Maven 3

Evgeny Borisov

bsevgeny@gmail.com

About myself

Owner of startup “democracy”

Partner of Trainologic

Partner of JFrog

Consulting

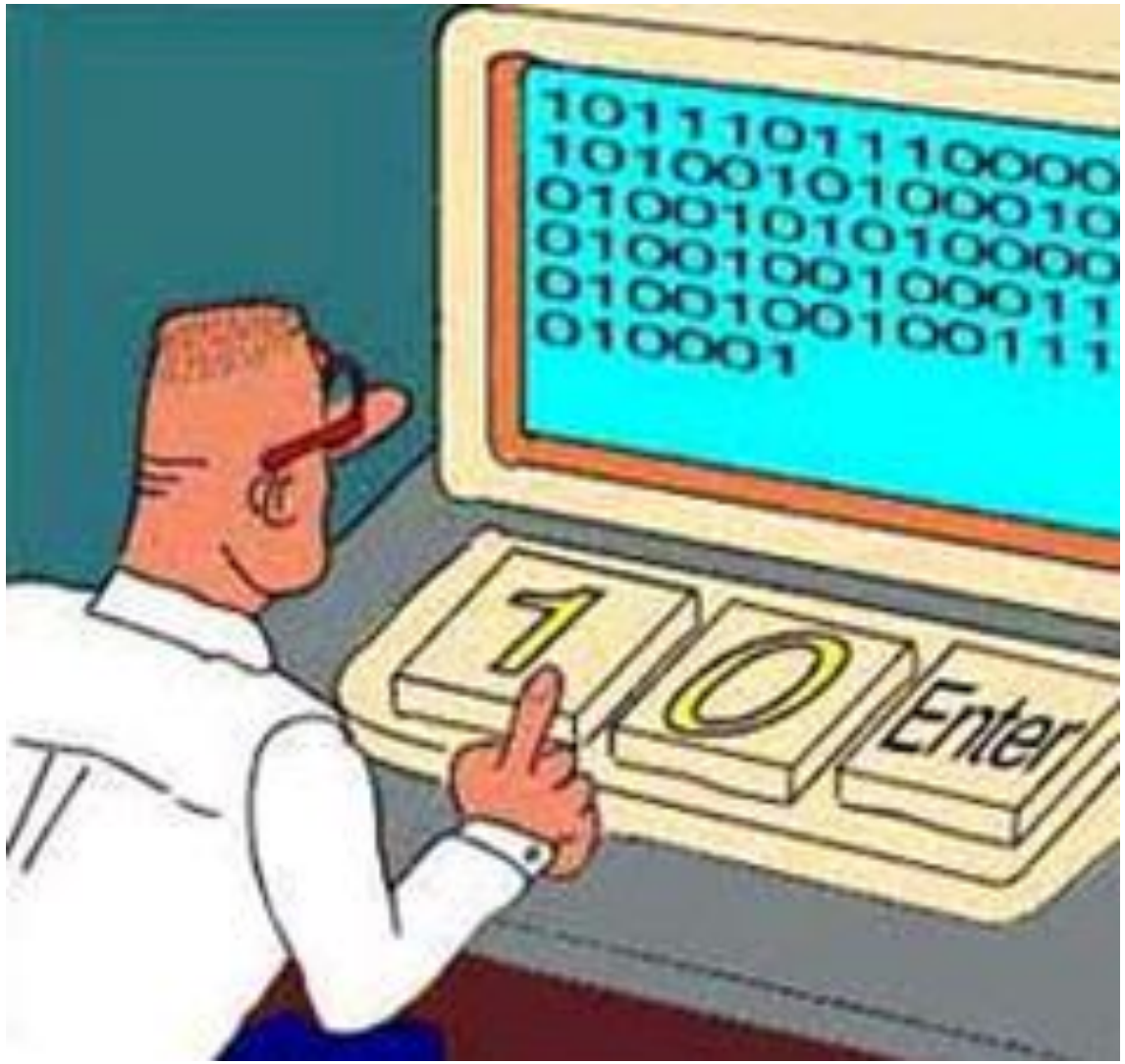
Lecturing

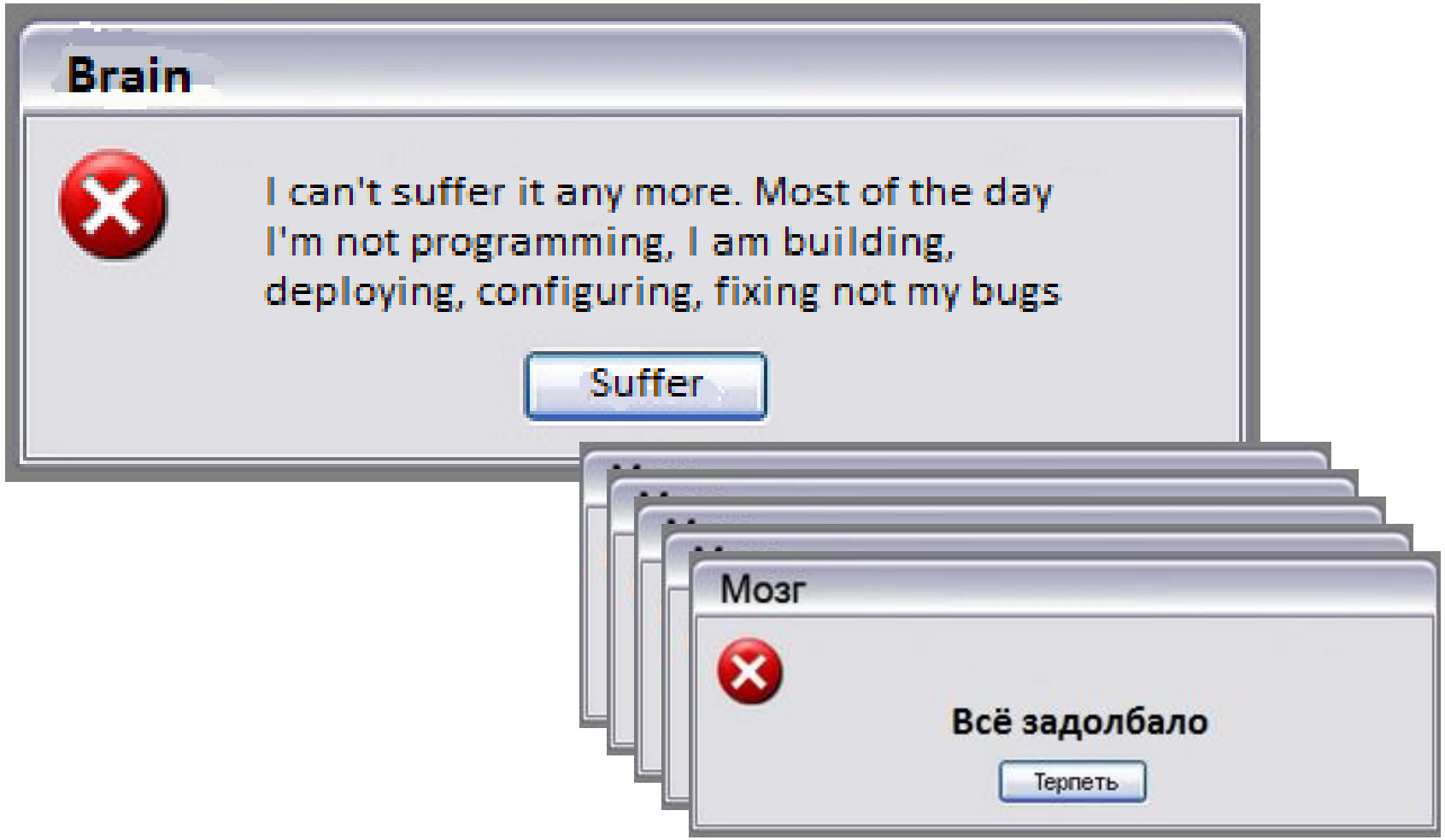
Writing courses

Writing code

I heard the course about maven, why do we need it at all?

Good to be programmer…




Brain
I can't suffer it any more. Most of the day I'm not programming, I am building, deploying, configuring, fixing not my bugs
Suffer
Мозг
Всё задолбало
Терпеть

You start crooked, everything is crooked

What are the tasks of build today?

- VCS
- CI scripts & different xml configurations
- Dependency management
- Deployment (clustering)
- Unit and sanity Tests
- Documentation
- …


Grandson will not come today, he sent mail that the build failed.

Which build tools do you know?

I use…

1. Gradle
2. Maven
3. Ant
4. Ivy
5. I build with my hands…

Declarative & Imperative

Declarative & Imperative

- ANT – Imperative
- Maven - Declarative

Not fully

maven

Table of Contents

Maven and Procedural Build Tools

But Ant is Working Just Fine!

- Ant provides building blocks for a toolset - Maven provides a working tool
- Ant has no build lifecycle, repositories, standard project layout, dependency management … you have to do it by yourself!
- Ant scripts quickly become complex
- Complex scripts are not reusable – duplication

MyApp – Ant Build File – 141 lines

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="my-app" default="all">


  <property file="my-app.properties"/>

  <taskdef name="javac2" classname="com.intellij.ant.Javac2"/>

  <property name="compiler.debug" value="on"/>
  <property name="compiler.generate.no.warnings" value="off"/>
  <property name="compiler.args" value=""/>
  <property name="compiler.max.memory" value="128m"/>
  <patternset id="ignored.files">
    <exclude name="**/CVS/**"/>
    <exclude name="**/SCCS/**"/>
    <exclude name="**/RCS/**"/>
    <exclude name="**/rcs/**"/>
    <exclude name="**/.DS_Store/**"/>
    <exclude name="**/.svn/**"/>
    <exclude name="**/vssver.scc/**"/>
    <exclude name="**/vssver2.scc/**"/>
  </patternset>
  <patternset id="compiler.resources">
    <exclude name="**/?*.java"/>
  </patternset>

  <property name="jdk.bin.1.5" value="${jdk.home.1.5}/bin"/>
  <path id="jdk.classpath.1.5">
    <fileset dir="${jdk.home.1.5}">
      <include name="jre/lib/charsets.jar"/>
      <include name="jre/lib/deploy.jar"/>
      <include name="jre/lib/javaws.jar"/>
      <include name="jre/lib/jce.jar"/>
      <include name="jre/lib/jsse.jar"/>
      <include name="jre/lib/plugin.jar"/>
      <include name="jre/lib/rt.jar"/>
      <include name="jre/lib/ext/dnsns.jar"/>
      <include name="jre/lib/ext/localedata.jar"/>
      <include name="jre/lib/ext/sunjce_provider.jar"/>
      <include name="jre/lib/ext/sunpkcs11.jar"/>
    </fileset>
  </path>

  <property name="project.jdk.home" value="${jdk.home.1.5}"/>
  <property name="project.jdk.bin" value="${jdk.bin.1.5}"/>
  <property name="project.jdk.classpath" value="jdk.classpath.1.5"/>

  <dirname property="module.my-app.basedir" file="${ant.file}"/>


  <property name="module.jdk.home.my-app" value="${project.jdk.home}"/>
  <property name="module.jdk.bin.my-app" value="${project.jdk.bin}"/>
  <property name="module.jdk.classpath.my-app" value="${project.jdk.classpath}"/>

  <property name="compiler.args.my-app" value="${compiler.args}"/>

  <property name="my-app.output.dir" value="${module.my-app.basedir}/target/classes"/>
  <property name="my-app.testoutput.dir" value="${module.my-app.basedir}/target/test-classes"/>

  <path id="my-app.module.bootclasspath">
    <!-- Paths to be included in compilation bootclasspath -->
  </path>

  <path id="my-app.module.classpath">
    <path refid="${module.jdk.classpath.my-app}"/>
    <pathelement location="C:/Documents and Settings/Dror Bereznitsky/.m2/repository/junit/junit/3.8.1/junit-3.
  </path>


  <patternset id="excluded.from.module.my-app">
    <patternset refid="ignored.files"/>
  </patternset>

  <patternset id="excluded.from.compilation.my-app">
    <patternset refid="excluded.from.module.my-app"/>
  </patternset>

  <path id="my-app.module.sourcepath">
    <dirset dir="${module.my-app.basedir}">
      <include name="src/main/java"/>
    </dirset>
  </path>

  <path id="my-app.module.test.sourcepath">
    <dirset dir="${module.my-app.basedir}">
      <include name="src/test/java"/>
    </dirset>
  </path>


  <target name="compile.module.my-app" depends="compile.module.my-app.production,compile.module.my-app.tests" description="

  <target name="compile.module.my-app.production" description="Compile module my-app; production classes">
    <mkdir dir="${my-app.output.dir}"/>
    <javac2 destdir="${my-app.output.dir}" debug="${compiler.debug}" nowarn="${compiler.generate.no.warnings}" memorymaxim
      <compilerarg line="${compiler.args.my-app}"/>
      <bootclasspath refid="my-app.module.bootclasspath"/>
      <classpath refid="my-app.module.classpath"/>
      <src refid="my-app.module.sourcepath"/>
      <patternset refid="excluded.from.compilation.my-app"/>
    </javac2>

    <copy todir="${my-app.output.dir}">
      <fileset dir="${module.my-app.basedir}/src/main/java">
        <patternset refid="compiler.resources"/>
        <type type="file"/>
      </fileset>
    </copy>
  </target>

  <target name="compile.module.my-app.tests" depends="compile.module.my-app.production" description="compile module my-app;
    <mkdir dir="${my-app.testoutput.dir}"/>
    <javac2 destdir="${my-app.testoutput.dir}" debug="${compiler.debug}" nowarn="${compiler.generate.no.warnings}" memoryma
      <compilerarg line="${compiler.args.my-app}"/>
      <classpath refid="my-app.module.classpath"/>
      <classpath location="${my-app.output.dir}"/>
      <src refid="my-app.module.test.sourcepath"/>
      <patternset refid="excluded.from.compilation.my-app"/>
    </javac2>

    <copy todir="${my-app.testoutput.dir}">
      <fileset dir="${module.my-app.basedir}/src/test/java">
        <patternset refid="compiler.resources"/>
        <type type="file"/>
      </fileset>
    </copy>
  </target>

  <target name="clean.module.my-app" description="cleanup module">
    <delete dir="${my-app.output.dir}"/>
    <delete dir="${my-app.testoutput.dir}"/>
  </target>

  <target name="init" description="Build initialization">
    <!-- Perform any build initialization in this target -->
  </target>

  <target name="clean" depends="clean.module.my-app" description="cleanup all"/>

  <target name="all" depends="init, clean, compile.module.my-app" description="build all"/>
</project>
```

141 Lines of code !!!

MyApp - Maven Build File – 17 lines

17 Lines of code

