

A Tester's Guide to .NET Programming



Randal Root and
Mary Romero Sweeney

A Tester's Guide to .NET Programming

Copyright © 2006 by Randal Root and Mary Romero Sweeney

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13: 978-1-59059-600-5

ISBN-10: 1-59059-600-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editor: Jonathan Hassell

Technical Reviewer: Phil Leder

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser

Project Manager: Beth Christmas

Copy Edit Manager: Nicole LeClerc

Copy Editor: Linda Marousek

Assistant Production Director: Kari Brooks-Copony

Production Editor: Linda Marousek

Compositor: Susan Glinert Stevens

Proofreader: Kim Burton

Indexer: Valerie Perry

Artist: April Milne

Cover Designer: Kurt Krames

Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit <http://www.springeronline.com>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Source Code section.



Automated Software Testing with .NET

Software testing is not a new field but it's growing up in a rather fractured way. Around the country and around the world, you'll find software testing employed in a variety of ways. There is now Agile and Extreme software testing, and various other buzzwords and methodologies to accompany the traditional Black Box testing that is still deeply entrenched in many companies. The emphasis we have had for years on expensive commercial tools to aid our testing efforts has lessened as the demand for their increased capability has become virtually impossible for tool vendors to keep up with. Now many test organizations have turned to producing their own software to help in testing, often using a burgeoning list of open source tools and software.

In 2001, when *Visual Basic for Testers* (Mary Romero Sweeney, Apress, 2001) was published, many test professionals were finding they needed to complement their manual- and tool-automated testing efforts with their own software utilities. In the years since then, we've found that the ability to write code and produce tests and test utilities by writing our own software is even more necessary for the software test community. At every conference and training symposium there are courses and lectures teaching testers more technical topics, including programming, networking, and databases.

We still, and always will, need to get tested software out the door quickly, efficiently, and *profitably*. There are many trade-offs to automating your own software tests; however, if you do it thoughtfully, you can help a test project immensely. If you do it incorrectly, you can slow it down such that you'll run out of budget and fail to accomplish your goals. In this book, we'll explore the use of .NET application software for test projects and show in what ways this specific type of software can support your testing goals.

You will find enough here to get you started on a successful test project using .NET. To begin, you will need some discussion of automated software testing in general. In this chapter, you will look at some of the important management issues involved when starting automated testing, such as guidelines for when and when not to automate testing, what kind of personnel requirements you will need to address, and how to build an automated testing team. You will also look at some ground rules for creating good testing software, and some of the advantages as well as limitations of using two .NET languages, Visual Basic (VB) .NET and C#, for your test projects and utilities.

What a Tester Needs to Know About .NET Coding

Although .NET languages are powerful enough to accomplish some useful testing tasks, you must have knowledgeable testers and programmers to write the code. Unfortunately, there isn't a lot of information out there yet to help test professionals adapt programming for testing purposes. Most of the resources are geared for software developers, not testers.

Using .NET languages for testing requires a shift in perspective. A tester can come out of a standard Visual Basic course still wondering how it could ever be used on a test project. These courses and most books concentrate on the controls to use and the ways to create a great, user-friendly application. A tester doesn't care about that so much—what we want to know is how to quickly develop a utility or get to system information and other testing-related data using code. One of the differences between this book and others is that we won't focus on learning a myriad of cool controls or how to develop a slick front end for an application. While these are great things to learn, there are plenty of other books out there that will teach you this. Instead, we will focus on the things a tester must know to use .NET languages as quickly as possible on a test project:

- How to access intrinsic .NET Framework library functions that return relevant information about the platform, files, registry, operating system, and so on
- How to create a front end with basic controls to view test information and results as soon as possible
- How to access databases quickly and easily
- How to access the Windows Registry to return relevant application information

These topics are just the beginning, of course, but they represent some of the things the testers who have contributed time and code to this book have used to accomplish their testing tasks. We will cover all of these and more in the course of this text.

Why .NET Languages for Testing?

.NET languages are not testing tools; instead, they are programming languages used for software development. Why use Visual Basic or C# for testing—why not use Perl, C, or C++? Scripting languages, as they are popularly called, such as Perl, Python, VBScript, Rexx, and many others, have a large following. Why not use those? Actually, none of those languages were created with testing in mind either. Still, they can be a big benefit on a testing project, especially if they are already installed and you have experienced personnel. We would choose to do much of the testware coding using any of those if we had them readily available, as well as available employees who were at expert level in their use. If .NET languages are already available and there are employees with expertise in them, then they are an excellent choice.

