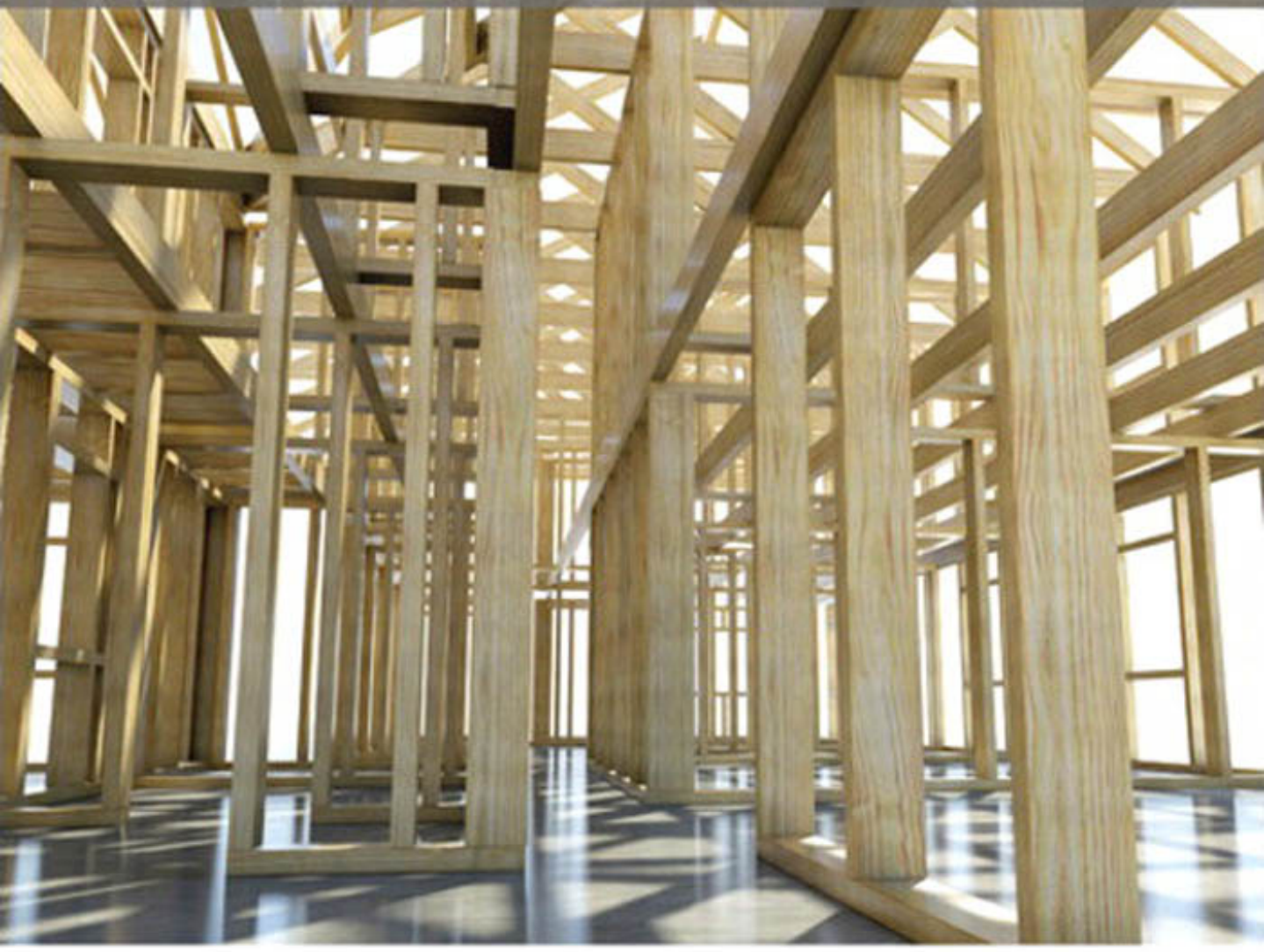
Software Build Systems

Principles and Experience



Peter Smith

Praise for *Software Build Systems*

"This book represents a thorough and extensive treatment of the software build process including the choices, benefits, and challenges of a well designed build process. I recommend it not only to all software build engineers but to all software developers since a well designed build process is key to an effective software development process."

—**Kevin Bodie**, Director Software Development, Pitney Bowes Inc.

"An excellent and detailed explanation of build systems, an important but often overlooked part of software development projects. The discussion of productivity as related to build systems is, alone, well worth the time spent reading this book."

—**John M. Pantone**, Objectech Corporation, VP,
IT Educator and Course Developer

"Peter Smith provides an interesting and accessible look into the world of software build systems, distilling years of experience and covering virtually every type of tool in the build engineer's toolbox. Well organized, well written, and very thorough; I would recommend this book to anyone with a build system under their responsibility."

—**Jeff Overbey**, Project Co-Lead, Photran

"*Software Build Systems* teaches how to think about building software. It surveys the tools and techniques for building software products and the ways things go wrong. This book will appeal to those new to build systems as well as experienced build system engineers."

—**Monte Davidoff**, Software Development Consultant,
Alluvial Software, Inc.

This page intentionally left blank

Software Build Systems

This page intentionally left blank

Software Build Systems

Principles and Experience

Peter Smith

Addison-Wesley
Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

Electric Cloud, ElectricAccelerator, and SparkBuild are registered trademarks of Electric Cloud, Inc.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data:

Smith, Peter, 1970-

Software build systems : principles and experience / Peter Smith.
p. cm.
Includes bibliographical references and index.
ISBN-13: 978-0-321-71728-3 (hardback : alk. paper)
ISBN-10: 0-321-71728-7 (hardback : alk. paper) 1. Compilers (Computer programs)
2. Programming software. 3. Self-adaptive software. 4. Application software--Development--
Computer programs. I. Title.
QA76.76.C65S65 2011
005.4'53--dc22

2010051013

Copyright © 2011 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671-3447

ISBN-13: 978-0-321-71728-3
ISBN-10: 0-321-71728-7

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.

First printing March 2011

Editor-in-Chief
Mark Taub

Executive Editor
Chris Guzikowski

Senior Development Editor
Chris Zahn

Managing Editor
Kristy Hart

Project Editor
Anne Goebel

Copy Editor
Krista Hansing
Editorial Services

Indexer
WordWise Publishing
Services, LLC

Proofreader
Apostrophe Editing
Services

Editorial Assistant
Raina Chrobak

Interior Designer
Gary Adair

Cover Designer
Anne Jones

Compositor
Gloria Schurick

To Grace and Stan

This page intentionally left blank

Contents

Preface

Are you a software developer? Are you interested in how build systems work? You're reading this book; so there's a good chance you answered "Yes" to both questions. On the other hand, many software developers aren't interested in how their program is compiled. Most people just want to press a button and have their source code turned into an executable program. If they need to fix a bug, they change the source code and press the same button again. Their joy is in seeing their program do all the exciting things it's supposed to do. The build system is just something that needs to be there in the background.

Anything more than a small collection of source files requires some type of automated build system. This may be a shell script that you run after each source code change, a makefile that knows the relationship between the source and object files, or a more complex build framework that scales to thousands of source files.

If you've developed code in a UNIX or Windows command-line environment, the following command should look familiar:

```
cc -o sorter main.c sort.c files.c tree.c merge.c
```

In this example, five C-language files are being compiled and linked to create a single executable program, named sorter. This may be unfamiliar to those who use an integrated development environment (IDE), but it's essentially the same as creating an IDE project with five source files and then pressing the `build` button on the toolbar.

After you've compiled your program a few times, you'll probably decide to store this command in a shell script and rerun it any time you make a code change. Alternatively, you can retrieve the command from your command-line history and replay the sequence each time you modify the code.

If you have some basic knowledge of the Make tool, you can create yourself a makefile and type make each time you need to rebuild. The advantage of Make is that it rebuilds the program only if any of the source files changed since the last compilation. Here's a simple makefile for compiling the `sorter` example:

```
sorter: main.c sort.c files.c tree.c merge.c
        cc -o sorter main.c sort.c files.c tree.c merge.c
```

If you're familiar with Make, you'll immediately realize that this isn't a good way to write a makefile. The first mistake is that the source files are listed twice, once for the dependency relationship and a second time in the compilation command. Next, all source files are compiled each time you rebuild the program, even if they haven't all been modified. Finally, there's no mention of dependencies that a C file may have on header files.

