Grok Code Book

@GrokCodeBook

September 7, 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 technologies—frameworks, 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# and Microsoft.Net, but the concepts set forth therein are relevant to all OO languages and frameworks.

Chapter 1

Setting Up a Programming Environment

1.1 DIY

One click installers for all-in-one Integrated Development Environments (IDEs) can be nice, but those installers and IDEs 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 a build engine? Why not see how the authors of the build engine did it? In order to write code, one must learn how to read code. By extension, to write code well, one 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# 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 the developer through how to configure a software package for first-time use. In the process of setting up software, developers learn a little bit more about how a computer runs programs in addition to making their computers a little bit more their own.

The biggest reason for configuring a development environment as one sees fit is personal taste. Some developers prefer typing text on the command line

in a program like Vim[1] over using an IDE exclusively.

When it comes to programmer tools, one size definitely does not fit all. Something like ReSharper[2] may afford developers unparalelled refactoring and auto-completion support, but starting Visual Studio to change a configuration setting or one line of code is overkill. One theme that will recur throughout 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

Choose tools wisely.

1.2 Tooling

1.2.1 Build Automation

Developers should always provide other developers on their team the ability to quickly build projects and test them (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 that extends developers this ability.

Tasks within a build automation system can be used to check style, run automated tests, and write reports as well as compiling source 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 a codebase grows, the number of tests for it can easily number in the hundreds. If there isn't 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 one's 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: it's value isn't apparent until it's benefits have been experienced or woefully missed.

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

1.3.1 Get NAnt

Download NAnt from http://sourceforge.net/projects/nant/files/nant/. As of the time of this writing, it is recommended that one 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 the *%PATH%* environment variable to point to ⟨NAnt-Installation-Directory⟩\bin (i.e. C:\Program Files\nant-0.90\bin).

Test NAnt by launching a command window and typing *nant -help* at the prompt. This should display NAnt's help text to the command window.

1.3.2 Create a First Build File

With the open command window, change directories to the desktop.

Create a directory named "GrokCodeBook" using the "mkdir" command.

Open a file for editing with a favorite text-editor (a listing of text-editors is provided in the Tools appendix) and type the following:

```
⟨?xml version="1.0"?⟩
⟨project name="Hello Grok Code Book" default="msg" basedir="."⟩
    ⟨description⟩An example build file⟨/description⟩
    ⟨target name="msg"⟩
        ⟨echo message="Hello!"/⟩
    ⟨/target⟩
⟨/project⟩
```

Save the file as ⟨Desktop⟩\GrokCodeBook\default.build.

From the ⟨Desktop⟩\GrokCodeBook directory, issue the command *nant* . This should result in the display of some banner text and the message "Hello!" on the command window.

1.4 Getting It Under Control

1.4.1 Get Git

Download msysgit from http://code.google.com/p/msysgit/downloads/list, and run the installer.

Launch Git Bash from the desktop icon or the start menu. A terminal window with "MINGW32" in the title bar should display on screen.

A welcome message should be displayed on the terminal window followed by the name of the machine followed by a tilde—a tilde (~) is shorthand for the user's home directory in *NIX systems— followed by the prompt (the $ character).

1.4.2 Making Introductions With Git

At the prompt, type *git config –global user.name "⟨Your Name Here⟩"* . The *–global* option instructs Git to store the supplied values in the *~/.gitconfig* file versus the *.gitconfig* for a specific repository.

Next, type *git config –get user.name* . Whatever value that was specified for *user.name* should be displayed in the terminal window.

Now, supply Git with an email address by typing *git config –global user.email ⟨your-email@your-domain.com⟩* . These detials will help other developer know who to contact should a problem arise (hopefully that eventuality will be avoided with judicious use of TDD).

1.5 A First Repository

From the prompt, issue the command *mkdir ~/Desktop/GrokCodeBook/code* (when working with paths, you should use tab completion for speed). This will create the folder the examples will be contained in. Next, issue the command *cd ~/Desktop/GrokCodeBook/code* . Now the path to the newly created directory should appear before the prompt. The directory *~/Desktop/GrokCodeBook/code* is now the current working directory.

Create a folder for the repository to reside in by issuing the command *mkdir MessageN* . Change directories into *MessageN* .

Issue the command *git init* to create the repository. There is now a Git repository at *~/Desktop/GrokCodeBook/code/MessageN* .

Move the previously created build file to the *MessageN* directory using the command *mv ../../default.build default.build* .

Edit the build file so that it appears as follows:

```
⟨?xml version="1.0"?⟩
⟨project name="MessageN" default="build" basedir="."⟩
    ⟨description⟩A message server built in  C# .NET⟨/description⟩
    ⟨target name="build"⟩
        ⟨echo message="Building MessageN!"/⟩
    ⟨/target⟩
⟨/project⟩
```

Stage the build file for a commit to the repository from the Git Bash window with *git add default.build* . Commit the build file using *git commit -m "Added build file"* .

