Project build with Maven

D. Conan

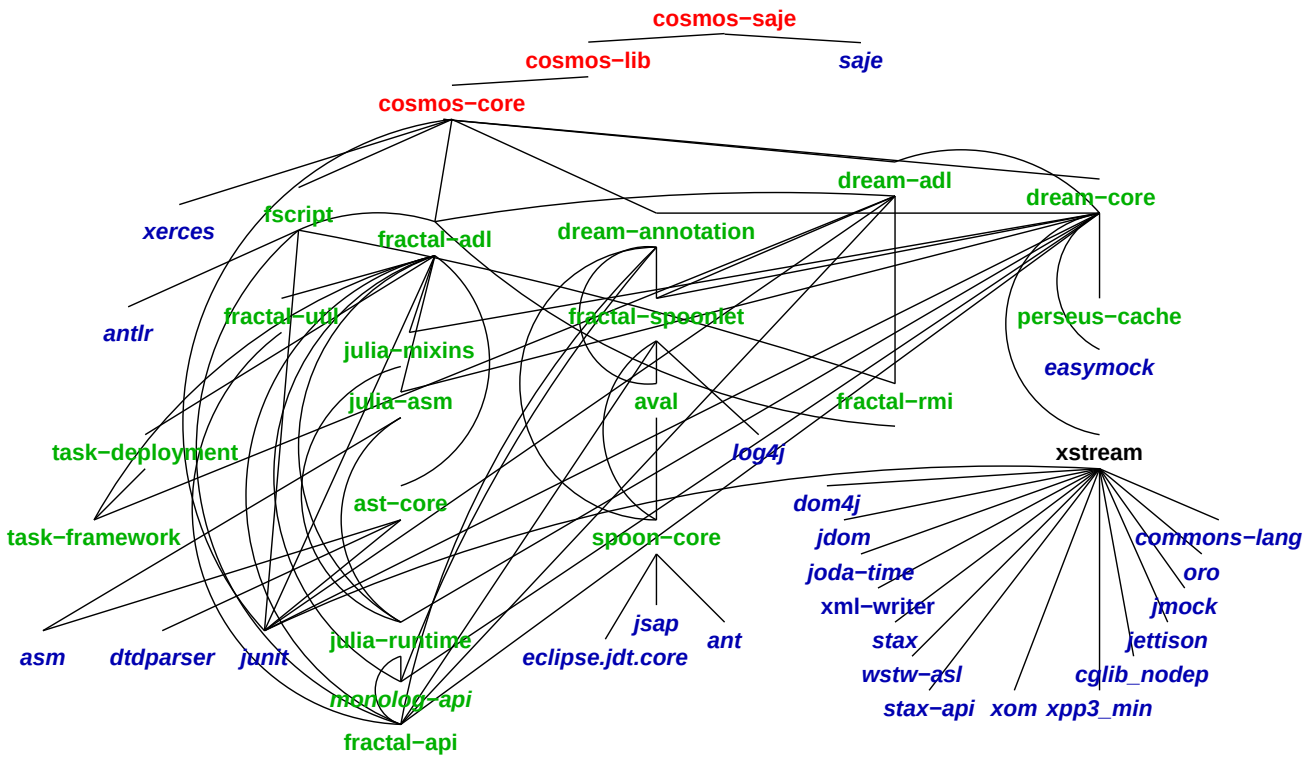
Internship course
March 2007

Outline

1 Motivations: Maven beyond Ant

- Simplification of builds, documentation, distribution and deployment
  - Declarative model for software projects: Project Object Model (POM)
    - Reuse of build logic and default strategies for the most common tasks
  - Tools/plug-ins that interact with the declarative model
    - There already exists a very long list of such plug-ins
  - Organisation of dependencies
    - Management the of *JAR hell*
    - Management the of *plug-in hell*
- Written in Java, dedicated for writing in Java
- Access to all the Ant tasks through the plug-ins `maven-ant-plugin` and `maven-antrun-plugin`