A better solution is to break up the compilation steps so that each source file is compiled, and recompiled, independently of the others. Additionally, there should be dependency files (with a suffix of `.d`) to track header file usage. The list goes on, so rather than go into all the technical details, take a look at the final makefile that does everything you need:

```
SOURCES = main.c sort.c files.c tree.c merge.c

OBJECTS = $(SOURCES:.c=.o)

sorter: $(OBJECTS)
        $(CC) -o $@ $^
-include $(SOURCES:.c=.d)

%.d: %.c
        @$(CC) -MM $(CPPFLAGS) $< | sed 's#\(.*\)\.o: #\1.o
➥\1.d: #g' > $@
```

That's all there is to it—a simple makefile that does the bare-minimum amount of work, with the least amount of repetition. Easy, right?

If you're a developer and not a build expert, though, do you really understand what's going on in the previous example? A seasoned Make expert certainly understands the syntax and would probably suggest a more efficient way of achieving the same result. However, most of us who just want a push-button build are destined to waste a lot of time getting the makefile correct in the first place.

Build systems tend to be complex to implement and maintain. A badly designed build system can waste many hours if a file isn't recompiled when it should have been. When scaled to thousands of source files, a developer can literally waste half a day tracking down a problem, only to find that starting the build from scratch (removing all the object files) is the only way to make things work. So much for a push-button build!

Why Do Build Systems Become Complex?

You might be surprised to read that build systems can be complex and hard to maintain. With graphical user interfaces so common these days, you'd expect build tools to be equally simple to use. Unfortunately, many see creating a build system as a black art. Only a few knowledgeable gurus understand the full syntax of the build tool or the subtleties in the dependency system. Although IDE-based build tools go part of the way toward solving this problem, they can't support the complexities of a large-scale build system.

In most cases, a software product starts with a small number of source files that are compiled and linked into a program. A simple makefile is sufficient in this case, and these can be thrown together in a couple hours by copying the makefile template from a user manual. For several months, nobody needs to change this build system, aside from adding new source files or libraries.

After a while, people start to see problems in the build process. They notice that files aren't recompiled when they should be, or perhaps that files are incorrectly being recompiled when none of the data they depend on has changed. In other cases, files may be compiled multiple times in the same build, leading to slower build times. It quickly becomes part of the engineering culture to always do a "clean build" (removing all object files first) or to modify files for the sole purpose of making them recompile.

When this simple build system becomes painful to use, a makefile expert needs to rethink the design. They might create a framework that solves all the build problems, while keeping the implementation detail away from the end users. For example, software developers want to have visibility into the list of source files, libraries, and compilation flags being used, but they aren't interested in how the dependencies are managed. For example:

```
SOURCES := main.c sort.c files.c tree.c merge.c

PROGRAM := sorter
LIBRARIES := libc libz

include framework.mk
```

The end goal is to have a correct and easy-to-use build system, while hiding all the complexity inside the `framework.mk` file. This is an ideal solution for the software developer who just wants a push-button build.

This framework approach works efficiently for a while, although growing pains start some time in the future. This is particularly true for a successful product whose software grows over a number of years. The build system that worked for a small-to-medium product no longer works when the product scales.

Consider how you'd integrate a new code module purchased from a third-party vendor. The new code already has its own build system and uses a different build framework than your original product. When developers modify the code, they create interdependencies between this newly acquired code and your existing code base, requiring the build system to understand the more complex file relationships. The end result is that one or both of the build frameworks requires significant rework—and possibly a complete rewrite.

As frameworks grow over time, maintaining them properly becomes challenging. In some cases, the original author of the framework is no longer available to make changes, so a nonguru steps in to perform the work. Developers who lack sufficient build experience often use quick-and-dirty techniques to get the software to build. As discussed later, these techniques include badly written shell scripts, copious use of symbolic links, and, worst of all, duplicate copies of source files. The build process becomes a rat's nest of complexity that nobody is comfortable maintaining.

It's sad that many organizations don't feel compelled to fix their build system. If they're experts in some other field (such as computer gaming, telecommunications, or business applications), their enthusiasm is directed toward creating their product and adding new features to entice and excite their end customer. The build system is viewed as a necessary part of the product life cycle, but people don't see it as their job to fix. The task certainly never appears in a company's corporate objectives or quarterly feature plan.

As you'll see throughout this book, plenty of issues must be considered when designing a build system. It's not just a matter of having a makefile guru on call to help with problems. You should also keep the development environment in a maintainable state. The time and money spent cleaning up a build system can pay off many times when you consider a software team's overall productivity.

The True Cost of a Build System

If you don't already believe that a reliable build system is important, think about the true cost. That is, what costs will you incur if you don't have a good build system? These aren't numbers that appear on any accountant's balance sheet; they're hidden inside the day-to-day productivity of software developers.

One industry survey [1] found that developers perceived an average productivity loss of 12% due to build problems, although some of the respondents felt that 20%–30% was not uncommon. It's worth noting that this survey focused on smaller development groups (with less than 20 people), who likely didn't suffer from the scalability problems encountered with much larger software.

Let's start by assuming that all software developers in your team lose 10% of their time to problems with the build system. Your reaction to this figure will vary, based on your previous experience with software projects. For some people, 10% may seem like an exaggerated figure, but for many groups, this is on the low side.

What are the reasons for this 10% loss of productivity? Consider some typical problems your team has almost certainly experienced in the past:

- **Bad dependencies causing false compile errors:** The build system has somehow acquired incorrect dependency information and is failing to recompile parts of the source code correctly. When this happens, the developers focus all their time on trying to complete a successful build. They're faced with cryptic error messages completely unrelated to the area of code they've been changing. Until these are fixed, they're unable to proceed with productive work.

- **Bad dependencies that create failed software images:** As in the previous case, bad dependencies cause parts of the build to compile incorrectly. However, instead of giving a compilation error, the program no longer generates the correct output. This simply gives the developer and software testers the impression that the code is buggy, and they often blame themselves instead of the build system. Developers waste a day or two trying to debug a test failure, only to discover that their private code changes aren't causing the problem. Starting with a fresh build tree makes the problem go away.

- **Slow compilation:** This is more of a problem for larger software systems, because smaller software can be built in a matter of minutes. If your software code base requires many hours to compile, developers waste time while they wait for the compilation to complete. This is particularly troublesome for incremental builds in which changing a single source file can result in a delay of 5–10 minutes before the program is ready to execute again.

  You may feel that people can productively do other work while they wait for their compilation, but this isn't always the case. Developers have many

types of "waiting" activities, such as reading the latest news headlines, updating social networking sites, getting more coffee, or going off to chat with a friend. Even if a developer can multitask while the build completes, the cost of context switching between the different tasks is a productivity loss. Developers can get distracted and completely forget about one of the tasks they were working on.

- **Time spent updating build description files:** If the software build framework isn't trivial to understand, developers may need to ask an expert to make modifications. For example, if they need to add a new type of source file or a new compilation tool, they must first engage in a discussion with a build guru. This can take days of waiting while the build guru finds time to help. After that, the build guru might need a few weeks to complete the job.

If you now believe that a 10% productivity loss is a realistic number, what's the financial cost of this loss? The best way to evaluate this is to determine 10% of your organization's salary payment. This clearly doesn't apply if you're volunteering to write the software (as is commonly the case in the open-source world), but the numbers are interesting all the same.

Assume that you have ten software engineers, each of which is paid $75,000 per year. This is high for some cities and low for others, so it's worth evaluating the numbers from your own perspective. An accountant would likely double this estimate when considering the additional costs of employee medical benefits, electricity, rent, parking, and other perks a developer enjoys. Assume, therefore, that each developer costs $150,000 per year.

> Thus, the total cost of paying your developers to deal with build problems is
> *10% x US$150,000 per year x 10 developers = $150,000 per year*

That's equivalent to having a full-time developer sitting around for a whole year without doing any productive work! If you assume 250 working days per year, your company is paying $600 every day simply because of build problems!

If you were a software manager, what would you consider to be more profitable? Continuing to pay $600 per day for your team to waste time, or paying $600 per day for a few months to hire a new build guru to fix your problems? It's definitely worth considering what your own organization is doing. Remember, a company can make a profit in two ways: either by increasing revenue by

selling more of the product, or by reducing the cost associated with creating the product in the first place.

The Focus of This Book

You should spend time reading this book for two reasons:

- **To understand the basic principles underlying a build system:** This book provides an end-to-end survey of build system features and usage scenarios, giving you an understanding of how a build tool performs its work.
- **To gather more experience about build systems:** This book encapsulates years of experience in creating and maintaining build systems, using many different build tools. After reading this book, you can avoid making the same trial-and-error mistakes that previous build system developers have made.

