A Build System for Complex Projects: Part 1

A different approach to build systems

July 07, 2009
URL:http://www.drdobbs.com/tools/a-build-system-for-complex-projects-part/218400678

Gigi Sayfan specializes in cross-platform object-oriented programming in C/C++/ C#/Python/Java with emphasis on large-scale distributed systems. He is currently trying to build intelligent machines inspired by the brain at Numenta (www.numenta.com).

Build systems are often a messy set of scripts and configuration files that let you build, test, package, deliver, and install your code. As a developer, you either love or loathe build systems. In this article series, I present a different approach to build systems, with the ultimate goal of completely hiding the build system from developers. But first, let me start with some personal history.

Early in my programming career I was a pure Windows developer (with the exception of my very first job, where I wrote Cobol programs for publishing Australia's Yellow Pages). While there was no build system to speak of, there was Visual Studio and Visual SourceSafe. I built Windows GUI clients, messed around with COM components, and picked up some nice C++ template tricks from ATL. And because automated unit testing wasn't very common back then, we created various test programs before passing code on to QA. This wasn't too painful since I worked for a small startup company and the projects weren't too big.

But I then moved to a company that developed software for chip fabrication equipment in the semi-conductor industry and BOOM! Life-critical and mission-critical real-time software running on six computers that controlled custom-built hardware in clean-room conditions. The software ran on several operating systems with about 50 developers contributing code. The development environment consisted of two machines running Linux and Windows/Cygwin. The deployment environment was Solaris and LynxOS RTOS. No more Visual Studio. After reading about 1000 pages of documentation in the first week and getting my .profile and .bashrc in order, I was assigned my first task -- designing and implementing a build system to replace the existing one, which was a nasty combination of Makefiles and Perl scripts that actually worked but nobody was sure why (the original author had left the building). There were a few bugs (for example, the build system didn't always follow the proper dependency path) and a big requirements document. Clearly it would be impossible to evolve the current build system, so I had to create a new one from scratch. This was lucky because I had zero experience with Makefiles and Perl, coupled with the tolerance threshold of a Windows developer to gnarly stuff. I still have the same tolerance, but I now know something about Makefiles.

Some of the requirements were pretty unusual, like running a commercial code generator that produces code from UML diagrams on a Windows machine, then uses the artifacts to compile code on Linux, Solaris, and LynxOS. The bottom line is that I decided to take an unusual approach and wrote the entire system in Python. It was my first big Python project and I was really surprised at how well it went. I managed everything in Python. I directly invoked the compiler and linker on each platform, then the test programs, and finally a few other steps. For instance, I implemented friendly error messages that provided helpful suggestions for common errors (e.g., "FrobNex file not found. Did you remember to configure the FrobNex factory to save the file?").

While I was generally pleased with the system, it wasn't completely satisfactory. In lieu of Makefiles, I created build.xml files, a la Ant. That was a mistake. The XML files were verbose compared to Makefiles, big chunks were identical for many subprojects, and people had to learn the format (which was simple, but something new). I wrote a script that migrated Makefiles to build.xml files, but it just increased code bloat. I created a custom build system without regard for the specific environment and its needs. I created a very generic system, with polymorphic tools that can do anything as long as you write the code for the tool and configure it properly. This was bad. Whenever someone says, "You just have to ..." I know I'm in trouble. What I took away from this experience is that Python is a terrific language. It's really fun when you can actually debug the build system itself. Having full control over the build system is great, too.

Background: What Does a Build System Do?

The build system is the software development engine. Software development is a complex activity that involves tasks such as: source-code control, code generation, automated source code checks, documentation generation, compilation, linking, unit testing, integration testing, packaging, creating binary releases, source-code releases, deployment, and reports. That said, software development usually boils down to four main phases:

1. Developers write source code and content (graphics, templates, text, etc.)
2. The source artifacts are transformed to end products (binary executables, web sites, installers, generated documents)
3. The end products are tested
4. The end products are deployed or distributed

A good automated build system can take care of steps 2-4. The distribution/deployment phase is usually to a local repository or a staging area. You will probably need some amount of human testing before actually releasing the code to production. The build system can also help with that by notifying users about interesting events, such as successful and/or failed builds and providing debugging support.

