Maven 2 – The powerful buildsystem

a presentation for EL4J developers

by Martin Zeltner (MZE)

6 October 2006

ELCA We make it work.

Agenda

- Introduction 5’
- Installation 20’
- Concepts 40’
- Where to find … 5’
- Daily usage 30’
- Advanced usage 15’
- (Plug-in development 30’)

Introduction

- Welcome to Maven
  - 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 project description.
  - In this context, when we talk about Maven we mean Maven 2.
- License
  - The Apache Software License, Version 2.0
- Homepage
  - http://maven.apache.org
- Versions
  - 2.0.4 (10.04.2006)
  - 2.1-SNAPSHOT (14.08.2006)
    ➔ We use this self-built version. It contains about 15-20 bug fixes (see http://jira.codehaus.org/browse/MNG-1412 and our global pom.xml)

Maven 2 – The powerful build system

1. Installation
2. Concepts
3. Where to find …
4. Daily usage
5. Advanced usage
6. (Plug-in development)

Installation prerequisite

- Download and install the Subversion tool, available at **`http://subversion.tigris.org/`**
- Choose a location where to checkout the new EL4J (internal and external stuff) configured for Maven 2 (i.e. **`D:/Projects/EL4J`**) and create this directory path. We name this path **`EL4J_ROOT`**.
- Open a command line in **`EL4J_ROOT`** and execute

```
svn checkout https://svn.sourceforge.net/
    svnroot/el4j/trunk/el4j external
```


Installation (1)

- Download the **maven-2.x-bin.zip** from the Maven Homepage (**`http://maven.apache.org/`**) or it's contained in the convenience.zip or the essential.zip of EL4J you can get from sourceforge.
- Create a directory path without spaces in directory names (i.e. **`EL4J_ROOT/external/external-tools/maven`**) and unzip it there. We name this path **`M2_HOME`**.
- Check if the files "**`mvn.bat`**" and "**`mvn`**" (shell script) are in directory path **`M2_HOME/bin`**.
- Set environment variable **`M2_HOME`** to its path.
- Set environment variable **`JAVA_HOME`**. It must point to the directory path of a Java5 JDK.
- Prepend the following to environment variable **`PATH`**:
  - Windows:
    - **`%JAVA_HOME%\bin;%M2_HOME%\bin;`**
  - Unix:
    - **`$JAVA_HOME/bin:$M2_HOME/bin:`**

Installation (2)

- Open a command line/shell and check the JDK version (should be the same)
  - java –version
  - javac –version
- Copy file **`M2_HOME/conf/settings.xml`** to **`~/.m2/`**
- A must for Windows users only
  - Choose a location where you would like to have the local repository and create this path (i.e. D:\m2repository). Do not use whitespaces! We name this path **`M2_REPO`**.
  - Edit file **`~/.m2/settings.xml`**
    - Uncomment element **`localRepository`** and set its value to **`M2_REPO`**.

Installation (3)

- Edit file **`~/.m2/settings.xml`**
  - Append as last element of **`settings`** the following snippet

```
<pluginGroups>
  <pluginGroup>org.codehaus.cargo</pluginGroup>
  <pluginGroup>ch.elca.el4j.plugins</pluginGroup>
</pluginGroups>
```

  - Append the following snippet as child of element profiles

```
<profile>
  <id>el4j.general</id>
  <properties>
    <el4j.home>???</el4j.home>
    <tomcat5x.basedir>${el4j.home}/external/
      external-tools/tomcat</tomcat5x.basedir>
    <tomcat5x.version>5.5.17</tomcat5x.version>
    <tomcat5x.home>${tomcat5x.basedir}/apache-tomcat-${tomcat5x.version}/
      apache-tomcat-${tomcat5x.version}</tomcat5x.home>
    <tomcat5x.zipDownloadUrl>http://www.apache.org/dist/tomcat/
      tomcat-5/v${tomcat5x.version}/bin/
      apache-tomcat-${tomcat5x.version}.zip</tomcat5x.zipDownloadUrl>
    <tomcat5x.container.log.file>${project.build.directory}/
      logs/tomcat5x/output.log</tomcat5x.container.log.file>
    <tomcat5x.cargo.log.file>${project.build.directory}/
      logs/tomcat5x/cargo.log</tomcat5x.cargo.log.file>
  </properties>
</profile>
```

  - Adapt the value of element **`el4j.home`** to the root directory of EL4J (**`EL4J_ROOT`**), i.e. **`D:/Projects/EL4J`** (use slashes on windows too!).

