Grok Code Book

@GrokCodeBook
http://twitter.com/#!/GrokCodeBook

August 19, 2011

Contents

Introduction

The solitary nature of much of computer programming and the number of different avenues that may lead to a career in programming make it difficult for best practices to permiate the field. A responsible author can no longer assume that a person seeking to learn about programming has any background in either computer science or mathematics. All jargon should be introduced in a glossary at minimum; regardless of how many works the author has authored. If a programming text is to build on some assumed knowledge, it must explicitly define said knowledge from the outset, and ought contain a "Read First" section.

Also, the volume of information forthcoming about new technologiesframeworks, toolkits, tooling, etc.drowns out information about programming well. By providing this text free of charge, holding ourselves to the highest standard of authorship and editorship, trying to make this text accessible to all that might read it, and acting as stewards and care-takers of this living document, we hope to raise the bar for published works pertaining to computer programming.

The examples included hereafter are executed in C sharps and Microsoft.Net, but the concepts set forth therein are relevant to all OO languages and frameworks.

Chapter 1

Setting Up Your Programming Environment

1.1 DIY

One click installers for all-in-one Integrated Development Environments (IDEs) can be nice, but they tend to encourage dependency on automata. They also promote, "Well, if there isn't an installer for it, it can't be any good," thinking. It should be noted that often times creating an installer that works for Windows can be prohibitively expensive for authors of tools who do not use Windows as their primary operating system or Visual Studio as their primary development environment.

Free/Libre/Open Source Software (FLOSS) software will be used in the examples hereafter; not because if its price-point, but rather because of the ability of open source software to transfer knowledge to its end user. Want to know the best way to write a task for your build engine? Why not see how the authors of your build engine did it? In order to write code, you must learn how to read code. By extension, to write code well, you must learn how to read code well. It is important to figure out why a piece of code was written the way it was and not just settle for duplicating and succesfully compiling it. Reading the code behind open source software is a cheap and easy way to learn from the masters. Not Ant (NAnt) contains some of the most well written C sharps code out there.

FLOSS software can be a little more difficult to set up, but doing so is worth the extra effort. There are many freely available tutorials online that will walk you through how to configure a software package for first-time use. In the process of setting up software, you will learn a little bit more about your computer runs programs in addition to making your computer a little bit more your own.

The biggest reason for configuring your own environment is personal taste.

You may prefer typing text on the command line in a program like Vim[1] over using an IDE.

When it comes to programmer tools, one size definitely does not fit all. Something like ReSharper[2] may give you unparalelled refactoring and auto-completion support, but starting Visual Studio to change a configuration setting or one line of code is overkill. One theme you should expect to encouter repeatedly in the passages that follow is, "Use the right tool for the job at hand."

> If all you have is a hammer, everything looks like a nail.
> —Bernard Baruch

Equip yourself appropriately.

1.2 Tooling

1.2.1 Build Automation

You should always provide other developers on your team the ability to quickly build your code and test it (with automated tests as well as manually). Testing should always be a part of the review process as exercising the code can and does reveal errors that a visual inspection may miss. Build automation is the mechanism by which you provide other developers this ability.

Tasks within your build automation system can be used to check style, run automated tests, and write reports as well as compiling your code.

Build automation is a critical to the practice of Test Driven Development (TDD). If tests cannot be easily automated, they will not be run. Not out of developer laziness, but for the reason of sheer number. As your codebase grows, your tests can easily number in the hundreds. If you don't have a test harness for automating those tests, they simply won't be run often enough.

NAnt will be used in the examples that follow. MSBuild has some advantages in Windows-only shops—i.e. it is installed by default on Windows Server machines. At the time of writing, it more of a hassle to run all the unit tests for a project from the command -line in MSBuild than in NAnt. Also, there seems to be more of an air of acceptance towards customization—specifically writing one's own build tasks—in the NAnt community.

While there is virtue in the philosophy, "Why write what you can download for free?" sometimes the needs of customization outweigh the benefits of using easily downloadable/purchased software. The philosophy set forth herein is, "Weigh all options without bias, and make the logical choice." Developers are—or at least always should be—part of a team. Sometimes, for the sake of progress, a developer needs to make decisions that compromise on their own happiness. For instance, not using your favorite build tool on a fresh project in a new job, because all the other developers are invested in the use of another tool.

[1] http://www.vim.org/
[2] http://www.jetbrains.com/resharper/

1.2.2 Test Driven Development

TDD should be thought of as a developer's safety net. Trapeze artists use a safety net to keep them from falling to their deaths. The presence of that net allows them to attempt things they might not otherwise. It allows them to overcome their fear to achieve greatness. TDD is first and foremost about giving developers the confidence to improve their code.

NUnit will be used in later examples as more support exists to automate it from the client machine.

1.2.3 Version Control

Things don't always go as planned. Sometimes defects will evade detection in testing. Having a way to quickly revert to a known working state is a luxury no developer can afford to go without. Version control can be a lot like insurance to the uninitiated: you don't know you need it until it is too late.

Version Control Systems (VCSs) also provide a facility called branching. Branching permits experimentation within libraries. Merging allows succesful experiments be kept. Otherwise, the branch can simply be abandoned.

1.2.4 Backup

To prevent lost work, source code and application configuration should be backed up regularly. Distributed Version Control Systems (DVCSs) have the advantage that each developer has a copy of the entire repository. Thus, as changes are passed back and forth between developers, backup occurs organically.

1.2.5 Continuous Integration

Based on the philosophy of making the hard things to do easier by doing them more often, ideally, continuous integration provides a means for keeping software in a ready-to-deploy state.

Continuous Integration (CI) systems wait for a trigger, and then perform a series of tasks once that trigger has been detected such as running unit tests, emailing reports, updating documentation, and even performing version control tasks.

Common triggers are changes to the source code repository, elapsed time, and forced builds.

The CI server has oft been called the heartbeat of the software project.

In the following pages, Buildbot will be used as the CI server.

1.3 A First Build

Download NAnt from `http://sourceforge.net/projects/nant/files/nant/`. As of the time of this writing, it is recommended that you get either the stable

release of 0.90 or the nightly build. Unzip to C:\Program Files\nant-⟨version⟩ or another centrally available location.

Modify your *%PATH%* environment variable to point to ⟨NAnt-Installation-Directory⟩\bin (i.e. C:\Program Files\nant-0.90\bin).

Appendix A

Windows Administrative Tasks

A.1 Setting the PATH Environment Variable

The *PATH* environment variable represents a semi-colon delimited (separated) list of all the paths in your computer where executable files are kept. This allows you to execute an executable from the command line by typing only it's name and not its full path (i.e. *C:\\firefox* as opposed to *C:\\"Program Files\Mozilla Firefox\firefox.exe"*).

You may set the *PATH* environment variable by launching the System Properties window. This can be done by pressing the *Windows* and *Pause* keys simultaneously. This key comibination is denoted by *Windows Pause* (you will see this *button + button* shorthand for key combinations again). A dialog named *System Properties* will show up on-screen. Click the *Advanced* tab. Then click the *Environment Variables* button. If you have administrative access, select the variable named *PATH* (spelling may also be *Path*) in the list called *System variables* , and push the *Edit* button.