But really, who cares about all this stuff? Actually everybody -- developers, administrators, QA, managers, and even users. The developers interact most closely with the build system because every change a developer makes must trigger at least a partial build. When I say "developer" I don't necessarily mean a software engineer. I could be referring to a graphic artist, technical writer, or any other person that creates source content. When a build fails, it's most often because a developer changed something that broke the build. On rare occasions, it would be an administrator action (e.g., changing the URL of a staging server or shutting down some test server) or a hardware problem (e.g., source control server is down). A good build system saves time by automating tedious and error-prone activities.

Think about a developer manually building and unit testing a program. Without a build system, he has to very carefully build it properly, test it, and hand it over to QA. The QA person needs to run his own tests, then hand it to the administrator for deployment to a staging site, where more tests are run against the deployed system. If anything goes wrong in this process, someone must determine what happened. Automated build systems eliminate a whole class of errors. They never forget a step and they can pinpoint and resolve other errors by verifying that the source artifacts and intermediate artifacts are available and by scanning through log files and detecting failures.

Managers can also benefit from build systems. A passing build is the pulse of a project. If you have an automated build system with good test coverage (at the system level), managers can monitor project progress and be ready to release at each point. This in turn enables more agile development practices (if you are so inclined).

A build system can even help users in some cases. Think about systems that incorporate user-generated content and/or plug-ins. In most cases, you need to go over the content and ensure it doesn't break your system. A build system that automates some/all of these checks allows for shorter publish/release cycles for user-generated content.

Build System Problems

Okay, build systems are the greatest thing since Microsoft Bob. However, they still don't always live up to their potential:

- **They Don't Do Enough (Not Fully Automated)**. This is one of the most common problems. A build system that is not fully automated can compile the software, create documentation, and package the final binary, but it requires a lot of user intervention to run various scripts, wait for previous stages to finish, check error reports, and so on.
- **Requires a Lot of Discipline to Use Properly**. Some build systems fail inexplicably if you don't follow a slew of obscure steps, like logging into the test server with a specific user, removing directory A, renaming directory B, making sure you perform step X only if the report generated by step Y says okay.
- **Requires Too Much Configuration**. Some build systems are very powerful and flexible, but are almost unusable due to excessive configuration. You have to define six different environment variables, modify three local config files, and pass eight different command-line options to the main build script. The end result is that 99% of the users use a single default configuration that probably doesn't fit their needs.
- **Caters Mainly To a Sole Stakeholder**. Another common problem is that a build system is often suitable for just one kind of stakeholder. For example, if the build system was developed mainly by the programmers who compile, link, and unit test all day, then the build system will have good support for these activities, but running integration tests or generating documentation may be poorly supported, if at all. On the other hand, if the build system was developed mainly by a release engineering team, then it will have good support for packaging final executables and will generate good reports about the percentage of passing test, but it may not be possible for developers to run just a single unit test and its dependencies, and they will either have to run the full-fledged build every time or hack the build system in a quick and dirty way (which might lead to errors).
- **Intractable Error Messages When Something Is Wrong**. Build systems perform many activities that involve external tools. The errors generated by these tools are often swallowed by the build system that much later generates its own error message, which doesn't point to the root cause. This is a serious problem that hurts productivity and causes people to revert to manual but understandable build practices.
- **Inextensible and Undebuggable Franken-Code**Build systems are often one of the earliest tools created at project initiation. The requirements of this early build system are usually minimal. As time goes by and the project grows, the demands from the build system grow too. Since the build system is an internal tool, less effort is dedicated to making it high quality code. More often than not, it is just a bunch of scripts slapped together and extended to support additional requirements by the tried and true practice of copy and paste. Such build systems quickly become a maintenance nightmare and can't be extended easily to accommodate new requirements.
- **Not Integrated With Developer's IDE**Most build systems that don't come with an IDE built-in don't support IDEs. They are command-line based only and if a developer wants to work in an IDE, the IDE project files must be maintained and synchronized with the build system build files. For example, the build system may be Makefile-based, and a developer that uses Visual Studio has to maintain a .vcproj file for each project, and any additional files must be added to the Makefile as well.