1.1 Jar hell and plug-in hell



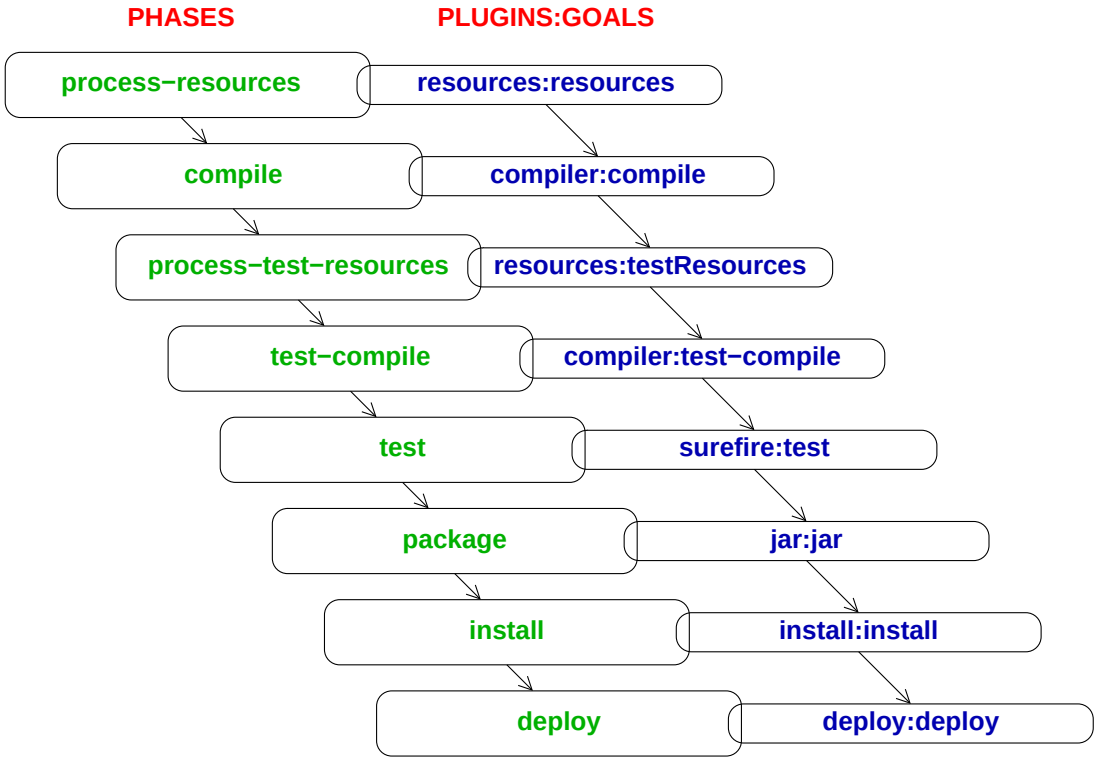
2 References and principles

- Documentation
  - Web site: `http://maven.apache.org`
  - Free book (license GPL) [van Zyl et al., 2007] `http://www.sonatype.com/book`
  - Another Free book (after registration) [Casey et al., 2006]
  - Articles and tutorials: `http://maven.apache.org/plugins`
- Main principles
  - Single project producing a single output (package type)
  - Standard life-cycles and naming conventions
  - Version management and dependency management
    - `groupId`: Fully qualified domain name such as `eu.it-sudparis`
    - Artifact name of the form `<artifactId>-<version>.<extension>`
  - Repositories: Several remote and one local (`~/.m2/repository` populated from remote ones)
    - No more `externals` or `lib` and no more `jars` in developer's directories

3 Build life-cycles, phases and goals

- Build life cycle: Ordered sequence of phases
  - One life cycle per package type
    - Default, jar, pom, ejb, war, ear, par
- Phase: A step in a build life cycle
  - May have zero or more goals bound to it
  - Executing a phase first execute all the preceding phases
  - `$ mvn compile`
- Goal: Maven unit of work bound to a phase
  - Bound goals are run when their phases execute
- Plug-in: May contain several goals
  - Executing a goal of plug-in does not execute preceding phases' goals
  - `$ mvn compiler:compile`

3.1 Exemplification with the JAR build life-cycle



3.2 Life-cycle for jar packaging

1. `process-resources`: Copy files and filter variables
   - `maven-resources-plugin`
2. `compile`: Compile project source code
   - `maven-compiler-plugin`
3. `process-test-resources`: Copy non-source-code for unit tests
   - `maven-resources-plugin`
4. `test-compile`: Compile unit-tests source code
   - `maven-compiler-pluging`
5. `test`: Execute project unit tests
   - `maven-surefire-pluging`
6. `package`: Create **the** `jar` of the project
   - `maven-jar-plugin`
7. `install`: Install the `jar` into local Maven repository `~/.m2/repository`
   - `maven-install-plugin`
8. `deploy`: Deploy the `jar` into a remote Maven repository for web access
   - `maven-deploy-plugin`