```
1   <project>
2     <modelVersion>4.0.0</modelVersion>
3     <groupId>com.mycompany.app</groupId>
4     <artifactId>my-app</artifactId>
5     <packaging>jar</packaging>
6     <version>1.0-SNAPSHOT</version>
7     <name>my-app</name>
8     <url>http://maven.apache.org</url>
9     <dependencies>
10      <dependency>
11        <groupId>junit</groupId>
12        <artifactId>junit</artifactId>
13        <version>3.8.1</version>
14        <scope>test</scope>
15      </dependency>
16    </dependencies>
17  </project>
```

Maven vs. Ant

Our experience shows

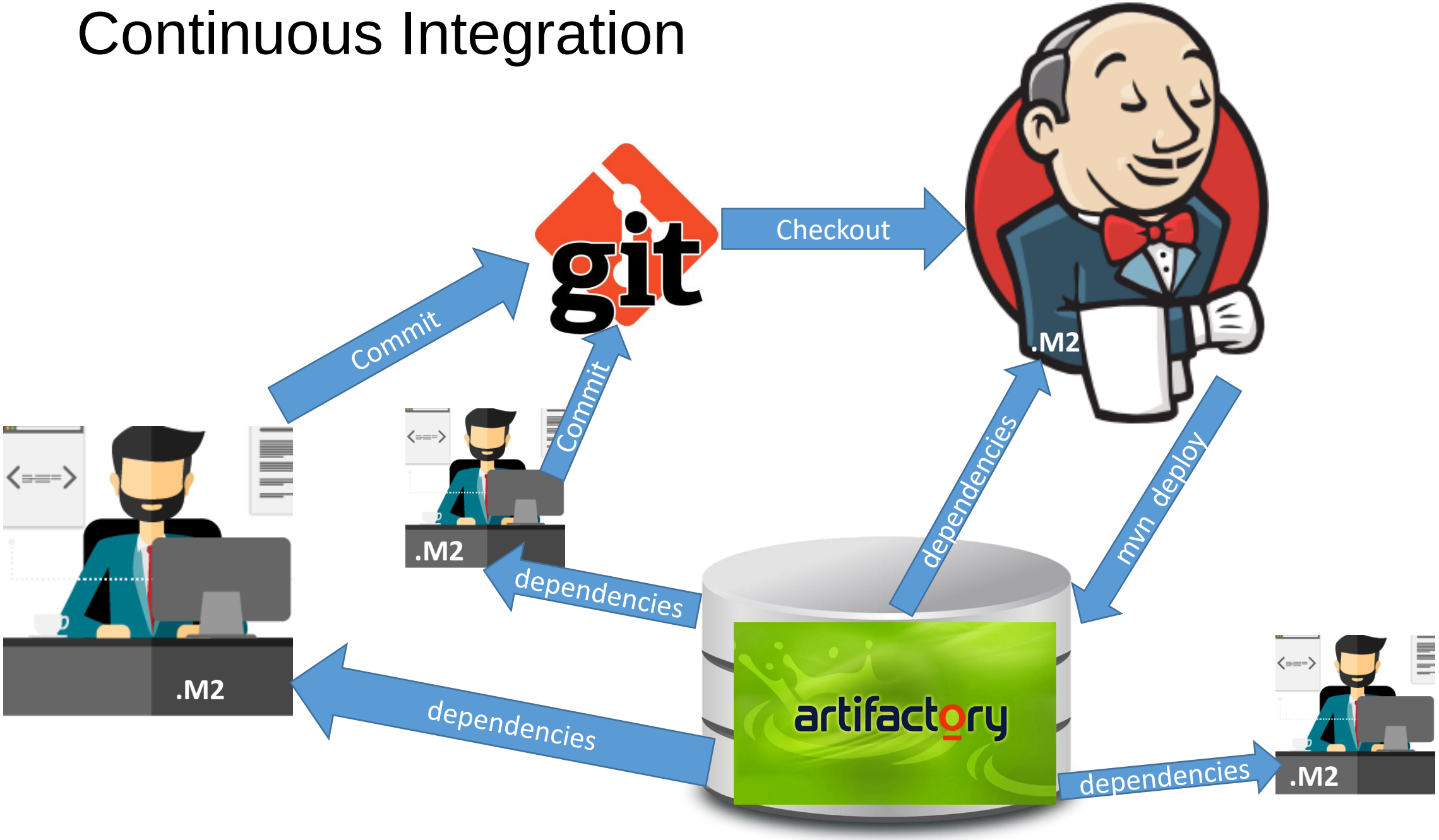
1:30 ratio in the number of

LOC between Maven2 and Ant.

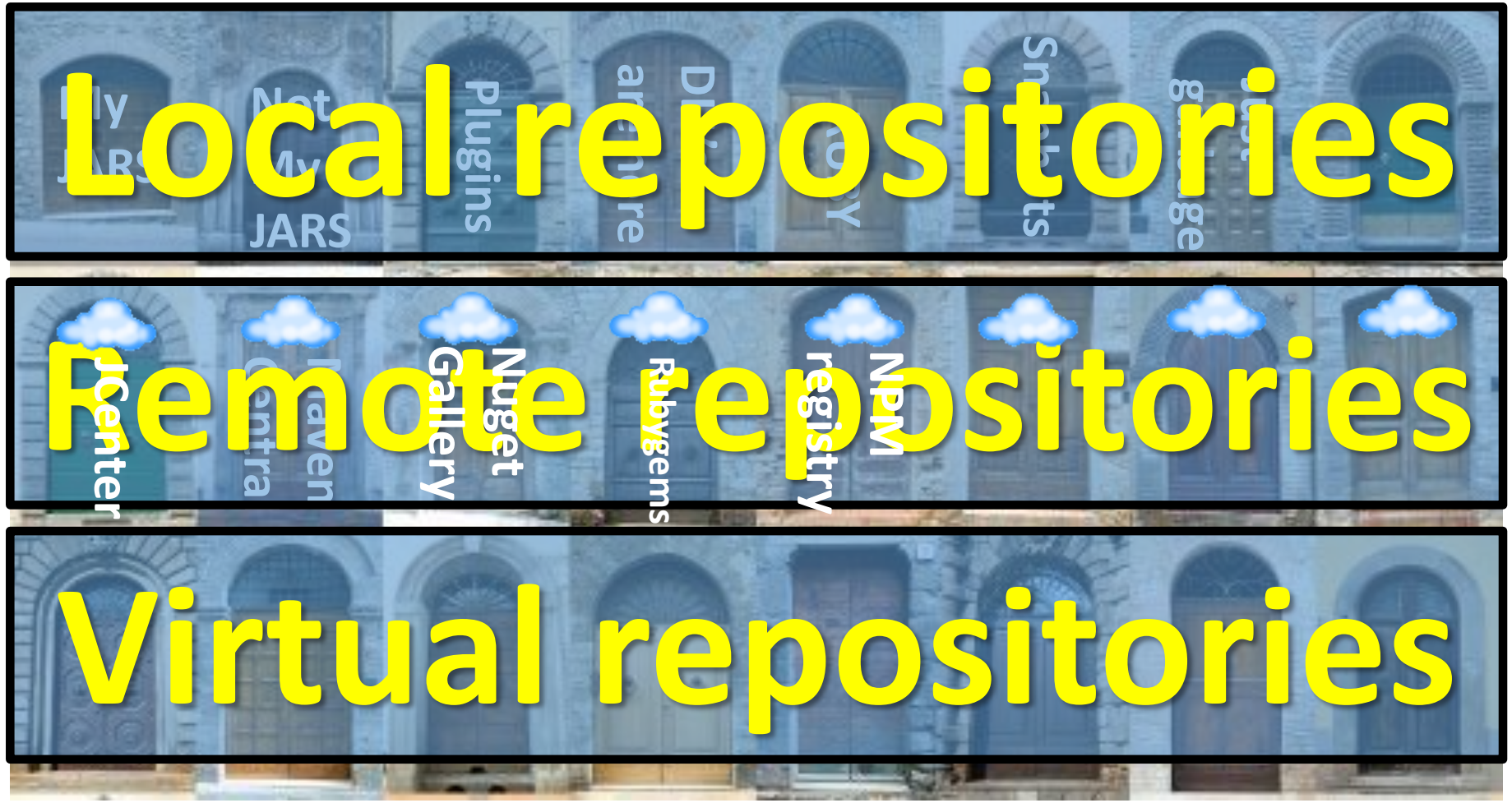
CI in 3 sentences

- Commit early, Commit often
- Build on **every** commit
- Test on **every** build

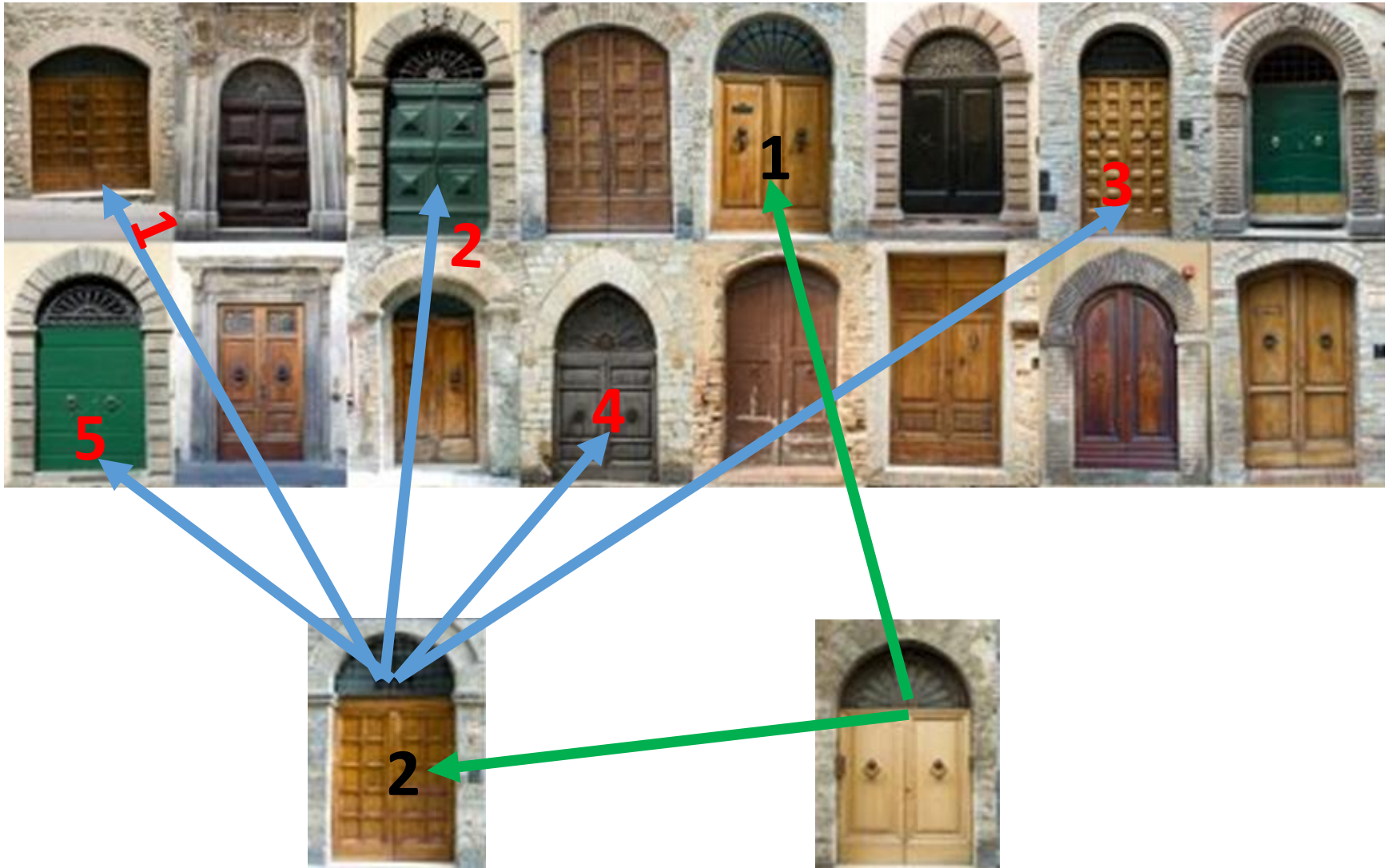
Continuous Integration



Artifacfory



Virtual Repository



What Maven does?

- Fetch dependencies
- Configure classpath and compile
- Run tests
- Create archive (packaging…) jar / war …
- Deploy artifact to artifactory (don’t do that locally)
- Maven has a lot of standards known in Java world

What Does Maven Standardize?

- Build lifecycle and order
- Project layout
- 3rd party dependencies storage
- Dependency resolution
- Transitive dependencies
- Project documentation and reports
- A lot of plugins

The Project Object Model (POM)

Maven's Declarative Nature

- To provide standardization Maven takes a Declarative Approach
  - In contrast to procedural build tools, such as Ant
- Similar to a complete development framework
- Project metadata is provided using a declarative XML document
  - Metadata about project's layout, execution, documentation etc.
- This declarative xml-based metadata is known as the POM – Project Object Model

The POM

- *What* is being built, not *how*
- Contains detailed metadata information about the project:
  - Versioning
  - Configuration management
  - Dependencies
  - Project structure
  - Application and testing resources
  - Developers/Contributors
  - Issue tracking system
  - Etc., etc….

Sample POM

```xml
<?xml version="1.0" encoding="UTF-8"?>
<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/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.naya</groupId>
    <artifactId>naya-microservice</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>

    <dependencies>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.17</version>
        </dependency>
    </dependencies>
</project>
```

The POM (contd.)

- POMs may inherent from each other
- Most metadata adheres to a W3C XML Schema (.xsd)
- Order of declaration between elements of the same level is not important in most cases
  - Uses xsd:all

POM Properties

- May contain dynamic expressions: `${some.expression}`
- All elements in the model of the POM can be retrieved through properties, using XPATH-like syntax, e.g.:

```
${project.scm.url}
${project.version}
${settings.localRepository}
```

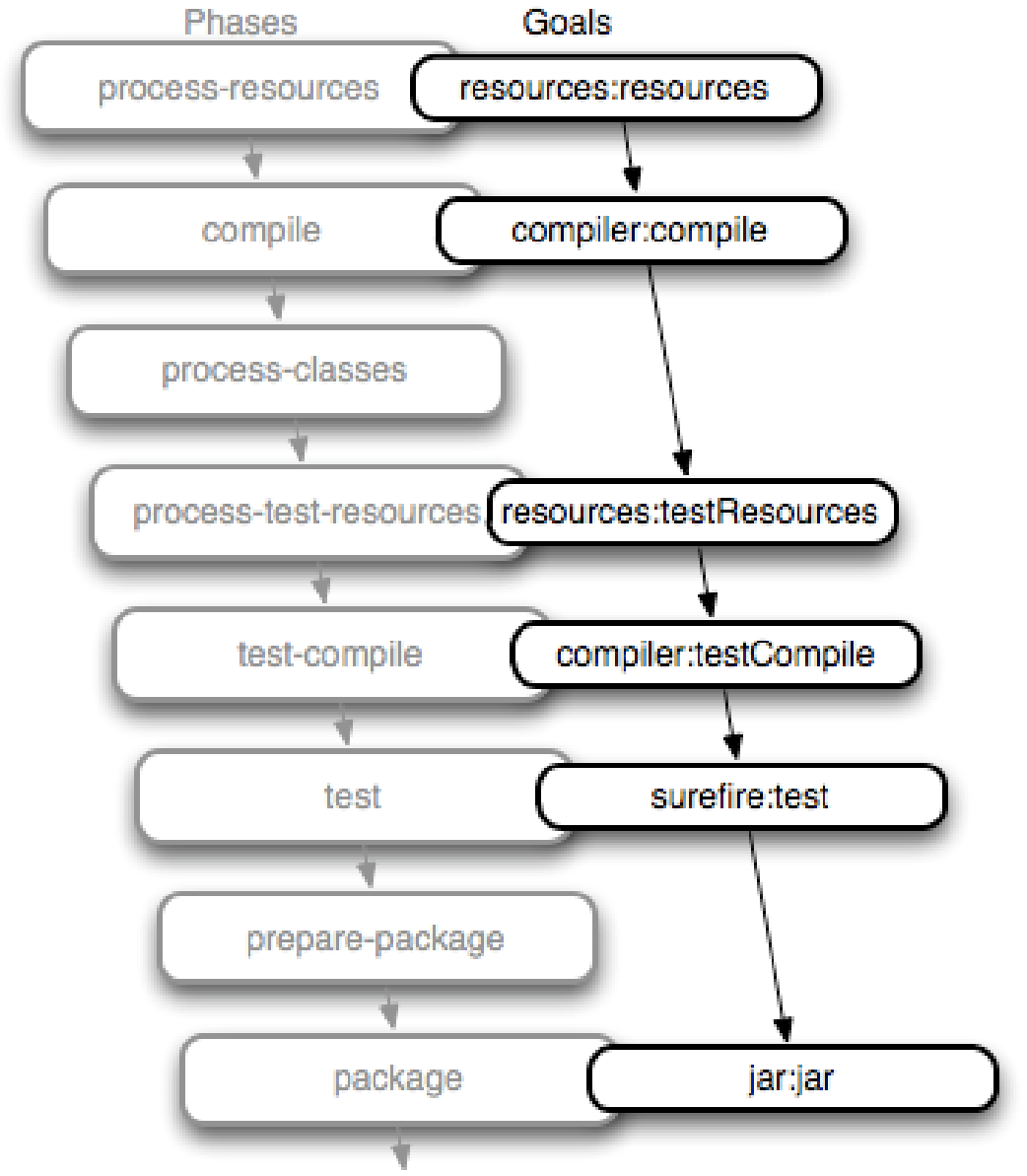
- Normally the `project.` prefix can be omitted

The Build Lifecycle

Build Lifecycle and Phases

- Every build process/lifecycle is made up of several well-known phases
- The exact type of lifecycle phases and their order may vary according to the desired build output (project type)
- At every phase certain tasks are performed
- Such tasks are written as Maven plugins

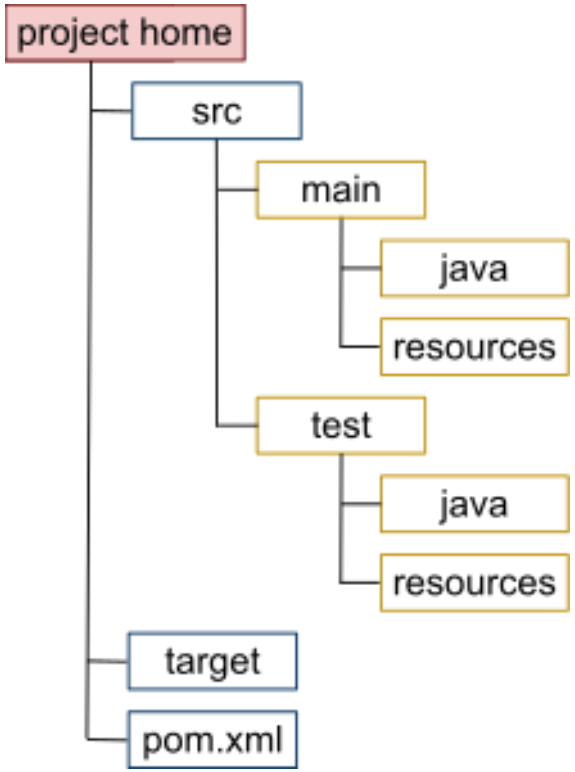
Maven default lifecycle



Note: There are more phases than shown above, this is a partial list

Standard Project Layout

Maven's Project Layout



- Maven standardizes a recommended project layout
- May be overridden in the POM
  - But is usually not desirable

```
<build>
  <sourceDirectory>../../src</sourceDirectory>
  <testSourceDirectory>../../test</testSourceDirectory>
</build>
```

- The standard layout is well-known and expected by any Maven user

Running Maven

The Maven Command Line

- Maven is run through a simple CLI
- Under `$M2_HOME/bin`

```
mvn -help
```

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

Options:
 -C,--strict-checksums          Fail the build if checksums don't match
 -c,--lax-checksums             Warn if checksums don't match
 -P,--activate-profiles         Comma-delimited list of profiles to
                                activate
 -ff,--fail-fast                Stop at first failure in reactorized builds
 -fae,--fail-at-end             Only fail the build afterwards; allow all
                                non-impacted builds to continue
 -B,--batch-mode                Run in non-interactive (batch) mode
 -fn,--fail-never               NEVER fail the build, regardless of project
                                result
 -up,--update-plugins           Synonym for cpu
 -N,--non-recursive             Do not recurse into sub-projects
 -npr,--no-plugin-registry      Don't use ~/.m2/plugin-registry.xml for
                                plugin versions
 -U,--update-snapshots          Update all snapshots regardless of
                                repository policies
 -cpu,--check-plugin-updates    Force upToDate check for any relevant
                                registered plugins
 -npu,--no-plugin-updates       Suppress upToDate check for any relevant
                                registered plugins
 -D,--define                    Define a system property
 -X,--debug                     Produce execution debug output
 -e,--errors                    Produce execution error messages
 -f,--file                      Force the use of an alternate POM file.
 -h,--help                      Display help information
 -o,--offline                   Work offline
 -r,--reactor                   Execute goals for project found in the
                                reactor
 -s,--settings                  Alternate path for the user settings file
 -v,--version                   Display version information
```

Maven Command Line Parameters

- POM file to execute is assumed to be `./pom.xml`
- Can be overridden with `-f`, for example:

```
mvn -f build/mypom.xml ...
```

- Order of parameters and "`-`" flags is not important
- Can pass VM arguments with `-Dmy.arg=value`
  - These are later available as Maven properties: `${my.arg}`

Command Line Phases & Goals

- Maven can either execute an entire phase with all its bound plugin goals

```
mvn install
```

- Or a specific plugin goal
  - The plugin can internally cause a run of a series of preliminary phases

- Can be co

```
mvn plugin-alias:goal
```

```
mvn clean source:jar install
```

More Info & Troubleshooting

- We can get Maven to be extremely verbose

```
mvn -X ...
```

  - Useful mainly in tracking dependency resolution issues
- If a build error occurs, we can ask for the stack trace of the Java exception inside Maven

```
mvn ... -e
```

Lab 1: First Maven Project

I have a problem! Build failure

Proxy!!!
Inside your maven settings.xml

```
<proxies>
    <active>true</active>
    <protocol>http</protocol>
    <host>192.168.101.52</host>
    <port>8080</port>

  </proxy>
 </proxies>
```

This must be a per of settings tag

Artifacts &
Dependency Management

Artifacts

- Maven uses the notion of Artifact
- Artifacts may be any resource used by the build process or produced by it
- My refer to:
  - Build outcomes
  - Build dependencies (libs)
  - Other resources
- Retrieved from artifact repositories

Artifact Identification

- An artifact is identified by its:
  - Group ID
  - Artifact ID
  - Version
  - Classifier (optional)
  - Extension/Type
- For example:
  - `javax.servlet`
  - `servlet-api`
  - `2.5`
  - `sources`
  - `jar`

Dependencies

- Special form of artifacts that are required by the build process
- Normally JAR libraries in Java-based builds
- Also include a “scope” attribute

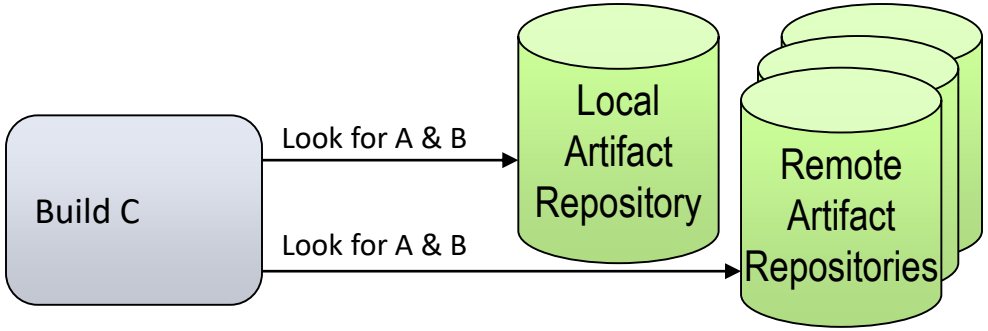
```
<dependency>
        <groupId>org.apache.wicket</groupId>
        <artifactId>wicket</artifactId>
        <version>1.3-incubating-SNAPSHOT</version>
        <type>jar</type>
        <scope>compile</scope>
</dependency>
```

Dependency Scopes

- Scope defines where in the build phases dependencies are available.
- 5 available scopes:
  - **compile** – the dependency is available in all classpaths. The default scope.
  - **provided** – the dependency is provided by the JDK or Container.
  - **runtime** - the dependency is not required for compilation. It is in the runtime and test classpaths, but not the compile classpath.
  - **test** – the dependency is available only in the test compilation and execution classpath.
  - **system** - similar to provided except that the JAR path has to be stated explicitly.

Dependency Resolution

- During the build process Maven will try to resolve the build dependencies
- When looking for dependency artifacts Maven consults 2 types of repositories:
  - A local repository on the user's machine
  - A set of remote repositories



Dependency Version

- A dependency version can be either:
  - A release version
  - A snapshot version
  - A version range
- Versions affect the dependency resolution process

```
<dependency>
        <groupId>org.apache.wicket</groupId>
        <artifactId>wicket</artifactId>
        <version>1.3-incubating-SNAPSHOT</version>
</dependency>
```

Dependency Resolution by Version

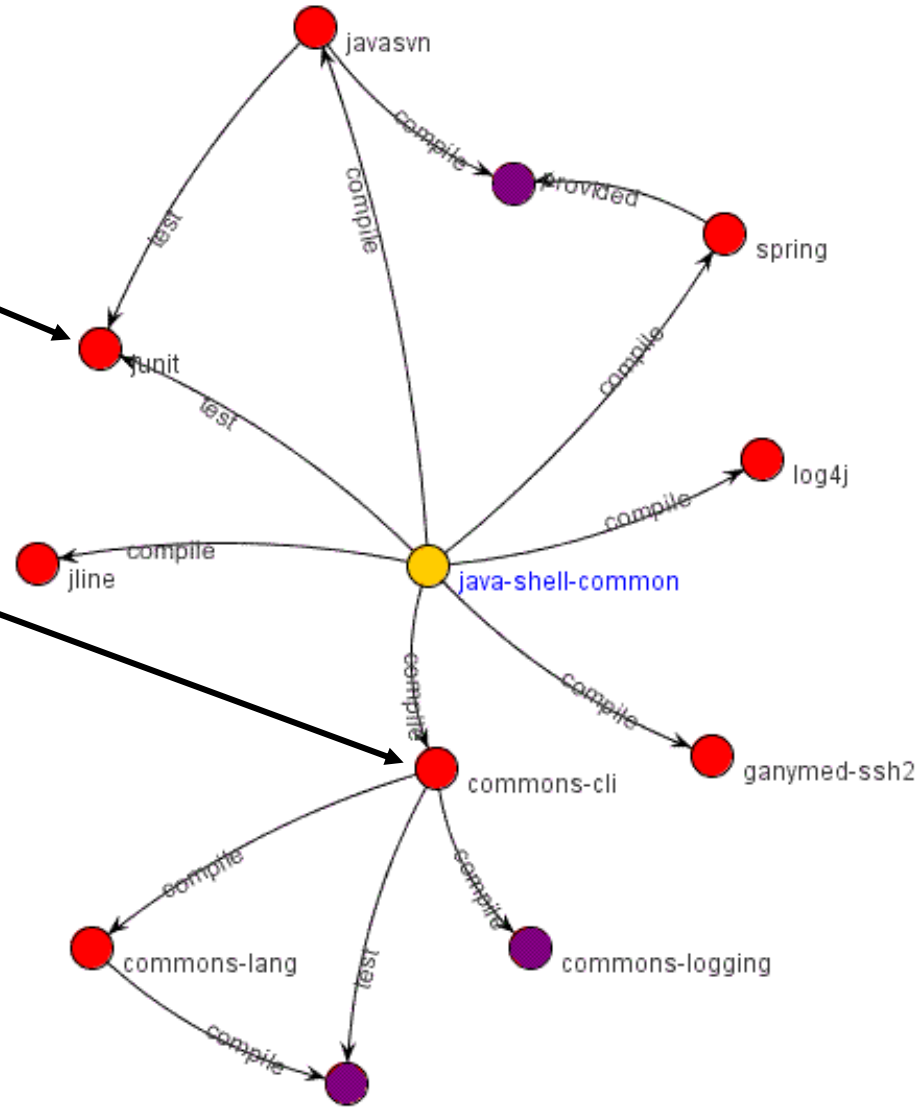
- Release versions are retrieved only once from the repository
- Snapshot versions are assumed to be changing
  - Automatically updated daily by default
  - Can force update with `mvn -U`
- Version range is a concrete release version case:

`(,1.0]`	v <= 1.0
`[1.2,1.3]`	1.2 <= v <=1.3
`[1.0,2.0)`	1.0 <= v < 2.0
`[1.5,)`	1.5 <= v
`(,1.1),(1.1,)`	v != 1.1

  - Don’t use version ranges!

Transitive Dependency

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>commons-cli</groupId>
        <artifactId>commons-cli</artifactId>
        <version>1.0</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>commons-launcher</groupId>
        <artifactId>commons-launcher</artifactId>
        <version>1.1</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.8</version>
        <scope>compile</scope>
    </dependency>
```



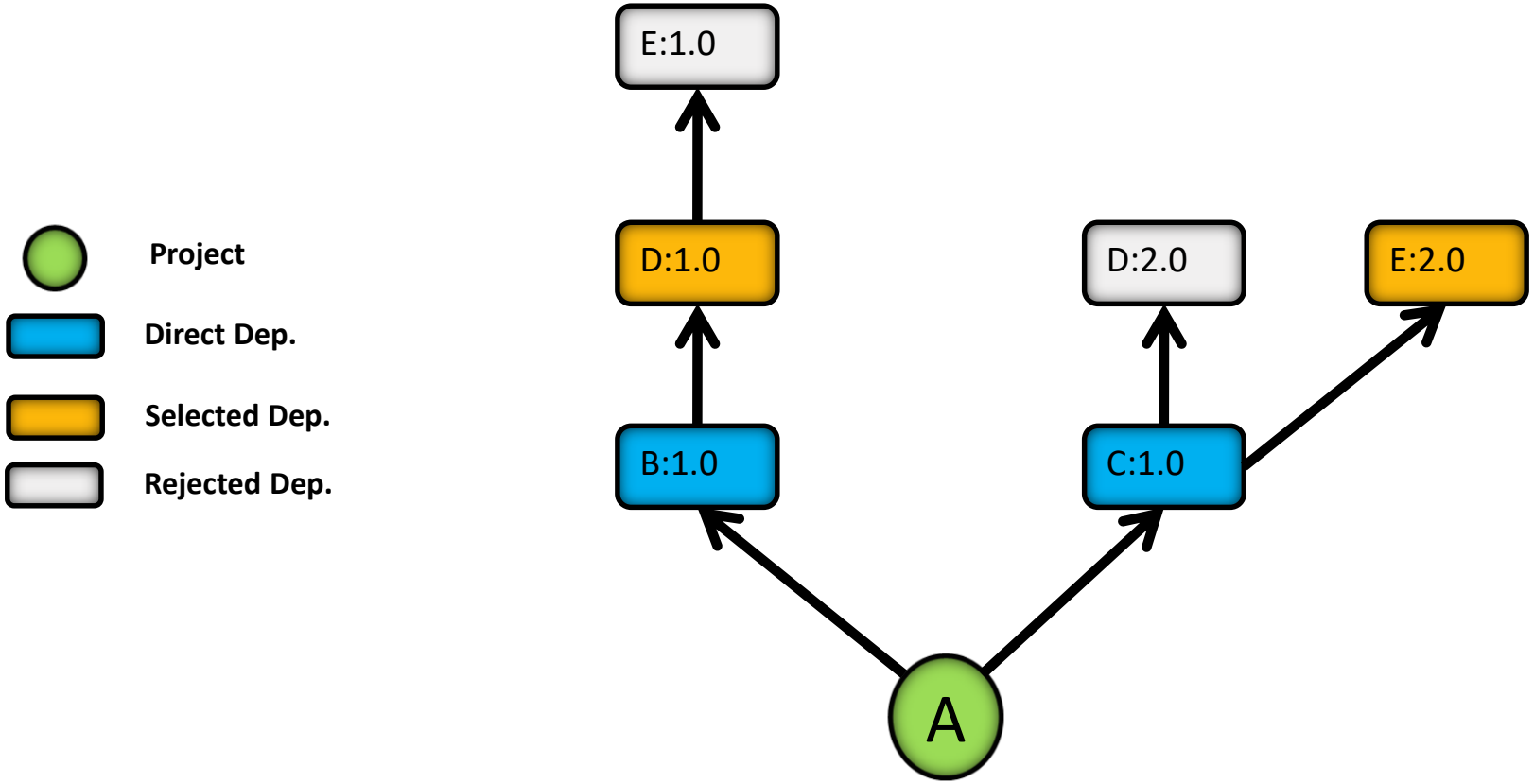
Transitivity Exclusion

- Can exclude dependencies from being transitively included

```
<dependency>
    <groupId>org.codehaus.plexus</groupId>
    <artifactId>plexus-container</artifactId>
    <version>1.0</version>
    <exclusions>
        <exclusion>
            <groupId>org.codehaus.plexus</groupId>
            <artifactId>plexus-utils</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Transitive Dependencies Conflicts

- Nearest dependency matching



How can I know what is the last version?

- http://search.maven.org/

Lab 2: Dependencies

Dependency Graph->Tree

- Transitive dependencies result in a graph
- The `dependencies` plugin can dump the graph as a tree:

```
mvn dependency:tree
```

```
C:\WINDOWS\system32\cmd.exe
[INFO]                org.apache.maven.doxia:doxia-sink-api:jar:1.0-alpha-9-SNAPSHOT:compile
[INFO]             org.apache.maven.wagon:wagon-http-lightweight:jar:1.0-beta-2:runtime
[INFO]                org.apache.maven.wagon:wagon-http-shared:jar:1.0-beta-2:runtime
[INFO]                   jtidy:jtidy:jar:4aug2000r7-dev:runtime
[INFO]             org.apache.maven.wagon:wagon-ssh-external:jar:1.0-beta-2:runtime
[INFO]             org.apache.maven.wagon:wagon-file:jar:1.0-beta-2:runtime
[INFO]             org.apache.maven:maven-plugin-api:jar:2.1-SNAPSHOT:compile
[INFO]             org.apache.maven:maven-monitor:jar:2.1-SNAPSHOT:compile
[INFO]             org.apache.maven:maven-error-diagnostics:jar:2.1-SNAPSHOT:compile
[INFO]             org.apache.maven:maven-plugin-registry:jar:2.1-SNAPSHOT:compile
[INFO]          org.apache.maven:maven-tools:jar:2.1-SNAPSHOT:compile
[INFO]       javax.xml.bind:jaxb-api:jar:2.0:compile
[INFO]          javax.activation:activation:jar:1.1:compile
[INFO]          javax.xml.bind:jsr173_api:jar:1.0:compile
[INFO]       org.apache.maven:maven-artifact-manager:jar:2.1-SNAPSHOT:compile
[INFO]          org.codehaus.plexus:plexus-container-default:jar:1.0-alpha-14:compile
[INFO]             org.codehaus.plexus:plexus-component-api:jar:1.0-alpha-13:compile
[INFO]          org.apache.maven:maven-repository-metadata:jar:2.1-SNAPSHOT:compile
[INFO]          org.apache.maven.wagon:wagon-provider-api:jar:1.0-beta-2:compile
[INFO]             org.codehaus.plexus:plexus-utils:jar:1.1:compile
[INFO]       org.apache.jackrabbit:jackrabbit-core:jar:1.2-SNAPSHOT:compile
[INFO]          concurrent:concurrent:jar:1.3.4:compile
[INFO]          org.apache.jackrabbit:jackrabbit-jcr-commons:jar:1.2-SNAPSHOT:compile
[INFO]          org.apache.geronimo.specs:geronimo-jta_1.0.1B_spec:jar:1.0.1:compile
[INFO]          org.apache.derby:derby:jar:10.2.1.6:compile
[INFO]          org.apache.lucene:lucene-core:jar:2.0.0:compile
[INFO]          org.apache.jackrabbit:jackrabbit-api:jar:1.2-SNAPSHOT:compile
[INFO]          org.slf4j:slf4j-log4j12:jar:1.0:compile
[INFO]          org.apache.jackrabbit:jackrabbit-text-extractors:jar:1.2-SNAPSHOT:compile
[INFO]             nekohtml:nekohtml:jar:0.9.4:compile
[INFO]                xerces:xercesImpl:jar:2.4.0:compile
[INFO]             poi:poi:jar:2.5.1-final-20040804:compile
[INFO]             org.textmining:tm-extractors:jar:0.4:compile
[INFO]             pdfbox:pdfbox:jar:0.6.4:compile
[INFO]       com.sun.xml.bind:jaxb-impl:jar:2.0.1:compile
[INFO]       commons-io:commons-io:jar:1.2:compile
[INFO]       org.apache.maven:maven-project:jar:2.1-SNAPSHOT:compile
[INFO]          org.apache.maven:maven-model:jar:2.1-SNAPSHOT:compile
[INFO]          org.apache.maven:maven-profile:jar:2.1-SNAPSHOT:compile
[INFO]          org.apache.maven:maven-settings:jar:2.1-SNAPSHOT:compile
[INFO]       org.acegisecurity:acegi-security:jar:1.0.2:compile
[INFO]          org.springframework:spring-remoting:jar:1.2.7:compile
[INFO]             org.springframework:spring-webmvc:jar:1.2.7:compile
[INFO]                org.springframework:spring-web:jar:1.2.7:compile
[INFO]             org.springframework:spring-dao:jar:1.2.7:compile
[INFO]                org.springframework:spring-context:jar:1.2.7:compile
[INFO]                   org.springframework:spring-aop:jar:1.2.7:compile
[INFO]          commons-codec:commons-codec:jar:1.2:compile
[INFO]          org.springframework:spring-support:jar:1.2.7:runtime
```

I want to put decency tree graph to a text file

- mvn dependency:tree > myFile.txt

Lockdown Transitive Dependencies

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>commons-lang</groupId>
      <artifactId>commons-lang</artifactId>
      <version>2.1</version>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Don’t Rely On Transitive Depends.

- Declare all the dependencies you use directly to compile your code
  - Especially if it is available transitively from external dependency
- Don’t believe your eyes ☺ when you look on your local repository
  - Not all artifacts you see will be in your classpath
  - Some artifacts comes because of maven plugins and have no connection to your project

1. Direct dependency version
2. Dependency Management version
3. Transitive dependency version

Verify Dependencies Tree

- Always check your dependencies tree after adding or upgrading a dependency
  - Dependency tree with log4j-1.2.14:

```
[dependency:tree]
com.mycompany:maven:jar:1.0
\- log4j:log4j:jar:1.2.14:compile
```

Verify Dependencies Tree

- Dependency tree with log4j-1.2.15:

```
[dependency:tree]
com.mycompany:maven:jar:1.0
\- log4j:log4j:jar:1.2.15:compile
   +- javax.mail:mail:jar:1.4:compile
   |  \- javax.activation:activation:jar:1.1:compile
   +- javax.jms:jms:jar:1.1:compile
   +- com.sun.jdmk:jmxtools:jar:1.2.1:compile
   \- com.sun.jmx:jmxri:jar:1.2.1:compile
```

  - Adds
  - JMX

Lab 3: Dependency Management

fix the previous lab without declaring direct dependencies or using exclusions

Building Multi-module Dependencies

- Modules allow Maven to resolve dependencies not yet installed into the local repository

```
<modules>
  <module>core</module>
  <module>webapp</module>
  <module>standalone</module>
</modules>
```

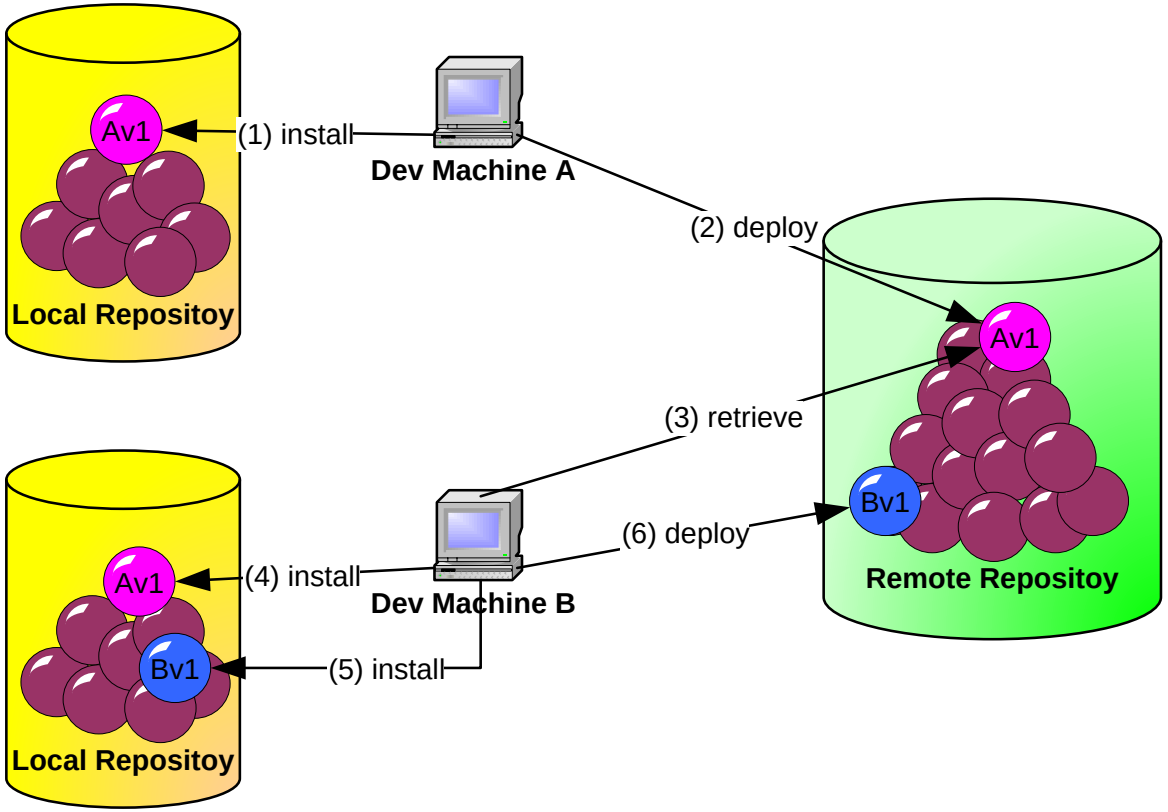
- Inter-dependent projects are built in one go and governed by a single "reactor" that is aware of all artifacts built so far
- Reactor executes build in the same order modules are specified

Lab 4: Multi-module project

Sharing Dependencies – Repositories

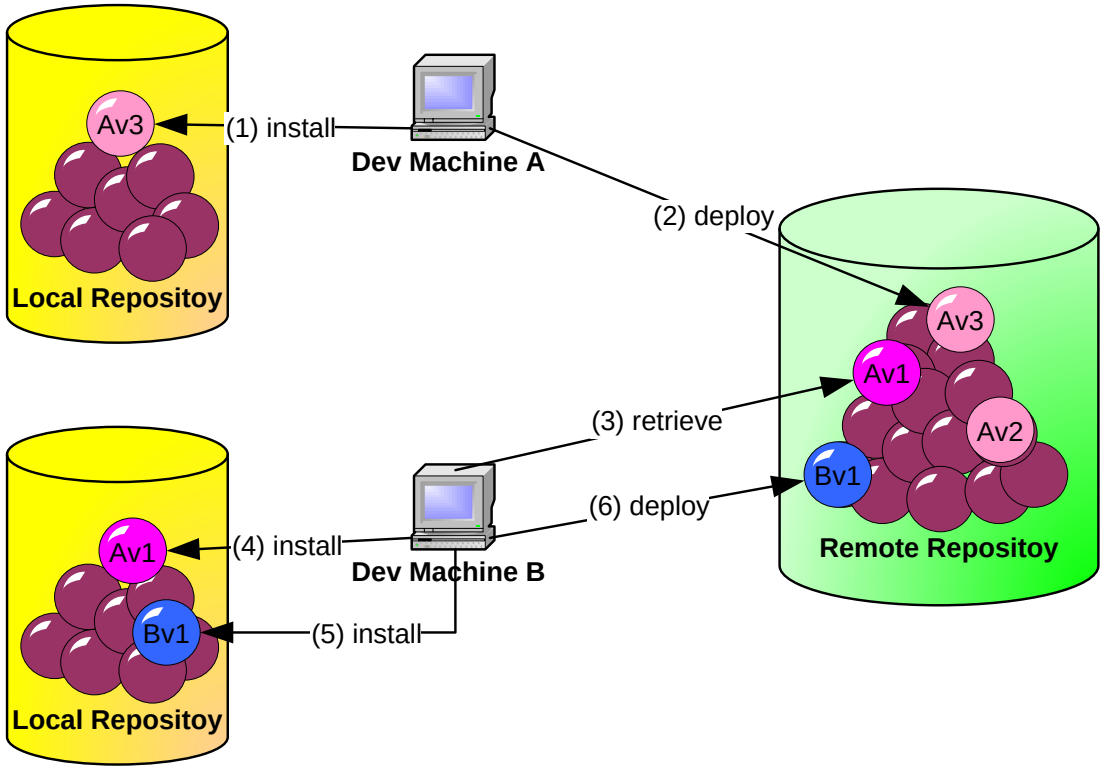
Sharing Dependencies

- Sharing done through remote repositories
- Developer B needs A-v1 built by developer A in order to build module B -v1



Versioned Storage

- Always version (or version-range) based
  - Developer B still depends on v1 of A
  - Inter-module API is version-specific

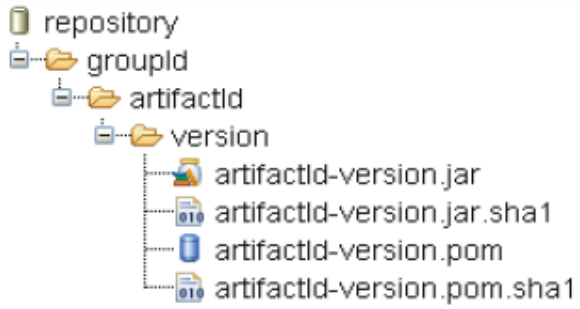


Local Repository

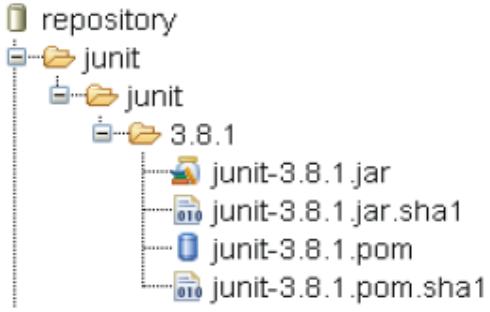
- A repository stored locally on the machine
- The default is `~/.m2/repository`
  - Can be overridden in Maven's `settings.xml` or by specifying `-Dmaven.repo.local=/path/to/m2/repository` to Maven
- Consulted first in dependency resolution
- Updated according to various update policies