Armed with such knowledge, you can make well-informed choices on which build tool to use, how to construct a reliable build system, and how to foresee traps and pitfalls before they impact your productivity. The outcome is that building software should get faster, easier, and more reliable.

It's also important to note what this book does not attempt to address:

- **Not a hands-on tutorial:** Except for a few small examples (such as those in Chapter 2, "A Make-Based Build System"), this book doesn't provide a hands-on tutorial on any particular build tool or technology. Popular build tools already have web sites and books devoted to teaching you every syntactic and semantic detail you'll ever need. Refer to those books for the finer details of each tool.
- **Doesn't show a fully functional build system:** Although this book contains a number of examples on how to use each build tool, and many supporting tools, it doesn't demonstrate the end-to-end creation of a full build system. Again, you should refer to each build tool's documentation to see fully worked-out examples.

Of course, read this book first so that you understand the pros and cons of each build tool and can judge for yourself which features your build system should use.

Instead of staying specific to a single development environment or programming language, this book offers examples and concepts from a variety of different angles:

- **C/C++ builds:** This is perhaps the most traditional type of build process. This style of building originated in the 1970s and hasn't changed much since then. The only recent challenge is the growth in the number of files and third-party libraries that are now used in a typical software product.
- **Java builds:** The Java language became popular in the late 1990s and has had a considerable impact on the design of build systems. As one example, Java source files must be stored in a directory hierarchy that matches the software package structure.
- **C# builds:** Whereas C, C++, and Java are platform-neutral programming languages and can thus be used on any operating system (such as Linux, Solaris, Mac OS X, and Windows), the C# build environment is more tailored toward the Microsoft way of doing things.

In addition to covering multiple programming languages, this book discusses two different approaches to constructing large software products:

- **Monolithic builds:** In this approach, the entire code base is compiled from source code into an executable program in a single build process. This is a common approach for small programs, but it doesn't scale well because it leads to large source trees and long compilation times.
- **Component builds:** In contrast to monolithic builds, this approach breaks the source code into multiple stages, each compiled separately. The final step is to integrate the various prebuilt components, to produce the final executable program.

Finally, this book goes beyond the common assumptions that Make is the primary tool of choice for C/C++ development and that all Java and C# software should be built inside an IDE.

Who Should Read This Book?

This book was written with several audiences in mind, although the primary focus is software developers:

- **Developers:** If you're a software developer with years of experience writing source code but only minimal experience with build systems, you can learn about the issues involved in constructing and maintaining a build system. You can also study the different tools that describe the build process.
- **Managers:** From this book, you can learn the concepts and tricks-of-the-trade at a fairly high level instead of seeing too much of the complex detail. This enables you to evaluate the work your team is doing, and ask the appropriate "direction-setting" questions.
- **Build gurus:** Even with years of experience in constructing build systems, you can expect to learn new things. Not only will you be exposed to modern build tools that you may never have used, but the discussions on scalability and performance of large build systems will make you think twice when you start to write your next build framework.

How This Book Is Organized

This book is divided into four main parts, each looking at build systems from a slightly different angle. Depending on your experience and level of interest, you might choose to focus on different parts of the book. Novice developers should focus on Parts I and II, whereas more experienced users should skim through Part I but focus their attention on Parts II, III, and IV.

Part I: The Basics

This first part provides a gentle introduction to build systems, for software developers who haven't had much exposure to the topic. Even advanced users should skim these chapters to ensure that they have a complete picture of the basic concepts. For example, C/C++ developers can learn new things about the C# language.

Chapter 1, "Build System Overview," provides an introduction to high-level build system concepts such as source and object trees, build tools, and compilation tools. Chapter 2, "A Make-Based Build System," provides a quick tutorial on writing a makefile, for those who have never done so. Chapter 3, "The Runtime View of a Program," describes the structure of a program as it executes on a computer, with the goal of describing what a build system needs to construct. Chapter 4, "File Types and Compilation Tools," goes into detail on the different types of input and output file used in the build process and uses examples

in the C/C++, Java, and C# languages. Chapter 5, "Subtargets and Build Variants," describes the basic idea behind build variants, which later chapters cover in more detail.

After reading Part I, you'll have a good understanding of the basic concepts surrounding the design of build systems.

Part II: The Build Tools

The second part of this book compares five build tools. Each tool was selected both because of its popularity and because it demonstrates a particular way of building software. Each chapter starts with an introduction to the syntax of the build tool and then describes the tool's main usage scenarios. To provide a meaningful comparison, a standard set of examples is used across all chapters.

Chapter 6, "Make," discusses the GNU Make tool, which is the most common tool for C/C++ development. Chapter 7, "Ant," examines the Ant build tool, which is the de facto standard for compiling Java. Chapter 8, "SCons," investigates the more recent SCons build tool, which uses the Python language to describe the build process. Chapter 9, "CMake," shows the CMake tool, which generates a native build system (such as a Make-based system) from a high-level description of the build process. Finally, Chapter 10, "Eclipse," describes the build-related features of the Eclipse IDE.

After reading Part II, you'll have an appreciation for the state of the art in build tools and will understand the pros and cons of using each.

Part III: Advanced Concepts

The third part discusses more advanced build system concepts, such as dependency analysis, software packaging and installation, version management, and the management of build machines and compilation tools. These chapters assume that you've had experience working on nontrivial software projects and can therefore relate to the issues discussed.

Chapter 11, "Dependencies," goes into detail on various dependency-checking techniques that discover whether a file must be recompiled. Chapter 12, "Building with Metadata," shows how a build system can generate metadata to aid with debugging, profiling, and source code documentation. Chapter 13, "Software Packaging and Installation," provides simple examples of packaging the software and getting ready to install it on the target machine. Chapter 14, "Version Management," surveys version-control issues as they relate to build systems. Chapter 15, "Build Machines," provides best practices for managing

the build machine on which the software is compiled. Chapter 16, "Tool Management," provides a similar discussion for compilation tools.

After reading Part III, you'll understand many of the advanced topics involved in constructing a build system and a number of best practices.

Part IV: Scaling Up

The final part of this book discusses the design of build systems for large software products. As a software product grows in size, it faces scalability problems, such as an increase in complexity, a dramatic increase in disk usage, and an increase in build times. All these problems tend to make software development less productive.

Chapter 17, "Reducing Complexity for End Users," provides approaches for reducing the complexity of a build system, as perceived by the end user. Chapter 18, "Managing Build Size," describes how a large software product can be divided into multiple components to make development more efficient. Finally, Chapter 19, "Faster Builds," discusses techniques for measuring and improving the time taken to perform a software build.

After reading Part IV, you'll have a better appreciation of how you should design your small-scale build system, in case it ends up becoming much larger.

Summary

A good-quality build system isn't easy to construct, and failure to do so causes significant problems for your software team. If source code isn't recompiled when it should be, your team members will face longer build times or random build failures. They may also waste days debugging an invalid software image. It's worth putting in the time to make sure your build system is doing the correct thing.

The true cost of using a poor quality build system can be measured in monetary terms. A typical software organization might find that developers waste 10% of their time with build problems, which translates into large sums of money wasted each year.

This book explains a number of build system concepts, introduces you to a range of commonly available build tools, provides a number of best practices, and discusses the issues surrounding the construction and maintenance of large build systems.

This page intentionally left blank

Acknowledgments

This book wouldn't be complete without a big thanks to my wife, Grace. I spent many evenings and weekends in my "man cave," tapping away at the keyboard. Grace understood the value I placed on writing this book (it was on my bucket list), and her patience and support made it all possible. Thanks also to Stan (our Bichon Maltese), who learned that sitting on the floor is often better than on my lap or keyboard.

Thanks go to my parents, Sally and Smithy, for allowing me to author several chapters from their dining room table. I also thank them for years of correcting my spelling and grammar, making it easier to write something the size of this book.

I appreciate the support of the team at Pearson Education who accepted this book for publication. Thanks to Raina Chrobak, Chris Zahn, and Chris Guzikowski for their guidance through the writing and editing process. Thanks also to the manuscript reviewers who provided feedback either from a practitioner's perspective or from the eyes of a build system expert. The reviewers include Monte Davidoff, Jeffrey Overbey, J. T. Conklin, Kevin Bodie, Brad Appleton, John Pantone, and Usman Muzaffar.

