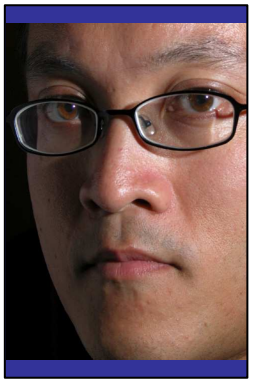
practical eye

FOR THE .NET GUY

Issue 1: October 21, 2003

Why do I do this?



It was 4:30 am on September 15, 1983. I was hunched over my desk in the basement of my parent's house in Scarborough, Canada. Another full day of Grade 10 beckoned at Agincourt Collegiate, yet I hadn't gone to bed yet. I *couldn't* go to bed yet.

I basked in the pale blue glow of my Commodore 64's 1701 monitor. I was doggedly inserting NOP statements into a critical piece of timing code. I was attempting to do something that had never been done before: asynchronously transmit two bits at a time between my 1541 floppy drive and my Commodore 64.

Moments later, I tested my creation. A wave of euphoria swept over me as I saw the results; I had just created *the* key piece of code that would boost the performance of Commodore's 1541 disk drive by over 500%!

I realized then and there that I write software because I can't *not* do it. It's a creative outlet for me, much like photography is. What's more, I feel compelled to share my creativity with others.

I have created this newsletter to share my latest research and ideas with you. In each issue, you will find an article that will help your software development team write better code, faster. To help ease the stress from long hours of coding, I also present you with a photograph from my portfolio.

I invite you to call me to discuss how together, we can help your team write better code, faster. You can reach me directly at 416-716-3718 or jlam@iunknown.com.

Cheers,

John Lam

Pipeline

I'm putting the finishing touches on the inaugural issue of Practical Eye for the .NET Guy four weeks after my son, **Matthew**, was born. He's a wonderful bundle of joy and a blessing to our new family.

If you have a group of developers in the Greater **Toronto** Area, and you want to get a **free** two-hour lunchtime briefing on **what was presented at the PDC**, please contact me ASAP and I'll make arrangements to come out to your company!

Microsoft is getting ready to unveil **Avalon**, **Indigo**, **Whidbey**, and **Yukon** at this fall's Professional Developers Conference in Los Angeles. I have just begun working with Microsoft on a **key piece of technology** that will be used extensively by all developers working on the new platform. Immediately after the **PDC**, I'll be publishing a special edition of this newsletter that examines this badly needed product.

This newsletter is generated using a **publishing framework** that I created using **open source** tools. Right now I'm working out the kinks in the system, and I'll be opening up the CVS project for others to collaborate on improving it in the near future. A future installment of this newsletter will discuss the details of my publishing framework.

You will find two articles in this issue. The first article, **software assembly lines**, looks at the parallels between software and hardware assembly lines. It also motivates why software assembly lines are key components of high-quality software products. The second article, **a practical guide to NAnt**, is a quick start guide that will help you implement a software assembly line in your own projects.

Software assembly lines

I spent the week of August 24, 2003 teaching a great group of students at Hewlett Packard's facility in Corvallis, OR. My students that week were a bunch of nice folks *who build assembly lines for a living*. That's right, I taught ASP.NET to a bunch of ladder-logic PLC (Programmable Logic Controller) developers! I only hope that they got as much out of the class as I learned from them over conversations at lunchtime and after class. This article describes what I learned about the parallels between what they do for a living and software builds.

Every time you build your application, you are submitting your raw materials (source code) to a machine (a compiler) that generates your product. This is strikingly similar to a modern assembly line for physical products. They take a number of raw materials or intermediate products and combine them together to create the final product.

Before my HP visit, I had very little appreciation for the amount of effort that goes into manufacturing a $30 dollar product. That week, I learned a lot about how ink jet pens are made. An ink jet pen is a print head + ink cartridge combination that is used in most of HP's ink jet printers.

After class one day, Steve and Patrick took me on a guided tour of one of their assembly lines. The line they showed me measured some 400 feet in length and was capable of running 24 x 7 to create 1 million ink jet pens per month. It was a fully automated assembly line; no human hands touch an ink jet pen throughout the entire process. I was so in awe of the entire process. And the line that they showed me was only the final assembly line; there were two other lines that are responsible for

fabricating other parts that go into creating one of these pens.

