheduled
    schDayTypeProduction
    schDayTypeDownTime
    schDayTypeHoliday
End Enum
```

If you don’t indicate what values you want to give your enumeration member elements, VBA automatically assigns a value of zero to the first element in the list and increments that value by one for each additional element. You can easily override this behavior and assign a different starting point from which VBA will begin incrementing. For example, to make the preceding enumeration list begin with one instead of zero you would do the following:

```
Private Enum schDayType
    schDayTypeUnscheduled = 1
    schDayTypeProduction
    schDayTypeDownTime
    schDayTypeHoliday
End Enum
```

VBA continues to increment by one for each element after the last element for which you’ve specified a value. You can override automatic assignment of values to all your enumeration elements by simply specifying values for all of them.

Figure 3-3 shows one of the primary advantages of using enumeration types. VBA provides you with an auto-complete list of potential values for any variable declared as a specific enumeration type.

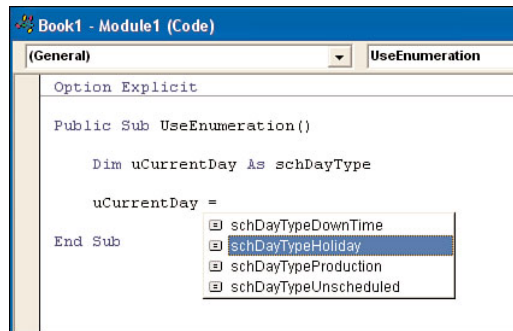


FIGURE 3-3 Even custom enumeration types get a VBA auto-complete listing.

Procedures

Subroutines and functions are grouped under the more general term procedure. Always give your procedures very descriptive names. Once again, you are allowed up to 255 characters for your procedure names. Procedure names are included in the Ctrl+Spacebar auto-complete list, so don't sacrifice a name that makes the purpose of a procedure obvious for one that's simply short.

It is not a common practice to do so, but we find that giving functions a prefix indicating the data type of their return value to be helpful in understanding code. When calling a function, always place open and closed parentheses after the function name to distinguish it from a variable or subroutine name, even if the function takes no arguments. Listing 3-1 shows a well-named Boolean function being used as the test for an If...Then statement.

Listing 3-1 An Example of Naming Conventions for Function Names

```
If bValidatePath("C:\Files") Then
    ' The If...Then block is executed
    ' if the specified path exists.
End If
```

Subroutines should be given a name that describes the task they perform. For example, a subroutine named ShutdownApplication leaves little doubt as to what it does. Functions should be given a name that describes the value they return. A function named sGetUnusedFilename() can reasonably be expected to return an available filename.

The naming convention applied to procedure arguments is exactly the same as the naming convention for procedure-level variables. For example,

the `bValidatePath` function shown in Listing 3-1 would be declared in the following manner:

```
Function bValidatePath(ByVal sPath As String) As Boolean
```

Modules, Classes, and UserForms

In our sample naming convention, the names of standard code modules should be prefixed with an uppercase “M,” class modules with an uppercase “C,” and UserForms with an upper case “F.” This has the advantage of neatly sorting these objects in the VBE Project Window if you don’t care for the folder view, as shown in Figure 3-4.

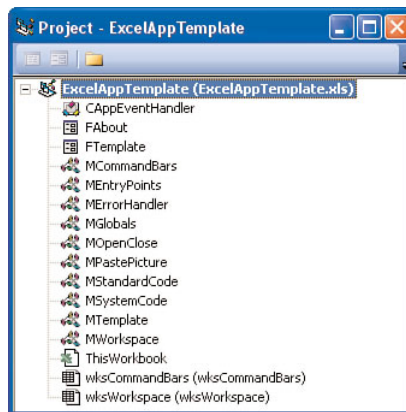


FIGURE 3-4 Class modules, UserForms, and standard modules sorted in the Project window

This convention also makes code that uses classes and UserForm objects much clearer. In the following code sample, for example, this naming convention makes it very clear that you are declaring an object variable of a certain user-defined class type and then creating a new instance of that class:

```
Dim clsMyClass As CMyClass  
Set clsMyClass = New CMyClass
```

In each case, the name on the left is a class *variable* and the object on the right is a *class*.

Worksheets and Chart Sheets

Because the CodeNames of worksheets and chart sheets in your project are treated by VBA as intrinsic object variables that reference those sheets, the CodeNames given to worksheets and chart sheets should follow variable naming conventions. Worksheet CodeNames are prefixed with “wks” to identify them in code as references to Worksheet objects. Similarly, chart sheets are prefixed with “cht” to identify them as references to Excel Chart objects.

For both types of sheets, the prefix should be followed by a descriptive term indicating the sheet’s purpose in the application. In Figure 3-4 for example, `wksCommandBars` is a worksheet that contains a table defining the command bars created by the application. For sheets contained within an add-in or hidden in a workbook and not designed to be seen by the user, the sheet tab name should be identical to the CodeName. For sheets that are visible to the user, the sheet tab name should be a friendly name, and one that you should be prepared for the user to change. Wherever it is reasonably possible to do so, you should rely on sheet CodeNames rather than sheet tab names within your VBA code.

The Visual Basic Project

In Figure 3-4, you’ll notice the Visual Basic Project has been given the same name as the workbook it’s associated with. You should always give your VBProject a name that clearly identifies the application it belongs to. There’s nothing worse than having a group of workbooks open in the VBE with all of them having the same default name “VBAProject.” If you plan on creating references between projects you will be required to give them unique names.

Excel UI Naming Conventions

Excel user interface elements used in the creation of an application should also be named using a consistent and well-defined naming convention. We covered worksheets and chart sheets in a previous section. The three other major categories of Excel UI elements that can be named are shapes, embedded objects, and defined names.

Shapes

The term “shapes” refers to the generic collection that can contain the wide variety of objects you can place on top of a worksheet or chart sheet. Shapes can be broadly divided into three categories: controls, drawing

objects and embedded objects. Shapes should be named similarly to object variables, which is to say they should be given a prefix that identifies what type of object they are followed by a descriptive name indicating what purpose they serve in the application.

Many controls that can be placed on UserForms can be placed on worksheets as well. Worksheets can also host the old Forms toolbar controls, which are similar in appearance to the ActiveX MSForms controls but with their own unique advantages and disadvantages. We'll talk more about these in Chapter 4, "Worksheet Design." Controls placed on worksheets should be named using exactly the same conventions you'd use for controls placed on UserForms.

Worksheets can also host a wide variety of drawing objects (technically known as Shapes) that are not strictly controls, although you can assign macros to all of them. These fall into the same naming convention category as the wide variety of objects that you can use in VBA. It would be very difficult to devise unique prefixes for all of them, so use well-defined prefixes for the most common drawing objects and use a generic prefix for the rest. Here are some sample prefixes for three of the most commonly used drawing objects:

pic	Picture
rec	Rectangle
txt	TextBox (not the ActiveX control)

Embedded Objects

The term "embedded object" is used here to refer to Excel objects such as PivotTables, QueryTables, and ChartObjects, as well as objects created by applications other than Excel. Worksheets can host a variety of embedded objects. Common examples of non-Excel embedded objects would include equations created with the Equation Editor and WordArt drawings. Sample prefixes for embedded objects are as follows:

cht	ChartObject
eqn	Equation
qry	QueryTable
pvt	PivotTable
art	WordArt

Defined Names

Our naming convention for defined names is a bit different than for other program elements. In the case of defined names, the prefix should indicate the broad purpose of the defined name, as opposed to the data type it's expected to hold. This is because non-trivial Excel applications typically have many defined names that are much easier to work with if they are grouped together by purpose within the Define Name dialog. When a worksheet contains dozens or hundreds of defined names, there are significant efficiencies to be gained by having names with related functions grouped together in the defined name list by prefix.

The descriptive name portion of a defined name is used to specify exactly what purpose the name serves within its broader category. The following list shows some examples of purpose-prefixes for defined names.

cht	Chart Data Range
con	Named Constant
err	Error Check
for	Named Formula
inp	Input Range
out	Output Range
ptr	Specific Cell Location
rgn	Region
set	UI Setting
tbl	Table

Exceptions—When Not to Apply the Naming Convention

Two specific situations are commonly encountered in which you want to break the general rule and not apply your naming convention. The first is when you are dealing with elements related to Windows API calls. These elements have been named by Microsoft and the names are well known within the programming community. The Windows API constants, user-defined types, procedure declarations, and procedure arguments should appear in your code exactly as they appear in the Windows API

Reference, which can be viewed on the MSDN Web site at [http://msdn2.microsoft.com/en-us/library/aa383749\(VS.85\).aspx](http://msdn2.microsoft.com/en-us/library/aa383749(VS.85).aspx). Note that this reference is provided in C/C++ format only.

The second situation where you want to avoid applying your own naming conventions is when you use plug-in code from an outside source to perform a specific task. If you modify the names used in this code and refer to those modified names from code elsewhere in your application, you make it very difficult to upgrade the plug-in code when a newer version becomes available.

Best Practices for Application Structure and Organization

Keeping your applications well structured and well organized makes them much easier to maintain and upgrade. In this section, we examine a number of best practices for improving the structure and organization of your application.

Application Structure

The first decision you must make when designing your application structure is how many separate workbooks should it be divided into. The number of workbooks used in an Excel application is driven primarily by two factors: the complexity of the application itself and the limitations imposed by application distribution issues.

Simple applications and those for which you cannot impose a formal installation sequence demand the fewest number of workbooks. Complex applications and those over which you have complete control of the installation process allow division into multiple workbooks or other file types such as DLLs. Chapter 2, “Application Architectures,” discusses the various types of Excel applications and the structure suited to each.

When you have the liberty to divide your application across multiple files, there are a number of good reasons to do so. These include separation of the logical tiers in your application, separation of code from data, separation of user-interface elements from code elements, encapsulating functional elements of the application, and managing change conflicts in a team development environment.

Separation of Logical Tiers

Almost every non-trivial Excel application has three distinct logical tiers or sections (see Figure 3-5):

- **User-interface tier**—Consists of all the code and visible elements required for your application to interact with the user. In an Excel application, the user-interface tier consists of visible elements such as worksheets, charts, command bars, UserForms, and the code required to directly manage those visible elements. The user-interface tier is the only logical tier that contains elements visible to the user.
- **Business logic or application tier**—Completely code-based, this tier performs the core operations the application was designed to accomplish. The business logic tier accepts input from the user-interface tier and returns output to the user-interface tier. For long-running operations, the business logic tier may transmit periodic updates to the user-interface tier in the form of status bar messages or progress bar updates.
- **Data access and storage tier**—Responsible for the storage and retrieval of data required by the application. This can be as simple as reading from and writing data to cells on a local, hidden worksheet or as complex as executing stored procedures in a SQL Server database across a network. The data access and storage tier communicates directly only with the business logic tier.

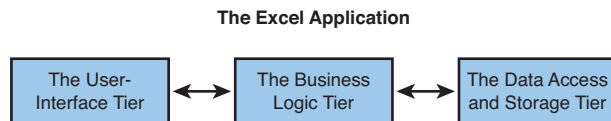


FIGURE 3-5 The relationships among the three tiers of an Excel application

As Figure 3-5 shows, all three tiers are necessary for a complete application, but they must not be inextricably linked. The three tiers of your application should be loosely coupled, such that a significant change in one tier does not require significant changes to the other two. Strongly coupled application tiers inevitably lead to maintenance and upgrade difficulties.

For example, if your data access and storage tier needs to move from using an Access database for storage to using a SQL Server database for storage you want the changes required to be isolated within the data access and storage tier. In a well-designed application, neither of the other two tiers

would be affected in any way by such a change. Ideally, data should be transferred between the business logic tier and the data access and storage tier in the form of user-defined types. These provide the best trade-off between efficiency and loose coupling. Alternatively, ADO Recordset objects can be used, but these introduce subtle linkage issues that it would be better if the business logic layer didn't rely on, such as the order of fields returned from the database.

Similarly, if you need to provide an alternate Web-based presentation interface for your application, loose coupling between the user-interface tier and the business logic tier will make it much easier to accomplish. This is because no implicit assumptions will be built into the business logic tier regarding how the user interface is constructed. Elements that accept data input from the user should be completely self-contained. The business logic tier should pass the user-interface tier the data it requires for initialization as simple data types. The user-interface tier should collect the user input and pass it back to the business logic tier as simple data types, or as a UDT for more complex interfaces. Because the business logic tier should have no intrinsic knowledge of how the user-interface is constructed, referencing controls on a UserForm directly from a business logic tier procedure is expressly forbidden.

Separation of Data/UI from Code

Within the user-interface tier of many Excel applications lie two unique subtiers. These consist of the workbook and sheet elements used to construct the user interface and the code supporting those elements. The concept of separation should be applied rigorously to these subtiers. A workbook-based interface should contain no code, and the UI code that controls a workbook-based interface should reside in an add-in completely separated from the workbook it controls.

The reasoning for this separation is the same as the reasoning described previously for separating the main application tiers: isolating the effects of change. Of all the application tiers, the user-interface tier tends to undergo the most frequent changes. Therefore it's not sufficient to simply isolate user interface changes to the user-interface tier, you should also isolate changes to the visible elements of the user interface from the code that controls the user interface.

We provide real-world examples of application tier separation in the chapters that follow, so don't be concerned if what we've discussed here is not totally obvious to you at this point.

Application Organization for Procedural Programming

Procedural programming is the programming methodology most developers are familiar with. It involves dividing an application into multiple procedures, each of which is designed to perform a specific task within the application. An entire application can be written in procedural fashion, procedural elements can be combined with object oriented elements, or an entire application can be written in object oriented fashion. This section focuses on best practices for procedural programming. We discuss object oriented programming techniques in Chapter 7, “Using Class Modules to Create Objects.”

Organizing Code into Modules by Function/Category

The primary purpose of separating code into modules is to improve the comprehensibility and maintainability of the application. In a procedural application, procedures should be organized into separate code modules in a logical fashion. The best way to do this is to group procedures that perform similar functions into the same code module.

TIP VBA has an undocumented “soft limit” on the maximum size of any single standard code module. A single standard code module should not exceed 64KB as measured by its text file size when exported from the project. (The VBETools utility included on the CD reports module sizes for you automatically.) Your project will not crash immediately upon a single module exceeding this 64KB limit, but consistently exceeding this limit will almost invariably lead to an unstable application.

Functional Decomposition

Functional decomposition refers to the process of breaking your application into separate procedures such that each procedure is responsible for a single task. In theory, you could write many applications as one large, monolithic procedure. However, doing so would make your application extremely difficult to debug and maintain. By using functional decomposition you design your application such that it consists of multiple procedures that are each responsible for a well-defined task that is easy to understand, validate, document, and maintain.

Best Practices for Creating Procedures

A comprehensive set of guidelines for creating good procedures could easily fill a chapter of its own. We cover the most important guidelines in the following list:

- **Encapsulation**—Whenever possible, a procedure should be designed to completely encapsulate the logical operation it performs. Ideally, your procedures should have no linkages to anything outside them. This means, for example, that a properly encapsulated procedure can be copied into a completely different project and work just as well there as it did in the project where it originated. Encapsulation promotes code reuse and simplifies debugging by isolating different logical operations from each other.
- **Elimination of duplicate code**—When writing a non-trivial Excel application, you will frequently discover you are writing code to perform the same operation in multiple places. When this occurs, you should extract this duplicated code and place it in a separate procedure. Doing so reduces the number of places where that operation needs to be validated or modified from many to one. The common procedure can also be optimized in one place and the benefits will be felt throughout your application. All this leads to a significant improvement in code quality. It also serves a second important purpose: making your code more reusable. As you factor common operations into dedicated procedures, you will discover that you can often reuse these procedures in other applications. This type of code forms the basis of a code library that you can use to increase your productivity when writing new applications. The more logical operations you have available as complete, fully tested library procedures, the less time it will take you to develop a new application.
- **Isolation of complex operations**—In many real-world applications you will find that some sections of the business logic are both complex and specific to the application for which they were designed (that is, not reusable). These sections of business logic should be isolated into separate procedures for ease of debugging and maintenance.
- **Procedure size reduction**—Procedures that are overly long are difficult to understand, debug, and maintain, even for the programmer who wrote them. If you discover a procedure containing more than 150 to 200 lines of code, it is probably trying to accomplish multiple goals and therefore should be factored into multiple single-purpose procedures.

- **Limiting the number of procedure arguments**—The more arguments a procedure accepts, the more difficult it will be to understand and the less efficient it will be to execute. In general, you should limit the number of procedure arguments to five or fewer. And don't simply replace procedure arguments with public or module-level variables. If you find yourself requiring more than five procedure arguments it's probably a good sign that your procedure, or your application logic, needs to be redesigned.

General Application Development Best Practices

Each chapter in this book explains the best development practices related specifically to the subject of that chapter. This section covers best development practices common to all application development areas.

Code Commenting

Good code commenting is one of the most important practices in Excel application development. Your code comments should provide a clear and complete description of how your code is organized, how each object and procedure should be used, and what you are trying to accomplish with your code. Comments also provide a means of tracking changes to your code over time, a subject we cover later in this chapter.

Code comments are important to both you and other developers who may need to work on your code. The value of code comments to other developers should be self-evident. What you may not realize until the cruel fist of experience has pounded it into you is that your comments are very important to you as well. It is very common for a developer to write an initial version of an application and then be asked to revise it substantially after a long period of time has passed. You would be surprised at how foreign even your own code looks to you once it has been out of sight and out of mind for a long period of time. Code comments help solve this problem.

Comments should be applied at all three major levels of your application's code: the module level, the procedure level, and individual sections or lines of code. We discuss the types of commenting appropriate to each of these levels in the following sections.

Module-Level Comments

If you used the module naming conventions described previously in this chapter, anyone examining your code will have a rough idea of the purpose of the code contained within each module. You should supplement this with a brief comment at the top of each module that provides a more detailed description of the purpose of the module.

NOTE For the purposes of code commenting, when we use the term “module,” we mean it to include standard modules, class modules, and code modules behind UserForms and document objects like worksheets and the workbook.

A good module-level comment should be located at the top of the module and look something like the example shown in Listing 3-2.

Listing 3-2 A Sample Module-Level Comment