Next, I appreciate the support of Kevin Cheek and Bob McLaren, along with others on the team at Ericsson who allowed me to renegotiate my ongoing contract. This provided me with enough time to write a book. Thanks also to the many friends and colleagues who relayed experiences of their past and present build systems. I hope that I've given each of their experiences a suitable place in this book.

Finally, acknowledgment must be given to everybody who has contributed to the design or construction of a build tool. Most software projects use some type of build tool, making the build system a critical piece of technology. The people who create these tools don't always get the credit they deserve.

This page intentionally left blank

About the Author

Peter Smith is a freelance consultant for Arapiki Solutions, Inc. (www.arapiki.com), based in Vancouver, Canada. He obtained a Ph.D. in computer science from the University of British Columbia in 1998, focusing on compilers and language design. He spent several years teaching undergraduate courses in compiler design, programming language design, software engineering, and computer networks. He also served on the Object-Oriented Programming, Systems, Languages & Applications (OOPSLA) conference committee for three years. Peter has worked primarily in the telecommunications industry, as a software engineer, a project manager, and the manager of a tools support team. Recent consulting jobs included the adoption and development of new software tools to improve end-user productivity.

This page intentionally left blank

PART I

The Basics

Part I provides a gentle introduction to the concepts used in software build systems. This part starts with a high-level view of the various stages of the build process, describes what a build system aims to create, shows the various input and output files used during compilation, and introduces the concepts of build targets and variants. You'll explore these topics:

- **Chapter 1, "Build System Overview":** A brief tour of the major components of a build system, including a number of important definitions that you need for later chapters.
- **Chapter 2, "A Make-Based Build System":** A short tutorial on using the GNU Make build tool, for those who've never been exposed to a text-based build system.
- **Chapter 3, "The Runtime View of a Program":** The many ways in which a program can be loaded into a computer and executed. A software build system must create the executable programs, libraries, and data files that are loaded into memory.
- **Chapter 4, "File Types and Compilation Tools":** The tools used to compile C/C++, Java, and C# source code. These compilation tools are the building blocks of a complete build system.
- **Chapter 5, "Subtargets and Build Variants":** The approach taken when building software for multiple target CPUs or creating multiple editions of the product.

Although Part I provides an introduction to build systems and their purpose, this book doesn't discuss build tools until Part II; there you more deeply immerse yourself in studying GNU Make, Ant, SCons, CMake, and the Eclipse builders. By the time you finish reading Part I, you'll be in a good position to evaluate each of these build tools.

Chapter 1

Build System Overview

This first chapter provides a complete overview of software build systems. Before diving into the details of how a build system works, it's important to understand the high-level process of building software. This chapter also acts as a roadmap for the rest of the book.

The most common goal of a build system is to translate human-readable source code into an executable program. In addition, build systems support the packaging of web-based applications, the generation of documentation, the automatic analysis of source code, and many related activities. Although the exact details of this process vary for each programming language and for each operating system, the basic concepts are universal.

This chapter starts with an end-to-end view of a few common build system scenarios. You then get an introduction to some of the high-level concepts involved; later chapters cover the finer details of each topic. By the end of this chapter, you'll understand each of the main steps in the build process, along with the common build-related concepts and terminology.

What Is a Build System?

With such a wide range of programming languages and development environments, no single model can represent all possible build systems. A build system can manage any type of activity that involves translating one form of data (the input) into another form of data (the output). This discussion focuses on constructing software, hence the emphasis on *software* build systems.

In any software development environment, you're likely to encounter the following build-related scenarios:

- The compilation of software written in traditional compiled languages, such as C and C++. This can be extended to include newer languages such as Java and C#.
- The packaging and testing of software written in interpreted languages such as Perl and Python.
- The compilation and packaging of web-based applications. These include static HTML pages, source code written in Java or C#, hybrid files written using JSP (JavaServer Pages), ASP (Active Server Pages), or PHP (PHP: Hypertext Preprocessor) syntax, along with numerous types of configuration file.
- The execution of unit tests to validate small portions of the software in isolation from the rest of the code.
- The execution of static analysis tools to identify bugs in a program's source code. The output from this build system is a bug report document rather than an executable program.
- The generation of PDF or HTML documentation. This type of build system consumes input files in a range of different formats but generates human-readable documentation as the output.

Of course, this list isn't exhaustive, and you can probably think of many other uses for a build system. To simplify the discussion, this book focuses primarily on the traditional model of compiled languages. It's important to note that many of the build system concepts are the same, no matter what you're building.

Compiled Languages

Figure 1.1 depicts the high-level view of a traditional build system for compiled languages such as C, C++, Java, and C#. In this model, source files are compiled into object files, which are then linked into code libraries or executable programs. The resulting files are collected into a release package that can be installed on a target machine. This model should be quite familiar to software developers.

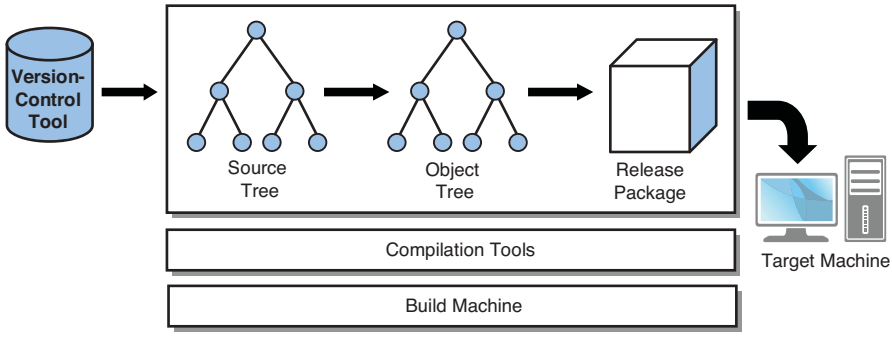


Figure 1.1 *Overview of a traditional build system for compiled languages.*

The key components of Figure 1.1 are listed here:

- **Version-control tool:** A tool that stores the program's source code and enables multiple developers to make concurrent changes to the code base. It also facilitates the retrieval of historical versions of the code. Common examples of a version-control tool include CVS [2], Subversion [3], Git [4], and ClearCase [5].
- **Source trees and object trees:** The set of source files and compiled object files that a particular developer works with. Developers can make their own private changes in these trees, without impacting other people.
- **Compilation tools:** The tools that take input files and generate output files (for example, converting source code files into object code and executable programs). Common examples of compilation tools include a C or Java compiler, but they also include documentation and unit test generators.
- **Build machines:** The computing equipment on which the compilation tools are executed.
- **Release packaging and target machines:** The method by which the software is packaged, distributed to end users, and then installed on the target machine.

Each of these topics is discussed in more detail, both later in this chapter and later in this book. Many of these topics are so detailed that they warrant a full chapter of their own.

Interpreted Languages

For interpreted languages, the build system model is slightly different (see Figure 1.2).

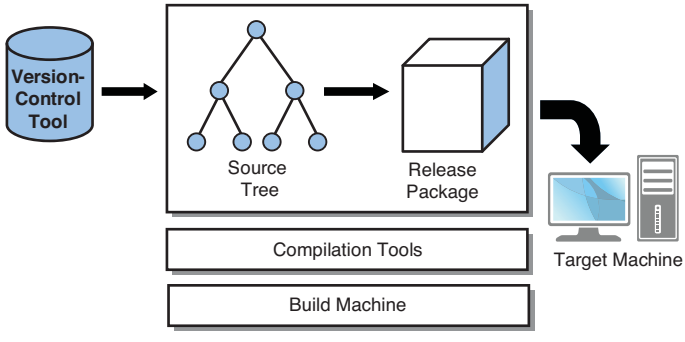


Figure 1.2 *Overview of a build system for interpreted languages.*

Interpreted source code isn't compiled into object code, so there's no need for an object tree. Instead, the source files themselves are collected into a release package, ready to be installed on the target machine. If compilation tools are required in this type of build system, which they often are, their focus is on transforming source files and storing them in the release package. Compilation into machine code is not performed at build time, even though it may happen at runtime.

Web-Based Applications

The build system for a web-based application is a mix of compiled code, interpreted code, and configuration or data files. As Figure 1.3 shows, some files (such as HTML files) are copied directly from the source tree to the release package, whereas others (such as Java source files) are first compiled into object code. In addition, both the web application server and the end user's web browser play a role in interpreting or compiling code, but that's beyond the scope of this build system.

