Build Management

Kenneth M. Anderson
University of Colorado, Boulder

CSCI 5828 — Lecture 17 — 03/09/2010

Goals

- Review material from Chapter 6.5 of Pilone & Miles
  - Build Management
    - How do you build your code
  - Examples
    - make
    - Ant
    - IDEs

Build Management (I)

- The process for constructing a system should be engineered
  - What are the steps to build a system?
    - what subsystems need to be built?
    - what libraries are needed?
    - what resources are required?

Build Management (II)

- The process for constructing a system should be engineered
  - Who is authorized to build a system?
    - Small projects: individual programmers
    - Large projects: build managers and/or configuration managers

Build Management (III)

- The process for constructing a system should be engineered
  - When are system builds performed?
    - e.g. perhaps a system is so large that it can only be built at night when there are enough resources available…

Build Management (IV)

- Most modern programming environments have build management capabilities built into them
  - For instance, a Java development environment typically has the notion of a “project” and it can compile all project files in the correct order (and it only compiles files dependent on a change)
- These capabilities free developers from accidental difficulties
  - having to remember the correct compilation order
  - correctly identifying all files dependent on a change

Make: The Granddaddy of Build Management Systems

- In Unix, a common build management tool is “make”
  - Make provides very powerful capabilities via three types of specification styles
    - declarative
    - imperative
    - relational
  - These styles are combined into one specification
    - “the make file”

Why talk about Make?

- In modern projects, *make* is not used directly
  - IDEs: have build management features built in
  - modern build tools: *ant*, *maven*, etc. operate at a higher level
  - open source environments: *autoconf* and *configure* generate makefiles: developers write *configure* specs, *autoconf* does the rest
- The reason?
  - The same reason calculus students learn how to do limits the “hard way” before they are taught l'Hôpital's rule

Specification Styles?

- **Operational (or Imperative)**
  - Described according to desired actions
  - Usually given in terms of an execution model
- **Descriptive (or Declarative)**
  - Described according to desired properties
  - Usually given in terms of axioms or algebras
- **Structural (or Relational)**
  - Described according to desired relationships
  - Usually given in terms of a graph
    - e.g. UML class diagrams

make Specification Language

- Hybrid Declarative/Imperative/Relational
  - Dependencies are Relational
    - Make specifies dependencies between artifacts
  - Rules are Declarative
    - Make specifies rules for creating new artifacts
  - Actions are Imperative
    - Make specifies actions to carry out rules
- This is true of *ant* and other tools with similar specs.

Example makefile

```
Target1: Target2 Target3 … TargetN
  \t     Action1
  \t     Action2
  \t     …
  \t     ActionN
Target2: Target5 Target6
  \t     Action3
Target3: Target5 Target7
  \t     Action4
```

A Makefile consists of a set of rules.

Each rule contains a target followed by a colon followed by a list of dependencies

Each subsequent line of a rule begins with a tab character (required) followed by an action

If a dependency changes, make invokes a rule's action to recreate the target

What would happen if Target5 changed?

Power from Integration

- make is well integrated into the Unix environment
  - Targets and Dependencies are file names
  - Actions are shell commands

```
program: main.o input.o output.o
    g++ main.o input.o output.o -o program

main.o: main.cpp defs.h
    g++ -c main.cpp

input.o: input.cpp defs.h
    g++ -c input.cpp

output.o: output.cpp defs.h
    g++ -c output.cpp
```

When you realize that any shell command can go here, you begin to grok the power of make

It is possible to automate the creation and deployment of large systems with make

Why use make at all?

- Why use all the complexity of multiple specification styles when ultimately *make* just invokes shell commands?
  - Why not just write a shell script?

```
#!/bin/bash
g++ -c main.cpp
g++ -c input.cpp
g++ -c output.cpp
g++ main.o input.o output.o -o program
```

What style does this specification use?

Why not use a shell script?

