Foreword

I have to confess that I had absolutely no idea that Ant, the little build tool that could, would go as far as it did and make such a mark on the Java developer community. When I wrote the first version of Ant, it was a simple hack to help me solve a cross-platform build problem that I was having. Now it's grown up and being used by thousands of developers all over the planet. What's the magic behind this? How did this little program end up being used by so many people? Maybe the story of how Ant came to be holds some clues.

Ant was first written quite some time before it was checked into Apache's CVS servers. In mid-1998, I was given the responsibility at Sun Microsystems to create the Java Servlet 2.1 specification and a reference implementation to go with it. This reference implementation, which I named Tomcat, was to be a brand new codebase, since the previous reference implementation was based somewhat on code from the Java Web Server, a commercial product that was migrated from JavaSoft to iPlanet. Also, the new implementation had to be 100% Pure Java.

In order to get the 100% Pure Java certification, even for those of us working on the Java Platform at Sun, you had to show Key Labs (an independent certification company) that you could run on three different platforms. To ensure that the servlet reference implementation would run anywhere, I picked Solaris, Windows, and the Mac OS. And not only did I want Tomcat to run on these three platforms, but I wanted to be able to build and develop on all three platforms as well as on Linux. I tried using GNU Make. And shell scripts. And batch files. And God knows what else. Each approach had its own unique problem. The problems stemmed from the fact that all of the existing tools had a worldview rooted in building C programs. When these practices were applied to Java, they worked, but slowly. Even though Java programs themselves can perform well, the startup overhead associated with the Java Virtual Machine is lengthy. And when Make creates a new instance of the VM with every file that needs to be compiled, compile times grow linearly with the number of source files in a project.

I tried many approaches to write a make file that would cause all of the source files in a project that needed to be recompiled to be passed to `javac` in one go. But, no matter how hard I tried, and how many Make wizards I consulted with, I couldn't get an approach that would work the same way on multiple platforms. I got very, very tired of the *!&#$%#ing* tab formatting of make files. As much as I've been a proponent of Emacs in my life, any tool that requires Emacs to properly write its files so that you can make sure that no unintended spaces creep in should not be tolerated.[1]

It was on a flight back from a conference in Europe that I finally got fed up once and for all of trying to create some make file that would work the same way everywhere. I decided to "make" my own tool: one that would examine all the Java source files in a project, compare them with any compiled classes, and pass the list of sources that needed to be compiled directly to *javac*. In addition, it would do a couple of other things like stuff all the classes into a JAR file and copy some other files around to make a distributable version of the software. In order to ensure that things would work the same way on every supported platform, I decided to write the tool in Java.

[1] I've been told that the original designer of the make file format knew after the first week that the tab thing would be a problem. But he already had dozens of users and didn't want to break compatibility.

A few hours later, I had a working tool. It was simple, crude, and consisted of just a few classes. It used the functionality of `java.util.Properties` to serve as its data layer. And it worked. Beautifully. My compile times dropped by an order of magnitude. When I got back to the states and tested it out on Solaris, Linux, and Mac OS, it worked just fine on all of them. Its biggest problem at that time was that the number of things it could do was limited to compiling files and copying files — and that this functionality was hardcoded.

A few weeks later I showed the tool, which I had named Ant because it was a little thing that could build big things,[2] to my friend Jason Hunter (author of *Java Servlet Programming*, published by O'Reilly). Jason thought that it was a decent enough tool, but didn't really think it was a big deal. That is, until I mentioned that I was thinking of using Java's reflection abilities to provide a clean way to extend Ant's abilities so that a programmer could write their own tasks to extend it. Then the light bulb went off over his head and I had my first Ant user as well as evangelist. Jason also has an uncanny ability to find a bug in any piece of software within moments and helped me stomp out quite a few problems.

Once the reflection layer was in place, I wrote a few more tasks and Ant became useful to other groups at Sun. However, the build file format was getting a bit bulky. Properties files don't really lend themselves to hierarchical grouping well, and with the introduction of tasks came the idea of targets (collections of tasks). I played around with a few different ways of solving the problem, but hit on the solution when I was on another flight back from Europe. This solution structured the project-target-task hierarchy to follow an XML document hierarchy. It also leveraged the reflection work I had done earlier to associate XML tag names with task implementations.

Evidently I do my best coding while flying over the ocean. I wonder if there's something about the increased radiation at high altitude that helps. Or maybe trips to Europe bring out something creative in me. Only more experimentation will tell.

Ant, as we know it, had come into being. Everything that you see in the version of Ant that you use today (the good and the bad) is a result of the decisions made up to that point. To be sure, a lot has changed since then, but the basics were there. It was essentially this source code that was checked into Apache's CVS repository alongside Tomcat in late 2000. I moved on to other things, principally being Sun's representative to the Apache Software Foundation as well as working on XML specifications such as JAXP from Sun and DOM from the W3C.

Amazingly enough, people all over the world started talking about Ant. The first people to find it were those that worked on Tomcat at Apache. Then they told their friends about it. And those friends told their friends, and so on. At some point more people knew about and were using Ant than Tomcat. A strong developer and user community grew up around Ant at Apache, and many changes have been made to the tool along the way. People now use it to build all manner of projects, from very small ones to incredibly huge J2EE applications.

The moment I knew that Ant had gone into the history books was during JavaOne in 2001. I was at a keynote presentation in which a new development tool from a major database software company was being demoed. The presenter showed how easy it was to draw lines between boxes to design software, and then hit the build button. Flashing by in the console

[2] Also, the letters ANT could stand for "Another Neato Tool." Silly, I know. But true.

window were those familiar square brackets that every user of Ant sees on a regular basis. I was stunned. Floored.

The number of Ant users continues to increase. Evidently the little itch that I scratched is shared by Java developers world wide. And not just Java developers. I recently stumbled across NAnt, an implementation of Ant's ideas for .NET development.[3]

If I had known that Ant was going to be such a runaway success, I would have spent a bit more time on it in the first place polishing it up and making it something more than the simple hack it started out as. Yet that might have defeated exactly the characteristic that made it take off in the first place. Ant might have become over-engineered. If I had spent too much time trying to make it work for more than just my needs, it might have become too big a tool and too cumbersome to use. We see this all the time in software, especially in many of the Java APIs currently being proposed.

It might be that the secret to Ant's success is that it didn't try to be successful. It was a simple solution to an obvious problem that many people were having. I just feel honored to be the lucky guy who stumbled across it.

The book you now hold in your hands will guide you in using Ant as it exists today. Jesse and Eric will teach you how to use Ant effectively, extend it, and tell you how all the various tasks, both the built-in ones as well as widely used optional ones, can be used. In addition, they will give you tips to avoid the pitfalls created by some of Ant's design decisions.

Before placing you in their capable hands, I want to leave you with just one last thought: always scratch your own itch where possible. If a tool out there doesnt do what you need it to do, then look around for one that will. If it doesnt exist, then create it. And be sure to share it with the world. Thousands of other people might have just the same itch that you do.

—James Duncan Davidson

San Francisco, CA, April 2002

[3] You can find NAnt at http://nant.sourceforge.net/.