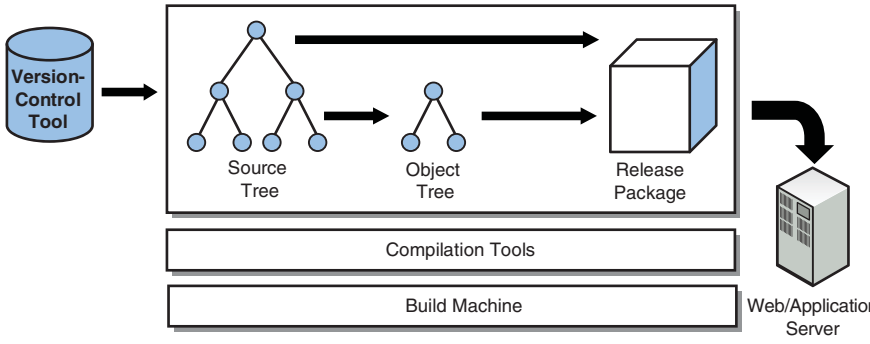


Figure 1.3 *Overview of a build system for web-based software.*

A typical web application deals with many of the following file types:

- Static HTML files, containing nothing more than marked-up data to be displayed in a web browser. These files are copied directly to the release package.
- JavaScript files containing code to be interpreted by an end user's web browser. These files are also copied directly to the release package.
- JSP, ASP, or PHP pages, containing a mix of HTML and program code. These files are compiled and executed by the web application server rather than by the build system. These files are also copied to the release package, ready for installation on the web server.
- Java source files to be compiled into object code and packaged as part of the web application. The build system performs this transformation before packaging the Java class files. The Java classes are executed on the web application server or even within the web browser (using a Java applet).

Of course, there's no reason that the build system can't autogenerate some of these HTML, JavaScript, or JSP/ASP/PHP files (from other input file formats). Many compilation steps might take place before the output is finally copied to the release package.

Unit Testing

The build system for a unit testing environment is simply an extension of the models already discussed. Instead of producing a release package to be installed on the target machine, the build system produces a number of smaller unit test

suites. Each suite is executed on the target machine and produces a "pass" or "fail" result to indicate whether the software behaved as expected.

Figure 1.4 shows how the traditional compiled language build system (shown in Figure 1.1) can be extended to generate unit tests rather than a standard release package.

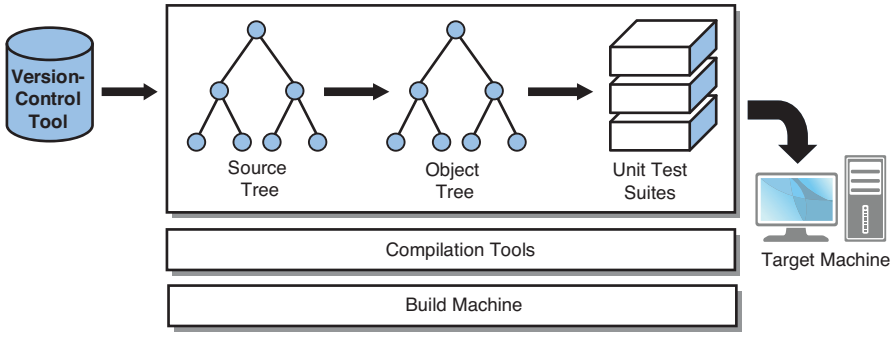


Figure 1.4 *Overview of a build system that generates unit tests.*

For interpreted languages (see Figure 1.2) and web-based applications (see Figure 1.3), a similar unit test build system can be created. In fact, a unit test build system is simply a variant of a standard build system. Chapter 12, "Building with Metadata," discusses unit testing in more detail.

Static Analysis

Figure 1.5 shows a build system that performs static analysis. A static analysis tool, such as Coverity Prevent [6], Klocwork Insight [7], and FindBugs [8], examines a program's source code with the goal of identifying potential bugs. The analysis is done statically (at build time) instead of the more common approach of executing the software to see if it behaves correctly.

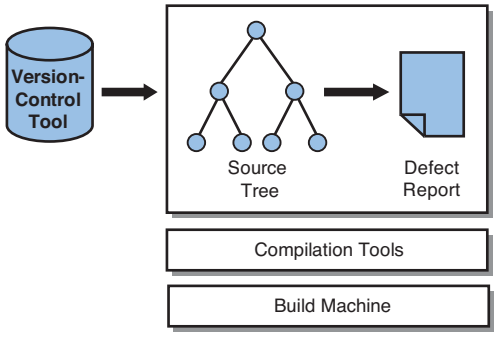


Figure 1.5 *Overview of a build system for static analysis.*

The input to a static analysis system is the same source code used in a regular build system. However, instead of generating an object tree and release package, the output is some type of defect report document (often in text or HTML format). Chapter 12 discusses static analysis in more detail.

Documentation Generation

The final build system scenario considers the generation of human-readable documentation, as shown in Figure 1.6.

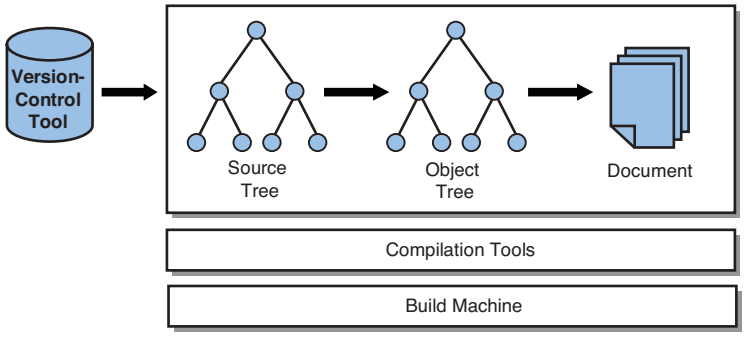


Figure 1.6 *Overview of a build system for generating documentation.*

The output from this build system is a PDF file, a collection of HTML pages and graphic images, or anything else that could be considered documentation. Generating documentation might also include a number of intermediate data files, so the concept of an object tree still applies. No target machine is mentioned in this case, although, technically, the document would need to be viewed in some way, whether via a printer, a web browser, or a PDF viewer.

In summary, a build system can be used for many different purposes. This book focuses more on build systems for traditional compiled languages, although the concepts are the same for other scenarios.

The important point to understand for now is the process by which a build system operates. Although Figures 1.1–1.6 don't show it, a **build tool** is used to orchestrate the entire build process. Common build tools include GNU Make, Ant, SCons, CMake, and the Eclipse builders; Part II, "The Build Tools," discusses each one.

Components of a Build System

Now that you've seen the high-level view of a software build system, you can dig deeper in each of the main sections. Later chapters cover many of these topics, so for now you'll cover only the basics.

Version-Control Tools

Although you won't explore version-control systems until Chapter 14, "Version Management," a version-control tool is the first component of a build system. Before any software can be compiled, the developers must obtain a private copy of the source code. As part of their assigned work (fixing a bug or adding a new feature), each developer changes the appropriate source files and then triggers the build system to compile the software.

Version-control tools enable you to perform a number of operations:

- Obtain a copy of the source code, ready for private modifications to be made.
- Control **check-ins** or **commits** so that private changes can be made available for other developers to use.
- Facilitate the creation of multiple code streams to manage the development and maintenance of different versions of the same product.
- Control access to files so that only authorized developers can change certain source files.
- Enable a developer to view older (historical) versions of each source file, even if newer revisions have superseded them.

This isn't a book about version control, so it doesn't discuss specific version-control tools. However, Chapter 14 focuses extensively on the many ways in which the build system must interact with a version-control tool. There you'll consider which files should or shouldn't be kept under version control, and you explore the use and management of version numbers.

The next section focuses more on the source code stored within the version-control system.

Source and Object Trees

As you might expect, a program's source code is stored as a number of disk files. This arrangement of the files into different directories (or folders, in Windows terminology) is known as the **source tree**. The way in which the source code is structured within the source tree has a significant impact on the design of the build system.

The structure of the source tree often reflects the architecture of the software. Figure 1.7 illustrates how the source code files for a Microsoft Windows-based accounting application can be stored, based on the various major components of the system.

Notice that each directory contains a file named Makefile. The implication here is that you use Make to build the software, which is common only for older Windows applications. The build description files (known as makefiles) are stored in the same directory as the source files they describe. This isn't the only way to store the build description, but it does make it easy to locate the parts of the build system that deal with the files in each directory.