Software vs. hardware assembly lines

A good software build process has much in common with the assembly line that I saw. First, and most important, there is no human intervention in the build process unless something goes wrong. In a software build, breakage can be due to any number of things; some code could fail to compile, one or more of your smoke tests could fail, your version control server could be offline. In an assembly line, it could be due to a failure in one of the robots, or a bad batch of materials from a supplier, or a failure in one of the computers that control the line.

Next, a good software build process should be self-diagnosing. It must contain tests that are designed to quickly diagnose a build failure. In a small build, these tests could very well be your unit test suite. In a large build, it is not practical to run your unit tests during the build since it would take far too long to complete the build. Instead, a subset of tests, called smoke tests, is run to test whether the major subsystems in the code function correctly. In the assembly line that I saw, they had both types of tests. They had the equivalent of smoke tests; at the end of the line, *_every_* pen that comes off the line is test fired. They had the equivalent of unit tests; a certain number of pens are pulled aside and test printed using real printers to guarantee that their overall quality is up to spec. Very impressive.

Finally, a good software build process should collect information about what is built. The build should be instrumented in such a way that the output of various pieces of the build (e.g. the smoke tests) can be collected into one or more reports. These report(s) can then be sent via email to all interested parties at the end of the build. HP invested a lot in their tracking system for their pens; statistics for every single pen is tracked by the production

controller. Each pen is identified by a unique identifier, so in case there is a defective pen, the support engineer can call up the entire history of that pen. It's amazing what you can learn about your product when the cost of storage of the information is effectively zero.

Your first software assembly line

So, what does this mean for your application? If you're using Visual Studio .NET to build your application, you're missing out on many of the benefits provided by modern software assembly lines. The solution-based build system in Visual Studio .NET is fine for private developer builds. This is especially so since your private builds are created as part of the edit-compile-debug cycle of development, where F5 (Debug|Start) is such an important part of the user experience. Unfortunately, as many people have found, it is totally inadequate for your daily builds, which typically involve integrating code created by many different developers.

For your daily builds, you *need* a software assembly line. Here's a list of things that your first build script should do for you when you type "build" from the command prompt:

- Get the latest version of the source files from your source repository.
- Scrub all of the output and intermediate directories used by the build script.
- Generate a build number that identifies this build.
- Perform a full compile, as opposed to an incremental build.
- Run your smoke tests. If this is a small application, it is likely that your smoke tests are your unit tests. In a larger application, your smoke tests will be independently developed.
- If your smoke tests are successful, label the source code files with the build number.

- Optionally check in all of the output files from your compiler. For builds that will be released to parties outside of your dev team (like your QA department) you should also check in files like PDB (program database) and MAP (linker map files). This will help you pinpoint where your application crashed if all you receive is an address.
- Build your installation media. If you want other people to test your application, you need to make their user experience as painless as possible. You're going to need an installer sooner or later, so it's best to build it up front.
- Send an email notification to all members of the team summarizing the outcome of the build, along with a link to a detailed report if they are interested in the details of what happened.

Tools

There are a number of tools that you can use to create your software assembly line. Some folks get by with a bunch of custom VB/JavaScript + make files. Others (like Microsoft) use a tool called `build.exe` that ships with the DDK. However, there is one tool that has received a lot of attention recently: Ant.

Ant was the brainchild of James Duncan Davidson, and legend has it that it was created during a plane ride. It's a remarkably powerful tool because it attempts to define as little as possible. Ant is a framework for organizing *tasks* into *targets*, and determining the order that targets should be built. A task can be as simple as creating a directory, or as complex as generating the data access layer of your application.

Ant is remarkably extensible; anyone can create a new task and incorporate it into the framework. Its extensibility

model has been praised as *viral*, because it encourages folks to add onto what it provides out of the box.

A vibrant community has created and nurtured Ant. Much has been written about Ant, and there are a number of excellent books on Ant development. Visit my bookshelf (http://www.iunknown.com/bookshelf.html) to see my Ant book recommendations.

Ant has also spawned a number of clones. The next article in this newsletter explores a port of Ant by the .NET community called NAnt. My goal is to get you writing your first NAnt build script within 30 minutes of reading the article.