The Perfect Build System

The build system I present in this series is open ended and can be used to automate any software process that is mainly file-based. However, the focus is on a cross-platform build system for large-scale C++ projects because these are often the most complicated to build. The perfect build system solves or minimizes the problems associated with existing build systems.

"Convention over configuration" is a principle that has successfully governed in domains like web frameworks, reducing the learning curve and increasing developer productivity. It demands that you organize your project in a consistent way (which is always good practice in any event):

- **Regular directory structure.** This is the key principle on which the entire build system rests. Even in the most complicated systems, there is usually a relatively small high-level directory structure that contains a potentially huge number of similar directories. For example, a project may have a **libs** directory that contains all the C++ static libraries. The contents of the **libs** directory may grow and change, but it always contains a single type of entities.
- **Well-known locations.** The build system should be aware of the location and names of the top-level directories and "understand" what they mean. For example, it should know that the directories under **libs** generate static libraries that should later be linked into executables and dynamic libraries

that depend on them.
- **Automatic discovery of files based on extension**. Each directory usually contains a small number of file types. Again, in the **libs** example, it should contain .h and .c/.cpp files and potentially a couple of other metadata files. The build system should know what files to expect and how to handle each file type. Once you have the regular directory structure in place, the build system "knows" a lot about your system and can do many tasks on your behalf automatically. In particular, it doesn't need in a build file in each directory that tells it what files are in it, how to build them, etc.
- **Capitalize on the small variety of sub-project types**. In the C/C++ world, there are really only three types of subprojects: a static library, a dynamic library, and an executable. Static libraries (a compiled set of files bundled together) are the simplest. They are later linked into dynamic libraries and executables. Dynamic libraries and executables are similar from a build point of view. They both have source files and depend on precompiled static libraries to link against. It is important to build the dependent dynamic libraries and executables after building all the required static libraries. Many libraries (both static and dynamic) and executables use the same set of compiler and linker flags. Placing these groups under a parent directory informs the build system of these common flags and automatically builds all the subprojects.
- **Generate build files from templates for any IDE**. Different IDEs, as well as command-line based tools like Make, use different build files to represent the meta information needed to build the software. The build system I present here maintains the same information via its inherent knowledge combined with the regular directory structure and can generate build files for any other build system by populating the appropriate templates. This approach lets developers build the software via their favorite IDE (like Visual Studio) without the hassle involved in adding files, setting dependencies, and specifying compiler and linker flags.
- **Automatic dependency management based on #include analysis**. Managing dependencies can be simple or complicated depending on the project. In any case, missing a dependency leads to linking errors that are often hard to resolve. This build system analyzes the **#include** statements in the source files and recursively creates a complete dependencies tree. The dependencies tree determines what static libraries a dynamic library or executable needs to link against.
- **Automatic discovery of added/removed/renamed files and directories**. The regular directory structure, combined with knowledge of files types (e.g., .cpp or .h files), allows the build system to figure out what files it needs to take into account, so developers just need to make sure the right files are in the right directory.
- **Flexibility**
  - Support static libraries, dynamic libraries, executables, and custom artifacts. All possible build artifacts are supported including custom ones like code generators, preprocessors, and documentation generators. The ability to put similar files and subprojects under top-level directories in the regular directory structure is open to any subproject type.
  - Control the level of error messages. The build system is designed to support different users, such as QA, developers, and managers. Each type of user may be interested in different error messages.
  - Generate custom artifacts like language bindings. The build system is focused on building C/C++ code, but using the same practices and mechanisms it is possible to extend it to support additional artifacts, while maintaining all the existing benefits.
  - Allow overriding defaults. While the build system is intended to provide a hands-free experience, where all the necessary build information is derived automatically from the directory structure, it is possible to override it for special purposes, such as a single library that needs different flags.