1.6 Begin With A Test

1.6.1 Get NUnit

Download NUnit from http://www.nunit.org/index.php?p=download, and run the installer.

Ensure that the directory containing *nunit.exe* appears in your *PATH* environment variable. Check this by issuing the command *echo %PATH%* from a Windows command line (won't work in msysgit/cygwin). Alternatively, execute the command *nunit* . This should launch the NUnit Graphical User Interface (GUI) if the directory containing NUnit's binaries is in your *PATH* .

Providing one location for all your downloaded third-party libraries (DLLs), makes it easy for other developers to tell if a DLL has been left behind in a commit. Create a *lib* directory as a sub-directory of *MessageN* . Place the *nunit.framework.dll* located in the NUnit installation directory (use one for the most recent .NET framework targeted—i.e. 2.0.x in rather than 1.0.x) in the *lib* directory.

Create *projects* sub-directory in *MessageN* . Within *projects* , create a directory named *MessageN.Test* .

Now for the first test. In a text editor, add the following to *MessageN.Test\MessageTest.cs*

```
using NUnit.Framework;

namespace MessagN.Test
{
    [TestFixture]
    public class MessageTest
    {
        [Test]
        public void Should_Set_CreationTime()
        {
            Assert.Fail();
        }
    }
}
```

To compile the test and run it, add the following text to *MessageN.Test\default.build*

```
<?xml version="1.0"?>
<!-- Use current working directory as basedir. -->
<!-- Make "test" the target that gets executed when "nant" is run
     with no arguments. -->
<project name="MessageN.Test" default="test" basedir=".">
    <property name="lib.dir" value="..\..\lib"/>
    <property name="build.dir" value="bin"/>

    <description>Tests for the MessageN namespace</description>

    <!-- Need to copy the NUnit DLL to then build directory so
         the CLR automagic can load it when the test is running.
```

```
    -->
    <target name="copyReferences">
        <copy todir="${build.dir}">
            <fileset basedir="${lib.dir}">
                <include name="*.dll" />
            </fileset>
        </copy>
    </target>

    <target name="build" depends="copyReferences">
        <csc
            target="library"
            output="${build.dir}\${project::get-name()}.dll"
            debug="true">
            <sources>
                <!-- Includes all C# files in basedir and its
                     sub-directories. -->
                <include name="**/*.cs" />
            </sources>
            <references>
                <include name="System.dll" />
                <!--Need to specifiy the path to the NUnit DLL-->
                <include
                    name="${build.dir}\nunit.framework.dll" />
            </references>
        </csc>
    </target>

    <target name="test" depends="build">
        <nunit2>
            <formatter type="Plain" />
            <test
                assemblyname="bin\${project::get-name()}.dll"
                appconfig="${project::get-name()}.config"
            />
        </nunit2>
    </target>

</project>
```

Since NAnt was built against an earlier version of NUnit, the runtime needs told which version of NUnit to use. This is done by redirecting the assembly version.

In a text editor, save the following to *MessageN.Test\MessageN.Test.dll.config*

```
<?xml version="1.0"?>
```

```
<configuration>
    <runtime>
        <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
            <dependentAssembly>
                <assemblyIdentity
                    name="nunit.framework"
                    publicKeyToken="96d09a1eb7f44a77"
                    culture="Neutral" />

                <!-- The <nunit2> task was built targeting
                     version 2.2.8.0 of NUnit. This means we need
                     to redirect the assembly from 2.2.8.0 to the
                     newer version.

                     N.B.: Make sure to use the version of NUnit
                     you have installed for "newVersion", and not
                     necessarily 2.5.10.11092!
                -->
                <bindingRedirect
                    oldVersion="2.2.8.0"
                    newVersion="2.5.10.11092" />
            </dependentAssembly>
        </assemblyBinding>
    </runtime>
</configuration>
```

Now run the test by issuing the command *nant* from the *MessageN.Test* directory.

The *nunit2* NAnt task should print a test report to the console reporting one failed test.

Remembering that a development environment should be tailored to suit the individual developer, the programs installed above will serve as a solid foundation for a development environment. Further programs and assemblies will be downloaded in other chapters as needed.

Chapter 2

Keeping It Classy

2.1 Why OO?

Countless examples of structured programing in an object oriented language presented as proper technique exist both online and in programming texts. This is the equivalent of telling someone to buy a set of golf clubs, but only ever giving them instruction on how to use the seven iron.