Feel free to forward this article newsletter to others. Send your comments and feedback directly to me at jlam@iunknown.com. You can subscribe to my newsletter at http://www.iunknown.com.

A Practical Developers' Guide to NAnt

Ant was created and used most extensively by the Java community. While Ant can be used to build .NET applications, the .NET community decided to create its own build tool called NAnt. NAnt started life as a port of the Ant codebase, but as development progressed its code shed its Java roots to exploit.NET-specific features such as custom attributes. The spirit of Ant, however, continues to live on in NAnt.

In this section, I'll walk you through a series of steps that most developers follow when creating their first build script. By the end of this guide, you will have a build script that will build an EXE and a DLL using both Debug and Release settings.

Step 1

Let's begin by compiling a simple "Hello, World" application:

```
using System;

public class App
{
  public static void Main()
  {
    Console.WriteLine( "Hello, World" );
  }
}
```

Here's the NAnt build script that we will use to compile app.cs:

```
<project name="App" default="compile">

  <target name="compile">
    <csc target="exe" output="app.exe"
debug="true">
      <sources>
        <includes name="app.cs"/>
      </sources>
    </csc>
  </target>

</project>
```

As you can see, NAnt build scripts are written in XML. NAnt scripts can be easily maintained using a text editor, and more importantly can be easily *generated* using tools. Let's take a quick tour of this simple build script.

A build script contains exactly one *project*. Projects are collections of *targets*. Targets represent things that are generated by the build. For example, each assembly in your project would be generated by its own target. Targets in turn are collections of *task*s. A task represents a single unit of work that is done by the build. It could be as simple as creating a directory, or as complex as generating the data access layer of your application.

To run this build script, invoke NAnt from the directory that contains your build script. NAnt automatically looks for a file with a .build extension and runs it.[1] When NAnt executes your build script, it must try and build at least one target. As you can see by examining the build script, the project element's `default` attribute identifies the default target.[2]

The `compile` target invokes the C# compiler to compile `app.cs` into an executable assembly called `app.exe`. The C# compiler is invoked by the `csc` task, and the inputs to the task are defined by the `sources` element.

Step 2

The `sources` element is quite flexible. In the next step, we're going to write a somewhat more complicated application. The new application contains two source files, `app.cs`:

```
using System;

public class App
{
  public static void Main()
  {
    Console.WriteLine( "1 + 1 = {0}",
Calculator.Add( 1, 1 ) );
  }
}
```

and `lib.cs`:

```
public class Calculator
{
  public static int Add( int x, int y )
  {
    return x + y;
  }
}
```

Since both source files are to be compiled into the same assembly, we only need to modify our build script so

that it compiles both `app.cs` and `lib.cs`. Here's the updated build script:

```
<project name="App" default="compile">

  <target name="compile">
    <csc target="exe" output="app.exe"
debug="true">
      <sources>
        <includes name="*.cs"/>
      </sources>
    </csc>
  </target>

</project>
```

As you can see, I've modified the sources element so that I compile *all* of the `.cs` files from the source directory. Wildcards are often used in NAnt build scripts because it simplifies maintenance of the build scripts. It is unlikely that you will have rogue source files in your source directories, so it's far more reasonable to use wildcards instead of explicitly listing all of the source files to be compiled. If you choose not to use wildcards, you will have to do some additional work to keep your Visual Studio .NET solution files and your NAnt build scripts synchronized.

Step 3

Next, we're going to generate two different assemblies from the source files introduced in the previous step. Here's the revised build script:

```
<project name="App" default="compile">

  <target name="compile">
    <csc target="library" output="lib.dll"
debug="true">
      <sources>
        <includes name="lib.cs"/>
      </sources>
    </csc>
    <csc target="exe" output="app.exe"
debug="true">
      <sources>
        <includes name="app.cs"/>
      </sources>
      <references>
        <includes name="lib.dll"/>
      </references>
    </csc>
  </target>

</project>
```

As you can see, the `compile` target now contains two `csc` tasks. The first task compiles `lib.cs` into an assembly called `lib.dll`. The second task compiles `app.cs` into an assembly called `app.exe`.

There is a new element introduced in this step: `references`. The code in `app.cs` needs to reference

compiled code that lives in `lib.dll`. The references element lets us tell the C# compiler to reference `lib.dll` when it compiles `app.cs`.

