



Maven 2

Part 1 – Evolution of the build process

Ondřej Žížka

Brno, February 2009

Agenda of part 1

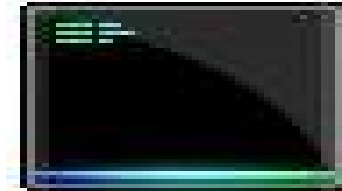
- Evolution of project build process
 - Common steps of build process
 - Command line
 - Make
 - Ant
 - Maven

Evolution of the build process

- What do we do during the build process?
- Build:
 - 1) Compile all *.java from src/ to target/classes/
 - Use lib/log4j1.2.8.jar in classpath
 - 2) Copy everything that's not *.java from src/ to target/
 - 3) Create .jar from the contents of target/classes
- Clean:
 - Delete the target/ directory

Evolution of the build process - cmdline

- Compilation, cleaning:

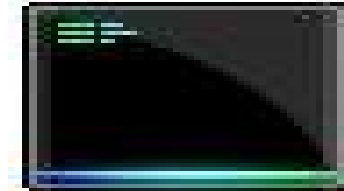


```
mkdir -p target/classes/  
javac -cp lib/log4j-1.2.8.jar src/ -d target/classes/  
cp `find src/ -type f | grep -v *.java` target/classes/  
jar -c target/example-1.0.jar target/classes/
```

```
rm -r target/
```

Evolution of the build process - make

- Compilation, cleaning:



```
CC=g++
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.cpp hello.cpp
OBJECTS=$(SOURCES:.cpp=.o)
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

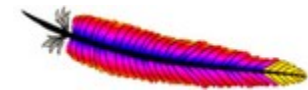
$(EXECUTABLE): $(OBJECTS)
    $(CC) $(LDFLAGS) $(OBJECTS) -o $@

.cpp.o:
    $(CC) $(CFLAGS) $< -o $@

clean:
    rm -rf *.o hello
```

Evolution of the build process - Ant

- Ant is (still) most used build tool for Java projects
- Defines targets and their dependencies
- Uses XML “scripts”
- Provides “tasks” to perform in targets
 - Easily extensible, many existing tasks



Evolution of the build process - Ant

```
<?xml version="1.0" encoding="utf8" ?>
<project name="example" default="compile" basedir=".>

    <property name="app.name" value="example"/>
    ...

    <target name="clean" description="Delete build directory">
        <delete dir="${build.home}" />
    </target>

    <target name="compile" depends="compile">
        <mkdir dir="${build.home}/classes/" />
        <javac srcdir="${basedir}/src/java" destdir="${build.home}/classes">
            <classpath>
                <pathelement location="${basedir}/lib/log4j-1.2.8.jar" />
            </classpath>
        </javac>
    </target>
    ...

</project>
```



Evolution of the build process - Maven

- Becoming de-facto standard in Java world
- Descriptive approach - project's characteristics in `pom.xml`
- Provides “plugins” to perform various operations
 - Easily extensible, many existing plugins
- Leads developers to a standard structure of project files
- Takes care of dependencies
 - Standardizes and automatizes their storage and downloading
- Handles projects hierarchy and inheritance
- Unobtrusive – usually only one extra file per project (`pom.xml`)

Maven's objectives

- Primary goal:
 - To allow a developer to comprehend the complete state of a development effort in the shortest period of time.
- Maven attempts to deal with:
 - Making the build process easy
 - Providing a uniform build system
 - Providing quality project information
 - Providing guidelines for best practices development
 - Allowing transparent migration to new features

Maven: Advantages

- When creating a project:
 - “Factory defaults” fit your needs in many cases
 - Unobtrusive – usually only one extra file per project (`pom.xml`)
 - Takes care of many areas of software project development
 - Good support for project modularization

- When building a downloaded project:
 - Once you have learned Maven, you understand build processes of all Maven projects
 - No need to search and download dependencies
 - Usually, `mvn package` really gives you the resulting binaries
 - Very good support in most relevant IDEs (Eclipse, NetBeans, IDEA)

Maven: Disadvantages

- Human factor: Incorrect metadata
 - When someone makes a mistake, you have to edit metadata by hand in `~/ .m2/`
- Human factor: Messy central repository
 - `<repo>/org/foo/bar/bar/` **vs.** `<repo>/bar/bar/`
- Very very long and verbose `pom.xml`
 - No wonder - it has to contain all project's information
- Not appropriate when you want to build dependencies from source
 - Fedora
 - solution: Maven-JPP

Maven vs. Ant + Ivy comparison

- Ant + Ivy: Script-like tool + dependency tool
- So, we can compare only dependency part of Maven.
- Ant + Ivy:
 - More freedom, more versatile; But is it really necessary?
 - More properly written dependency metadata (at least as cited on Ivy site)
 - Ivy uses “configurations” → Maven uses “profiles”; Seems equally powerful.
 - Pluggable dependency manager and conflict manager
- Maven:
 - Less freedom → less care: Let's rely on defaults
 - Version conflict policy: Use the closest in the dependency tree
 - Mess in metadata; Ivy site says about Maven:

The only problem some may face is that module descriptors are not always checked, so some are not really well written.

The Maven logo, featuring the word "maven" in a bold, sans-serif font. The letter 'a' is orange, while the other letters are black.

Maven 2

Part 2 – Introducing Maven 2

Ondřej Žížka

Brno, February 2009

Agenda of part 2

- What is Maven
- Project Object Model (POM)
- Simple Maven project
- Inter-project relations
 - Dependencies
 - Inheritance
 - Modules
- Maven plugins
- Profiles
- Maven repositories
- Maven command line usage
- Maven in JBoss projects

What is Maven

- Apache's description:
 - “Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.”
- Maven's goal is to standardize the build process with all of it's aspects and provide easy-to-use tool to perform this process.
- Maven's “sphere of interest” includes:
 - Code generation
 - Compilation
 - Coverage reports
 - Integration tests
 - Deployment
 - IDE integration
 - Source code management
 - Test database initialization
 - Project packaging
 - Project's site content generation
 - Collaboration
 - Issue management (JIRA, ...)
 - ...

Project build lifecycle

- When Maven is run, it progresses from the first lifecycle phase to the specified one.
 - ``mvn compile`` invokes all phases up to `compile`.
- Default build lifecycle phases: →
- The phases correspond to the usual build scenario.

maven

```
validate
initialize
generate-sources
process-sources
generate-resources
process-resources
compile
process-classes
generate-test-sources
process-test-sources
generate-test-resources
process-test-resources
test-compile
process-test-classes
test
prepare-package
package
pre-integration-test
integration-test
post-integration-test
verify
install
deploy
```


Some build lifecycle phases

initialize	Initialize build state, e.g. set properties or create directories.
compile	Compile the source code of the project.
process-classes	Post-process the generated files from compilation, for example to do bytecode enhancement on Java classes.
test-compile	Compile the test source code into the test destination directory
test	Run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed.
package	Take the compiled code and package it in its distributable format, eg. JAR.
integration-test	Process and deploy the package if necessary into an environment where integration tests can be run.
verify	Run any checks to verify the package is valid and meets quality criteria.
install	Install the package into the local repository, for use as a dependency in other projects locally.
deploy	Done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

Plugins

- Maven is just a shell that invokes plugins “goals” (sim. to Ant's tasks)
- Goals can be bound to the phases of the lifecycle
- Executing goals bound to a phase: `mvn <phase>`
- Default goal binding:

<code>process-resources</code>	<code>resources:resources</code>
<code>compile</code>	<code>compiler:compile</code>
<code>process-test-resources</code>	<code>resources:testResources</code>
<code>test-compile</code>	<code>compiler:testCompile</code>
<code>test</code>	<code>surefire:test</code>
<code>package</code>	<code>ejb:ejb or ejb3:ejb3 or jar:jar or par:par</code>
<code>install</code>	<code>install:install</code>
<code>deploy</code>	<code>deploy:deploy</code>

Plugins

- Both plugin and its goal's "execution" can be customized
 - `<configuration/>`
- Well known plugins repositories:
 - Apache: <http://maven.apache.org/plugins/>
 - "Core" plugins – compiler, clean, jar, ...
 - Codehaus: <http://mojo.codehaus.org/plugins.html>
 - jboss – operates Jboss server
 - sql – performs SQL operations, puts database into a known state
 - xslt – performs XSLT transformations
 - cargo – deploys applications (WAR, EAR, ...) to the containers
 - findbugs, ...
 - EL4J: <http://el4j.sourceforge.net/plugins/index.html>
 - and others...



Standard directory layout

- All sources are in `/src`
- All build output goes to `/target`
- Follows commonly used structure:

```
|-- pom.xml
|-- target
|-- src
|   |-- main
|       |-- java
|           |-- org
|               |-- jboss
|                   |-- mavenapp
|                       |-- App.java
|               |-- webapp
|                   |-- WEB-INF
|                       |-- web.xml
|   |-- test
|       |-- java
|           |-- org
|               |-- jboss
|                   |-- mavenapp
|                       |-- AppTest.java
```

maven

Artifacts – basic elements of project

- Basic construction element is called *artifact*
- Artifact can be:
 - an archive - .jar, .war, .ear, .sar, ...
 - or a POM – used as a node in the graph of relations between projects.
- Artifacts build up a graph of relations:
 - dependencies, parent, modules.
- Each artifact is defined by one XML file – `pom.xml` – and one directory.
- Minimal `pom.xml` example:

```
<project>
  <groupId>      org.jboss      </groupId>
  <artifactId>    myapp          </artifactId>
  <packaging>     jar            </packaging>
  <version>       1.0-SNAPSHOT   </version>
</project>
```



POM - Project Object Model

- Describes the artifact, how to build it, where to put the result, who is the developer, how to test it, ... Defaults taken from the “super-pom”
- Basic structure: (org/apache/maven/project/pom-4.0.0.xml)

```
<project>
  <groupId>org.jboss</groupId>
  <artifactId>myapp</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>

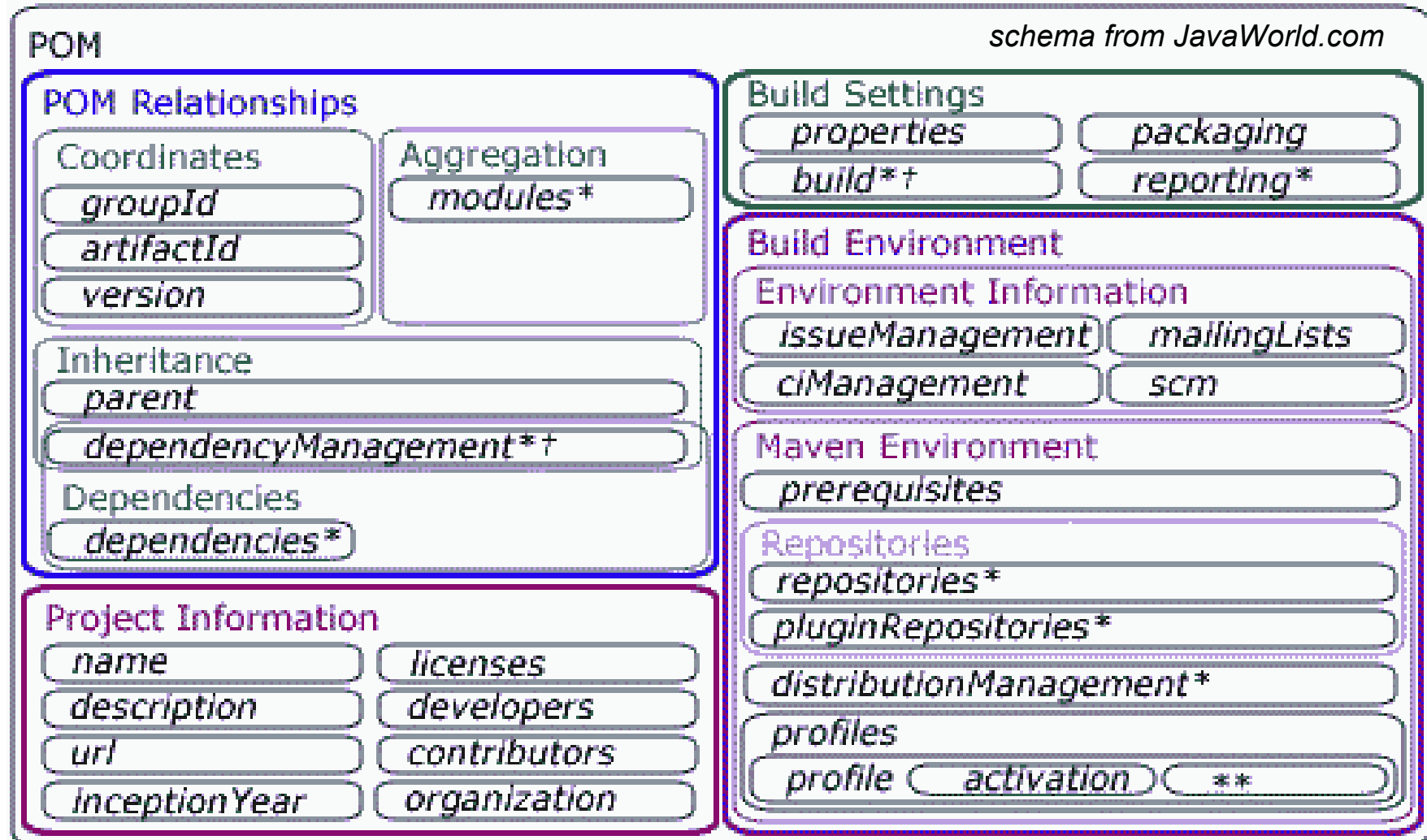
  <build>
    <pluginManagement/>
    <sourceDirectory>src/main/java</sourceDirectory>
    <testSourceDirectory />
    <resources/>
    <outputDirectory>target/classes</outputDirectory>
    <finalName>
      ...
  </build>

  <reporting/>
  <profiles/>
  <repositories/>
  ...
</project>
```

The Maven logo, consisting of the word 'maven' in a bold, sans-serif font. The 'm' is black, and the 'a' is orange.



POM - Project Object Model



* Element may be overridden (at least mostly) by *profile* element settings

** Profile elements are the *-suffixed elements

† Contains elements for meant for inheritance

Inter-project relations

- **Dependency:** `<dependency>`
 - Use an artifact (a library) in your project

- **Inheritance:** `<parent>`
 - Share project characteristics
for more artifacts

- **Modules:** `<modules>`
 - Split integral project
into several artifacts

```
<project>

  <dependencies>
    <dependency>
      <groupId>...</groupId>
      <artifactId>...</artifactId>
      <version>...</version>
    </dependency>
    ...
  </dependencies>

  <parent>
    <groupId>...</groupId>
    <artifactId>...</artifactId>
    <version>...</version>
  </parent>

  <modules>
    <module>...<module>
    ...
  </modules>

</project>
```




Dependencies

- Project depends on artifacts
- Dependencies are downloaded from repositories
- Dependencies are described in POM: (*italics* = optional)

```
<project>
  <dependencies>

    <dependency>
      <groupId>org.jboss.jsfunit</groupId>
      <artifactId>jboss-jsfunit-richfaces</artifactId>
      <version>1.0.0.GA-SNAPSHOT</version>
      <scope>compile</scope>
      <classifier>jdk15</classifier>
      <optional>>false</optional>
      <exclusions>...</exclusions>
      <type>jar</type>
    </dependency>
    ...
  </dependencies>
</project>
```

The Maven logo, consisting of the word "maven" in a bold, lowercase, sans-serif font. The letter "a" is orange, and the letters "m", "v", "e", and "n" are black.



Dependencies

```
<dependency>
  <groupId>org.jboss.jsfunit</groupId>
  <artifactId>jboss-jsfunit-richfaces</artifactId>
  <version>1.0.0.GA-SNAPSHOT</version>
  <scope>compile</scope>
</dependency>
```

- **<groupId>** – group name
 - Usually matches package name.
 - Groups together related artifacts of a software project.
- **<artifactId>** – name of the artifact
 - Usually matches final archive file name.
- **<version>**
 - should follow the `a.b.c.d[-SNAPSHOT]` notation
 - `-SNAPSHOT` is parsed token – this is how Maven recognizes snapshots!
 - `a.b.c.d` is parsed for numbers, used in comparisons
 - Range can be used: `(1.0, 2.0]` etc. See [www docs](#).



Dependencies

```
<dependency>
  <groupId>org.jboss.jsfunit</groupId>
  <artifactId>jboss-jsfunit-richfaces</artifactId>
  <version>1.0.0.GA-SNAPSHOT</version>
  <scope>compile</scope>
</dependency>
```

■ <scope> – when to use this dependency

- **One of:** compile (default), test, provided, runtime, system, import
- **compile:** dependency is available on all classpaths
- **provided:** like compile, but not packaged; eg. Servlet API for web apps
- **runtime:** Available for tests, not for compiler; eg. JDBC drivers
- **test:** avail. only for test phases; typically, testing frameworks like JUnit
- **system:** like provided, but not downloaded
- **import:** used in <dependencyManagement> to affect transitional deps.



Dependencies

```
<dependency>
  <groupId>org.jboss.jsfunit</groupId>
  <artifactId>jboss-jsfunit-richfaces</artifactId>
  <version>1.0.0.GA-SNAPSHOT</version>
  <scope>compile</scope>
</dependency>
```

- This is enough information for Maven to: *
- Find the dependency and check available version
- Download if not in local repository
- Use it in proper situations (copilation, tests, packaging)
- Resolve transitive dependencies
- *) if metadata are OK – unfortunately, often they're not :-(

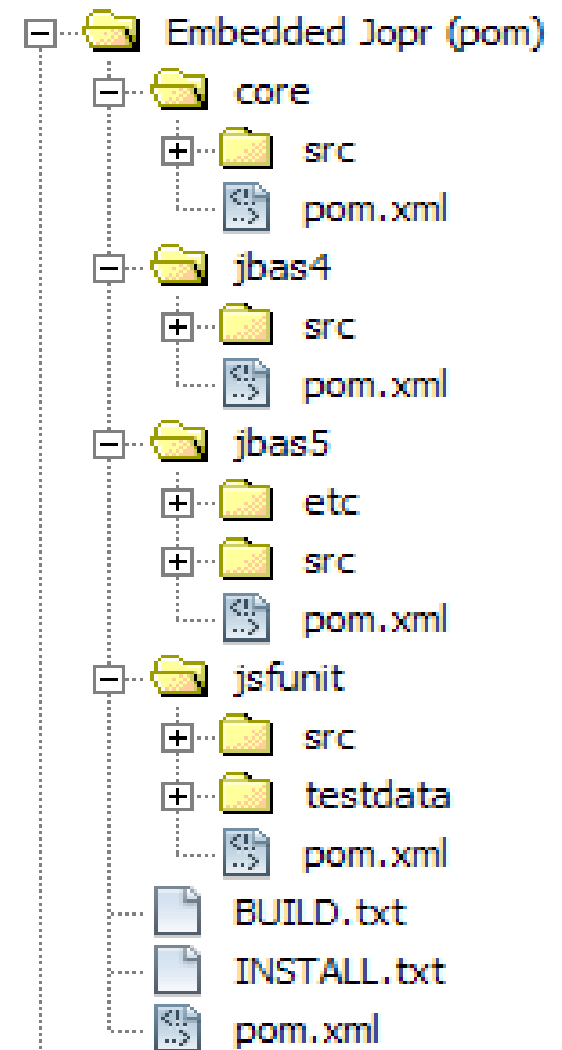
maven

Structured project: Modules

- Used to split project to several parts
 - In opposite to `<parent>` inheritance, `<modules>` makes the “root” project aware of it's “children” - building root also builds modules

```
<project>
...
  <modules>
    <module>core</module>
    <module>jbas4</module>
    <module>jbas5</module>
    <module>jsfunit</module>
  </modules>
</project>
```

- Modules are built in order of appearance
- Thus, modules can depend one on each other
 - e.g. `jsfunit` module needs `core` and one of `jbas`



Structured project: Modules

- “Root” pom.xml can also be used as a parent.
- `<embjopr>/pom.xml`:

```
<project>
```

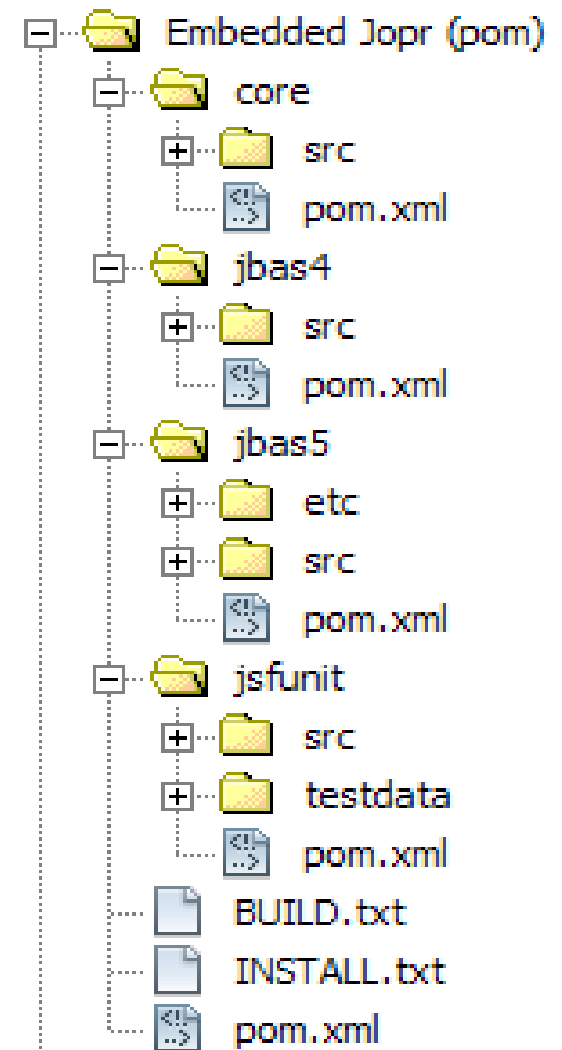
```
  <groupId>org.jboss.jopr</groupId>  
  <artifactId>jopr-embedded-parent</artifactId>  
  <version>1.1.0-SNAPSHOT</version>  
  <packaging>pom</packaging>
```

```
  <modules>
```

```
    <module>core</module>  
    <module>jbas4</module>  
    <module>jbas5</module>  
    <module>jsfunit</module>
```

```
  </modules>
```

```
</project>
```



Structured project: Modules

- Sub-directories `<embjopr>/<module-name>/` contain module's `pom.xml`
- Example - `<embjopr>/jsfunit/pom.xml`:

```
<project>
```

```
  <!-- Module's "root" project -->
```

```
  <parent>
```

```
    <groupId>org.jboss.jopr</groupId>
```

```
    <artifactId>jopr-embedded-parent</artifactId>
```

```
    <version>1.1.0-SNAPSHOT</version>
```

```
  </parent>
```

```
  <groupId>org.jboss.jopr</groupId>
```

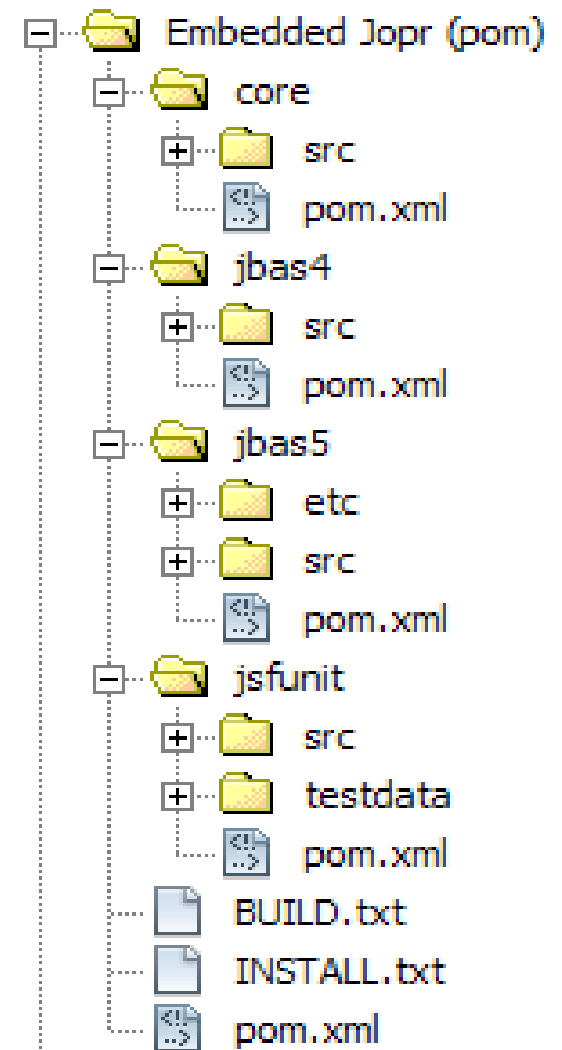
```
  <artifactId>jopr-embedded-jsfunit</artifactId>
```

```
  <version>1.1.0-SNAPSHOT</version>
```

```
  <packaging>war</packaging>
```

```
  ...
```

```
</project>
```



Structured project: Parent

- By using `<parent>`, you can inherit arrangements of other project.
 - POM elements merged from parent: plugins (lists, executions w/ matching IDs, configuration), resources, dependencies, developers
- Relation is “semi-transitive” – parent's parent is my... grandpa :-)
- See EmbJopr's pom.xml:

```
<project>
```

```
<!-- NOTE: We extend the RHQ parent pom, because we essentially  
want all the same base settings - plugins, dependencies, etc. -->
```

```
<parent>
```

```
  <groupId>org.rhq</groupId>  
  <artifactId>rhq-parent</artifactId>  
  <version>1.2.0-SNAPSHOT</version>
```

```
</parent>
```

```
<groupId>org.jboss.jopr</groupId>  
<artifactId>jopr-embedded-parent</artifactId>  
<version>1.1.0-SNAPSHOT</version>
```

```
...
```

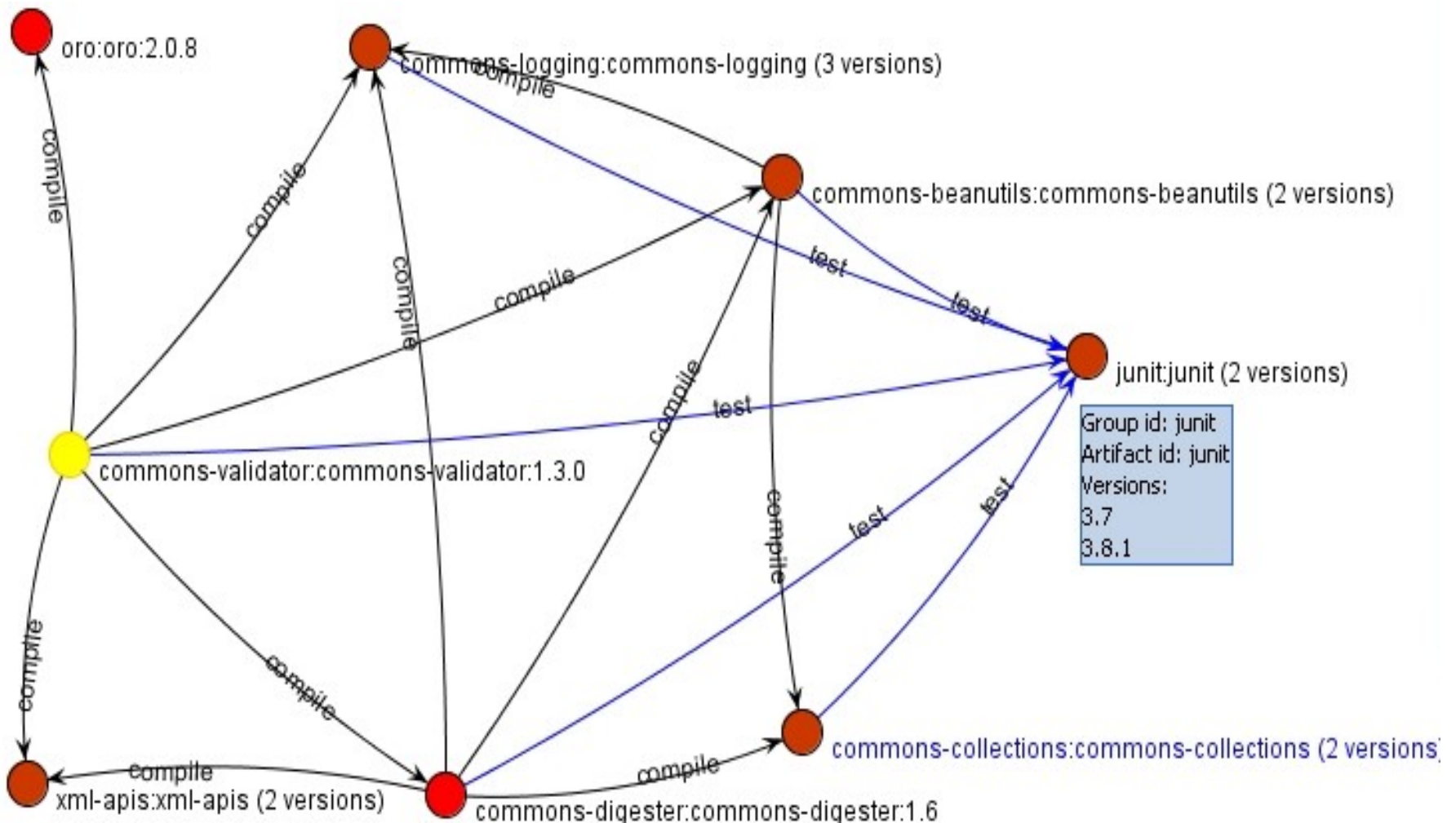
```
</project>
```


Dependency conflicts

- What happens when the parent project has different dependency than the child project has?
 - In Maven, every project is built separately, using closest dependency settings.
 - When creating app distribution, the behavior depends on the respective plugin.

Dependency Analyzer for Maven

- Creates graphical representation of dependencies
- <http://www.jfrog.org/sites/dep-analyzer/1.0/screenshots.html>



Maven plugins

- Again - maven is a shell that just executes plugin's goals.
- Plugin's behavior affected mainly by
 - Overall plugin <configuration/>
 - Specific goal execution <configuration/>

```
<project>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <compilerVersion>1.5</compilerVersion>
          <source>1.5</source>
          <target>1.5</target>
          <encoding>utf-8</encoding>
        </configuration>
      </plugin>
    </build>
  </project>
```

Maven plugins

- Plugin is also an artifact, though treated specially:
 - Special repository settings

```
<project>

  <repositories>
    <repository>...</repository>
  </repositories>

  <pluginRepositories>
    <pluginRepository>...</pluginRepository>
  </pluginRepositories>

</project>
```

Plugin configuration + *binding* example

- Binding plugin goal to a build phase:
- Plugin vs. goal execution configuration:

```
<project> <reporting> <plugins> <plugin>

    <!-- Code quality metrics -->
    <artifactId>maven-pmd-plugin</artifactId>

    <configuration>
        <targetJdk>1.5</targetJdk>
    </configuration>

    <executions>
        <execution>
            <phase>validate</phase>
            <goals><goal>cpd</goal></goals>
            <configuration>
                <outputDirectory>target/pmd-reports</outputDirectory>
            </configuration>
        </execution>
    </executions>
</plugin> </plugins> </reporting> </project>
```

Profiles

- Intended to affect build lifecycle
- Means of affection:
 - `${properties}` → `<properties>...</properties>`
 - plugin/goal configuration → `<build><plugins>...`
 - dependencies, repositories, and few others
- Activate a profile by: `mvn -P<profile-name> ...`

```
<project>
  <profiles>
    <profile>
      <id>my-profile</id>
      <properties>...</properties>
      <build>
        <plugins>...</plugins>
      </build>
      ...
    </profile>
  </profiles>
</project>
```

Profiles – use cases

- Build / test / create distribution with different libs (dependencies)
 - Example: Hibernate's cache and pooling libraries
- Build different mutation of your application
 - Example: EmbJopr WAR builds for Jboss AS 4.x and 5.x
- Create app distribution with different files
 - Example: Different defaults in a .properties file
 - Example: Different Look & Feel (CSS and images)
- Deploy to other server
 - Example: Regular internal build / Release and publish on public repository
- Test WAR against different application servers
- ...etc.

Profiles - drawbacks

- Use of profiles makes `pom.xml` very verbose

- Many of the elements must be repeated
- Not always avoidable by using `${properties}`

- Limited possibility to move to `profiles.xml`

- Bad news: Maven people *like it* this way

- Good news: You'll get used to it

- Future hopes:

- Short-cutting with XML namespaces (like in Spring)
- XSLT preprocessor for `pom.xml`




```
<profiles>
  <profile>
    <id>jboss4.2</id>
    <dependencies>
      <dependency>
        <groupId>org.jboss.jopr</groupId>
        <artifactId>jopr-embedded-jbas4</artif
        <version>${project.version}</version>
        <type>war</type>
      </dependency>
    </dependencies>
    <build>
      <plugins>
        <plugin>
          <groupId>org.codehaus.cargo</groupId>
          <artifactId>cargo-maven2-plugin</art
          <version>1.0-SNAPSHOT</version>
          <configuration>
            <wait>false</wait>
            <container>
              <containerId>jboss42x</container
              <home>${JBOSS_HOME}</home>
              <log>${basedir}/target/jboss4.2.
              <timeout>300000</timeout>
              <systemProperties>
                <jsfunit.testdata>${basedir}/t
              </systemProperties>
            </container>
          </configuration>
          <executions>
            <execution>
              <id>start-container</id>
              <phase>pre-integration-test</pha
              <goals>
```


Maven repositories

- Every artifact Maven works with must be in *some* repository.
- Repository is a storage of Maven artifacts.
- It's has well-defined directory structure and naming schema.
- Full path to an artifact:

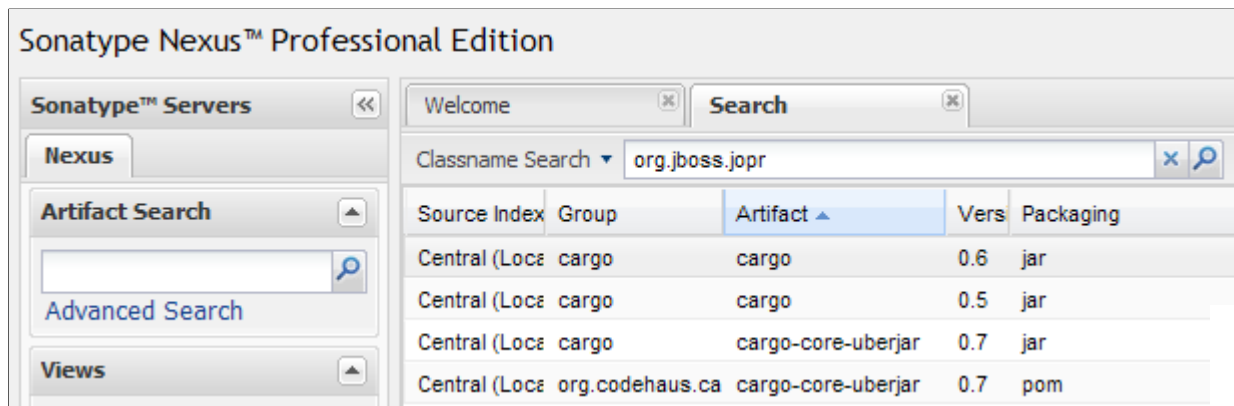
```
<repository>/<groupId-with-slashes>/<artifactId>/<version>/  
<artifactId>-<version>[-SNAPSHOT] [-<classifier>].<packaging>
```

Index of /maven2/org/hibernate/hibernate-core/3.3.1.GA

	hibernate-core-3.3.1.GA-javadoc.jar	10-Sep-2008	17:23	7.3M
	hibernate-core-3.3.1.GA-javadoc.jar.md5	10-Sep-2008	17:23	32
	hibernate-core-3.3.1.GA-javadoc.jar.sha1	10-Sep-2008	17:23	40
	hibernate-core-3.3.1.GA-sources.jar	10-Sep-2008	17:23	2.0M
	hibernate-core-3.3.1.GA-sources.jar.md5	10-Sep-2008	17:23	32
	hibernate-core-3.3.1.GA-sources.jar.sha1	10-Sep-2008	17:23	40
	hibernate-core-3.3.1.GA.jar	10-Sep-2008	17:23	2.2M

Maven repositories

- Local repository: cache of artifacts on local filesystem
 - Default: `~/.m2/repository`; override in `settings.xml`: `<localRepository/>`
 - Gets filled as you download from remote repos; safe to delete any time
 - Secret tip: Backup sometimes, and if something goes wrong with SNAPSHOTS, restore.
- Remote repository: Usually a simple HTTP(S) server
 - Use `<repositories>` and `<pluginRepositories>` to set where to get artifacts from
- Proxies / caches / searchable indexes:
 - Artifactory, Archiva, *Nexus (ClassName search!)*
 - <http://java11.englab.brq.redhat.com:1212/nexus>



Maven command line usage

- `mvn [options] [<goal(s)>] [<phase(s)>]`
- Phases: phases of Maven lifecycle (see before) – e.g. `install`
- Goals: IDs of plugin goals (see before) – e.g. `jboss:start`
- Important options:
 - `-P<profile, ...>` activates given profiles.
 - `-D<name=value>` defines a property (usage: `<target>${name}</target>`)
 - `-U, --update-snapshots` bypasses local cache
- Examples:
 - `mvn clean install` – invokes `clean` phase of the `clean` lifecycle, then `install` phase of the `build` lifecycle
 - `mvn javadoc:jar` – invokes the `jar` goal of the `javadoc` plugin
 - `mvn -Pjboss4x -Djboss.home=./jboss-eap-4.3.0 install`
 - `mvn clean compile jboss:start test jboss:stop`

Maven help tools

■ Show plugin configuration options:

- `mvn help:describe -DgroupId=org.apache.maven.plugins -DartifactId=maven-compiler-plugin -Dfull=true`

■ Show effective POM:

- `mvn help:effective-pom`

■ Show effective settings:

- `mvn help:effective-settings`

■ Show artifact dependencies:

- `mvn -X package`

Maven project as Hudson job

- Hudson has a Maven plugin
 - Possible to create a job using only pom.xml
- However, many times it's easier to run Maven from a bash script
 - e.g. when you need to wget / unzip / check out something before running Maven – it's easier to write few bash lines than to create an extra profile for Hudson.
 - Example: Embedded Jopr's Hudson job:

```
### Build JBoss AS, checked out / updated by Hudson
cd jbossas-5.x/build
./build.sh -Dbuild.unsecured=true
cd ../..
```

```
### Set JBoss AS home dir
export JBOSS_HOME=`pwd`/`ls -d -1 jbossas-5.x/build/output/* | tail -n 1`
```

```
cd embjopr
mvn install -Dmaven.test.skip=true -DJBOSS_HOME=$JBOSS_HOME
cd jsfunit
mvn -Pjboss5x install
sleep 5
```

Maven in JBoss projects

- Currently we're moving most of our project to Maven.
 - <http://wiki.jboss.org/wiki/Wiki.jsp?page=MavenizationStatus>
- Some projects still use Ant, which leads to complexities in build system.
- JBoss projects exploit Maven up to its limits
 - Sometimes we have to fall back to calling Ant tasks – see `antrunner` plugin.
- Seam build:
 - All dependencies are built from source and put to jboss.org repository.
 - Central Maven repository mirrored to jboss.org.
- Java projects in Fedora/RHEL: Even wilder
 - Dependencies parsed from `pom.xml`, recursively
 - Their source code downloaded in `.rpm` and built into local repository
 - Results copied to wherever yum likes them to be
 - But, source codes mostly not present in Maven repos
 - → Lots of manual work

Maven in JBoss projects

- Citation from Maven site (“What is Maven”):

Maven does encourage best practices, but we realize that some projects may not fit with these ideals for historical reasons. While Maven is designed to be flexible, to an extent, in these situations and to the needs of different projects, it can not cater to every situation without making compromises to the integrity of its objectives.

If you decide to use Maven, and have an unusual build structure that you cannot reorganise, you may have to forgo some features or the use of Maven altogether.

- From the same page:

Maven is a tool that can now be used for building and managing any Java-based project. We hope that we have created something that will make the day-to-day work of Java developers easier and generally help with the comprehension of any Java-based project.

•

- Which one fits better for us?

- Time will tell...



Thanks for attention! Questions?

maven

- What is Maven
- Project Object Model (POM)
- Simple Maven project
- Inter-project relations
 - Dependencies
 - Inheritance
 - Modules
- Maven plugins
- Profiles
- Maven repositories
- Maven command line usage
- Maven in JBoss projects

Resources

- Maven book: <http://www.sonatype.com/books/maven-book/reference/>
- Maven site: <http://maven.apache.org/>
- Maven tips: <http://cvs.peopleware.be/training/maven/maven2/>
- Maven based JBoss build system:
 - <http://jboss.org/community/docs/DOC-11358>
- Codehaus plugins: <http://mojo.codehaus.org/>
- SoftEU.cz: <http://blog.softeu.cz/prednasky/2006/maven/> - czech intro

Maven 2

Part 3 – How to do ... with Maven 2?

... see you later!

Ondřej Žížka

Brno, February 2009

Agenda of part 3

- Order of goals execution
- Order of repositories being queried for artifacts
- How do I put files in .jar (on classpath)?
- How to *Mavenize* a legacy project? (example: Smack library)
- Maven plugins – finest selection :-)
 - Surefire
 - Cargo
- ...stay tuned.

Deploying a file as an artifact

```
mvn deploy:deploy-file  
-Durl=scpexe://yourhost.com/path/to/maven2/repository  
-DrepositoryId=somerepo-ssh  
-DgroupId=javax.j2ee  
-DartifactId=javaee  
-Dversion=5.0.FCS  
-Dpackaging=jar  
-Dfile=e:/temp/javaee-5.0.FCS.jar
```

Maven 2

Part 4 – IDEs support for Maven ...demonstrations.

Ondřej Žížka

Brno, February 2009

Agenda of part 4

- NetBeans Maven support: *Mevenide*
 - Project structure
 - Editing pom.xml
 - POM XSL
 - Plugin configuration probe
 - Repository artifacts probe
 - Enabling profiles, binding goals to IDE actions
- IDEA Maven support
 - <http://plugins.intellij.net/plugin/?id=1166>
- Eclipse Maven support
 - <http://m2eclipse.codehaus.org/>