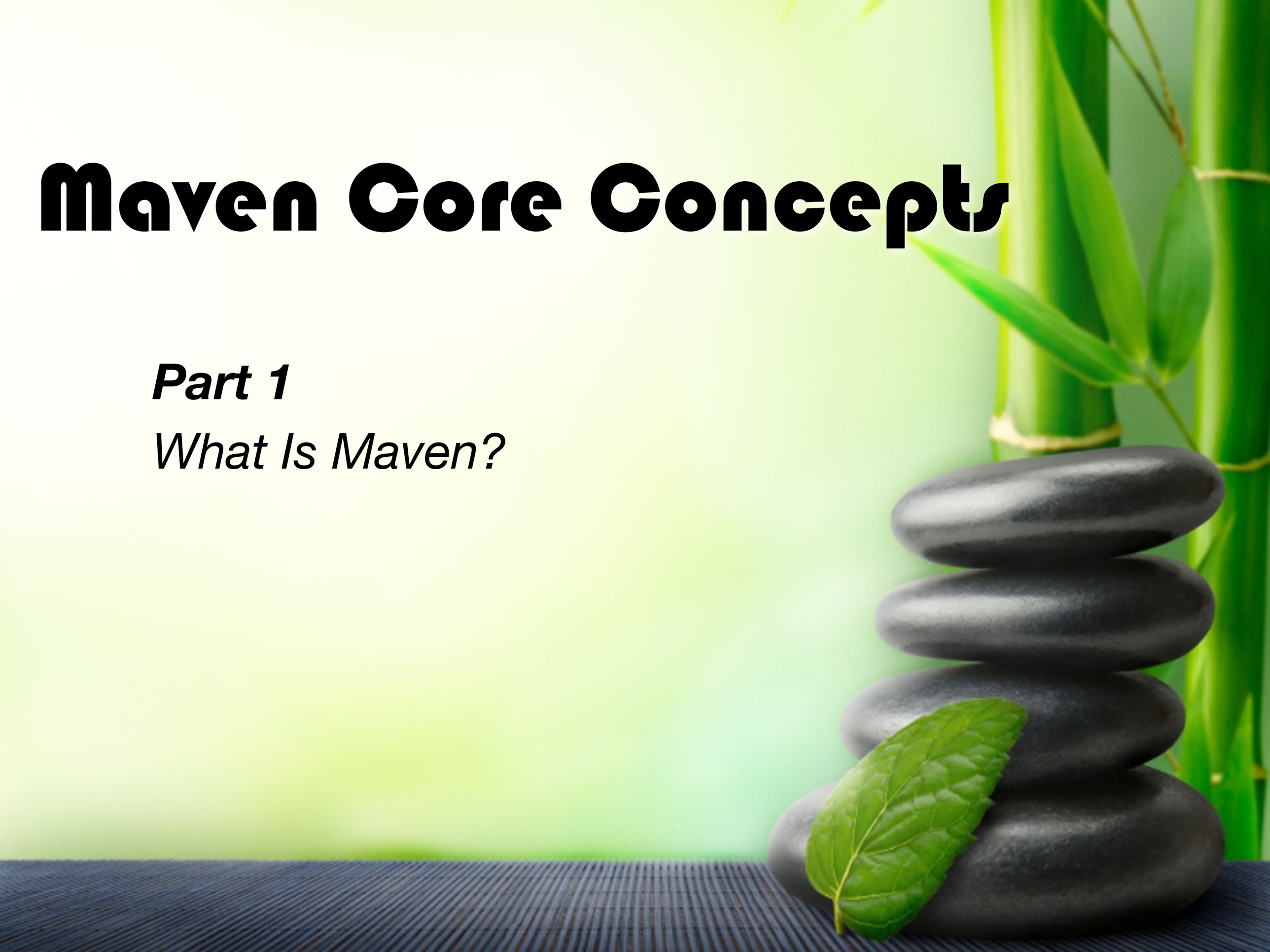
Maven Core Concepts

Part 1

What Is Maven?



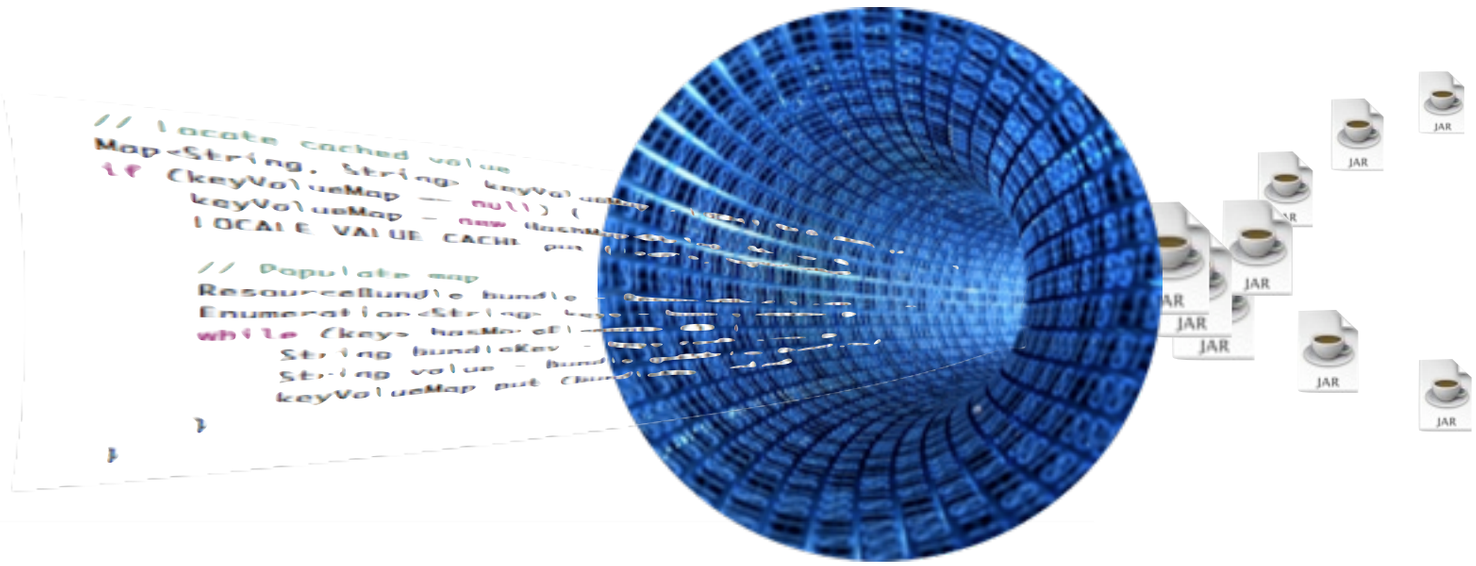
Maven Motivations

Software Project Management

- Most software projects have many things in common
  - Code compilation
  - Test execution
  - Packaging
  - Documentation
  - Distribution

Automation

- Automate these common actions



- Compiling
- Testing
- Packaging
- Building Documentation
- Releasing (Sharing)

Maven Conventions

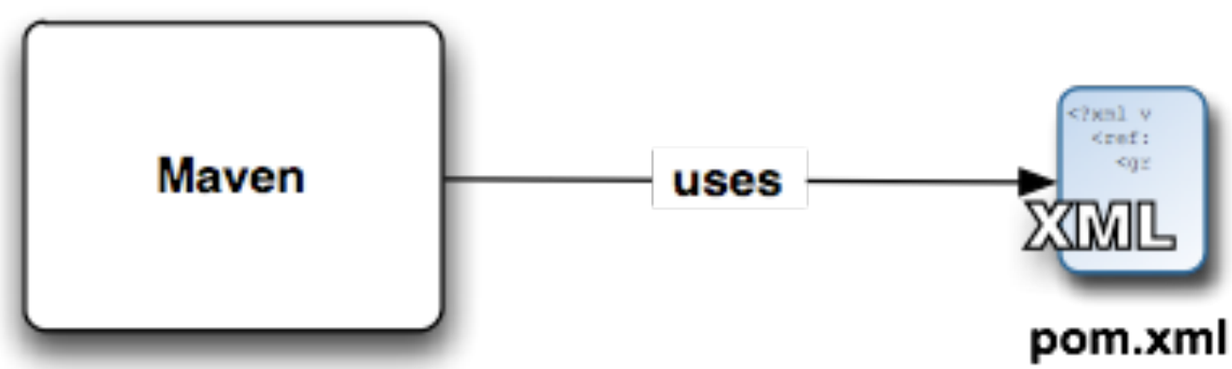
- File layout
- Vocabulary for running the build
- Always running tests
- Generating documentation
- Binary naming in a central location

Creating Your Build

- Should you create your own build from scratch?
- Build Systems present an Illusion of Simplicity
- Often the Last Priority in a Software Organization
- Spend time on Your Software, not on Your Build!
- Developers shouldn't be required to write logic to achieve such common tasks.

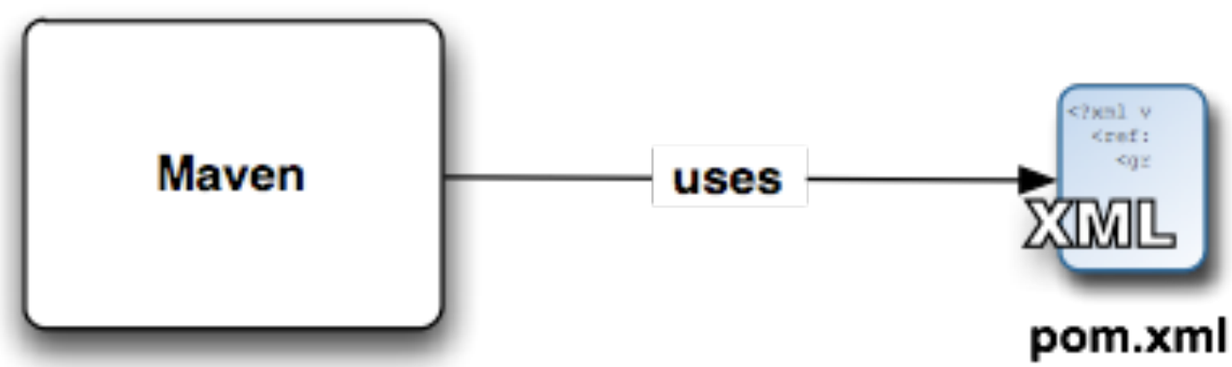
Welcome to Maven

- Maven manages the build process
  - Reuse standard build logic (compile, package,...)
  - Applies it's logic to a project, guided by project description (or "metadata")



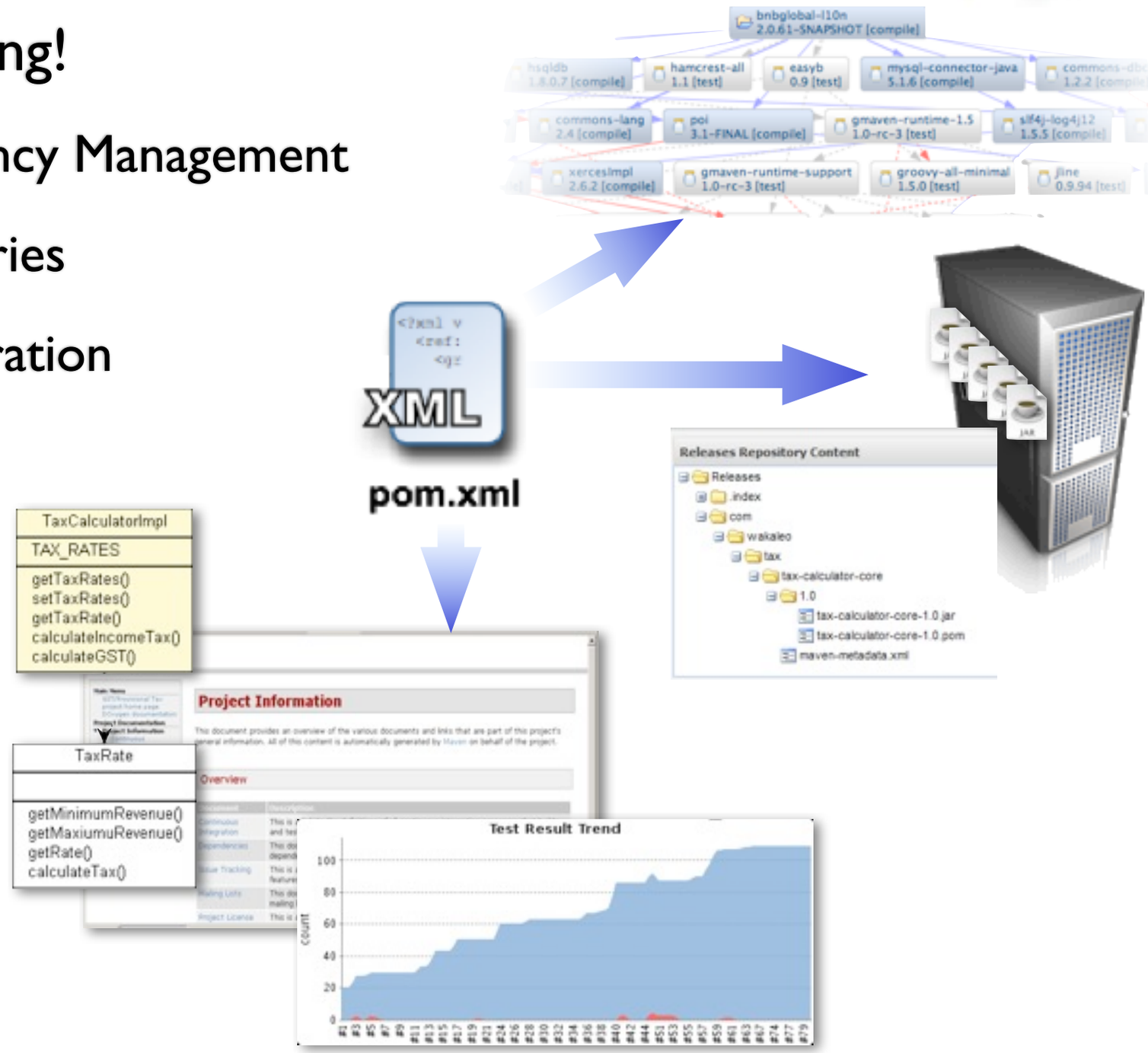
Welcome to Maven

- Maven uses a declarative approach
  - Describe your project, not just the steps required to build it
  - Your project description (or "object model") goes in a `pom.xml` file



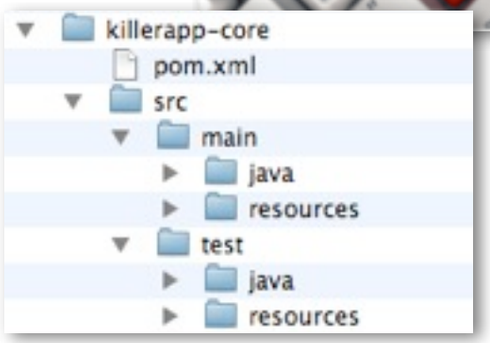
More Than Just a Build Tool

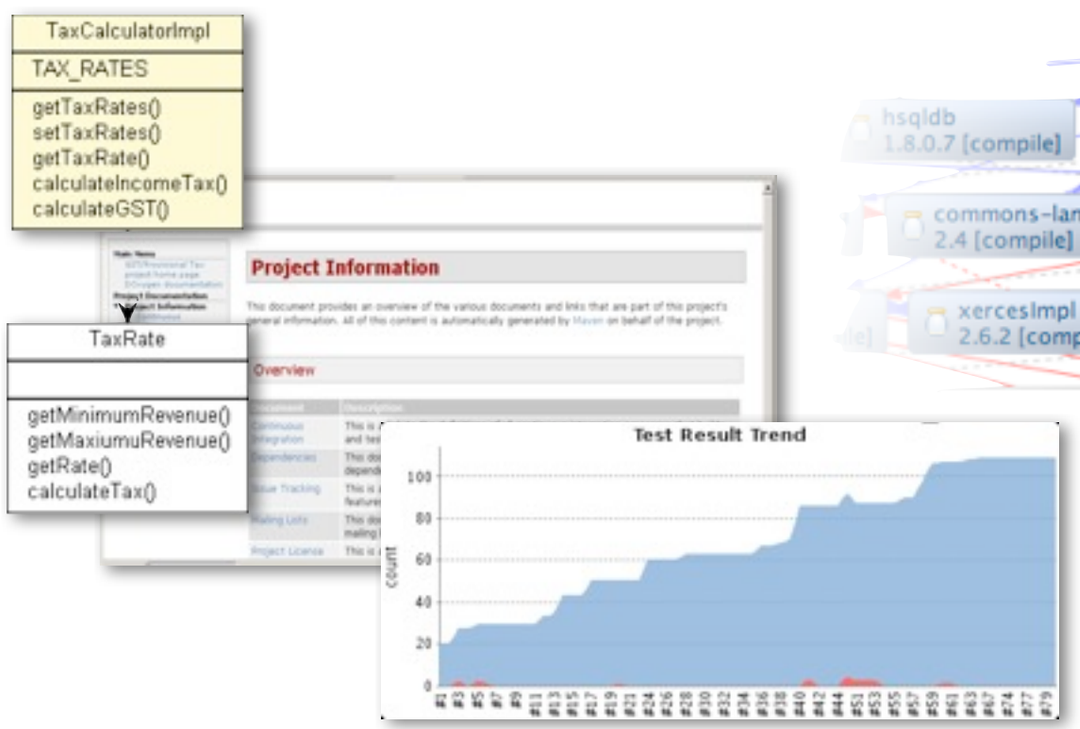
- Not just build scripting!
  - Declarative Dependency Management
  - Centralized Repositories
  - Documentation generation
  - Reporting
  - And more...

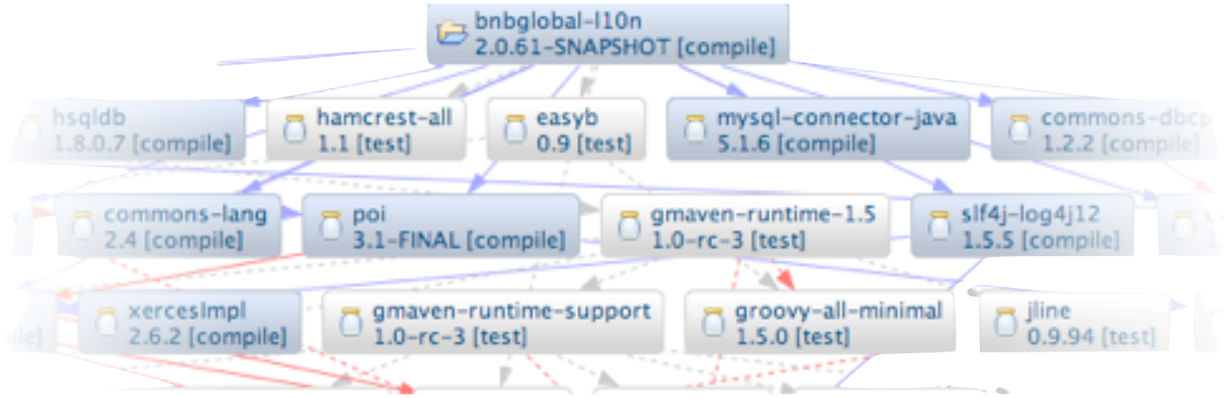


Key Features of Maven

- So how can Maven help me and my team?
  - A standardized build and deployment process
  - A standardized project directory structure
  - Improved dependency management
  - Easy to generate reasonable technical reports







Benefits of Maven

- Build standardization - “A Common Interface”
  - All basic functionality is provided no matter what Maven project you use
- Dependency management
  - No more manual management of dependencies and guessing versions
- Lifecycle management
  - Provides a build life cycle instead of making completely you build your own
- Project management best practices
  - Consistent directory structure provides easy understanding of artifacts

History of Maven

- 2001 - Jakarta Alexandria
  - Originally started to simplify the Jakarta Turbine Build
  - Previously used Ant and committed JARs to SCM
- 2002 to 2004 - Maven 1
  - Established Central Repository
  - Rapid Adoption by Open Source Project
- 2004 to present - Maven 2
  - Drastic Change from Maven 1 - Improved Model
  - Complete Redesign
- October 2010 - Maven 3
  - Incremental Improvements
  - Focus on Interoperability / Plugin

Resources

Maven Resources

- Sonatype: http://www.sonatype.com
  - Sonatype Blogs: http://blogs.sonatype.com
  - Maven Support Available from Sonatype

- Apache Maven Project
  - Website: http://maven.apache.org
  - Mailing Lists: http://maven.apache.org/mail-lists.html
    - Maven Users Mailing List
    - Maven Developers Mailing List

Maven Books

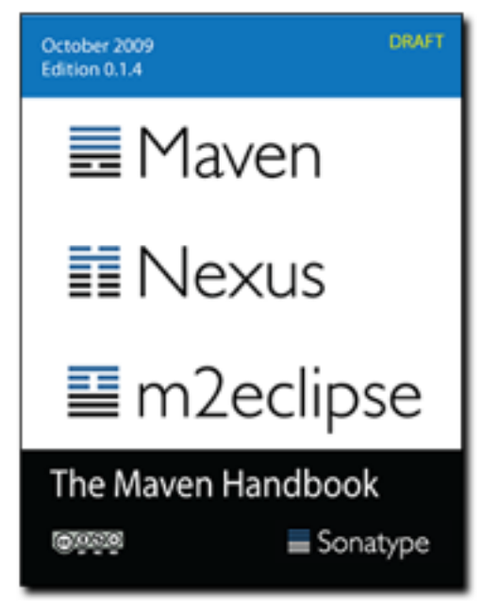
- Maven: By Example
- Maven: The Complete Reference

- http://books.sonatype.com

More Maven Books

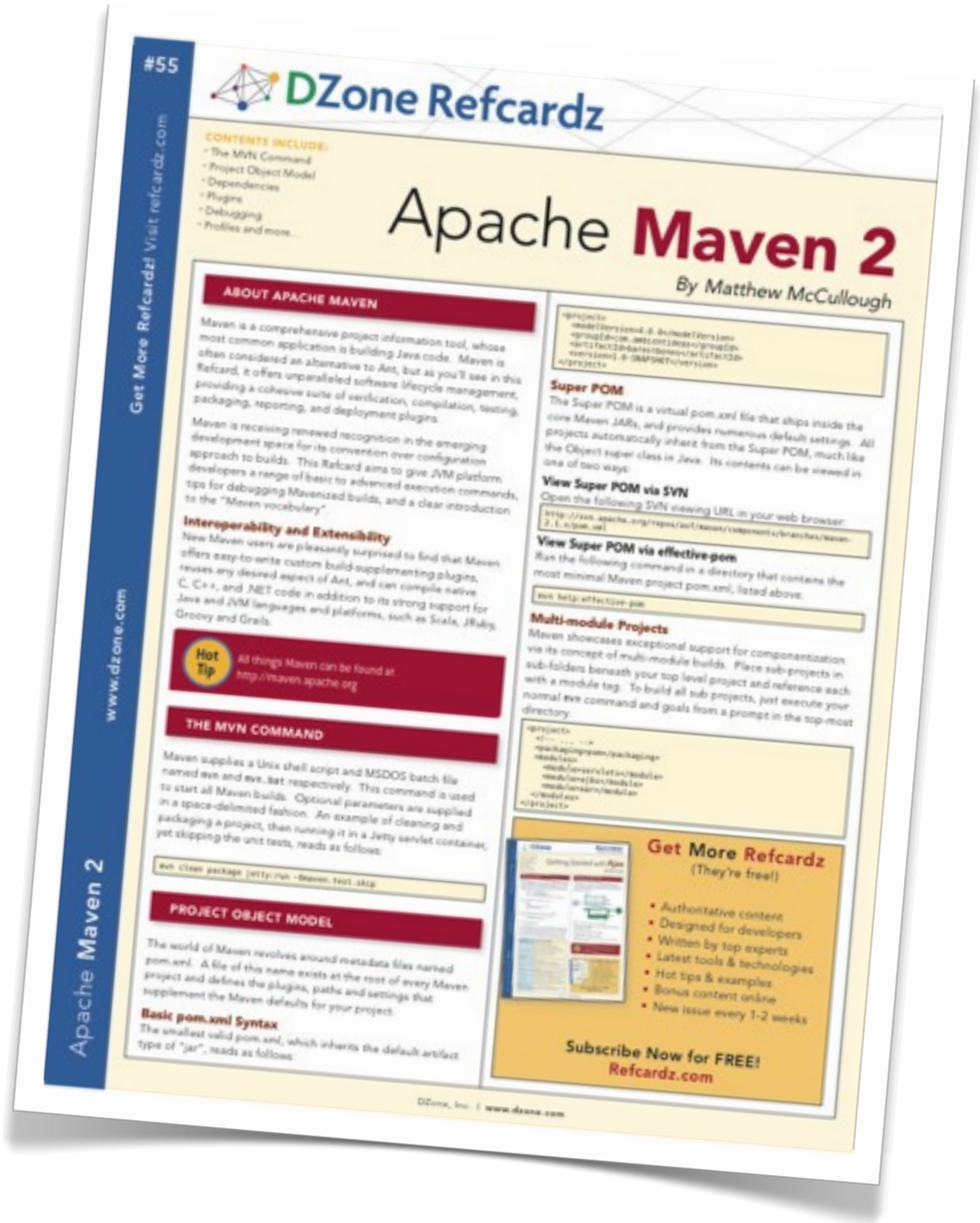
- Repository Management with Nexus
- Developing with Eclipse and Maven
- The Maven Handbook

- http://books.sonatype.com



DZone Maven RefCard

- **Card Homepage**
  - http://refcardz.dzone.com/refcardz/apache-maven-2
- **Quick Download**
  - http://dl.dropbox.com/u/53401/SonatypeTraining/rc055-010d-apachemaven-1.pdf



Release Notes

Maven – Release Notes

http://maven.apache.org/release-notes-all.html

Apache Maven Project
http:// maven.apache.org

Apache > Maven

Last Published: 2010-01-21

- Main
  - Welcome
- Get Maven
  - Download
  - Release Notes (2.2.1)
  - Release Notes (2.0.10)
  - License
- IDE Integration
  - Eclipse
  - Netbeans
- About Maven
  - What is Maven?
  - Features
  - FAQ (official)
  - FAQ (unofficial)
  - Powered By
- Documentation
  - Maven Plugins
  - Index (Category)
  - Running Maven
  - User Centre
  - Plugin Developer Centre
  - Maven Repository Centre
  - Maven Developer Centre
  - Books and Resources

Release Notes

2.2.1 Release Notes

Maven 2.2.1 aims to correct several critical regressions related to the selection of the HttpClient-based Wagon implementation for HTTP/HTTPS transfers in Maven 2.2.0. The new release reverts this selection, reinstating the Sun-based - or `lightweight` - Wagon implementation as the default for this sort of traffic. However, Maven 2.2.1 goes a step further to provide a means of selecting which `provider` - or implementation - the user wishes to use for a particular transfer protocol. More information on providers can be found in our Guide to Wagon Providers.

In addition, Maven 2.2.1 addresses some long-standing problems related to injecting custom lifecycle mappings and artifact handlers. These custom components are now correctly loaded regardless of whether they come from a plugin with the extensions flag enabled, or from a pure build extension. In addition, custom artifact handlers now will be used to configure the attributes of the main project artifact in addition to any artifacts related to dependencies or project attachments created during the build.

The full list of changes can be found in our issue management system, and is reproduced below.

Bug

- MNG-3265 - maven-model Extension.equals causes NPE when any field is uninitialized

http://maven.apache.org/release-notes-all.html

Other Build Tools

Other build tools

- GNU Make
  - Use a Makefile
  - Not designed for Enterprise Java Development
- Apache Ant
  - Uses a build.xml file
  - Maven can use Ant
  - Maven uses a pom.xml file
  - That's where the similarities end!