Alongside the source tree is the **object tree** (see Figure 1.8). Although it's entirely possible to store object files in the same directory as the source files, it's often considered a messy approach (as you see in later chapters). You should instead create a separate tree hierarchy that stores any object files or executable programs constructed by the build process. Notice that Figure 1.8 contains not only object files, but also the final executable program (accounting.exe).

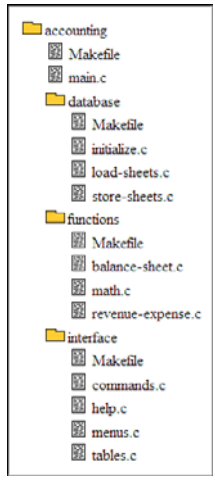


Figure 1.7 *Source tree for a small Microsoft Windows application.*

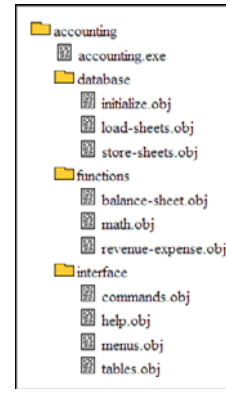


Figure 1.8 *Corresponding object tree for a small Microsoft Windows application.*

Although a small program such as this accounting application could be stored inside a single source code file, this is unrealistic for larger programs. Several important considerations call for dividing a program into multiple source files and then placing those files into different directories on the disk:

- **Comprehension:** Conceptually, people find it easier to think about programs when they're divided into logical subsections. This is the basic premise of object-oriented programming, in which people can think about the program as a collection of different classes. Each class must have both an external behavior that programmers can keep fresh in their minds and an internal implementation that hides the complexity of the class away from view. In a build system, therefore, it's best to divide the source code into multiple sections, each encapsulating a specific area of the program's functionality.
- **Source code control:** When a program's source code is spread across multiple files and directories, it becomes easier to manage them with a source code control tool. Conversely, if the entire program was stored inside a single disk file, it would be challenging for different developers to submit code changes without constantly stepping on each other's work.
- **Performance:** Development tools such as editors and compilers perform much more efficiently with smaller units of work. Although these tools are capable of dealing with source files that are megabytes in size, they do so inefficiently.

Throughout this book, you'll learn more about the design and construction of source and object trees.

Compilation Tools and Build Tools

When developers have a source tree to work with, they must have some way to translate the human-readable source files into the machine-readable executable program. A **compilation tool** is a program that reads input files and translates them into output files. This might sound like a generic statement, but there's no limit to the type of data translation these tools could undertake.

The following are common examples of compilation tools:

- **C compiler:** Reads human-written C language source files and produces object files that contain a machine code translation of that same program. In this scenario, the output from the compilation tool should be functionally equivalent to the input, although closer to what the target machine can understand.
- **Linker:** Joins a number of different object files to produce a single executable program image. In this case, the object files are the input to the linker tool, but in the previous build step, they were the output from a compiler. In this example, it makes more sense to talk about input and output files than source and object files.
- **UML-based code generator:** Reads a UML model file as input and produces an equivalent program written in a general-purpose programming language such as Java, C++, or C#.
- **Documentation generator:** Reads a human-written file written in a markup language and generates a PDF file (or similar) as output.
- **Command-line tool for making a new directory:** Creates a new directory on the file system (for example, using the UNIX mkdir command). In this scenario, the name of the new directory is the only input data provided.

At this point, it's worth noting the distinction between a compiler and a compilation tool. A compiler typically translates high-level programming language source code into object code, which is the first of the previous examples given. However, a compilation tool is defined as any tool that translates input data to output data.

In contrast, a **build tool** is a program that functions at a level above compilation tools. That is, it must have sufficient knowledge of the relationship between source files and object files that it can orchestrate the entire build process. The

build tool calls upon the necessary compilation tools to produce the final build output.

This book takes care to distinguish between compilation tools and build tools. Both play a critical role in creating a good build system, but they do so in different ways. Chapter 4, "File Types and Compilation Tools," looks at a number of compilation tools (such as gcc and javac) and explores how they manipulate the various types of files in the source and object trees. Chapter 16, "Tool Management," discusses some best practices for managing compilation tools over the lifetime of the software. Part II looks in more detail at build tools (Make, Ant, SCons, CMake, and the Eclipse builders) that orchestrate the entire build process.

Build Machines

It may not appear so at first, but the machine on which the compilation and build tools execute plays a vital role in the management of a build system. Each of the tools must be capable of executing on the build machine, even though the underlying machine hardware and operating system might change over time. As you'll learn in Chapter 15, "Build Machines," numerous issues surround the management of build machines, particularly when you need to reproduce older versions of software or to provide a uniform environment in which different developers can compile the same source code.

You must also consider whether the software itself is being compiled and executed on the same type of machine or whether the software is destined to run in a completely different environment (CPU type and operating system). Figure 1.9 illustrates both a **native compilation** environment and a **cross-compilation** environment. In the native case, the software is executed on a **target machine** that's identical to the **build machine**; the cross-compilation case requires two different machines, with a different operating system or CPU on the target machine.

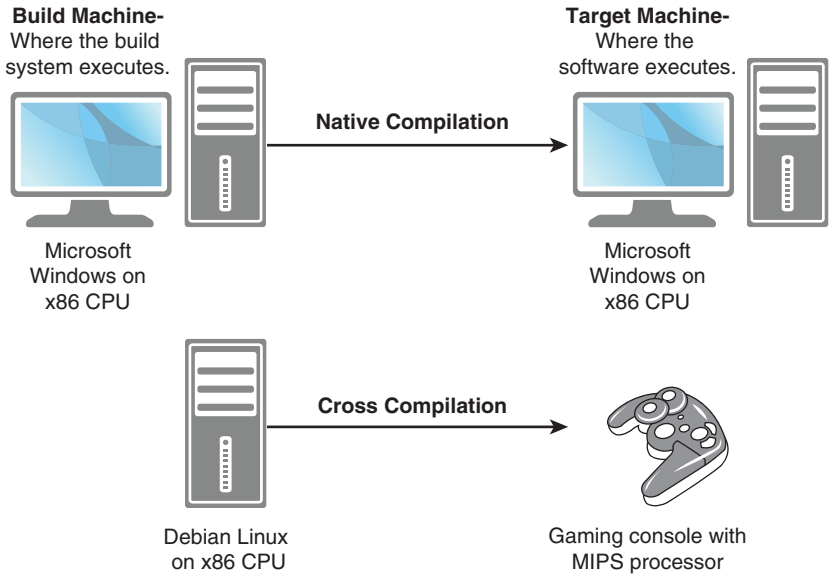


Figure 1.9 *Native compilation versus cross-compilation.*

You'll learn more about native and cross-compilation environments in Chapter 15, which studies build machines in more detail.

Release Packaging and Target Machines

Although much of a build tool's work focuses on generating object files and executable programs, the final packaging step produces something that you can actually install on a user's machine. It's not realistic to hand novice users a number of executable programs and data files and expect them to install and configure them by hand. Instead, you need to provide a single file that they can download, or a single CD or DVD that they can insert into their computer's CD-ROM drive. For software written for the home consumer market, the installation process should involve nothing more than double-clicking an icon and answering a few basic questions.

The final step of a build process is therefore to extract the relevant files from the source and object trees and store them in a **release package**. If at all possible, the release package should be a single disk file and should be compressed, to reduce the amount of time it takes to download or the number of DVDs required. Additionally, any nonessential debug information should be removed so that it doesn't clutter the software's installation.

Chapter 13, "Software Packaging and Installation," examines three common ways of packaging and installing software:

- **Archive files:** This is the most straightforward approach, with files compressed and joined into a single disk file. The end user must perform the reverse operation to install the software.
- **Package-management tools:** These are common in UNIX-like environments where complete software packages are downloaded from the Internet and installed as an optional part of the operating system. Installation is a one-step process, and any prerequisite packages are installed at the same time. Common examples include .rpm and .deb package files.
- **Custom-built GUI installation tools:** These are familiar to anyone who has installed software on the Microsoft Windows operating system. The installation process is started by double-clicking an icon, and the end user interacts with a custom-built GUI to install the software.

One final option, which isn't discussed in detail here, is that the software may be partially installed yet partially accessed at runtime. A portion of the software is installed on the end user's computer, but the rest of the code and data is accessed when the program is running. Common examples include video games in which graphic images, movies, and sound files are loaded off the DVD whenever they are required, but are never stored on the target machine's hard disk. Additionally, tools such as Google Earth [9] require that a client program be installed, but the rest of the data is downloaded from the Internet when required.