3.3 Default life-cycle

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. Prepare-package
15. Package
16. Pre-integration-test
17. Integration-test
18. Post-integration-test
19. Verify
20. Install
21. Deploy

3.4 Ways to manipulate build life-cycle *

- Five ways (not studied in this presentation, but cited here for the sake of completeness)
  1. Change project package
  2. Manually bind a goal to a phase in the POM
  3. Write a plug-in and bind in the goal definition in the plug-in Java class
  4. Create a forked life-cycle and execute it in a plug-in
  5. Create your own packaging type with a custom default life-cycle

4 Artifact names and project relationships

- Coordinates
  - `groupId`: Group, company, team, organisation, project, or other group
  - `artifactId`: A unique identification under the `groupId` representing a single project
  - `version`: A snapshot in time of the project in question
    - Version number = major.minor.bug fix-qualifier-build number
      - ★ *E.g.*, `1.0.1-20070514-1`
  - `packaging`: Type of build life-cycle this project will encounter
  - And sometimes `classifier`, *e.g.* `linux`, `windows`, etc.
- Three types of relationships between projects
  - Dependency: This project requires another one
  - Transitive dependencies: A dependency of a dependency
    - Maven does not allow circular dependencies
  - Parent: This project inherits some attributes from another project

5 Repositories

- Repository: A collection of installed or deployed project artifacts and other meta-data information
- Local:
  - System directory structure on the building machine containing all the artefacts that have been downloaded by Maven
  - Either required by plug-ins used, dependencies of project built
    Or manually installed projects via the install plug-in
  - `~/.m2/repository`
    - `groupId/artifactId/version/artifactId-version.packaging`
    - + a POM file describing meta-data, *e.g.* mainly dependencies

5.1 Local repository and snapshot versions

- A snapshot in Maven is an artifact that has been prepared using the most recent sources available
- Snapshot dependencies are assumed to be changing
- Specifying a snapshot version for a dependency = Maven looks for new versions
  - By default, on a daily basis
  - `-U` command line option to force for updates
- Specifying a non-snapshot version of a dependency = download that dependency only once
- `mvn -X plugin:goal` to display details, *e.g.* downloads to the local repository

6 Standard directory structure

- `pom.xml`: Maven POM (Project Object Model) file always at the top-level
- `LICENSE.txt`: License file is encouraged
- `README.txt`: Simple note to start with the project
- `src/main/java`: Source code of the project
- `src/main/filters`: Filters (properties files) of the project for the build
- `src/main/resources`: Project non-source code (resources)
- `src/main/config`: Project source code filters
- `src/main/filters`: Project resource filters
- `src/main/assembly`: Project assembly filters
- `src/test/java`: Source code of the unit tests
- `src/test/resources`: non-source code (resources) of the unit tests
- `src/test/filters`: Resource filters of the tests
- `src/site`: Resources used to generate the web site
- `target`: Target for all generated output: classes, web site...

7 POM and relationships

7.1 Aggregation in multi-module projects

- Similar to virtual packages in GNU/Linux packaging
- Declaring a parent project = packaging pom

helloworld/pom.xml

```
<project>
  <groupId>eu.it-sudparis</groupId>
  <artifactId>helloworld</artifactId>
  <packaging>pom</packaging>
  <version>0.1-SNAPSHOT</version>
  <name>Hello world example</name>
  <modules>
    <module>first-sub-module</module>
    <module>second-sub-module</module>
  </modules>
...
</project>
```

- mvn install copies pom.xml to ~/.m2/repository/helloworld/pom.xml

7.2 POM file of a submodule

- Declaring a parent module = inheriting parent configuration

helloworld/first-sub-module/pom.xml

```
<project>
  <parent>
    <groupId>eu.it-sudparis</groupId>
    <artifactId>helloworld</artifactId>
    <version>0.1-SNAPSHOT</version>
  </parent>
  <packaging>jar</packaging>
  <artifactId>first-sub-module</artifactId>
  <name>The first sub-module</name>
...
</project>
```

- Main configuration values, *e.g.*, `$user.home`, are inherited from the Super POM file
  - `help:effective-pom` displays the effective POM

7.3 Dependency declaration

- Add a dependency in the dependency graph with specifying a version

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

- Available scopes: compile (default = in all classpaths), provided (by the JDK or a container), runtime, test, system (artifact not in a repository but `systemPath` element to point in the local system file)