More Build Tools

- Buildr
  - An extension of the Ruby build tool (named Rake)
- Gant
  - Ant scripting in Groovy
- Gradle
  - Groovy-based, uses conventions like Maven but no XML
- Ivy
  - Dependency management for Ant

Maven and Ant

- Comparing Maven and Ant

Ant	Maven
• Procedural • Highly flexible, but complex • Transitive task dependencies • Each build script is very different • Extensible via Ant tasks	• Declarative • Convention over Configuration • Well-defined build lifecycle • Build scripts are standardized • Extensible via plugins

Maven and Ant

- Task dependencies vs predefined life cycle phases

Ant	Maven
Manual dependencies between tasks compile application <task name="compile"> *depends* compile test cases <task name="compile-tests"> *depends* run test cases <task name="test">	*Pre-defined life cycle phases* compile application **$mvn compile** compile test cases **$mvn test-compile** run test cases **$mvn test**

Maven and Ant

- Comparing Maven and Ant - a quick tour
  - Compiling in Ant

```
<project name="killer-app">
    ...
    <property name="src.dir" location="src/main/java"/>
    <property name="target.dir" location="target/classes"/>
    ...
    <path id="compile.classpath">
        <fileset dir="lib">
            <include name="**/*.jar"/>
        </fileset>
    </path>
    ...
    <target name="init">
        <mkdir directory="${target.dir}"/>
    </target>

    <target name="compile" depends="init"
            description="Compile the application classes">
        <javac srcdir="${src.dir}" destdir="${target.dir}"
               classpathref="compile.classpath" source="1.5" />
    </target>
</project>
```

Define directory definitions

Fetch your class dependencies

Create the target directory

Compile using **javac**

```
$ ant compile
```

Maven and Ant

- Comparing Maven and Ant - a quick tour
  - The same project in Maven

```
<project...>
    ...
    <build>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.5</source>
                    <target>1.5</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
    ...
</project>
```

Define some compiler options

No “javac” or “compile” target
- Maven knows how to compile your Java code

```
$ mvn compile
```

Maven and Ant

- Comparing Maven and Ant - a quick tour
  - But wait! There's more! In Maven, you also...

```
<!-- PROJECT DESCRIPTION -->
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycompany</groupId>
    <artifactId>myapp</artifactId>
    <packaging>jar</packaging>
    <name>Killer application</name>
    <version>1.0</version>
    <description>My new killer app</description>
    ...
```

Describe your project

Maven and Ant

- Comparing Maven and Ant - a quick tour
  - But wait! There's more! In Maven, you also...

```
<!-- DEPENDENCIES -->
  <dependencies>
      <!-- Hibernate -->
      <dependency>
          <groupId>org.hibernate</groupId>
          <artifactId>hibernate</artifactId>
          <version>3.2.4</version>
      </dependency>
      <!-- Log4j -->
      <dependency>
          <groupId>log4j</groupId>
          <artifactId>log4j</artifactId>
          <version>1.2.14</version>
      </dependency>
      ...
  </dependencies>
```

Declare your dependencies

Maven will download them for you as required

Maven and Ant

- Comparing Maven and Ant - a quick tour
  - The final Maven build script

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycompany</groupId>
    <artifactId>myapp</artifactId>
    <packaging>jar</packaging>
    <name>Killer application</name>
    <version>1.0</version>
    <description>My new killer app</description>
    <dependencies>
        <!-- Hibernate -->
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate</artifactId>
            <version>3.2.4</version>
        </dependency>
        ...
    </dependencies>
    <build>
        <plugins>
           <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.5</source>
                    <target>1.5</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Describe your project, not the low-level build logic

Built-in goals executed against the project meta-data

```
$ mvn compile
$ mvn test
$ mvn package
$ mvn install
$ mvn deploy
$ mvn ...
```

Maven and Ant

- Two very different approaches
  - Ant describes the build process.
    Maven describes the project
  - In Ant, you write tasks for every build phase.
    In Maven, the lifecycle phases are predefined.
  - In Ant, you keep your JAR files in a lib directory.
    In Maven, dependencies are declared.
  - *The two approaches are not necessarily in opposition!*

Review and Questions

- Maven
- Ant
- Gradle
- Buildr

Installing Maven

Overview

- Installing Maven
- Configuring Maven
- Maven installation directories

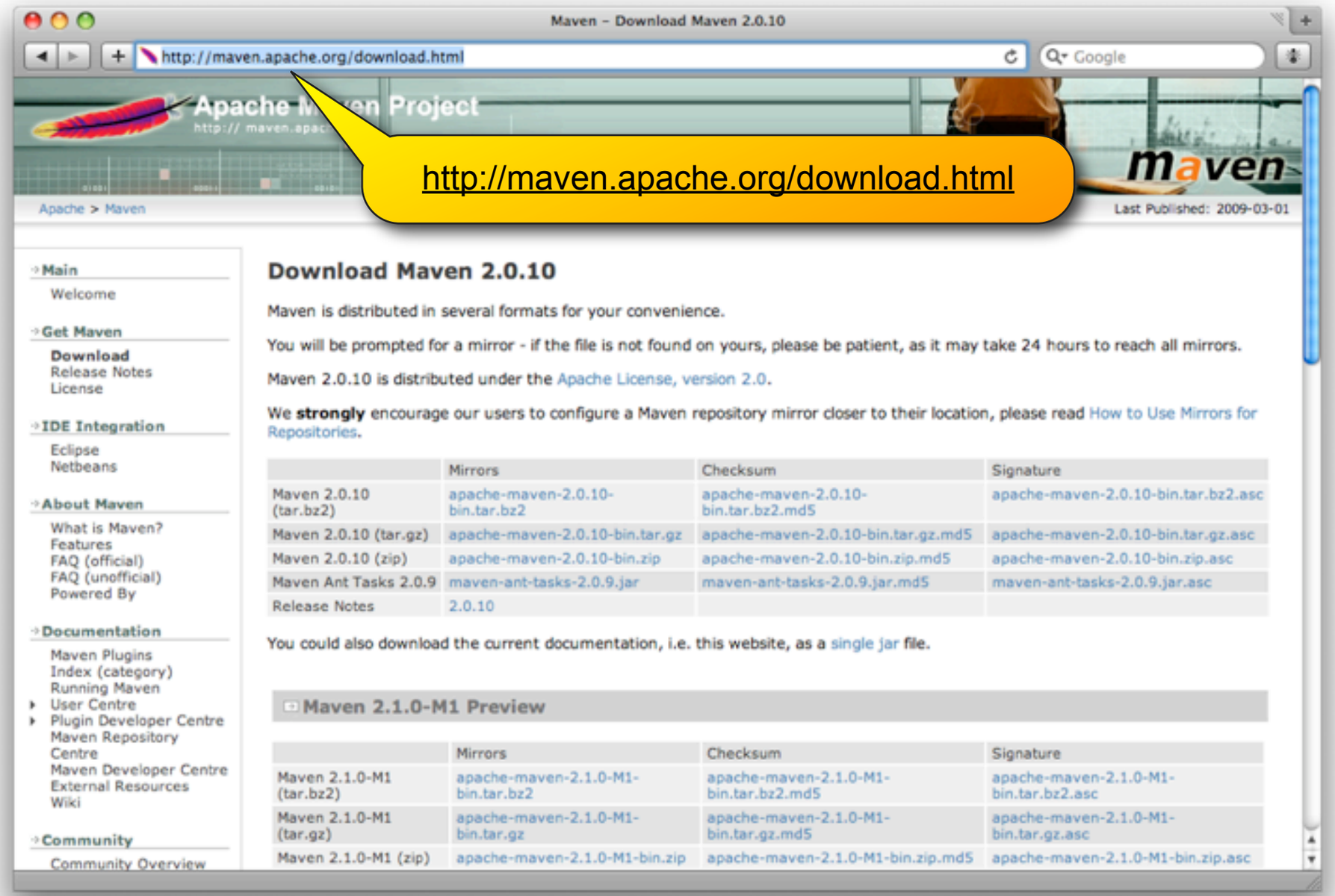
System pre-requisites

- Java
  - Maven 2.0.10 needs Java JDK 1.4 or better
  - Maven 2.2.1 needs Java JDK 1.5 or better

```
$ java -version
java version "1.5.0_16"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_16-b06-284)
Java HotSpot(TM) Client VM (build 1.5.0_16-133, mixed mode, sharing)
```

Installing Maven

- Download and decompress the latest binary bundle



The Installation Directory

- Unpack the Maven binary in an appropriate directory, .e.g.
  - Windows: C:\apache-maven-2.2.1
  - Mac OSX: ~/apache-maven-2.2.1 (or /usr/local/apache-maven-2.2.1)
  - Unix: ~/apache-maven-2.2.1 (or /usr/local/apache-maven-2.2.1)
- Set the M2_HOME environment variable
- On Mac OSX or Unix/Linux, create a symbolic link

```
/usr/local# tar xvfz apache-maven-2.2.1-bin.tar.gz
apache-maven-2.2.1/conf/
apache-maven-2.2.1/LICENSE.txt
.
.
.
# ln -s /usr/local/apache-maven-2.2.1 /usr/local/maven
# export M2_HOME=/usr/local/maven
# export PATH=${M2_HOME}/bin:${PATH}
```

Unpack the distribution

Create a symbolic link

Use the symbolic link for paths and environment variables

Installing Maven on Linux

- Maven is also bundled for many Linux distributions
  - e.g. use **apt-get install** on Debian-based distributions
  - You might not get the latest and greatest version

```
$ sudo apt-get install maven2
...
Unpacking maven2 (from .../maven2_2.0.9-2_all.deb) ...
Processing triggers for man-db ...
Setting up maven2 (2.0.9-2) ...

$ mvn --version
Maven version: 2.0.9
Java version: 1.6.0_10
OS name: "linux" version: "2.6.27-11-server" arch: "i386" Family: "unix"
```

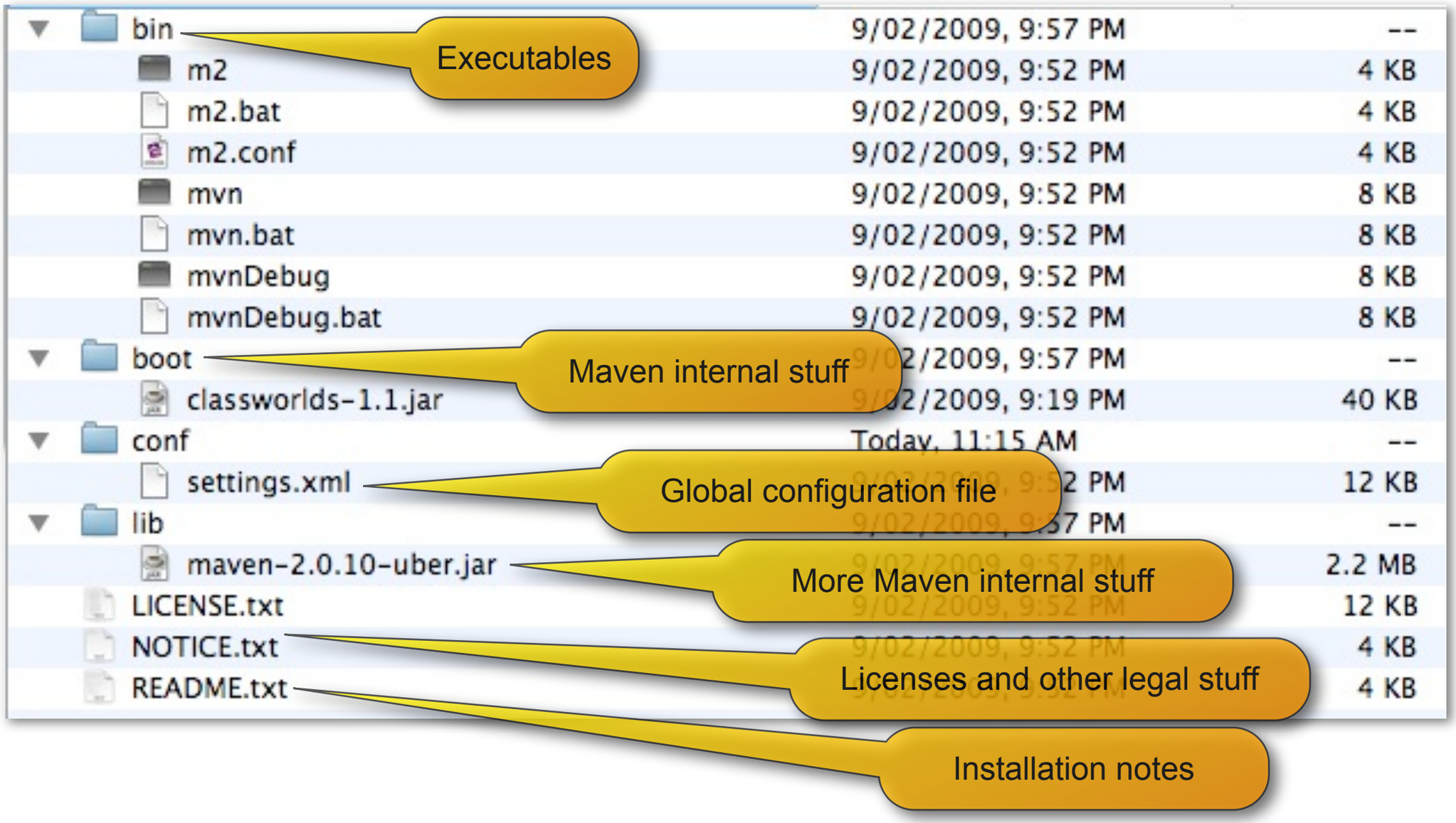
Maven is bundled for many Linux distributions

For Ubuntu, Maven is installed in /usr/share/maven2

- **Warning:** We recommend downloading Maven directly from maven.apache.org, packaged distributions are often out of date.

Maven Binary Directory

- When you download Maven and expand it, this is what you see:



Maven Directories

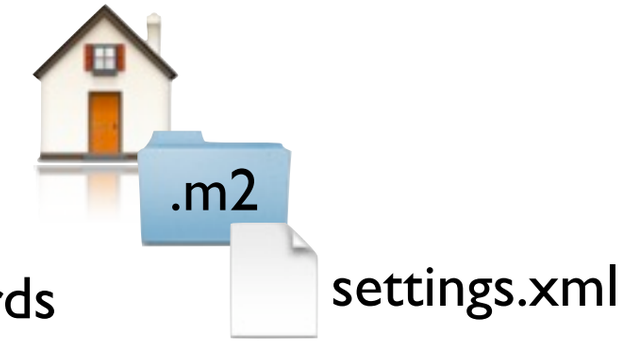
- Directories you should know about:
  - Maven Installation config directory
    - Stores Maven-system-wide settings.xml file
  - Local Maven directory
    - Stores user's Maven settings and downloaded artifacts
    - Downloads artifacts are placed in ~/.m2/repository

${M2_HOME}|/conf

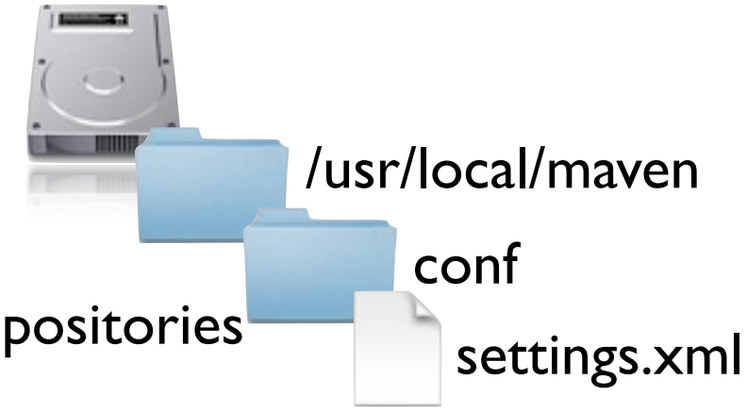
~/.m2/repository

Configuring Maven

- Maven configuration goes in the `settings.xml` file(s)
- There are two `settings.xml`
  - Local `settings.xml` file
    - `~/.m2/settings.xml`
    - Useful for private data such as usernames and passwords



  - Global `settings.xml` file
    - Located in a centrally installed copy of Maven
      - `${M2_HOME}/conf/settings.xml`
    - Useful for organizational settings such as enterprise repositories
    - All developers use this version of Maven



The settings.xml file

- A very simple `settings.xml` file:

```
<settings>
    <servers>
        <server>
            <id>acme-repo</id>
            <username>john</username>
            <password>t0ps3cr3t</password>
        </server>
    </servers>
</settings>
```

Specify authentication information to use when connection to some server

- A slightly more complicated `settings.xml` file:

```
<settings>
    <profiles>
        <profile>
            <id>myprofile</id>
            <activation>
                <activeByDefault>true</activeByDefault>
            </activation>
            <properties>
                <jdbc.user>john</jdbc.user>
                <jdbc.password>t0ps3cr3t</jdbc.password>
            </properties>
        </profile>
    </profiles>
</settings>
```

Define some properties in a build profile

The settings.xml file

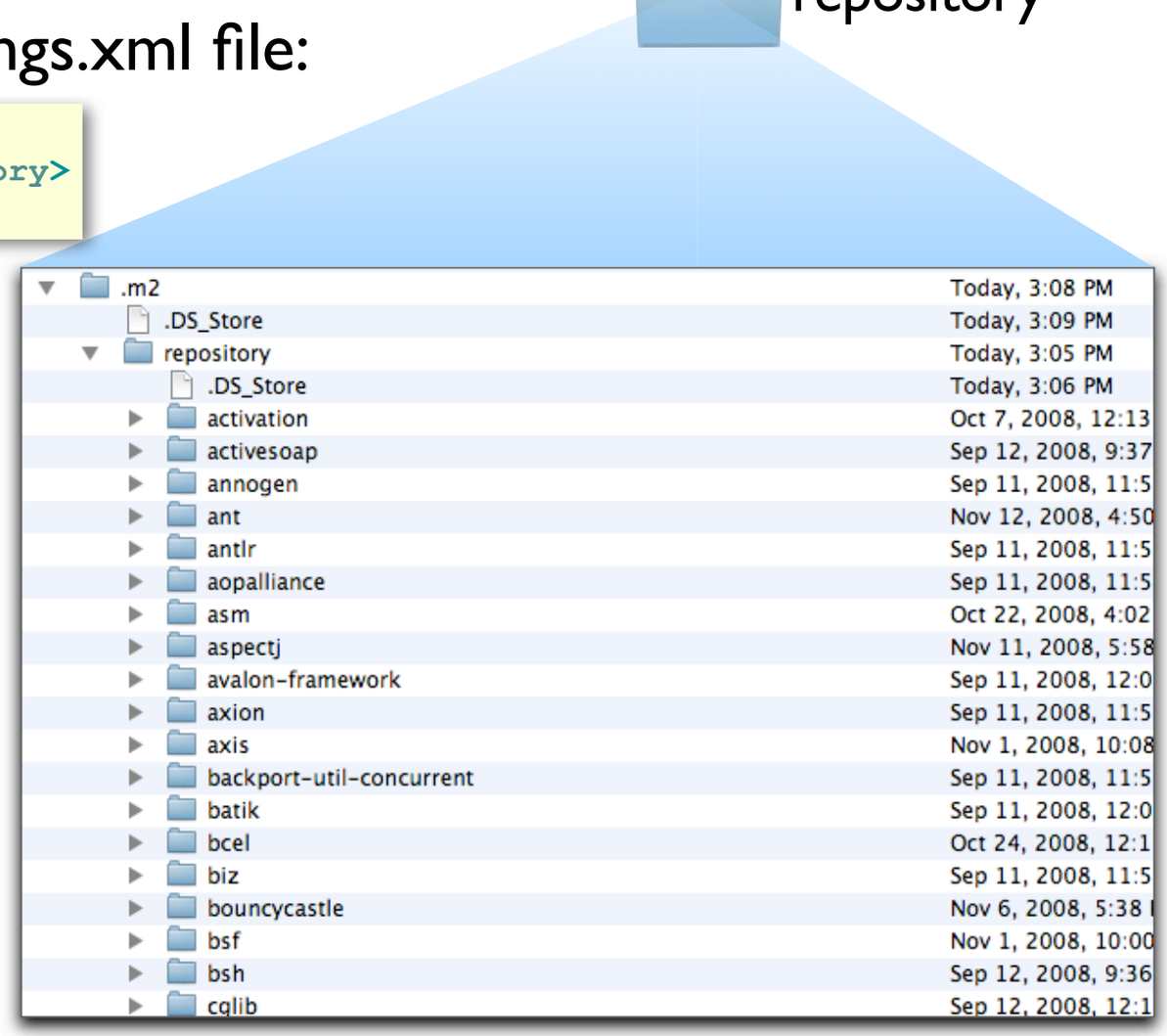
- The `settings.xml` files are useful for:
  - Security credentials for accessing servers
  - Configurations for mirror and proxy repositories

The local Maven repository

- Maven stores downloaded libraries locally
  - By default, downloaded artifacts are placed in ~/.m2/repository
  - You can override this in the settings.xml file:

```
<settings>
    <localRepository>C:/repos</localRepository>
</settings>
```

Store the local repository in C:\repos



Executing Maven

- Running Maven from the command line

```
$ mvn compile
...
```

Invoke Maven

A Maven goal or life cycle phase

Maven options

```
[~]$ mvn -h

usage: mvn [options] [<goal(s)>] [<phase(s)>]

Options:
 -am,--also-make                          If project list is specified, also build projects required by the list
 -amd,--also-make-dependents              If project list is specified, also build projects that depend on prjs on the
list
 -B,--batch-mode                          Run in non-interactive (batch) mode
 -C,--strict-checksums                    Fail the build if checksums don't match
 -c,--lax-checksums                       Warn if checksums don't match
 -cpu,--check-plugin-updates              Force upToDate check for any relevant registered plugins
 -D,--define <arg>                        Define a system property
 -e,--errors                              Produce execution error messages
 -emp,--encrypt-master-password <arg>     Encrypt master security password
 -ep,--encrypt-password <arg>             Encrypt server password
 -f,--file                                Force the use of an alternate POM file.
 -fae,--fail-at-end                       Only fail the build afterwards; allow all non-impacted builds to continue
 -ff,--fail-fast                          Stop at first failure in reactorized builds
 -fn,--fail-never                         NEVER fail the build, regardless of project result
 -gs,--global-settings <arg>              Alternate path for the global settings file
 -h,--help                                Display help information
 -l,--log-file <arg>                      Log file to where all build output will go.
 -N,--non-recursive                       Do not recurse into sub-projects
 -npr,--no-plugin-registry                Ineffective, only kept for backward compatibility
 -npu,--no-plugin-updates                 Suppress upToDate check for any relevant registered plugins
 -nsu,--no-snapshot-updates               Supress SNAPSHOT updates
 -o,--offline                             Work offline
 -P,--activate-profiles <arg>             Comma-delimited list of profiles to activate
 -pl,--projects <arg>                     Build specified reactor projects instead of all projects. A project
                                          can be specified by [groupId]:artifactId or by its relative path.
 -q,--quiet                               Quiet output - only show errors
 -rf,--resume-from <arg>                  Resume reactor from specified project
 -s,--settings <arg>                      Alternate path for the user settings file
 -U,--update-snapshots                    Forces a check for updated releases and snapshots on remote repositories
 -up,--update-plugins                     Synonym for cpu
 -V,--show-version                        Display version information WITHOUT stopping build
 -v,--version                             Display version information
 -X,--debug                               Produce execution debug output
```

Common Maven Options

- -o (offline)
  - Used to tell Maven not to go online for downloads
- -X (debug)
  - Enable Maven's debug output for troubleshooting
- -D<property>
  - -Dmaven.test.skip=true
- -e (errors)
  - Output errors during the build process
- -P <profile>
  - Activate a Build Profile

Upgrading Maven

- Installing a new version of Maven
  - Download the new version
  - Expand it and place it right next to the existing version
  - Update the M2_HOME environment variable to point to the new version
  - If the PATH is set to reference $M2_HOME/bin, no need to change it.

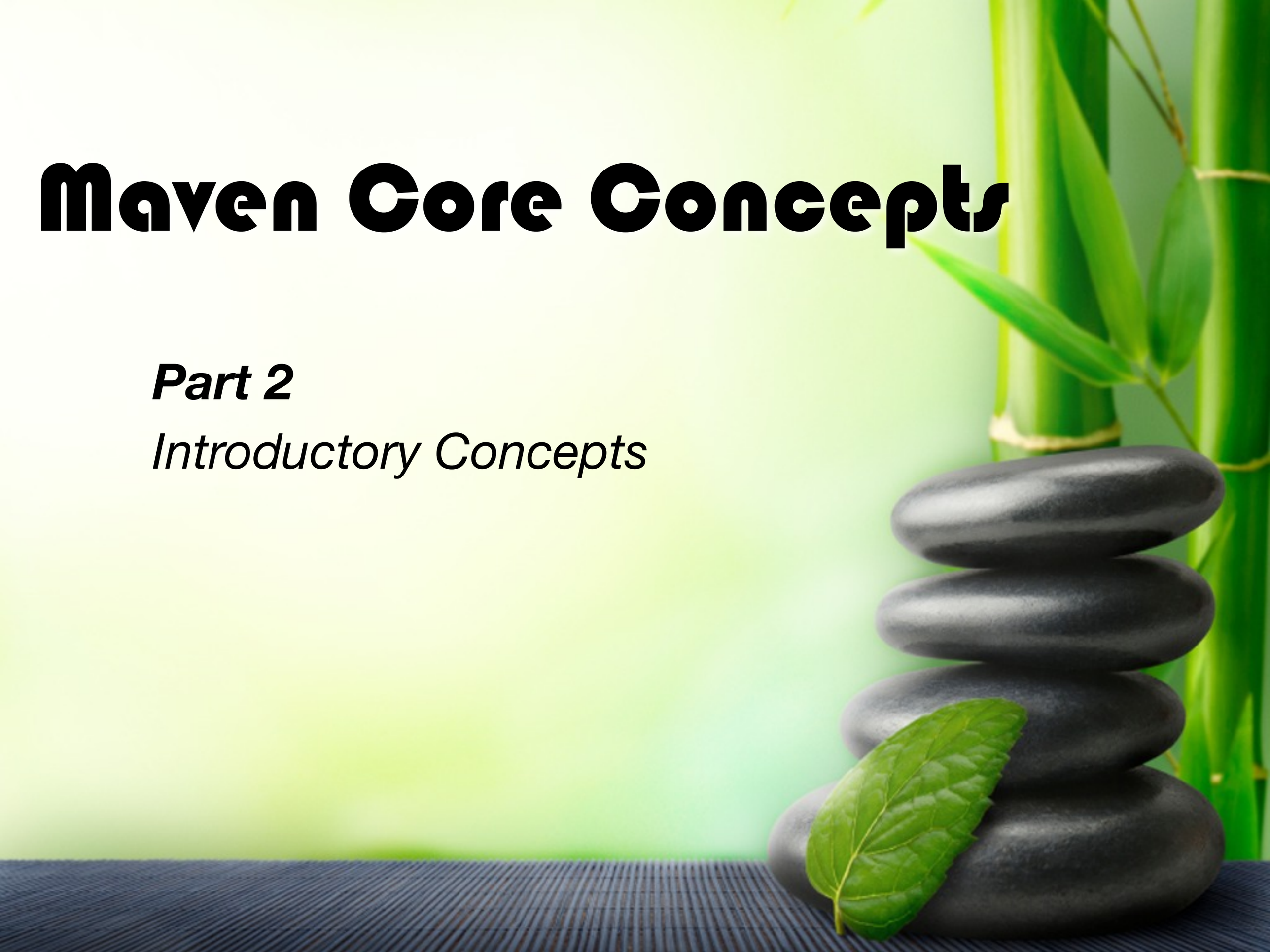
Questions

- Versions of Maven in use?
- Multiple versions on your team?
- Upgrade process?
- What does upgrading the core of Maven give you?
- Check the release notes.

Maven Core Concepts

Part 2

Introductory Concepts



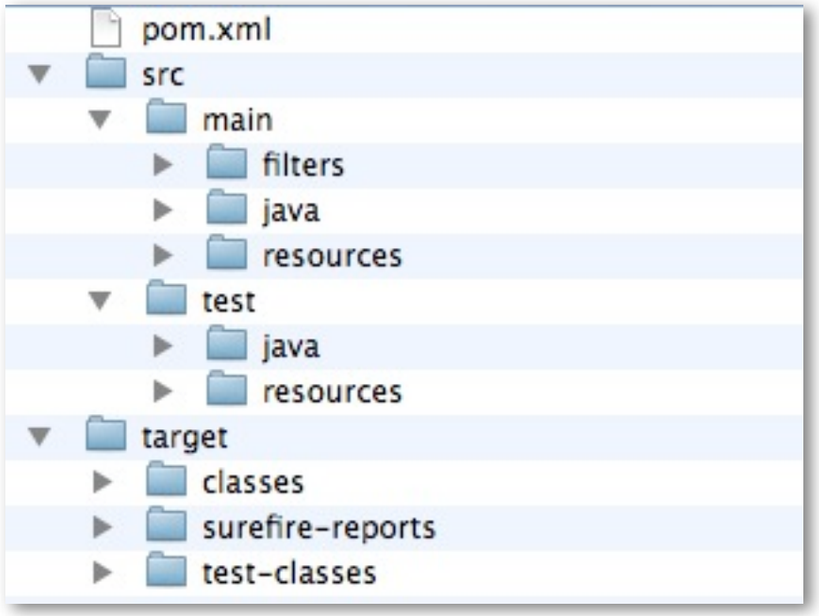
Overview

- Maven Core Concepts
  - A Standard Directory Structure
  - Artifact Co-ordinates
  - A Standard Lifecycle
  - Plugins and Goals
  - Customizing the Lifecycle
  - Maven Repositories

Standard Directory Structure

Maven Directory Structure

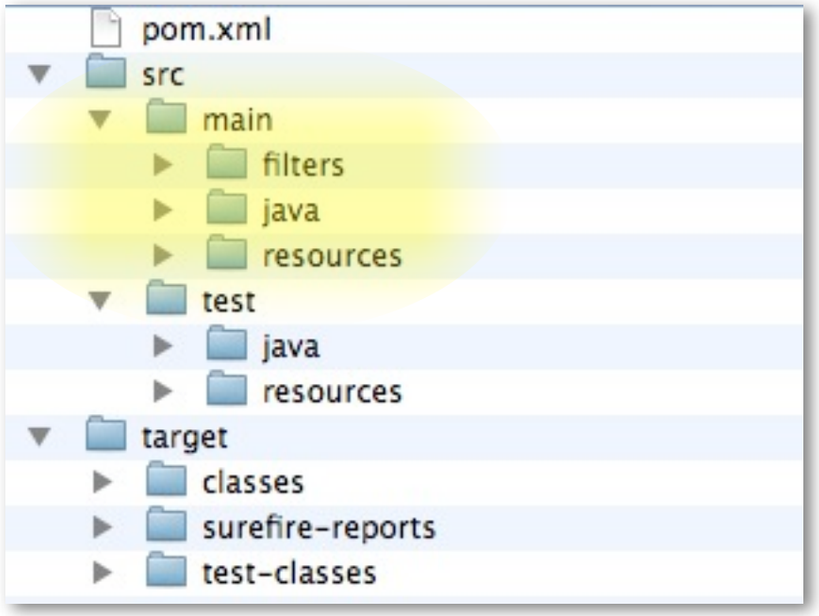
- Maven defines a Standard Directory Structure
  - Familiar structure across all projects
  - Convention Over Configuration
  - Less low-level scripting
  - Lower learning curve on new projects



Maven Directory Structure

- Maven defines a Standard Directory Structure
  - Production source code goes in `src/main`

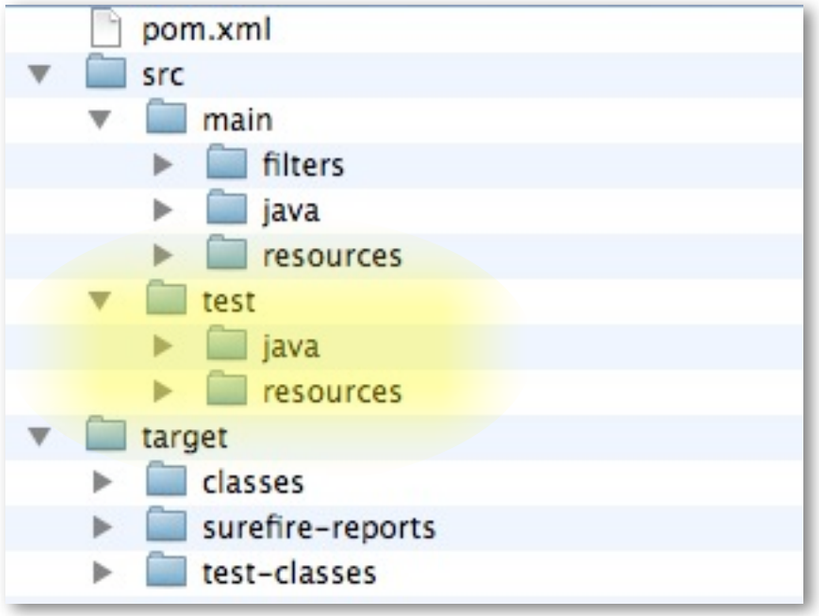
`src/main/java`	Java source code
`src/main/resources`	Other resources your application needs
`src/main/webapp`	The web application directory for a WAR project
`src/main/filters`	Resource filters



Maven Directory Structure

- Maven defines a Standard Directory Structure
  - Test source code goes in `src/test`

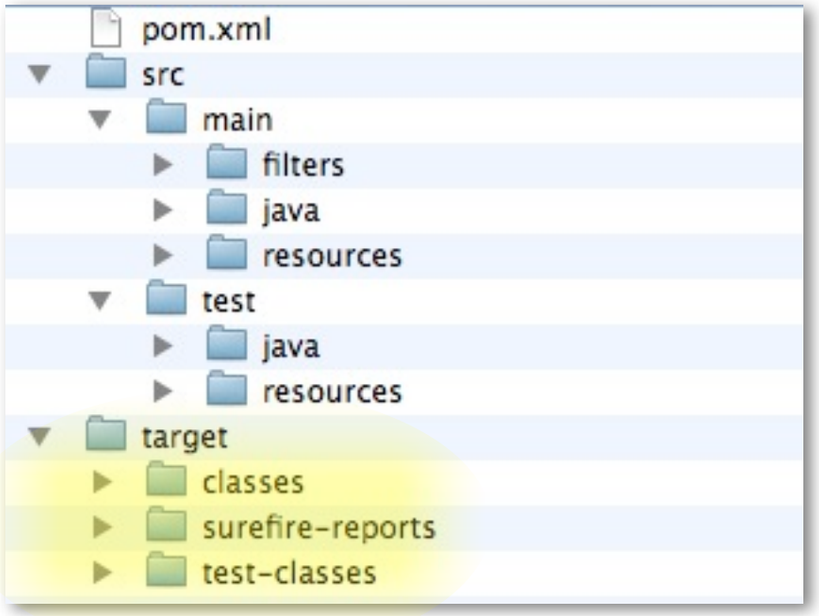
`src/test/java`	Test classes source code
`src/test/resources`	Resources used for testing
`src/test/filters`	Filters to be used for testing purposes



Maven Directory Structure

- Maven defines a Standard Directory Structure
  - Compiled code and reports go in `target`

`target/classes`	Compiled Java classes
`target/test-classes`	Compiled test classes
`target/surefire-reports`	Unit test reports



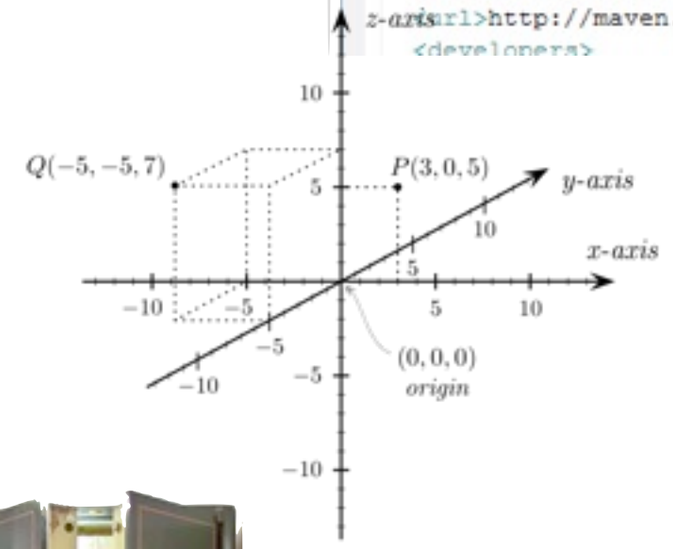
The pom.xml file

Describing your project

- The `pom.xml` file - a description of your project
  - Versioning and packaging details
  - Team members
  - SCM and issue tracking configuration
  - Dependencies
  - Build lifecycle extensions
  - Reporting
  - ...