```
'  
' Description:      A brief description of the purpose of the code in  
'                  this module.  
'  
'  
Option Explicit
```

Procedure-Level Comments

Procedure-level comments are typically the most detailed comments in your application. In a procedure-level comment block you describe the purpose of the procedure, usage notes, a detailed list of arguments and their purposes, and a description of expected return values in the case of functions.

Procedure-level comments can also serve a rudimentary change-tracking purpose by providing a place to add dates and descriptions of changes made to the procedure. A good procedure-level comment like the one shown in Listing 3-3 would be placed directly above the first line of the procedure. The procedure-level comment in Listing 3-3 is designed for a function. The only difference between a comment block for a function and a comment block for a subroutine is the subroutine comment block does not contain a Returns section, obviously because subroutines do not return a value.

Listing 3-3 A Sample Procedure-Level Comment

```

.....
' Comments:      Locates the chart to be operated on or asks the
'                user to select a chart if multiple charts are
'                located.
'
' Arguments:      chtChart      Returned by this function. An object
'                               reference to the chart to be
'                               operated on, or Nothing on user
'                               cancel.
'
' Returns:        Boolean        True on success, False on error or
'                               user cancel.
'
' Date           Developer      Action
' -----
' 07/04/02       Rob Bovey      Created
' 10/14/03       Rob Bovey      Error trap for charts with no series
' 11/18/03       Rob Bovey      Error trap for no active workbook
'

```

Internal Comments

Internal comments appear within the body of the code itself. These comments should be used to describe the purpose of any code where the purpose is not self-evident. Internal comments should describe the *intent* of the code rather than the operation of the code. The distinction between intent and operation is not always clear, so Listing 3-4 and Listing 3-5 show two examples of the same code, one with a bad comment and the other with a good comment.

Listing 3-4 Example of a Bad Internal Code Comment

```

' Loop the asInputFiles array.
For lIndex = LBound(asInputFiles) To UBound(asInputFiles)
    '...
Next lIndex

```

The comment in Listing 3-4 is monumentally unhelpful. First of all, it describes only the line of code directly below it, giving you no clue about

the purpose of the loop structure as a whole. Second, the comment is simply an exact written description of that line of code. This information is easy enough to determine by simply looking at the line of code. If you removed the comment, you would not lose any information at all.

Listing 3-5 Example of a Good Internal Code Comment

```
' Import the specified list of input files into the working area  
' of our data sheet.  
For lIndex = LBound(asInputFiles) To UBound(asInputFiles)  
    '...  
Next lIndex
```

In Listing 3-5, we have a comment that adds value to the code. Not only does it describe the intent, rather than the operation of the code, it also explains the entire loop structure. After reading this comment you know what you're looking at as you delve into the code within the loop.

As with most rules, there are exceptions to the internal comment guidelines specified previously. The most important exception concerns comments used to clarify control structures. `If...Then` statements and `Do...While` loops can make code difficult to understand as they become wider, because you can no longer see the entire control structure in a single code window. At that point, it becomes difficult to remember what the applicable control expression was. For example, when evaluating a lengthy procedure we have often found ourselves looking at something like the code snippet shown in Listing 3-6.

Listing 3-6 Inscrutable Control Structures

```
End If  
  
lNumInputFiles = lNumInputFiles - 1  
  
Loop  
  
End If
```

In Listing 3-6, what are the logical tests being made by the two `If...Then` statements and what expression controls the `Do...While` loop? Once these structures have been filled with a substantial amount of code, you simply can't

tell without scrolling back and forth within the procedure, because the entire block is no longer visible within a single code window. This problem can be alleviated easily by using the end of control block commenting style shown in Listing 3-7.

Listing 3-7 Understandable Control Structures

```
End If      ' If bContentsValid Then

lNumInputFiles = lNumInputFiles - 1

Loop              ' Do While lNumInputFiles > 0

End If          ' If bInputFilesFound Then
```

The comments in Listing 3-7, although they simply restate the code at the top of each control structure, make it completely obvious what you are looking at. These types of comments should be used anywhere you have a control structure within your code that is too large to fit completely into one code window.

Avoiding the Worst Code Commenting Mistake

It may seem obvious, but the most frequent and damaging mistake related to code commenting is not keeping the comments updated as you modify the code. We have frequently seen projects that appeared at first glance to implement good code commenting practices, but upon closer examination discovered the comments were created for some ancient version of the project and now bore almost no relationship to the current code.

When attempting to understand a project, bad comments are worse than no comments at all because bad comments are actively misleading. Always keep your comments current. Old comments can either be deleted or retained as a series of change tracking records. We recommend removing obsolete in-line comments, or they will quickly clutter your code, making it difficult to understand simply due to the number of lines of inapplicable comments that accumulate. Use procedure-level comments as a change tracking mechanism where necessary.

Code Readability

Code readability is a function of how your code is physically arranged. Good visual layout of code allows you to infer a significant amount of information about the logical structure of the program. This is a key point. Code layout makes not one bit of difference to the computer. Its sole purpose is to assist humans in understanding the code. Like naming conventions, the consistent use of good code layout conventions makes your code self-documenting. The primary tool of code layout is white space. White space includes space characters, tabs, and blank lines. In the following paragraphs we discuss the most important methods of using white space to create a well-designed code layout.

Group related code elements together and separate unrelated code elements with blank lines. Sections of code separated by blank lines within a procedure can be thought of as serving a similar function to paragraphs within the chapters of a book. They help you determine what things belong together. Listing 3-8 shows an example of how blank lines can improve code readability. Even without the code comments, it would be obvious which lines of code were related.

Listing 3-8 Using Blank Lines to Group Related Sections of Code

```
' Reset Application properties.
Application.ScreenUpdating = True
Application.DisplayAlerts = True
Application.EnableEvents = True
Application.StatusBar = False
Application.Caption = Empty
Application.EnableCancelKey = xlInterrupt
Application.Cursor = xlDefault

' Delete all custom CommandBars
For Each cbrBar In Application.CommandBars
    If Not cbrBar.BuiltIn Then
        cbrBar.Delete
    Else
        cbrBar.Enabled = True
    End If
Next cbrBar

' Reset the Worksheet Menu bar.
With Application.CommandBars(1)
    .Reset
```



```
.Enabled = True
.Visible = True
End With
```

Within a related section of code, horizontal alignment is used to indicate which lines of code belong together. Indentation is used to show the logical structure of the code. In Listing 3-9 we show a single section from Listing 3-8 where alignment and indentation have been used to good effect. You can look at this section of code and understand immediately which elements go together as well as deduce the logical flow of the code's execution.

Listing 3-9 Proper Use of Alignment and Indentation

```
' Delete all custom CommandBars
For Each cbrBar In Application.CommandBars
    If Not cbrBar.BuiltIn Then
        cbrBar.Delete
    Else
        cbrBar.Enabled = True
    End If
Next cbrBar
```

Line continuation can be used to make complex expressions and long declarations more readable. Keep in mind that breaking code into continued lines solely for the purpose of making the entire line visible without scrolling is not necessarily a good practice and can often make code more confusing. Listing 3-10 shows examples of judicious use of line continuation.

Listing 3-10 Judicious Use of Line Continuation

```
' Complex expressions are easier to understand
' when properly continued
If (uData.lMaxLocationLevel > 1) Or _
    uData.bHasClientSubsets Or _
    (uData.uDemandType = bcDemandTypeCalculate) Then

End If

' Line continuations make long API declarations easier to read.
Declare Function SHGetSpecialFolderPath Lib "Shell32.dll" _
    (ByVal hwndOwner As Long, _
    ByRef szBuffer As String, _
```