7.4 Dependency inheritance

- Submodules inherit dependencies of parent modules + transitive dependencies
  1. Inherit a version of a dependency from parent
  2. Declare a version dependency and use it
     - Use the dependency management section
       - State only the preference for a version, do not affect the dependency graph

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>otherorg.proj</groupId>
      <artifactId>artifactApp</artifactId>
      <version>1.0.9</version>
    </dependency></dependencies></dependencyManagement>
```

       - Declare that the dependency's usage, without any version number

```
<dependency>
  <groupId>otherorg.proj</groupId>
  <artifactId>artifactApp</artifactId>
</dependency>
```


7.5 Dependencies conflict resolution *

- Reasons for excluding transitive dependencies
  - Current project requires an alternately named version
    $\Longrightarrow$ two copies of a different versioned project in the classpath
  - Artifact unused
  - Artifact provided by your runtime container
  - An API which might have multiple vendors

```
<dependency>
  <groupId>otherorg.proj</groupId>
  <artifactId>artifactApp</artifactId>
  <version>1.0.9</version>
  <exclusions>
    <exclusion>
      <groupId>otherorg.proj</groupId>
      <artifactId>otherArtifactApp</artifactId>
    </exclusion></exclusions></dependency>
```

8 Plug-ins

8.1 Filtering resources: mvn process-resources *

- Filtering = Parse some files to fill value to be supplied at build time
  - In `pom.xml` of the resource directory

```
<properties><my.filter.value>hello</my.filter.value></properties>
<build>
  <filters><filter>src/main/filters/build.properties</filter></filters>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
  </resources></resources></build>
```

  - In an external file (*e.g.*, `build.properties`) or in the properties section of `pom.xml`
    - Syntax `${property name}`, *e.g.*, `application.name = ${project.name}`
  - In command line parameters (`"-Dmy.prop=hello world!"`)
- It is possible to prevent some files (*e.g.*, binary) from filtering

8.2 Compilation: mvn compile

- Allow JDK 5.0 source code

```
<build><plugin><plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>2.0</version>
  <configuration>
    <source>1.5</source>
    <target>1.5</target>
  </configuration></plugin></plugins></build>
```

- Another JRE with a fork

```
<configuration>
  <executable>/usr/.../javac</executable>
  <compilerVersion>1.5</compilerVersion>
  <fork>true</fork>
</configuration>
```

- Other configuration elements, *e.g.* compiler arguments

8.3 Unit tests: mvn test

- Surefire plug-in

```
<dependencies><dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.0</version>
    <scope>test</scope>
  </dependency></dependencies>
```

- By default, the following tests are included
  - `**/*Test.java`
  - `**/Test*.java`
  - `**/*TestCase.java`
- The following tests are excluded
  - `**/Abstract*Test.java`
  - `**/Abstract*TestCase.java`
- Skip the tests using the command line option `-Dmaven.test.skip=true`

8.4 Packaging: mvn package

- Include project classes in `target/classes`
- Include project resources in `src/main/resources`
- Insert a manifest automatically generated by Maven
- Insert a `pom.xml` and a `pom.properties` files
  - To be self-describing and to utilise meta-data during project execution
- To override the manifest file, in `pom.xml` of the project

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <configuration>
    <archive>
      <manifestFile>src/meta/META-INF/MANIFEST.MF</manifestFile>
    </archive></configuration></plugin>
```


8.5 Execution: mvn exec:java

- Plug-in `org.codehaus.mojo:exec-maven-plugin` in a profile

```
<profiles>
  <profile>
    <id>run</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <build><plugins><plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>exec-maven-plugin</artifactId>
          <executions>
            <execution>
              <goals><goal>java</goal></goals>
              <phase>runtime</phase>
            </execution></executions>
          <configuration>
            <mainClass>ClassWithTheMain</mainClass>
          </configuration></plugin></plugins></build></profile></profiles>
```

8.6 Installation (mvn install) and deployment (mvn deploy)

- Installation = local deployment in your local repository
  - In the directory `~/.m2/repository`
- Deployment = Internet deployment
  - See for instance `http://maven.objectweb.org/maven2`

8.7 Eclipse: mvn eclipse:eclipse|clean

- From the POM file `pom.xml`
  - Generation of `.classpath` and `.project` files
    - `mvn eclipse:eclipse`
  - To remove `.classpath` and `.project` files
    - `mvn eclipse:clean`
- One Eclipse project per Maven module