Installation (4)

- Edit file `~/.m2/settings.xml`
  - Append as last element of **`settings`** the following snippet

```
<activeProfiles>
  <activeProfile>el4j.general</activeProfile>
</activeProfiles>
```

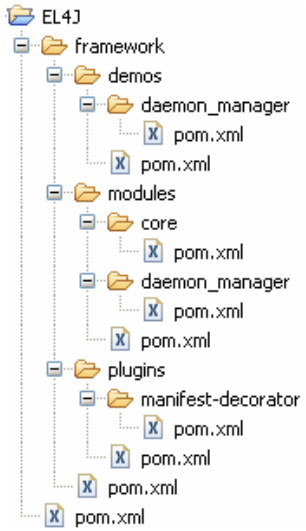
  ➔ Is used to permanently activate profile **`el4j.general`** (so even if you enable other profiles explicitly, this profile remains active; this is an exception: when you explicitly chose some profiles, the default profiles are disabled)

- Verify on the command line that Maven works
  - **`mvn --version`**
- Let Maven work in the background while we look at the concepts
  - **`cd EL4J_ROOT/external`**
  - **`mvn install`**

Maven 2 – The powerful build system

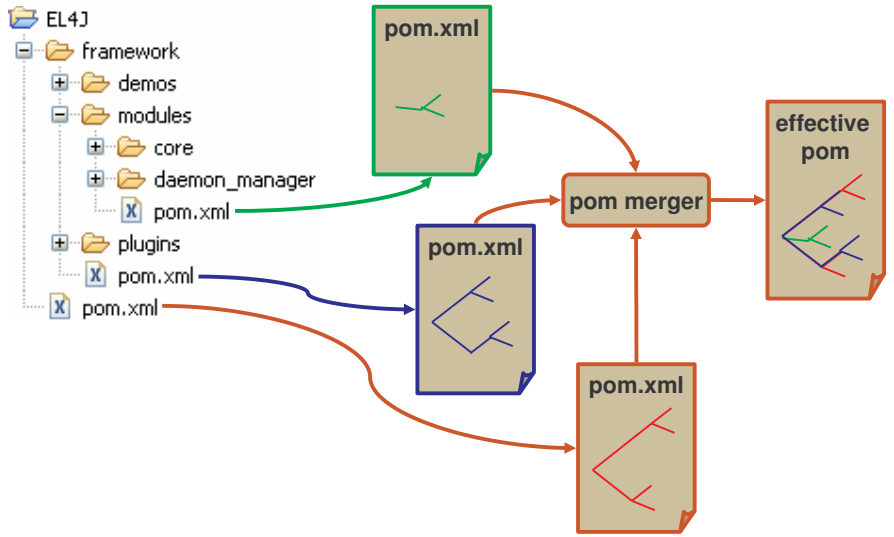
1. Installation
2. Concepts
3. Where to find …
4. Daily usage
5. Advanced usage
6. (Plug-in development)

Concepts (1) - Inheritance



- Every “pom.xml” file contains Maven configuration. In Maven 1 this file had the name “project.xml”.
- The project “EL4J” contains the “root pom” (meaning that it has no parent pom). It has one module, the “framework”.
- The “framework” itself is also a project, depends on “EL4J” and has the three modules “plugins”, “modules” and “demos”. And so on …

Concepts (2) – Merging of pom files



Concepts (3) – Properties

A *property* is an important concept in Maven. A property is a name-value pair (e.g. el4j-home = c:/Projects/EL4J/checkout)

Properties can be set in various ways

- In the global settings file (in `M2_HOME/conf/settings.xml` and `~/.m2/settings.xml`, the second one has precedence over the first)
- In the properties section of `pom.xml` files
- In profiles (in the global settings or in `pom.xml` files)
- On the command line with the prefix –D (e.g. –Dname=value). These properties will actually be standard Java System Properties. When looking up a maven property, java system properties are always checked first.

How to access properties

- In the normal strings of the pom.xml files you can always refer to a property via ${*name*} where *name* is the name of the property
- When certain files are copied, a *filter* applies, i.e. occurences of properties in the form ${*name*} are replaced

Concepts (4) – Properties

How to see all active properties defined?