Notice that we must compile `lib.dll` before we compile `app.exe`. In this case we explicitly instruct NAnt to compile it first by placing it before app.exe.[3]

Step 4

Build scripts commonly generate both release and debug builds of the product. In this step, we're going to see how we can use two different targets to generate either a release or a debug build:

```
<project name="App" default="debug">

  <target name="debug">
    <csc target="library" output="lib.dll"
debug="true">
      <sources>
        <includes name="lib.cs"/>
      </sources>
    </csc>
    <csc target="exe" output="app.exe"
debug="true">
      <sources>
        <includes name="app.cs"/>
      </sources>
      <references>
        <includes name="lib.dll"/>
      </references>
    </csc>
  </target>

  <target name="release">
    <csc target="library" output="lib.dll">
      <sources>
        <includes name="lib.cs"/>
      </sources>
    </csc>
    <csc target="exe" output="app.exe">
      <sources>
        <includes name="app.cs"/>
      </sources>
      <references>
        <includes name="lib.dll"/>
      </references>
    </csc>
  </target>

</project>
```

As you can see, I took the easy way out when I created this build script. I used *editor inheritance*. That is, I simply duplicated the existing `compile` target and named the two new targets `debug` and `release`. In the `release` target, I simply removed the `debug` attribute from the `csc` tasks. Using this build script, you only need to type `nant debug` or `nant release` to generate debug and release builds respectively.

Step 5

Duplication is evil. In the previous step, I created two targets that differ only by how they set the debug attribute in their csc tasks. Fortunately, NAnt lets us define properties and use them to set the value of attributes in tasks. Here's an updated build script that uses properties:

```
<project name="App" default="debug">

  <target name="compile">
    <csc target="library" output="lib.dll"
debug="${debug}">
      <sources>
        <includes name="lib.cs"/>
      </sources>
    </csc>
    <csc target="exe" output="app.exe"
debug="${debug}">
      <sources>
        <includes name="app.cs"/>
      </sources>
      <references>
        <includes name="lib.dll"/>
      </references>
    </csc>
  </target>

  <target name="debug">
    <property name="debug" value="true"/>
    <call target="compile"/>
  </target>

  <target name="release">
    <property name="debug" value="false"/>
    <call target="compile"/>
  </target>

</project>
```

Notice that I've created a new target called compile. This target assigns the string "${debug}" to the debug attribute. The NAnt runtime substitutes the value of a property called debug in place of the string ${debug}.

The existing debug and release targets were stripped of all of their functionality. The revised targets do exactly one thing: they set the value of the debug property and call the compile target. This is a procedural style of programming which is very natural to developers using NAnt for the first time.

Step 6

A clean target is commonly found in build scripts. It ensures that the output directories for the build script are scrubbed clean of any files that might be left over from previous builds. Here's an updated build script:

```
<project name="App" default="debug">

  <target name="clean">
    <delete dir="bin"/>
    <mkdir dir="bin"/>
  </target>
```

```
  <target name="compile">
    <call target="clean"/>
    <csc target="library"
output="bin\lib.dll" debug="${debug}">
      <sources>
        <includes name="lib.cs"/>
      </sources>
    </csc>
    <csc target="exe" output="bin\app.exe"
debug="${debug}">
      <sources>
        <includes name="app.cs"/>
      </sources>
      <references>
        <includes name="bin\lib.dll"/>
      </references>
    </csc>
  </target>

  <target name="debug">
    <property name="debug" value="true"/>
    <call target="compile"/>
  </target>

  <target name="release">
    <property name="debug" value="false"/>
    <call target="compile"/>
  </target>

</project>
```

Notice that I've created a new target called `clean` that contains two tasks. These tasks delete and then re-create a `bin` directory, where all of the generated assemblies will be placed. I've also modified the `compile` target to call the `clean` target, and to place the generated assemblies into `bin`.

This build script actually contains a bug. It will fail when run, since the `clean` target attempts to delete a `bin` directory that doesn't exist yet. This is caused by task failures aborting the build script by default. This is by design, but can be overridden for non-critical tasks, as we'll see in the next step.