Object orientation allows for the separation of concerns and facilitates refactoring. Two methodologies that allow changes to be introduced to software systems relatively easily.

Going forward, coding to interfaces will be presented as the appropriate way to implement object oriented systems. This approach allows for the development of loosely coupled and highly cohesive systems. Coding to interfaces also allows the developer to prioritize development by factors like risk (high risk features should be tackled first—it's better to know something can't be done sooner rather than later) rather than requiring a class to wait on all it's components to be concretely implemented before starting work on that class. Coding to interfaces also allows for division of labor and collaboration. Perhaps most importantly, coding to interfaces allows objects to be tested in isolation.

2.2 The Assignment

The system to be built is a generic message queing and delivery service along with a few clients for scheduling and sending messages. The client wants this because he would like to write messages when he thinks of them and would like for those messages to be sent at the oportune moment. The client has expressed the most interest in being able to schedule email and Facebook updates.

Back at the office, after a whiteboard session, the following interfaces are agreed to for the project to be built upon.

```
using System;
```

```
using System.Collections.Generic;

namespace MessageN
{
    public interface IUser
    {
        string Name {get;}
    }

    public interface IMessage
    {
        string Text {get;}
        DateTime CreationTime {get;}
        IUser From {get;}
    }

    public interface IDeliveryAttempt
    {
        IMessage Message {get;}
        DateTime Time {get;}
        bool Success {get;}
        string Error {get;}
    }

    ///<summary>
    ///Responsible for getting the message to its destination.
    ///</summary>
    public interface IMessageRouter
    {
        IMessage Message {get;}
        List<IAttempt> Attempts {get;}
        bool Active {get;}
        void Send(IMessage message);
    }

    ///<summary>
    ///The communication channel for the message to be sent along.
    ///</summary>
    public interface IMessageTransport
    {
        bool InService { get; }
        void Send(IMessage message);
    }
}
```

Now that a starting point has been established, time to make the first test

pass.

Chapter 3

Extensibility Through Configuration

Extensibility is a delicate subject. Big Design Up Front is a perjorative term coined to describe the fool-hardy attempt to completely specify a software system in full detail before writing a line of code. This approach strives to prevent waste by not working on the problem until it has been completely defined. The problem is that the only absolute in software is uncertainty. Implied requirements usually can't be discovered until a customer has seen the software in action and has the opportunity to say, "That isn't what I meant."

Designing for extensibility where it isn't required or isn't high-priority can easily lead to over-engineering the software. The ways to prevent over-engineering are to deliver working tested software ASAP rather than a huge feature set (contracts can be extended later if need be and only successful projects deserve extension), to think and reason, and to continually check priorities with the customer (communicate as much as possible with as much efficiency as possible).

Preferring composition (using components that perform specific functions such as sending a message) over inheritence is one development strategy that aims to reduce over-engineering and promote possible future re-use by avoid unnecessary abstraction. Creating base classes is an activity few people do well. Following the Single Responsibility Principle can also help keep developers focused on just the task at hand.

From discussions with the customer for the messaging service, it can be determined that extensibility is a priority in at least one case: the customer would like to be able to schedule both email messages and Facebook updates. Scheduling Facebook messages has also been prioritized over scheduling email.

As a starting point, here's a failing test for a list of transports retrieved from configuration:

```
using NUnit.Framework;

namespace MessagN.Test
```

```
{
    [TestFixture]
    public class MessageTransportConfigurationSectionTest
    {
        [Test]
        public void Should_Return_List_Of_Transports_From_Transports()
        {
            Assert.Fail();
        }
    }
}
```

Running the build file from the *MessageN.Test* directory, you should see a message that the *Should_Return_List_Of_Transports_From_Transports* test failed.

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 a computer where executable files are kept. This allows the execution of 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"*).

The *PATH* environment variable may be set 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 administrative priveliges are available, select the variable named *PATH* (spelling may also be *Path*) in the list called *System variables* , and push the *Edit* button.

The *Edit System Variable* dialog will display on screen. In the *Variable value* box, paste the path to the program (full path to the directory it is in) at the end of the list of paths. Ensure there is a semi-colon separating new path from the one before it. Click the *OK* button to close the *Edit System Variable* dialog. Click *OK* to close the *Environment Variables* dialog, and close the *System Properties* window.

N.B.: any open command windows while modifing your PATH variable will not reflect the changes to it. Be sure to close and reopen all command windows when editing the PATH.

A.2 Launching a Command Window

Launching a command window can be done by typing "cmd" into the *Run* box and pressing *Enter* . The *Run* box can be found from the *start* menu on Windows (look for the text "Run..." on the start menu). Alternatively, the *Run* box may be launched by the pressing *Windows + R* key combination on the keyboard.