- The (Primary) Answer
  - A shell script will compile each file every time its run… even if the file has not changed since the last compilation!
- When building large systems, such an approach does not scale!
  - You only want to recompile changed files and the files that depend on them
- Make is much “smarter”
  - by only recompiling changed files and their dependencies, make can scale to building large software systems

make wrap-up

- Build management has been around a long time
  - make was created by Stuart Feldman in 1977
    - Feldman was part of the group that created Unix at Bell Labs
    - He was an author of the first Fortran 77 compiler
    - Now works for Google as Vice President of Engineering (East Coast)
- When you click “build” in your IDE and it builds your project, you have *make* to thank

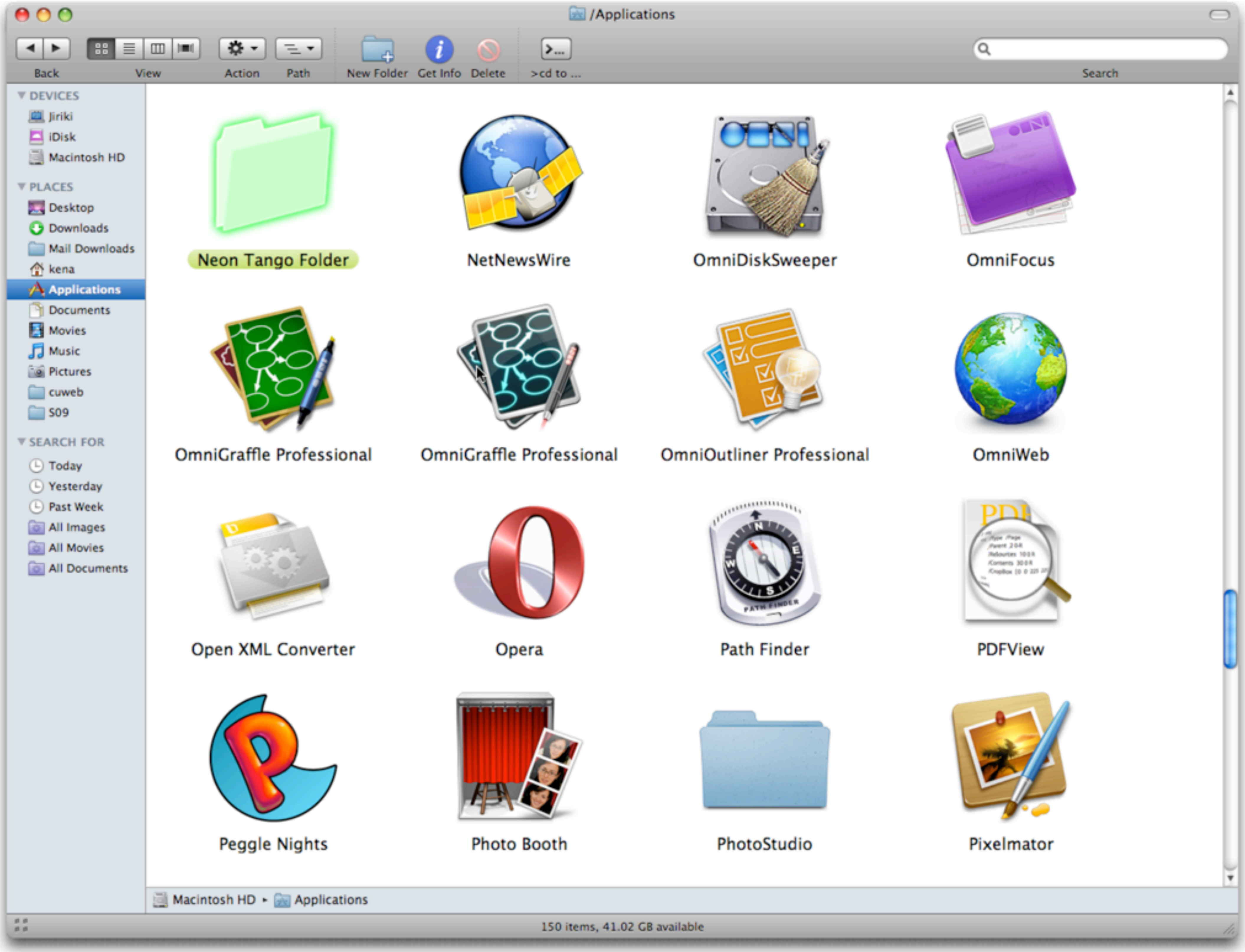
The Textbook Scenario

- The book highlights another reason for build management
  - Configuration Management is not enough to support the day to day tasks of software development
  - If a new developer joins the team, simply checking out a copy of HEAD is not enough
    - How do I compile the system?
    - A search finds five main() methods, which one do I invoke?
    - What configuration do I have to do before the system will run?

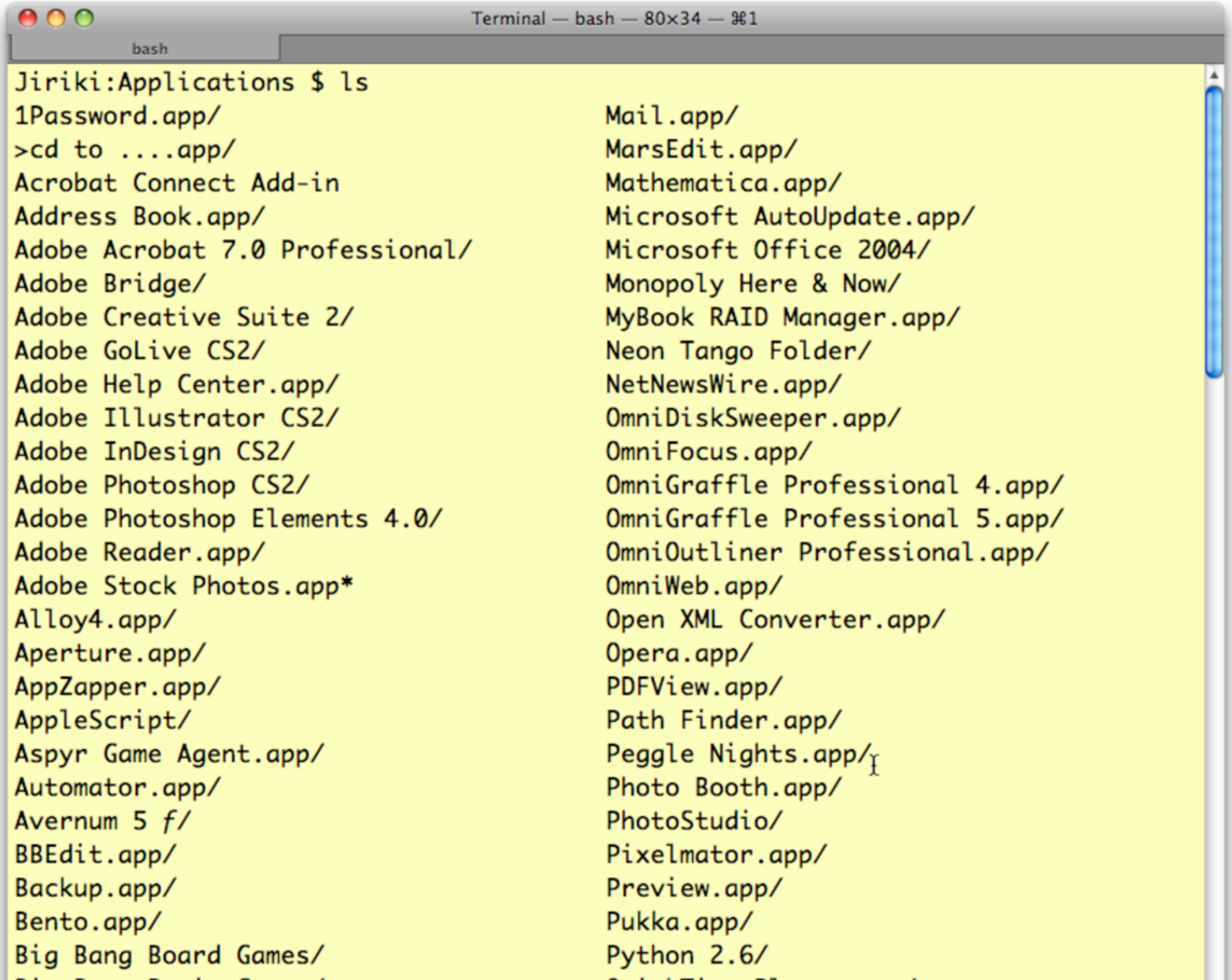
Building your project in one step

- Build management is all about reducing the complexity of creating your system
  - You do some work up front
  - Then invoke a single command: “make” or “ant”
  - Then run your system
- As the book says, modern applications are complex beasts
  - consisting of not just code, but libraries (aka frameworks), resources (images, sounds, movies, etc.) and more

Example: Applications in Mac OS X Finder

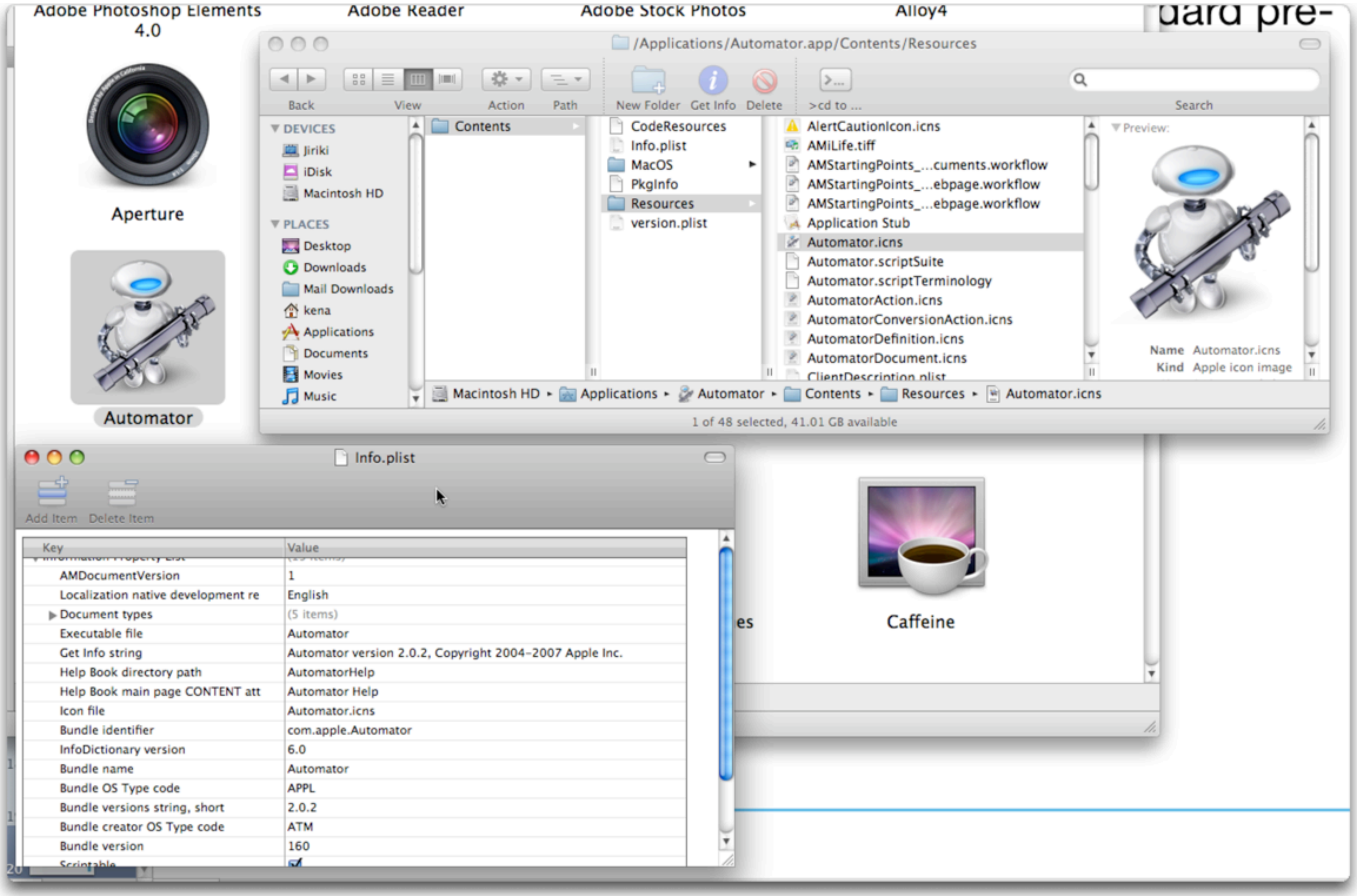


But look behind the curtain: Apps in Mac OS X Terminal



```
Jiriki:Applications $ ls
1Password.app/                        Mail.app/
>cd to ....app/                       MarsEdit.app/
Acrobat Connect Add-in                Mathematica.app/
Address Book.app/                     Microsoft AutoUpdate.app/
Adobe Acrobat 7.0 Professional/       Microsoft Office 2004/
Adobe Bridge/                         Monopoly Here & Now/
Adobe Creative Suite 2/               MyBook RAID Manager.app/
Adobe GoLive CS2/                     Neon Tango Folder/
Adobe Help Center.app/                NetNewsWire.app/
Adobe Illustrator CS2/                OmniDiskSweeper.app/
Adobe InDesign CS2/                   OmniFocus.app/
Adobe Photoshop CS2/                  OmniGraffle Professional 4.app/
Adobe Photoshop Elements 4.0/         OmniGraffle Professional 5.app/
Adobe Reader.app/                     OmniOutliner Professional.app/
Adobe Stock Photos.app*               OmniWeb.app/
Alloy4.app/                           Open XML Converter.app/
Aperture.app/                         Opera.app/
AppZapper.app/                        PDFView.app/
AppleScript/                          Path Finder.app/
Aspyr Game Agent.app/                 Peggle Nights.app/
Automator.app/                        Photo Booth.app/
Avernum 5 f/                          PhotoStudio/
BBEdit.app/                           Pixelmator.app/
Backup.app/                           Preview.app/
Bento.app/                            Pukka.app/
Big Bang Board Games/                 Python 2.6/
```

Apps are just directories whose names end in the suffix .app; those pretty icons just live in a standard pre-defined place in the “application bundle” or “package”



/Applications/Automator.app/Contents/Resources

DEVICES: Jiriki, iDisk, Macintosh HD
PLACES: Desktop, Downloads, Mail Downloads, kena, Applications, Documents, Movies, Music

Contents

CodeResources
Info.plist
MacOS
PkgInfo
Resources
version.plist

AlertCautionIcon.icns
AMiLife.tiff
AMStartingPoints_...cuments.workflow
AMStartingPoints_...ebpage.workflow
AMStartingPoints_...ebpage.workflow
Application Stub
Automator.icns
Automator.scriptSuite
Automator.scriptTerminology
AutomatorAction.icns
AutomatorConversionAction.icns
AutomatorDefinition.icns
AutomatorDocument.icns
ClientDescription.plist

Preview:
Name Automator.icns
Kind Apple icon image

Macintosh HD ▸ Applications ▸ Automator ▸ Contents ▸ Resources ▸ Automator.icns

1 of 48 selected, 41.01 GB available

Info.plist

Key	Value
AMDocumentVersion	1
Localization native development re	English
▶ Document types	(5 items)
Executable file	Automator
Get Info string	Automator version 2.0.2, Copyright 2004-2007 Apple Inc.
Help Book directory path	AutomatorHelp
Help Book main page CONTENT att	Automator Help
Icon file	Automator.icns
Bundle identifier	com.apple.Automator
InfoDictionary version	6.0
Bundle name	Automator
Bundle OS Type code	APPL
Bundle versions string, short	2.0.2
Bundle creator OS Type code	ATM
Bundle version	160
Scriptable	☑

How complex?

- Lets take a look at the application bundle for
  - OmniGraffle Pro
    - The application I use to create all of my diagrams
- As we will see, it contains
  - Code
  - Frameworks, Libraries, Plugins, Scripts
  - Images (tiff, png, icons, …), color pickers (!)
  - nib files (“frozen” objects), “localized” files for internationalization, etc.

Ant

- The book delves into the details of Ant
  - Ant is a build system that is used mainly for Java-based software development
- The specification is contained in an XML file called “build.xml”
  - This specification consists of
    - projects
    - properties
    - targets
    - tasks

Projects

- The build.xml file exists to build a single project
- <project name="BeatBox" default="dist">
- It defines the name of the project and its default target
    - The default target is the target that gets executed if "ant" is invoked with no arguments

Properties

- Properties allow you to define values that might change
- <property name= “version” value = “1.1” />
- <property name= “src” location= “src” />
  - Note: location field supports both absolute and relative paths
- Build scripts == Code
  - Since build scripts are executable, we want to apply best practices when writing them
    - So, if something about a build script can change stick it in a properties (i.e. variable)

Targets

- A target is an “intermediate” step in the build process
  - In make, they represented files and contained the actions required to produce the associated file
    - e.g. “to create foo.o compile foo.c”
  - In ant, targets typically represent stages
    - init, compile, test, package, deploy, clean
- Targets have names and dependencies and group tasks
  - <target name= “compile” depends= “init”>

Tasks

- Tasks are actions that need to be performed to complete the goal of its associated target
- If an “init” target needs to create a bunch of directories and copy a bunch of files into them then its tasks might look like
  - <mkdir dir= “${build.dir}” />
  - <copy todir="${build.dir}/metis/gui/help">
  -     <fileset dir="gui/help"/>
  - </copy>
  - …
- ${var} is a prop. reference; You can create your own tasks

Good Build Scripts will…

- reference required libraries
- compile your project
- generate documentation
- run your application
- check out code, run tests, send e-mail, etc.
  - (all via supplied tasks or custom tasks)

Examples

- InfiniTe build.xml file
- metis build.xml file
- Build management in XCode
  - Visual Studio, Eclipse, NetBeans have similar capabilities

Why do all this?

- We’ve touched on the fact that build management reduces accidental difficulties but the primary reason is that
- build management lets you focus on writing code
  - it automates repetitive tasks so you can focus on completing user stories and making progress
- In addition, it allows you to tackle integration and deployment issues early in the life cycle
  - and ensures that this process stays stable throughout the project; if someone “breaks the build” you find out quickly!

Wrapping Up

- Building a project should be repeatable and automated
  - All but the smallest projects have a nontrivial build process
  - You want to capture and automate the knowledge of how to build your system, ideally in a single command
- Build scripts are code (executable specifications) that need to be managed just like other pieces of code
- Use a build tool to script building, packaging, testing, and deploying your system
  - Most IDEs have an integrated build system

Coming Up

- Lecture 18: Shared Objects and Mutual Exclusion
  - Material drawn from the optional textbook