Local Repository Layout

- The repository assumes a standard layout



To verify completeness artifacts include SHA1 checksum files

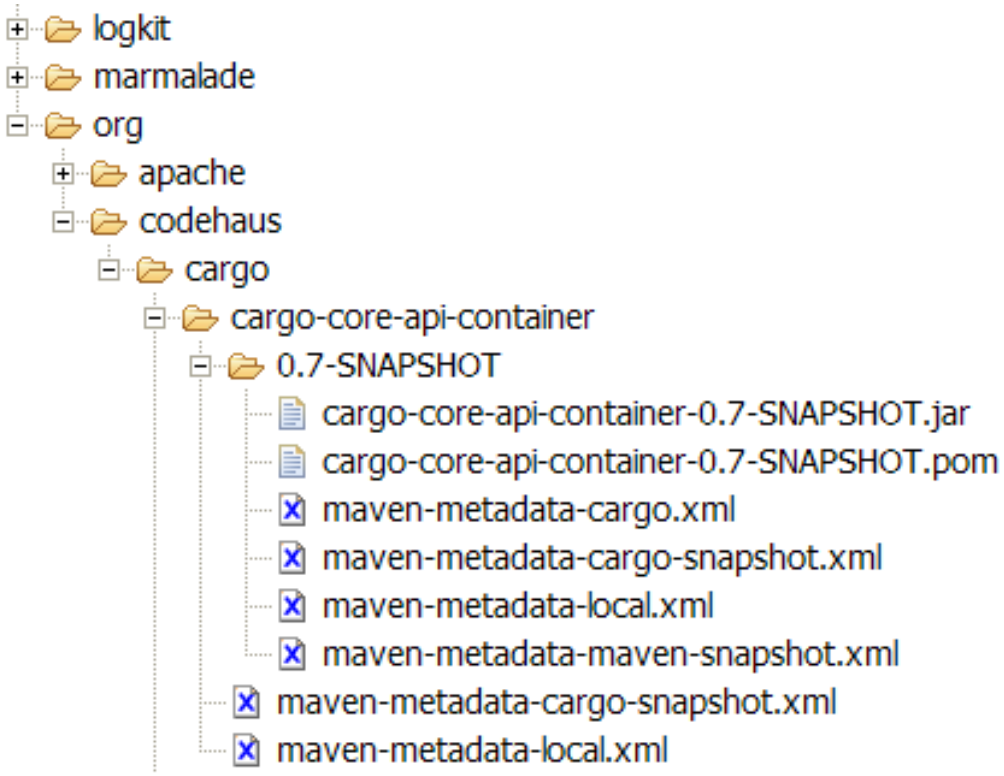


Remote Repositories

- Can be accessed by various protocols (http/s, ftp, ssh, filesystem, etc.)
- Maven provides an open transport layer of protocol providers - **wagon**
- Artifacts are deployed using wagon transports as well
- If communication is broken, the whole repository will be blacklisted for the current build session

Remote Repositories Layout

- Contain metadata about releases, latest deployed artifact, etc.



Defining Remote Repositories

- Repositories can be defined in:
  - POMs
  - Settings.xml
- How remote repos can be defined
  - Profiles (external or in settings)
  - Mirrors (in settings)

Remote Repositories Properties

```
<repository>
    <id>jdk14</id>
    <name>Repository for JDK 1.4 builds</name>
    <url>http://www.myhost.com/maven/jdk14</url>
    <layout>default</layout>
    <snapshots>
        <enabled>true</enabled>
    </snapshots>
</repository>
```

- Do not use the same id for 2 or more repositories.
  - Although Maven will not complain your build will become non-deterministic

Plugin vs. Regular Repositories

- Maven makes a difference between 2 kinds of remote repositories:
  - Regular repositories, and
  - Plugin repositories

Regular Repositories

- Contain artifacts needed for compiling and/or running the target software
  - 3rd party libs
  - Other modules
- Normally packaged with the resulting build artifact
- Licensing may need special attention since may be included in shipped software

Plugin Repositories

- Contain plugin artifacts required for running maven goals
  - Compile plugin, JavaDoc, code-gen etc.
- Never packaged with the resulting build artifact
- Has its own definition:

```
<pluginRepository>
            <id>Maven Snapshots</id>
            <url>http://repo.mergere.com/maven2/</url>
</pluginRepository>
```

Remote Repositories Update Policy

- Remote artifacts are checked for updates according to the remote/plugin repository updates policy:
  - Always
  - Daily (default)
  - interval:XXX (in minutes), or
  - Never (only if does not exist locally)

```
<repository>
            ...
            <updatePolicy>always</updatePolicy>
</repository>
```

Remote Repositories Update Policy

- Only snapshots are checked for updates according to the update policy
- Force snapshots update

```
mvn -U ...
```

Built-in Remote Repositories

- By default Maven has 2 built-in repositories identified by the ids:
  - “central” and “snapshots”
- These repositories are used by default in every build and are **heavily loaded**
- suffer from:
  - Downtimes, network timeouts
  - Broken or incomplete artifacts
    - Invalid POM XML, bad checksums
  - No authenticity of artifact source

Remote Repository Mirrors

- Both built-in and user-defined repositories are used during build
- To overcome repositories downtime/timeouts you can define mirrors inside `settings.xml`
- Mirroring is done by id:

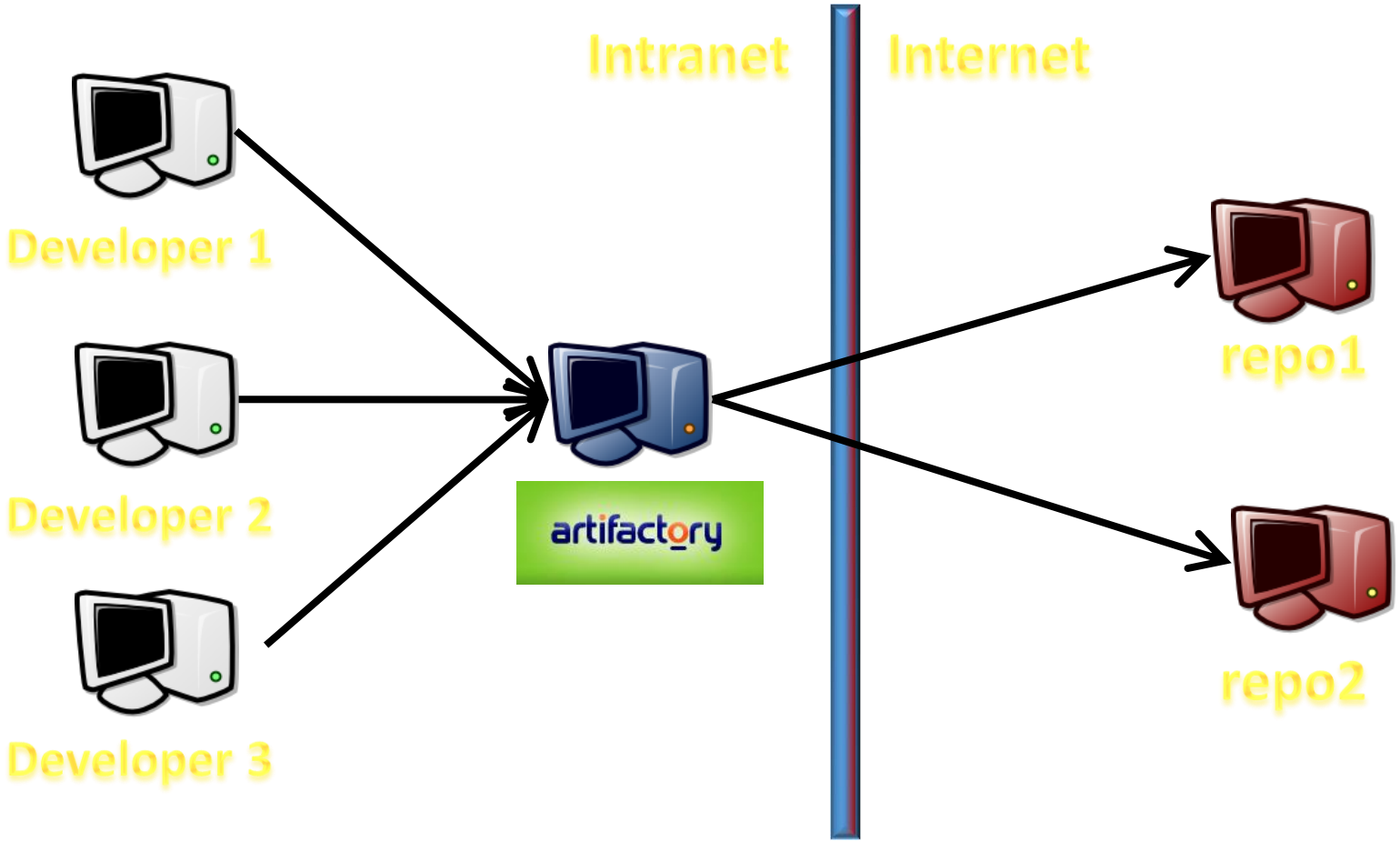
```
<mirrors>
   <mirror>
   <id>dotsrc.org</id>
   <url>http://mirrors.dotsrc.org/maven2</url>
   <mirrorOf>central</mirrorOf>
</mirrors>
```

Remote Repository Mirrors

- Both built-in and user-defined repositories are used during build
- To overcome repositories downtime/timeouts you can define mirrors inside `settings.xml`
- Mirroring is done by id:

```
<mirrors>
   <mirror>
   <id>dotsrc.org</id>
   <url>http://mirrors.dotsrc.org/maven2</url>
   <mirrorOf>*</mirrorOf>
</mirrors>
```

Internal Repository



Internal Repository (Artifactory)

- Controlled by you
- A must for reproducible build
- Caches remote repositories (faster)
- More secure
- Makes binaries sharing between teams possible

Internal Repository

- Faster builds
- Can fix broken poms
- Filter artifacts
- Arrange artifacts in different repositories

Lab 5: Repositories

POM Inheritance

POM Inheritance

- One of the main strengths of Maven is the ability to create inheritance between POMs

```
<parent>
   <artifactId>module-parent</artifactId>
   <groupId>org.snowflakes</groupId>
   <version>1.1-SNAPSHOT</version>
</parent>
```

POM Inheritance (Contd.)

- POM inheritance creates a unified object model, made of the current running POM and all of its ancestors
- To dump and examine the actual runtime-constructed POM, you can use:

```
mvn help:effective-pom
```

POM Inheritance (Contd.)

- POM inheritance is a useful technique to achieve reuse between modules:
  - High-level POMs define project-wide or module-wide POM configuration:
    - Plugins configuration, source control, common dependencies etc.
  - Module-specific POMs inherit and customize

Parent POM Resolution

- You cannot use dynamic properties as part of parent POM declaration:

```
<parent>
   <artifactId>${my.parent}</artifactId>
   <groupId>org.snowflakes</groupId>
   <version>1.1-SNAPSHOT</version>
</parent>
```

- All parents need to be resolved statically (like most OO-langs)

Parent POM – Using RelativePath

- A parent POM optional relative path is a development-supporting info
- Helps retrieve the parent POM if it is not installed (in the local repository)
- Default value is `../pom.xml`

```
<parent>
   <relativePath>../parent/pom.xml</relativePath>
   <artifactId>${my.parent}</artifactId>
   <groupId>org.snowflakes</groupId>
   <version>1.1-SNAPSHOT</version>
</parent>
```

Parent POM Resolution Order

The order of parent POM resolution is:

1. The reactor of currently building projects
2. The relativePath location on the file system
   - Maven will not complain if GroupId, artifactId or version do not match the POM data in the location given
   - Will simply try the next resolution option
3. Local repository
4. Remote repositories

Dependency & Plugin Management

- We can define rules for dependency and plugin versions using the `dependencyManagement` and `pluginManagement` elements

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.5</version>
    </dependency>
    ...
  <dependencies>
</dependencyManagement>
```

Dependency & Plugin Management

- Usually done in a parent POM
- Child POMs can repeat the dependency without specifying any version

```
<dependencies>
   <dependency>
     <groupId>javax.servlet</groupId>
     <artifactId>servlet-api</artifactId>
   </dependency>
   ...
<dependencies>
```

Lab 6: POM Inheritance

Maven’s Cross-project Configuration

Settings.xml

- Maven stores global, non-POM, configuration in `settings.xml` files
- What can be configured:
  - Local repository location
  - Remote repositories
  - Network proxies
  - Permissions and authentication details of servers
  - Repository mirrors
  - Profiles

Settings.xml

- There are 2 `settings.xml` files:
  - Global: `$M2_HOME/conf/settings.xml`
  - User: `~/.m2/settings.xml`
- The 2 files are merged in runtime
- User settings take precedence over global settings

Settings.xml (Contd.)

- To view the runtime settings being used:

```
mvn help:effective-settings
```

- Dynamic property values cannot be used in settings (Maven limitation)

Profiles

Profiles

- Profiles provide a convenient way to change the build behavior dynamically
- Profiles can be included in:
  - Settings.xml (user & global)
  - POMs
  - External profiles descriptor
    - `profiles.xml` under the project basedir
- Can be activated manually or automatically

Manual Profile Activation

- Using -P on the Maven CLI

```
mvn -P profileX,profileY ...
```

- Defining active profiles in settings.xml

```
<settings>
  ...
  <activeProfiles>
    <activeProfile>profileX</activeProfile>
  </activeProfiles>
  ...
</settings>
```

Automatic Profile Activation

- Using special activation directive
  - By JDK version (prefix)

```
<profile>
    <activation>
      <jdk>1.4</jdk>
    </activation>
    ...
</profile>
```

```
<profile>
<activation>
 <jdk>[1.3,1.6)</jdk>
 </activation>
 ...
</profile>
```

  - By system property with/without a value

```
<profile>
    <activation>
      <property>
        <name>location</name>
        <value>tel-aviv</value>
      </property>
    </activation>
    ...
</profile>
```

What Can Be Defined inside Profiles?

- In `settings.xml` or `profiles.xml`
  - Repositories and pluginRepositories
  - Properties
- In addition, `pom.xml`profiles can define
  - dependencies
  - plugins
  - modules
  - reporting
  - dependencyManagement
  - distributionManagement
  - build subset:
    - defaultGoal
    - resources & testResources
    - finalName

What can be activation trigger

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

What Profiles are Active?

- Using the help plugin

```
mvn ... help:active-profiles ...
```

1. In your dependency management declare Junit version 3.8.1
2. In your profile declare dependency of Junit version 4.11
3. Profile must be activated if environment variable BLA = xxx

- Hint: env.BLA

Lab 07: Profiles

Installation & Deployment

Maven Install

- Installing is the process of storing artifacts into the local repository
- Installation can be manual or occur as a result of dependency resolution
- Manual install of a built artifact
  - Runs all previous phases (package, compile…)
  - Artifacts are usually produced into `${basedir}/target`

```
mvn clean install ...
```

Maven Install (Contd.)

- Manual install of a 3rd party artifact file
  - Using the install-file mojo of the install plugin

```
mvn install:install-file -DgroupId=group-id \
                         -DartifactId=artifact-id \
                         -Dversion=version \
                         -Dpackaging=packaging \
                         -Dfile=fileToInstall
```

- The installation process also creates:
  - Checksum files
  - Maven meta files with version info, etc.

Maven Deploy

- Deploying is the process of storing artifacts into a remote repository
- Deplyment is done according to a distribution management spec:

```
<distributionManagement>
...
  <repository>
    <id>unique-repository-id</id>
    <name>human-readable-name</name>
    <url>wagon-url</url>
  </repository>
...
</distributionManagement>
```

Maven Deploy (Contd.)

- Security credentials may be specified in `settings.xml`
- Deploy
  - Previous phases will automatically run
  - Usually combined with profiles to determine deployment destination

```
mvn clean deploy ...
```

Manual File Deployment

- Use the deploy plugin
  - Similar to manual artifact install:

```
mvn deploy:deploy-file -Durl=file://C:\m2-repo \
                       -DrepositoryId=some.id \
                       -Dfile=your-artifact-1.0.jar \
                       [-DpomFile=your-pom.xml] \
                       [-DgroupId=org.some.group] \
                       [-DartifactId=your-artifact] \
                       [-Dversion=1.0] \
                       [-Dpackaging=jar] \
                       [-Dclassifier=test] \
                       [-DgeneratePom=true] \
                       [-DgeneratePom.description="Desc..."] \
                       [-DrepositoryLayout=legacy] \
                       [-DuniqueVersion=false]
```

POM Stored Inside A Repository

- When installing or deploying a POM, the repository stores: The “effective POM” after all properties expansion, but without parent POM inclusion
  - Still uses `parent` reference declaration
- The installed POM must be usable by people who don’t have the original execution environment (settings, profile files, system properties, etc.)

1. Add server tag in your settings.xml (id, username & password of your artifatory)
2. Add distributionManagement tag in your pom
   - Copy paste it from specifiv artifactory repository
3. Run mvn deploy and check that artifact deployed to artifactory

Lab 08: Deployment

Using Plugins

What is a Plugin?

- A Plugin is a jar that contains MOJOs
  - Maven Old Java Objects
- Each MOJO is equivalent to a goal and is identified by a unique goal name

xfire-maven-plugin

Goal	Description
xfire:wsdlgen	WsdlGen mojo. Implemented as a wrapper around the XFire WsdlGen Ant task.
xfire:wsgen	WsGen mojo. Implemented as a wrapper around the XFire WsGen Ant task.

- MOJOs are bound to build lifecycle phases
  - perform the actual build work

Calling Plugins Directly

- The Plugin is identified as any other Maven artifact
  - artifactId, groupId, version
- To call a plugin we use the plugin id and the goal id:
  - Run the JavaDoc plugin's `javadoc` MOJO

```
mvn groupId:artifactId:[version:]goal
```

```
mvn org.apache.maven.plugins:maven-javadoc-plugin:2.1:javadoc
```

Calling Plugins – Plugin Prefixes

- Maven assumes plugin prefix is used to save some typing
- For example, when typing –

```
mvn clean:clean
```

Maven will assume the plugin artifactId is `maven-${prefix}-plugin`, or what is specified in the plugin's pom `goalPrefix` element, i.e.:

```
mvn org.apache.maven.plugins:maven-clean-plugin:2.0:clean
```

Calling Plugins – Plugin Groups

- For groupId, Maven tries to locate the plugin in the repositories by iterating over `pluginGroups` in `settings.xml`

```
<settings>
  ...
  <pluginGroups>
    <pluginGroup>myGroupId</pluginGroup>
  </pluginGroups>
  ...
<settings>
```

- The internal plugin groups `org.apache.maven.plugins` and `org.codehaus.mojo` are always checked

Controlling Plugin Execution

- We can manually bind MOJOs to new build lifecycle phases
  - Always *in addition* to any MOJO built-in phase bindings

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-javadoc-plugin</artifactId>
  <executions>
    <execution>
      <phase>process-classes</phase>
      <goals>
        <goal>javadoc</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Plugin Configuration

- Plugin can be configured either globally or per execution:

```
<build>
    <plugins>
      <plugin>
        <groupId>groupId</groupId>
        <artifactId>artifactId</artifactId>
        <configuration/>
        <executions>
          <execution>
            <configuration/>
            ...
          </execution>
        </executions>
      </plugin>
      ...
    </plugins>
  </build>
```

Plugin Configuration

- Configuration is an open element and is plugin specific
- For example, the compiler plugin can take:

```
<build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.6</source>
          <target>1.5</target>
        </configuration>
      </plugin>
      ...
    </plugins>
</build>
```

Plugin Versioning

- Don’t have to state explicit version
  - Because all poms explicitly extends maven parent pom which knows about every known plugin version
- Will use default version or latest version
  - See org.apache.maven:maven-parent for default versions
- Implicit updates
  - Less reproductive build

Lockdown Plugin Versions

```
<pluginsManagement>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.0.2</version>
    </plugin>
  </plugins>
</pluginsManagement>
```

- Even if you use Maven 2.0.9 or later

Lab 09: Plugins Usage

Important Plugins

Help Plugin

- Already shown:
  - effective-settings
  - effective-pom
  - active-profiles

Dependency Plugin

Discover your project dependencies

```
mvn dependency:list
```

Lists all the dependencies of the project and their scope

```
The following files have been resolved:
  antlr:antlr:jar:2.7.6:runtime
  cglib:cglib:jar:2.1_3:runtime
  cglib:cglib-nodep:jar:2.1_3:compile
  com.google.code.guice:guice:jar:1.0:compile
  com.intellij:annotations:jar:6.0.5:provided
  commons-cli:commons-cli:jar:1.0:compile
  commons-lang:commons-lang:jar:2.2:compile
```

Dependency Plugin

Or even better - the dependency tree

```
mvn dependency:tree
```

- Source of each dependency

```
+- org.hibernate:hibernate:jar:3.2.6.ga:runtime
|  +- cglib:cglib:jar:2.1_3:runtime
|  +- dom4j:dom4j:jar:1.6.1:runtime
|  |  +- xpp3:xpp3:jar:1.1.3.3:runtime
|  |  \- stax:stax-api:jar:1.0:runtime
+- org.easymock:easymock:jar:2.3:test
|  \- cglib:cglib-nodep:jar:2.1_3:test
```

Dependency Plugin

There’s a potential problem here

```
+- org.hibernate:hibernate:jar:3.2.6.ga:runtime
|  +- cglib:cglib:jar:2.1_3:runtime
|  +- dom4j:dom4j:jar:1.6.1:runtime
|  |  +- xpp3:xpp3:jar:1.1.3.3:runtime
|  |  \- stax:stax-api:jar:1.0:runtime
+- org.easymock:easymock:jar:2.3:test
|  \- cglib:cglib-nodep:jar:2.1_3:test
```

Dependency Plugin

One way to solve this:

```
<dependency>
    <groupId>org.easymock</groupId>
    <artifactId>easymockclassextension</artifactId>
    <version>2.3</version>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>cglib</groupId>
            <artifactId>cglib-nodep</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Dependency Plugin

- Also displays the reason for modified versions and scopes

```
+ - jackrabbit:jackrabbit-core:jar:1.4.5:compile
| +- concurrent:concurrent:jar:1.3.4:runtime
            (scope managed from compile)
| +- collections:collections:jar:3.2.1:compile
            (version managed from 3.1)
```

Dependencies Cleanup

```
mvn dependency:analyze -DignoreNonCompile
```

- Used but undeclared dependencies
- Unused declared dependencies
- Also displays dependency management mismatches

Dependencies Cleanup

Dependency analyzer output

```
Used undeclared dependencies found:
   commons-io:commons-io:jar:1.4:compile
   org.springframework:spring-core:jar:2.5.5:compile
   com.thoughtworks.xstream:xstream:jar:1.3:compile
   org.easymock:easymock:jar:2.3:test

Unused declared dependencies found:
   org.springframework:spring-jdbc:jar:2.5.5:compile
   org.springframework:spring-aop:jar:2.5.5:compile
```

Dependency Plugin

Detect mismatches between dependency management and resolved dependencies

```
mvn dependency:analyze-dep-mgt
```

- Very useful for pre 2.0.6 releases
- Helps to detect dependencies that were excluded but still exists in the dependency tree

Enforcer Plugin

```
mvn enforcer:enforce
```

- Ban dependencies
- Require system properties
- Require Java/Maven version
- File exists / not exists
- Beanshell scripts
- Custom rules

Enforce Plugin Configuration

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-enforcer-plugin</artifactId>
  <version>1.0-mycompany-4</version>
  <executions>
    <execution>
      <goals>
        <goal>enforce</goal>
      </goals>
      <configuration>
        <rules>
          ........
        </rules>
      </configuration>
        </execution>
    </executions>
</plugin>
```

Enforce Banned Dependencies

```
<configuration>
  <rules>
    <bannedDependencies>
      <excludes>
        <!-- we use cglib:cglib -->
        <exclude>cglib:cglib-nodep</exclude>
        <!-- avoid the full spring jar -->
        <exclude>org.springframework:spring</exclude>
        <!-- old artifact id -->
        <exclude>xerces:xerces</exclude>
      </excludes>
    </bannedDependencies>
  </rules>
</configuration>
```

Enforce Plugin Versions

```
<configuration>
  <rules>
    <requirePluginVersions>
      <banLatest>true</banLatest>
      <banRelease>true</banRelease>
      <banSnapshots>true</banSnapshots>
       <additionalPlugins>
        <additionalPlugin>
          groupId:artifactId
        </additionalPlugin>
      </additionalPlugins>
    </requirePluginVersions>
  </rules>
</configuration>
```

Enforcer - Sample Standard Rules

```
<configuration>
  <rules>
    <requireJavaVersion>
      <version>[1.5.0, 1.6.0),[1.6.0-4,)</version>
    </requireJavaVersion>
    <requireMavenVersion>
      <version>3.0.2</version>
    </requireMavenVersion>
    <requireProperty>
      <property>driver.class</property>
      <message>Property driver.class not set</message>
    </requireProperty>
  </rules>
</configuration>
```

Lab

- Use enforcer plugin – build must failed if jdk version is not 8
- Build must fail if environment variable BLA is not exist or exists, but contains no xxx value.
- Each failure must have it’s own message
- Hint: http://maven.apache.org/enforcer/enforcer-rules/requireProperty.html

Lifecycles & Packaging

Built-in Lifecycles

- Maven has 3 built-in lifecycles:
  - Default (phases: deploy, install, package…)
  - clean
  - site
- Each lifecycle defines a collection of phases that execute all their preceding phases
- The default lifecycles are defined inside `org.apache.maven:maven-core` under **`META-INF/plexus/components.xml`**

The Default Lifecycle

- The default lifecycle is the most commonly used in Java projects
- It has the following phases in order:

1. validate
2. generate-sources
3. process-sources
4. generate-resources
5. process-resources
6. compile
7. process-classes
8. generate-test-sources
9. process-test-sources
10. generate-test-resources
11. process-test-resources
12. test-compile
13. test
14. package
15. pre-integration-test
16. integration-test
17. post-integration-test
18. verify
19. install
20. deploy

The Clean & Site Lifecycles

- The clean lifecycle has the following phases:
  1. pre-clean
  2. clean
  3. post-clean
- The site lifecycle has the following phases:
  1. pre-site
  2. site
  3. post-site
  4. site-deploy

Packaging

- Lifecycles are merely static definitions
- To make use for them in runtime we need to bind plugin goals to the defined phases
- This is done using the packaging concept
- Packaging is defined in the POM:

```
<project>
    <groupId>foo</groupId>
    <artifactId>bar</artifactId>
    <version>1.1</version>
    <packaging>war</packaging>
    ...
```

The Jar Packaging

- Jar is the default packaging type (if unspecified)
- It defined the following goals-to-phase bindings:

phase	plugin-prefix: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

Custom Packaging

- Custom packaging can be defined in a **`META-INF/plexus/components.xml`** file and packaged into a plugin
- For example:

```
<component-set>
  <components>
    <component>
      <role>org.apache.maven.lifecycle.mapping.LifecycleMapping</role>
      <role-hint>multijar</role-hint>
      ...
      <configuration>
        <phases>
          ...
          <compile>org.apache.maven.plugins:maven-compiler-plugin:compile</compile>
          ...
          <package>com.mycompany.maven.plugins:jade-multijar-plugin:jar</package>
          ...
        </phases>
      </configuration>
  </component>
  ...
```

Original default lifecycle

```
<component>
    <role>org.apache.maven.lifecycle.mapping.LifecycleMapping</role>
    <role-hint>jar</role-hint>
    <implementation>
      org.apache.maven.lifecycle.mapping.DefaultLifecycleMapping
    </implementation>
    <configuration>
      <lifecycles>
        <lifecycle>
          <id>default</id>
          <phases>
            <process-resources>
              org.apache.maven.plugins:maven-resources-plugin:resources
            </process-resources>
            <compile>
              org.apache.maven.plugins:maven-compiler-plugin:compile
            </compile>
            <process-test-resources>
              org.apache.maven.plugins:maven-resources-plugin:testResources
            </process-test-resources>
            <test-compile>
              org.apache.maven.plugins:maven-compiler-plugin:testCompile
            </test-compile>
            <test>
              org.apache.maven.plugins:maven-surefire-plugin:test
            </test>
            <package>
              org.apache.maven.plugins:maven-jar-plugin:jar
```

Custom Packaging (Contd.)

- When packaged into a plugin, the plugin must tell Maven it provides an extension (to the built-in packaging)
- This is done in the POM by:

```
<packaging>multijar</packaging>
 ...
<plugin>
   <groupId>bar</groupId>
   <artifactId>maven-multijar-plugin</artifactId>
   <extensions>true</extensions>
</plugin>
...
```

Plugin Execution

- Another way to bind plugin goals to lifecycle phases is with the `executions` element of the plugin:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-source-plugin</artifactId>
    <executions>
        <phase>package</phase>
        <execution>
            <id>attach-sources</id>
            <goals>
                <goal>jar</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Plugin Execution (Contd.)

- The phase can be omitted if the plugin developer already bound a Mojo to well-known phases in the plugin descriptor

```
/**
@phase package
*/
public class MyMojo {
    ...
}
```

Maven and Version Control

Maven SCM

- Maven's SCM - an API abstraction above Source Control Management Systems
  - Supports most common SCMs
    - CVS, SVN, Perforce, VSS, Clear Case, etc.
  - Supports most common SCM operations
    - checkout, tag, check in, remove, update, merge, etc.
- SCM operations can be controlled by running the `maven-scm-plugin`

Maven SCM Declaration

- For examp

```
<scm>
  <connection>read-only (no-commit) url</connection>
  <developerConnection>committer</developerConnection>
  <url>Browsable repo URL</url>
</scm>
```

- For more documentation, see:

http://docs.codehaus.org/display/SCM/SCM+Matrix

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

The Release Plugin

- Leverages Maven's SCM transparency to control the process of releasing a new artifact version
  - Check for no outstanding changes
  - Change POM
  - Tag version control
  - Extend version history file
  - Deploy artifact
  - Deploy site
  - Send mails to announce new release
  - Change POM for next development (next version-SNAPSHOT)

The Release Plugin (Contd.)

- For more documentation, see:

  `http://maven.apache.org/plugins/maven-release-plugin/`

Using Archetypes

Maven Archetypes

- A bootstrapping plugin that enables quick creation of project skeletons
- Very useful for quickly creating new modules
- Archetype = project template
  - An archetype is a normal artifact
  - Stored and retrieved from a repository

Available Archetypes

- Easy to package new archetype types
  - Based o

Index of /pub/mirrors/maven2/org/apache/maven/archetypes

Name	Last modified	Size	Description
Parent Directory		-	
maven-archetype-archetype/	02-Dec-2006 23:41	-	
maven-archetype-bundles/	02-Dec-2006 23:24	-	
maven-archetype-j2ee-simple/	02-Dec-2006 23:41	-	
maven-archetype-marmalade-mojo/	07-May-2006 19:40	-	
maven-archetype-mojo/	02-Dec-2006 23:39	-	
maven-archetype-plugin-site/	02-Dec-2006 23:42	-	
maven-archetype-plugin/	02-Dec-2006 23:42	-	
maven-archetype-portlet/	02-Dec-2006 23:42	-	
maven-archetype-profiles/	07-May-2006 19:40	-	
maven-archetype-quickstart/	02-Dec-2006 23:40	-	
maven-archetype-simple/	02-Dec-2006 23:40	-	
maven-archetype-site-simple/	02-Dec-2006 23:40	-	
maven-archetype-site/	02-Dec-2006 23:40	-	
maven-archetype-webapp/	02-Dec-2006 23:41	-	
maven-archetypes/	07-May-2006 19:40	-	

Creating an Archetype

- Run the archetype plugin