A.3 Changing Directories

To change the current working directory (cwd) in a command window, issue the command "cd" followed by the name of directory to be made the new current working directory.

For example: *C:\Users\GrokCodeBook⟩ cd Desktop* .

The above command would take you from the user's home directory (the user's profile directory—C:\Users\GrokCodeBook if you are the user "GrokCodeBook") to the Desktop in Vista or Windows 7.

A.4 Create a Directory

The "mkdir" command can be used to create a sub-directory from a directory if have sufficient permissions have been granted.

For example: *C:\Users\GrokCodeBook\Desktop⟩ mkdir GrokCodeBook* .

The above command would create the directory C:\Users\GrokCodeBook\Desktop\GrokCodeBook.

A.5 Tab Completion

Tab completion functionality saves the user of the command window from having to type the full name of a directory when working with paths.

For example, when launching a command window, the command window will default to the user's profile directory. To change directories to the Desktop, one can type *cd D* then press the *Tab* key. This will complete the path. "cd Desktop" should be visible after the prompt if there are no other directories that begin with a "D" in the user's profile directory.

If there are sub-directories with similar names (for instance "Desktop" and "Downloads") in the current working directory, the user need only type enough of the path ("De" or "Do" in the case of "Desktop" and "Downloads") for the system to be able to discriminate between the two.

If the system can't tell the difference (for example: only "D" was typed from the previous example), in Windows, typing the *Tab* key successive times will toggle between subdirectories with the same beginning (on *NIX systems, a list of sub-directories with the same beginning will be printed to the command window).

Appendix B

Tools

B.1 Text-editors

B.1.1 CLI

- gvim
- edit (ships with Windows)

B.1.2 GUI

- Notepad++
- GVim
- notepad.exe (ships with Windows)

Glossary

Continuous Integration An automated process whereby source code is built and tested continuously, and reports on the outcome of the build/test process are sent to the developers.. 21

Distributed Version Control System A distributed form of Version Control System. Whole repositories exist on the developer's machine rather than on a centralized server. Changes are sent between developers rather than client to server.. 21

Free/Libre/Open Source Software "Free/Libre/Open-source software (FLOSS) is liberally licensed to grant the right of users to use, study, change, and improve its design through the availability of its source code" [1]. 21

Graphical User Interface Programs that are interacted with via interfaces that are rendered to the screen rather than through typing into a terminal window.. 21

Integrated Development Environment "An integrated development environment (IDE) (also known as integrated design environment, integrated debugging environment or interactive development environment) is a software application that provides comprehensive facilities to computer programmers for software development"[2]. 21

NAnt A FLOSS build automation tool written in C# [3]. 21

Test Driven Development A software development methodology whereby the developer writes a failing test, writes code to make the test pass, and refactors the test as needed to prevent duplication of code and brittleness. The tests are automated with the use of an automated test harness.. 21

Version Control System A software tool for managing changes made to source code. Make branching and merging possible.. 21

[1] http://en.wikipedia.org/wiki/Free_and_open_source_software
[2] http://en.wikipedia.org/wiki/Integrated_development_environment
[3] http://nant.sourceforge.net/

auto-completion "A feature [of software programs] that suggests text automatically based on the first few characters that a user types."[4]. 2

branching Creating a copy *abranch* of files under version control for the purpose allowing the files of the new branch and the files they were branched from to be modified independently of each other. . 3

Buildbot A FLOSS distributed build and CI tool written in Python.[5]. 3

merging Using a VCS to intelligently combine the files of branch being merged in with the files of the target of the merge. . 3

NUnit A FLOSS unit testing tool written in C# .[6]. 3

refactoring "The process of modifying source code for the purpose of improving its readability and maintainability while retaining the program's functionality and behavior." [7]. 2

text-editor A program that allows it's user to edit the text of a file. Useful for quick editing as they are typically much lighter weight than full-blown IDEs, therefore they startup and shutdown much faster.. 4

[4] http://msdn.microsoft.com/en-us/library/dd921717(v=office.12).aspx
[5] http://trac.buildbot.net/
[6] http://www.nunit.org/
[7] http://developer.apple.com/library/ios/#documentation/DeveloperTools/Conceptual/Xcode_Glossary/20-Glossary/Glossary.html

Acronyms

CI Continuous Integration. 3, 19

DVCS Distributed Version Control System. 3

FLOSS Free/Libre/Open Source Software. 1, 19

GUI Graphical User Interface. 6

IDE Integrated Development Environment. 1, 2, 19

NAnt Short for "Not Ant"[8]. 1, 2, 4

TDD Test Driven Development. 2, 3, 5

VCS Version Control System. 3, 19

[8] http://nant.sourceforge.net/release/latest/help/introduction/fog0000000042.html