- **Integrated Build System**
  - Build phases are executed from the same program. The build system is a cohesive program that operates on a set of templates and source files. This one-stop shop approach is very powerful for keeping the build process manageable.
  - Invoke external programs as a last resort. Ideally, the build system contains the entire logic of each build step. External programs are invoked only when the effort to implement the logic in the build system itself is deemed too costly. For example, the compiler and linker are invoked as external programs.
  - Full debugging of the build system. The fact that the build system is a single program allows users to debug the build process in real-time including setting breakpoints, viewing the current state, and finding live build system bugs. This is very different from standard declarative build files that usually only provide obscure error messages at a much later stage.

Hello, World (Platinum Enterprise Edition)

I hope you agree that this build system sounds awesome. But is it for real? To demonstrate and explore its capabilities, I will follow an imaginary software team that just started working on a new project.

The project is called "Hello, World!". The goal is to print it to the screen. To do this, over the course of this series the team will create a complex project with multiple executables, static and dynamic libraries, and even Ruby bindings. The project will run on Windows, Linux, and Mac OS X. It will be built using a custom build system. To whet your appetite, here is a prototype in Python of the finished project:

```
print 'Hello, World!'
```

Project Kick-Off

Isaac, the sage development manager, assembled a team of brilliant software developers with umpteenth-years of experience in delivering high-performance enterprise applications. The kick-off meeting went well and the developers quickly reached a few decisions:

- The project will be developed mostly in C++,
- The system must be cross-platform and support Windows, Linux, and Mac OS X,
- The developers will be divided into four teams.

  - Team H will develop a static library called libHello that returns "Hello".
  - Team P will develop a dynamic library called libPunctuator that produces commas and exclamation points (and can be reused in future projects requiring punctuation).
  - Team W will develop the complicated libWorld static library that must return the long and difficult word "World".
  - Team U will develop an infrastructure project called libUtils that provides utility services to the other teams.
- The project will also deliver a Ruby language binding to make it more buzzword-compliant.

- The test strategy is to develop multiple test programs to test every library. Each team will be responsible for developing the test program for its library.
- The build system will be developed in Python by the renowned build expert Bob (aka "The Builder"). (No connection to Microsoft Bob, thank you.)

Bob carefully observed the source and required artifacts of the system and came up with the following directory structure. Each kind subproject is contained in a top-level directory under the source tree:

```
root
 |___ ibs
 |___ src
       |___apps
       |___bindings
       |___dlls
       |___hw (static libraries)
       |___test
```

- The **ibs** directory contains the files and templates of the build system. Note that it is completely separate from the source tree under **src**.
- The **src** directory contains all the source files of the system. Let's take a quick look at the top-level directories under **src**.
    - **apps**. This directory contains all the (executable) applications generated by the system. Each application will have its own directory under apps.
    - **bindings**. This directory will contain Ruby bindings at some point. At the moment it is empty.
    - **dlls**. This directory contains the project's dynamic libraries.
    - **hw**. This directory contains the project's static libraries. The reason it is called **hw** (as in "hello world") and not **libs** or a similar name is that it is very important to prevent name clashes with system or third-party static libraries. The automatic dependency discovery of the build system relies on analysis of **#include** statement. The unique **hw** part of the path of each static library allows unambiguous resolution of **#include** statements.
    - **test**. This directory contains a subdirectory for each test program. Each test program is a standalone executable linked against the static libraries it is designed to test.

Next Time

In the next installment of this series, Bob and I delve into the innards of the build system and explain exactly how it works. Stay tuned.

- A Build System for Complex Projects: Part 2