```
mvn archetype:create -DarchetypeGroupId=archetypeGroupId \
                     -DarchetypeArtifactId=archetypeArtifactId \
                     -DarchetypeVersion=archetypeVersion \
                     -DgroupId=newProjectGroupId \
                     -DartifactId=newProjectArtifactId
```

- For example

```
mvn archetype:create -DarchetypeGroupId=org.apache.myfaces.maven \
                     -DarchetypeArtifactId=maven-archetype-myfaces \
                     -DarchetypeVersion=1.0-SNAPSHOT \
                     -DgroupId=myAppId \
                     -DartifactId=testApp
```

Lab 10: Archetypes

Site & Project Reports

Site Reports

built by: maven

- Meta-information for users and developers
  - name, version, date
  - Dependencies
  - Mailing Lists & newsgroups
  - Continous Integration
  - Source Repository
  - Issue Tracking
  - Project Team
  - License
- User and developer documentation
  - changelog, version history,
  - release notes
  - documentation
  - JavaDoc
- Project health and quality reports
  - test reports
  - style reports
  - metrics

`project-info-reports`

Site Reports Creation

- Plugins that generate reports need to be added inside a special `reporting` section
- For example, to generate POM-derived reports:

```
...
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-project-info-reports-plugin</artifactId>
    </plugin>
  </plugins>
</reporting>
...
```

Site Content Creation

- Edit site content in
  - **apt** - “almost plain text”, WIKI style editing
  - **fml** – FAQ generation format

```
+- src/
   +- site/
      +- apt/
      |  +- index.apt
      |
      +- xdoc/
      |  +- other.xml
      |
      +- fml/
      |  +- general.fml
      |  +- faq.fml
      |
      +- site.xml
```

Site Navigation Creation

- Create a site descriptor
  - `src/site/site.xml`
  - Controls site layout and navigation menus

```
...
<menu name="Maven 2.0">
  <item name="Introduction" href="index.html"/>
  <item name="Download" href="download.html"/>
  <item name="Release Notes" href="release-notes.html" />
  <item name="General Information" href="about.html"/>
  <item name="Road Map" href="roadmap.html" />
</menu>

${reports}
...
```

Generation & Deployment

- To generat

```
mvn site
```

- To deploy a site
  - Add site

```
<distributionManagement>
  <site>
    <id>website</id>
    <url>scp://www.mycompany.com/www/docs/project/</url>
  </site>
</distributionManagement>
```

  - Then run the deploy goal

```
mvn site-deploy
```

Site Examples – Code Coverage

Coverage Report - org.apache.maven.plugin

Package	# Classes	Line Coverage	Branch Coverage	Complexity
org.apache.maven.plugin	4	84%	93%	3.5

Classes in this Package	Line Coverage	Branch Coverage	Complexity
AbstractCompilerMojo	82%	92%	6
CompilationFailureException	100%	100%	1.5
CompilerMojo	92%	100%	1.667
TestCompilerMojo	86%	100%	1.833

Report generated by Cobertura 1.7 on 11/8/06 6:34 AM.

Site Examples – Surefire Report

Summary

[Summary][Package List][Test Cases]

Tests	Errors	Failures	Success Rate	Time
14	0	0	100%	2.516

Note: failures are anticipated and checked for with assertions while errors are unanticipated.

Package List

[Summary][Package List][Test Cases]

Package	Tests	Errors	Failures	Success Rate	Time
org.apache.maven.plugin.install	14	0	0	100%	2.516

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

org.apache.maven.plugin.install

	Class	Tests	Errors	Failures	Success Rate	Time
	InstallFileMojoTest	7	0	0	100%	1.625
	InstallMojoTest	7	0	0	100%	0.891

Test Cases

[Summary][Package List][Test Cases]

InstallFileMojoTest

	testInstallFileTestEnvironment	0.578
	testBasicInstallFile	0.219
	testInstallFileWithClassifier	0.125
	testInstallFileWithGeneratePom	0.203
	testInstallFileWithPomFile	0.141

Site – Custom Look & Feel

WICKET

Last Published: 10/29/2006

Wicket
Home
Features
Buzz
Introduction

Getting Started
Examples
Download

Project News
News
Blogs

Generated Reports

This document provides an overview of the various reports that are automatically generated by Maven Each report is briefly described below.

Overview

Document	Description
Maven Surefire Report	Report on the test results of the project.
JavaDocs	JavaDoc API documentation.

Maven Site Plugin

- For more documentation, see:

  `http://maven.apache.org/plugins/maven-site-plugin/`

Lab 11: Site Generation

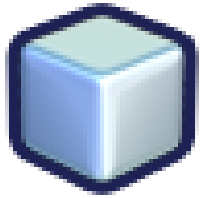
IDE Support

IDEs Integration

- Integration with the popular IDEs is bidirectional
- The preferred way is the IDE built in support (native or via plugin)

Maven Integration Plugins

mvn idea:idea



mvn netbeans-freeform:generate-netbeans-project

mvn eclipse:eclipse

IDE Plugins

IntelliJ IDEA – maven integration

Eclipse m2eclipse, q4e

NetBeans – maven plugin, maven dependency graph plugin

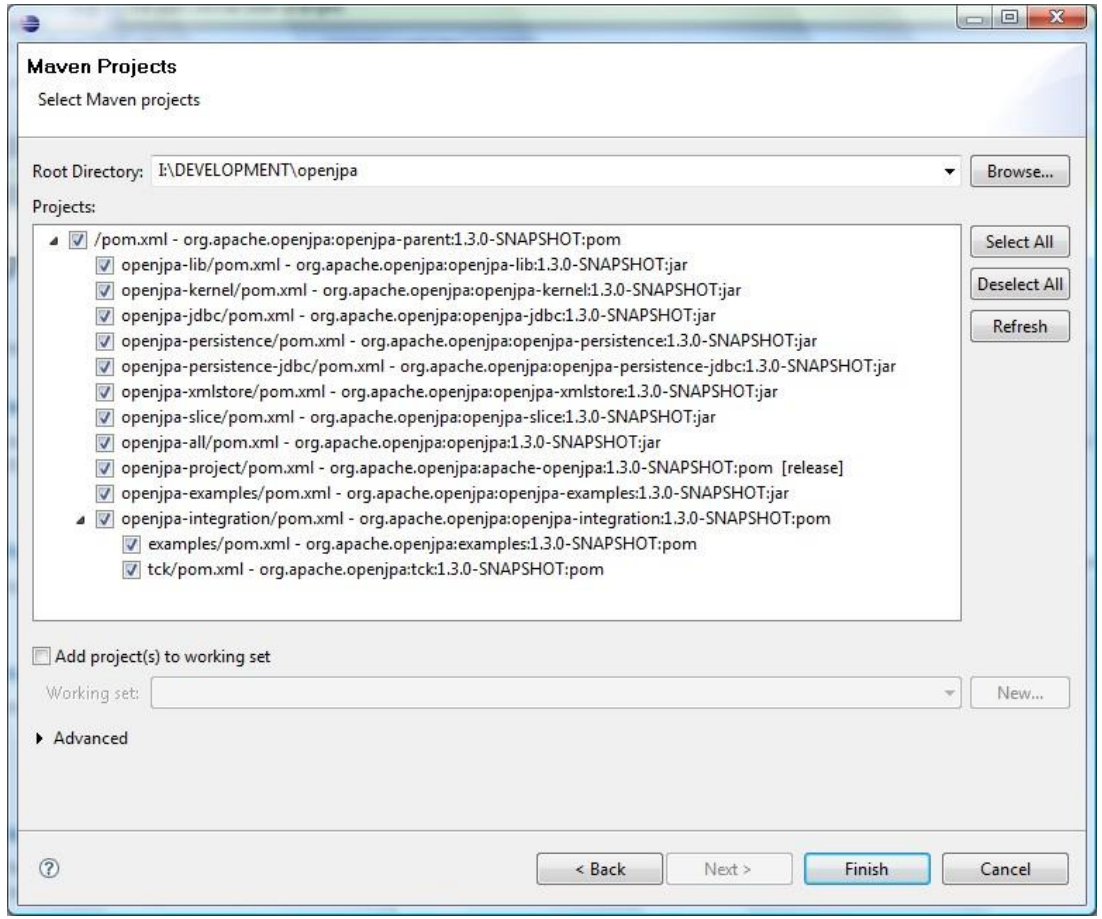
IDE Support

Open Project

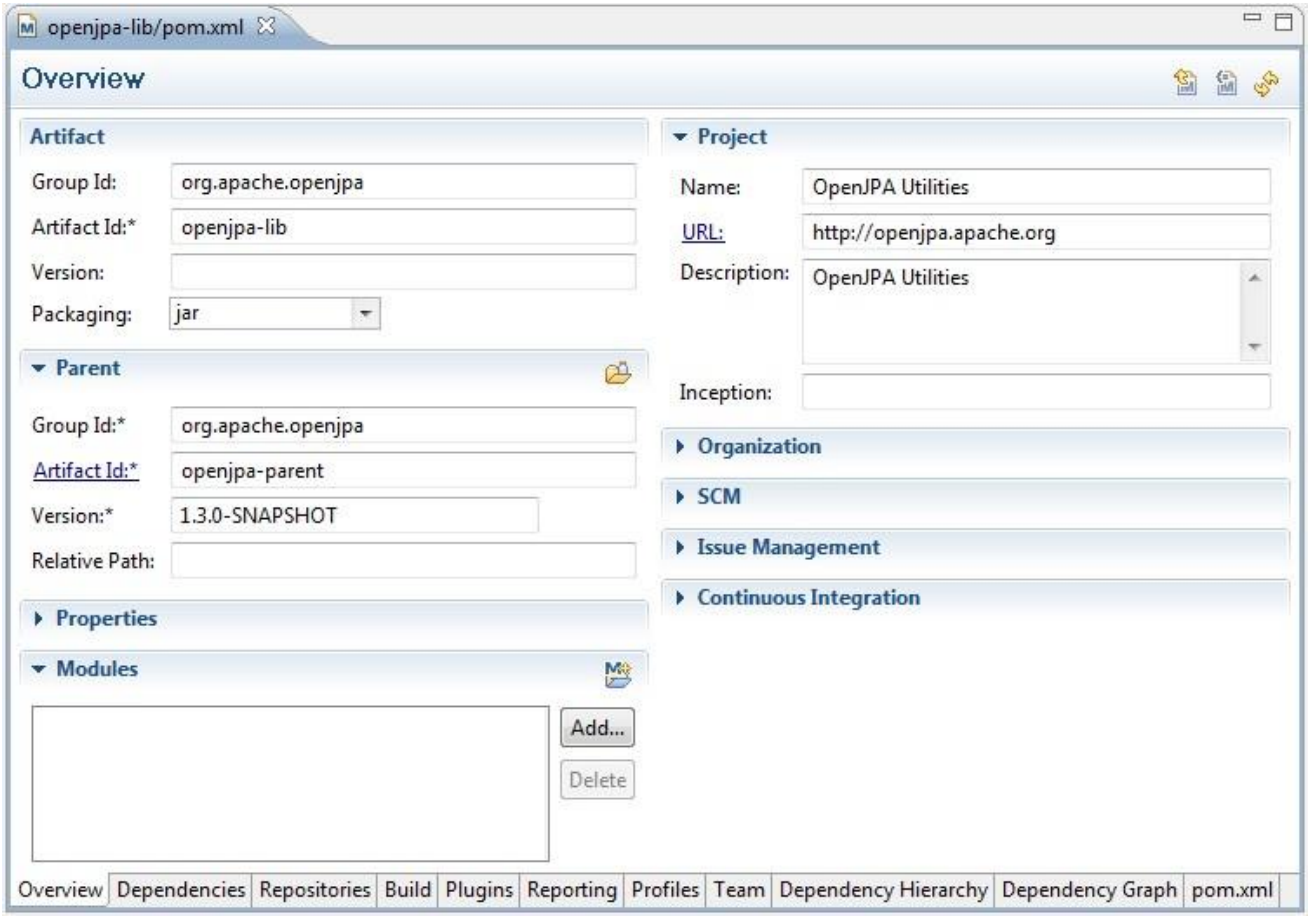
Open Project

Import Project

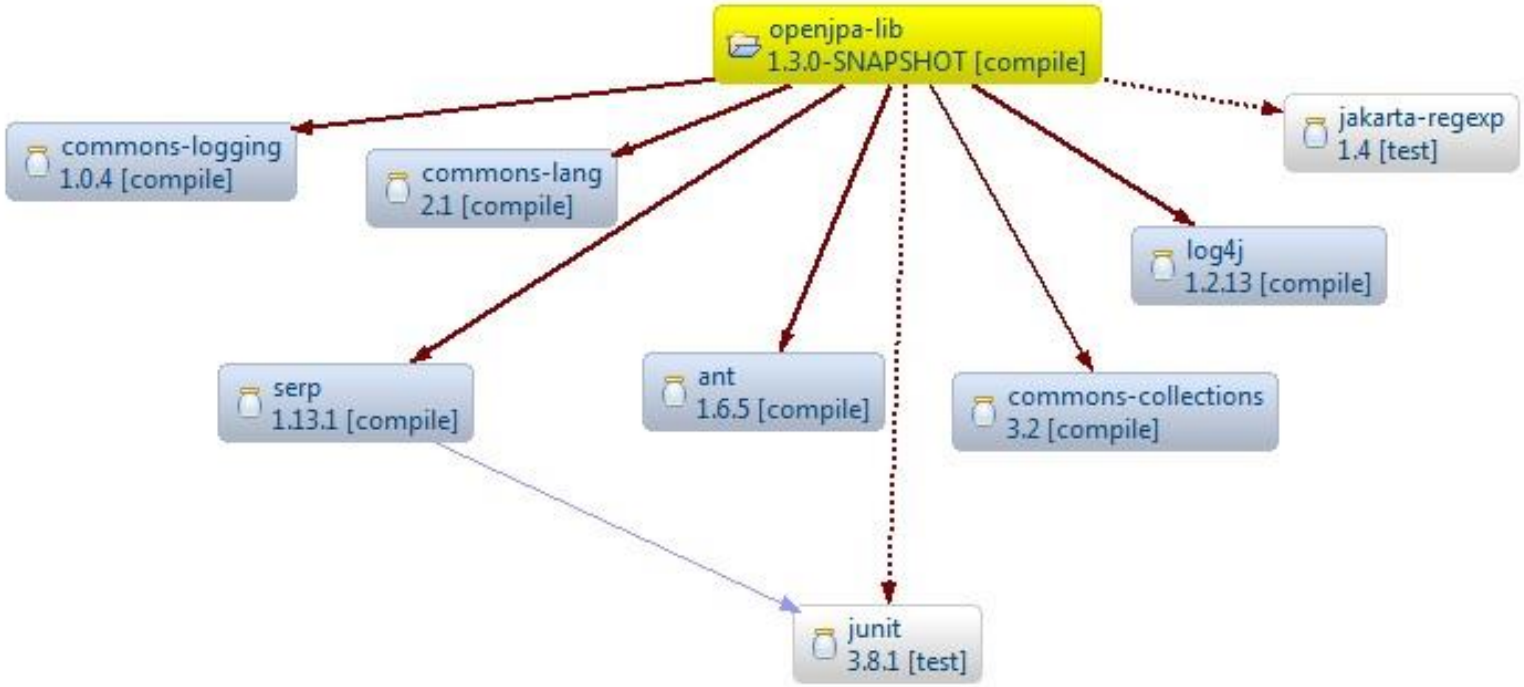
Import project from pom.xml



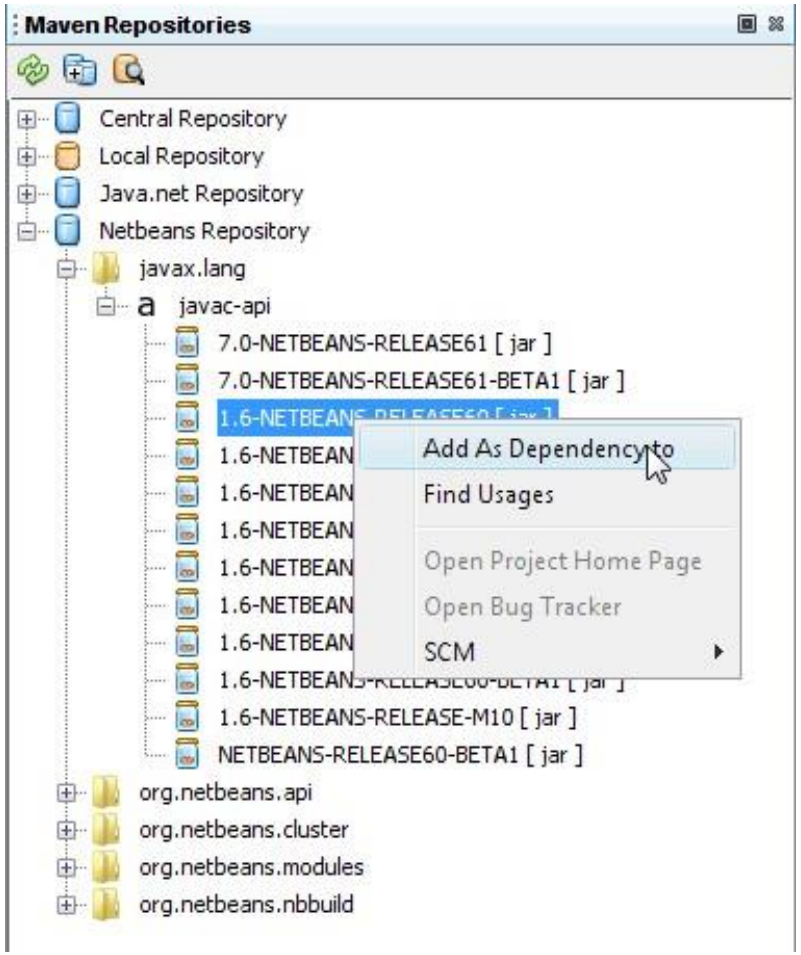
Eclipse Visual POM Editor



Dependency graph

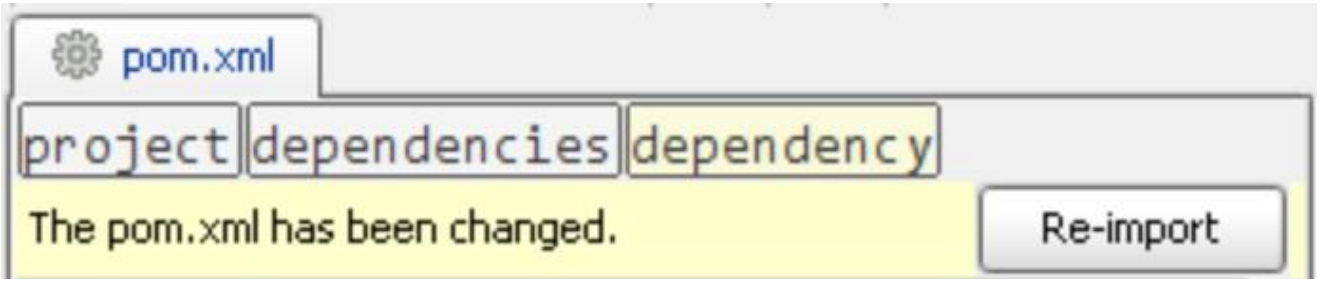


Repositories browsing



More Integration Features

- Automatic synchronization with the maven POM



- Executing maven goals
- Indexing of local and remote repositories

More Integration Features

- Automatic artifacts download
  - Classes, Sources, Javadocs
- Profiles support
- Code completion

