Maven, or Make for Non-Cooks

David Collier-Brown
The Original Problem

- cc *.c takes too long
- Why not compile just what's changed?

```bash
theOtherThing: this.o that.o; ln -o
theOtherThing *.o
this.o: this.c; cc -c this.c
that.o: that.c; cc -c that.c
```

Third line is the Recipe

- You can add lots of lines of recipe
  
  that.o: that.c;

  lint that.c && \

  cc -c that.c

- And you can add targets for common recipes
  
  clean:; rm *.o
But It's Still Hard

• You have to write all those dependency lines
• The linker knows some of the information
  > theOtherThing: this.o that.o
• The c compiler knows all the .h files
  > this.o: this.c stdio.h
• So have them write the non-recipe lines
The Next Problem: Repetition

• Two pairs of near-identical recipes in this simple example

• So we added macros
  > FOO=/usr/local/obscure
  > @echo “Install in ${FOO}”

• And rules
  > %.o: %.c; cc -o $<

• And more rules
  > Etc, etc, ite ad nauseam
Any Fool can write Makefiles

• And many do.
  > No standardization
  > Lots of make dialects

• Conventional targets
  > all, clean, install, test

• Lots of standards, all different
XKCD Says...

(Situation: there are 14 competing standards.)

14?! Ridiculous! We need to develop one universal standard that covers everyone’s use cases. Yeah!

(Soon: Situation: there are 15 competing standards.)
So Start Again From Scratch

• QEF, with one common notation
• And other, less-well-known approaches
  > But mostly they reinvented square wheels
• Ant, try writing everything in XML
Apache Ant

• Ant was still annoying
  > No standard build targets
  > Every antfile contained another re-invented wheel
  > It was xml, but mostly imperative
  > And it was xml

• Couldn't we get simpler?
  > Or, if it was a wheel, rounder?
Apache Maven

• Grew out of dissatisfaction with Ant
  > Simplify and streamline a mongo antfile, from Apache Turbine

• Ant provided primitives like “mkdir” and “copy”

• Maven provided “compile” and “install”
  > Bootstrapped with ant, jelly xml

• M2 Upgraded to clean it up some more
  > Java, and XML as a declarative language
No Recipes

- Write plugins to do common operations
  - install jar
  - install war
  - Create project
A Standard Set of Targets

• By default, anyway:
  > Clean
  > Compile
  > Test
  > Install

• “mvn clean install”
  > Does just what you expect
Common infrastructure

• If you say plugin:download -D... gnurrs, it will
  > Download the “gurrs” extension
  > Install it in your environment
• Most common steps already written
• For example, install a project into Eclipse
Plugins: gee, looks like everything else

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-eclipse-plugin</artifactId>
  <version>2.9</version>
  <configuration> ... </configuration>
Easy things should be easy

• All sorts of common operations are already written
• The almost all work
• But when they don't...
Hard things should be at least possible

- Just *try* debugging install under eclipse on Linux
- You'd better know Maven, Eclipse and Linux
- Or google a lot
The good part

- Everything is a dependency
- It's really make, recursively self-applied
- The O'Reilly book teaches you to make and debug plugins
Using MVN

- Mvn phase
- Mvn specific:command
- They say “convention”
Three main variables

- Group id, `maven-plugins`
- artifact id, `maven-axis-plugin`
- Version, 0.7
  > or
- Version, 0.7-SNAPSHOT
  > Means latest version of 0.7
- Maps to paths, eg
  > `maven-plugins/plugins/maven-axis-plugin.0.7.jar`
Three main variables, ctd

```xml
<project ...> ...
  <groupId>com.skilledgaming</groupId>
  <artifactId>platform</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
```
For example

• Add an actual plugin
  > `Mvn plugin:download -DgroupId=maven-plugins -DartifactId=maven-axis-plugin -Dversion=0.7`
  > Will download a plugin used to in turn download NOAA data, used in a weather-map coding example (U.S. National Oceanic and Aeronautic Administration)
Repositories for all the bits

• If we had declared a dependency on axis, maven would download it itself

  <dependency>

    <groupId>axis</groupId>

    <artifactId>axis</artifactId>

    <version>1.2.1</version>

  ... 

• Covers annoying long lists of dependencies in build and test tools

• Huge time-saver
Repository declarations

- The installer created several

  <repository>
    <id>central</id>
    <url>
      http://artifactory.virgin/artifactory/repo
    </url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>

...
Multi-project builds

- Each project produces one artifact (deliverable)
  - The can have dependencies between them
  - They can depend on external binaries
- A collection of projects is a collection of dependencies, like make, but with most of the recipes taken out
Multi-project builds, ctd

- Can include continuous integration systems
- And revision control targets
- Ditto remote repositories, using snapshots
- Also used for building plugins
Multi-project builds, ctd

• Even this one has some substructure
  
  <dependency>
    <groupId>
      com.skilledgaming.platform
    </groupId>
    <artifactId>jskills</artifactId>
    <version>1.0</version>
  </dependency>
In Practice

• Build a web app
• Back end uses NOAA data
• Delivered as a jar or war
• All the components used to build and install are dependencies
Conclusions?

• Make with standards
• and compiled recipes
• Scales via recursing on dependencies

• Easy to use, hard to learn
  > A traditional tradeoff
• As the English would say, “Not half bad”