The generation of a release package marks the end of the software build process. The next section considers how this process is implemented within a build tool.

The Build Process and Build Description

Now that you've covered each part of the build system at a high level, take a brief look at a couple of examples. In Figure 1.10, you can see the process by which the build tool invokes each of the compilation tools to get the job done (using the traditional compiled languages model shown in Figure 1.1). This end-to-end sequence of events is known as the **build process**.

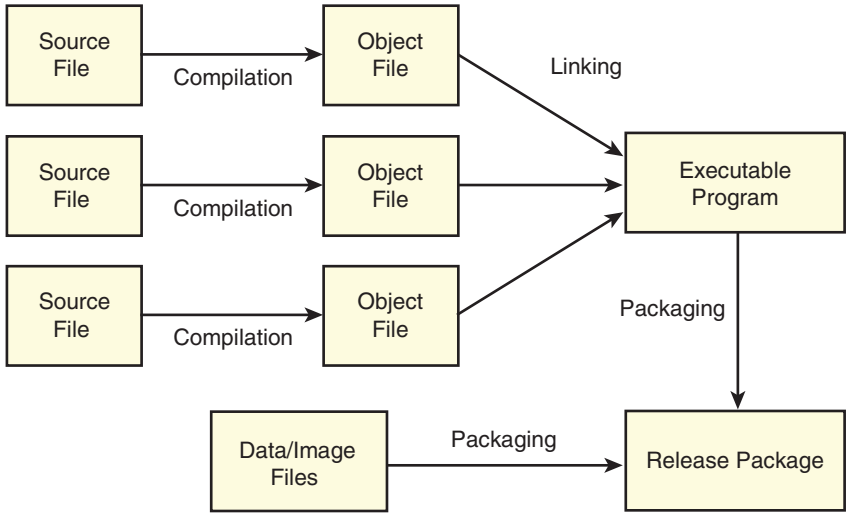


Figure 1.10 *Overview of a build system for compiled languages.*

Although it's easy for humans to visualize this process in the form of a diagram, a build tool needs the **build description** to be written in a text-based format. For example, when using Make, the interfile dependency information is specified in the form of **rules**, which are stored in a file named Makefile. In contrast, the SCons build tool uses Python-language functions to describe the compilation steps; it keeps this information in a file named SConstruct.

To illustrate, the following SCons build description file states that the stock program should be generated by compiling the source files, `ticker.c` and `currency.c`.

```
Program("stock", ["ticker.c", "currency.c"])
```

In this case, SCons uses the default C compiler to create `ticker.o` and `currency.o`, even though the build description does not explicitly state that step. It then links those object files into the final executable program, `stock`. Figure 1.11 shows the equivalent diagram, to help you visualize the individual steps in the process:

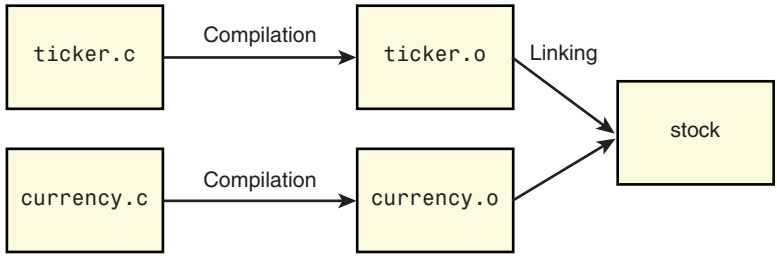


Figure 1.11 *Overview of a SCons-based build process.*

Because the example stock program consists of only a small number of source files, the build description remains simple and fits nicely into a single text file. For larger programs (with thousands of source files in the code base), the build description may consist of hundreds of small files that work together to capture the build recipe for the entire program.

From a software developer's perspective, the text-based build description is at the heart of the whole build process. Every build tool has its own syntax for describing the build process, including file dependencies and compilation commands. You'll learn more about these build description languages in Part II.

How a Build System Is Used

In a software development organization, three different types of software build are commonly performed. Each uses the same build system, but the end purpose of the build is different:

- **Developer (or private) build:** The developer has checked out the source code from version control and is building the software in a private workspace. The resulting release package will be used for the developer's private development instead of being shared with other people. The developer makes source code changes many times a day, incrementally recompiling the software each time.
- **Release build:** One or more people, known as **release engineers**, are assigned to perform release builds. The sole purpose is to provide a complete software package for the test group to validate. When the testers are convinced that the software is of high enough quality, that same package is made available to customers. The source tree used for a release build is compiled only once, and the source tree is never modified.

- **Sanity build:** This is similar to a release build, except that the software package isn't destined for a customer. Instead, the build process determines whether the current source code in the version-control system is "sane"—that is, whether the software build is free of errors and passes a basic set of sanity tests. This type of build can occur many times per day and tends to be fully automated. Many developers use the terms **daily build** or **nightly build** to describe this scenario.

As you can see, the key distinction among these three scenarios is how the build system is used—how often it's invoked and how the final program image is used. For the purposes of this book, the upcoming chapters don't discuss these topics in much detail, unless there's a need to distinguish how the build system accommodates each type of user.

Build-Management Tools

The use of **build-management** tools has increased in recent years. Given the focus of this book, a build-management tool should be viewed as an extra layer of management on top of an existing build system rather than as part of the build system itself. Figure 1.12 illustrates the distinction.

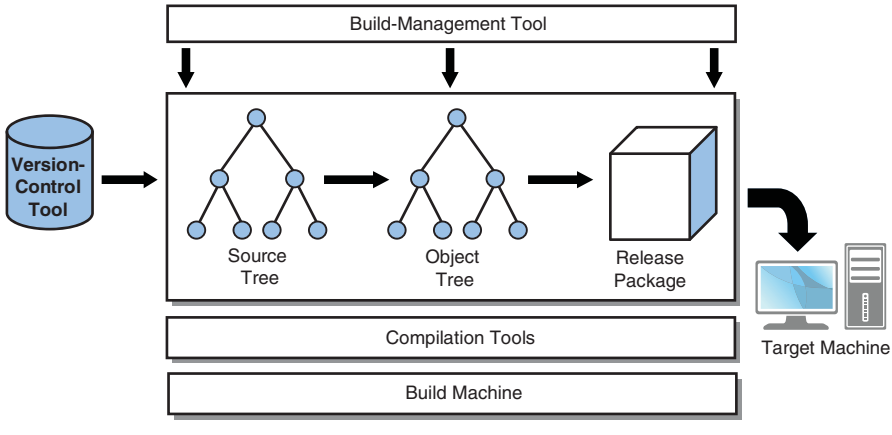


Figure 1.12 *The use of a build-management tool, to oversee the use of a build system.*

The build-management tool communicates with the version-control tool to check out a build tree, calls upon the build system to compile the software, and then informs the developer when the build is complete. Depending on your perspective, you may view a build-management tool as just another part of the build system, but this book keeps them separate.

A good build-management tool provides the following features:

- Checks out and builds a source tree on a predetermined schedule, or simply when new code has been committed.
- Provides a queuing mechanism so that multiple build jobs can share a pool of build machines. When a sufficient number of machines are available, the next job is started.
- Sends email notification messages to various groups of users (when the build starts, completes, succeeds, or fails).
- Provides a graphical user interface to show when builds took place and whether they failed or succeeded.
- Manages version numbers, incrementing them after each successful build.
- Stores the final software package in an archive directory, ready for testers to use.
- Starts executing sanity tests on any successful build.
- Can identify which developers are on the "guilty list" of people who may have recently checked in bad code.

A build-management tool is vital for any software projects that have more than a few developers. A number of tools are available, either commercially developed and supported or from the open-source world. Some of these common tools include Build Forge [10], ElectricCommander [11], CruiseControl [12], and Hudson [13]. With the wide range of tools available, you can easily find something that meets your needs, and you won't need to implement your own build-management solution.

Aside from this brief introduction to build-management tools, this book doesn't cover the topic in any detail. Instead, it focuses on all the build system functionality below the build-management tool (see Figure 1.12). For a good overview of build management and the concepts of continuous integration, refer to [14].