```
<dependency>
    <groupId>hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <version>1.8.0.</version>
    <scop
</depende
```

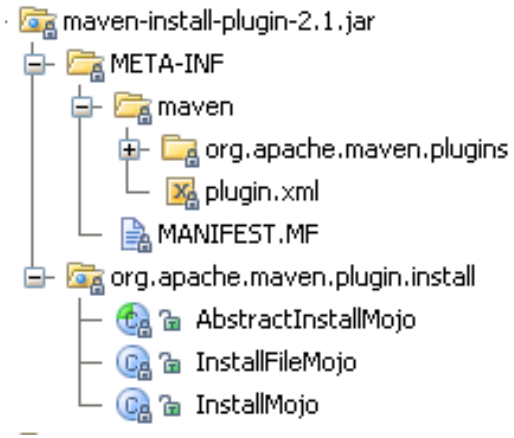
```
1.8.0.1
1.8.0.4
1.8.0.5
1.8.0.7
```

Maven 2 Plugins Development

What is a Maven Plugin?

What is a Maven Plugin?

- Maven Plugin is a JAR containing a plugin descriptor and one or more MOJOs
- MOJO – Maven Old Java Object



What is Maven Plugin ..

- Each MOJO is equivalent to a goal and is identified by a unique goal name

xfire-maven-plugin

Goal	Description
xfire:wsdlgen	WsdlGen mojo. Implemented as a wrapper around the XFire WsdlGen Ant task.
xfire:wsgen	WsGen mojo. Implemented as a wrapper around the XFire WsGen Ant task.

- A Mojo specifies metadata about a goal: the goal name, which phase of the lifecycle it fits into, and the parameters it is expecting

The Plugin Framwork

- Maven is a framework for collection of Maven plugins.
- Maven Provides the following services to the plugins:
  - Build Lifecycle
  - Dependency Management
  - Parameters Resolution and Injection

Your First MOJO

Your First MOJO

- We can create a new plugin project using the archetype plugin:

```
mvn archetype:create
   -DgroupId=com.mycompany.maven.plugins
   -DartifactId=maven-message-plugin
   -DarchetypeGroupId=org.apache.maven.archetypes
   -DarchetypeArtifactId=maven-archetype-mojo
```

Message MOJO Example

MessageMojo.java

```
package com.mycompany.maven.plugins;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;

/**
 * Goal that prints a message to the console.
 * @goal message
 */
public class MessageMojo extends AbstractMojo {

    public void execute() throws MojoExecutionException {
        getLog().info ("Message MOJO");
    }
}
```

Message MOJO Example

- Extends the abstract class AbstractMojo which provides most of the infrastructure except the execute() method
- The javadoc annotation @goal is required and defines the mojo goal name
- The execute method can throw two exceptions:
  - MojoExecutionException: if unexpected error occurred. Will fail with “Build Error” message
  - MojoFailureException: if expected error occurred. Will fail with “Build Failed” message

The Plugin POM

pom.xml

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.mycompany.maven.plugins</groupId>
    <artifactId>maven-message-plugin</artifactId>
    <packaging>maven-plugin</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>maven-message-plugin Maven Mojo</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>org.apache.maven</groupId>
            <artifactId>maven-plugin-api</artifactId>
            <version>2.0</version>
        </dependency>
    </dependencies>
</project>
```

The Plugin POM

- The plugin unique identity is composed of the groupId, artifactId and version
- Packaging – should be maven-plugin
  - Similar to jar packaging but add plugin descriptor generation phase
- Dependencies – The dependencies list must include the maven-plugin-api for AbstractMojo and related classes

Executing Plugins

- Executing directly from the command line
- Attaching the plugin to the project’s build lifecycle

Executing from the Command Line

- Execute with the fully qualified name of the goal:

```
mvn com.mycompany.maven.plugins:maven-message-plugin:1.0-
   SNAPSHOT:message
```

```
C:\WINDOWS\system32\cmd.exe

C:\project>mvn com.alphacsp.maven.plugins:maven-message-plugin:1.0-SNAPSHOT:message
[INFO] Scanning for projects...
[INFO] ----------------------------------------------------------------------------
[INFO] Building Training Project
[INFO]    task-segment: [com.alphacsp.maven.plugins:maven-message-plugin:1.0-SNAPSHOT:message]
[INFO] ----------------------------------------------------------------------------
[INFO] [message:message]
[INFO] Message MOJO
[INFO] ----------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ----------------------------------------------------------------------------
[INFO] Total time: < 1 second
[INFO] Finished at: Sun Jan 07 09:51:52 IST 2007
[INFO] Final Memory: 1M/3M
[INFO] ----------------------------------------------------------------------------
```

Executing from the Command Line ..

- The version can be omitted from the command to execute the latest version found in the local repository
- The plugin's group id can be added to the list of group ids searched by Maven by adding to the settings.xml:

```
<pluginGroups>
  <pluginGroup>com.mycompany.maven.plugins</pluginGroup>
</pluginGroups>
```

```
mvn message:message
```

Attaching Plugin to the Build Lifecycle

- We can bind the plugin to the build lifecycle:

```
<build>
    <plugins>
        <plugin>
            <groupId>com.mycompany.maven.plugins</groupId>
            <artifactId>maven-message-plugin</artifactId>
            <version>1.0-SNAPSHOT</version>
            <executions>
                <execution>
                    <phase>compile</phase>
                    <goals>
                        <goal>message</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Attaching Plugin to the Build Lifecycle ..

- The plugin is attached to the compile phase
- The goal to execute is “message”

```
C:\WINDOWS\system32\cmd.exe

C:\project>mvn compile
[INFO] Scanning for projects...
[INFO] ----------------------------------------------------------------------
[INFO] Building Training Project
[INFO]    task-segment: [compile]
[INFO] ----------------------------------------------------------------------
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
Compiling 1 source file to C:\project\target\classes
[INFO] [message:message {execution: default}]
[INFO] Message MOJO
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Sun Jan 07 10:16:36 IST 2007
[INFO] Final Memory: 2M/11M
[INFO] ------------------------------------------------------------------------
```

Binding MOJO to Lifecycle

The @phase annotation binds the mojo to a particular phase

MessageMojo.java

```
package com.mycompany.maven.plugins;
import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
/**
 * Goal that prints a message to the console.
 * @goal message
 * @phase validate
 */
public class MessageMojo extends AbstractMojo {
    public void execute() throws MojoExecutionException {
        getLog().info ("Message MOJO");
    }
}
```

Lab 12 (if time allows): Writing Maven Plugin

MOJO Parameters

MOJO Parameters

- Maven can inject parameters to mojo
- Parameter’s values can come from:
  - Plugin configuration
  - System properties (command line)
  - Maven Plugin Framework

Defining Parameters within a MOJO

- To define a parameter add an instance variable and annotate it with the @parameter javadoc annotation:

```
/**
* Message to display
* @parameter
*/
private String message;
```

Configuring Parameter Values

- Configuration element of the plugin inside the pom.xml of the project:

```
<build>
    <plugins>
        <plugin>
            <groupId>com.mycompany.maven.plugins</groupId>
            <artifactId>maven-message-plugin</artifactId>
            <version>1.0-SNAPSHOT</version>
            <configuration>
                <message>My Message</message>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Configuring Parameters

- With the expression attribute we can also set the value using system property or maven variable

```
/**
* Message to display
* @parameter expression="${messagemojo.message}"
*/
private String message;
```

```
mvn message:message -Dmessagemojo.message="message"
```

Parameter Default Value

- The default-value attribute defines the default value to use in case the parameter value is not defined

```
/**
* Message to display
* @parameter expression="${messagemojo.message}"
*                    default-value="Message MOJO"
*/
private String message;
```

Parameter Value Priority

- If both expression and default-value attribute are defined maven will try to set the value in this order:
  1. Plugin configuration in the pom.xml
  2. Maven expression
  3. System property
  4. The default value

Using Java 5 Annotations – The Anno-Mojo

Message MOJO with Annotations

MessageAnnoMojo.java

```
package com.mycompany.maven.plugins;
import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
import org.jfrog.maven.annomojo.annotations.*;

@MojoGoal("messageanno")
@MojoPhase("validate")
public class MessageAnnoMojo extends AbstractMojo {

    @MojoParameter(expression = "${my.parameter}", defaultValue = "Message Mojo")
    private String message;

    public void execute() throws MojoExecutionException {
        getLog().info(message);
    }
}
```

Anno-Mojo Benefits

- Inheritance & code reuse between plugins
- All the benefits of Java 5.0:
  - Generics, auto-boxing
- Compile-time checks
- Delegation of execution to POJOs
- Natural Ant java tasks integration
  - not XML based
- Source URL:
  - http://mvn-anno-mojo.svn.sourceforge.net/viewvc/mvn-anno-mojo/trunk/

Configuring Maven to use Annotations

- Configuring maven-plugin-plugin:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-plugin-plugin</artifactId>
  <dependencies>
    <dependency>
      <groupId>org.jfrog.maven.annomojo</groupId>
      <artifactId>maven-plugin-tools-anno</artifactId>
      <version>1.3.1</version>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
</plugin>
```

Configuring Maven to use Annotations ..

- Configurir

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

- Adding dependency:

```
<dependency>
    <groupId>org.jfrog.maven.annomojo</groupId>
    <artifactId>maven-plugin-tools-anno</artifactId>
    <version>1.3.1</version>
</dependency>
```

Criticism

Maven2 Downsides

- Frameworkitis
- Tedious dependencies exclusion
- Corrupted local repository files
- Serialized artifacts download
- Profiles activation not flexible
- Still some bugs with snapshots and plugins updates
- Plugins documentation

Frameworkitis

- Frameworkitis is the disease that a framework wants to do too much for you or it does it in a way that you don't want but you can't change it. It's fun to get all this functionality for free, but it hurts when the free functionality gets in the way. But you are now tied into the framework. To get the desired behavior you start to fight against the framework. And at this point you often start to lose, because it's difficult to bend the framework in a direction it didn't anticipate... **The bigger the framework becomes, the greater the chances that it will want to do too much, the bigger the learning curves become, and the more difficult it becomes to maintain it.**

POM Verbosity

For example this:

```
<dependency>
    <groupId>mycompany</groupId>
    <artifactId>projectname</artifactId>
    <version>2008</version>
    <scope>provided</scope>
</dependency>
```

Can become:

```
<dependency groupId="mycompany" artifactId="projectname"
    version="2008" scope="provided"/>
```

Static Parent Versioning

- Why can't it be like this?

```
<parent>
    <groupId>parent</groupId>
    <artifactId>parent-id</artifactId>
    <version>${parent-version}</version>
</parent>
```

Thank You!

Q & A