If the development project is itself written in .NET, then it can make sense for the testers to use it in this situation. Although, a common misconception is that if you are testing a .NET application, you will need to use a .NET language for your test automation. That's not true; however, using .NET languages on a Windows platform will provide you with all the power you need to do essentially anything you need to do.

Since .NET languages are not really testing tools, how is it possible to adapt them for use in testing? The .NET Framework libraries have many features that can support the testing process. For example, there are a host of intrinsic functions that can return important information about the test platform and the application under test. .NET's `Shell` function and `SendKeys` class can also be used to run an application and manipulate its Graphical User Interface (GUI). The Visual Studio Database Tools allow you to connect to a database and examine its structure and data. You can also get very sophisticated and write essentially anything you want, such as a load testing application. Of course, the trade-off for a more sophisticated programming endeavor is that you will need both the programmers *and* the time.

.NET languages can also be used to test many behind-the-scenes operations of the application. For example, scripts can be written to access the system environmental variables and performance counters. Automated test scripts can verify the correct loading and retrieval of the information from files. The very fact that the .NET languages are powerful development tools makes them promising and capable tools for testing.

Choosing a .NET Language for a Test Project

.NET platforms include a lot of language options. This is because the language implementations are now just a thin layer on top of the .NET Common Language Runtime (CLR). They all compile down into the same Intermediate Language (IL). This makes choosing a language a matter of preference and not a technical decision. You can choose a language based on how easy it is to learn. For beginners, Visual Basic will be a good choice. Alternatively, if you already have done a little work in another language, such as C or Java, you can choose the .NET implementation of those languages: C++ or J#. Also a good choice, in that case, is C#, a new language that is developed specifically for .NET platforms and will be familiar enough to anyone who has programmed in a “swirly brace” language.

It's also true that some testers on the same team can choose to code in VB .NET, while others choose C#, or another language, and still be able to have their software interoperate nicely. We have to confess a little bias towards VB .NET, of course, having written a lot of code in it as well as product literature. A big advantage to using Visual Basic is that it is a popular language because it is easy to learn, and it happens to be the macro language for the widely-used Microsoft Office products and the scripting language for most of the world's ASP web pages. Many other software companies use a form of Basic for their own products. This popularity means there is a wide base of people with a knowledge of Basic, so there should be no shortage of people able to use it or willing to learn it. There is also a proliferation of books and resources available for Visual Basic. Although they may not be written specifically for testing, once you get the hang of it, you will find lots of code available in user's groups and books that you can adapt for testing purposes.

C# has been steadily gaining in popularity since it has some of the ease of VB .NET, as well as a lot of similar programming constructs to languages like Java and Perl. A common misconception about C# is that it is somehow more powerful than VB .NET. However, in the end, both C# and VB .NET are the same.

So, which to choose? How about both? Because they compile down into IL, components written in one language can be used in another. Since both Visual Basic and C# are very popular and simple to learn, we have chosen to include code from both of these languages in the examples in this book. (For reasons of space and time, we will not include J# or C++.)

What Is Automated Software Testing?

Let's take a step back and talk about what automated testing is in general and define some terms. First of all, *automated testing* is any testing that is done using software. In other words, we write code to test other code. Automated testing includes the use of tools written by others since the tools they have written *are* software that tests software. *Automated test scripting* is the process of creating the program code—that is, actually writing the code that will be used to test. *Automated test scripts*, or testware, are the program code used to test the software.

Historically, most testing has been done manually. That is, a tester sits down and runs the application using defined processes to try to find bugs so that they can be fixed prior to releasing the product. Automated software testing goes a step further. Since basic software testing has become more rigorous and more defined, testers have found ways to automate some of the process of testing software by writing software to do it. Of course, many successful testing projects have been completed without ever using automated test scripting. In fact, many applications are still primarily tested manually. There is just no substitute for testing the product in the same way that the user would and there is no substitute for the abilities of an able, experienced tester. So, automated testing will never (and shouldn't) replace manual testing of an application. Used appropriately though, automated testing can significantly enhance the testing process.

Automated testing has received a lot of focus lately due to the ever-increasing complexity and size of software applications that require better and faster ways to test. Rather than replace testers, which might be one of the benefits a manager might expect from automated software testing, automated testing can enhance the testing process with increased capabilities. (In fact, at the start of a new test automation project, often more testers, and more technically astute testers, are required, not less.) There are some tedious and time-consuming, yet important, testing tasks that you may choose not to perform on a project due to time and budget constraints. For example, verifying the transfer of large amounts of files or data from one system to another could be prohibitive if done manually; writing code to do that makes it achievable. There are many benefits to enhancing a testing process with automated test scripting. Here are just a few:

- Performing tedious or repetitive manual-testing tasks, such as platform and application start-up, shutdown, and clean-up routines
- Running tests in batch
- Setting a reference to a COM object or .NET class and testing its interfaces
- Attaching to a database for data verification testing
- Accessing and interrogating the Windows Registry
- Creating testing utilities that support the testing process, such as logging and start-up scripts

Within this book, we will explore all of these uses of automated testing and quite a few more.

Technical vs. Nontechnical Testing

Perhaps it would make sense to think that, as the authors of this book, we would be in favor of all testers being technically adept. After all, both of us are not only testers, but also developers

as well. Instead, we think requiring all testers to write code and essentially be programmers in their own right is a mistake. Although it is beneficial and advisable to have some programming-proficient testers on every test team, it should not be required of all testers on the team. This is because, in general, the programmer and the test professional think differently. And they should. The tester who has technical experience as a programmer enhances the project by knowing the kinds of things and situations in which the software developer might be more likely to make mistakes. However, this technical knowledge encourages a person to think analytically and not generally, like a typical user or nontechnical tester would. So the technical tester may miss the errors that nontechnical testers would find, and vice versa. Still, there is no substitute for a professional, knowledgeable, nontechnical test professional with experience; her knowledge and thoroughness cannot be replaced by user-testing only. So you need all three kinds of people: technical testers, nontechnical testers, and user testers.

When to Automate?

Not all testing situations benefit from writing your own test code. In fact, there are many times when it's not a good idea to automate testing. So how do you decide whether and when to automate or not? The decision to automate requires analysis and the definition of boundaries between the automated test plan and the manual test plan. Using a programming language like Visual Basic or C# requires additional careful planning since writing test scripts is essentially software development in its own right and can eat up plenty of time in a schedule.

How do you determine what to test manually and what to test using automated test scripts? While experience is the best judge, there are also some basic questions you can ask yourself prior to embarking on an automated test project. The following three sections will help you determine whether your project is a good candidate for automated testing.

Project and Personnel Issues

Too many times test managers undertake automated testing projects without fully considering the abilities and availability of their personnel. Proper staffing is critical to the success of any project. Here are some important things to consider:

- *What is the scope of the automated testing?* If your goal is to fully automate all tests, then your scope is unrealistic. If you are trying to incorporate automated testing into existing projects or into a new one, then it's best to start with small, manageable goals. For example, you can ask your team to write some simple utilities to support your test project using Visual Basic or C#. This has the added advantage of checking their experience level as well. Not all testers/programmers have the same capabilities!
- *What is the automated testing skill level of your testing personnel?* If automated testing is new to your personnel, then you need to allow time and budget for them to take classes and learn. You will need to add experienced automated testing personnel to your staff *prior* to your first project. The level of experience will determine the level of automation you will be able to undertake. One introductory course in programming will not be enough to enable your testers to undertake a large project. However, they could possibly use some of Visual Studio's tools and wizards to support a test project, and perhaps create and use some simple test utilities.

- *What is the availability of your technically skilled testers?* If you do have technically skilled testers, are they actually available? Many projects start with some experienced members who get pulled off for other projects. This may seem like a no-brainer, but we have seen this situation occur too many times to think it is just an aberration.

Product Issues

Not all applications to be tested are created equal. In general, when you write automated test code yourself, you are working behind the GUI. That is, you are harnessing just pieces of the software. You have to spend some time to determine if this is appropriate for your product. You must do a thorough analysis. The following questions are a short list of things to consider:

- *Is the feature set of the application you are testing relatively stable?* If not, the scripts you write need to change as often as the application changes. You may find yourself spinning your wheels if you start too soon, using up precious budget. Automated testing works best for products that are relatively stable in structure and components.
- *Do you plan to test the UI? Is your product GUI-based?* Some automated testing tools are geared specifically for the GUI. If your project is to test the application's GUI, then certain automated testing tools, commercial or open source, may be a better choice than others. .NET languages can be used for GUI testing to a certain extent, but they require a significant amount of coding to do so. For this reason, in most cases, we would not choose using .NET for extensive GUI-based testing.
- *Does your product have areas where tests are run repetitively, greater than ten times per test?* Any repetitive tasks are candidates for automated testing. Computers perform repetitive tasks well. For example, writing regression tests for high-priority bugs or developing a Build Verification Test (BVT) suite for verification of product robustness after each build are good examples of tests that will be required to be run many times.
- *Will your product need to be compatible with multiple platforms?* Most products need to run on the various versions of Windows: Windows XP, Windows 2K, Windows NT, Windows 95, and Windows 98. There are many other compatibility issues, of course. Automated testing scripts can be written to address some of these compatibility issues.
- *Is your project size and budget large enough to support an automated test piece?* Last but certainly not least, you must consider the additional time and budget required for automated testing. Although automated testing adds a lot to a test project, it can, especially initially, be time-consuming and costly. On a relatively small test project, adding automated test capability may not be worth it.

Note These questions may seem a bit Microsoft-centric. It is true that .NET languages can be a tool for testing mostly Windows-based software systems. There are some exceptions to this, but not many.

Additional Test-Management Issues

Here are a few more management-level questions to ask yourself and your team:

- *Do you have Visual Studio .NET software available to the project?* If not, can you purchase the proper number of licenses and have them in place in time?

Note The .NET Framework can be downloaded and installed for free. With the Framework libraries come compilation tools so that it is possible to write test scripts, i.e., code, in .NET for free. In this book, we focus on using Visual Studio rather than taking this direction because of the simplicity of using the interface. It's a lot more difficult to write the code without Visual Studio .NET unless you're an experienced programmer.

- *Can you insert automated testing without affecting existing testing?* For example, installing Visual Studio .NET and investigating the integration of the test scripting with other tests, such as manual tests or commercial tools, takes time and planning. Can you do this without adversely affecting your total project time and budget?
- *Do you have enough time to analyze requirements as well as code, debug, and maintain test scripts?* Development of automated test scripts is software development and requires all of the same considerations. It's easy to use up time and budget.
- *Who will manage the automated testing for each project and across projects?* An important consideration is to keep and maintain the work done on a project for future use. For example, scripts for logging test results (covered in Chapter 5) can be used in any project. Identify a group or an individual who will be responsible for ensuring that code that can be reused on other projects is maintained for future use.

Managing the testing process is a big topic and an important one. There are many excellent texts available so we won't attempt to compete with them here; check Appendix C of this book for more information on this topic.

Building a Team for Automated Testing

What is the makeup of a good test team? Ideally, automated testing personnel and those members of the team using manual processes should not be kept separate. They can enhance each other's capabilities and they do need to keep in close communication. If you are part of a fairly large company, it is beneficial to have members of the team experienced in different kinds of automated testing: some with applied experience in one or more of the major tools available on the market, and at least a couple of testers who have significant programming experience using Perl, C#, Visual Basic, C/C++, Java, or other languages.

One large company we worked with had a sizeable test group dedicated to automated testing. In this group, they hired personnel experienced in several major tools. On each test project, this team determined which tools, if any, were appropriate for that project, as well as the backgrounds and experience required for the test team. They were integrally involved in the setup for all company test projects and monitored each project as it progressed. This is an

ideal way to proceed. The team was able to keep a repository of test plans and code, as well as a detailed history of the projects and their results. They were instrumental in arranging appropriate training in the tools selected for the project, as well. This model worked quite well for this company, although so far it's the only company we have seen do it quite this way. It takes time and money to set up such a model, but it has many benefits in the long run.

If you are a midsize company with a team of ten testers, the makeup of your test team could be something like this:

- Four to five testers experienced in traditional manual-testing processes
- Three testers experienced in automated test tools such as Segue, Mercury, and Rational
- Two to three testers experienced in software development, at least two of whom could be considered advanced programmers

Note Testers who develop code for use in testing are increasingly gaining titles of their own. This type of testing specialist is sometimes called an *automator* or *Developer-in-Test*—or a *Software Design Engineer in Test (SDE/T)*, as Microsoft calls them.

Of course, there could and should be some overlap. However, don't make the mistake of having nine manual testers but only one person experienced in software development, test tools, and so on. If there is only one person with experience, that person will end up spending all of his time coaching everyone else and getting nothing done. It's best to avoid depending too much on a single person or, nearly as bad, only two people in a test project.

