Editor: Diomidis Spinellis ■ Athens University of Economics and Business ■ dds@aueb.gr

Software Builders

Diomidis Spinellis

The tools and processes we use to transform our system's source code into an application that we can deploy or ship have always been important, but nowadays they can mean the difference between success and failure. The reasons are simple: larger code bodies; teams that are bigger, more fluid, and more widely distributed; richer interactions with other code; and sophisticated tool chains. All these mean that a slapdash software build process will be an endless drain on productivity and an embarrassing source of bugs, while a high-quality one will give us developers more time and traction to build better software.

Automate

Software building's golden rule is that you should automate all build tasks. This automation's scope includes setting up the build environment, compiling the software, performing unit and regression testing, typesetting the documentation, stamping a new release, and updating the project's Web page. You can never automate too much. In a project I manage, I've arranged for each release distribution to pick up from the issue-management database the bugs that the release fixes and then include them in the release notes. This has markedly improved the performance of external testers. Automation serves three purposes: it documents the processes, it speeds up the corresponding tasks, and it eliminates mistakes and forgotten steps. (Did we correctly update the documentation to indicate the software's current version?)

At the simplest level you can automate processes by writing small scripts or programs, using your operating system's shell language or a general-purpose scripting language. In some cases—for instance, flashing an embedded device's memory image—you might even need to develop a purpose-built program to avoid mouse-clicking on that pesky GUI application that your hardware vendor supplied. However, this approach is suitable for only the most specialized purposes. In most cases, a build tool will standardize your process and provide you with many useful facilities.

Choose

The most popular tool options for automating your build are the facilities that your IDE provides, the various implementations of Make, and Apache Ant and Maven. I don't recommend basing your build process on your IDE for anything but the most trivial projects. The build process gets tied down to the specific IDE and the platforms it runs on. Even if the IDE is popular, why needlessly restrict the developers' choice? Also, most IDEs provide limited build facilities, often restricting how you can abstract tasks and options.

Maven is an interesting choice, if your project is Java based. It's a tool with an attitude, sporting a range of built-in patterns for software builds. If you're willing to adopt its predefined patterns, you end up with a well-defined, complete, and standardized build process and with less verbiage than other alternatives.

The differences between Make and Ant are noteworthy, but the choice isn't difficult. Make has been used for everything—from typesetting books to setting up phone exchanges. Ant's domain is limited mostly to the Java world. However, nowadays some circles consider building a Java application with anything but Ant or Maven downright eccentric. So, if you're working with Java, you should have a very convincing story to explain a contrarian choice. However, keep in mind that Sun's Java Development Kit ships with 471 makefiles (Make's default input file) and just 36 Ant build files.

Both tools work on a dependency graph of

 0740-7459/08/$25.00 © 2008 IEEE

tasks. For this, you describe your build processes as a series of tasks that depend on each other. For instance, to link together your project's modules, you must first compile them. Make's graph nodes are typically files; its decisions on what to build are based on those files' time stamps. This lets it short-circuit large parts of the build process when the build is incremental, giving you a performance edge. Ant's tasks are abstract named blocks. Ant will always traverse the whole graph, but some tasks, such as that of the Java compiler, can internally determine that their work is already done.

A major advantage of Ant is its scripts' portability. In contrast to Make, which invokes external programs to accomplish its work, Ant's tasks are built into it (or loaded as extensions, written in Java). So, an Ant script should behave identically on any Java platform, whereas with Make you must spend effort to avoid or abstract-away system-specific commands. If you're using Make on a Windows platform, installing a Unix-compatibility suite such as Cygwin can help your makefiles run on both Windows and Unix. Alternatively, you can inject cross-platform compatibility into your build system through CMake (http://cmake.org).

Portability aside, because Make does its work with normal shell commands, you can easily dry-run any part of your build process on the command prompt. Debugging a build process (yes, unfortunately this is sometimes needed) is also easier with Make, because the output you see from it is the commands that the system runs. In contrast, Ant's behavior is opaque: to analyze what a task is doing, you must add print statements and (at a deeper level) look at the Java source code that implements it.

Some developers have reimplemented and extended the original 1970s Make program. Versions such as those connected with the Berkeley Software Distribution (BSD) and GNU offer file inclusion, conditionals, more readable ways to specify implicit rules and their variables, string processing, and loops. These features increase the expressiveness of your build scripts but can make them less portable, because they become tied to a specific implementation of Make.

Optimize

After automating your build process, the next step is to optimize it. As much as I nostalgically remember the days when I could cook and eat dinner while compiling an application, a quick build cycle can keep developers focused by robbing them of the excuse to browse Slashdot (and worse) while their code is compiling. The first optimization step involves the correct handling of dependencies, so that a part (for instance, an object file) is built if and only if one of its constituents (the corresponding source file) changes. This step poses two possible problems. Extraneous processing (for example, compiling many C source files together by invoking the compiler with a wild card) is a waste of time. On the other hand, missing a dependency, such as the fact that a C file must be recompiled when a header file changes, can introduce subtle bugs that are difficult to track down. For preprocessor-based languages such as C and C++, dependency tracking can become so complicated that tools (such as ccache) will cache each compile cycle's input and output to transparently skip compilations for which they have the correct cached result.

An additional neat optimization possibility is to exploit idle workstations in your organization or processor cores on your machine. Many parts of a build process are trivially parallelizable and contain a nice mix of I/O and CPU-intensive processing. So, you can shave significant fractions of the build time by running parts in parallel. Most modern Make programs can do that (with a -j option specifying the number of jobs to run simultaneously), while Ant has the equivalent "parallel" container task. In addition, tools such as distcc and Icecream can distribute a build across many machines.

Excel

Once you've got that build process in place, take the extra steps needed to make it shine. A makefile or an Ant build file is also source code, and you should treat it with the same respect. Put it under version control, document it with ample comments, use descriptive variable and target names, put often-used sequences into reusable blocks, and don't repeat yourself. Appropriate reuse can keep your build specifications short and sweet. For instance, almost half of the 2,400 makefiles that control the FreeBSD operating system's build process are shorter than a dozen lines, while the whole system (the kernel, 706 commands, and 725 libraries) can be built through them with just two commands.

Finally, invest some effort to squeeze the most out of your build process. Once you've automated the process, you can couple it with your version control system and arrange for nightly or continuous builds. A small script can retrieve the source code's latest version, build it (with full compiler warnings enabled and treated as errors), and test it. This "tinderbox script" can then immediately send any error messages to all the developers, thus exerting peer pressure that keeps your system always ready to ship.

Build automation is one of those remarkable places where product and process, programmers, and managers meet with common interests and goals. Invest in it and you won't regret it.

A quick build cycle can keep developers focused by robbing them of the excuse to browse Slashdot (and worse) while their code is compiling.

Post your comments online by visiting the column's blog: **www.spinellis.gr/tools**

Diomidis Spinellis is an associate professor in the Department of Management Science and Technology at the Athens University of Economics and Business and the author of *Code Quality: The Open Source Perspective* (Addison-Wesley, 2006). Contact him at dds@aueb.gr.