```
ByVal lFolder As Long, _  
ByVal bCreate As Long) As Long
```

General VBA Programming Best Practices

In this section, we examine a number of VBA programming best practices that help you write code that is more robust and easier to maintain and update.

Use of Module Directives

Module directives are statements at the top of a code module that instruct VBA how to treat the code within that code module. Although these directives are not required, you should always use at least one or two of them, as explained in the following list:

- **Option Explicit**—Always use the `Option Explicit` statement in every module. The importance of this practice cannot be overstated. Without `Option Explicit`, any typographical error you make results in VBA automatically creating a new Variant variable. This type of error is insidious because it may not cause an immediate runtime error, but it will almost certainly cause your application to eventually return incorrect results. Errors caused by the lack of an `Option Explicit` statement often pass without notice until your application is distributed, and they are difficult to debug under any circumstances.

The `Option Explicit` statement forces you to explicitly declare all the variables you use. `Option Explicit` causes VBA to throw a compile-time error (initiated by selecting *Debug > Compile* from the VBE menu) whenever an unrecognized identifier name is encountered. This makes it easy to discover and correct typographical errors. You can ensure that `Option Explicit` is automatically placed at the top of every module you create by choosing *Tools > Options > Editor* from the VBE menu and checking the *Require Variable Declaration* check box. This setting is strongly recommended.

- **Option Private Module**—The `Option Private Module` statement makes all procedures within the module where it is used unavailable from the Excel user interface or from other Excel projects. Use this statement to hide procedures that should not be called from outside your application.

TIP The `Application.Run` method can circumvent the `Option Private Module` statement and run private procedures in modules where this statement has been used. Also, if a user knows the exact name of your procedure and your procedure does not require any arguments, the user can type the name of your procedure into the Macro dialog and run it manually. These scenarios can be made much less likely by protecting your project so that your private procedure names are not visible in the Object Browser.

- **Option Base 1**—The `Option Base 1` statement causes all array variables whose lower bound has not been specified to have a lower bound of 1. Do not use the `Option Base 1` statement. Instead, always specify both the upper and lower bounds of every array variable you use. A procedure created in a module that uses `Option Base 1` may malfunction if copied to a module in which this statement isn't used. This behavior inhibits one of the most important procedure design goals, that of reusability.
- **Option Compare Text**—The `Option Compare Text` statement forces all string comparisons within the module where it is used to be text-based rather than binary. In a text-based string comparison, upper- and lowercase versions of the same character are treated as identical, whereas in a binary comparison they are different. The `Option Compare Text` statement should be avoided for the same reason `Option Base 1` should be avoided. It makes procedures behave differently when placed in modules with the statement versus modules without it. Text-based comparisons are also much more computationally expensive than binary comparisons, so `Option Compare Text` slows down all string comparison operations in the module where it's located. Most Excel and VBA string comparison functions provide an argument you can use to specify binary or text-based comparison. It's much better to use these arguments to provide text-based comparisons only where you need them. There are some rare cases where `Option Compare Text` is required. The most frequent case occurs when you need to do case-insensitive string comparisons with the VBA `Like` operator. The only way to get the `Like` operator to perform in a case-insensitive manner is to use the `Option Compare Text` statement. In this case, you should isolate the procedures that require this statement in a separate code module so that other procedures that don't require this option aren't adversely affected. Be sure to document why you have done this in a module-level comment.

Best Practices for Variables and Constants

Variables and constants are the most fundamental building blocks of an application. We can become so used to them that we forget they are active pieces of our application that must be used properly or the quality of our application will suffer. In this section, we cover a number of best practices to follow when using variables and constants.

Avoid Reusing Variables Each variable declared in your program should serve one purpose only. Using the same variable for multiple purposes saves you only one variable declaration line but introduces massive potential for confusion within your program. If you are trying to determine how a procedure works and you have figured out what a certain variable does in a certain place, you will naturally assume the variable serves the same purpose the next time you see it. If this is not the case, the code logic will become difficult to understand.

Avoid the Variant Data Type Avoid the use of the Variant data type whenever possible. Unfortunately, VBA is not a strongly typed programming language. Therefore, you can simply declare variables without specifying their data type and VBA will create these variables as Variants. The main reasons not to use Variants are

- **Variants are inefficient**—This is because internally, a Variant is a complex structure designed to hold any data type in the VBA programming language. Variant values cannot be accessed and modified directly as can fundamental data types such as Long and Double. Instead, VBA must use a series of complex Windows API calls behind the scenes whenever it needs to perform any operation on a Variant.
- **Data stored in a variant can behave unexpectedly**—Because Variants are designed to hold any type of data, the data type that goes into a Variant is not necessarily the data type that will come out of it. When accessing the data in a Variant, VBA attempts to coerce the data into whatever data type it thinks makes the most sense in the context of the operation. If you must use Variants, convert them explicitly to the data type you want when using their values using one of the VBA functions provided for this purpose (CStr, CLng, CDate, and so on).

Variants do have one valuable characteristic that you can take advantage of, which is if you assign a multicell range to them using the

Range.Value property they automatically become a two-dimensional array containing all the values in that range. When you need to manipulate large numbers of values contained in a range of cells, it is faster to dump them into a Variant array, loop the Variant array, and perform your operations on it, and then dump the Variant array back into the range of cells. Listing 3-11 shows an example of how to do this.

Listing 3-11 Using a Variant Array to Manipulate Range Values

```
Sub UseVariantArray()  
  
    Dim lRow As Long  
    Dim lCol As Long  
    Dim vaArray As Variant  
  
    vaArray = Sheet1.Range("A1:E5").Value  
  
    For lRow = LBound(vaArray, 1) To UBound(vaArray, 1)  
        For lCol = LBound(vaArray, 2) To UBound(vaArray, 2)  
            vaArray(lRow, lCol) = vaArray(lRow, lCol) * 2  
        Next lCol  
    Next lRow  
  
    Sheet1.Range("A1:E5").Value = vaArray  
  
End Sub
```

Beware of Evil Type Coercion Evil Type Coercion (ETC) is another symptom that results from VBA not being a strongly typed programming language. ETC occurs when VBA automatically converts one data type to another data type in a way you did not intend. The most frequent examples are Strings that hold numbers being converted to Integers and Booleans being converted to their String equivalents. Don't mix variables of different data types in your VBA expressions without using the explicit casting functions (CStr, CLng, CDate, and so on) to tell VBA exactly how you want those variables to be treated.

Avoid the As New Declaration Syntax Never declare object variables using the As New syntax. For example, the following form of an object variable declaration should never be used:


```
Dim rsData As New ADODB.Recordset
```

If VBA encounters a line of code that uses this variable and the variable has not been initialized, VBA automatically creates a new instance of the variable. This is *never* the behavior you want. Good programming practice implies that the programmer should maintain complete control over the creation of all the objects used in the program. If VBA encounters an uninitialized object variable in your code, it is almost certainly the result of a bug, and you want to be notified about it immediately. Therefore, the proper way to declare and initialize the object variable shown previously is the following:

```
Dim rsData As ADODB.Recordset  
Set rsData = New ADODB.Recordset
```

Using this style of declaration and initialization, if the object variable is destroyed somewhere in your procedure and you inadvertently reference it again after that point, VBA immediately throws the runtime error “Object variable or With block variable not set,” notifying you of the problem.

Always Fully Qualify Object Names Always fully qualify object names used in variable declarations and code with their class name prefix. This is because many object libraries share the same object names. If you simply declare a variable with an object name alone and there are multiple object libraries with that object name being referenced by your application, VBA creates a variable from the first library in the *Tools > References* list where it finds the object name you used. This is often not what you want.

UserForm controls present the most common situation where problems result from object variable declarations that aren’t fully qualified. For example, if you wanted to declare an object variable to reference a TextBox control on your UserForm, you might be inclined to do the following:

```
Dim txtBox As TextBox  
Set txtBox = Me.TextBox1
```

Unfortunately, as soon as VBA attempted to execute the second line of code, a “Type mismatch” error would be generated. This is because the Excel object library contains a TextBox object that is different from the object you are trying to reference and the Excel object library comes before the MSForms object library in the *Tools > References* list. The correct way to write this code is as follows:

```
Dim txtBox As MSForms.TextBox  
Set txtBox = Me.TextBox1
```


Never Hard-Code Array Bounds When you are looping the contents of an array variable, never hard-code the array bounds in the loop. Use the `LBound` and `UBound` functions instead, as shown in the Listing 3-12.

Listing 3-12 The Correct Way to Loop an Array

```
Dim lIndex As Long
Dim allListItems(1 To 10) As Long

' Load the array here.

For lIndex = LBound(allListItems) To UBound(allListItems)
    ' Do something with each value.
Next lIndex
```

The reason for this is because array bounds frequently change over the course of creating and maintaining an application. If you hard-code the array bounds 1 and 10 in the loop shown in Listing 3-12, you will have to remember to update the loop any time the bounds of the `allListItems` array change. Failure to do so is a frequent source of errors. By using `LBound` and `UBound` you make the loop self-adjusting.

Always Specify the Loop Counter after a Next Statement Listing 3-12 demonstrates another good coding practice. You should always specify the loop counter variable after a `Next` statement. Even though this is not strictly required by VBA, doing so makes your code much easier to understand, especially if the distance between the `For` and `Next` statements is long.

Make Use of Constants Constants are useful programming elements. They serve the following purposes in your code, among others:

- Constants eliminate “magic numbers,” replacing them with recognizable names. For example, in the following line of code, what does the number 50 mean?

```
If lIndex < 50 Then
```

There is no way of knowing unless you wrote the code and you still remember what 50 represents. If instead you saw the following, you

would have a very good idea of what the `If...Then` test was looking for:

```
Const lMAX_NUM_INPUT_FILES As Long = 50
```

```
' More code here.
```

```
If lIndex < lMAX_NUM_INPUT_FILES Then
```

If you need to know the value of a constant at design time, you can simply right-click over the constant name in the VBE and choose *Definition* from the shortcut menu. You will be brought directly to the line where the constant is defined. In break mode at runtime it's even easier. Simply hover your mouse over the constant and a ToolTip window containing its value appears.

- Constants improve coding efficiency and avoid errors by eliminating duplicate data. In the preceding example, assume you reference the maximum number of input files in several places throughout your program. At some point you may need to upgrade your program to handle more files. If you hard-coded the maximum number of input files everywhere you've needed to use it, you will have to locate all these places and change the number in each one. If you used a constant, all you need to do is modify the value of the single constant declaration and the new value automatically is used wherever the constant has been used in your code. This situation is a frequent source of errors that can be eliminated by simply using constants instead of hard-coded numbers.

Public variables are dangerous. They can be modified anywhere in your application without warning, making their values unpredictable. They also work against one of the most important programming principles—encapsulation. Always create variables with the minimum scope possible. Begin by creating all your variables with local (procedure level) scope and only widen the scope of a variable when it is absolutely necessary.

As with most rules, there are a few cases where the variable scope rule should be broken because the use of public variables is useful and/or necessary:

- When data must be passed deep into the call stack before it is used. For example, if procedure A reads some data and then passes that data to procedure B, which passes it to procedure C, which passes it to procedure D where the data is finally used, a good case can be

made that the data should be passed directly from procedure A to procedure D by way of a public variable.

- Certain inherently public classes, such as an application-level event handling class, require a public object variable so they never go out of scope while your application is running.

Early Binding Versus Late Binding The distinction between early binding and late binding is widely misunderstood and often confused with how an object is created. The **only** thing that affects whether an object is early bound or late bound is how the object variable holding the reference to the object was declared. Variables declared as a specific object data type are always early bound. Variables declared with the Object or Variant data type are always late bound. Listing 3-13 shows an example of a late bound reference, while Listing 3-14 shows an example of an early bound reference.

Listing 3-13 A Late Bound Reference to an ADO Connection Object

```
Dim objConnection As Object

' It doesn't matter how you create the object, it's still
' late bound due to the As Object variable declaration.
Set objConnection = New ADODB.Connection
Set objConnection = CreateObject("ADODB.Connection")
```

Listing 3-14 An Early Bound Reference to an ADO Connection Object

```
Dim cnConnection As ADODB.Connection

' It doesn't matter how you create the object, it's still early
' bound due to the data type used in the variable declaration.
Set cnConnection = New ADODB.Connection
Set cnConnection = CreateObject("ADODB.Connection")
```

Note that to use early binding with objects outside the Excel object model you must set a reference to the appropriate object library using the *Tools > References* menu in the Visual Basic Editor. For example, to create early bound variables referencing ADO objects, you must set a reference to the Microsoft ActiveX Data Objects 2.X Library, where X is the version of ADO

you intend to use. You should use early bound object variables wherever possible. Early bound object variables provide the following advantages over late bound variables:

- **Improved performance**—When you use an object variable whose data type is known to VBA at compile time, VBA can look up the memory locations of all property and method calls you use with this object and store them with your code. At runtime, when VBA encounters one of these early bound property or method calls, it simply executes the code located at the stored location. (This is a bit of an oversimplification. What VBA actually stores is a numeric offset to the code to be executed from a known starting point in memory, which is the beginning of a structure called the object's Vtable.)

When you use a late bound object variable, VBA has no way of knowing in advance what type of object the variable will contain. Therefore, it cannot optimize any property or method calls at compile time. This means that each time VBA encounters a late bound property or method call at runtime, it must query the variable to determine what kind of object it holds, look up the name of the property or method being executed to determine where in memory it is located, and then execute the code located at that memory address. This process is significantly slower than an early bound call.

- **Strict type checking**—In the late bound example shown previously in Listing 3-13, if you accidentally set your object variable to reference an ADO Command object instead of a Connection object, VBA would not complain. You would only discover you had a problem later in your code when you tried to use a method or property not supported by the Command object. With early binding, VBA immediately detects that you are trying to assign the wrong type of object reference to your object variable and notifies you with a “Type mismatch” error. Incorrect property and method calls can be detected even earlier, before the code is ever run. VBA attempts to look up the name of the property or method being called from within the appropriate object library at compile time and throws a compile-time error if the name cannot be located.
- **IntelliSense availability**—Early bound object variables make for much easier programming as well. Since VBA knows exactly what type of object a variable represents, it can parse the appropriate object library and provide a drop-down list of all available properties

and methods for the object as soon as you type a dot operator after the variable's name.

As you might expect, there are some cases where you need to use late binding rather than early binding. The two most common reasons for using late binding instead of early binding are

- When a newer version of an application's object library has broken compatibility with an earlier version.

This is an all too common situation. If you set a reference to the later version of the application's object library in your application and then attempt to run it on a computer that has the earlier version, you will get an immediate compile-time error "Can't find project or library," and the reference on the target machine will be prefixed with "MISSING." The worst problem with this error is that the line of code flagged as being the source of the error often has nothing to do with the object library actually causing the problem.

If you need to use objects from an application that exhibits this problem and you can't develop against the earliest possible version of the application that you might encounter, you need to use late binding for all variables referencing objects from the application. If you are creating new objects, you also need to use the `CreateObject` function with the version independent `ProgID` of the object you want to create, rather than the `= New ObjectName` syntax.

- When you want to use an application that you cannot be sure will exist on the user's computer and that you cannot install yourself.

In this case, you need to use late binding to avoid the compile-time error that would immediately result from attempting to run an application that referenced an object library that did not exist on the user's computer. Your application can then check for the existence of the object library in question and exit gracefully if that library is not installed on the user's computer.

TIP Even if you will eventually use late binding in your code, early binding offers such a great increase in productivity while coding that you should write and test the application using early binding. Convert your code to late binding only for the final round of testing and distribution.

Defensive Coding

Defensive coding refers to various programming practices designed to help you prevent errors rather than having to correct them after they occur.

Write Your Application in the Earliest Version of Excel That You Expect It to Run In Although the Microsoft Excel team has done a better job than most of maintaining backward compatibility with earlier versions of Excel, there are many subtle differences between the versions. If you do not write your application in the earliest version of Excel that you expect it to run in you can easily write an application that will not run on earlier versions of Excel because some feature you used did not exist in those versions.

The solution to this problem is to always develop your applications in the earliest version of Excel that you expect them to run in. This may force you to do one of the following in order of worst to best practice:

- Maintain multiple versions of Excel on one computer (not recommended).
- Maintain separate computers for each version of Excel.
- Use virtualization software such as VMWare or Virtual PC to maintain as many separate development environments as you need on a single computer.

Developing in the earliest version of Excel you expect to run in is essential. If you develop an application in Excel 2003 and then discover it doesn't run properly in Excel 2000, you will have much debugging and rewriting ahead. You will save considerable time and stress by simply developing the application using Excel 2000 to begin with.

Explicitly Use ByRef or ByVal If a procedure takes arguments, there are two ways to declare those arguments: ByRef or ByVal.

- **ByRef**—This convention means you are passing the memory address of the variable rather than the value of the variable. If the called procedure modifies a ByRef argument, the modification will be visible in the calling procedure.
- **ByVal**—This convention means you are passing a value to the procedure. A procedure can make changes to a ByVal argument but these changes will not be visible to the calling procedure. In fact, a procedure can use ByVal arguments exactly as if they were locally declared variables.

Always explicitly declare your procedure arguments as ByRef or ByVal. If you do not specify this, all arguments are created ByRef by default. You should declare procedure arguments ByVal unless you have a specific need for the calling procedure to see changes made to the arguments. Declaring arguments ByVal prevents changes made to those arguments from being propagated back to the calling procedure.

The only exceptions are when you are passing large strings (very large strings), which are far more efficiently passed ByRef, or when your procedure argument is of a data type, such as an array, that cannot be passed ByVal. Be aware that declaring procedure arguments ByVal does leave you more exposed to Evil Type Coercion. A ByRef procedure argument **must** be passed exactly the same data type as it is declared to accept; otherwise a compile-time error will result. By contrast, VBA attempts to coerce a value passed to a ByVal procedure argument into a compatible data type.

Explicitly Call the Default Property of an Object With the possible exception of the Item property of a Collection object, it's never a good idea to implicitly invoke the default property of an object by simply using the object's name in an expression. Listing 3-15 shows the right way and the wrong way of accessing the default property of an object using an MSForms.TextBox control for demonstration purposes (the Text property is the default property of an MSForms.TextBox control).

Listing 3-15 Default Properties