When it isn't possible to form your ideal team with technical programming professional skills onboard immediately, some teams rely on developers to assist them with writing testware. And, if there's sufficient time, alongside these developers, place your most technical testers and get them some training. Hopefully, this will work, until you can compose the team you need.

Test Scripts Are Software

When test engineers write automated test scripts, they must take the time to define and analyze the requirements. This is when the process of writing the scripts actually begins. This process is necessarily interactive as the testers repeatedly run the scripts, then improve and perfect them to meet the ever-changing testing requirements.

After the scripts are working, they must be updated on a regular basis to ensure they work with new versions of the application being tested. A professional software developer would recognize this process of developing and updating automated test scripts as essentially the same one used to develop software applications. So, the writing of automated test scripts *is* software development. The skills needed to be a good automator are similar to those required of a good software developer. In addition to software development skills, automators must also be skilled at testing. To find out more about how to be a good tester, get Cem Kaner's book, *Testing Computer Software, 2nd Edition* (Wiley, 1999). Another good book is Edward Kit's *Software Testing in the Real World* (Addison-Wesley, 1995). (For publication information and other good books on software testing, see Appendix C of this book.)

It is important to recognize that the same rules for developing good software apply to developing good test scripts. Good planning and design are important, as is allowing sufficient time to develop the code and supporting utilities.

Goals of Good Testing Software

Test scripts, like application code in general, should be

- *Readable*: Using standard naming conventions and constants, and creating project standards for code development make code more readable. If code is readable, it can be more readily understood and modified, which makes it easier to work with and to adapt to future projects.
- *Reusable*: Writing routines that can be reused within the same project (and sometimes modified to work within another project, as well) can save time and duplication of effort. Some possibilities may include logging utilities to document test results, front-end or driver routines to make running test suites easy, or specialized utilities that make working with your test target easier, like start-up and shutdown routines.
- *Maintainable*: Writing code that is easy to update is important. You can account for a changing application in many ways, including the use of constants, library files, the Windows Registry, and initialization files.
- *Portable*: Writing suites that can be easily changed makes them portable. For example, don't directly place file paths into your code. Instead place files within the assembly of your application and use relative paths to reference them, or let the user specify locations by providing options for them to set. (We'll see how to do this as we proceed in the book.)

Of course, these are guidelines to follow within reason. As testers, we are far more likely than developers to write simple testware that ends up being thrown away because it's a one-time, special-purpose situation we are addressing. And there are times that copying and pasting can be a good thing. The most important of the four "able" qualities is the ability of your testware to be readable. Reusability, maintainability, and portability should be considered for testware that is intended to be used many times, such as test utilities, and test drivers, i.e., code, that are used as a front end to run other tests.

Code can be written in many different ways. When we review code written by others, we try to determine its readability and, therefore, the ease with which we can maintain or alter it. When we write code for testing purposes, we will try to levy that same mandate on our own work. We will then reap the same rewards as the developers and, in the process, learn a bit about why they do the things they do. Throughout this text, we will emphasize good programming technique. Even when in a hurry on a test project, it is absolutely true that it is just as easy to write code properly as it is to write it poorly.

In this book, you will explore ways to implement these goals for yourself and for the applications you test.

Limitations of Programming Languages for Testing

Because they are not intended as test tools, programming languages usually do not include many of the bells and whistles that most commercial automated test tools, and some open source test tools, do. For example, in all but the Team and Enterprise level editions, .NET languages have no inherent support for bug reporting or test design and documentation as many testing tools have. In .NET's Enterprise and Team editions, they have attempted to help integrate the software development and software testing process using the new Team Test software. We provide an introduction to that software in Chapter 11. If you want these kinds of things in your .NET testware, your company must purchase the Enterprise level software versions of .NET (or your test team will have to write them). If you decide to write this kind of functionality yourself, you might find you have entered the business of test-tool writing instead of the testing business. That will be a time-consuming effort. So, .NET languages should not be considered a substitute for the major test tools, or manual testing, but simply a powerful adjunct to them and all of your test strategies.

Summary

.NET languages are powerful enough to accomplish any testing task as long as you have testers with the skill and ability to write effective code, and test management capable of administrating it effectively. This book is intended to guide you and your team in your efforts.