8.8 Helping

- `help:active-profiles` lists the profiles which are currently active for the build
- `help:effective-pom` displays the effective POM for the current build, with the active profiles factored in
- `help:effective-settings` prints out the calculated settings for the project
  - Any profile enhancement
  - The inheritance of the global settings into the user-level settings

  `help:describe` describes the attributes of a plug-in
  - `help:describe -Dplugin=compiler`
  - `help:describe -Dplugin=compiler -Dfull`
    - For discovering all of the goals of a plug-in as well as their parameters

8.9 More on plug-ins *

- The Maven command `mvn aname:agoal` executes the plug-in named
  - Either `maven-aname-plugin` or `aname-maven-plugin`
- Plug-ins are configured in POM files under the `<build>` element
  1. Configuring a plug-in or a goal already in the life-cycle

```
<build><plugins><plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <configuration>
        ...use properties declared by the plug-in...
      </configuration>
```

  2. Plug-in or goal not in the life-cycle, *e.g.* `mvn antrun:run`

```
<build><plugins><plugin>
  <artifactId>maven-antrun-plugin</artifactId>
  <configuration><tasks><echo>HOME: $env.JAVA_HOME</echo></tasks></configura
```

  3. The same as before, but bound to a phase: `<execution><goal>` elements
  4. The same as before + conditional execution

8.9.1 Properties

- `env.X`: Environment variable of the shell, *e.g.* `${env.PATH}`
- `project.x`: Project element value, *e.g.* `${project.groupId}`
- `settings.x`: Variables in `settings.xml`
- Java System Properties accessible via `java.lang.System.getProperties()`
- `x`: Property sets with POM element `<properties>`

9 Default life-cycle for project's web site generation *

1. `mvn site`: Generate all configured project reports
   - `maven-site-plugin`
2. `site-deploy`: Deploy the generated web site to the web server path specified in the POM distribution Management section
   - `maven-site-plugin`

- Lots of web site reports: Javadoc, JXR (source code cross reference), Checkstyle/PMD (source code formatting rules), Tag list (track TODO items), Cobertura (unit test's and execution's coverage), Surefire Report (visual result of unit tests) Clirr (compare jars for binary compatibility), Changes, etc.

9.1 Site descriptor *

- In file `src/site/site.xml`
  - `bannerLeft` and `bannerRight`: Appearance of the banner
  - `skin`: Skin used for the site
  - `publishtDate`: Format of the published date
  - `body/links`: Links displayed below the banner
  - `body/head`: Meta-data to be fed into the `<head>` element
  - `body/menu`: Menu items displayed in the navigation column
  - `body/${reports}`: To control which project reports are displayed in the web site

9.2 Web site reports *

- Javadoc: API reference from Javadoc
- JXR: Source code cross reference for any Java code
- Checkstyle: Check source code against formatting rules
- PMD: Checks source code against known rules for code smells (overlap with Checkstyle)
- CPD: Checks for duplicate source code blocks (copy/paste) (part of PMD)
- Tag list: Outstanding tasks or other markers (track `TODO` items)
- Cobertura: Analyse code statement coverage during unit tests or code execution
  - Help in identifying untested or unused source code blocks
- Surefire Report: Show the result of unit tests visually
- Clirr: Compare two versions of a `jar` for binary compatibility
- Changes: Produce release notes and road maps from issue tracking systems

10 More on Maven *

- Filtering resources: Parse some files to fill value to be supplied at build time
- Artifact deployment
- Creating plug-ins and inserting them in build life-cycles
  - In MOJOs (play-on-word of the Java term POJO [Plain Old Java Object])
- Migrating from Ant to Maven
- Creating a new packaging type
- Creating profiles, *e.g.* to add classifiers (*e.g.*, `linux` or `windows`)
- Team collaboration
  - Creating a shared repository
  - Creating an organisation POM
  - Continuous integration
  - Team dependency management using snapshots
  - Creating a standard project archetype

References

[Casey et al., 2006] Casey, J., Massol, V., Porter, B., Sanchez, C., and van Zyl, J. (2006). *Better Builds with Maven: The How-To Guide for maven 2.0*. Mergere Library Press.

[van Zyl et al., 2007] van Zyl, J., Casey, J., and Redmond, E. (2007). Maven: The Definitive Guide (1.0 Alpha 2). http://www.sonatype.com/book/. Creative Commons License.