Step 7

It is often useful to build both `debug` and `release` targets during your daily build. This helps to uncover subtle bugs that exist only in your release builds, especially if you run your unit tests on both sets of binaries. Here's a build script that contains a new target called `both`, which builds both the `debug` and `release` targets:

```
<project name="App" default="debug">

  <target name="clean">
    <delete dir="${outputdir}"
failonerror="false"/>
    <mkdir dir="${outputdir}"/>
  </target>

  <target name="compile">
    <call target="clean" force="true"/>
    <csc target="library"
output="${outputdir}\lib.dll"
debug="${debug}">
```

```
      <sources>
        <includes name="lib.cs"/>
      </sources>
    </csc>
    <csc target="exe"
output="${outputdir}\app.exe"
debug="${debug}">
      <sources>
        <includes name="app.cs"/>
      </sources>
      <references>
        <includes
name="${outputdir}\lib.dll"/>
      </references>
    </csc>
  </target>

  <target name="debug">
    <property name="debug" value="true"/>
    <property name="outputdir"
value="bin\debug"/>
    <call target="compile" force="true"/>
  </target>

  <target name="release">
    <property name="debug" value="false"/>
    <property name="outputdir"
value="bin\release"/>
    <call target="compile" force="true"/>
  </target>

  <target name="both">
    <call target="debug"/>
    <call target="release"/>
  </target>

</project>
```

You run into a problem whenever you want a target to be built more than once in a script. NAnt is designed so that targets are built exactly once, since problems can arise when targets are built multiple times during a build. In a best-case scenario, you are wasting CPU cycles. In a worst-case scenario, you are introducing bugs in your code due to build order dependencies.

In our scenario, however, we want the `both` target to build both the `debug` and `release` targets. While these are different targets, they call common targets: `compile` and `clean`. To override NAnt's default behavior, notice that I've added a `force` attribute to the `call` tasks in the `debug`, `release`, and `compile` targets.

Remember how the `delete` task failed in the previous step? We can override NAnt's default behavior of aborting the build when a task fails by adding a `failonerror="false"` attribute to the offending task. Notice that I've added this attribute to the `delete` task in the `clean` target.

What's Next?

This guide has given you a quick start to writing NAnt build scripts. You should spend some time working on build scripts for your own application. You should, without much additional effort, be able to write a build script that compiles all of the targets in your application.

As we saw in the Software Assembly Lines article in this newsletter, however, your build script has to do much more than just compile your source code. When you have your build script compiling your source code successfully, your next step is to integrate your source control system into your software assembly line.

Extending NAnt

You can extend NAnt by writing custom classes using the .NET programming language of your choice. There is a vibrant community of developers that have built custom NAnt tasks. These tasks provide:

- Integration with a number of different source control repositories.
- Integration with different .NET programming languages. C#, VB.NET, and Managed C++ are supported out of the box.
- Integration with unit testing harnesses such as NUnit.
- Limited integration with Visual Studio .NET solutions.

Make sure that you take a look at the NAntContrib tasks that ship with the main NAnt distribution.

To whet your appetite, here's a simple C# NAnt class that writes the name of the current project to standard output:

```
[ TaskName("mytask") ]
public class MyTask : Task
{
  protected override void ExecuteTask()
  {
    this.Log(Level.Info, "Hello, World");
  }
}
```

There is much to writing custom NAnt tasks than I have space for in this newsletter. Look for more coverage on custom NAnt tasks in a future article.

Feel free to forward this article newsletter to others. Send your comments and feedback directly to me at jlam@iunknown.com. You can subscribe to my newsletter at http://www.iunknown.com.

---

[1] If there is more than one .build script in a directory, NAnt raises an error since it doesn't know which one to build. In these cases you must identify the name of the build script as the first parameter to NAnt e.g. `nant app.build`.

[2] It is not an error to have a build file without a default target. You can always identify the target that you want to build as an optional parameter on the command line. This means that you must either identify the target e.g. `nant compile` or `nant app.build compile` if *compile* is your target.

[3] Note that I am generating two assemblies in a single target to simplify the discussion. In most build scripts, each assembly is generated by a separate target.

Final Frame

Each evening during my class at HP, I longed to see the ocean. On evenings where I had more energy, I drove over 100 miles in search of the evening light. I found this magnificent scene at the Yaquina Point lighthouse, just north of Newport, OR. The egrets give a sense of scale to the cliffs which aren't as imposing as the image makes them look at first glance.