The Project Object Model

- XML Representation of a Project
  - Describes Project Structure + Contents
  - Describes Project Build
- Defines Project Coordinates
  - Unique Identifier for a Project
  - Allows relationships between projects
- Provides Project Metadata
  - Where is the source code?
  - Where are the resources?
  - What is the packaging type?



Comparing the POM

- POM is Declarative
  - Maven can also be procedural when necessary
- Most Other Build Tools are Procedural
  - Make, Ant, Gant
  - Other Build Tools focus on "Scripts"
- Maven and the POM are not specific to managing just Java projects
  - Given the correct plugins, Maven can build any language
    - E.g., C/C++, C#, Actionscript, Flex, Ruby, Javascript, Docbook XML, etc.

The Simplest POM

- Defines the three required coordinates
  - groupId, artifactId, version
- This POM defines no dependencies
- This Project can...
  - ...build Java code in `src/main/java`
  - ...execute tests in `src/main/test`
  - ...create a jar artifact
  - ...produce a project site

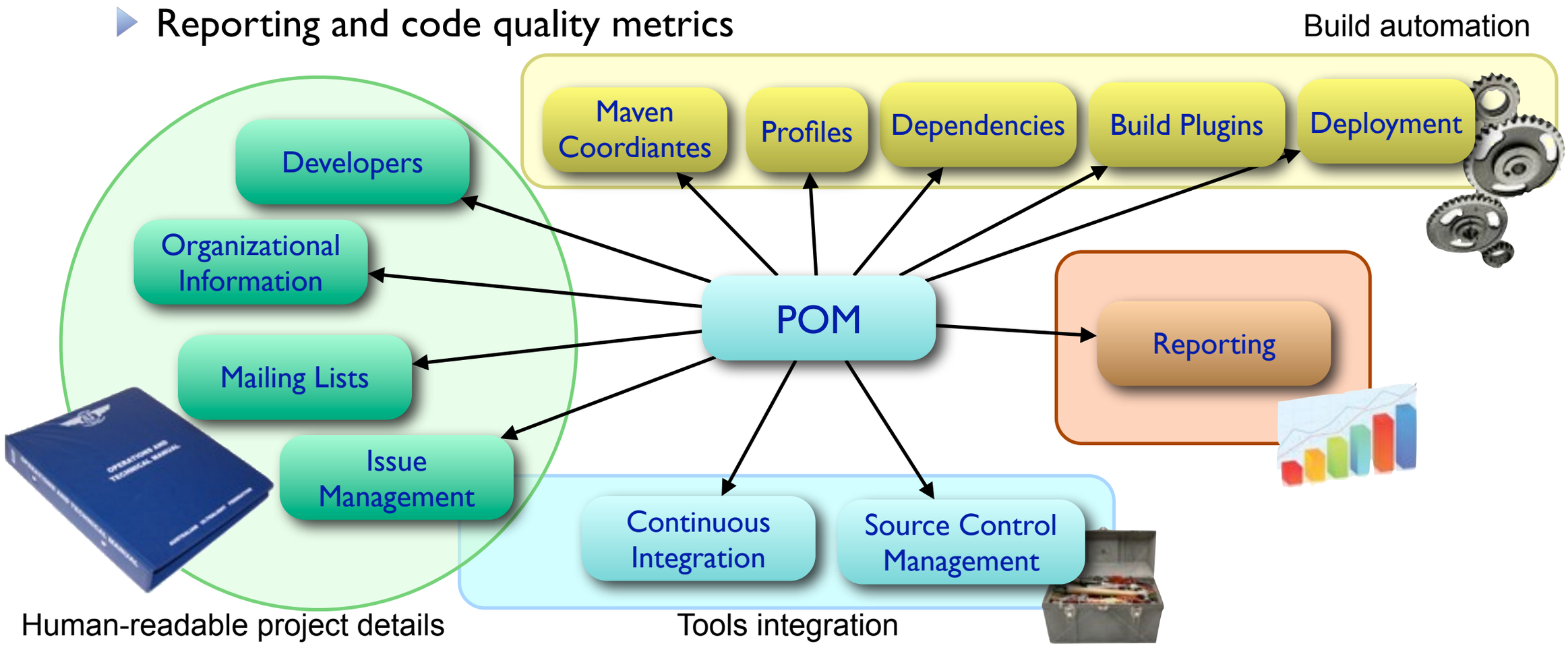
Identify the project

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany</groupId>
  <artifactId>simplest-project</artifactId>
  <version>1</version>
</project>
```

```
$ mvn compile
$ mvn test
$ mvn package
$ mvn site
```

What goes into a POM file?

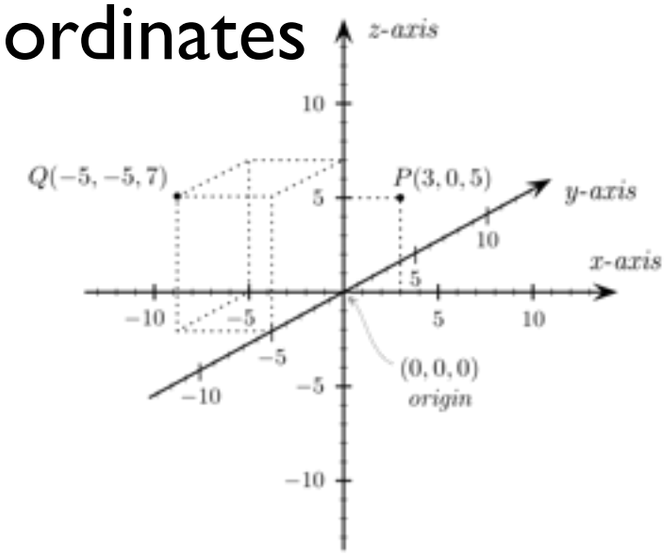
- A real-world POM file can contain much more than that...
  - Build automation details (dependencies, plugins, profiles,...)
  - Human-readable details (team members, issue trackers...)
  - Tools integration
  - Reporting and code quality metrics



Artifact Coordinates

Identifying your project

- Each Maven artifact is identified by a set of co-ordinates
  - groupId
  - artifactId
  - version
  - packaging
  - classifier



- These co-ordinates Provide a Unique Identifier for an Artifact
  - Basis for dependency management
  - Repositories Organized According to Maven Coordinates
- *groupId*, *artifactId* and *version* are required attributes

```
<project...>
    ...
    <groupId>com.mycompany</groupId>
    <artifactId>myapp</artifactId>
    <version>1.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>myapp</name>
   ...
```

Maven Coordinates Examples

- Maven coordinates in action...

Project coordinates

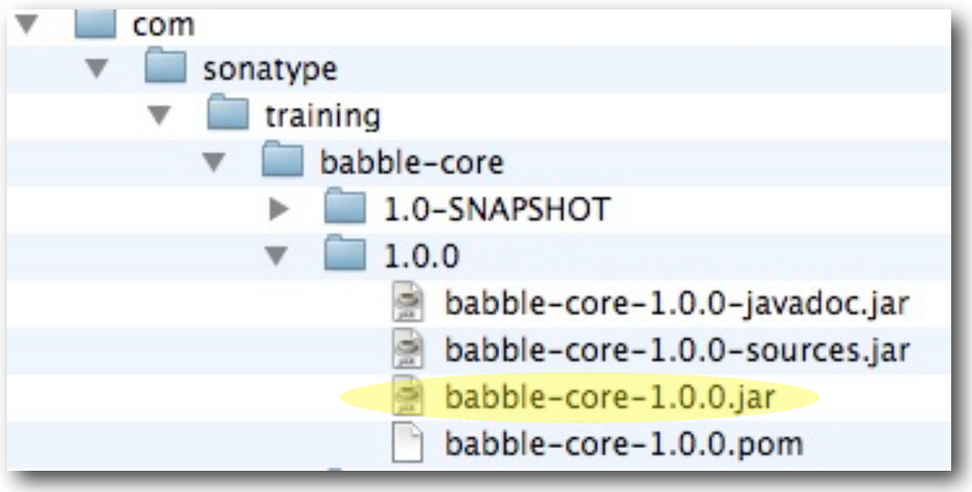
Coordinates in a dependency

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany</groupId>
  <artifactId>helloworld</artifactId>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
 </dependencies>
</project>
```

Maven Coordinates

- Maven uses artifact coordinates:
  - To uniquely identify an artifact
  - To store and retrieve artifacts in a repository
  - To resolve dependencies
- A set of artifact co-ordinates maps to a path, e.g.

```
<project...>
    ...
    <groupId>com.sonatype.training</groupId>
    <artifactId>babble-core</artifactId>
    <packaging>jar</packaging>
    <version>1.0.0</version>
    ...
```



The GroupId Element

- groupId
  - Generally unique to an organization or a project
  - Usually adheres to reverse domain name + category/purpose
  - Lower-case, alpha-numeric characters
  - In Repository: Each dot replaced by a separator to form a relative file path
  - Examples:
    - JUnit - junit
    - Maven - org.apache.maven
    - Castor - org.codehaus.castor

The ArtifactId Element

- artifactId
  - A project identifier
  - Lower-case, alpha-numeric characters
  - Dashes allowed as word separators
  - groupId and artifactId creates a Unique Key for a Project
  - Examples:
    - JUnit - junit:junit
    - Maven - org.apache.maven:maven-core
    - Castor - org.codehaus.castor:castor

The Version Element

- The version of the project
- Triplet number is the best-practices pattern
  - 1.0.0
  - 3.8.1-releasecandidate
  - 4.5-beta-SNAPSHOT
- If using non-standard versions like 1.0.0.1, version ranges don't work (are alpha parsed)

The Version Element

- groupId, artifactId, and version create a Unique Key for a Project Version
- E.g.:
  - JUnit - junit:junit:4.4
  - Maven - org.apache.maven:maven-core:2.0.9
  - Castor - org.codehaus.castor:castor:1.2

Project Versions

- A project's version is used to group and order releases
- Proper Maven project versions
  - <major version>.<minor version>.<incremental version>-<qualifier>
    - E.g.:
      - 1.0.0
      - 3.3.2-b1
      - 4.0.0-beta01
    - Maven handles project version comparison so the format is important

Project version

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany</groupId>
  <artifactId>helloworld</artifactId>
  <version>1.0.0</version>
  <name>Maven Quick Start Archetype</name>
  ...
</project>
```

Snapshot Versions

- The word SNAPSHOT has significant meaning to Maven
  - SNAPSHOT = this project is under active development
- SNAPSHOT is interpolated to a date/time value upon install or release
  - E.g.
    - 2.6.1-SNAPSHOT will become 2.6.1-20100711-024752-01 (2010 July 11 at 2:47:52am)

Root Version

Timestamp

Build Number

- A snapshot from a specific date/time of a given project

A SNAPSHOT version

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany</groupId>
  <artifactId>helloworld</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  ...
</project>
```

Snapshot Versions

- Lets look at the contents of a snapshot folder in a repository
  - Index of /content/repositories/snapshots/org/sonatype/training/training-test/1.0.0-SNAPSHOT/

GroupId

ArtifactId

Version

```
maven-metadata.xml        Wed Jul 15 13:27:30 CDT 2009      582
maven-metadata.xml.md5
maven-metadata.xml.sha1   Wed Jul 15
training-test-1.0.0-20090715.182635-1.jar
training-test-1.0.0-20090715.182635-1.jar.m
training-test-1.0.0-20090715.182635-1.jar.sh
training-test-1.0.0-20090715.182635-1.pom
training-test-1.0.0-20090715.182635-1.pom.
training-test-1.0.0-20090715.182635-1.pom.
training-test-1.0.0-20090715.182647-2.jar
training-test-1.0.0-20090715.182647-2.jar.m
training-test-1.0.0-20090715.182647-2.jar.sh
training-test-1.0.0-20090715.182647-2.pom
training-test-1.0.0-20090715.182647-2.pom.
training-test-1.0.0-20090715.182647-2.pom.sha1
```

```
<metadata>
  <groupId>org.sonatype.training</groupId>
  <artifactId>training-test</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <versioning>
    <snapshot>
      <timestamp>20090715.18647</timestamp>
      <buildNumber>2</buildNumber>
    </snapshot>
    <lastUpdated>20090715182647</lastUpdated>
  </versioning>
</metadata>
```

Snapshot Versions

- Maven updates SNAPSHOT dependencies and plugins every 24 hours
  - The SNAPSHOT may have changed since the last time you built your project.
  - The update frequency is configurable
  - You can manually force an update check with -U
- Maven will not attempt to retrieve SNAPSHOTs from **central** by default
  - Snapshots must explicitly be enabled on the repository or pluginRepository

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <name>Maven Default Project</name>

  <repositories>
    <repository>
      <id>central</id>
      <name>Maven Repository Switchboard</name>
      <layout>default</layout>
      <url>http://repo1.maven.org/maven2</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>central</id>
      <name>Maven Plugin Repository</name>
      <url>http://repo1.maven.org/maven2</url>
      ...
    </pluginRepository>
  </pluginRepositories>
  ...
</project>
```

Snapshot Versions

- Snapshots and Product Releases
  - Never a good idea to release your project(s) with SNAPSHOT dependencies
  - Maven **release:prepare plugin goal** disallows SNAPSHOT dependencies
  - Artifacts published to the central repository are not allowed to depend on SNAPSHOTs

The Packaging Element

- packaging
  - Determines the type of artifact to be produced
  - Optional element - defaults to JAR
  - Determines the set of goals to bind to a particular phase
  - Available packaging types include jar, war, ear, pom, etc.
  - Plugin executions can be associated with lifecycle phases via the POM

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany</groupId>
  <artifactId>amq-jmx</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>amq-jmx</name>
  <url>http://maven.apache.org</url>
   ...
</project>
```

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany</groupId>
  <artifactId>amq-jmx-webapp</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>amq-jmx</name>
  <url>http://maven.apache.org</url>
   ...
</project>
```

The Classifier Element

- classifier
  - Used as a further distinguishing characteristic
  - Distinguishes different builds of the same application version
    - E.g. Different builds for different JDKs
  - E.g.
    - JRE 1.4 - org.apache.foo:bar:jar:jdk14:1.2
    - JRE 1.5 - org.apache.foo:bar:jar:jdk15:1.2
    - Source - org.apache.foo:bar:jar:sources:1.2
    - Javadoc - org.apache.foo:bar:jar:javadoc:1.2

```
<dependency>
  <groupId>org.testng</groupId>
  <artifactId>testng</artifactId>
  <version>5.7</version>
  <classifier>jdk15</classifier>
  <scope>test</scope>
</dependency>
```

Index of /maven2/net/sf/json-lib/json-lib/1.0/

http://repo2.maven.org/maven2/net/sf/json-lib/json-lib/1.0/

Index of /maven2/net/sf/json-li...

Index of /maven2/net/sf/json-lib/json-lib/1.0/

../		
json-lib-1.0-jdk13-javadoc.jar	12-Feb-2007 01:26	121689
json-lib-1.0-jdk13-javadoc.jar.md5	12-Feb-2007 01:26	32
json-lib-1.0-jdk13-javadoc.jar.sha1	12-Feb-2007 01:26	40
json-lib-1.0-jdk13-sources.jar	12-Feb-2007 01:26	54597
json-lib-1.0-jdk13-sources.jar.md5	12-Feb-2007 01:26	32
json-lib-1.0-jdk13-sources.jar.sha1	12-Feb-2007 01:26	40
json-lib-1.0-jdk13.jar	12-Feb-2007 01:27	54334
json-lib-1.0-jdk13.jar.md5	12-Feb-2007 01:28	32
json-lib-1.0-jdk13.jar.sha1	12-Feb-2007 01:28	40
json-lib-1.0-jdk15-javadoc.jar	12-Feb-2007 01:26	124575
json-lib-1.0-jdk15-javadoc.jar.md5	12-Feb-2007 01:26	32
json-lib-1.0-jdk15-javadoc.jar.sha1	12-Feb-2007 01:26	40
json-lib-1.0-jdk15-sources.jar	12-Feb-2007 01:27	55710
json-lib-1.0-jdk15-sources.jar.md5	12-Feb-2007 01:27	32
json-lib-1.0-jdk15-sources.jar.sha1	12-Feb-2007 01:27	40
json-lib-1.0-jdk15.jar	12-Feb-2007 01:27	54803
json-lib-1.0-jdk15.jar.md5	12-Feb-2007 01:27	32
json-lib-1.0-jdk15.jar.sha1	12-Feb-2007 01:27	40
json-lib-1.0.pom	12-Feb-2007 01:28	10277
json-lib-1.0.pom.md5	12-Feb-2007 01:28	32
json-lib-1.0.pom.sha1	12-Feb-2007 01:28	40

Display a menu

Questions

- Why the strong conventions on the version numbers?
- Does every Maven project follow this pattern?
- Why do artifacts have alpha and beta in the name for years and years?

Standardized Project lifecycle

A Standardized Life Cycle

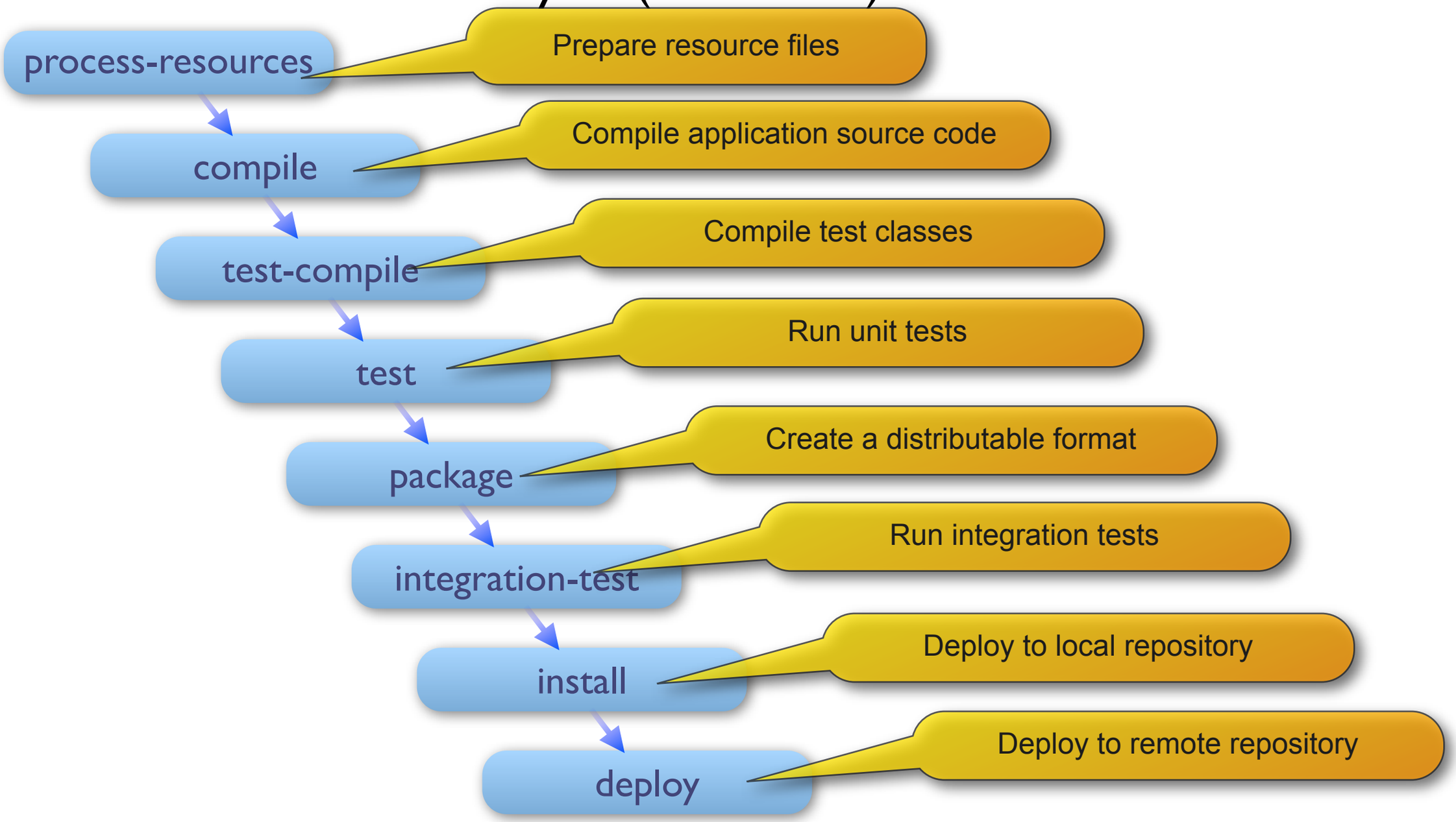
- Maven provides standardized life cycles for projects
  - Reduced learning curve between projects
  - Allows for standardized, repeatable builds across projects
  - Customized through choices in POMs

A Standardized Life Cycle

- Provides for many typical development steps:
  - Preparing source code for compilation
  - Compiling code
  - Running unit tests
  - Packaging applications
  - Running integration tests
  - Deploying to local and remote repositories

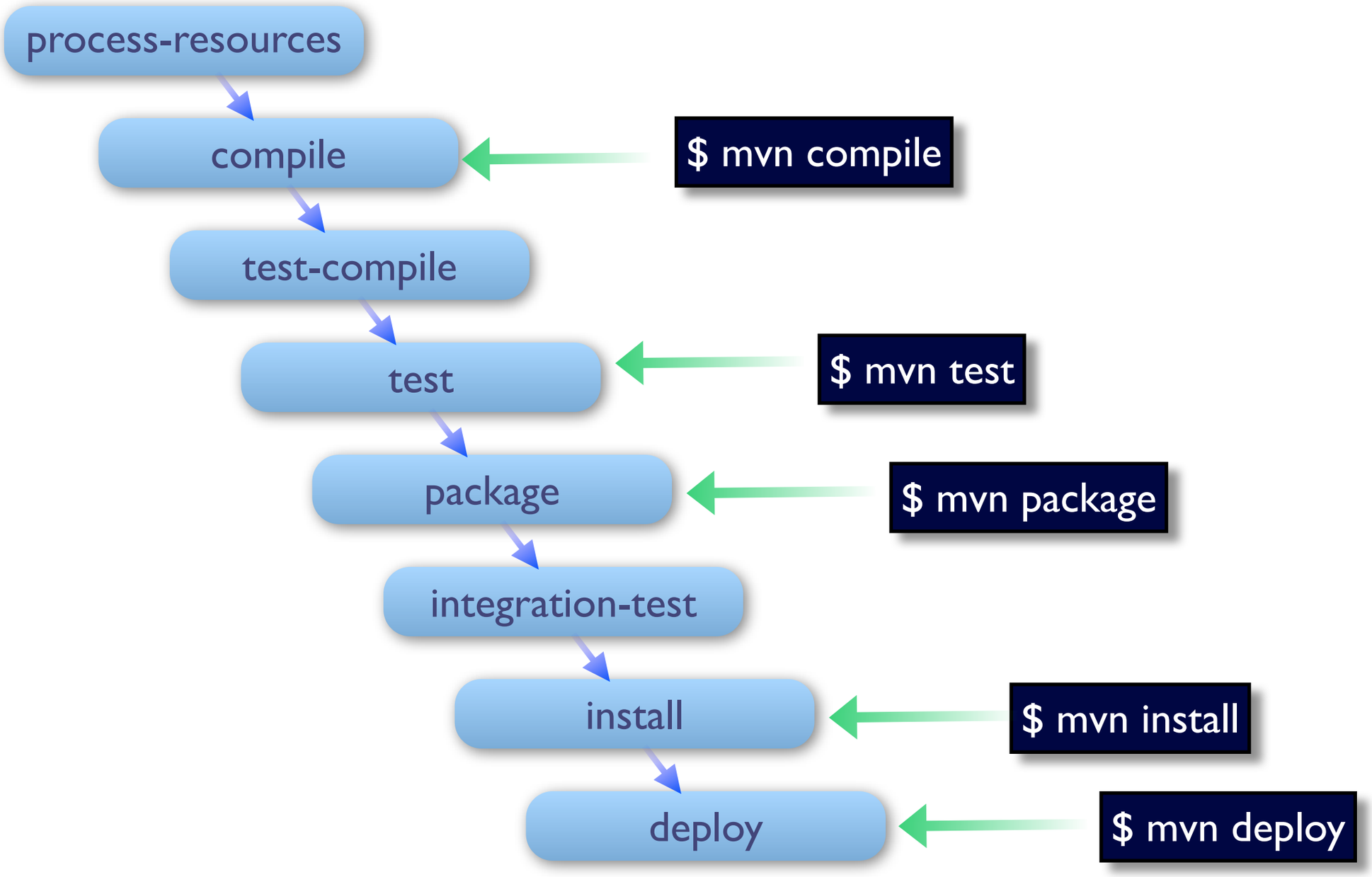
A Standardized Life Cycle

- The standard Maven lifecycle (an extract)



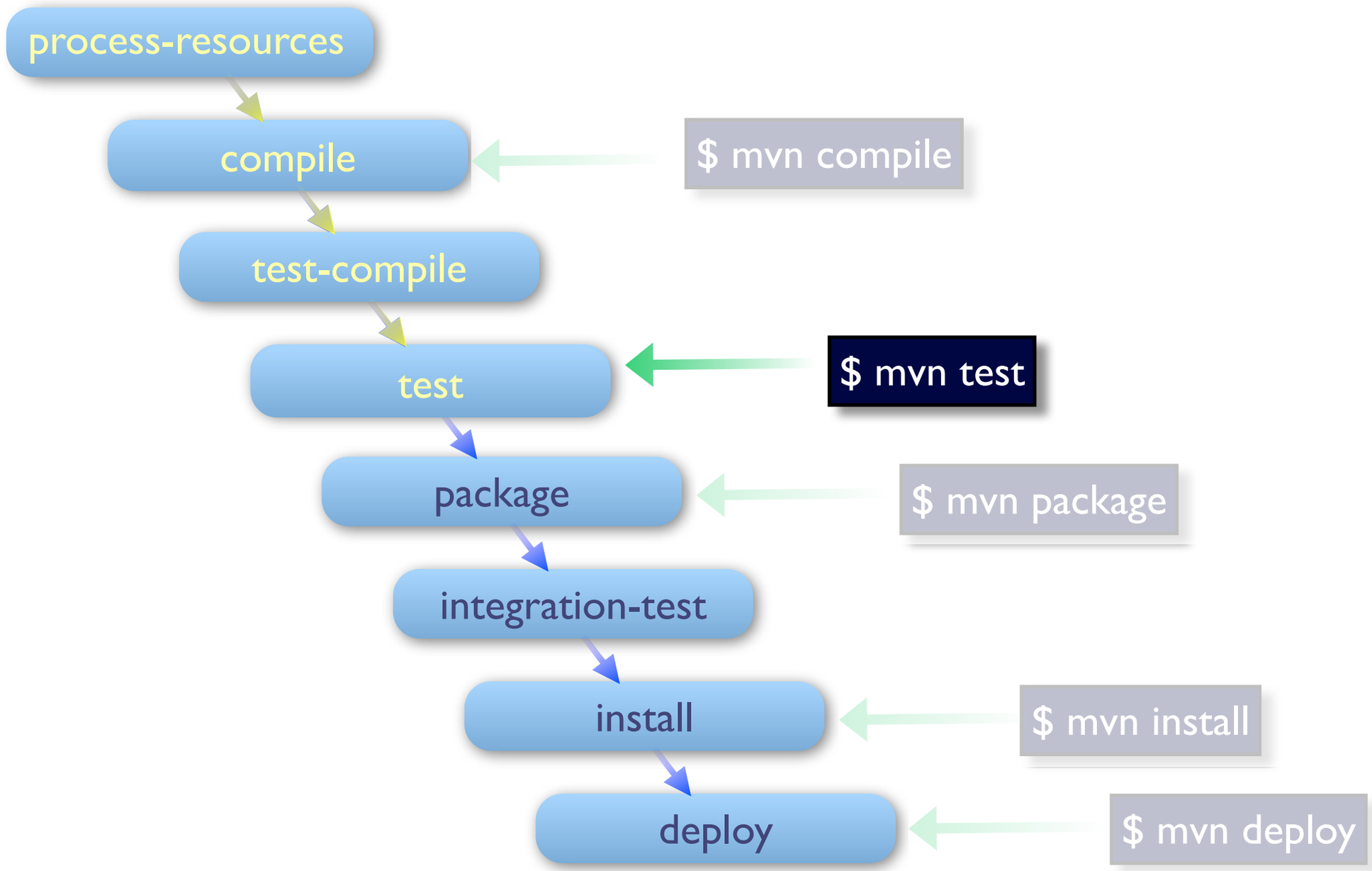
A Standardized Life Cycle

- You can invoke life cycle phases directly...



A Standardized Life Cycle

- Invoking a life cycle phase will also invoke the previous phases



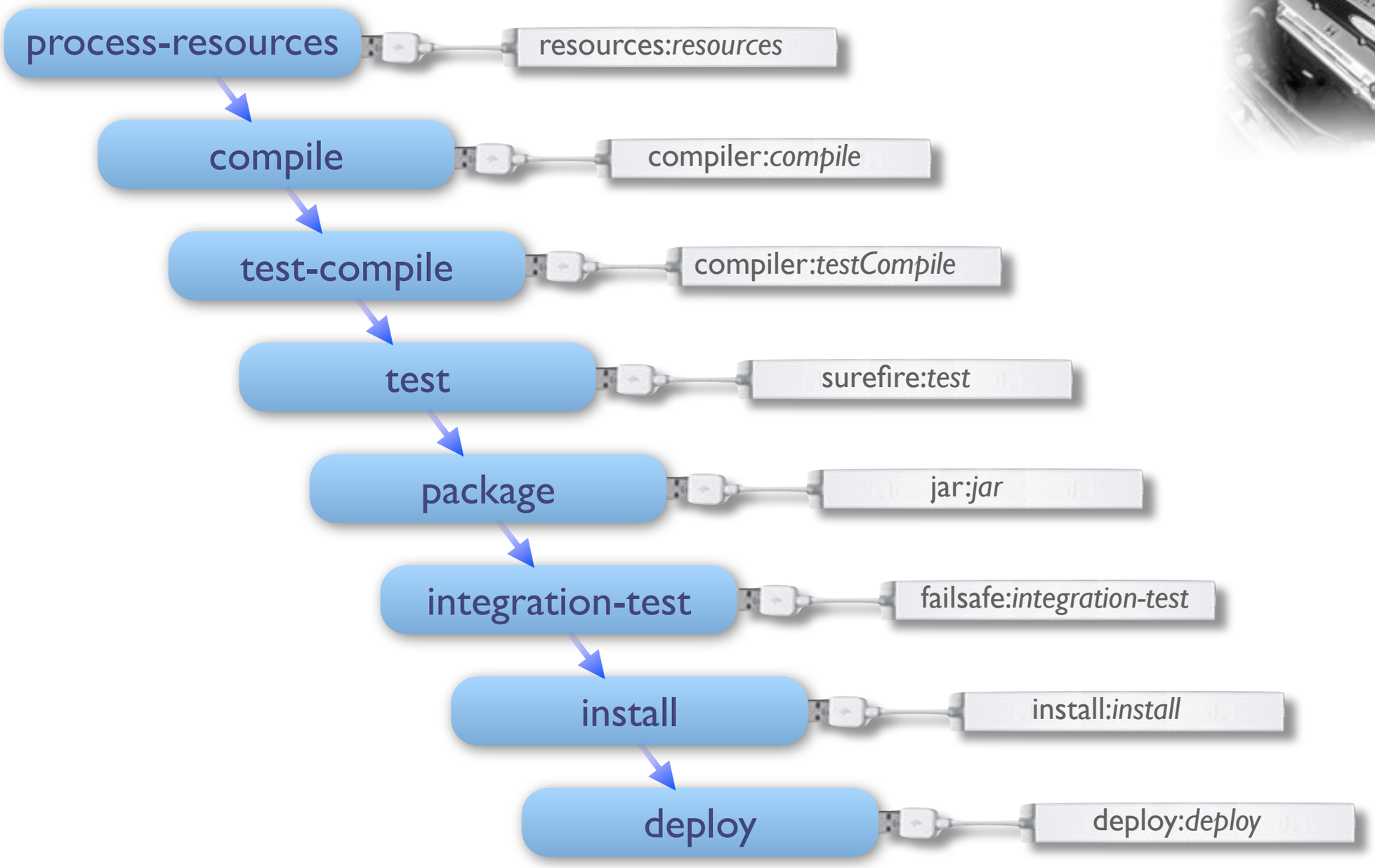
Plugins and Goals

Maven Plugins and Goals

- Maven Plugins
  - Most functionality in Maven is provided by plugins
  - A Maven plugin:
    - Perform a particular task or set of related tasks in the build life cycle
    - Contains a number of goals
  - Many plugins exist, e.g.
    - maven-compiler-plugin - used to compile classes into bytecode
    - maven-jar-plugin - used to create jar files
    - maven-surefire-plugin - used to execute tests and generate reports
    - maven-antrun-plugin - used to execute Ant scripts
    - maven-dependency-plugin - used to manage dependencies
  - Custom plugins can be created using Java, Ant, Groovy, Beanshell or Ruby

Maven Plugins and Goals

- Each life cycle phase is implemented using plugins



Maven – Introduction to the Build Lifecycle

http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

Lifecycle Reference

The following lists all build phases of the default, clean and site lifecycle, which are executed in the order given up to the point of the one specified.

Clean Lifecycle

pre-clean	executes processes needed prior to the actual project cleaning
clean	remove all files generated by the previous build
post-clean	executes processes needed to finalize the project cleaning

Default Lifecycle

validate	validate the project is correct and all necessary information is available.
initialize	initialize build state, e.g. set properties or create directories.
generate-sources	generate any source code for inclusion in compilation.
process-sources	process the source code, for example to filter any values.
generate-resources	generate resources for inclusion in the package.
process-resources	copy and process the resources into the destination directory, ready for packaging.
compile	compile the source code of the project.
process-classes	post-process the generated files from compilation, for example to do bytecode enhancement on Java classes.
generate-test-sources	generate any test source code for inclusion in compilation.
process-test-sources	process the test source code, for example to filter any values.
generate-test-resources	create resources for testing.
process-	copy and process the resources into the test destination directory.

http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

Maven Plugins and Goals

- Goals
  - A goal is a unit of work
  - A goal encapsulates functionality of a plugin
  - Goals can be executed individually or along with a sequence of other goals
  - Example goals:
    - compile - compiles a project's source code into bytecode
    - package - packages up a project's artifacts (default type is jar)
    - test - executes a project's tests and produces reports
    - install - actually a lifecycle phase; walks through many goals to fulfill a Maven lifecycle
  - Goals can be configured in the POM

Maven Plugins and Goals

- You can invoke a plugin in two ways:
  - Invoking a life cycle phase

```
$ mvn compile
```

Invoke the **compile** life cycle phase

  - Invoking a plugin goal directly

```
$ mvn jar:test-jar
```

Invoke the **test-jar** goal of the **jar** plugin

Fully Qualified Plugin Calls

- Instead of setting up a plugin group, can also fully qualify the plugin being called

```
$ mvn com.atlassian.maven.plugins:maven-clover2-plugin:2.6.1:clover
[INFO] Scanning for projects...
[INFO] ----------------------------------------------------------------------
[INFO] Building BNB Global Web Application
[INFO]    task-segment: [clover2:clover]
[INFO] ----------------------------------------------------------------------
```

Configuring PluginGroups

- Prefixes are looked up from two locations by default
  - /org/apache/maven/plugins
  - /org/codehaus/mojo
- You can add other groups in your `settings.xml` file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<settings>
   <pluginGroups>
      <pluginGroup>com.atlassian.maven.plugins</pluginGroup>
   </pluginGroups>
   ...
</settings>
```

maven-clover2-plugin

```
$ mvn clover2:clover
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'clover2'.
[INFO] ------------------------------------------------------------------------
[INFO] Building BNB Global Web Application
[INFO]    task-segment: [clover2:clover]
[INFO] ------------------------------------------------------------------------
```

key phases

process-resources Phase

- process-resources Phase
  - Takes action on files in ${project.basedir}/src/main/resources
    - Copies resources from ${project.basedir}/src/main/resources to ${project.basedir}/target/classes or the directory defined in ${project.build.outputDirectory}
    - Usually bound to the resources:resources goal
    - Filters can be applied along the way
      - Property processing, etc.
      - Used with build profiles, this can be used for producing platform-specific builds

compile Phase

- compile Phase
  - Used to compile source in ${project.basedir}/src/main/java to ${project.basedir}/target/classes
  - Bound to the compiler plugin's compile goal
  - Let's look at the maven-compiler-plugin customization again

The source and output directory locations can be customized, but it's not recommended. I.e., don't do it!

process-test-resources Phase

- process-test-resources Phase
  - Almost exactly the same as process-resources phase
  - Takes action on files in ${project.basedir}/src/test/resources
  - Copies resources from ${project.basedir}/src/test/resources to ${project.basedir}/target/test-classes
  - Usually bound to the resources:resources goal
  - Filters can be applied along the way

test-compile Phase

- test-compile Phase
  - Almost exactly the same as compile phase
  - Used to compile source in ${project.basedir}/src/test/java to ${project.basedir}/target/test-classes
  - Bound to the compiler plugin's compile:testCompile goal
  - Let's look at the maven-compiler-plugin customization again

test Phase

- test Phase
  - Looks for all classes whose name ends in Test.java to execute
  - Usually bound to the surefire:test plugin
  - Produces reports in ${project.basedir}/target/surefire-reports directory
  - It's common to skip test
- mvn install -Dmaven.test.skip=true
  - Skips compilation and execution of tests

install Phase

- install Phase
  - Copies the project artifacts to the local repository
  - Bound to the install:install goal

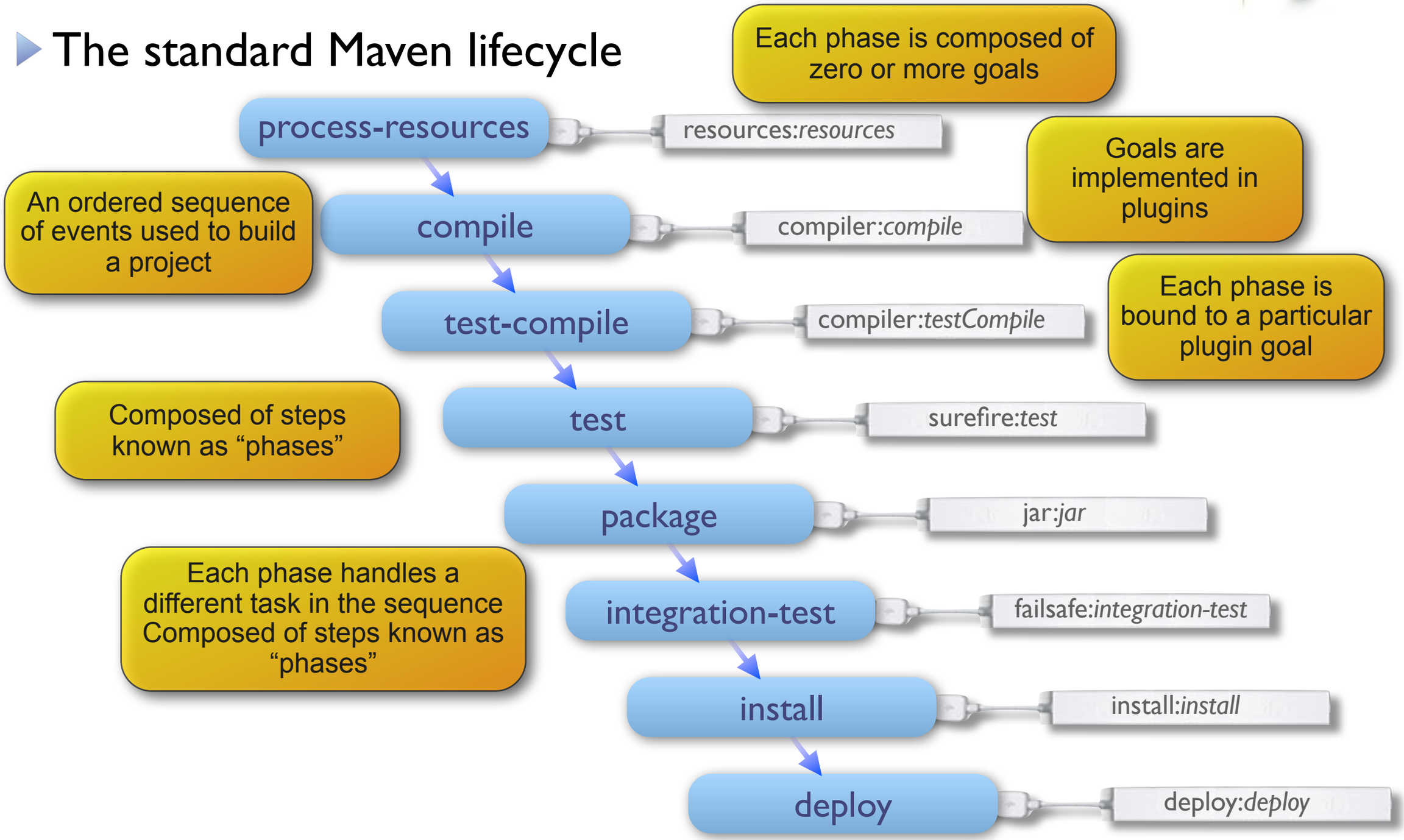
deploy Phase

- deploy Phase
  - Used to copy an artifact to a remote repository
  - Complexity can vary
    - Simple file copy
    - Reference to a Nexus URL
    - SCP using a public key
  - Settings are typically located in the local settings.xml file
    - Can involve user credentials
  - Bound to the deploy:deploy goal

customizing the lifecycle

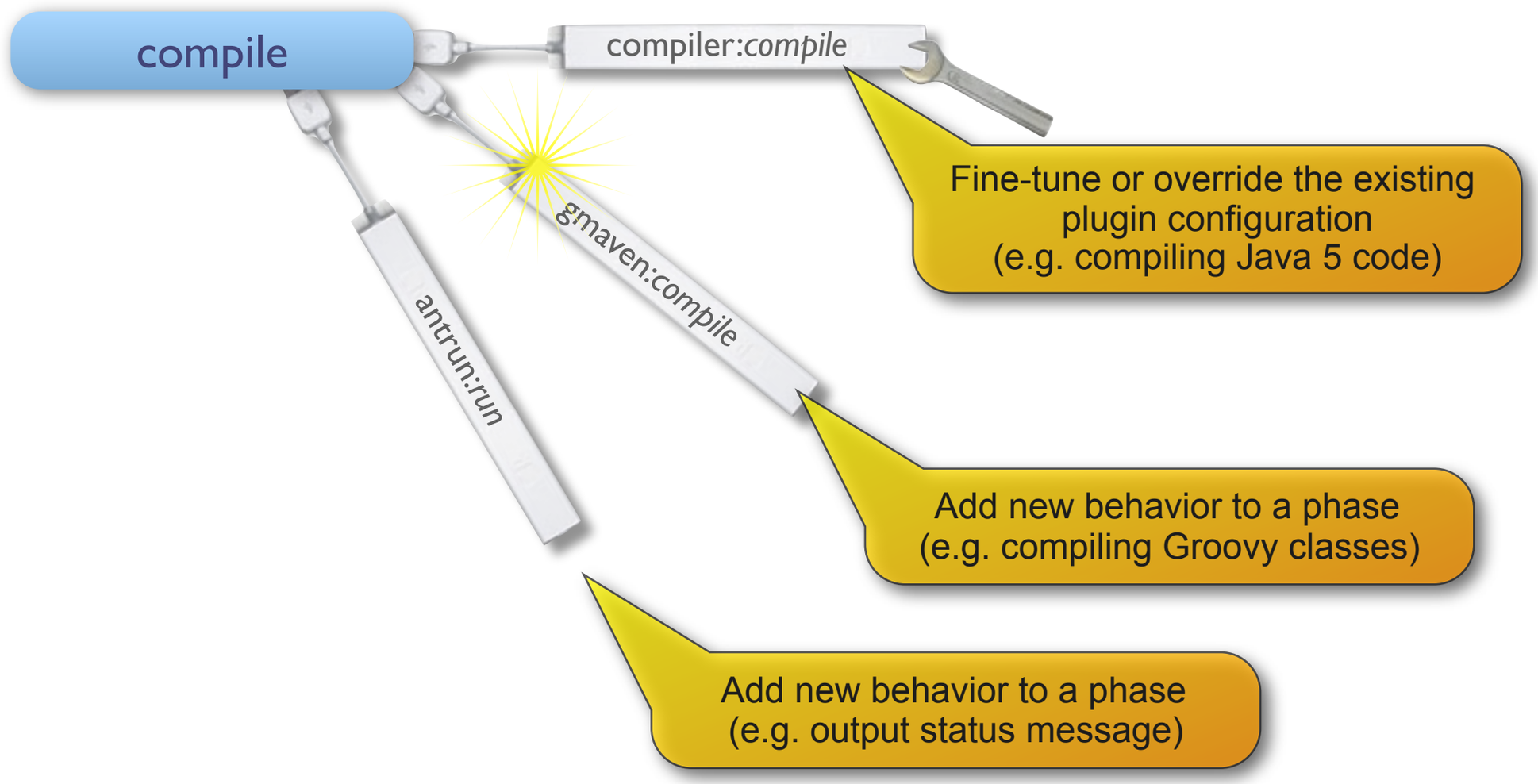
Customizing the lifecycle

- The standard Maven lifecycle



Customizing the lifecycle

- You can customize the lifecycle in two ways
  - Configure the standard plugin associated with a phase
  - Add a new plugin to add extra behavior to this phase



Configure the standard plugin associated with a phase

Customizing the lifecycle

- Customizing an existing configuration: Compiling Java 5 code
  - Java compilation is done by the **maven-compiler-plugin**
  - This plugin is configurable:
    - Compiles sources for projects
    - Supports multiple compilers
      - Jikes
      - CSharp
      - AspectJ
      - Eclipse
    - Supports compiler options
    - Support pinning the compiler to a particular source and target version

Customizing the lifecycle

- An example of customization - compiling for Java 5

```
<project>
  ...
    <build>
      <pluginManagement>
        <plugins>
          <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
              <source>1.5</source>
              <target>1.5</target>
            </configuration>
          </plugin>
        </plugins>
      <pluginManagement>
    </build>
  ...
</project>
```

Plugin configuration always goes in the <configuration> block

Here, compile for Java 5 code

Maven Compiler plugin – Using Non-Javac Compilers

http://maven.apache.org/plugins/maven-compiler-plugin/non-javac-compilers.html

Apache Maven Project
http:// maven.apache.org

Apache > Maven > Plugins > Maven Compiler Plugin

Last Published: 2010-01-02 | Version: 2.1

Overview

Introduction
Goals
Usage
FAQ

Examples

Compile Using A Different JDK
Compile Using -source and -target javac Options
Compile Using Memory Allocation Enhancements
Pass Compiler Arguments
Non-javac compilerIds

Project Documentation

Project Information
Project Reports

Maven Projects

Ant Tasks
Doxia
JXR
Maven 1.x
Maven 2
Plugins
SCM

Display a menu

Using Non-Javac Compilers

Contrary to this plugin's name, the Compiler Plugin does not compile the sources of your project by itself. To compile, the Compiler Plugin uses another class to compile them. The parameter `compilerId` determines which class will be used. Currently, the Compiler Plugin is bundled with the `javac` compiler artifact with artifactId `plexus-compiler-javac`, which can be seen as one of the dependencies declared inside the Compiler Plugin's POM.

Plexus have some other compiler Ids available under the groupId `org.codehaus.plexus`:

- `aspectj` with artifactId `plexus-compiler-aspectj`.
- `csharp` with artifactId `plexus-compiler-csharp`.
- `eclipse` with artifactId `plexus-compiler-eclipse`.
- `jikes` with artifactId `plexus-compiler-jikes`.

Using The Other Plexus Compilers

To use any of the non-javac compilers, you need to make the artifact available in the classpath when the Compiler Plugin runs. This is possible by adding the dependency when declaring the plugin in your project's pom.xml. The example below shows how to use the `csharp` compiler:

```
<project>
  [...]
    [...]
```

http://maven.apache.org/plugins/maven-compiler-plugin/non-javac-compilers.html

Add a plugin to add extra behavior to a phase

Customizing the lifecycle

- An example of customization - echoing the current project each time we compile

```
<project>
  ...
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-antrun-plugin</artifactId>
          <executions>
            <execution>
              <id>echodir</id>
              <goals>
                <goal>run</goal>
              </goals>
              <phase>compile</phase>
              <configuration>
                <tasks>
                  <echo>ECHO1 Build Dir: ${project.build.directory}</echo>
                </tasks>
              </configuration>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  ...
</project>
```

During the *compile* phase...

Output a message to the console

Customizing the lifecycle

- Adding new behavior - executing a Groovy script
  - Maven and Groovy integrate well with the gmaven-plugin plugin
  - In this example we want to **execute** a utility Groovy script during the compile phase:

```
<project>
  ...
    <plugin>
      <groupId>org.codehaus.groovy.maven</groupId>
      <artifactId>gmaven-plugin</artifactId>
      <executions>
        <execution>
          <phase>compile</phase>
          <goals>
            <goal>execute</goal>
          </goals>
          <configuration>
            <source>${project.basedir}/src/main/script/myscript.groovy</source>
          </configuration>
        </execution>
      </executions>
    </plugin>
  ...
</project>
```

Add the gmaven-plugin

During the *compile* phase...

Call the plugin's *execute* goal

Plugin-specific configuration

packaging

Package-Specific Lifecycle Bindings

- There are specific lifecycle bindings for the following package types:
  - EAR
  - EJB
  - JAR
  - Maven Plugin
  - POM
  - WAR

Package Lifecycle Bindings

- The pom package type
  - A project with packaging pom has a different set of default goals from a project with a packaging of jar or ejb

Lifecycle Phase	Goal
package	site:attach-descriptor
install	install:install
deploy	deploy:deploy

Package Lifecycle Bindings

- The jar package type
  - A project with packaging jar has a different set of default goals from a project with a packaging of war or ear

Lifecycle Phase	Goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar
install	install:install
deploy	deploy:deploy

Package Lifecycle Bindings

- The war package type
  - A project with packaging war has a different set of default goals from a project with a packaging of jar or ear

Lifecycle Phase	Goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	war:war
install	install:install
deploy	deploy:deploy

Package Lifecycle Bindings

- The ear package type
  - A project with packaging ear has a different set of default goals from a project with a packaging of jar or war

Lifecycle Phase	Goal
generate-resources	ear:generate-application-xml
process-resources	resources:resources
package	ear:ear
install	install:install
deploy	deploy:deploy

3 lifecycles

The Maven lifecycles

- Maven provides the following lifecycles:
  - ***clean*** lifecycle - project cleaning
  - ***default*** lifecycle - project deployment
  - ***site*** lifecycle - project site creation

Clean Lifecycle Phases

- The ***Clean*** Lifecycle phases:
  - **pre-clean** - a hook to do any unlocking or checking of files prior to actually cleaning. Rarely used.
  - **clean** - clean all generated, compiled, and test-generated artifacts
  - **post-clean** - a hook for any post-cleaning steps, such as restoring file locks or permissions. Rarely used.

Default Lifecycle Phases

- The ***Default*** Lifecycle phases (an extract):
  - **validate** - validate the project is correct and all necessary information is available
  - **compile** - compile the source code of the project
  - **test** - test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
  - **package** - take the compiled code and package it in its distributable format, such as a JAR.
  - **integration-test** - process and deploy the package if necessary into an environment where integration tests can be run
  - **verify** - run any checks to verify the package is valid and meets quality criteria
  - **install** - install the package into the local repository, for use as a dependency in other projects locally
  - **deploy** - done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects

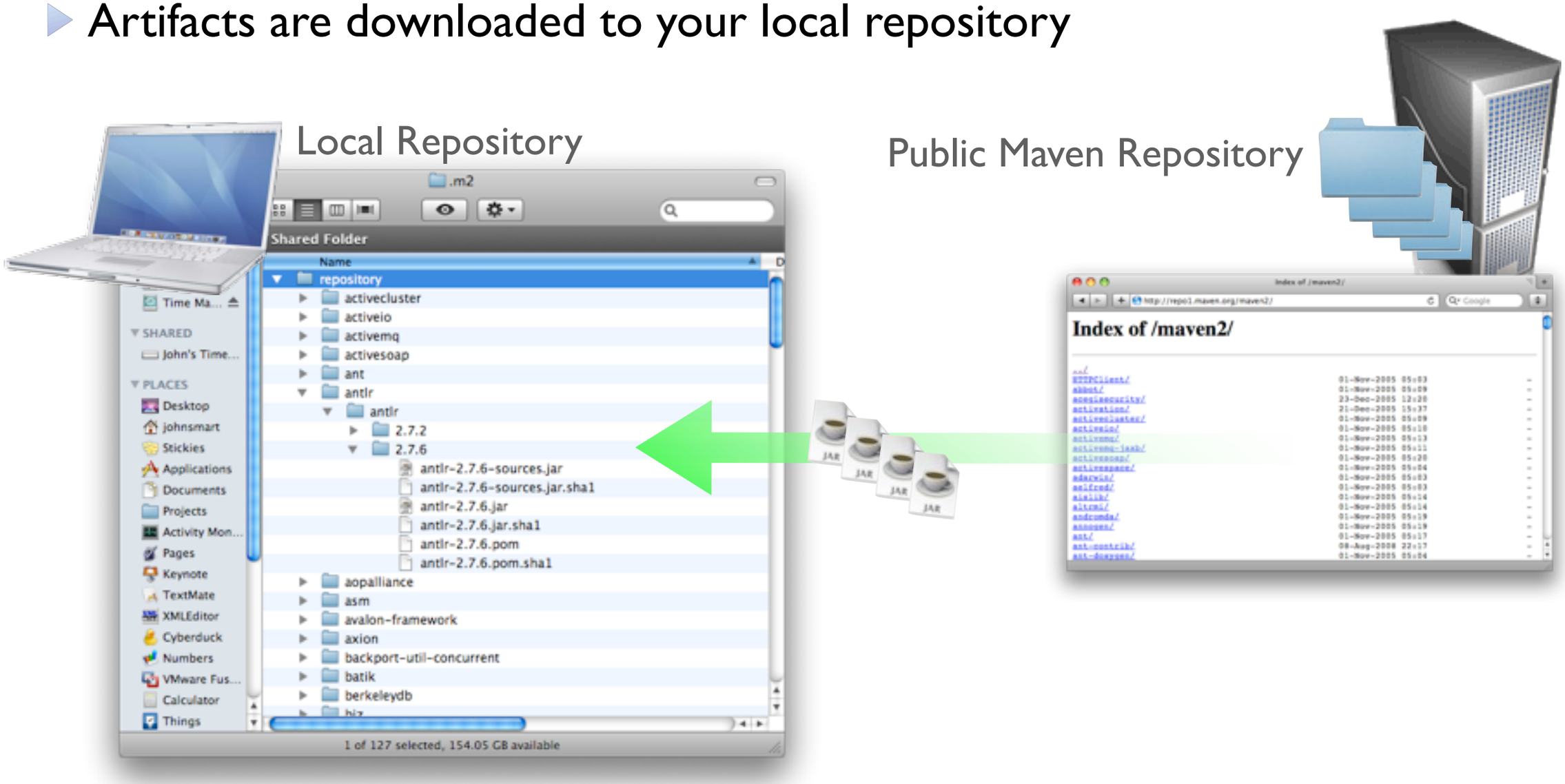
Site Lifecycle Phases

- ***Site*** Lifecycle Phases:
  - **pre-site -** executes processes needed prior to the actual project site generation
  - **site -** generates the project's site documentation
  - **post-site -** executes processes needed to finalize the site generation, and to prepare for site deployment
  - **site-deploy -** deploys the generated site documentation to the specified web server

Repositories

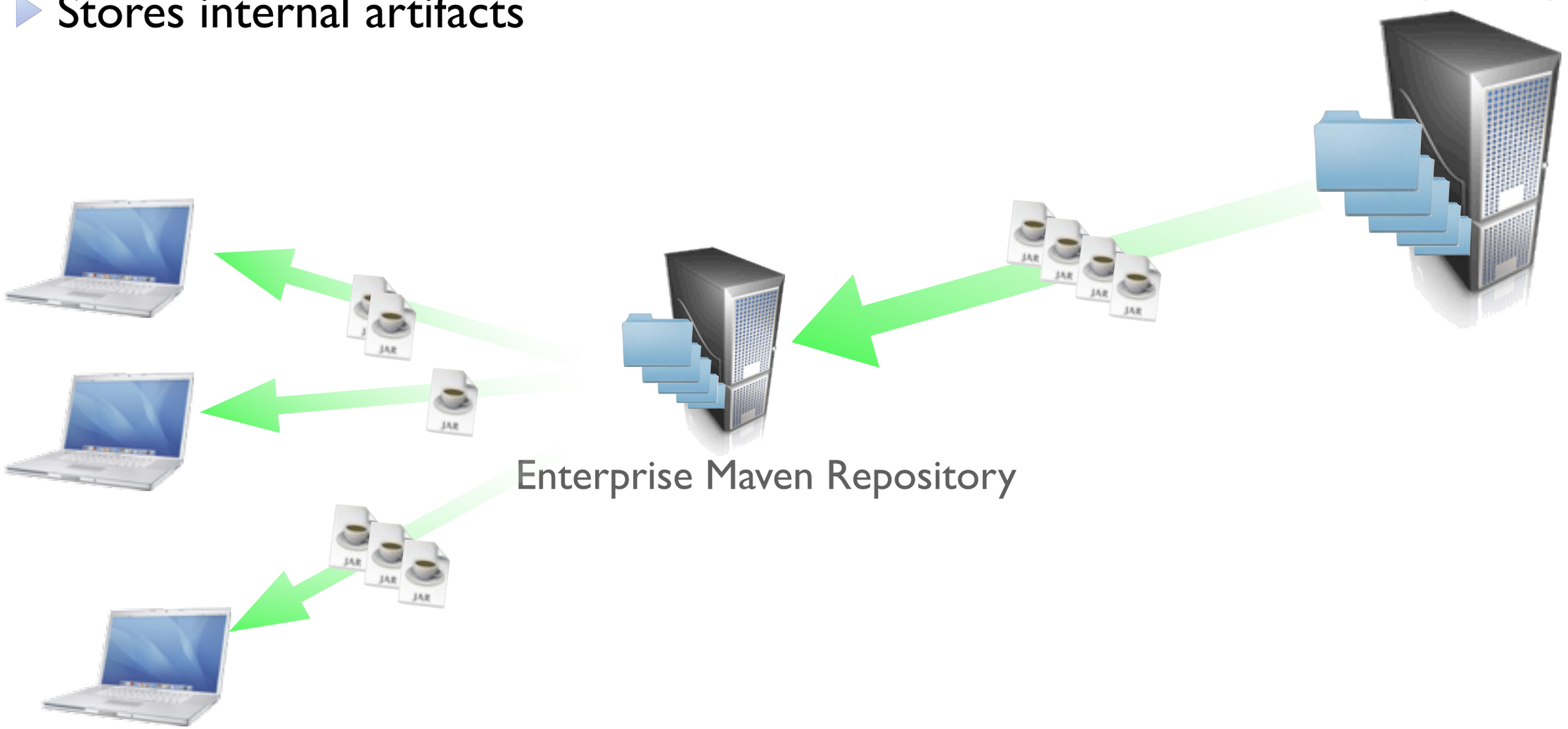
Maven Repositories

- So where do these artifacts come from?
  - Public Maven Repositories - store public O/S artifacts
  - Artifacts are downloaded to your local repository



Maven Repositories

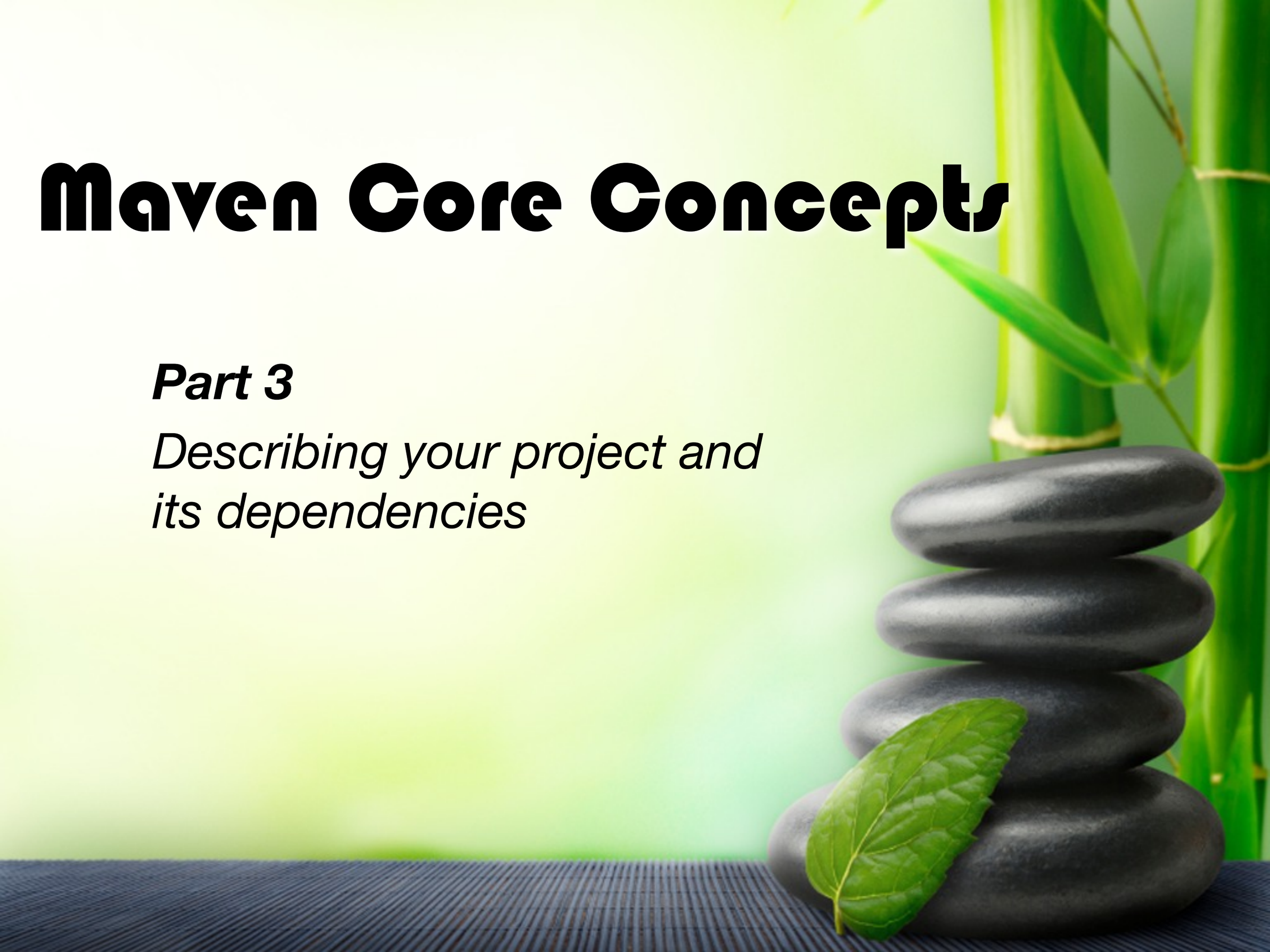
- Enterprise Maven Repositories
  - Acts as a proxy/cache to public repositories
  - Stores internal artifacts



Maven Core Concepts

Part 3

Describing your project and its dependencies



Archetypes

Maven Archetypes

- Project archetypes
  - Standard templates for different types of projects
  - Provide a convenient way to create a skeleton project structure
  - Many templates are available
    - Web applications
    - Spring
    - Struts
    - JSF
    - Grails
    - ...
  - You can also make your own

Maven Archetypes

- Project archetypes
  - You can generate a skeleton Maven project using the Archetype plugin

```
$ mvn archetype:generate
[INFO] Scanning for projects...
...
[INFO]
------------------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO]    task-segment: [archetype:generate] (aggregator-style)
[INFO]
------------------------------------------------------------------------
[INFO] Preparing archetype:generate
...
Choose archetype:
1: internal -> appfuse-basic-jsf (AppFuse archetype for creating a web
application with Hibernate, Spring and JSF)
...
15: internal -> maven-archetype-quickstart ()
...
40: internal -> gmaven-archetype-basic (Groovy basic archetype)
41: internal -> gmaven-archetype-mojo (Groovy mojo archetype)
Choose a number:
(1/2/3/4/5/6/7/8/9/10/11/12/13/14/15/16/17/18/19/20/21/22/23/24/25/26/27/
28/29/30/31/32/33/34/35/36/37/38/39/40/41) 15: :
```

Create a new skeleton project using a command-line wizard

Choose the archetype you want to use (over 40 to pick from!)

Maven Archetypes

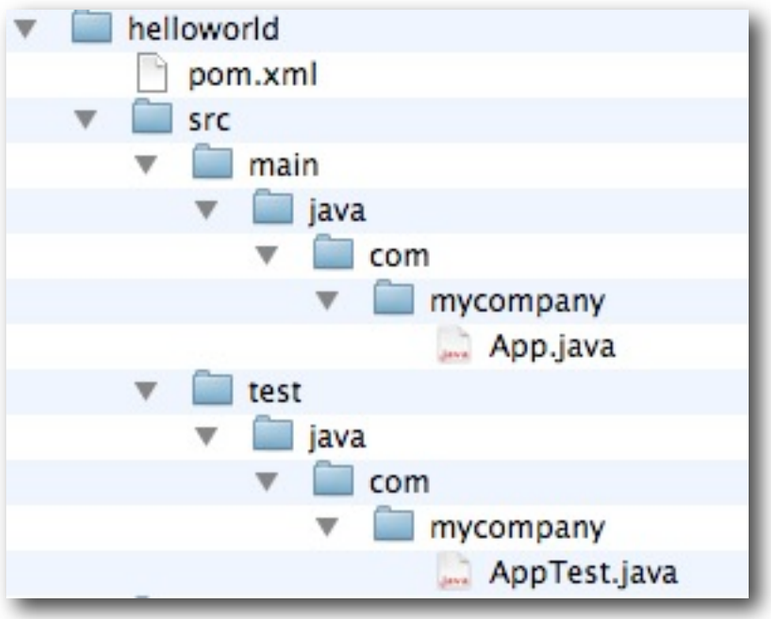
- Project archetypes
  - Maven will prompt you for the project co-ordinates

```
...
40: internal -> gmaven-archetype-basic (Groovy basic archetype)
41: internal -> gmaven-archetype-mojo (Groovy mojo archetype)
Choose a number:
(1/2/3/4/5/6/7/8/9/10/11/12/13/14/15/16/17/18/19/20/21/22/23/24/25/26/27/
28/29/30/31/32/33/34/35/36/37/38/39/40/41) 15: :
Define value for groupId: : com.mycompany
Define value for artifactId: : helloworld
Define value for version:  1.0-SNAPSHOT: :
Define value for package:  com.mycompany: :
```

Enter your groupId, artifactId, initial version number and base package

Maven Archetypes

- Project archetypes
  - Maven will create a simple but working project



  - The Maven life cycle goals will work out of the box

```
$ mvn compile
$ mvn test
$ mvn package
$ mvn install
```

Maven Archetypes

- Project archetypes
  - You can build and test this project using the standard life cycle phases

Run **mvn package**

Compile the code

Run the unit tests

Build a JAR file

```
$ mvn package
...
[INFO] [compiler:compile]
[INFO] Compiling 1 source file to /Users/someuser/helloworld/target/classes
...
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.mycompany.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.027 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] [jar:jar]
[INFO] Building jar: /Users/someuser/helloworld/target/helloworld-1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3 seconds
[INFO] Finished at: Wed Mar 04 21:00:06 NZDT 2009
[INFO] Final Memory: 8M/16M
[INFO] ------------------------------------------------------------------------
```

Manual Archetype Instantiation

- To bypass the interactive prompt:

```
mvn archetype:generate
 -DgroupId=<your group>
 -DartifactId=<your artifact>
 -DarchetypeArtifactId=<wanted artifact>
 -DarchetypeGroupId=<wanted artifact group>
```

Alternate Archetype Catalog

- To use an alternate catalog other than central:

```
mvn archetype:generate
 -DarchetypeCatalog=<repo URL>

mvn archetype:generate
 -DarchetypeCatalog=http://maven-blazeds-
spring-archetype.googlecode.com/svn/maven2
```

Questions

- What about other archetype catalogs?
  - Archetypes List
    - http://docs.codehaus.org/display/MAVENUSER/Archetypes+List
  - Stack Overflow Question about Catalogs
    - http://bit.ly/cqaQ1x
- Why aren't all archetypes in central?

labs

- Lab 4 - Your first Maven project via command line archetypes

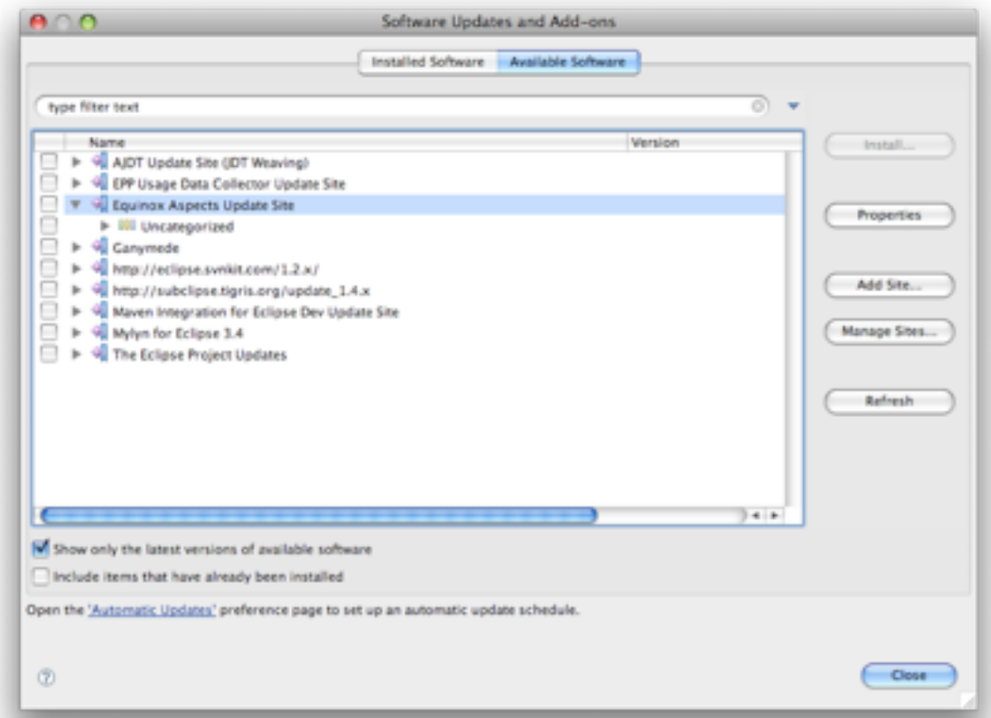
M2Eclipse

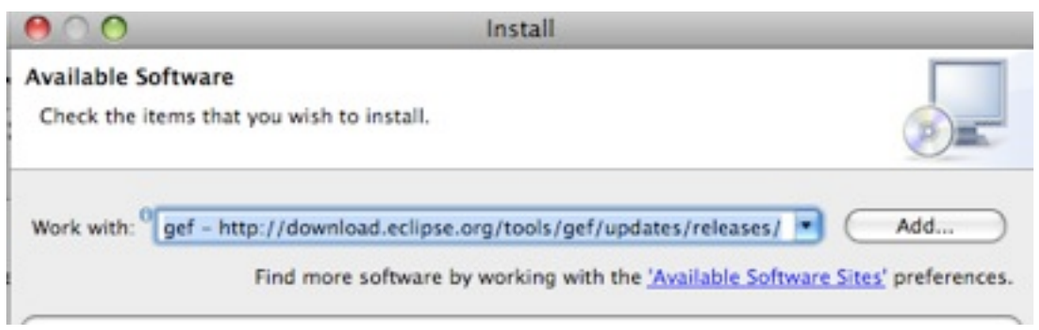
Maven in Eclipse

- The *m2eclipse* plugin currently provides the best IDE support for Maven
  - Creating and importing Maven projects
  - Dependency management and integration with the Eclipse classpath
  - Automatic dependency downloads and updates
  - Artifact Javadoc and source resolution
  - Creating projects with Maven Archetypes
  - Browsing and searching remote Maven repositories
  - POM management with automatic update to dependency list
  - Materializing a project from a Maven POM
  - Checking out a Maven project from several SCM repositories
  - Form-based POM Editor
  - Graphical Display of Dependency Graphs
  - GUI Presentation of Dependency Tree and Resolved Dependencies

Maven in Eclipse

- Installing the *m2eclipse* plugin:
  - Core m2eclipse Plugin Update Site
    - http://m2eclipse.sonatype.org/sites/m2e/
  - Extensions Plugin Upate Site
    - http://m2eclipse.sonatype.org/sites/m2e-extras/



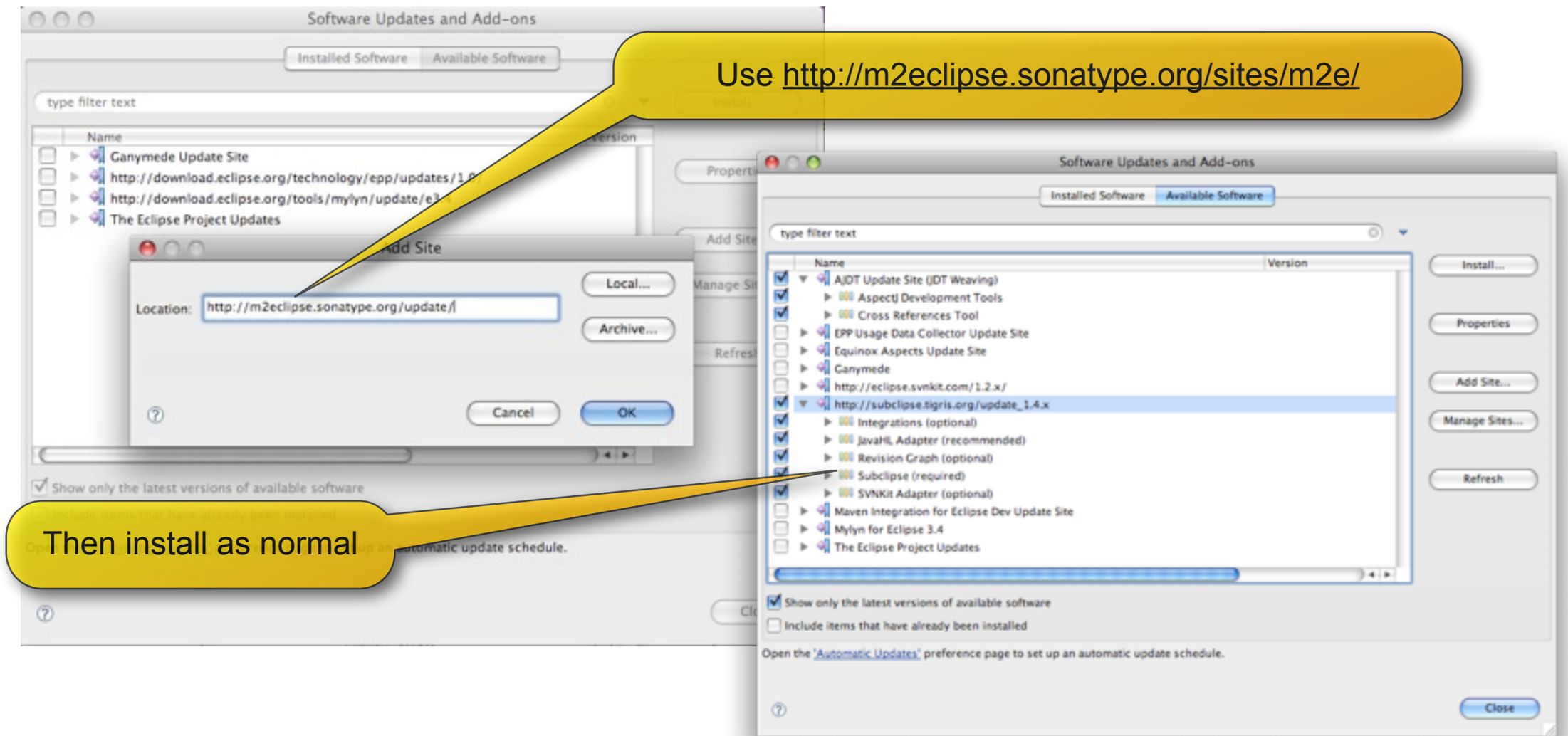


Maven in Eclipse

- Installing the *m2eclipse* plugin in Eclipse 3.5 or greater
  - You can decide later on installing the optional modules
    - Maven SCM Integration
    - Maven SCM handler for Team/CVS
    - Maven SCM handler for Subclipse
    - Maven issue tracking configurator for Mylyn 3.x
    - Maven Integration for WTP
    - M2Eclipse Extensions Development Support

Maven in Eclipse

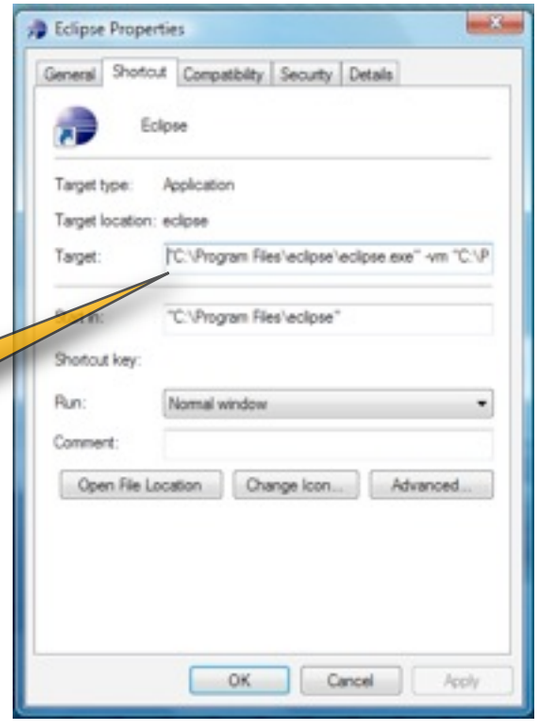
- Installing the *m2eclipse* plugin:
  - Install the m2eclipse plugin from the Sonatype update site



Maven in Eclipse

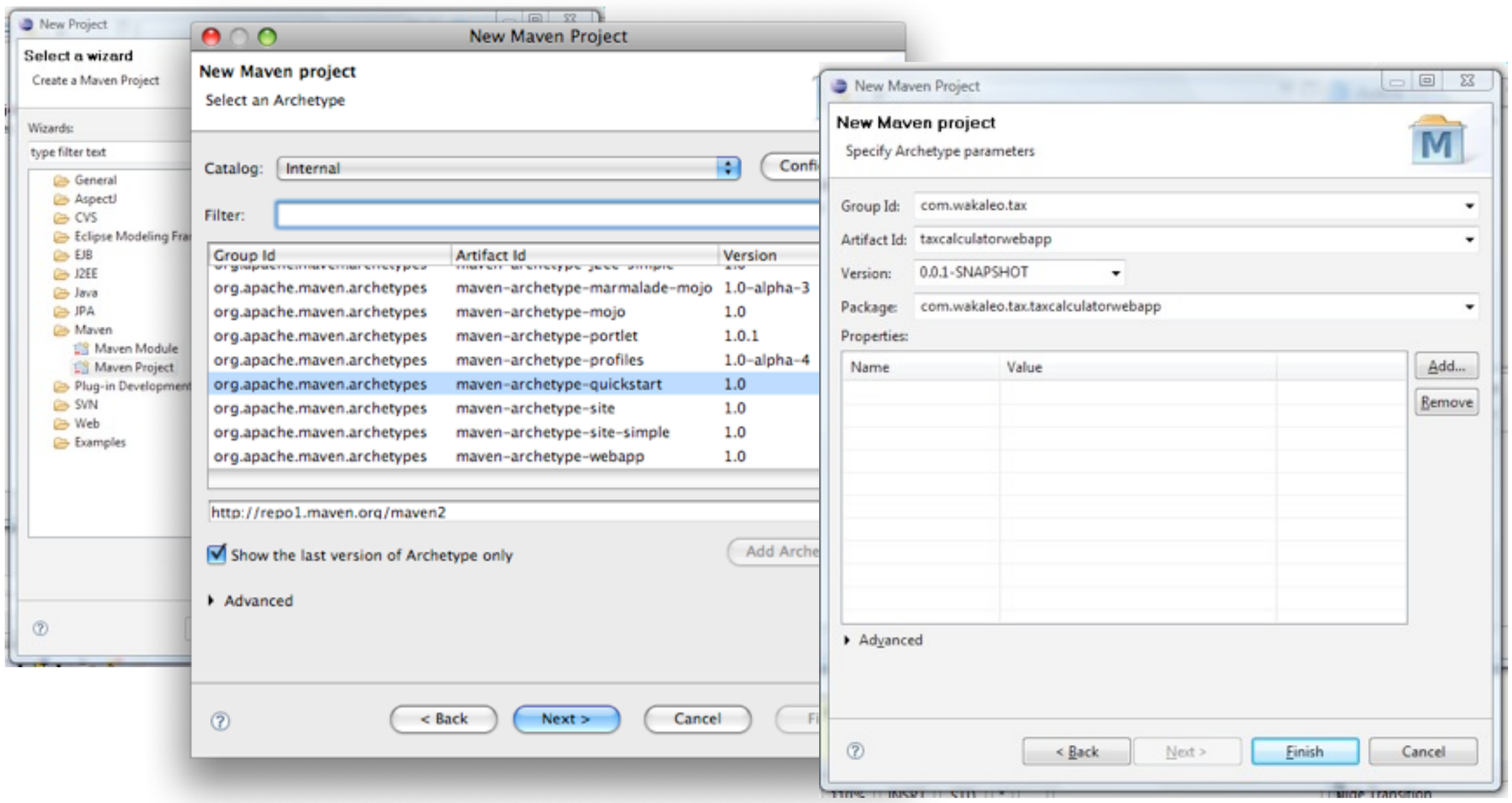
- Always run Eclipse using a JDK
  - The m2eclipse plugin expects a JDK.
  - Eclipse doesn't always run with a JDK by default
  - Use the Compiler Preferences from within Eclipse to add a JDK
  - Or use the **-vm** option to point to your JDK in Windows

```
"C:\Program Files\eclipse\eclipse.exe"
-vm "C:\Program Files\Java\jdk1.6.0_05\bin"
```



Maven in Eclipse

- Creating a Maven Project in Eclipse
  - With m2eclipse, you can choose your archetype from within Eclipse



Questions

- Why Eclipse over the command line?
- Is one better than the other?
- What about Q4E, IAM?

labs

- Lab 6 - Your first Maven project using m2eclipse
  - Uses the archetype process
  - Leverages the quickstart archetype
  - A quick “proper” Maven project directory structure to build upon

Development Infrastructure Data

Development Infrastructure Details

- Development Infrastructure Details

```
<project ...>
  ...
  <name>myapp</name>
  <url>http://www.mycompany.com</url>
  <description>My Comany killer app.</description>
  <scm>
    <connection>scm:svn:http://svn.mycompany.com/svn/myapp...</connection>
    <developerConnection>
      scm:svn:http://svn.mycompany.com/svn/myapp...
    </developerConnection>
    <url>
      http://devserver.mycompany.com/trac/myapp/browser...
    </url>
  </scm>
  <issueManagement>
    <system>trac</system>
    <url>http://devserver.mycompany.com/trac/myapp</url>
  </issueManagement>
  <inceptionYear>2009</inceptionYear>
  ...
</project>
```

Project name and description

SCM details
(also used to automate releases)

Issue tracking details

Development Infrastructure Details

- List the development team
  - Very useful for large or distributed teams

```
<project ...>
  ...
  <developers>
    <developer>
      <id>jsmith</id>
      <name>John Smith</name>
      <email>jsmith@example.com</email>
      <url>http://blogs.example.com/jsmith</url>
      <organization>ESF</organization>
      <organizationUrl>http://www.example.com</organizationUrl>
      <roles>
        <role>Architect</role>
        <role>Developer</role>
      </roles>
      <timezone>-5</timezone>
      <properties>
        <gpg-key>jsmith@personal.com</gpg-key>
      </properties>
    </developer>
  </developers>
  ...
</project>
```

Development Infrastructure Details

- Development Infrastructure Details
  - Used to generate the Maven project web site

```
<project ...>
  ...
  <name>myapp</name>
  <url>http://www.mycompany.com</url>
  <description>My Comany killer app.</description>
  <scm>
    <connection>scm:svn:http://svn.mycompany.com/svn/myapp...</connection>
    <developerConnection>
      scm:svn:http://
    </developerConnec
    <url>
      http://devserve
    </url>
  </scm>
  <issueManagement>
    <system>trac</sys
    <url>http://devse
  </issueManagement>
  <inceptionYear>2009
  ...
</project>
```

Project Information

This document provides an overview of the various doc
content is automatically generated by Maven on bel

Overview

Document	Description
Continuous Integration	This is a link to the definitions o regular basis.
Dependencies	This document lists the projects
Issue Tracking	This is a link to the issue manag created and queried using this li
Mailing Lists	This document provides subscri
Project License	This is a link to the definitions o
Project Summary	This document lists other relate
Project Team	This document provides informa contributed to the project in on
Source Repository	This is a link to the online sourc

Overview

This project uses Subversion to manage its source code. Instructions on Subversion use can be found at http://svnbook.red-bean.com/ .

Web Access

The following is a link to the online source repository.

```
http://svn.mycompany.com/trac/accounting-core/browser/tax-calculator...
```

Anonymous access

The source can be checked out anonymously from SVN with this command:

```
$ svn checkout http://svn.mycompany.com/svn/tax-calculator... modelplanes
```

Developer access

Refer to the documentation of the SCM used for more information about developer checked out. The connection url is:

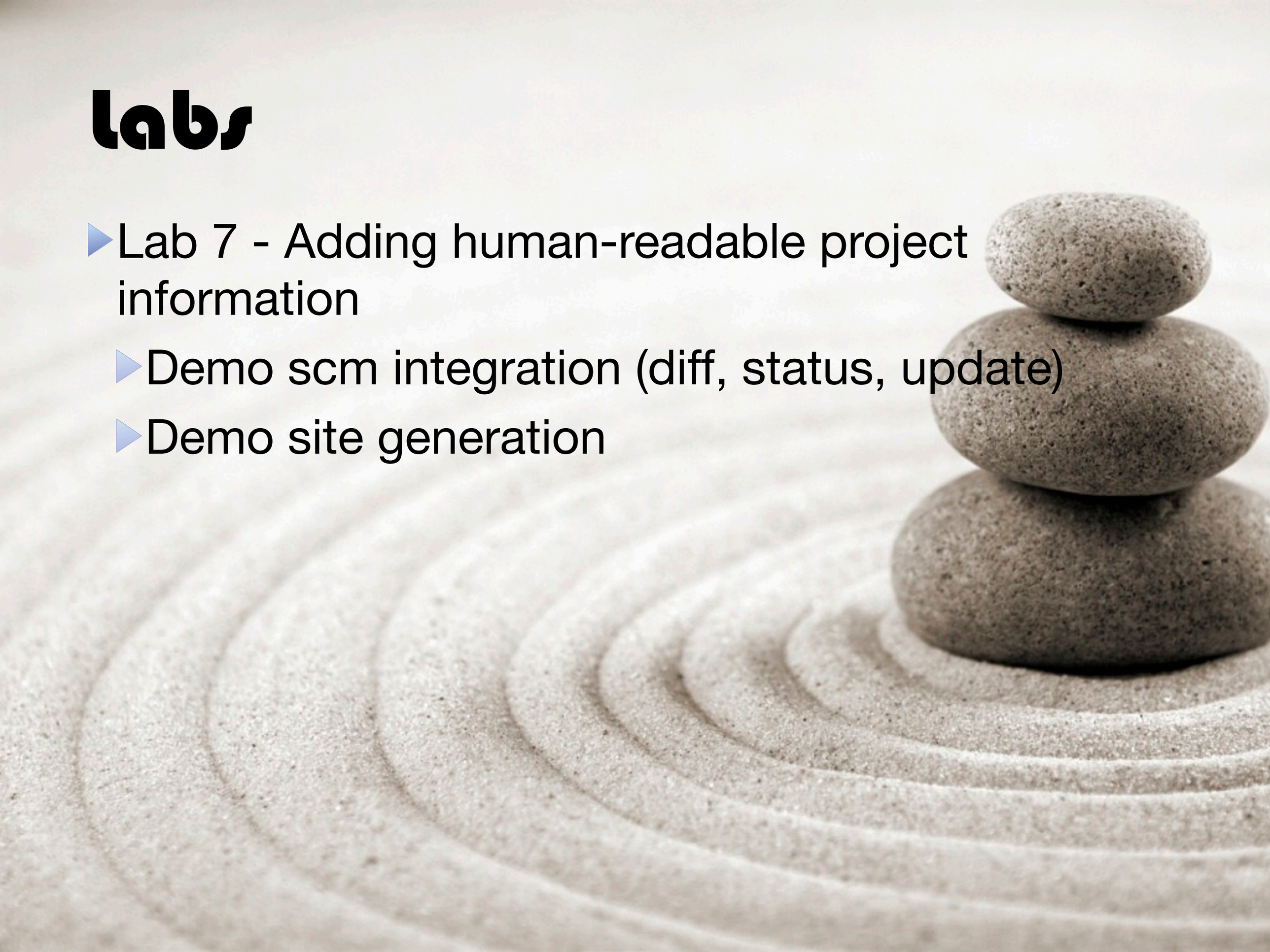
```
svn:hhttp://svn.mycompany.com/svn/tax-calculator...
```

Questions

- Why "waste" the time to put in these development specific fields?
- Does anyone actually look at these?
- Take a look at the Maven web site for examples
- Why would I duplicate the corporate address book?
- Is linking to a bug-tracking system frivolous?

labs

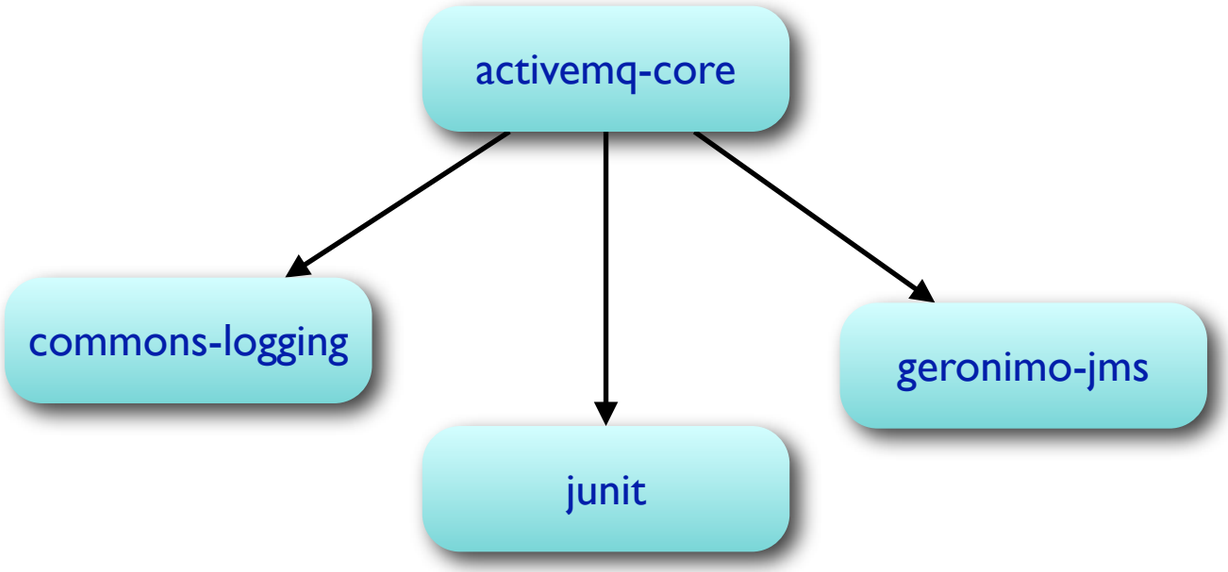
- Lab 7 - Adding human-readable project information
  - Demo scm integration (diff, status, update)
  - Demo site generation



Dependencies

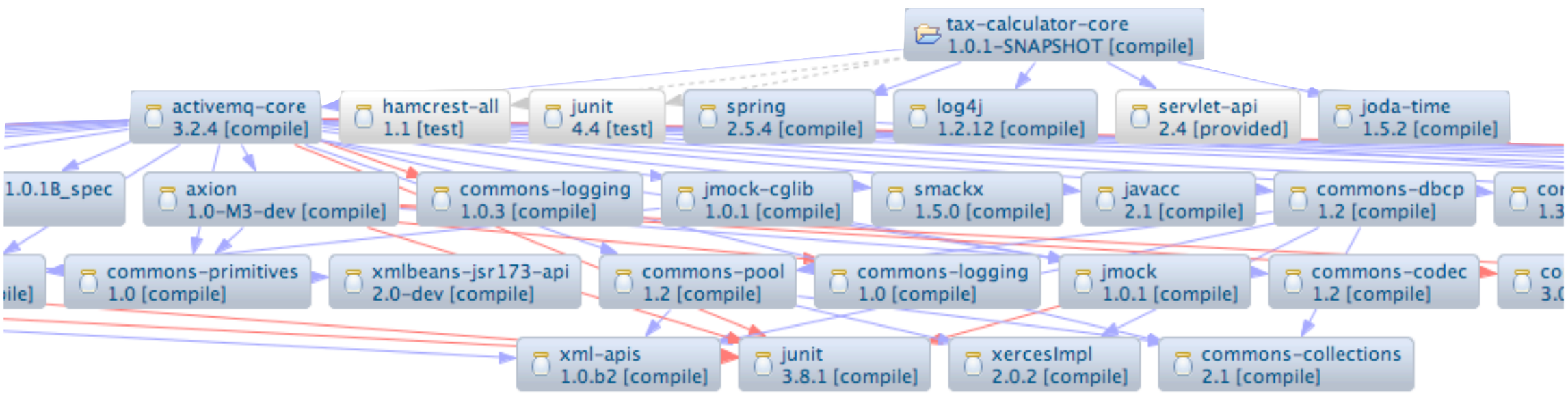
Dependencies

- What are dependencies?
  - Artifacts on which a project relies to compile, test, run, etc.
  - Example
    - activemq-core dependencies
      - commons-logging
      - junit
      - geronimo-jms



Dependency Management

- What are dependencies?
  - Your project needs libraries (artifacts) to compile, test, and run
    - Third-party libraries or frameworks
    - Enterprise shared libraries
    - Other modules in a multi-module project
  - These libraries also need libraries of their own
  - And so on...

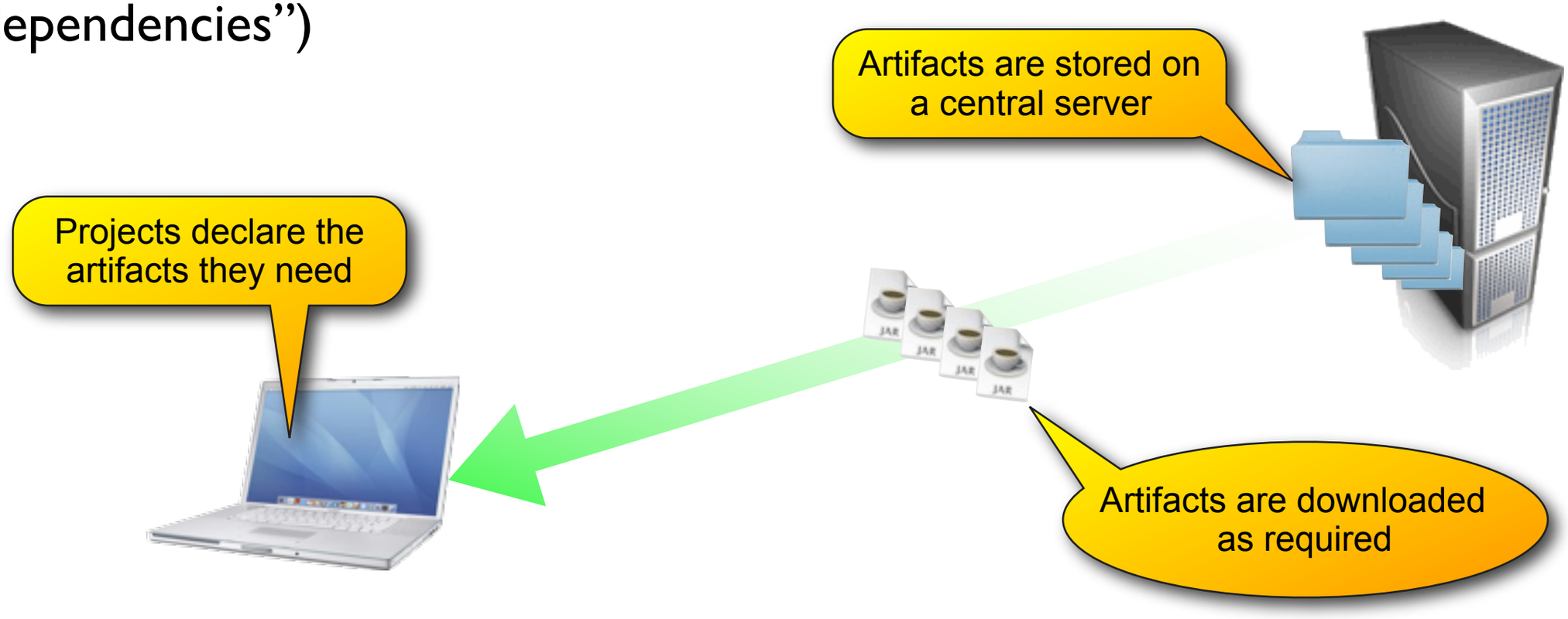


Dependency Management

- Traditional Dependency Management
  - Store all your JAR files in a `lib` directory
    - Hard to work out what JARs you need
    - Each project has its own set of JAR files
    - Duplicated JAR files
    - Hard to keep track of versions
    - Overloads the source code repository

Dependency Management

- Declarative Dependency Management - a better solution
  - Versions of artifacts are stored on a central server
  - Each project "declares" what artifacts it requires, and what versions
  - Artifacts are not stored with the project itself
  - Secondary dependencies are also downloaded automatically ("transitive dependencies")



Dependencies

- Project dependencies are defined in the POM:
  - Defined using the Maven artifact coordinates
  - Defined in the <dependencies> section
  - Finer details are specified here

```
<dependencies>
  <dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
    <version>1.1</version>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.specs</groupId>
    <artifactId>geronimo-jms_1.1_spec</artifactId>
    <version>1.1.1</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.4</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Dependency on commons-logging 1.1

geronimo-jms will be provided by the application server

JUnit 4.4 is only required to compile and execute the tests

Dependency Scope

- Different dependencies have different uses:
  - The commons-logging dependency is a **compile** dependency:
    - The project depends on this artifact for compilation, testing and runtime
  - The geronimo-jms_1.1_spec dependency is a **provided** dependency:
    - The project needs this artifact for compilation and testing; at deployment runtime the container will supply it
  - The junit dependency is a test dependency is a **test** dependency:
    - This projects needs this artifact for testCompile and test phases
- Why scoping?
  - Scoping helps to define the various classpaths for different phases
  - Scoping also affects the packaging phase
    - Whether a dependency is included in the artifact package

Dependency Scope

- Dependency scopes
  - Dependencies can have different scopes:
    - compile
    - provided
    - test
    - runtime
    - system
    - imported

Dependency Scope

- Compile scope
  - The default scope
  - Available in all classpaths
  - Bundled with the packaged application
  - Examples: Hibernate, Spring,...

```
<dependencies>
    ...
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring</artifactId>
        <version>2.5.3</version>
    </dependency>
    ...
</dependencies>
```

Dependency Scope

- Provided scope
  - Supplied by the JDK or a container at runtime
  - Include in the compile and test classpaths
  - Don't include it in the final package
  - Examples: ServletAPI,...

```
<dependencies>
    ...
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>2.4</version>
        <scope>provided</scope>
    </dependency>
    ...
</dependencies>
```

Dependency Scope

- Test scope
  - Not needed for normal use of the application
  - Included in the test compilation and execution classpaths
  - Not bundled with the packaged application
  - Examples: JUnit, TestNG,...

```
<dependencies>
    ...
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.4</version>
        <scope>test</scope>
    </dependency>
    ...
</dependencies>
```

Dependency Scope

- Runtime scope
  - Required to test and execute the application
  - Not required for compilation
  - Bundled with the packaged application
  - Examples: Oracle JDBC

```
<dependencies>
    ...
    <dependency>
        <groupId>oracle</groupId>
        <artifactId>ojdbc14</artifactId>
        <version>10.2.0.2.0</version>
        <scope>runtime</scope>
    </dependency>
    ...
</dependencies>
```

Dependency Scope

- System scope
  - Similar to the provided scope
  - Provide the artifact explicitly as a file path
  - Rarely used (better to use the repositories)

```
<dependencies>
    ...
   <dependency>
      <groupId>com.sun</groupId>
      <artifactId>tools</artifactId>
      <version>1.4.2</version>
      <scope>system</scope>
      <systemPath>${java.home}/lib/tools.jar</systemPath>
    </dependency>
    ...
</dependencies>
```

Dependency Scope

- Import scope
  - Only works with Maven 2.0.9 onwards
  - Import dependencies in the `<dependencyManagement>` section of another project
  - Lets you import dependencyManagement info from several sources
  - Only used by project type pom (e.g. parent projects)

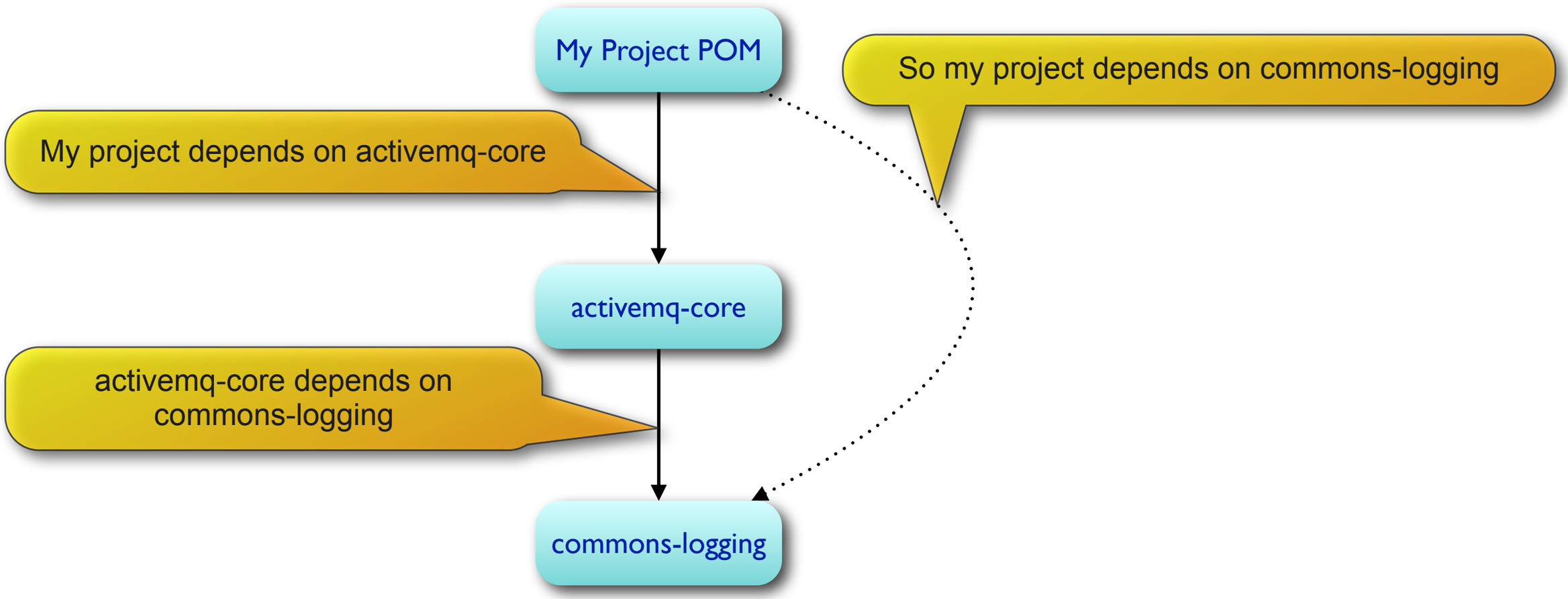
```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>a-parent</artifactId>
  <version>1.0.0</version>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.2</version>
      </dependency>
      ...
    <dependencies>
  </dependencyManagement>
```

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>b-parent</artifactId>
  <version>1.0.0</version>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.sonatype.mavenbook</groupId>
        <artifactId>a-parent</artifactId>
        <version>1.0.0</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
      ...
    <dependencies>
  </dependencyManagement>
```

Transitive Dependencies

Transitive Dependencies

- Dependencies of a dependency
  - Golden Rule: *"The dependency of my dependency is my dependency. (Mostly)"*



Transitive Dependencies

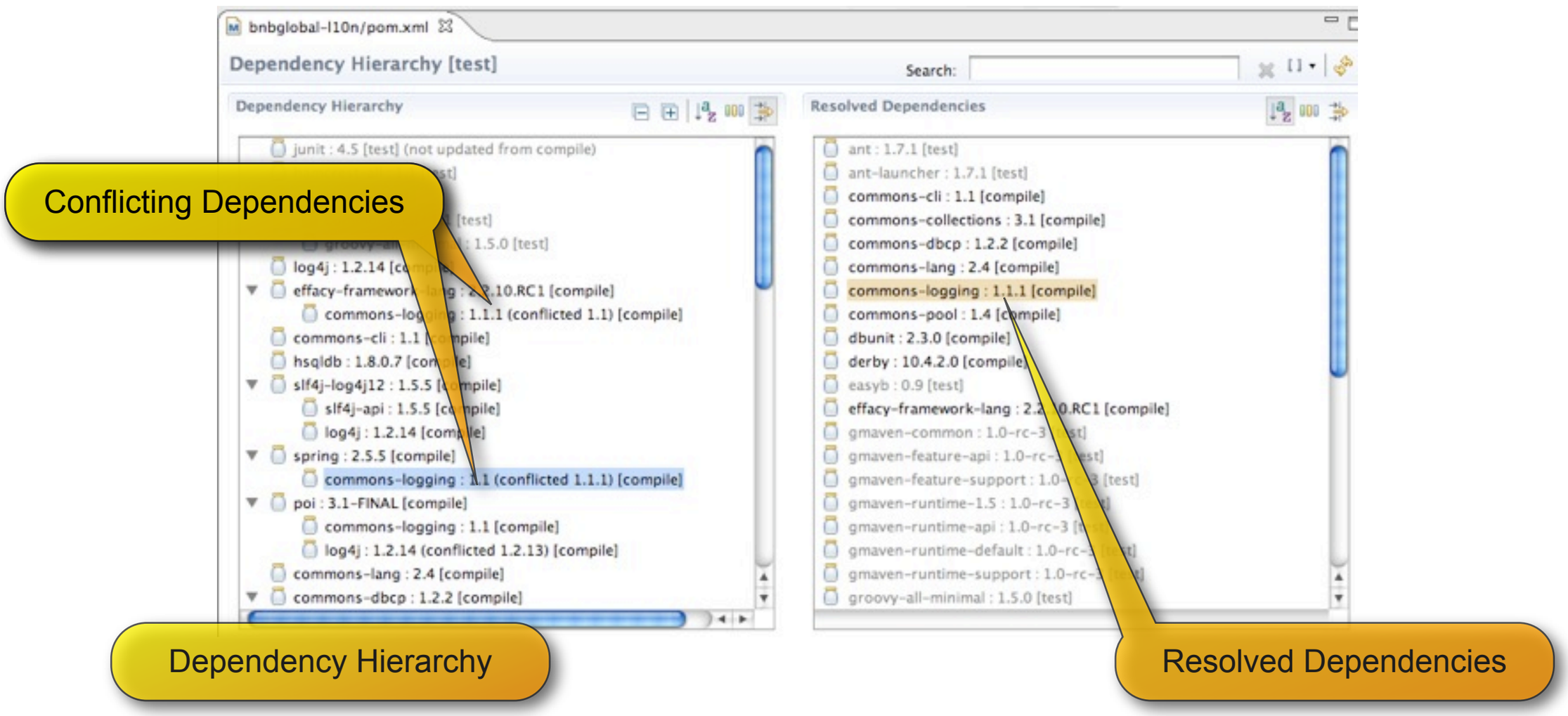
- Transitive Dependencies
  - POMs declare dependencies on other artifacts
  - Using Maven coordinates, Maven recursively adds the dependencies to the current project
- Maven builds graphs of dependencies and handles any conflicts that may occur
  - Always favors a more recent version of any artifact when selecting from a range

Visualizing Dependencies

- Visualizing Dependencies in Eclipse
  - Dependency Hierarchy
  - Dependency Graph

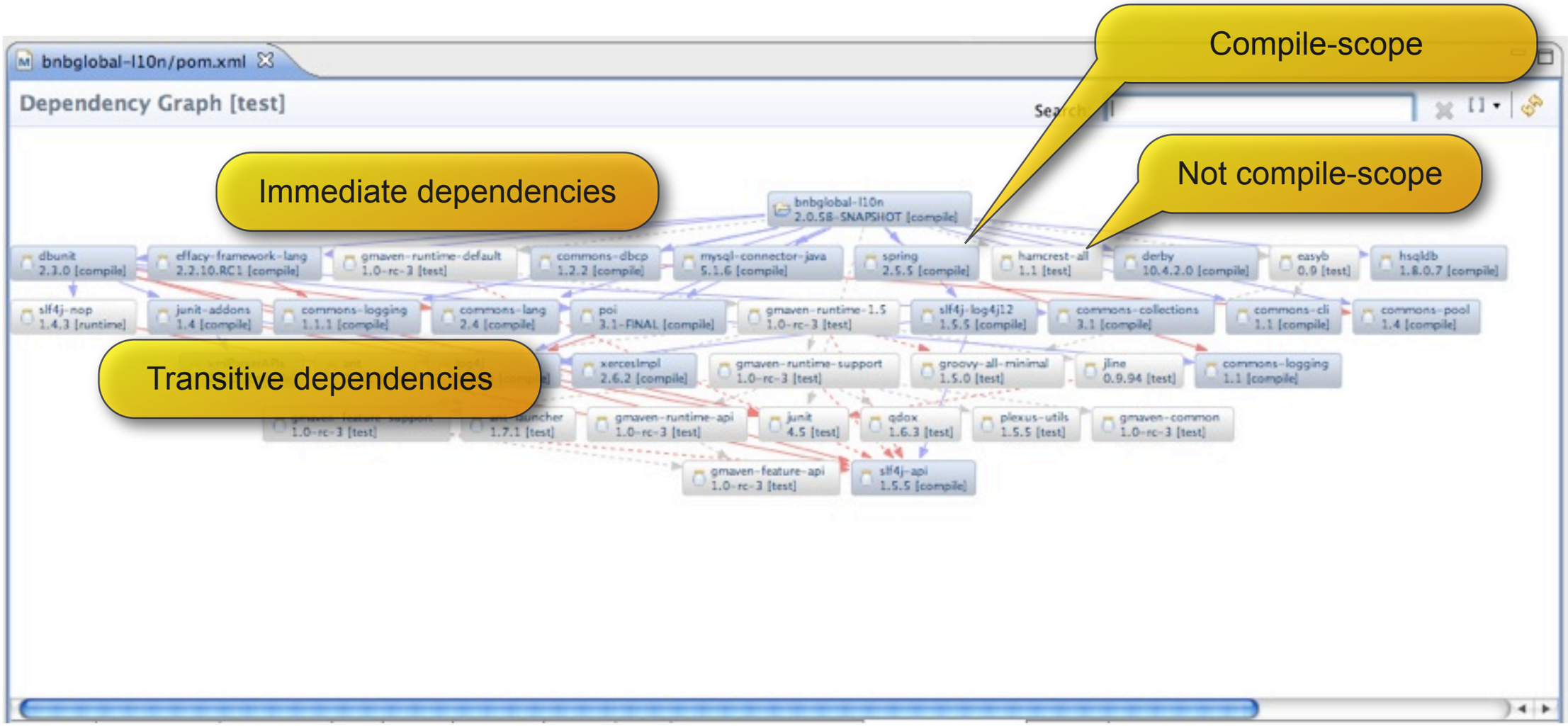
Visualizing Dependencies

- Dependency Hierarchy



Visualizing Dependencies

- Dependency Graph



Visualizing Dependencies

- From the command line
  - List your dependencies

mvn dependency:resolve

```
$ mvn dependency:resolve
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'dependency'.
[INFO] ------------------------------------------------------------------------
[INFO] Building babble-core
[INFO]    task-segment: [dependency:resolve]
[INFO] ------------------------------------------------------------------------
[INFO] [dependency:list]
[INFO]
[INFO] The following files have been resolved:
[INFO]    antlr:antlr:jar:2.7.6:compile
[INFO]    asm:asm:jar:1.5.3:compile
[INFO]    asm:asm-attrs:jar:1.5.3:compile
[INFO]    cglib:cglib:jar:2.1_3:compile
[INFO]    commons-collections:commons-collections:jar:2.1.1:compile
[INFO]    commons-logging:commons-logging:jar:1.0.4:compile
[INFO]    dom4j:dom4j:jar:1.6.1:compile
[INFO]    javax.persistence:persistence-api:jar:1.0:compile
[INFO]    javax.transaction:jta:jar:1.0.1B:compile
[INFO]    junit:junit:jar:4.5:test
[INFO]    net.sf.ehcache:ehcache:jar:1.2:compile
[INFO]    org.hamcrest:hamcrest-all:jar:1.1:compile
[INFO]    org.hibernate:hibernate:jar:3.2.0.ga:compile
[INFO]    org.hibernate:hibernate-annotations:jar:3.2.0.ga:compile
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
```

Displays a list of resolved dependencies

Visualizing Dependencies

- From the command line
  - View your dependencies

mvn dependency:tree

```
$ mvn dependency:tree
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'dependency'.
[INFO] ------------------------------------------------------------------------
[INFO] Building babble-core
[INFO]    task-segment: [dependency:tree]
[INFO] ------------------------------------------------------------------------
[INFO] [dependency:tree]
[INFO] com.sonatype.training:babble-core:jar:1.0-SNAPSHOT
[INFO] +- org.hibernate:hibernate:jar:3.2.0.ga:compile
[INFO] |  +- net.sf.ehcache:ehcache:jar:1.2:compile
[INFO] |  +- javax.transaction:jta:jar:1.0.1B:compile
[INFO] |  +- commons-logging:commons-logging:jar:1.0.4:compile
[INFO] |  +- asm:asm-attrs:jar:1.5.3:compile
[INFO] |  +- dom4j:dom4j:jar:1.6.1:compile
[INFO] |  +- antlr:antlr:jar:2.7.6:compile
[INFO] |  +- cglib:cglib:jar:2.1_3:compile
[INFO] |  +- asm:asm:jar:1.5.3:compile
[INFO] |  \- commons-collections:commons-collections:jar:2.1.1:compile
[INFO] +- org.hibernate:hibernate-annotations:jar:3.2.0.ga:compile
[INFO] |  \- javax.persistence:persistence-api:jar:1.0:compile
[INFO] +- junit:junit:jar:4.5:test
[INFO] \- org.hamcrest:hamcrest-all:jar:1.1:compile
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
```

Displays a tree-structure of your dependencies

Visualizing Dependencies

- From the command line
  - Optimize your dependencies
  - Find unused dependencies
  - Declare important dependencies more precisely

mvn dependency:analyse

```
$ mvn dependency:analyze
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'dependency'.
[INFO] ------------------------------------------------------------------------
[INFO] Building babble-core
[INFO]    task-segment: [dependency:analyze]
[INFO] ------------------------------------------------------------------------
[INFO] Preparing dependency:analyze
...
[INFO] [dependency:analyze]
[WARNING] Used undeclared dependencies found:
[WARNING]    javax.persistence:persistence-api:jar:1.0:compile
[WARNING] Unused declared dependencies found:
[WARNING]    org.hibernate:hibernate-annotations:jar:3.2.0.ga:compile
[WARNING]    org.hibernate:hibernate:jar:3.2.0.ga:compile
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 4 seconds
[INFO] Finished at: Mon Mar 30 14:19:04 NZDT 2009
[INFO] Final Memory: 13M/26M
[INFO] ------------------------------------------------------------------------
```

JPA annotations are used but not directly declared

Hibernate libraries are declared but not used

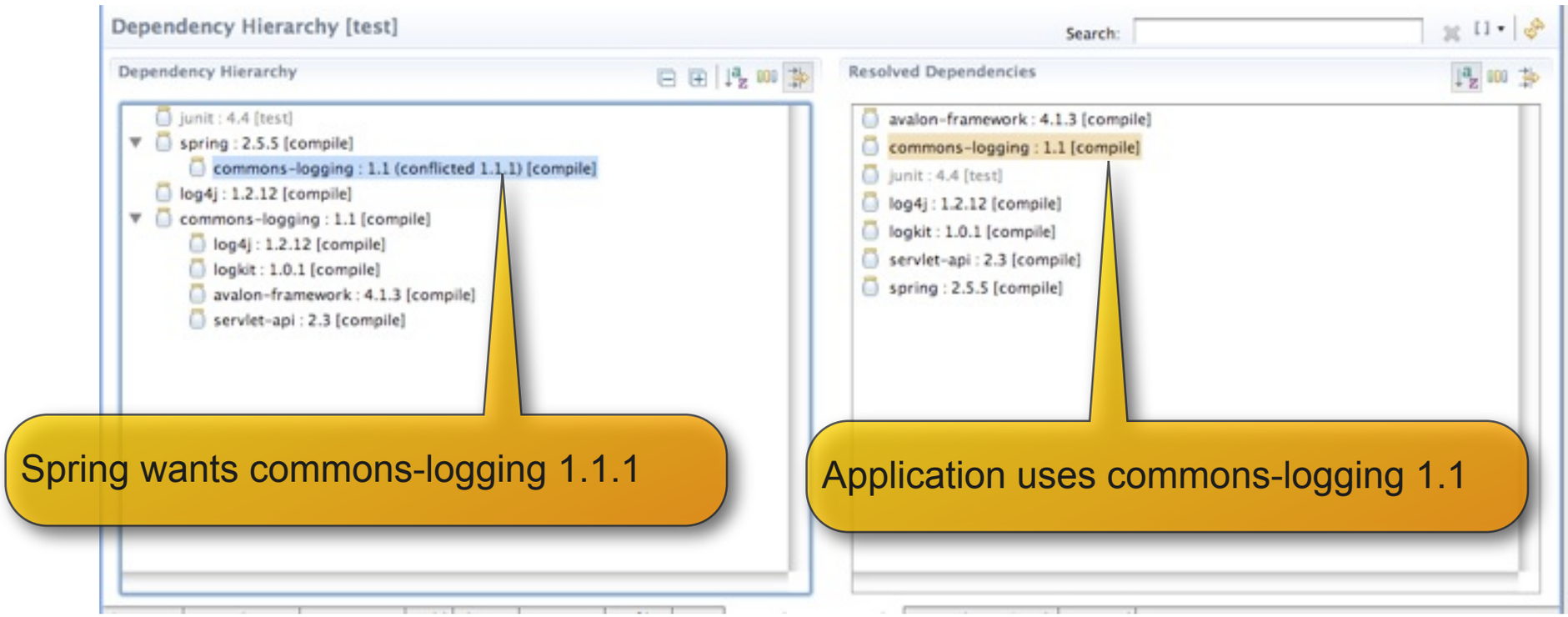
Dependency Conflicts

Dependency Conflicts

- Dependency Conflicts
  - Different libraries require different versions of the same dependency
  - By default:
    - The nearest dependency to the top wins
    - The first dependency declared at a given level wins
  - Sometimes, we need to override the default behavior

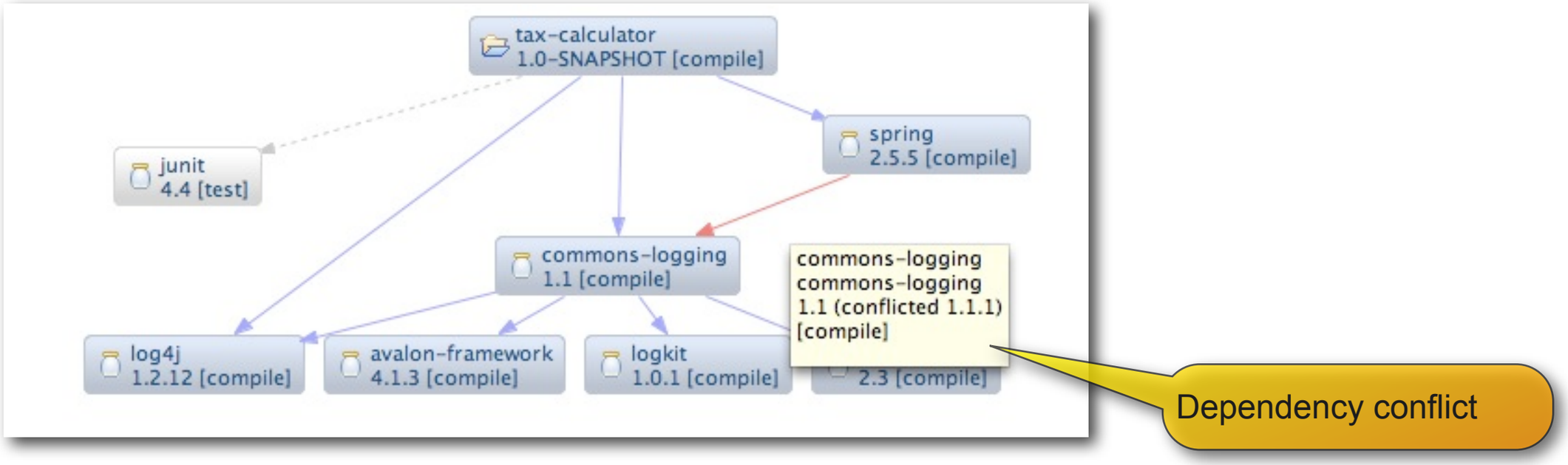
Dependency Conflicts

- Dependency Conflicts
  - You can visualize conflicts in the Dependency Hierarchy view



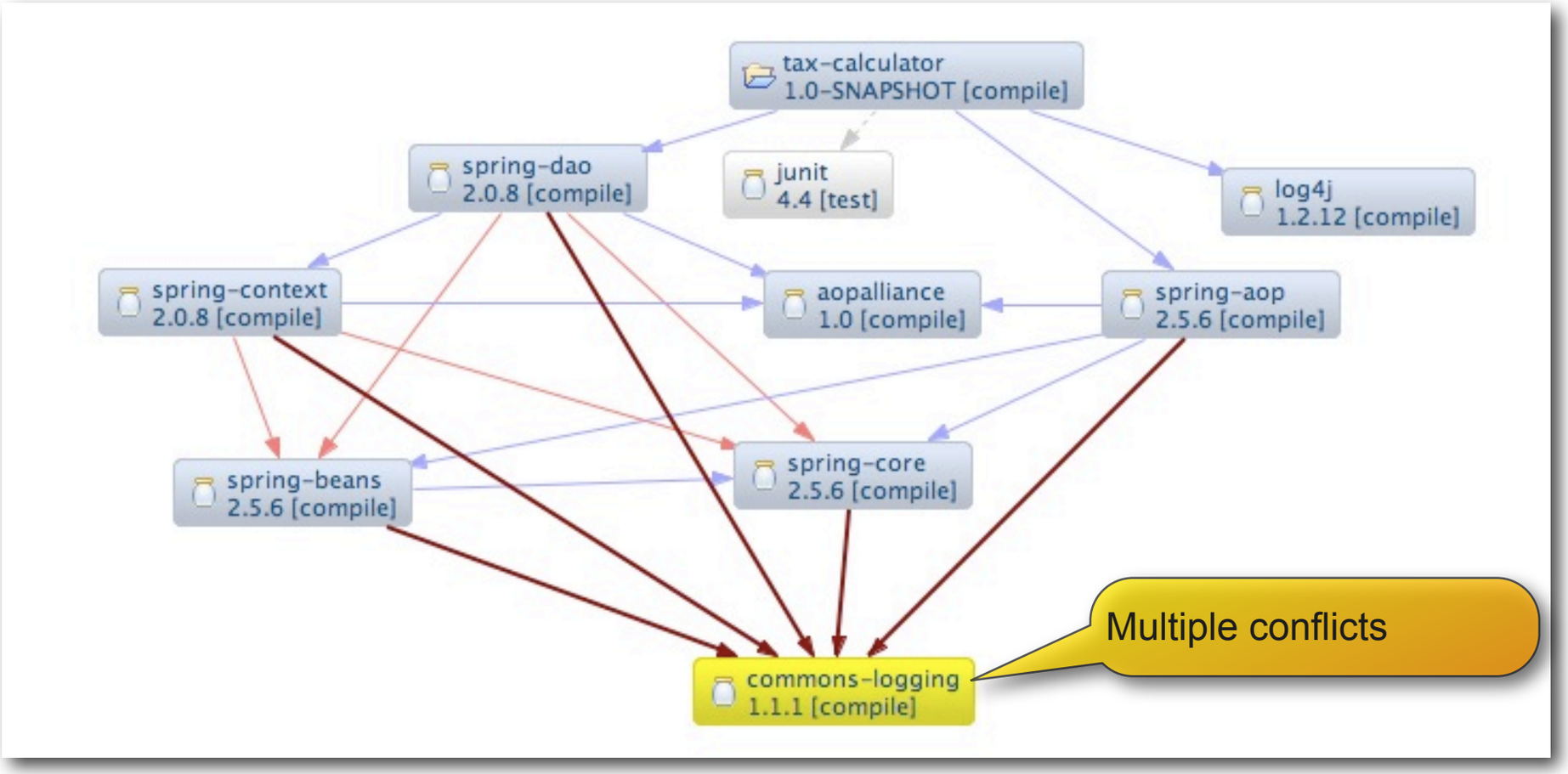
Dependency Conflicts

- Dependency Conflicts
  - Or in the Dependency Graph



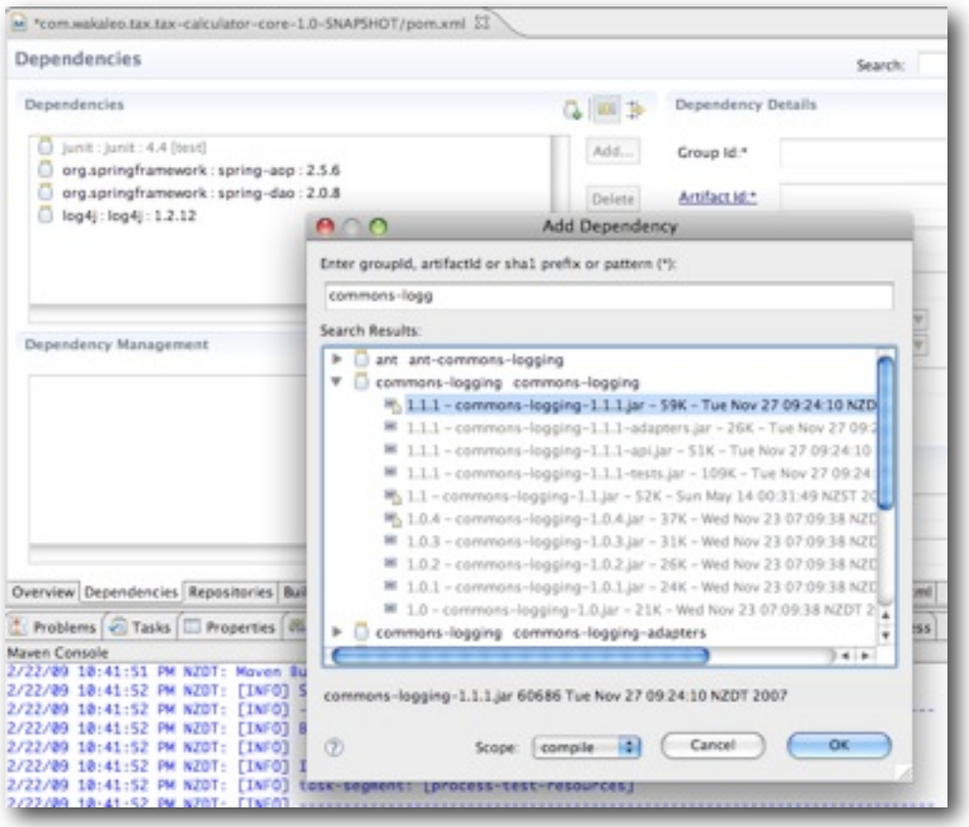
Dependency Conflicts

- Dependency Conflicts
  - A more complicated case:



Dependency Conflicts

- Dependency Conflict quick fix:
  - Declare the correct version in your POM file



  - Or exclude the unwanted version explicitly

<dependencyManagement> and <pluginManagement>

Inherited Behavior

- Using DependencyManagement to inherit dependencies
  - Use `<dependencyManagement>` to centralizes version numbers
    - Declare the official version numbers in a parent POM file
    - Only declare the artifacts in the child projects

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>a-parent</artifactId>
  <version>1.0.0</version>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifa
        <version>5.1.2</version>
      </dependency>
      ...
    <dependencies>
```

Official version number

Only declare the artifact here

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook</groupId>
    <artifactId>a-parent</artifactId>
    <version>1.0.0</version>
  </parent>
  <artifactId>project-a</artifactId>
  ...
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
    </dependency>
  </dependencies>
</project>
```

Inherited Behavior

- Using DependencyManagement to inherit dependencies
  - The `<dependencyManagement>` section lists dependency version numbers
  - It does *not* add any dependencies to the project

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>a-parent</artifactId>
  <version>1.0.0</version>
  ...
  <dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.5</version>
            <scope>test</scope>
        </dependency>
         <dependency>
            <groupId>org.hamcrest</groupId>
            <artifactId>hamcrest-all</artifactId>
            <version>1.1</version>
        </dependency>

      ...
    <dependencies>
  </dependencyManagement>
```

Official version numbers for JUnit and Hamcrest

To be used if these libraries are required by child projects

This project only needs JUnit

It will depend on JUnit 4.5

It will not have a hamcrest dependency

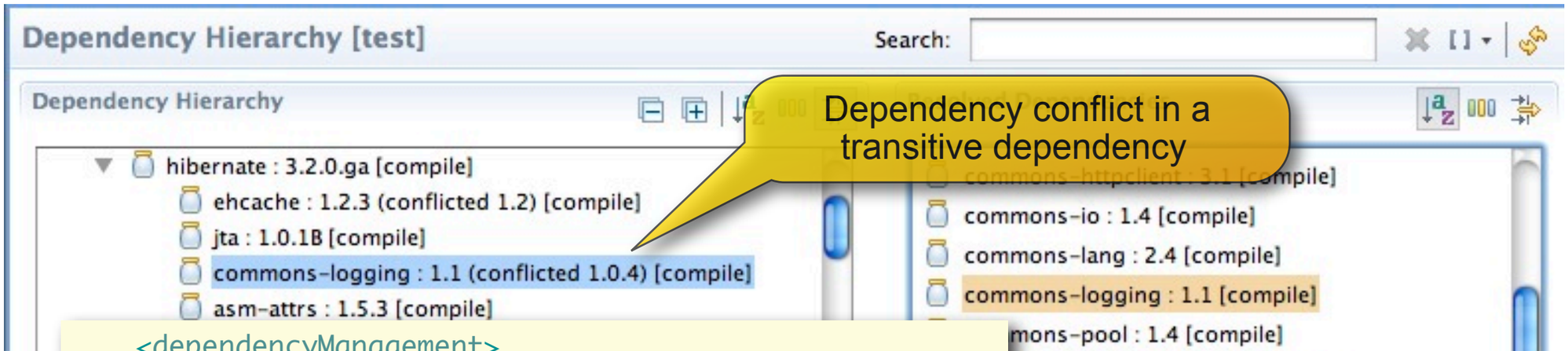
```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook</groupId>
    <artifactId>a-parent</artifactId>
    <version>1.0.0</version>
  </parent>
  <artifactId>project-a</artifactId>
  ...
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
    </dependency>
  </dependencies>
</project>
```

Overriding Transitive Dependencies

- DependencyManagement with Transitive Dependencies
  - `<dependencyManagement>` also applies to Transitive Dependencies
    - This can be used to override versions of dependencies you don't directly depend on.
    - For example, if a specific version of a logger conflicts with your application server or has a known bug.

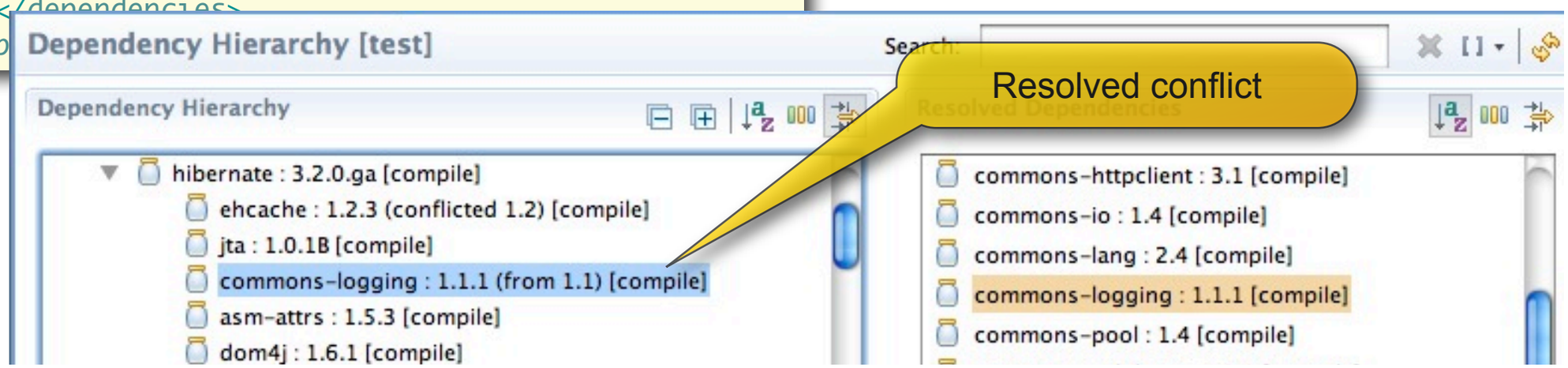
Overriding Transitive Dependencies

- DependencyManagement with Transitive Dependencies




```
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
                <version>1.1.1</version>
            </dependency>
        </dependencies>
    </dep
```

Declare the version we want in the DependencyManagement section



Inherited Plugin Behavior

- Optimizing plugin dependencies
  - Similar to the `<dependencyManagement>` section
  - It does *not* add any new plugins to the project

Parent POM

```
<build>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>org.easyb</groupId>
                <artifactId>maven-easyb-plugin</artifactId>
                <version>0.9.6</version>
                <configuration>
                    <storyType>html</storyType>
                    <storyReport>target/easyb/easyb.html</storyReport>
                </configuration>
                <executions>
                    <execution>
                        <goals>
                            <goal>test</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </pluginManagement>
</build>
```

Applies for any child project using this plugin

Child POM

```
<build>
    <plugins>
        <plugin>
            <groupId>org.easyb</groupId>
            <artifactId>maven-easyb-plugin</artifactId>
        </plugin>
        ...
```

Inherits plugin configuration from the parent

Inherited Plugin Behavior

- Optimizing plugin dependencies
  - Lifecycle-related plugins apply to *all* child projects

```
<pluginManagement>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>2.0.2</version>
            <configuration>
                <source>${jdk.version}</source>
                <target>${jdk.version}</target>
            </configuration>
        </plugin>
    </plugins>
</pluginManagement>
```

Parent POM

This is a lifecycle plugin

Applies for *all* child projects

Exclusions

Excluding Transitive Dependencies

- Excluding Dependencies lets you
  - Override normal transitive dependency management
  - Exclude certain libraries that would normally be transitively included

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring</artifactId>
    <version>2.5.5</version>
    <exclusions>
     <exclusion>
      <groupId>javax.jms</groupId>
      <artifactId>jms<artifactId>
    </exclusion>
   </exclusions>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.specs</groupId>
    <artifactId>geronimo-jms_1.1_spec</artifact>
    <version>1.1</version>
  </dependency>
<dependencies>
```

Exclude the javax.jms dependency

Use the Apache Geronimo JMS specs instead

Questions

- When would should something be excluded?
- Are exclusions used very often?
- What is a better technique than an exclusion for version management?

labs

- Lab 8 - Working with dependencies
  - Update an existing dependency
  - Add a dependency using M2Eclipse
  - View dependencies in M2Eclipse
  - Manage dependencies and resolve a conflict

Inheritance and Aggregation

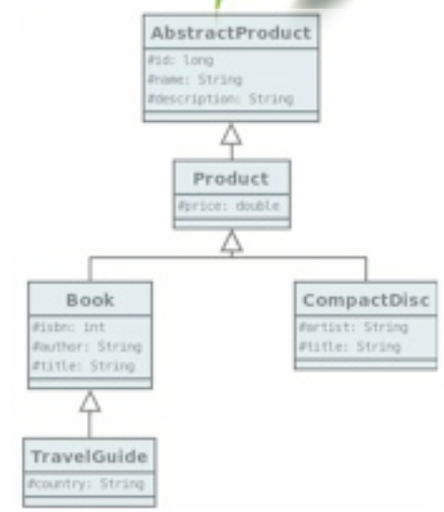
Inheritance and Aggregation

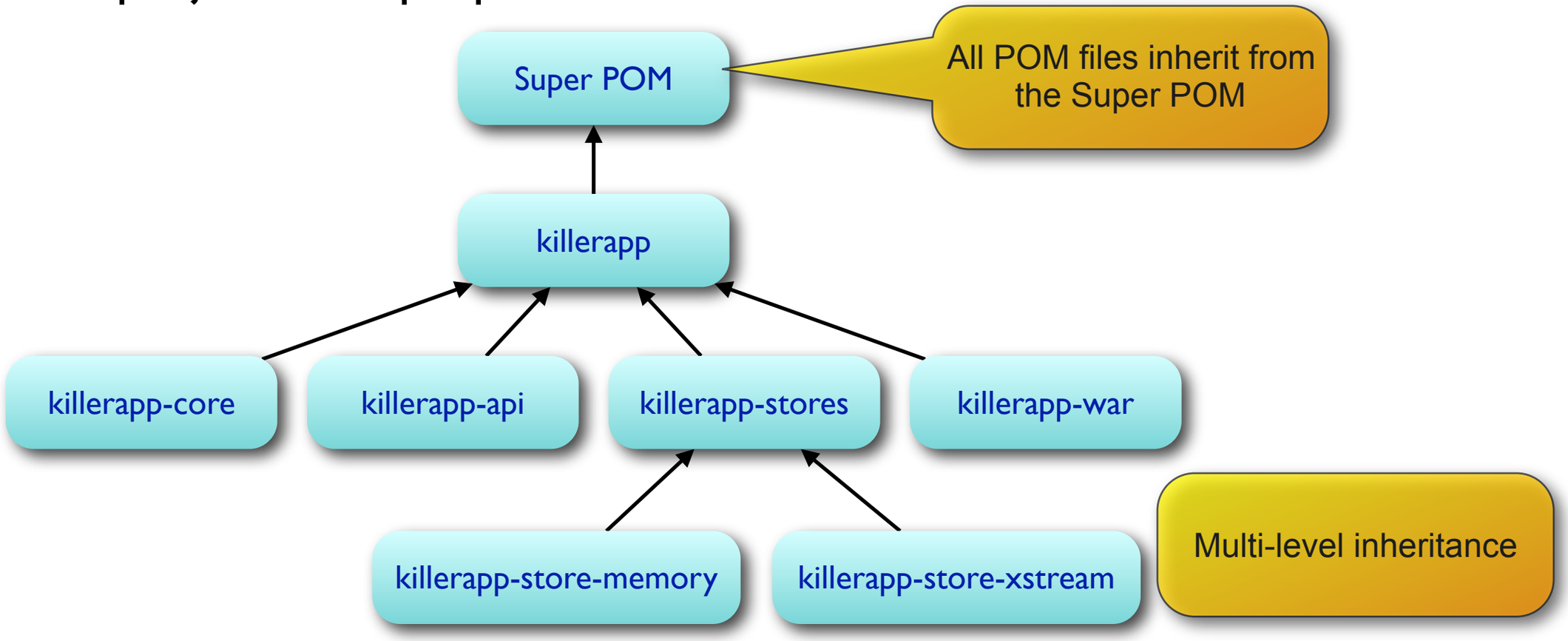
- Relationships with other projects
  - The POM is a "Project Object Model"
  - You can define relationships between projects (or POMs)
    - Dependencies
    - Inheritance
    - Aggregation (multi-module projects)
  - All POMs derive from a special POM, the Super-POM
    - The inheritance is undeclared...think Java "Object"

Inheritance

POM Inheritence

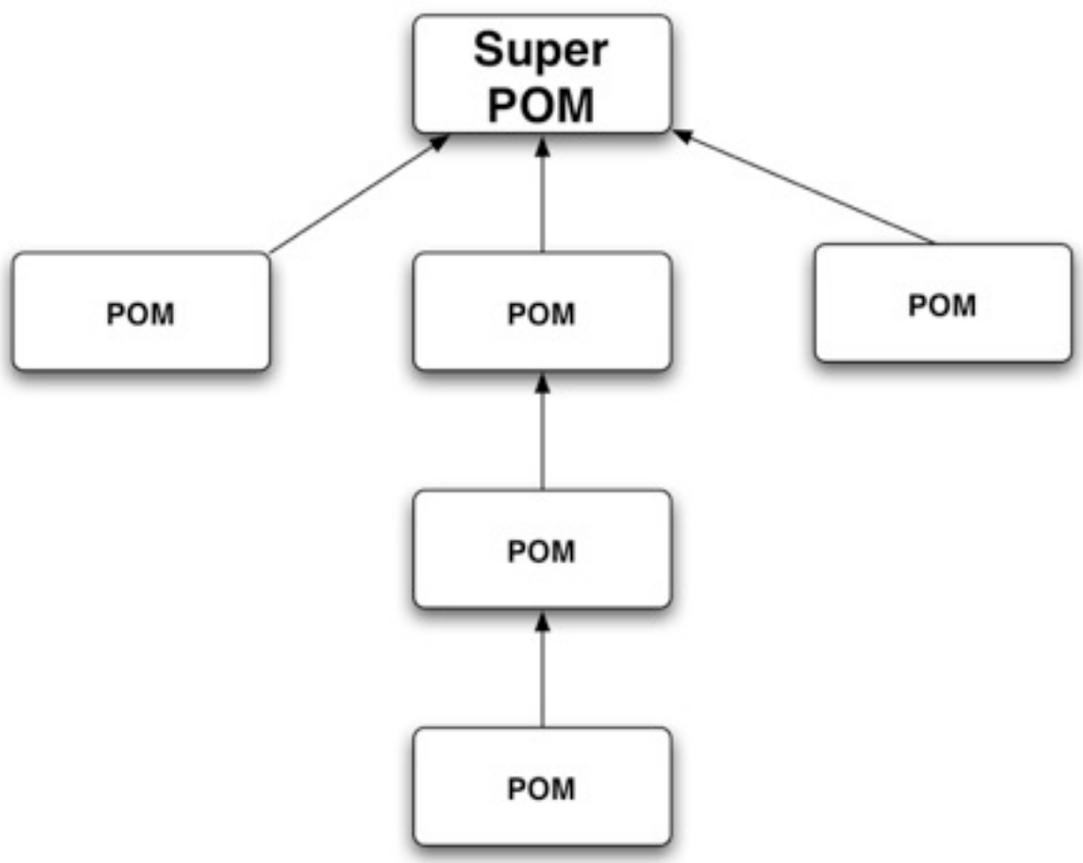
- Project inheritance
  - Encourages modular design
  - Small modules are easier to test
  - Encourages reuse
  - Define project-wide properties and behaviour





The Super POM

- The ultimate parent POM for all POMs
  - Implicitly inherited by all projects
  - Defines a set of defaults shared by all projects
  - The source of most Maven conventions



The Super POM

- Defines the central Maven repository as the default repo
  - Additional repos can be added in your project POM or a settings.xml
- Defines the central Maven repository as the default plugin repo
  - The update policy is set to never
  - Sets the default values for directories
  - Starting in Maven 2.0.9, default versions of some core plugins are included
    - Provides improved stability for users not defining plugin versions in their POMs
- The Super POM is installed with Maven
  - Located in the `maven-2.x.y-uber.jar` file in `${M2_HOME}/lib`

The Super POM

- The Super POM: a brief overview

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <name>Maven Default Project</name>

  <repositories>
    <repository>
      <id>central</id>
      <name>Maven Repository Switchboard</name>
      <layout>default</layout>
      <url>http://repo1.maven.org/maven2</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>central</id>
      <name>Maven Plugin Repository</name>
      <url>http://repo1.maven.org/maven2</url>
      ...
    </pluginRepository>
  </pluginRepositories>
  ...
</project>
```

Configure the central repository

Configure the central plugin repository

The Super POM

- The Super POM: a brief overview

```
<project>
  ...
 <build>
    <directory>${project.basedir}/target</directory>
    <outputDirectory>${project.build.directory}/classes</outputDirectory>
    <finalName>${project.artifactId}-${project.version}</finalName>
    <testOutputDirectory>${project.build.directory}/test-classes</testOutputDirectory>
    <sourceDirectory>${project.basedir}/src/main/java</sourceDirectory>
    <scriptSourceDirectory>src/main/scripts</scriptSourceDirectory>
    <testSourceDirectory>${project.basedir}/src/test/java</testSourceDirectory>
    <resources>
      <resource>
        <directory>${project.basedir}/src/main/resources</directory>
      </resource>
    </resources>
    <testResources>
      <testResource>
        <directory>${project.basedir}/src/test/resources</directory>
      </testResource>
    </testResources>
    ...
</project>
```

The standard directory structure and other conventions

The Super POM

- The Super POM: a brief overview

```
<project>
  ...
   <pluginManagement>
       <plugins>
         <plugin>
           <artifactId>maven-antrun-plugin</artifactId>
           <version>1.3</version>
         </plugin>
         <plugin>
           <artifactId>maven-assembly-plugin</artifactId>
           <version>2.2-beta-2</version>
         </plugin>
         <plugin>
           <artifactId>maven-clean-plugin</artifactId>
           <version>2.2</version>
         </plugin>
    ...
</project>
```

Versions of lifecycle plugins are fixed

The Super POM

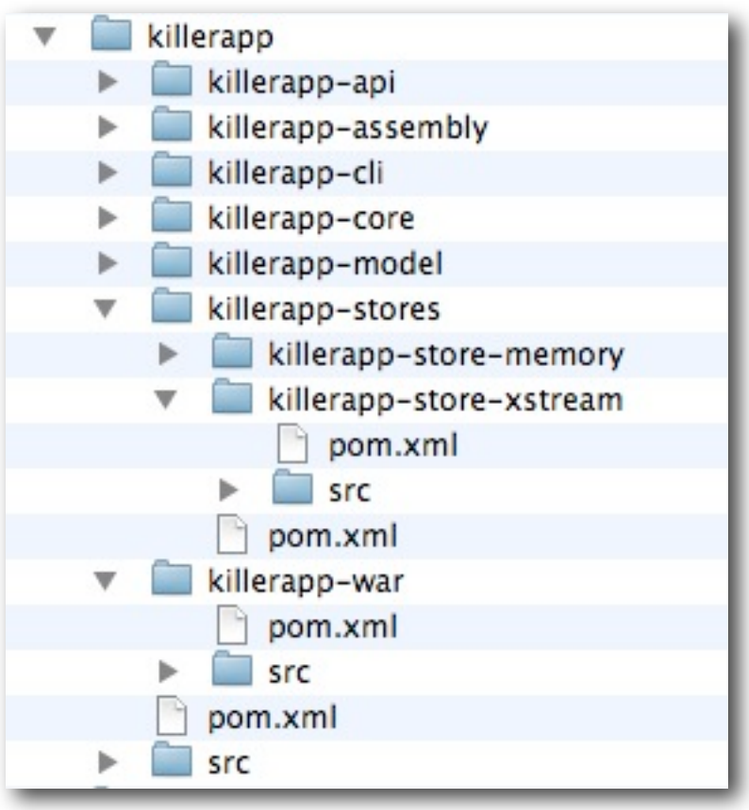
- The Super POM: a brief overview

```
<project>
  ...
<profiles>
    <profile>
      <id>release-profile</id>

      <activation>
        <property>
          <name>performRelease</name>
          <value>true</value>
        </property>
      </activation>

      <build>
        <plugins>
          <plugin>
            <inherited>true</inherited>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-source-plugin</artifactId>
            ...
          </plugin>
...
</project>
```

Special behavior for the release process (attach source code etc.)

POM Inheritence

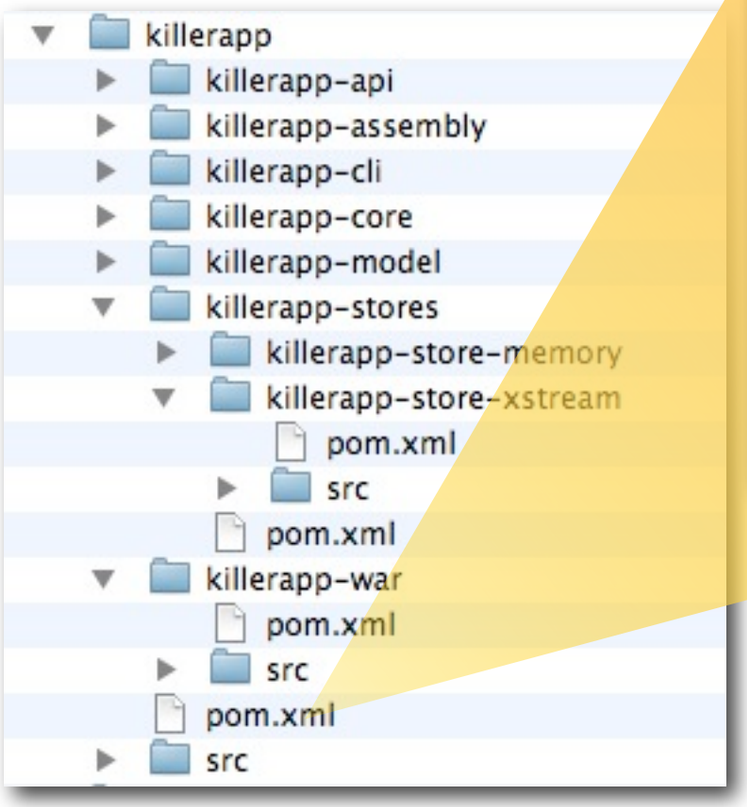
- The Parent POM
  - Regroups common project behavior and properties
  - Project often contains only a pom.xml file
  - Often placed in a parent directory
  - <relativePath> optional, defaults to ../pom.xml



Note: Using a non-standard directory structure can cause issues with the Release plugin

POM Inheritence

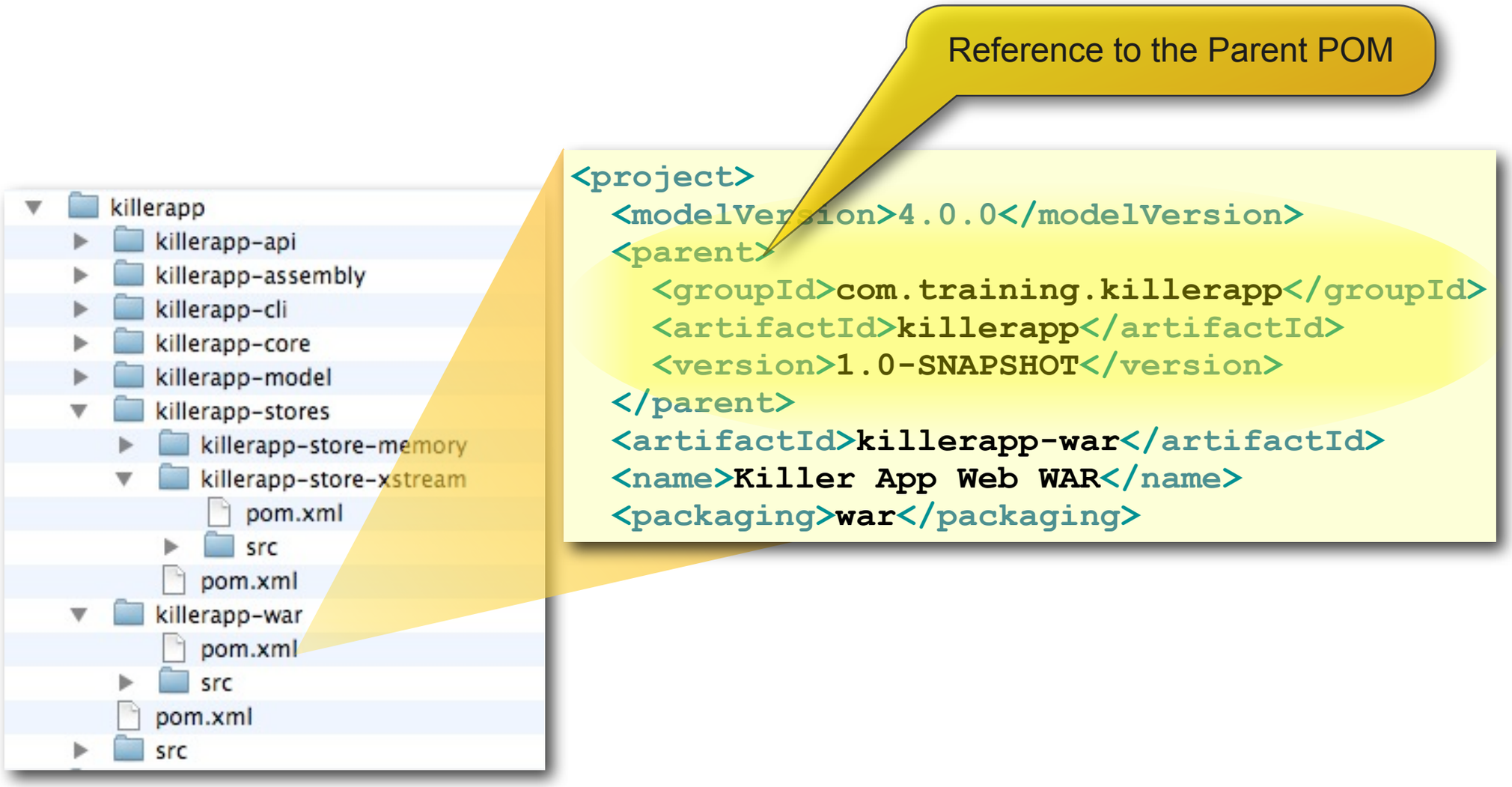
- The Parent POM
  - Uses the special 'pom' packaging




```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.training.killerapp</groupId>
  <artifactId>killerapp</artifactId>
  <packaging>pom</packaging>
  <name>Killer App</name>
  <version>1.0-SNAPSHOT</version>
  <description>A sample application</description>
  ...
</project>
```

POM Inheritence

- Creating a child project
  - Each child project refers to the parent POM



Inherited Behavior

- Typical inherited behavior and properties
  - Common build behavior
  - Common properties and build profiles
  - Common dependencies and/or dependency versions
  - Reporting

Inherited Behavior

- Inheriting common build behavior
  - Build behavior is often shared across modules
    - Reduce repetition in the pom.xml files
    - Easier to maintain
  - An example: common compiler configuration

```
<project...>
    ...
    <properties>
        <java-api.version>1.5</java-api.version>
    </properties>
    <build>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>${java-api.version}</source>
                    <target>${java-api.version}</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

This compiler configuration will be applicable for all the contained modules

Inherited Behavior

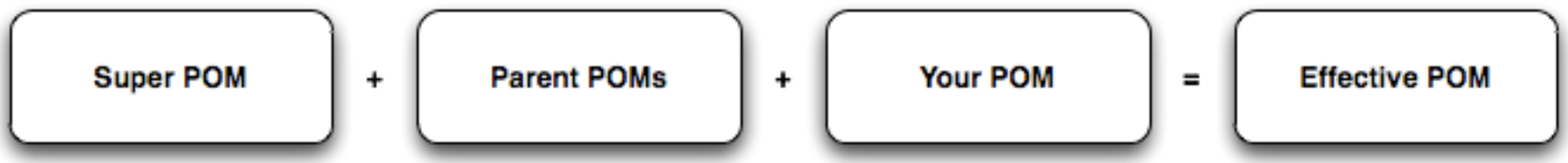
- Inheriting common dependencies
  - Shared dependencies can be placed in the parent `pom.xml`
    - More consistent dependencies
    - Reduced repetition
    - Easier to maintain

```
<project...>
    ...
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.5</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.15</version>
        </dependency>
    </dependencies>
    ...
</project>
```

These dependencies will be inherited by child projects

The Effective POM

- The Concept of the Effective POM
  - Maven POMs are composed of many POMS
  - The Effective POM is composed of the inheritance hierarchy
  - Useful for verifying inherited configuration and properties



The Effective POM

- Viewing the Effective POM
  - Print the Effective POM using **mvn help:effective-pom**

```
someuser$ mvn help:effective-pom
...
[INFO] ------------------------------------------------------------------------
[INFO] Building Killer App Core
[INFO]    task-segment: [help:effective-pom] (aggregator-style)
[INFO] ------------------------------------------------------------------------
[INFO] [help:effective-pom]
[INFO]
Effective POMs, after inheritance, interpolation, and profiles are applied:

<?xml version="1.0" encoding="UTF-8"?>
<!-- ========================================================================== -->
<!--                                                                            -->
<!-- Generated by Maven Help Plugin on 3/10/09 5:19 PM                          -->
<!-- See: http://maven.apache.org/plugins/maven-help-plugin/                    -->
<!--                                                                            -->
<!-- ========================================================================== -->

<!-- ========================================================================== -->
<!--                                                                            -->
<!-- Effective POM for project                                                  -->
<!-- 'com.training.killerapp:killerapp-core:jar:1.0-SNAPSHOT'                   -->
<!--                                                                            -->
<!-- ========================================================================== -->

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <artifactId>killerapp</artifactId>
    <groupId>com.training.killerapp</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <groupId>com.training.killerapp</groupId>
  <artifactId>killerapp-core</artifactId>
  <version>1.0-SNAPSHOT</version>
  ...
```

The Effective POM

- Viewing the Effective POM in Eclipse

tweeter-web/pom.xml

```
<groupId>com.wakaleo.training.tweeter</groupId>
<artifactId>tweeter-web</artifactId>
<version>1.0.0-SNAPSHOT</version>
<packaging>war</packaging>
<name>Tweeter web application</name>
<description>Tweeter - a Twitter clone used for training exercises</description>
<inceptionYear>2009</inceptionYear>
<build>
  <sourceDirectory>/Users/johnsmart/Projects/wakaleo-training/tdd-training/lab-solutions/tweeter/tweeter-w
  <scriptSourceDirectory>/Users/johnsmart/Projects/wakaleo-training/tdd-training/lab-solutions/tweeter/twe
  <testSourceDirectory>/Users/johnsmart/Projects/wakaleo-training/tdd-training/lab-solutions/tweeter/tweet
  <outputDirectory>/Users/johnsmart/Projects/wakaleo-training/tdd-training/lab-solutions/tweeter/tweeter-w
  <testOutputDirectory>/Users/johnsmart/Projects/wakaleo-training/tdd-training/lab-solutions/tweeter/tweet
  <resources>
    <resource>
      <filtering>true</filtering>
      <directory>/Users/johnsmart/Projects/wakaleo-training/tdd-training/lab-solutions/tweeter/tweeter-web/
    </resource>
  </resources>
  <testResources>
    <testResource>
      <directory>/Users/johnsmart/Projects/wakaleo-training/tdd-training/lab-solutions/tweeter/tweeter-web/
    </testResource>
  </testResources>
  <directory>/Users/johnsmart/Projects/wakaleo-training/tdd-training/lab-solutions/tweeter/tweeter-web/tar
  <finalName>tweeter-web</finalName>
  <pluginManagement>
    <plugins>
```

Overview | Dependencies | Plugins | Reporting | Dependency Hierarchy | Dependency Graph | Effective POM | pom.xml

Modules

Multi-Module Projects

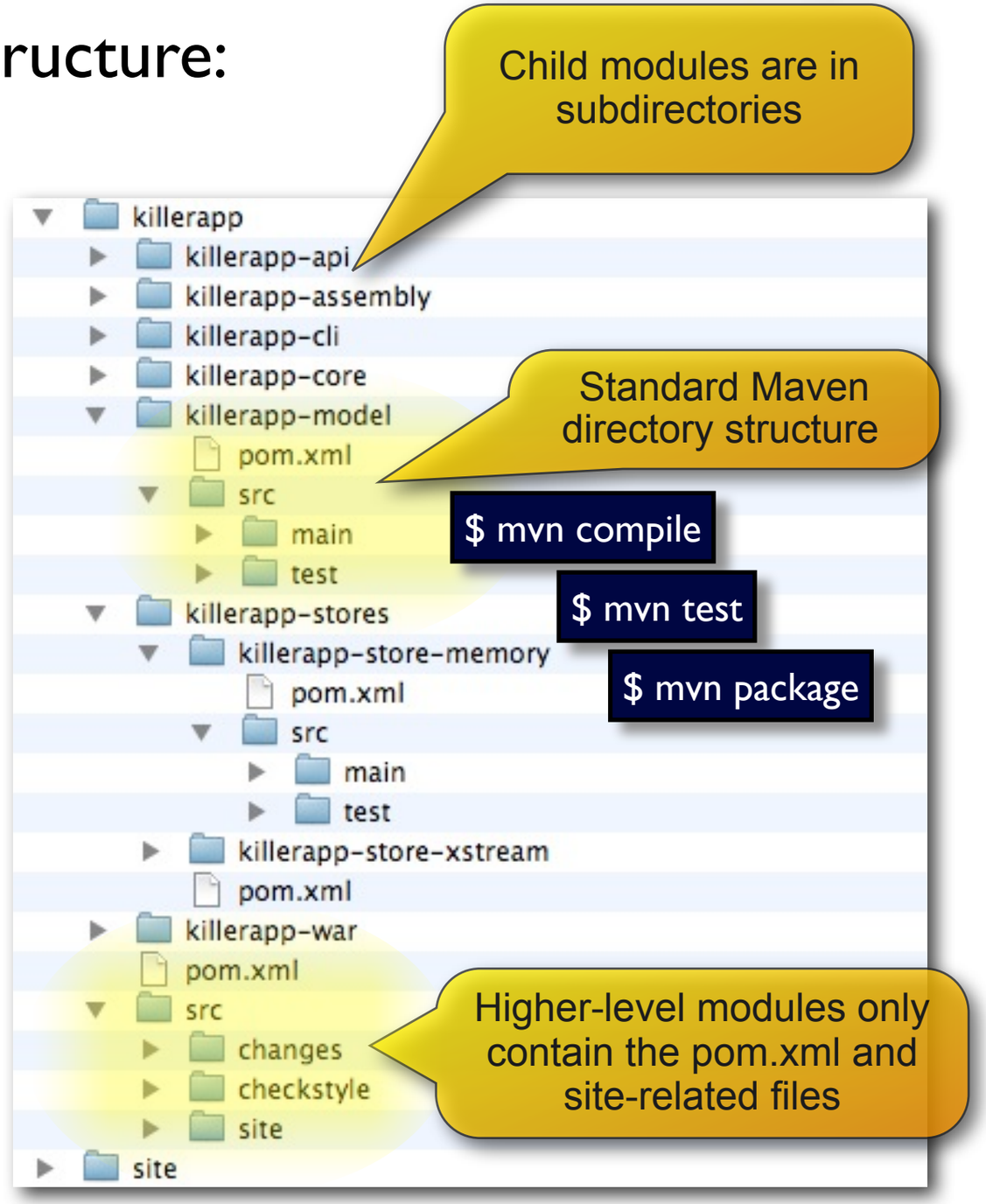
- It is good practice to break larger projects into modules
  - Smaller modules are less complex and easier to test
  - Easier to enforce software architecture design rules
  - Easier to separate work between team members
  - Promotes good design:
    - Modularity
    - Separation of concerns
    - Loose coupling

Multi-Module Projects

- In Maven, you do this using modules
  - Modules act as many project working together as one
    - Arbitrary number of levels
    - A project can contain any number of modules
  - Sub-projects focus on a specific task or architectural layer
    - persistence-api
    - services-api
    - webapp
    - ...

Directory Structure

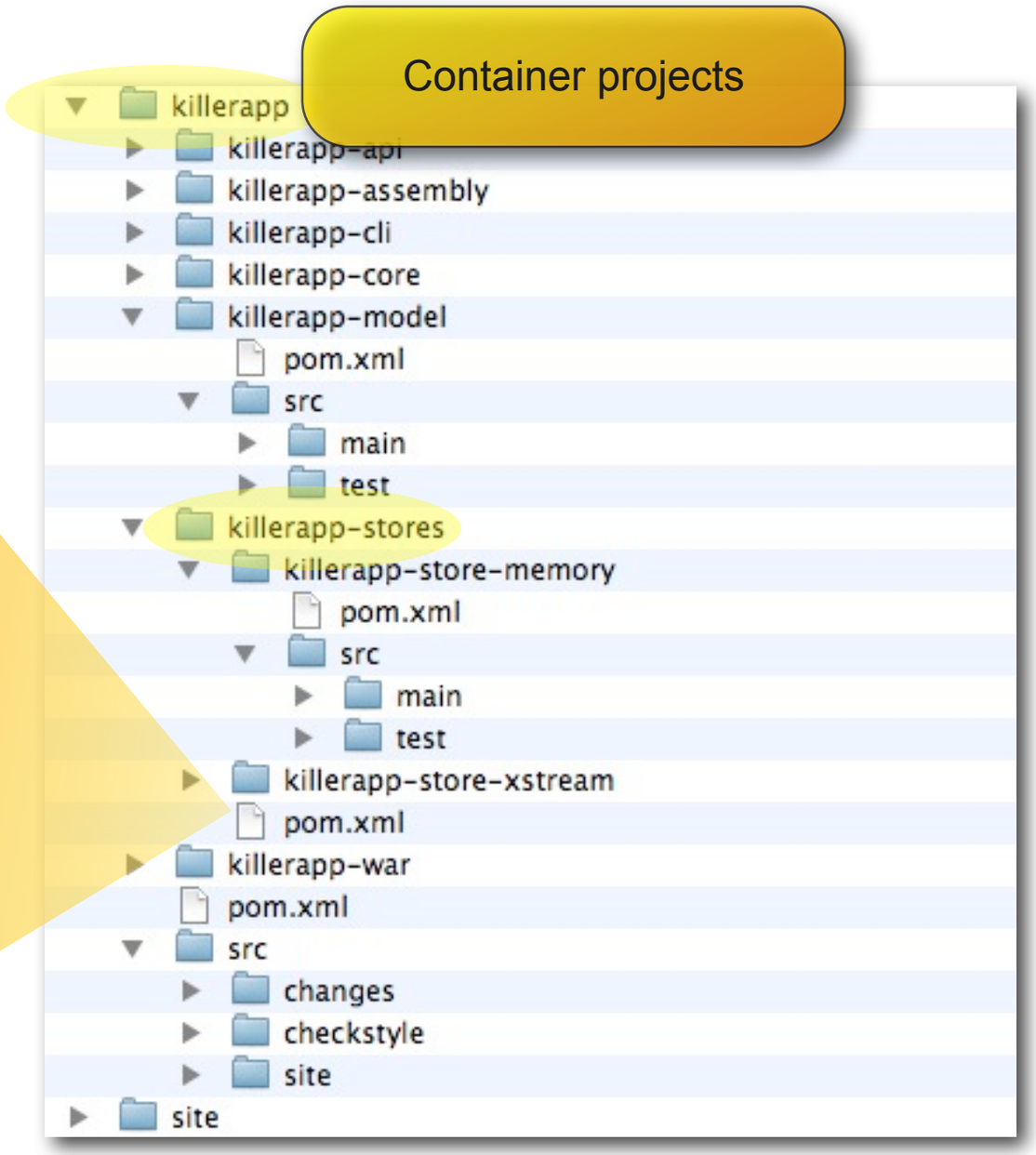
- Multi-module project directory structure:
  - Matches the project architecture
  - Each module has its own directory structure
  - Each module can be built in isolation
  - By convention, the directory name matches the module's artifactId



Container Projects

- Container projects
  - Cannot produce artifacts
  - Packaging must be type ‘pom’
  - Sub-modules are listed in the `<modules>` section

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.training.killerapp</groupId>
  <artifactId>killerapp</artifactId>
  <packaging>pom</packaging>
  <name>Killer App</name>
  <version>1.0-SNAPSHOT</version>
  ...
  <modules>
    <module>killerapp-model</module>
    <module>killerapp-api</module>
    <module>killerapp-core</module>
    <module>killerapp-stores</module>
    <module>killerapp-cli</module>
    <module>killerapp-war</module>
  </modules>
  ...
</project>
```



POM Aggregation

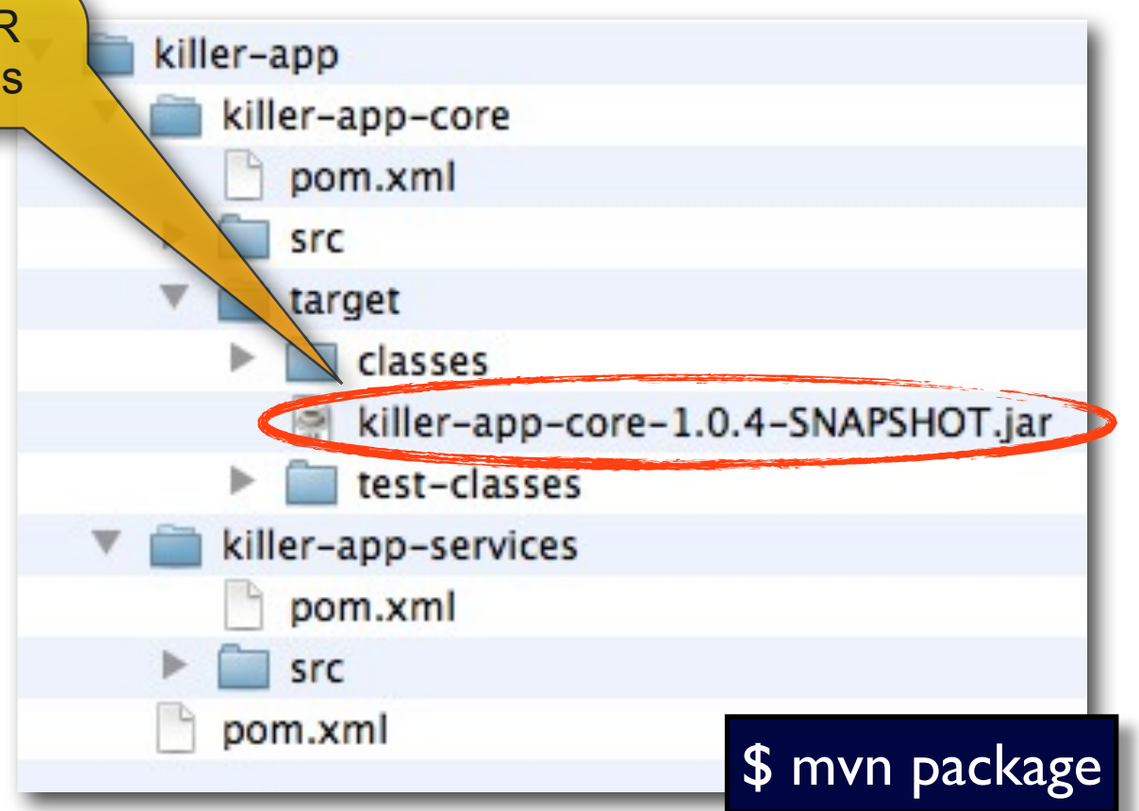
- Project Aggregation
  - Builds are executed on each module in top-down order

```
someuser$ mvn compile
...
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO] ------------------------------------------------------------------------
[INFO] Killer App ........................................ SUCCESS [0.550s]
[INFO] Killer App Model ................................... SUCCESS [0.379s]
[INFO] Killer App API ..................................... SUCCESS [0.046s]
[INFO] Killer App Stores .................................. SUCCESS [0.006s]
[INFO] Killer App Memory Store ............................ SUCCESS [0.018s]
[INFO] Killer App XStream Store ........................... SUCCESS [0.018s]
[INFO] Killer App Core .................................... SUCCESS [0.016s]
[INFO] Killer App Commandline Interface ................... SUCCESS [0.102s]
[INFO] Killer App Web WAR ................................. SUCCESS [3.103s]
[INFO] ------------------------------------------------------------------------
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 6 seconds
[INFO] Finished at: Tue Mar 10 17:50:47 NZDT 2009
[INFO] Final Memory: 10M/18M
[INFO] ------------------------------------------------------------------------
```

Reactor Dependencies

- What is the Reactor?
  - The container of modules in a multi-module project
  - Helps resolve class dependencies between modules
  - Find sibling dependencies *before* they are placed in the repository, e.g

The **package** phase looks for JAR files in the sibling target directories

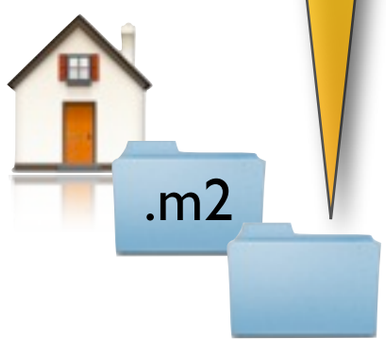


$ mvn package

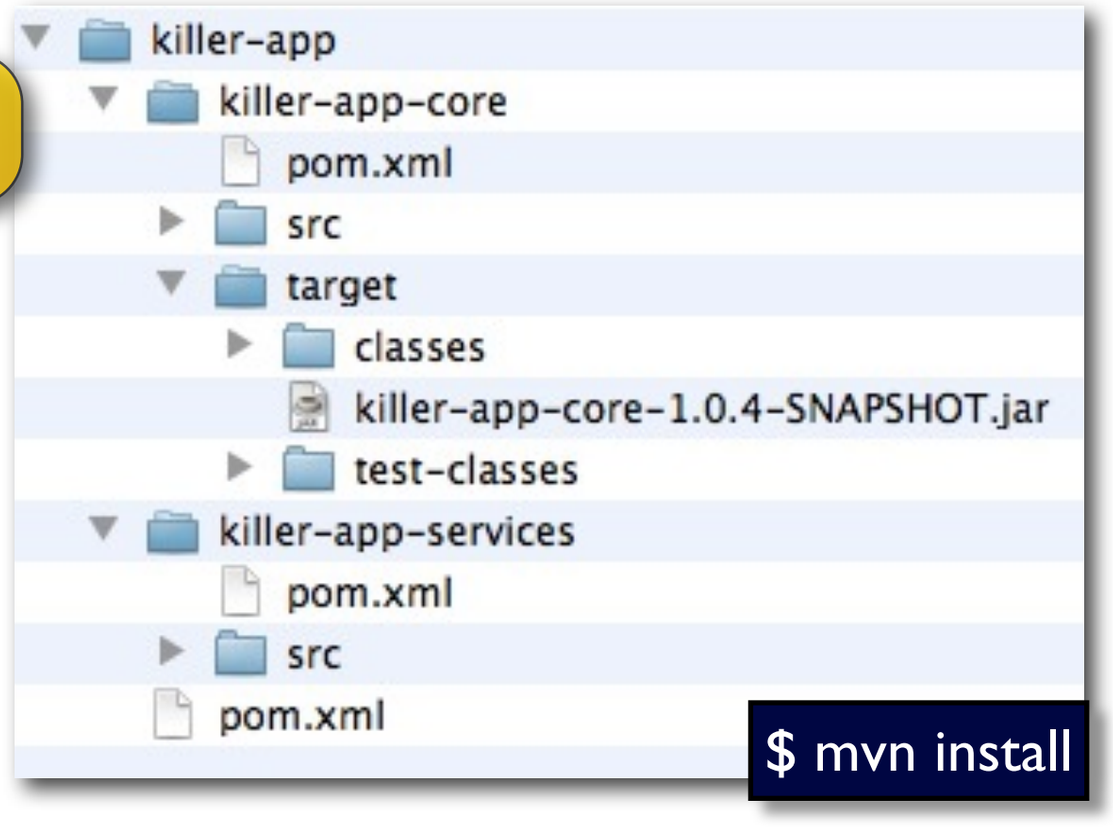
Reactor Dependencies

- What is the Reactor?
  - The container of modules in a multi-module project
  - Helps resolve class dependencies between modules
  - Find sibling dependencies *before* they are placed in the repository, e.g

The **install** phase looks for JAR files in the local repository



~/.m2/repository



Aggregation and Inheritance

- Aggregation and Inheritance - two distinct concepts

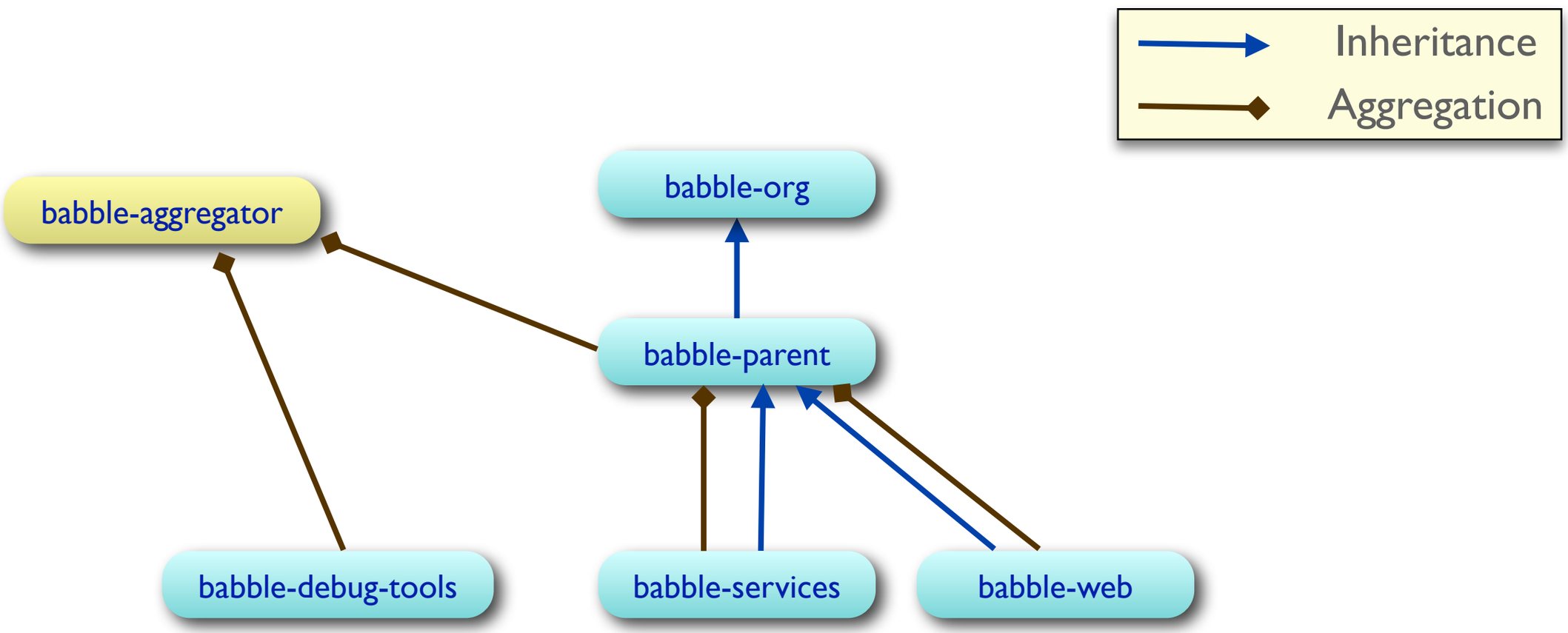
Aggregation <module>	Inheritance <parent>
• Container manages modules, sequence • Top-down (modules unaware of container) • Defined in the container	• A parent passes its values to its children • Bottom-up (parent unaware of children) • Defined in the child project

Aggregation and Inheritance

- Aggregation and inheritance are not always symmetrical
  - A container project (the one declaring a <module> list) is not always a <parent>
  - A <module> is not always a child

  - Parent projects contain inherited values or behavior
  - Container projects (declaring <module> lists) co-ordinate build sequence of multiple projects

Aggregation and Inheritance



Questions

- Why inheritance?
- Why modules?
- When would they not be symmetric?

labs

- Lab 9 - Creating a multi-module project

Survey of Plugins

A Survey of Useful Plugins

- There are many useful Maven plugins available
  - Reporting
  - Code Generation
  - Scripting
  - Testing
  - ...

A Survey of Useful Plugins

- Plugins come from different sources, e.g.

Maven central (http://maven.apache.org/plugins)

Maven – Available Plugins

http://maven.apache.org/plugins/index.html

Main
Welcome
Get Maven
Download
Release Notes (2.2.1)
Release Notes (2.0.10)
License
IDE Integration
Eclipse
Netbeans
About Maven
What is Maven?
Features
FAQ (official)
FAQ (unofficial)
Powered By
Documentation
Maven Plugins
Index (category)
Running Maven
User Centre
Plugin Developer Centre
Maven Repository Centre
Maven Developer Centre
Books and Resources
Wiki
Community
Community Overview
How to Contribute
Maven Repository
Getting Help
Issue Tracking
Source Repository
The Maven Team
Project Documentation
Project Information

Available Plugins

Maven is - at its heart - a plugin execution framework; all work is done by plugins. Looking for a specific goal to execute? This page lists the core plugins and others. There are the build and the reporting plugins:

- **Build plugins** will be executed during the build and they should be configured in the <build/> element from the POM.
- **Reporting plugins** will be executed during the site generation and they should be configured in the <reporting/> element from the POM.

Supported By The Maven Project

To see the most up-to-date list browse the Maven repository at http://repo1.maven.org/maven2/, specifically the org/apache/maven/plugins subfolder. *(Plugins are organized according to a directory structure that resembles the standard Java package naming convention)*

Plugin	Type*	Version	Release Date	Description	Source Repository	Issue Tracking
Core plugins				**Plugins corresponding to default core phases (ie. clean, compile). They may have muliple goals as well.**		
clean	B	2.4	2010-01-15	Clean up after the build.	SVN	JIRA
compiler	B	2.1	2010-01-02	Compiles Java sources.	SVN	JIRA
deploy	B	2.5	2009-12-24	Deploy the built artifact to the remote repository.	SVN	JIRA
failsafe	B	2.5	2010-01-17	Run the Junit integration tests in an isolated classloader.	SVN	JIRA
install	B	2.3	2009-03-25	Install the built artifact into the local repository.	SVN	JIRA
resources	B	2.4	2009-08-25	Copy the resources to the output directory for including in the JAR.	SVN	JIRA
site	B	2.1	2009-12-21	Generate a site for the current project.	SVN	JIRA
surefire	B	2.5	2010-01-17	Run the Junit unit tests in an isolated classloader.	SVN	JIRA
verifier	B	1.0-beta-1	2006-05-07	Useful for integration tests - verifies the existence of certain conditions.	SVN	JIRA

Codehaus (http://mojo.codehaus.org/plugins.html)

Mojo – Plugins

http://mojo.codehaus.org/plugins.html

Codehaus > Mojo | Maven | Last Published: 2010-01-19

Mojo
Introduction
Plugins
FAQ
Contribution
Reporting an Issue
Submitting a Patch
Submitting a Plugin
Development
Development Guidelines
Development Process
Performing a Release
The Sandbox
What is the Sandbox
Using Sandbox Plugins
Sub-projects
Animal sniffer
API Signatures
Project Information
Source Code
Mailing Lists
Internet Relay Chat
Issue Tracking

Plugins

Depending on their maturity, the available plugins are organized into groups as shown below. While using a pre-release or sandbox plugin might seem risky, it is your chance to provide feedback to the community such that the plugin can eventually reach production state.

Production Plugins

The following plugins have been released and are generally available for production use.

Plugin	Version	Description
Application Server		*Plugins that support integration with application servers.*
jboss	1.4	Control a JBoss server from Maven.
jboss-packaging	2.1.1	Packages various JBoss deployable archive files including SAR, HAR, ESB and Spring.
was6	1.1.1	IBM WebSphere Application Server 6.1 support.
Generators / Compilers / Translators		*Plugins providing support for code generation, compilation and translation.*
antlr	2.0	Process ANTLRv2 grammars.
antlr3	1.0	Process ANTLRv3 grammars.
aspectj	1.3	Weaves aspects with AspectJ.
axistools	1.3	Axis wsdl2java and java2wsdl support.
castor	2.0	Source generation with Castor XML.
commons-attributes	1.0	Source generation for Commons Attributes.
GWT	1.2	Google Web Toolkit.
hibernate3	2.2	Integration with Hibernate 3.x

A Survey of Useful Plugins

- Reporting Plugins
  - **javadoc** - *Generate API documentation*
  - **surefire-report** - *Test results*
  - **checkstyle** - *Coding standards and metrics*
  - **cobertura** - *Open source code coverage tool*
  - **findbugs** - *Coding errors*
  - ...

A Survey of Useful Plugins

- Code Generation Plugins
  - **hibernate3** - *Generate Hibernate database schemas, mapping files, etc...*
  - **jaxb2** - *XML to Java using Jaxb2*
  - **xmlbeans** - *XML Schemas to Java binding via XMLBeans*
  - **xdoclet** - *XDoclet integration with Maven*
  - **...**

A Survey of Useful Plugins

- Scripting Plugins
  - **exec** - *Execute an arbitrary Java program or command line instruction*
  - **antrun** - *Execute Ant scripts from your pom file*
  - **gmaven** - *Embed Groovy scripting in your pom file*
  - **sql** - *Execute SQL scripts*
  - **...**

A Survey of Useful Plugins

- Testing Plugins
  - **selenium** - *Automated web testing tool*
  - **easyb** - *Groovy-based Behavior-Driven Development tool*
  - **fitness** - *Integrate fitnesse tests*
  - ...

A Survey of Useful Plugins

- Other Plugins
  - **assembly** - *Build a binary distribution (e.g. source code + binaries)*
  - **cargo** - *Automated deployment to an application server*
  - **versions** - *Helps manage dependency versions in your poms*
  - ...

Maven 3

Maven 3

- Drop in replacement
- 400+ integration tests
- More warnings, error messages
- Performance tuned
- Library-focus
- Symmetry between CLI and GUIs
- Many derived tools
- Polyglot Maven
- Maven Shell

Conclusion

A review

- Coordinates
- Lifecycle
- Customizing the build
- Inheritance
- Modules
- Cool Plugins
- Maven 3

Thank you!