```
' The right way.  
txtUsername.Text = "My Name"  
  
' The wrong way  
txtUsername = "My Name"
```

By avoiding the implicit use of default properties, you make your code much more readable and protect yourself from errors if the default behavior of the object changes in some future version of Excel or VBA.

Validate Arguments before Using Them in Procedures If your procedure accepts input arguments that must have certain properties to be valid—for example, if they must be within a specific range of values—verify that the values passed to those arguments are valid before attempting to use them in your procedure. The idea is to catch erroneous input as soon as possible so you can generate a meaningful error message and simplify your debugging.

Wherever possible, create a test harness to validate the behavior of your procedure. A test harness is a wrapper procedure that can call the procedure being tested multiple times, passing it a wide range of arguments, and test the result to be sure it is correct. We discuss test harnesses in detail in Chapter 16.

Use Guard Counters to Protect Against Infinite Loops Program your loops to automatically handle infinite loop conditions. One of the most common mistakes made when using `Do...While` or `While...Wend` loops is to create a situation where the loop control condition is never satisfied. This causes the loop to run forever (or until you can force your code to break by pressing Ctrl+Break if you are lucky, or the Windows Task Manager to shut down your application if you are not). Always add a counter that automatically bails out when the number of loops executed is known to be more than the highest number that should ever occur in practice. Listing 3-16 shows a `Do...While` loop with an infinite loop guard structure.

Listing 3-16 Guard Against Infinite Loops

```
Dim bContinueLoop As Boolean
Dim lCount As Long

bContinueLoop = True
lCount = 1

Do

    ' The code that goes here should set the
    ' bContinueLoop variable to False once the
    ' loop has achieved its purpose.

    ' This infinite loop guard exits the loop
    ' with an error after 10000 iterations.
    lCount = lCount + 1
    If lCount > 10000 Then Err.Raise _
        Number:=9999, Description:="Infinite Loop Error!"

Loop While bContinueLoop
```

The only purpose of the `lCount` variable within the loop is to force the loop to exit if the code within the loop fails to set the control variable to exit

within 10,000 iterations (the appropriate number would depend on the particular situation). This type of construct adds very little overhead to your loop, but if performance is a significant concern, use the infinite loop guard until you are sure all the code within the loop is functioning properly; then delete it or comment it out.

Use *Debug > Compile Early and Often* Never let your code stray more than a few changes away from being able to run a flawless *Debug > Compile*. Failing to adhere to this practice will lead to long, inefficient debugging sessions.

Use *CodeNames to Reference Sheet Objects* Always reference worksheets and chart sheets in your application by their `CodeName`. Depending on sheet tab names to identify sheets is risky because you or your users may change these tab names, breaking any code that uses them.

Validate the Data Types of Selections If you write a procedure designed to operate on a specific type of object the user has selected, always check the object type of the selection using either the `TypeName` function or the `If TypeOf...Is` construct. For example, if you need to operate on a range selected by the user, ensure the selection really is a `Range` object before continuing, as shown in Listing 3-17.

Listing 3-17 Verify That the Selection Is the Correct Object Type

```
' Code designed to operate on a range.
If TypeOf Application.Selection Is Excel.Range Then
    ' OK, it's a Range object.
    ' Continue code execution.
Else
    ' Error, it's not a Range object.
    MsgBox "Please select a range.", vbCritical, "Error!"
End If
```

Change Control

Change control, also known as version control, at the most basic level involves two practices: maintaining a set of prior versions of your application that you can use to recover from various programming or technical errors and documenting changes made to your application over time.

Saving Versions

When most professional programmers talk about version control, they mean the use of dedicated version control software, like Microsoft Visual Source Safe. However, this type of software is expensive, has a steep learning curve, and doesn't integrate well with applications built in Excel. This is because Excel doesn't store its modules natively as separate text files. The version control method we suggest here is quick, simple, requires no special software, and delivers the most crucial benefits of a traditional version control system.

The most important objective of a version control system is to allow you to recover an earlier version of your project if you have encountered some significant problem with the version you are currently working on. If a significant code modification has gone terribly wrong or you suddenly find yourself with a corrupt file, you will be in a very difficult position if you do not have a recent backup to help you recover.

A simple version control system that can save you from these problems would be implemented in a fashion similar to the following. First create a folder named Backup as a subfolder to the folder in which your project is stored. Each time you prepare to make a significant addition or modification to your project, or once a day at minimum, use a file compression utility such as WinZip to zip all the files in your project folder into a file with the following name format: **Backup_YYYYMMDDHH.zip**, where Y stands for year, M stands for month, D stands for day, and H stands for hour. This naming format gives your backup file a unique name that will sort in correct sequential order when viewed in Windows Explorer. Move this file into your Backup folder and continue working.

If you encounter a problem, you can recover your project from the most recent backup. You will obviously lose some work, but if you save backup versions diligently you can minimize the loss. Each time you are sure you have a fully tested build of your project you can delete most of the intermediate files from your Backup folder. It is advisable to retain at least weekly backups throughout the life of a project.

Documenting Changes with Comments

When you are maintaining code, if you make a significant change to the logic of a procedure you should also make a note with a brief description of the change, the date it was made and your name in the procedure-level comment block (refer to Listing 3-3). All non-trivial modifications to your code should be noted with an internal comment that includes the

date the change was made and the name of the developer who made the change if multiple developers are working on the application.

Summary

Whether you use the naming convention proposed here or create your own, use a naming convention consistently across all your applications and over time. It makes your code self-documenting and easy to follow. Code the separate logical tiers of your application as independent entities. This prevents changes in one logical tier from forcing you to rebuild much of your application. Comment your code liberally at all levels. When trying to understand the purpose of a section of code, it's a lot easier if that purpose is explained by a code comment than if you have to figure it out yourself. Following these and all the other best practices presented in this chapter will result in robust, understandable, maintainable applications.

This page intentionally left blank

WORKSHEET DESIGN

A tremendous amount of Excel user interface design can and should be accomplished using the built-in features of Excel alone, with no VBA required. One of the guiding principles of Excel development is “let Excel be Excel.” Don’t try to reinvent the wheel. Excel provides a wide variety of prepackaged, performance-optimized features you can use to build your application’s user interface. In this chapter we examine how you can produce a fully functional user interface with just the features Excel provides for this purpose.

There are two fundamental sections of an Excel worksheet user interface: those designed to be visible to the user and through which the user operates your application, and those designed to be hidden from the user and used only by your application to perform the tasks required of it. We cover each of these sections in more detail in this chapter.

Principles of Good Worksheet UI Design

The following list provides some design guidelines that apply to all worksheet user interfaces:

1. Use formatting to create visual contrast among cells designed to serve different purposes, input cells versus formula cells for example, as well as visual separation among different sections of your user interface.
2. Use consistent formatting based on the purpose of each cell. For example, don’t format input cells with a white background in one area and a green background in another.
3. Don’t use garish colors. Your choice of formatting should not distract from the task at hand. Do try to use colors with enough contrast that people with color-impaired vision will be able to recognize the different sections of your user interface.

4. Create a logical, well-structured flow through your user interface. Your user interface should flow from left to right and then top to bottom within a worksheet and from left to right among multiple worksheets.
5. Make your user interface as uncluttered as possible. Provide sufficient space between and around the various sections of your user interface. Leave an empty row at the top and an empty column at the far left to separate your worksheet user interface from the Excel container (this is in addition to the program rows and columns we cover in the next section).
6. Make it obvious to users what they are supposed to do each time they are required to perform some action. Techniques for doing this include the use of cell comments, validation lists, validation input messages, default values, good descriptive field names, and so on.
7. Use dynamic input verification techniques to provide feedback as quickly as possible if the user has done something wrong. Waiting until the user has completed the entire form before pointing out data entry errors should be viewed as a last resort, to be used only when there are no good alternatives.
8. Don't create an environment that potentially allows the user to make catastrophic mistakes. Protect all user interface worksheets, leaving only cells that require data entry unlocked. This prevents critical formulas from being accidentally overwritten.
9. Don't allow the user to get lost. Restrict the area of the worksheet within which the user can navigate to just the working area of your user interface.

Program Rows and Columns: The Fundamental UI Design Technique

When you design a user interface on an Excel worksheet, one of the first things you should do is leave row 1 and column A empty. This section of the worksheet will be hidden from the user and will allow your application to perform many tasks associated with an advanced Excel UI, including error checking, storing validation lists, and calculating intermediate values. In complex worksheet user interfaces it is not uncommon to have several initial rows and/or columns used as hidden work areas. These are called **program rows** and **program columns**.

An Excel worksheet user interface is typically laid out in a table format: left to right, top to bottom. Implementing design principle #6 described previously is most easily accomplished if you have a hidden area you can use to automatically examine each of the user's entries and determine whether they meet all the criteria that are enforceable using worksheet-based constructs. The result of these tests can then be used by conditional formatting and/or VBA-based validation to signal users when they have entered data incorrectly.

In the simple time sheet example shown in Figure 4-1, the user completes the first three columns of the table. The last column of the table is calculated by the worksheet. The first column of the worksheet itself is designed to be a hidden column. It performs a simple validation check on each row of the time sheet table. It counts the number of entries made by the user in each row and returns True if the number of entries is incorrect (which is to say the user has not completed all of the required entries for that row).

A6		=IF(COUNTA(C6:E6)=0,FALSE,COUNTA(C6:E6)<>3)				
	A	B	C	D	E	F
1						
2						
3	errHasError		Activity	Start Time	Stop Time	Total Hours
4	FALSE		General Programming	8:00 AM	12:00 PM	4:00
5	FALSE		Phone Conference	1:00 PM	2:00 PM	1:00
6	TRUE		Technical Support	2:00 PM		
7	FALSE					
8	FALSE					
9	FALSE					

FIGURE 4-1 An example of hidden column data validation

Here there are only two possible valid conditions. Either a row has not yet been used, and therefore has zero entries, or a row has been completely filled out, in which case there will be three entries. Any other condition is an error. Notice the error checking formula for row 6 indicates there is a data entry error in that row. This is because the user has not yet entered a Stop Time. The user may very well eliminate this error by entering a Stop Time after he completes this task. If he doesn't, it is a simple matter for your application to examine the validation range in column A and determine there is an error.

Defined Names

Defined names are an integral part of worksheet user interface design. Defined names are a superset of the more commonly understood named

range feature. Defined names include named constants, named ranges, and named formulas. Each type of defined name serves an important purpose, and all non-trivial Excel worksheet user interfaces use some or all of the defined name types. The naming conventions used for the defined names in this chapter are described in Chapter 3, “Excel and VBA Development Best Practices.”

Named Constants

A defined name can refer to a constant value. For example, the `setHiddenCols` defined constant shown in Figure 4-2 refers to the value 1.

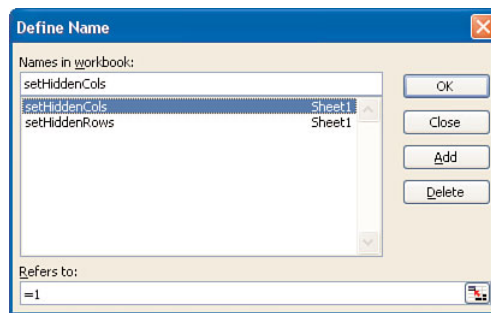


FIGURE 4-2 A sample named constant

This name illustrates a typical use of defined constants: storing settings that will be made to a user interface worksheet. In this case it indicates the number of initial columns that will be hidden. Named constants can also serve all of the same purposes on a worksheet that VBA constants serve in a VBA program, as discussed in Chapter 3.

Other common uses of named constants are worksheet identification, workbook identification, and version identification. It is common to have several broad classes of worksheet in your Excel application, such as input worksheets, analysis worksheets, reporting worksheets, and so on. You can use named constants to specify the type of each worksheet so that your application code can determine the type of the active worksheet and respond correctly, displaying the appropriate worksheet type-specific toolbar, for example.

Each user-interface workbook you create should have a unique named constant that identifies it as belonging to your application. The add-in for your application can then use this constant to determine whether the

currently active workbook belongs to it. You should also include a version constant so you can pinpoint exactly what version of your application a given workbook belongs to. This becomes important when you upgrade the application such that prior version user interface workbooks must be updated or handled differently than current version user interface workbooks in some way.

Named Ranges

Named ranges allow you to reference a location on a worksheet with a friendly name that conveys information about the location rather than using a range address that cannot be interpreted without following it back to the cell or cells it refers to. As we see in the following example, named ranges also allow you to accomplish things you cannot accomplish with directly entered cell addresses.

Everyone reading this book should be familiar with fixed named ranges, those referring to a fixed cell or group of cells on a worksheet. This section concentrates on the less well-understood topic of relative named ranges. A relative named range is called relative because the location it references is determined relative to the cell in which the name is used. Relative named ranges are defined in such a way that the cell or cells they refer to change depending on where the name is used. There are three types of relative named ranges:

- **Column-relative**—The referenced column can change but the referenced row remains fixed. These can be identified because the absolute reference symbol (\$) only prefixes the row number. The address A\$1 is an example of a column-relative address.
- **Row-relative**—The referenced row can change but the referenced column remains fixed. These can be identified because the absolute reference symbol (\$) only prefixes the column letter. The address \$A1 is an example of a row-relative address.
- **Fully relative**—Both the referenced row and the referenced column can change. In fully relative named ranges, neither the row nor the column is prefixed with the absolute reference symbol (\$). The address A1 is an example of a fully relative address.

To create a relative named range you must first select a cell whose position you will define the name relative to. This cell is your **starting point**. This cell is not the only cell where the name can be used; it simply gives you a point from which to define the relative name.

In the next example we demonstrate how to define and use a fully relative named range that allows you to create formulas that automatically adjust the range they refer to when a row is inserted directly above them. First let's see why this is important.

Figure 4-3 shows a simple table of sales for three hypothetical regions. The total sales for all three regions are calculated using the built-in SUM worksheet function, which you can see displayed in the formula bar.

B5		=SUM(B2:B4)			
	A	B	C	D	
1		Sales			
2	Region A	10			
3	Region B	10			
4	Region C	10			
5	Total Sales	30			
6					
7					

FIGURE 4-3 Total sales using a standard formula

Now assume we need to add a fourth region to our list. We insert a new row directly above the Total Sales row and add Region D. Figure 4-4 shows the result.

B6		=SUM(B2:B4)			
	A	B	C	D	
1		Sales			
2	Region A	10			
3	Region B	10			
4	Region C	10			
5	Region D	10			
6	Total Sales	30			
7					

FIGURE 4-4 Insert an additional region to the list

Because the new region was inserted at the bottom of the list, the SUM function range did not adjust and the Total Sales number reported by the function is now wrong. This example is overly simplistic and designed to make the problem blindingly obvious. In real-world worksheets, this type of mistake is frequent and rarely so obvious. In fact it is one of the most common errors we discover when auditing malfunctioning worksheets.

This error is easy to avoid by defining a fully relative named range that always refers to the cell directly above the cell where the name is used. To do this, choose *Insert > Name > Define* to display the Define Name dialog

(or better yet, use the Ctrl+F3 keyboard shortcut). In Excel 2007 select the *Formulas tab > Define Name*. As you can see in Figure 4-5, our starting point is cell B6 and we have defined a fully relative, sheet-level named range called ptrCellAbove that refers to cell B5.

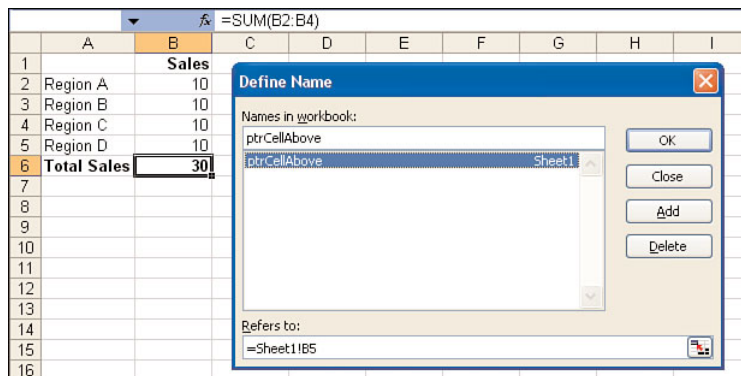


FIGURE 4-5 Creating a fully relative named range

Next we modify our SUM function so it references the ptrCellAbove named range rather than a specific ending cell address, as shown in Figure 4-6. We've also changed the first cell entry to B1.

	B6		Σ	=SUM(B1:ptrCellAbove	
	A	B	C	D	
1		Sales			
2	Region A	10			
3	Region B	10			
4	Region C	10			
5	Region D	10			
6	Total Sales	40			
7					

FIGURE 4-6 Using a fully relative named range in a worksheet function.

Not only does our SUM function now display the correct answer, you can insert as many rows as you want directly above it or directly below the header, and it will always sum the correct area. We use relative named ranges extensively in our sample application.

NOTE In Excel 2002 and higher there is a feature that attempts to automatically detect when you have invalidated a formula by inserting rows directly above it as shown in the previous example. This feature works well for simple scenarios like the one we describe here, but it can be confused by more complex scenarios as well as turned off completely in the *Tools > Options > Edit* settings. It is always better to construct your worksheets to be self-correcting in the first place.

Named Formulas

The least understood and most powerful defined name type is the named formula. Named formulas are built from the same Excel functions as regular worksheet formulas, and like worksheet formulas they can return simple values, arrays, and range references.

Named formulas allow you to package up complex but frequently used formulas into a single defined name. This makes the formula much easier to use, because all you need to do is enter the defined name you've assigned to it rather than the entire formula. It also makes the formula easier to maintain because you can modify it in one place (the Define Name dialog) and the changes automatically propagate to every cell where the defined name is used.

In the "Practical Example" section of this chapter we show an example of how to use a named formula to package a complex worksheet formula into a defined name to make it more maintainable and easier to use.

Named formulas can also be used to create **dynamic lists**. A dynamic list formula is used to return a reference to a list of entries on a worksheet when the number of entries in the list is variable. Worksheet user interface development makes extensive use of dynamic lists for data validation purposes, a topic we cover in depth in the "Data Validation" section later in the chapter, but let's revisit the time sheet from Figure 4-1 to show a quick example.

In this type of user interface, we wouldn't want the user to enter arbitrary activity names in the Activity column. To make our data consistent from user to user we would define a data validation list of acceptable activity names and the user would pick the activity that most closely described what they were doing from our predefined data validation list. We put our activity list on a background worksheet (one not designed to be seen by the user) and create a dynamic list named formula that refers to it. This named formula is shown in Figure 4-7.

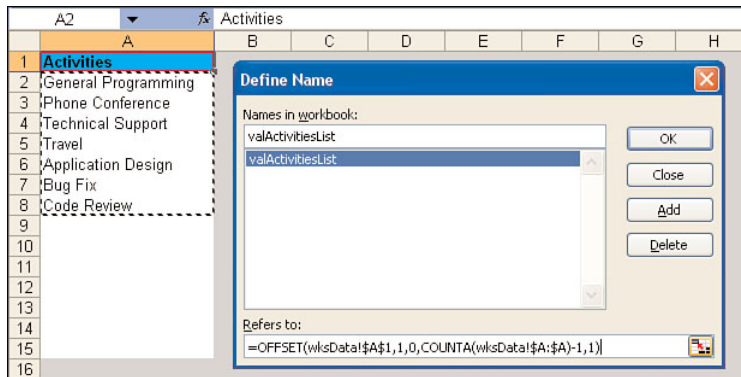


FIGURE 4-7 A dynamic named formula

The valActivitiesList named formula can now be used as the data validation list for the time sheet Activity column. A dynamic list named formula consists of the following parts:

- **Starting point**—The point at which the list begins. In this case our starting point is cell wksData!\$A\$1.
- **Data area**—The full range in which items of our list might be located. This includes not only cells that are currently being used, but also cells that might be used in the future. In this case our data area is the entire column A, or wksData!\$A:\$A.
- **List formula**—A formula that determines the number of items currently in the list and returns a range reference to just those items. This is a combination of the OFFSET and COUNTA worksheet functions.

Scope of Defined Names

Defined names can have one of two scopes: worksheet-level or workbook-level. These are roughly analogous to private and public variables. Like variables, defined names should be given the most limited scope possible. Always use worksheet-level defined names unless you must make a name workbook-level.

When your workbook contains a large number of defined names, using worksheet-level defined names helps reduce the number of names you have to manage in the Define Name dialog at the same time. Worksheet-level defined names can be used from other worksheets in most cases.

When they are used from another worksheet they are simply prefixed with the name of the worksheet from which they originated. This makes auditing worksheets that use defined names much simpler because you don't have to look up every defined name you come across in the Define Name dialog to determine which worksheet it references.

It is also often useful to have the same defined name on multiple worksheets in your user interface workbook. Two good examples of this are general-purpose, fully relative range names such as the `ptrCellAbove` range we discussed earlier and names that hold the values of settings you want to make to each worksheet using VBA code. We cover the latter in more detail in Chapter 5, "Function, General, and Application-Specific Add-ins."

Some circumstances require you to use workbook-level defined names. The most common case is demonstrated in Figure 4-7. A defined name that refers to a range located on a different worksheet that you want to use in a data validation list must be a workbook-level defined name. This is a limitation inherent in Excel's data validation feature.

In some cases a workbook-level defined name is simply appropriate, such as when the name truly refers to the entire workbook rather than to any individual worksheet. This would be the case with a named constant used to identify the version number of a workbook. In the "Practical Example" section of Chapter 7, "Using Class Modules to Create Objects," we demonstrate the use of a workbook-level defined constant to identify workbooks that belong to our application.

Styles

Styles provide a number of advantages that make them an integral part of any worksheet user interface. They provide a simple, flexible way to apply similar formatting to all the cells in your worksheet user interface that serve a similar purpose. The consistent use of styles also gives the user clear visual clues about how your user interface works. Using our time sheet example from Figure 4-1, Figure 4-8 shows how different styles define different areas of the worksheet user interface.

Styles allow you to apply the multiple formatting characteristics required for each user interface range all at once. Formatting characteristics commonly applied through the use of styles include number format, font type, background shading, and cell protection. Other style properties, such as text alignment and cell borders, are less commonly used because they tend to be different, even within cells of the same

[illegible]

FIGURE 4-8 Using styles as visual indicators of the structure of your user interface

style. Custom styles, which we discuss in the next section, can be configured to ignore the formatting characteristics you don't want to include in them.

If you need to change the format of a certain area of your user interface, you can simply modify the appropriate style and all the cells using that style will update automatically. Here's an all too common real-world example of where this is very useful.

You've created a complex, multisheet data entry workbook using white as the background color for data entry cells. When you show this to your client or boss, they decide they want the data entry cells to be shaded light yellow instead of white. If you didn't use styles to construct your user interface you would have to laboriously reformat every data entry cell in your workbook. If you did use styles, all that's required is to change the pattern color of your data entry style from white to light yellow and every data entry cell in your workbook will update automatically. Given the frequency with which people change their minds about how their applications should look, using styles throughout an application can save you a significant amount of time and effort.

Creating and Using Styles

Adding custom styles is not the most intuitive process in Excel, but once you've seen the steps required, you'll be creating styles like an expert in no time. Custom styles are created using the *Format > Style* menu (select the *Home tab > Cell Styles > New Cell Style* in Excel 2007). This opens the Style dialog, shown in Figure 4-9, from which all style confusions originate.

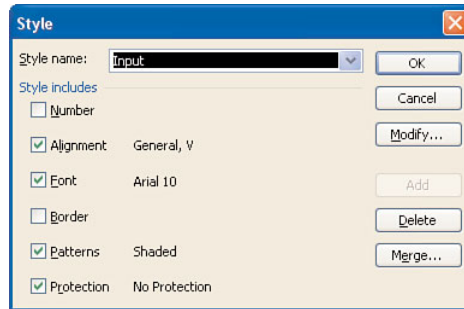


FIGURE 4-9 The Excel Style dialog

When the Style dialog first opens in Excel 2003 and earlier, it automatically displays the formatting characteristics of the cell that was selected when the dialog was invoked. In Figure 4-9, the Style dialog was invoked while the selected cell was in the Start Time column shown in Figure 4-8. As you can see, this cell was formatted with the Input style, so this is the style displayed by the Style dialog.

To create a new style, enter the name of the style you want to create in the *Style name* combo box as shown in Figure 4-10.

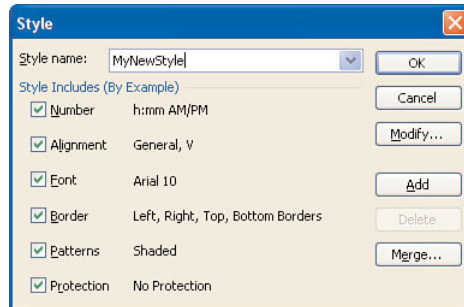


FIGURE 4-10 A new style is always based on the style of the cell selected when the Style dialog is displayed.

Once you do this you will encounter one of the more confusing aspects of the Style dialog. All of the *Style Includes* check boxes will be checked and their values will be set to the format of the cell that was selected when the Style dialog was invoked. This occurs even if those format characteristics are not part of the style currently applied to that cell.

NOTE In Excel 2007 the Style dialog opens with a default style name and default style settings regardless of the style applied to the selected cell when the dialog was displayed.

For example, Number, Alignment and Border attributes were excluded from the Input style that was displayed in the Style dialog immediately before we created our new style. All three of those attributes are included in our new style, however, and their specific values are drawn from the format applied to the cell that was selected when the Style dialog was first invoked. This is what the *By Example* in parentheses after the *Style Includes* title means. Don't worry; all of these attributes can easily be changed.

First, remove the check mark from beside any format option that you don't want to include in your style. When a style is applied to a range, only the format options you checked will be applied. Next, click the Modify button (or the Format button in Excel 2007) to define the properties of your new style. This displays the Format Cells dialog, shown in Figure 4-11.

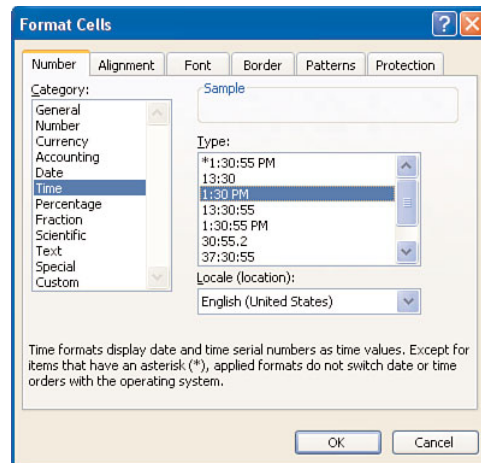


FIGURE 4-11 The Format Cells dialog as invoked from the Style dialog Modify button

Notice that the six tabs on the Format Cells dialog correspond exactly to the six *Style Includes* options shown in Figure 4-10. This is no accident. Styles are simply a way of grouping multiple cell format characteristics under a single name so they can be applied and maintained simultaneously through that name.

NOTE If you remove the check mark from a *Style Includes* option but then change any of the characteristics of that option in the Format Cells dialog, the option automatically becomes checked again in the Style dialog.

Modifying Styles

Modifying an existing style in Excel 2003 and earlier is exactly like creating a new style except that after selecting the *Format > Style* menu, you pick the style you want to modify from the *Style name* combo box rather than entering a new style name. Each time you select a style in the *Style name* combo box, that style will have its settings summarized and displayed for you in the *Style Includes* section of the dialog. Click the Modify button to display the Format Cells dialog and change any of the format options for the currently selected style. In Excel 2007 you modify an existing style by selecting the *Home tab > Cell Styles* drop-down, then right-click on the style sample button that displays the style name you want to modify, and choose *Modify* from the shortcut menu.

There is one minor caution to keep in mind when creating new styles or modifying existing styles in Excel 2003 and earlier. Once you have configured the style using the Format Cells dialog, be sure to click the Add button on the Style dialog to save your changes. If you click the OK button, your changes will be saved, but the style you have created or modified will also be applied to the currently selected cell. This is often not the result you want. Getting into the habit of using the Add button to add and update styles will save you from having to undo changes to a cell you didn't intend to change. Once you've used the Add button to create or modify a Style, you can safely use the Cancel button to dismiss the Style dialog without losing your work or formatting the currently selected cell.

Adding the Style Drop-Down to the Toolbar

If you're familiar with Word, you'll notice styles there are considered so important that a special style drop-down is automatically present on the Formatting toolbar. This not only allows you to quickly apply a style to a selection but also displays the style associated with the section of the document where your cursor is located. Excel 2003 and earlier has a similar toolbar control, but for some reason styles in Excel were not deemed important enough by Microsoft to have this control appear by default. You can add this control to one of your Excel toolbars manually, however, and if you plan on making full use of styles in Excel you should do so. Here's how:

1. Start by selecting *View > Toolbars > Customize* from the Excel menu.
2. In the Customize dialog select the *Commands* tab.
3. In the *Commands* tab select the *Format* item from the *Categories* list. As shown in Figure 4-12, the *Style* drop-down is the fifth item in the *Commands* list box.

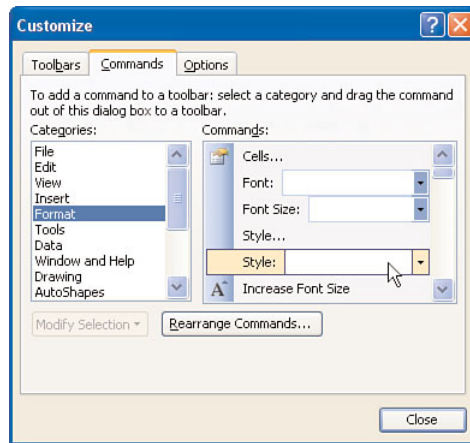


FIGURE 4-12 Selecting the Style drop-down from the list of format controls

4. Drag this control from the *Commands* list box and drop it onto one of your existing toolbars. You will now have a *Style* control that provides most of the same benefits as the *Style* control in Word. (It does not show the style names using their format as the Word *Style* control does.)

You can select a group of cells and apply a style to all of those cells by simply selecting the style name from the *Style* drop-down. And when you select a cell, the name of the style applied to that cell automatically is displayed in the *Style* drop-down. This feature is very helpful when creating complex worksheet user interfaces that utilize many different styles.

User Interface Drawing Techniques

Excel provides built-in tools with a surprising amount of flexibility for customizing worksheet user interfaces. In this section, we examine how to use these tools to improve the appearance and functionality of your worksheet user interface.

Using Borders to Create Special Effects

To keep the user focused on the elements of your worksheet user interface, it is often helpful to modify the normal style so that all unused areas of the worksheet have a consistent, light gray background color. This practice has been demonstrated in most of the user interface examples shown so far and will be used in our sample application. On top of this light gray background you can use cell borders to create some interesting special effects. One of the most commonly used border-based special effects gives a range of cells a 3D appearance, either raised or sunken. Examples of both effects are shown in Figure 4-13.

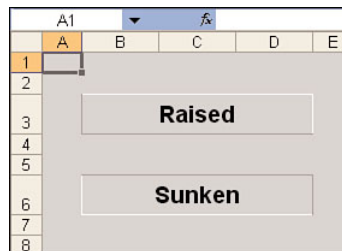


FIGURE 4-13 Using borders to create 3D visual effects

To create a raised effect you simply add a white border to the top and right sides of your range and add a 50% gray border to the left and bottom sides of your range. To create a sunken effect you do exactly the opposite. The width of the borders can be used to control the degree of the effect.

When you apply a background color to a worksheet, as we did in the previous example, Excel's standard gridlines are obscured. In many cases gridlines are a useful visual guide for the user, so you want to put them back. While there is no way to force Excel's standard gridlines to display over a background color, you can easily simulate gridlines by adding 25% gray borders with the lightest width to the area where you want the gridlines to appear. This effect is shown in Figure 4-14.

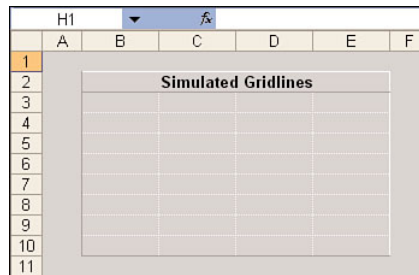


FIGURE 4-14 Using borders to simulate gridlines

Creating Well-Formatted Tables

Tables used within an Excel worksheet user interface typically have one or more of the following elements:

- Table description
- Row and column descriptions
- Data entry area
- Formula result area

Each section of your table should be formatted with a unique style that you use consistently throughout your user interface. Figure 4-15 shows a sample table with all four of the elements described previously.

	A	B	C	D	E	F	G
1							
2							
3							
4							
5							
6							
7							
8							
9							
10							
11							

FIGURE 4-15 A basic worksheet user interface table layout

As you can see, in its simplest form the table is not very attractive. You can give your tables a much more professional appearance by using borders to provide a 3D effect, adding simulated gridlines, and increasing the row heights and column widths to provide more visual separation. Turning off row and column headers and the formula bar completes the effect. The

table now looks like a completely custom user interface. Figure 4-16 shows the table with these added effects.

	Data Entry Area 1	Data Entry Area 2	Data Entry Area 3	Formula Result
Row 1				
Row 2				
Row 3				
Row 4				
Row 5				

FIGURE 4-16 A fully formatted worksheet user interface table

Cell Comments for Help Text

Cell comments are one of the most important user interface features provided by Excel. Their utility stems from the fact that in many cases they can serve the same purpose as a help file without requiring the user to do anything more complicated than hover the mouse cursor over the commented cell. Note that cell comments have several limitations that may make them inappropriate in certain situations:

If you are using the freeze panes feature on a worksheet and the worksheet is scrolled beyond the freeze point, if the comment window overlaps the frozen row and/or column it will be cut off at the point where the window is frozen.

Each cell comment is also associated with a specific status bar message whose structure cannot be modified. The status bar message displayed when a user hovers the mouse over a comment has the following structure, which is shown graphically in Figure 4-17.

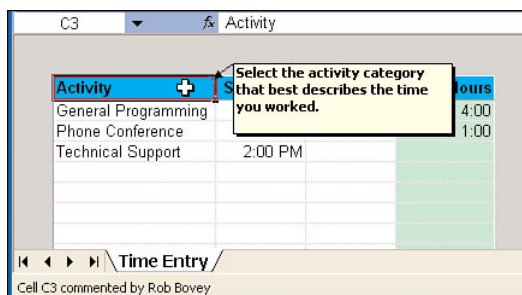


FIGURE 4-17 The format of an Excel comment status bar message

Cell address commented by **user name at the time the comment was created**.

The only part of this message you can modify is the **user name at the time the comment was created** section, which displays the contents of the *User name* entry located under the *Tools > Options > General* tab of the Excel menu (in Excel 2007 this is located under *Office Button > Excel Options > Popular*). If you are a consultant creating a worksheet user interface for a client, it's unlikely your client wants to see your name in the status bar each time she views a cell comment. In that case, one of the best workarounds is to change the *User name* setting on your machine to your client's company name while you create the comments for their client's user interface. Once the comments have been created, the user name displayed in the status bar is fixed and will not be affected when you change your *User name* setting back to your own name.

Remember that cell comments can be rich-text formatted. This means you can use formatting such as bold and italic fonts within the comment text as well as multiple fonts. Rich-text formatting allows you to create some sophisticated help messages. Figure 4-18 shows a rich-text formatted cell comment from a real-world worksheet user interface.

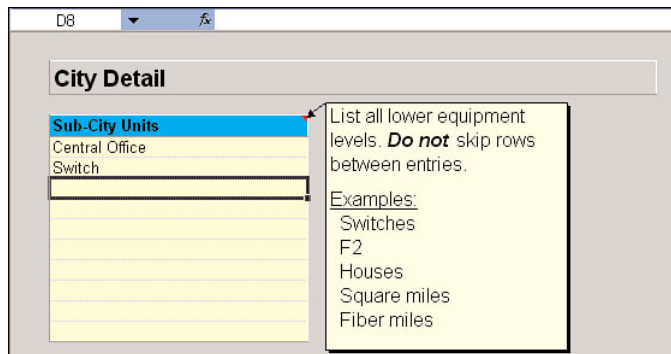


FIGURE 4-18 A rich-text formatted cell comment

Using Shapes

The ability to use shapes (objects drawn using the various options on the Drawing or Forms toolbars) on an Excel worksheet is a powerful user interface technique. Shapes are located in a special drawing layer that floats above the cells on a worksheet, so shapes cover (and obscure) worksheet cells. Shapes are also connected to the underlying worksheet through their properties, which allow them to

- Move and size with the worksheet cells they cover
- Move but don't size with the worksheet cells they cover
- Don't move or size with the worksheet cells they cover

Almost all shapes can contain text. A shape's text can either be manually entered or it can be linked dynamically to a specific cell on a worksheet by selecting the shape and entering the address of that cell as a formula in the formula bar. As you can imagine, the ability to assign formulas to shapes opens up a wide array of options for creating dynamic user interfaces. Shapes can also be given a macro assignment that causes them to execute the specified macro whenever the user clicks them. Simply right-click over the shape and choose *Assign Macro* from the shortcut menu. Figure 4-19 shows an excellent example of how shapes can be used to create a custom toolbar-like area across the top of a worksheet user interface.

These simulated toolbar buttons were created using professionally drawn clip-art images. These types of images can be found in many different places on the Web and using them in a situation like this is much preferable to laboriously drawing your own images.

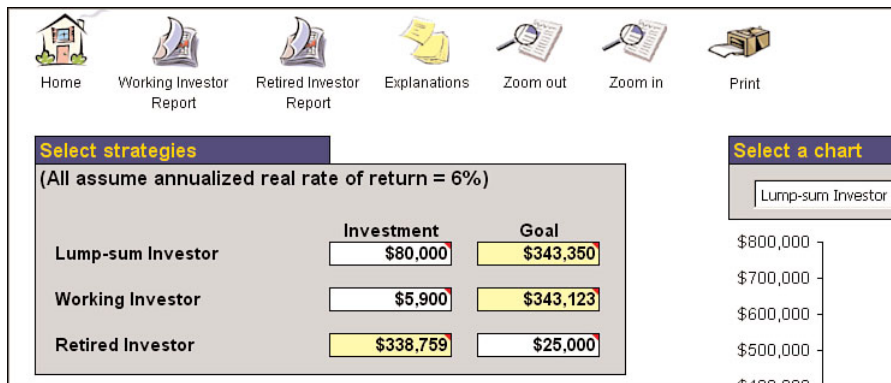


FIGURE 4-19 A custom on-sheet toolbar created with shapes

Data Validation

Data validation is one of the most useful yet underutilized features for worksheet user interface design. It allows you to ensure that most, if not all, of the inputs in your user interface are made correctly by disallowing

input that does not match the rules you specify. Data validation can be as simple as restricting cell entries to whole numbers or as complex as restricting cell entries to items on a list whose contents are conditionally determined based on an entry made in a previous cell.

We assume you understand the basic use of data validation and instead demonstrate two of the more complex validation scenarios that can be created with this feature. Most complex data validation scenarios involve data validated lists or custom data validation formulas.

Unique Entries

If you need the user to enter only unique item names in a data entry list you can use a custom data validation formula to enforce uniqueness. First select the entire data entry area you need to validate. Next, choose *Data > Validation* from the menu (*Data tab > Data Validation* in Excel 2007) and select the *Custom* option from the *Allow* list. The basic syntax of the formula you need to enter is the following:

```
=COUNTIF(<entire range>,<relative reference to input cell>)=1
```

The first argument to the COUNTIF function is a fixed reference to the entire data entry area that must contain unique entries. The second argument to the COUNTIF function is a relative reference to the currently selected cell in the data input range.

If each entry is unique, the COUNTIF function evaluates to 1 and the entire formula evaluates to True, meaning the data is valid. If the COUNTIF function locates more than one instance of an entry in the data entry area, the entire formula will evaluate to False and data validation will prevent that entry from being made. Figure 4-20 shows an example of this validation setup and Figure 4-21 shows it in action.

NOTE The enforce unique entries data validation technique described previously only works correctly if the range being examined is entered into the Data Validation dialog as a hard-coded range address. If you try to use an equivalent defined name, this data validation technique will fail. This is a bug in the Excel data validation feature.

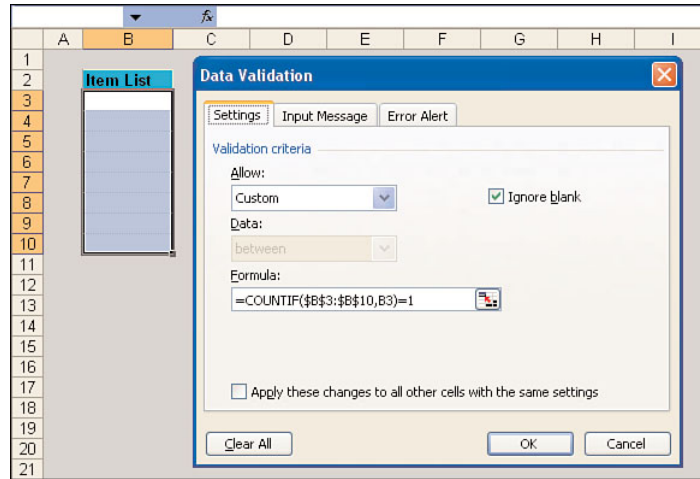


FIGURE 4-20 Data validation configuration to force unique entries in a list

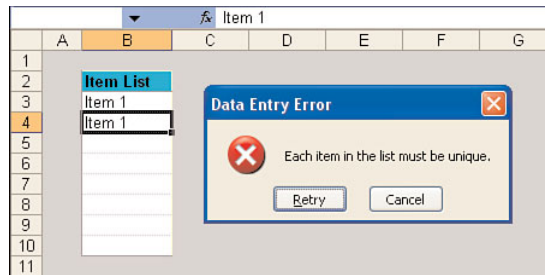
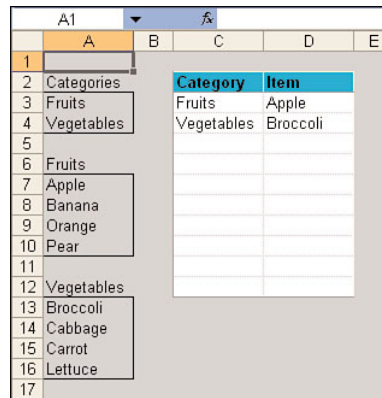


FIGURE 4-21 Unique entries data validation in action

Cascading Lists

In this type of validation, the specific data validation list that is displayed for a cell is determined by the entry selected in a previous cell. In Figure 4-22, the data validation list for the Item column is determined by the selection in the Category column. All of the data validation lists are located in the hidden column A.



	A	B	C	D	E
1					
2	Categories		Category	Item	
3	Fruits		Fruits	Apple	
4	Vegetables		Vegetables	Broccoli	
5					
6	Fruits				
7	Apple				
8	Banana				
9	Orange				
10	Pear				
11					
12	Vegetables				
13	Broccoli				
14	Cabbage				
15	Carrot				
16	Lettuce				
17					

FIGURE 4-22 Initial setup for cascading data validation lists

The Categories list is the data validation list for the Category column. The Fruits list is the data validation list for the Item column when the Category selected is Fruits. The Vegetables list is the data validation list for the Item column when the Category selected is Vegetables. Each of these lists has been given the worksheet-level defined name shown in the caption above their border. Figure 4-23 shows all of the defined names used in this example.

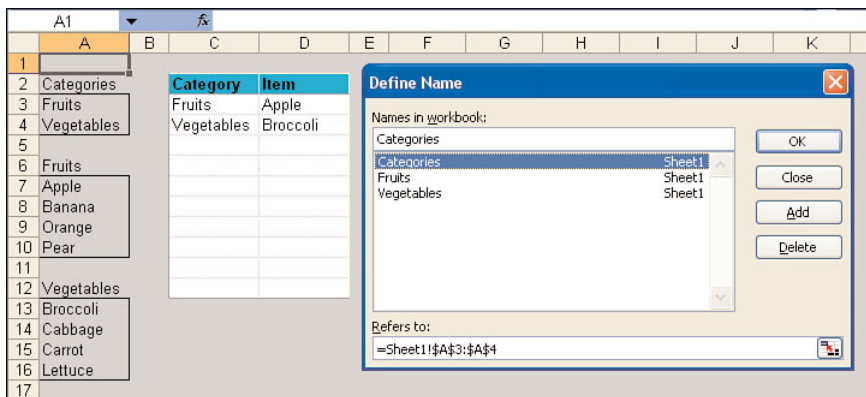


FIGURE 4-23 Defined names used for cascading data validation lists

The data validation list formula for the Category column is simple: `=Categories`. The data validation list formula for the Item column is a bit more complicated. It has to check the value of the corresponding Category entry and do one of three things: display no list if the Category entry has not been selected, display the list of fruits if Fruits has been selected, or display the list of vegetables if Vegetables has been selected. The formula that does this is shown here:


```
=IF (ISBLANK (C3) , " " , INDIRECT (C3) )
```

If the cell in the Category column is blank, the formula returns an empty string, which removes the data validation list from the Item cell next to it. If the cell in the Category column has an entry, the formula uses the INDIRECT worksheet function to coerce that Category column entry into a range reference. The range reference refers to either the Fruits list or the Vegetables list depending on which item the user selected in the Category column. As Figure 4-24 shows, this formula successfully displays two completely different data validation lists depending on the category selection.

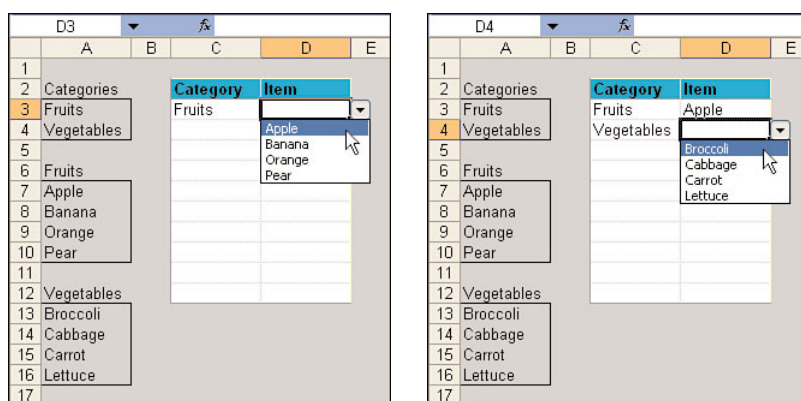


FIGURE 4-24 Cascading list validation in action

This logic can be extended to as many categories as you need. However, for cases with large numbers of categories, a table-driven approach that you'll see used in our sample time sheet application is much easier to set up and maintain.

Note that one drawback of this type of validation is that it doesn't work in both directions. In the scenario described previously, there is nothing to stop a user from accidentally changing the category entry in a row where a specific item has already been selected. In the next section you see how to use conditional formatting to provide a visual indication that this kind of error has been made.

Conditional Formatting

Conditional formatting is one of the most powerful features available for Excel user interface development. It allows you to use simple worksheet formulas to accomplish things that would otherwise require many lines of VBA

code. Conditional formatting works by modifying the appearance of cells it has been applied to only if one or more conditions that you specify have been met. Conditional formatting overrides any style setting when the condition is triggered. Once the condition that triggered the conditional formatting is no longer true, the affected cell reverts to its original format. The two most common uses of conditional formatting in Excel user interface development are the creation of dynamic tables and calling out error conditions.

Creating Dynamic Tables

When building non-trivial worksheet-based user interfaces, you will often be faced with the problem of providing a table that in extreme cases will allow the entry of some large number of rows but the most common scenarios will only require a few. Rather than hard-coding a visible table with the maximum possible number of rows, you can use conditional formatting to create a table that expands dynamically as data is entered into it. We demonstrate how this is done beginning with the sample table shown in Figure 4-25.

	A	B	C	D	E	F	G
1							
2		Item Name	Price	Quantity	Total Cost		
3							
4							
5							
6							
7							
8							
9							
10							
11							
12							
13							
14							

FIGURE 4-25 Data entry table prior to the addition of dynamic formatting

Let's assume this table really requires 200 rows for the largest projects but most users only need a few rows of input. Therefore, you'd like to hide the unused area of the table. As you can see, the first step in creating a dynamic table is to draw the entire table on the worksheet. You then use conditional formatting to hide the unused area of the table and reveal rows dynamically as needed. The trigger for displaying a data entry row is the user entering a new name into the Item Name column. For that reason, we always need to leave an empty Item Name entry cell at the bottom of the table.

When creating a dynamic table, it's a good idea to also create an outline showing the extent of the table in one of your hidden columns. Once we've added the conditional formatting the table disappears. This makes the table

difficult to maintain if you haven't provided yourself with a visual marker indicating its extent. The empty bordered area in column A serves this purpose in our example. This area doesn't need to be empty. It could include error-checking formulas, for example. As long as it gives you a visual indication of the extent of the hidden area of the table it serves its purpose.

Our dynamic table requires three different conditionally formatted sections. Referring back to Figure 4-25, the first section encompasses range C3:C12, the second section encompasses range D3:F12, and the third range encompasses range G3:G12. We add the conditional formats one step at a time so you can see the results as they occur. To make the operation of the conditional formats more obvious we add data to the first row of the table. Keep in mind that the purpose of all three conditional formatting sections is the same: to simulate the appearance of a table that is just large enough to hold the data that has been entered into it. Figure 4-26 shows the table with the first section of conditional formatting completed.

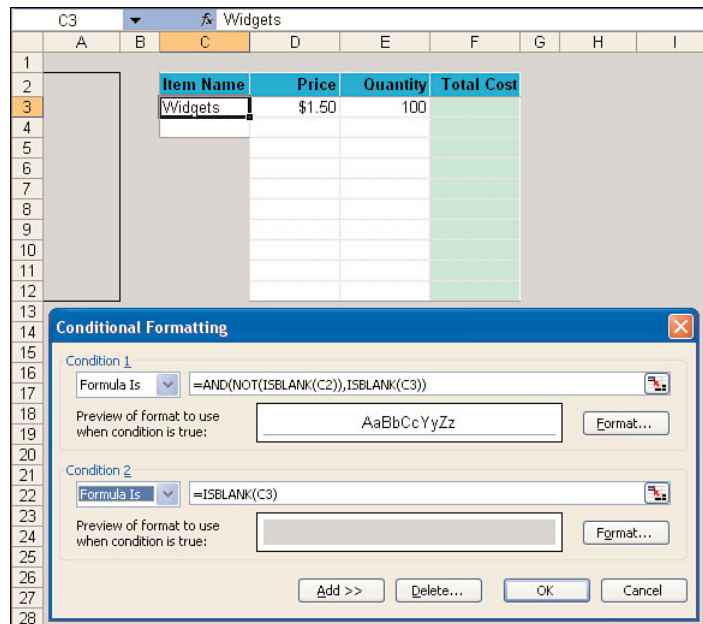


FIGURE 4-26 Conditional formatting for the first column

In addition to the purpose described previously, the first conditional format serves to leave a blank cell in front of the first unused table row to help prompt the user to enter the next item. The second conditional format is shown in Figure 4-27. It clears all unused rows in columns D through F

and draws a bottom border below the first unused row in the table, thereby helping to complete the table outline.

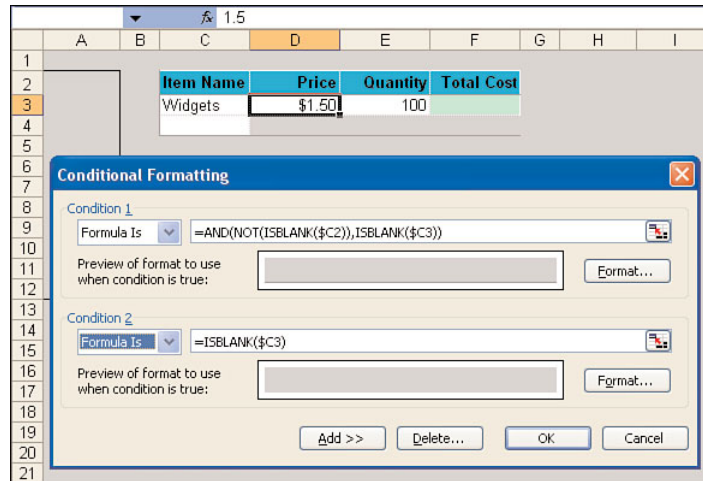


FIGURE 4-27 Conditional formatting for the remaining columns within the table

You can see the white border on the far right side of the table is missing in Figure 4-27. The purpose of the third conditional format is to complete the simulated table by drawing this border. Figure 4-28 shows the third conditional format.

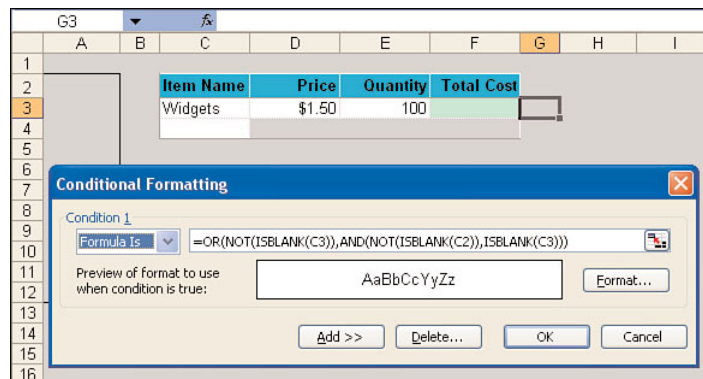


FIGURE 4-28 Conditional formatting outside the table to create the right-hand border

Figure 4-29 shows the fully formatted table with some additional entries. Each time a new entry is made, the conditional format reveals the row in which the entry was placed and adds a new prompt row below it.

Item Name	Price	Quantity	Total Cost
Widgets	\$1.50	100	\$150.00
Pieces	\$1.00	50	\$50.00
Parts	\$2.00	125	\$250.00

FIGURE 4-29 The complete dynamically formatted table

The one major caveat when considering the use of conditional formatting to create dynamic tables is that calculation must be set to automatic for it to work. If your user interface workbook is so calculation intensive that you need to set calculation to manual, you cannot create dynamic tables using this method (or use any other type of formula-based conditional formatting for that matter).

Calling Out Error Conditions

Conditional formatting can also work alone or in concert with formulas in hidden rows and columns to highlight invalid entries as soon as they are made. This should not be your method of first choice for pointing out data entry errors. Always try to use data validation to prevent data entry errors from being made in the first place.

A common situation where errors cannot be prevented by data validation is when you have two data entry columns such that the entry in the first column determines the allowable entries in the second column. In Figure 4-30 we revisit our cascading data validation list example from Figure 4-22.

Even though both columns' lists are data validated, an error can creep in if the user initially selects a valid category and item combination but then accidentally changes the category name at some later point in time. This type of mistake cannot be prevented by data validation, so we need to provide some visual indication that there is a mismatch between the category and item selections if this error occurs. This is a task for conditional formatting.

As you can see in Figure 4-30, we inserted a second hidden column. In this column we created an error check for each row that verifies the

entry selected in the Item column is valid for the selection in the Category column.

	B3		=IF(ISBLANK(E3),FALSE,ISERROR(
	A	B	C	D	E	F
1						
2	Categories	HasError		Category	Item	
3	Fruits	FALSE		Fruits	Apple	
4	Vegetables	FALSE		Vegetables	Broccoli	
5		FALSE				
6	Fruits	FALSE				
7	Apple	FALSE				
8	Banana	FALSE				
9	Orange	FALSE				
10	Pear	FALSE				
11		FALSE				
12	Vegetables	FALSE				
13	Broccoli					
14	Cabbage					
15	Carrot					
16	Lettuce					
17						

FIGURE 4-30 The error check formula column for the conditional format

The error check formula is as follows. Keep in mind that the purpose of the error check formula is to return True if the corresponding row in the table has a data entry error and False otherwise.

`=IF (ISBLANK (E3) , FALSE , ISERROR (MATCH (E3 , INDIRECT (D3) , 0)))`

The only type of error that can occur in this situation is the Item column entry not matching the Category column entry. If there is no Item column entry, the row is not complete and we cannot determine the validity of the Category column entry. The `ISBLANK` function checks for this condition and returns `FALSE` if this is the case. Once there is an entry in the Item column, the formula uses the `INDIRECT` function to return a reference to the list of valid entries (recall that the Category column entry is the same as the range name of the corresponding Item list). The formula then uses the `MATCH` function wrapped in the `ISERROR` function to return `TRUE` if the Category entry is located in the list or `FALSE` if it isn't.

The next thing we do is add a conditional format to the table that checks the value of the `HasError` column. If the `HasError` column indicates there is an error in one of the table rows, our conditional format will give that row a bright red shade. Error condition highlighting is one exception to the rule of not using garish colors in your user interface. We do recommend using red, however, as this is almost universally recognized as a warning color. The conditional format required to accomplish this is shown in Figure 4-31.

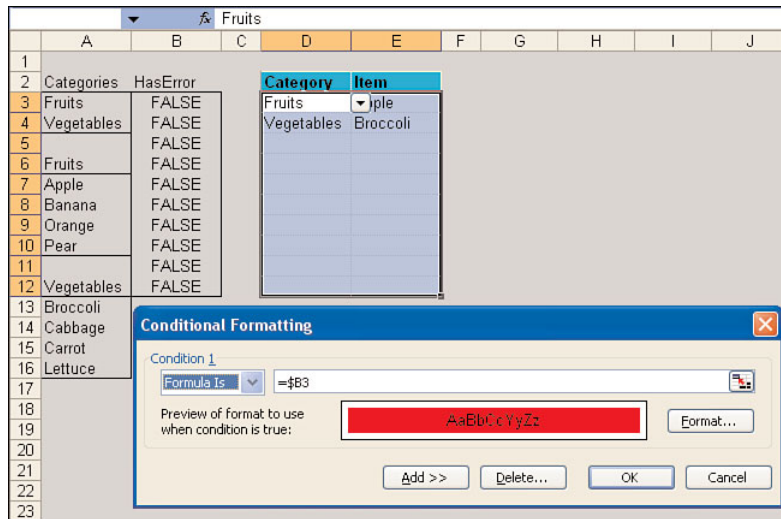


FIGURE 4-31 Setting up conditional formatting to flag an error condition

The result of the conditional format in response to an error condition is shown in Figure 4-32, where we've changed the Category column entry in the second table row from Vegetables to Fruits so it no longer matches the entry in the Item column.

A1					
	A	B	C	D	E
1					
2	Categories	HasError	Category	Item	
3	Fruits	FALSE	Fruits	Apple	
4	Vegetables	TRUE	Fruits	Broccoli	
5		FALSE			
6	Fruits	FALSE			
7	Apple	FALSE			
8	Banana	FALSE			
9	Orange	FALSE			
10	Pear	FALSE			
11		FALSE			
12	Vegetables	FALSE			
13	Broccoli				
14	Cabbage				
15	Carrot				
16	Lettuce				
17					

FIGURE 4-32 Conditional formatting flagging a bad entry in the table

Using Controls on Worksheets

Making extensive use of controls placed directly on worksheets is typically not the best user interface design. For most Excel application development,

we recommend you use custom command bars (or Ribbon controls in Excel 2007) as entry points into your code and substitute data validation lists for combo box controls on worksheets. Command bars are covered in Chapter 8, “Advanced Command Bar Handling,” while the Ribbon is covered in Chapter 10, “The Office 2007 Ribbon User Interface.” There are circumstances where placing controls directly on your worksheet user interface is the best option, so in this section we cover some of the things you need to watch out for when you do this.

When you do need to use controls on a worksheet you have to make the choice between ActiveX controls and controls from the Forms toolbar. Generally we recommend you use Forms controls unless you absolutely need ActiveX controls. Forms controls are very lightweight and don’t exhibit the many quirks you’ll run into when using ActiveX controls on worksheets. Figure 4-33 shows a worksheet in which Forms controls have been used to great effect.

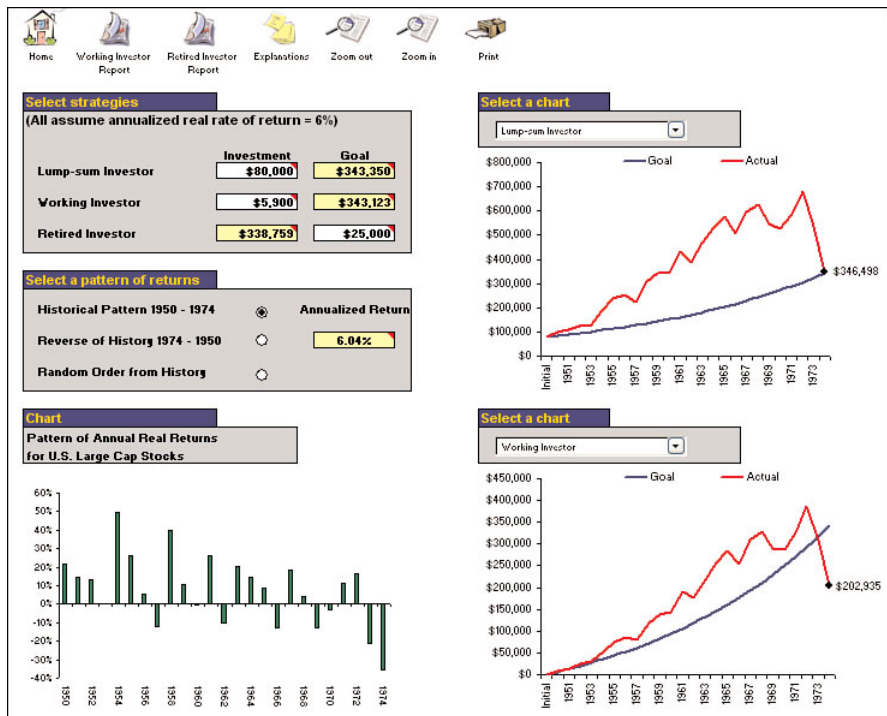


FIGURE 4-33 Good use of forms controls on a worksheet

Since everyone reading this chapter should be familiar with how controls work, we simply cover the details critical to deciding whether you can use

Forms controls in your worksheet user interface or whether you need ActiveX controls.

Advantages of Forms Controls

- Forms controls can be used on Chart sheets; ActiveX controls cannot.
- Forms controls are more tightly linked to Excel. You can select a Label or Button control and enter a formula in the formula bar that dynamically sets the captions of those controls. And unlike its ActiveX counterpart, a Forms control Listbox updates its contents in response to changes to a dynamic named range that has been assigned to its Input range property.
- It is easy to assign multiple Forms controls to run the same VBA procedure. Doing the same with ActiveX controls requires a more complicated class-based approach.
- If you use multiple windows or the split panes feature in your application to show two different views of the same worksheet, ActiveX controls will only work in the original window. Forms controls will work in any window.

Advantages of ActiveX Controls

- You can modify the appearance of ActiveX controls to a much greater degree than Forms controls.
- There are more varieties of ActiveX controls than there are Forms controls.
- ActiveX controls have a wide variety of event procedures that you can respond to, while Forms controls can only run a single macro.

Practical Example

In this chapter, we begin building a real-world Excel application that illustrates the points made in the chapter text. Our application is a time tracking system that starts as a simple, no-frills time sheet and works its way up to being a full-featured Excel application as we progress through the book. Due to space constraints we do not show every detail involved in creating

this application. We demonstrate the major features and allow you to examine the rest by perusing the finished sample of the application available on the accompanying CD. This time sheet application will henceforth be referred to by its acronym PETRAS, which stands for Professional Excel Timesheet Reporting and Analysis System.

The first version of PETRAS is a simple workbook containing a time entry table on one worksheet and data validation lists on a second hidden worksheet. The user is expected to complete the time entry table each week and manually copy the workbook to a central location for consolidation. This version of PETRAS can be located on the accompanying CD in the *\Application\Ch04-Worksheet Design* folder. It is displayed in Figure 4-34.

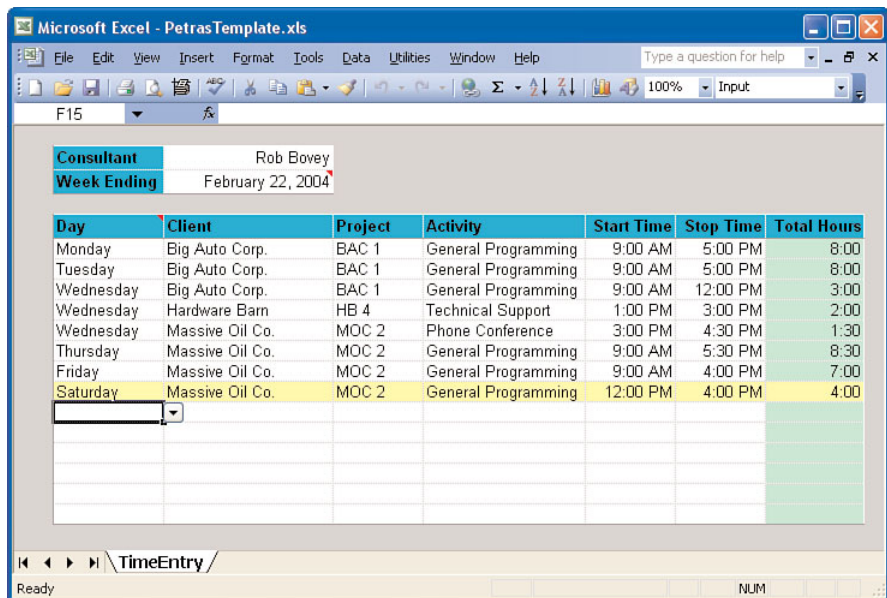


FIGURE 4-34 The first version of the PETRAS application

Most of the user interface design techniques discussed in this chapter have been used in the PETRAS application, including all variations of defined names, styles to differentiate areas by purpose, table formatting techniques, use of comments for help text, data validation, and conditional formatting. Let's quickly cover examples of how each of these techniques is used in practice.