Build System Quality

As with any software-related topic, a number of system attributes define whether it's perceived as high quality, low quality, or somewhere in between. According to one build tool expert [15], a good build system should have the following characteristics:

- **Convenience:** The tool and the description files should be easy to use and should not place too much burden on the software developers who need to use them. The developer should focus on writing source code rather than dealing with the complexities of the build tool.
- **Correctness:** The build tool should always compile/link the correct files, using the correct compiler options. When it matters, the tool should compile the files in the correct order so that the final executable program always reflects the content of the source files.
- **Performance:** In an ideal world, the build process would complete without any noticeable delay. Realistically, though, it must perform as fast as possible for the computing equipment it's running on.
- **Scalability:** The build tool must be convenient, provide correct release images, and perform well, even when the tool is building a large program (for example, with thousands of source files). Part IV, "Scaling Up," discusses this topic of scalability.

The rest of this book spends a lot of time examining both good and bad ways to create a build process, and the pros and cons of using a range of different build tools. The book makes a special effort to consider these four characteristics, because they're important in the operation of a build system.

Summary

This chapter offered a high-level overview of a complete **build system** and introduced the terminology for describing the steps in an end-to-end **build process**. Due to the wide range of build-related applications, there is no single type of build system.

The first step in a build system is usually to store and control access to the source code using a **version-control tool**. Next, you make source code changes in a **source tree** and generate the object files into the corresponding **object tree**. This depends on whether you're compiling source code or working with an interpreted language.

A **build tool** handles the end-to-end management of the build process. These build tools orchestrate the use of **compilation tools** to generate object files from source files (or whatever makes sense for the file types being used). Each of these tools must execute on the **build machine**.

The end product of the build system is called a **release package**. This is usually an archive file or an installation program that's capable of installing the software on the **target machine**. In some cases, the output of the build system is a documentation file rather than an executable file.

For the build tool to understand the details of the build process, you must create a suitable text-based file known as a **build description**. For example, with the SCons tool, the build description must be written in the Python programming language and stored in a file named `SConstruct`.

Chapter 2

A Make-Based Build System

One of this book's key assumptions is that you already have experience in developing software. However, this doesn't mean that you have experience writing your own build system, or even understanding an existing system. Many developers work on projects in which other people create and maintain the build system, or perhaps use an integrated development environment (IDE) to build at the push of a button. In either of these cases, you may not see the underlying build system.

This chapter introduces a build system for a small C-language program with only five source files. The build system is implemented using GNU Make [16] syntax, not only because it's an extremely popular tool, but also because Make syntax helps you understand the fundamental concepts underlying any build system.

If you've never written a makefile, take the time to study this example before moving to the more advanced concepts. Many of this book's examples use Make syntax, so understanding these concepts is important.

If you're already experienced with makefile syntax, feel free to skip forward to the next chapter. Chapter 6, "Make," presents more advanced details of the GNU Make tool.

Calculator Example

This chapter uses a simple calculator program as its running example. You don't need to understand how the program works, other than knowing that it contains five C-language source files: Four are `.c` files (`add.c`, `calc.c`, `mult.c`, and `sub.c`), and the fifth is a `.h` file (`numbers.h`). In the C language, files with a .c suffix contain the main body of the source code, whereas files ending with `.h` provide type, variable, and function definitions to be shared by all `.c` files. Everything is then linked together into a single executable program, named `calculator`.

Here's the content of the source code directory, before anything is compiled:

```
$ ls
add.c  calc.c  mult.c  numbers.h  sub.c
```

Figure 2.1 shows a corresponding source tree diagram, with all files in the same directory. Source trees are a fundamental part of a build system, so you'll see many of these diagrams throughout this book. As you can imagine, the build system for this program is one of the simplest you can create, other than the standard "Hello World" program.

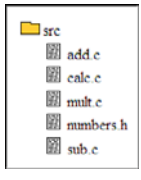


Figure 2.1 *The source tree for a simple calculator example.*

In the C programming language, each `.c` file is compiled into a single object file containing the compiled machine code instructions (`.o` suffix in UNIX-like systems, or `.obj` in Window systems). With four different `.c` files, you can expect four different compilation commands, each producing a unique `.o` file. You'll use the GNU C Compiler [17], commonly known as GCC, with all examples performed in a UNIX environment.

```
$ gcc -g -c add.c
$ gcc -g -c calc.c
$ gcc -g -c mult.c
$ gcc -g -c sub.c
```

In each gcc command, the `-c` option requests that an object file be created, with the `-g` option requesting that debugging be enabled. You'll learn more about GCC in Chapter 4, "File Types and Compilation Tools."

The source code directory now contains a few more files:

```
$ ls
add.c  calc.c  mult.c  numbers.h  sub.o
add.o  calc.o  mult.o  sub.c
```

If you look carefully, you see that each `.c` file has a corresponding `.o` file. Note that `numbers.h` doesn't have an object file; instead, it was included (imported) by the `add.c, calc.c, mult.c,` and `sub.c` files. In build system terminology, each of the `.c` files is dependent on `numbers.h`.

To build the final calculator program, these `.o` files are linked together into a single executable file.

```
$ gcc -g -o calculator add.o calc.o mult.o sub.o
$ ls
add.c  calc.c  calculator  mult.o     sub.c
add.o  calc.o  mult.c      numbers.h  sub.o
```

That completes the entire process of building the `calculator` program. To illustrate this graphically, consider the concept of a **dependency graph**, shown in Figure 2.2.

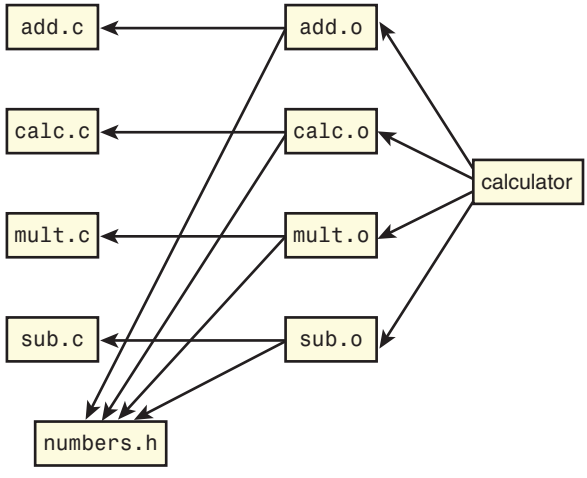


Figure 2.2 *The dependency graph for the simple calculator example.*

For several reasons, a dependency graph is important in build systems. Not only does it list the files involved in the build process, but it also shows the dependencies between those files. A **build tool** such as GNU Make uses a dependency graph to determine which files should be compiled, and when they should be compiled.

For example, the arrows originating from `add.o` toward both `add.c` and `numbers.h` state that both these sources files contribute to the compilation of `add.o`. Additionally, if either of these files is edited, `add.o` must be recompiled to include any recent changes. Conversely, if neither `add.c` nor `numbers.h` has changed since the last time `add.o` was compiled, it doesn't need to be compiled again.

With these concepts in mind, you'll now explore how GNU Make enables you to specify the dependency graph for the example program. This book spends a lot of time looking at different build tools (such as GNU Make, Ant, SCons,

CMake, and the Eclipse builders), to show different ways of specifying a build system's dependency graph.

Creating a Simple Makefile

This section examines how the example can be implemented using the GNU Make build tool. A dependency graph is a purely mathematical concept, so you need some way to express the graph in a source code format. This should use plain text to list the files, describe the dependencies between them, and show which compiler commands are to be used. The GNU Make tool offers a straightforward translation.

The following text file, called Makefile, is stored in the same directory as the source and object files.

```
 1  calculator: add.o calc.o mult.o sub.o
 2          gcc -g -o calculator add.o calc.o mult.o sub.o
 3
 4  add.o: add.c numbers.h
 5          gcc -g -c add.c
 6
 7  calc.o: calc.c numbers.h
 8          gcc -g -c calc.c
 9
10  mult.o: mult.c numbers.h
11          gcc -g -c mult.c
12
13  sub.o: sub.c numbers.h
14          gcc -g -c sub.c
```

As you'll see when you study GNU Make in more detail (see Chapter 6), this is an inefficient way of implementing a makefile. However, this direct translation of the dependency graph is easy to understand.

Each section of the makefile introduces a new **rule**. Line 1 of the listing states that the file named `calculator` is dependent on all the files, `add.o, calc.o, mult.o,` and `sub.o`. Line 2 then provides a UNIX command to generate the calculator file from all those object files.

Line 4 specifies that `add.o` depends on both `add.c` and `numbers.h`, and line 5 provides the UNIX command for compiling `add.o`. The rest of the makefile provides similar rules for the other source and object files.

One important warning is that all UNIX commands (lines 2, 5, 8, 11, 14) must be preceded by a TAB character instead of spaces. This feature is historic and confuses many new makefile developers. If you forget this rule, you see the following error: