Principles of Software Construction: Objects, Design, and Concurrency

Configuration Management

Christian Kästner **Bogdan Vasilescu**

(Adapted from Christopher Parnin/NCSU & Prem Devanbu/UC Davis & Kenneth Anderson/CU Boulder)

Intro to Java

Git, CI

UML

GUIs

Static Analysis

Performance

GUIs

Config management

More Git

Design

Part 1: Design at a Class Level	Part 2: Designing (Sub)systems	Part 3: Designing Concurrent Systems
Design for Change: Information Hiding, Contracts, Design Patterns, Unit Testing	Understanding the Problem	Concurrency Primitives, Synchronization
Design for Reuse: Inheritance, Delegation, Immutability, LSP, Design Patterns	Responsibility Assignment, Design Patterns, GUI vs Core, Design Case Studies	Designing Abstractions for Concurrency
	Testing Subsystems	Distributed Systems in a Nutshell
	Design for Reuse at Scale: Frameworks and APIs	

Scenario

A customer wants a bug fix to software version 8.2.1, which was released 2 years ago.

How to make sure we can fix, build, and release?

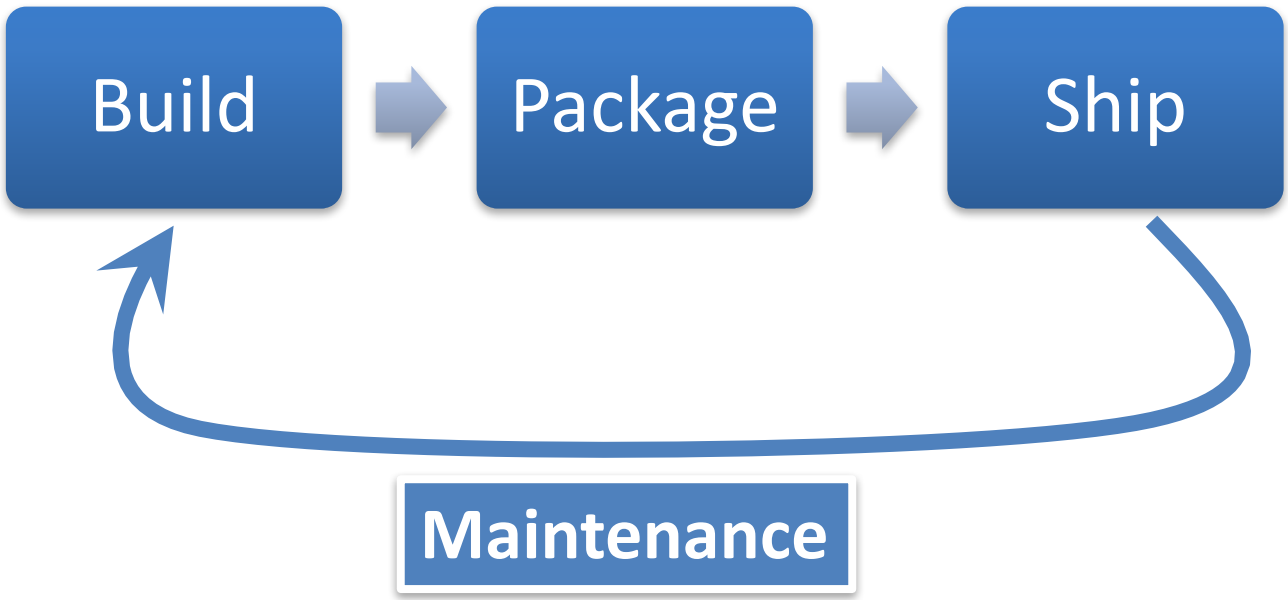
Configuration Management (CM)

Pressman:

"is a set of tracking and control activities that are initiated when a software engineering projects begins and terminates when software is taken out of operation"

Configuration management originates from the 50s, when spacecraft failures resulted from undocumented changes.

Most Basic Life Cycle.



Why is this hard?

- Software gets big (why? So what?)
- Software must evolve. (why? So what?)
- Software is built by teams (why? so what?)
- Software is built by many organizations (why? So what?)

Take Aways

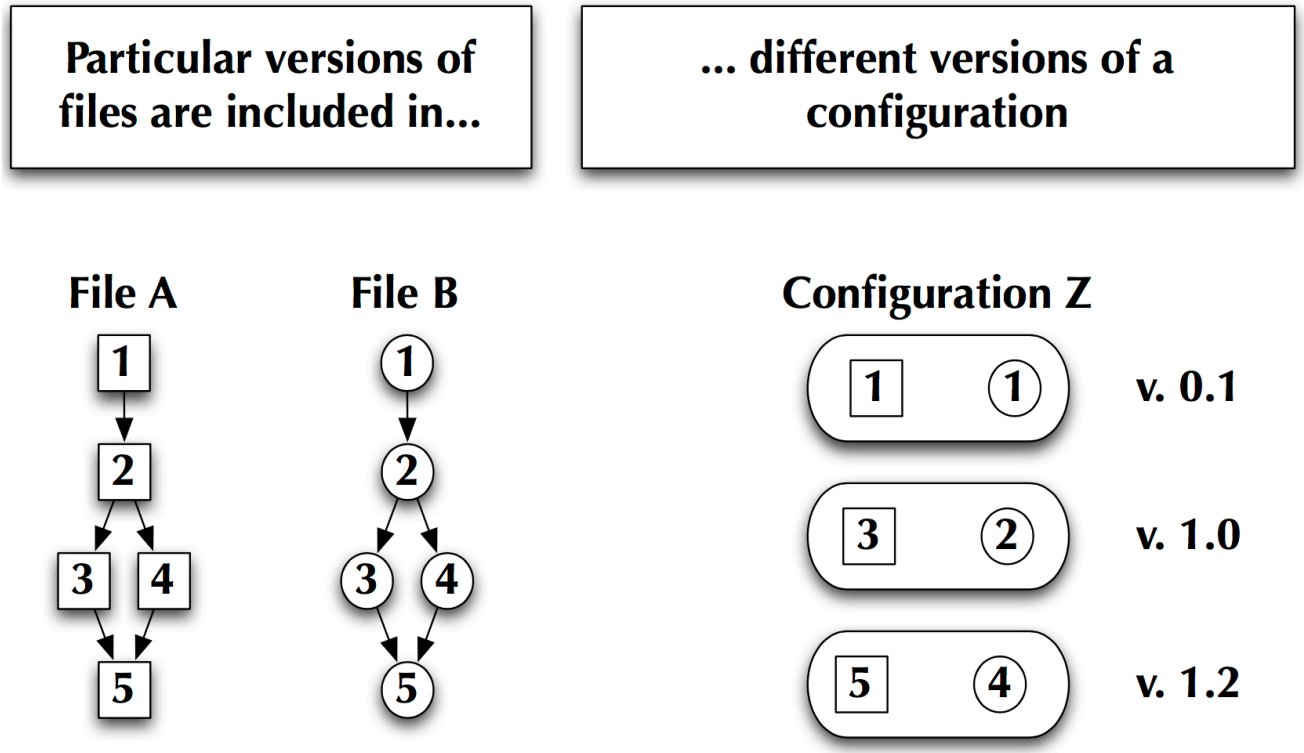
- Software isn’t just Java.
- One system can have thousands of files. All sorts.
- Can take a long time to build.
- Stuff get scattered over dozens of directories.
- Stuff exists in various stages of completion.
- Stuff gets built by several different organizations.
- …

Without a net

- Doing software development without config. management is “working without a net”
- Configuration management refers to both a process and a technology
  - The process encourages developers to work in such a way that changes to code are tracked
    - changes become “first class objects” that can be named, tracked, discussed and manipulated
  - The technology is any system that provides features to enable this process

Config. management vs version control

- "version control" is "versioning" applied to a single file while "configuration management" is "versioning" applied to collections of files



The Classical World

Traditional SCM Process

- Identify all items related to software.
- Manage changes to those items.
- Enable variations of items and changes.
- Maintain quality of versions and releases.
- Provide traceability between changes and requirements.

Traditional Definitions

A baseline is a snapshot of a set of configuration management items.

Any change to a baseline requires a change request. Change requests are approved by a change control authority.

Engineering change order is an approved change request with specified constraints and review criteria.

Changing Configuration Management

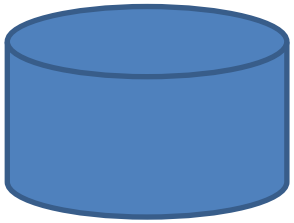
In traditional configuration management, the process is not fully triggered until deployment.

In modern configuration management, lightweight CM is integrated throughout the software process.

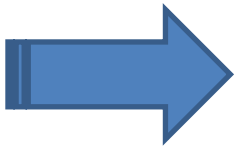

THE Jetsons

The Modern World

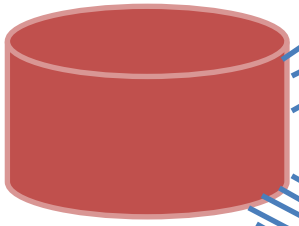
Cloud
Deployment
Distributed
Data
Virtualization
Load Balancing
Security



Complex Source
Languages,
Directories,
Dependencies
Source Files
Data
Versioning
Branching



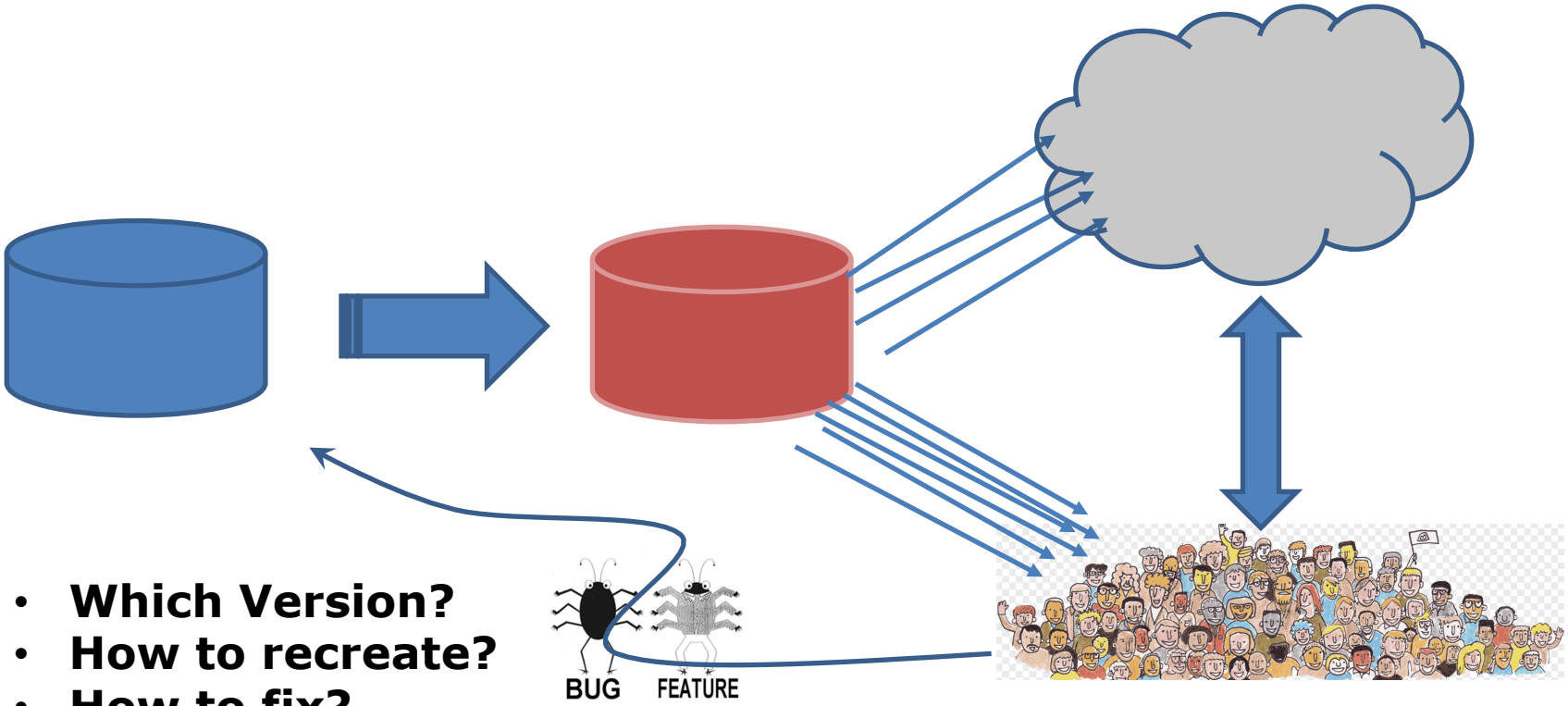
Many Tools
Compilers,
Linkers,
Code gens,
Translators
Traceability
Scalability
Configuring



Complex
Systems
Executables
Libraries
Dependencies
Config Files
Data
Consistency
Flexibility

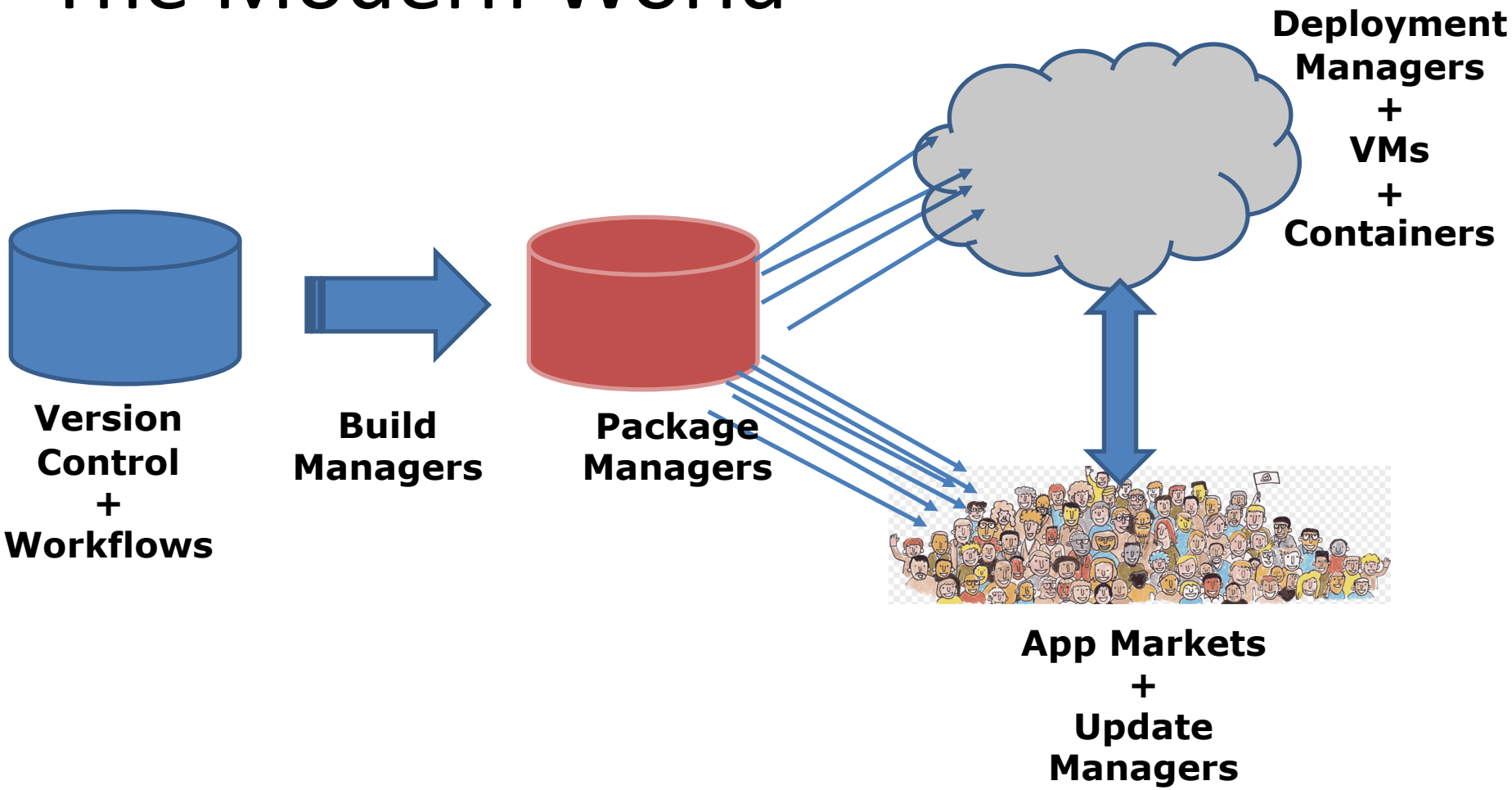
Diverse User Base
Many Platforms
Product Lines
Shared Libraries
Security
Localization

The Modern World



- **Which Version?**
- **How to recreate?**
- **How to fix?**
- **Where to apply the fix?**
- **How/when to Redistribute?**

The Modern World



Other concerns

- **Training:** onboarding new devs, tool and technology learning requirements
- **Audits**: Discovery request on changes made to system (e.g. no tracking in breathalyzer lawsuit)
- **Product lines** of software (Home, Business, Professional); different customer types.
- **Markets:** Asia, Europe, America (Language + feature variance)
- **Platforms:** Windows, Mac OS, Android, iOS

Components of Modern CM

Version Control: Branches/Forks/Workflow
Task and Build managers
Build machines, virtual environments (dev stacks)
Package managers
Containers, VMs, in the Cloud
Deployment –Infrastructure as Code.
Data migration

Other issues: orchestration, inventory, compliance

Package Managers

Package Managers

Avoid problems related to platform configuration.

Binary: apt-get, choco (windows), brew (mac)

Source: bower (web), nuget (C#), pip (python), npm (node.js), maven (Java)*, RubyGems

* http://stackoverflow.com/questions/2710266/is-there-a-package-manager-for-java-like-easy-install-for-python

Build Managers

Task and Build Managers

Analyze dependencies, and efficiently build (only) what needs to be built or rebuilt.

Tools: make, ivy, ant, maven, gradle, …

Build management

- 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?
  - Who is authorized to build a system?
    - Small projects: individual programmers
    - Large projects: build/config managers
  - When are system builds performed?
    - e.g. at night

Build Teams

- Build teams take over the role of building software.
- Can serve as gatekeepers on code integration.
- Create build configuration and sometimes verification scripts
- Putting out fires, helping with build failures

What a build system must handle.

- Difficulties of building complex system: scale, intricacy, overheads, diversity.
  - …and ***dependencies.***
- What kind of dependencies exist?
  - In C: Applications → Libraries, Libraries → Objects, Objects → Source Files, Source Files → Header Files
  - In Java: Applications → Jars, Jars → Class Files, Class Files → Source Files

- How do you figure out when to rebuild? What to install (for package managers)

Dealing with dependencies.

- Example Scenario: C program, 4 source files, 8 header files.
- Approaches to dependencies:
  - Explicitly tell the build system what the dependencies are.
    - Problems? In C? In Java?
  - Automatically figure out the dependencies? How?
- How to deal with versions in dependencies?
- Transitivity? Consistency?

Make

Make legacy

- 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
- Worked for Google as Vice President of Engineering (East Coast); also president of ACM
- When you click “build” in your IDE and it builds your project, you have *make* to thank

Specification styles

- Make provides very powerful capabilities via three types of specification styles
  - Declarative
    - Described according to desired properties
  - Imperative
    - Described according to desired actions
  - Relational
    - Described according to desired relationships
- These styles are combined into one specification: “the make file”

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.

Makefile structure

```
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?

Example Makefile

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

```
target … : prerequisites …
        recipe
        …
        …
```

- Rules
- Components or Rules
- Main Steps

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
```

Shell commands go here

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
```

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

Examples

File	Time Stamp
main.cpp	1
input.cpp	2
defs.h	3
main.o	4

- What happens when you invoke make on this makefile?

```
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
```

Examples

File	Time Stamp
main.cpp	1
input.cpp	2
defs.h	3
main.o	4

- What happens when you invoke make on this makefile?

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

main.o is up to date

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

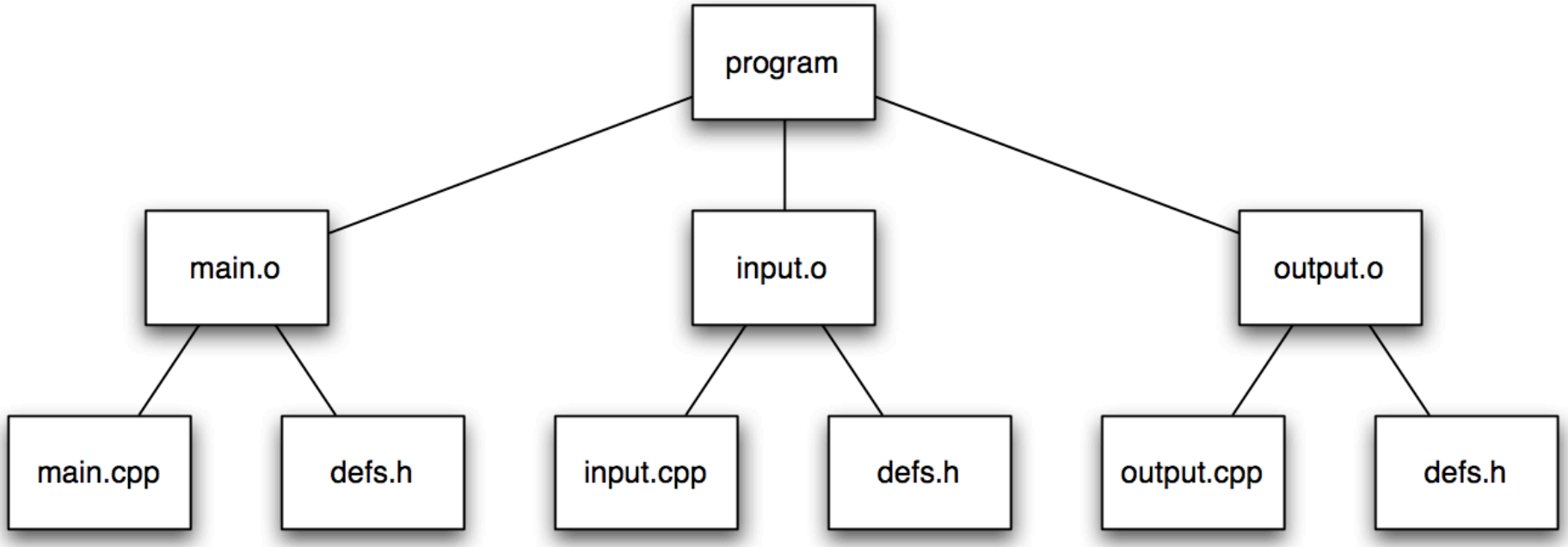
g++ -c input.cpp

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

make: Fatal error: Don't know how to make "output.cpp"

Make dependency graph

- A makefile can be modeled as a dependency graph. The make algorithm performs a traversal over the graph. Each node is checked after all of its children, and the actions are run if any child has a timestamp greater than its parent



Using Macros (Variables)

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit $(objects)
```

Macro substitution

- Make variables perform strict textual replacement so the following two rules are equivalent
- (Do not do this in practice!):

```
program: output.o
        g++ output.o -o program
FOO = o
pr$(FOO)gram: $(FOO)utput.$(FOO)
        g++ $(FOO)utput.$(FOO) -$(FOO) pr$(FOO)gram
```

Definition and Use of Make Macros

- A shell script is executed from top to bottom. As such, a shell variable cannot be used before it is defined.
- Makefiles, on the other hand, are not executed top to bottom. Execution follows dependencies which can be anywhere in the file

```
%echo $var
%set var = hello
```

```
all:
	echo $VAR
VAR = hello
```

Using Pre-Built rules (.o comes from .c)

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
        rm edit $(objects)
```

Rule Patterns:

```
%.o: %.c
        gcc –c -o $@ $<
```

Grouping by Pre-requisite

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
```

How do solve the problem of Dependencies?

- How do we know what the dependencies are (in C? in Java?)
- One approach: use the C Preprocessor (why does this work?)

  g++ -MM $(CFLAGS) $(SOURCES) > all.d

- Another approach,

  makedepend -- $(CFLAGS) -- $(FILES) –f- > all.d

- *How to get the dependencies into the makefile?*

Using Makedepend

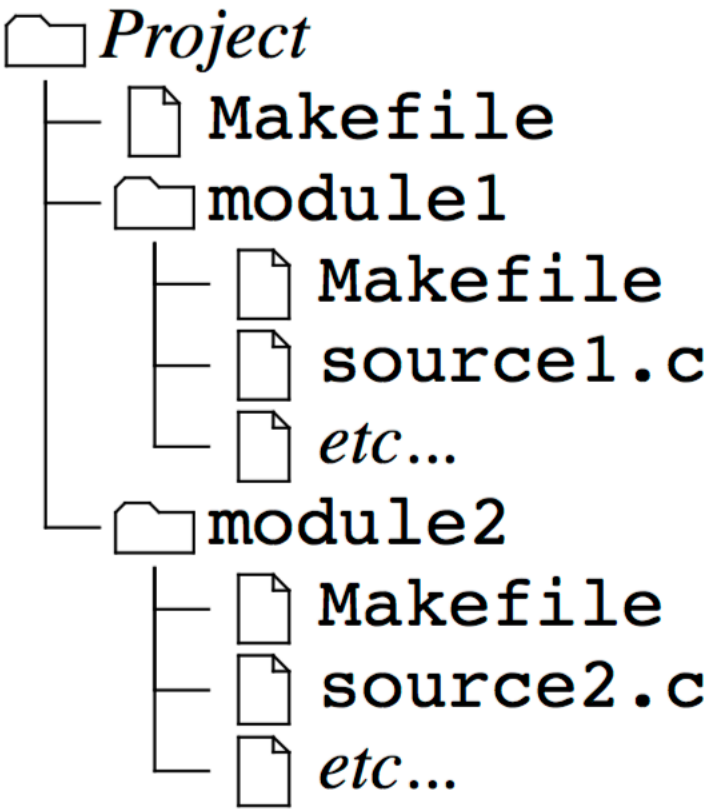
```
BINARY  = project.exe
CC      = gcc
CFLAGS  =
FILES   = $(shell find src/ -name "*.c")
HEADERS = $(shell find src/ -name "*.h")
OBJS    = $(FILES:.c=.o)

all: $(BINARY)

-include Makefile.deps

$(BINARY): Makefile.deps $(OBJS)
        $(CC) $(CFLAGS) $(OBJS) -o $(BINARY)

Makefile.deps: $(FILES) $(HEADERS)
        makedepend -- $(CFLAGS) -- $(FILES) -f- > Makefile.deps
```

Handling multiple directories.

```
                    $projectroot
                         |
          +---------------+----------------+
          |               |                |
        part1/          part2/           part3/
          |               |                |
   +------+-----+     +---+----+       +---+-----+
   |      |     |     |        |       |         |
 data/   src/  inc/  src/     inc/    src/      inc/
```

- How would you do this?

Handling multiple directories.



- Traditionally:
  - have a Makefile in each of the subdirectories (module1, module2, etc.), to be able to build them independently
  - have a Makefile in the root directory of the project which builds everything.

Handling multiple directories.

- Simple Recursive Strategy—put makefile in each directory. And then…

```
subsystem1:
            cd subsystem1 && make
```

- Problems with above?
- Many: See http://aegis.sourceforge.net/auug97.pdf
- Alternative very complex

Good things about make

- Available on pretty much every darn platform.
- Very fast.
- Fully featured programming language (but weird)
- First mover advantage

Bad things about make

- Weird syntax (indent is tab, NOT space)
- Has only global variables.
- Where shell can be used, and where make commands? Weird.
- No standards for anything. E.g.,: recursion, dependency analysis, file lists. (So what?)
- No “reuse” or inheritance of makefiles.
- Not portable across OS, even across Unix flavors.
- Debugging? Yeah, good luck with that.
- Can’t guarantee consistency/reproducibility.

Ant

Main Concepts:

- Project
- Property
- Target
- Task
- Dependency

Ant vs make

- Make file in XML.
- Replace weird indentations with weird angle brackets.
- Replace “variables” with <property />
- Replace “targets” with <target name=“jar” />
- Replace “rules” with <target name =“jar”,
  depends=“init, classes” />”
- Replace “recipes in shell” with tasks.
  “<javac />”,
  “<mkdir />”,
  “<jar />”

Ant's model

- Everything is a Task (sort of)
  - A task has an associated XML element in Ant build files and an associated Java class that implements the task.
  - The XML element can have various attributes and sub-elements, converted into parameters and passed to the Java class.
  - Build file called build.xml by convention
    - First task executed by invoking its associated Java class and passing it its input parameters (if any).

- What's the difference between tasks as shell commands vs tasks as Java?

Ant Project Format

- build.xml

- Project Name
- Property Values
- Paths
- Tasks
- Targets

Construction of Ant Build Files

- The default name for a Ant build file is build.xml
- The xml root element must be the ‘project’ element
  - The ‘default’ attribute of the project element is required and specifies the default target to use
- Targets contain zero or more AntTasks
  - The ‘name’ attribute is required
- AntTasks are the smallest units of the build process

Ant Build File Example

```
<project default="hello">
   <target name="hello">
      <echo message="Hello, World"/>
   </target>
</project>
```

Execution of build file:

```
% ant
Buildfile: build.xml
hello: [echo] Hello, World
BUILD SUCCESSFUL
Total time: 2 seconds
```

Ant Build File Example

```
<project default="hello">
   <target name="hello">
      <echo message="Hello, World"/>
   </target>
  <target name=”bye">
      <echo message=”goodbye, World"/>
   </target>
</project>
```

```
franky:xx devanbu$ ant bye
Buildfile: /private/tmp/xx/build.xml

bye:
     [echo] Bye, World

BUILD SUCCESSFUL
Total time: 0 seconds
franky:xx devanbu$ ant hello
Buildfile: /private/tmp/xx/build.xml

hello:
     [echo] Hello, World

BUILD SUCCESSFUL
Total time: 0 seconds
franky:xx devanbu$
```

Ant Properties

- <property name="lib.dir" value="lib"/>
- From command line
- In build.xml
- From external XML
- From external property files
- From environment

Ant Path, Ant Target/Task

```
<path id="classpath">
    <fileset dir="${lib.dir}"
            includes="**/*.jar"/>
</path>

<target name="compile">
    <mkdir dir="${classes.dir}"/>
    <javac srcdir="${src.dir}"
           destdir="${classes.dir}"
           classpathref="classpath"/>
</target>
```

Ant Target

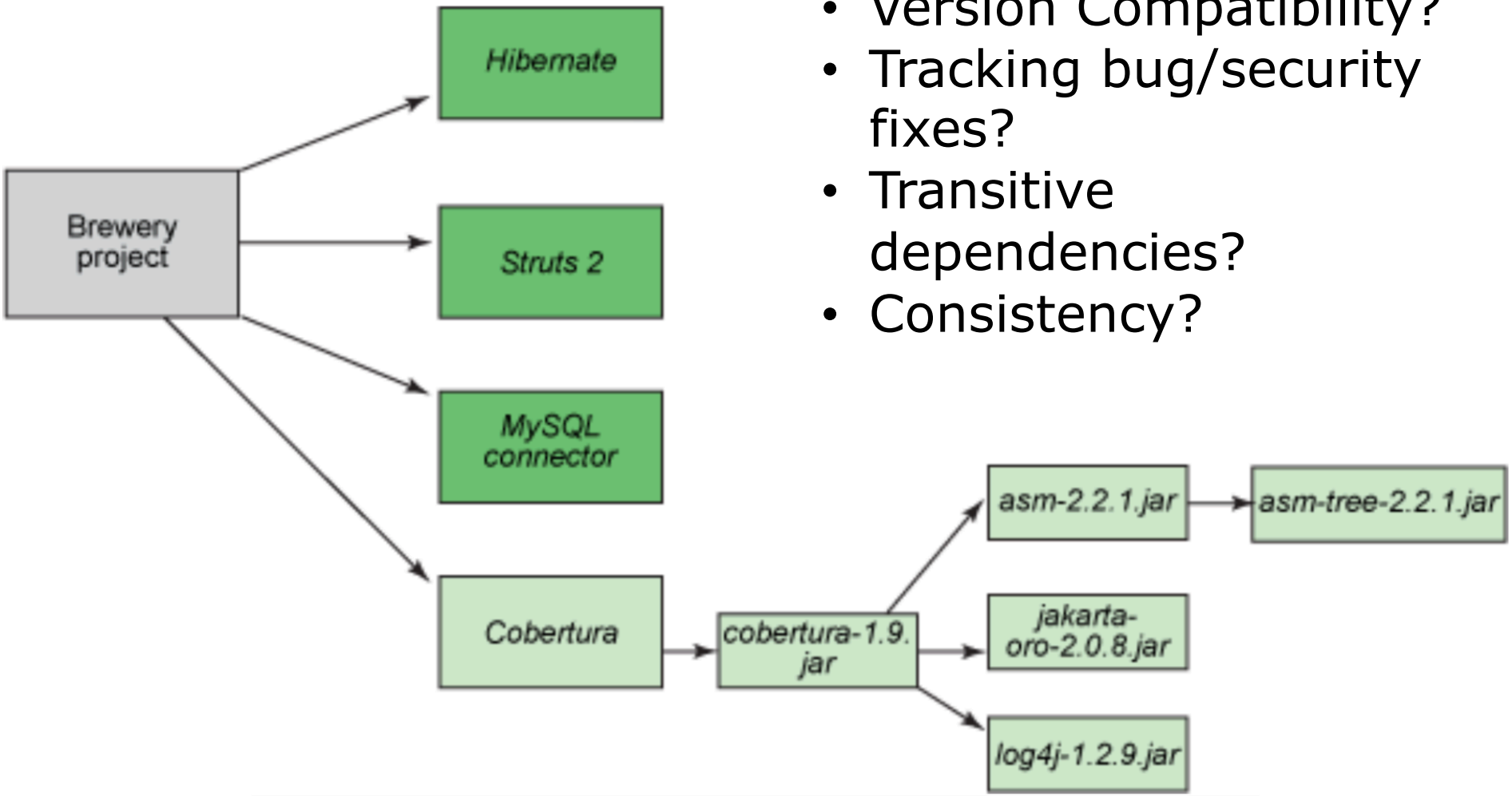
- Name
- Description
- Dependencies
- Conditionals
- <antcall> task

Ant Tasks

- Core Tasks
- Optional Tasks
- Custom Tasks

Ant AntCall Apply Basename BuildNumber Checksum Chmod Concat Condition Copy Cvs Delete DependSet Dirname Ear Echo EchoXML Exec Fail Filter FixCRLF Get Import Input Jar Java Javac Javadoc Length LoadFile LoadProperties LoadResource MakeURL Mail MacroDef Manifest ManifestClassPath Mkdir Move Nice Parallel Patch PathConvert PreSetDef Property Record Replace ResourceCount Retry Rmic Sequential SignJar Sleep Sql Subant Sync Tar Taskdef Tempfile Touch Truncate TStamp Typedef Unjar Untar Unwar Unzip Uptodate Waitfor WhichResource War XmlProperty XSLT Zip

Dependencies



- Version Compatibility?
- Tracking bug/security fixes?
- Transitive dependencies?
- Consistency?

Imperfect techniques to manage dependencies

- Placing all dependent projects (JAR files) in a directory that's checked into the project's version-control repository.
- Allocating dependent JARs to a common file server
- Copying JAR files manually to a specific location on each developer's workstation.
- Performing an HTTP Get to download files to a developer's workstation, either manually or as part of the automated build.

Ivy

Defining dependencies in ivy.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="./config/ivy/ivy-
doc.xsl"?>
<ivy-module version="1.0">
  <info organisation="com" module="integratebutton" />
  <dependencies>
    <dependency name="hsqldb" rev="1.8.0.7" />
    <dependency name="pmd" rev="2.0" />
    <dependency name="cobertura" rev="1.9"/>
    <dependency name="checkstyle" rev="4.1" />
    <dependency name="junitperf" rev="1.9.1" />
    <dependency name="junit" rev="3.8.1" />
  </dependencies>
</ivy-module>
```

- Note: no indication of file locations or URLs
- Convention: `dependency name="cobertura" rev="1.9"` translates to cobertura-1.9.jar

Specifying dependencies in Ivy

ivy.xml

```
<ivy-module version="2.0">
    <info organisation="org.apache" module="WebProject" />

    <dependencies>
        <dependency org="org.slf4j" name="slf4j-api" rev="1.7.6" conf="compile->default"/>
        <dependency org="jstl" name="jstl" rev="1.2" conf="compile->default"/>
        <dependency org="ch.qos.logback" name="logback-classic" rev="1.1.2" conf="compile->default"/>
        <dependency org="org.springframework" name="spring-core" rev="4.1.3.RELEASE" conf="compile->default"/>
        <dependency org="org.springframework" name="spring-beans" rev="4.1.3.RELEASE" conf="compile->default"/>
        <dependency org="org.springframework" name="spring-context" rev="4.1.3.RELEASE" conf="compile->default"/>
        <dependency org="org.springframework" name="spring-web" rev="4.1.3.RELEASE" conf="compile->default"/>
        <dependency org="org.springframework" name="spring-webmvc" rev="4.1.3.RELEASE" conf="compile->default"/>
    </dependencies>
</ivy-module>
```

- Note: no indication of file locations or URLs
- Convention: `dependency name="cobertura" rev="1.9"` translates to cobertura-1.9.jar

Ivy settings file

```
<ivysettings>
  <settings defaultResolver="chained"/>
  <resolvers>
    <chain name="chained" returnFirst="true">
      <filesystem name="libraries">
        <artifact pattern="${ivy.conf.dir}/repository/[artifact]-[revision].[type]" />
      </filesystem>
      <url name="integratebutton">
        <artifact pattern="http://www.integratebutton.com/repo/[organisation]/[module]/
           [revision]/[artifact]-[revision].[ext]" />
      </url>
      <ibiblio name="ibiblio" />
      <url name="ibiblio-mirror">
        <artifact pattern="http://mirrors.ibiblio.org/pub/mirrors/maven2/[organisation]/
           [module]/[branch]/[revision]/[branch]-[revision].[ext]" />
      </url>
    </chain>
  </resolvers>
</ivysettings>
```

Depending on dependencies

```
<?xml version="1.0" encoding="UTF-8"?>
<ivy-module version="2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://ant.apache.org/ivy/schemas/ivy.xsd">
  <info organisation="cobertura" module="cobertura" revision="1.9"/>
  <configurations>
    <conf name="master"/>
  </configurations>

  <publications>
    <artifact name="cobertura" type="jar" conf="master" />
  </publications>

  <dependencies>
    <dependency org="objectweb" name="asm" rev="2.2.1" conf="master"/>
    <dependency org="jakarta" name="oro" rev="2.0.8" conf="master"/>
    <dependency org="apache" name="log4j" rev="1.2.9" conf="master"/>
  </dependencies>
</ivy-module>
```

Use ivy.xml within an ANT build.xml

```
<project xmlns:ivy="antlib:org.apache.ivy.ant" name="HelloProject" default="main" basedir=".">
    <description>
        Create a Spring MVC (WAR) with Ant build script
    </description>

    <!-- Project Structure -->
    <property name="jdk.version" value="1.7" />
    <property name="projectName" value="WebProject" />
    <property name="src.dir" location="src" />
    <property name="resources.dir" location="resources" />
    <property name="web.dir" value="war" />
    <property name="web.classes.dir" location="${web.dir}/WEB-INF/classes" />
    <property name="target.dir" location="target" />
    <property name="target.temp.dir" location="target/temp" />
    <property name="lib.dir" value="lib" />

    <!-- ivy start -->
    <target name="resolve" description="retrieve dependencies with ivy">
        <echo message="Getting dependencies..." />
        <ivy:retrieve />

        <ivy:cachepath pathid="compile.path" conf="compile" />
        <ivy:cachepath pathid="runtime.path" conf="runtime" />
        <ivy:cachepath pathid="test.path" conf="test" />

    </target>

    <!-- install ivy if you don't have ivyide-->
    <target name="ivy" description="Install ivy">
        <mkdir dir="${user.home}/.ant/lib" />
        <get dest="${user.home}/.ant/lib/ivy.jar"
        src="http://search.maven.org/remotecontent?filepath=org/apache/ivy/ivy/2.4.0-rc1/ivy-2.4.0-rc1.jar" />
    </target>
    <!-- ivy end -->

    <!-- Compile Java source from ${src.dir} and output it to ${web.classes.dir} -->
    <target name="compile" depends="init, resolve" description="compile source code">
        <mkdir dir="${web.classes.dir}" />
        <javac destdir="${web.classes.dir}" source="${jdk.version}" target="${jdk.version}"
            debug="true" includeantruntime="false" classpathref="compile.path">
```

Maven

Main Ideas of Maven

- “Convention over Configuration”
- DESCRIBE, don’t IMPLEMENT.
- Reuse build logic & standards whenever possible (mostly done as “Maven plugins”)
- Organize dependencies clearly, logically, aesthetically

Main Benefits of Maven

- Reuse across multiple projects on the same platform.
- Smaller, more standardized, reusable, build procedures.
- Spend less time on Build, more time Coding Apps

A simple Java app

- `mvn archetype:generate -DgroupId=edu.cmu.cs -DartifactId=hello -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false`
  - What's in the directory structure?

- `Mvn compile`

- `Run: java -cp target/classes edu.cmu.cs.App`

- `mvn clean`

- `Mvn test` (see test results)
- `Mvn package`
- `Java –cp target/*jar edu.cmu.cs.App`

Gradle

Task-Based Managers: Gradle

- Combines the best of Ant and Maven
- From Ant keep:
  - Portability: Build commands described platform-independently
  - Flexibility: Describe almost any sequence of processing steps
- … but drop:
  - XML as build language, inability to express simple control flow
- From Maven keep:
  - Dependency management
  - Standard directory layouts & build conventions for common project types
- … but drop:
  - XML, inflexibility, inability to express simple control flow

Summary

What every build system must do:

- Manage dependencies within-project.
- Manage dependencies for outside libraries.
- Maintain consistency and versioning.
- Know tasks that ”complete” the dependencies.
- Deal with complex directory structures and many types of files.
- Be as simple as possible, but no simpler.

- *How do each of the build systems we discussed do at all this?*

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