- Please refer to the `help:effective-settings` goal

What files are filtered when copying (in EL4J)

- All files under `src/main/env` and `src/test/env`
- Please refer also to the env module (it defines support for different environments)

Concepts (5) – Maven model

```
<?xml version="1.0" ... ?>
<project xmlns="http://ma..."
  xmlns:xsi="http://www.w3..."
  xsi:schemaLocation="...xsd">

  <modelVersion/>
  <parent>
    <groupId/>
    <artifactId/>
    <version/>
    <relativePath/>
  </parent>

  ...

</project>
```

- The “pom.xml” file is a **schema validated** xml file.
- **modelVersion**
  Currently for Maven 2 it must be set to “4.0.0”. “3.0.0” is for Maven 1.1.
- **parent**
  Optionally points to the parent pom (the parent pom must be of type pom).
  - **relativePath**
    Is the location where the parent can be found (no must). Default: “../pom.xml”
  - groupId, artifactId, version
    ➔ see next slide…

Concepts (6) – Maven model

```
<project>

  ...

  <groupId/>
  <artifactId/>
  <version/>
  <packaging/>

  ...

</project>
```

- **groupId**
  Identifier such as “ch.elca.el4j.modules”
- **artifactId**
  Identifier such as “module-core”
- **version**
  Identifier such as “1.2-SNAPSHOT”
- **packaging**
  The type of the current artifact (pom):
  - **jar**
    Is the default. Means that this artifact contains java source files to compile.
  - **pom**
    For artifacts just used as descriptor. Normal for projects that are not “leafs” of the artifact hierarchy.
  - Further types:
    war, ear, maven-plugin

Concepts (7) – Maven model

```
<project>

  ...

  <build>
    <sourceDirectory/>
    <scriptSourceDirectory/>
    <testSourceDirectory/>
    <outputDirectory/>
    <testOutputDirectory/>

    ...

  </build>

  ...

</project>
```

- **build**
  Contains the info how to build the current artifact.
  - **sourceDirectory**
    Contains java source files.
    Default: “src/main/java”
  - **scriptSourceDirectory**
    Contains script files.
    Default: “src/main/scripts“
  - **testSourceDirectory**
    Like “sourceDirectory” but for test sources. Default: “src/test/java”
  - **outputDirectory**
    Where to compile java sources and copy scripts and other resources.
    Default: “target/classes”
  - **testOutputDirectory**
    Like “outputDirectory” but for the test part. Default: “target/test-classes”

Concepts (8) – Maven model

```
<project>

  ...
  <build>
    ...

    <defaultGoal/>
    <resources/>
    <testResources/>
    <directory/>
    <finalName/>

    ...
  </build>
  ...
</project>
```

- **build**
  - **defaultGoal**
    Is the goal to execute if no goal is defined on the command line. Goals will be explained later. There’s no global default.
  - **resources**
    Points to the resource directories. Content will be copied to the “outputDirectory”.
    By default: “src/main/resources”
  - **testResources**
    Points to test resource directories. Their content will be copied to the “testOutputDirectory”.
    By default: “src/test/resources”
  - **directory**
    Top-level directory where to put built parts.
    Default: “target”
  - **finalName**
    The name to use for built objects like jar, war and ear. Default: ${artifactId}-${version}

Concepts (9) – Maven model

```
<project>

  ...
  <build>
    ...

    <filters/>
    <plugins/>
    <pluginManagement>
      <plugins/>
    </pluginManagement>
  </build>
  ...
</project>
```

- **build**
  - **filters**
    Points to property files used for filtering.
    Filtering will be explained later.
  - **plugins**
    Are the plugins to be used in this artifact.
    These plugins join the Maven lifecycle.
    Typically plugins will not be configured here but only within the pluginManagement/plugins section.
  - **pluginManagement**
    - **plugins**
      Same as the plugins before but these plugins do not join the Maven lifecycle.
      Typically plugins are preconfigured here.

Concepts (10) – Maven model

```
<project>

  ...

  <profiles/>
  <modules/>
  <repositories/>
  <pluginRepositories/>

  ...

</project>
```

- **profiles**
  Contains *profiles* that can be dynamically activated by setting a property, via a jdk version, an os type or the presence of a file. A profile contains normal artifact content, it can *override* other artifact content.
- **modules**
  Are the child artifacts of the current artifact. It is required to add them.
- **repositories**
  Are the locations from where artifacts can be downloaded. These repositories are used for artifacts that are not maven plugins.
- **pluginRepositories**
  Same as “repositories” but only used to download maven plugin artifacts.

Concepts (11) – Maven model

```
<project>

  ...

  <dependencies/>
  <dependencyManagement/>
    <dependencies/>
  <reporting/>
  <properties/>

</project>
```

- **dependencies**
  Are the artifacts the current artifact depends on. Such an artifact has a scope i.e. `test` so it is only in classpath for testing (i.e. JUnit). The default scope is `compile`, meaning that the artifact is always in the classpath.
- **dependencyManagement**
  - **dependencies**
    Same as previous but the current artifact does not have a dependency to them. It is used to preconfigure dependencies, used in child artifacts. Analogue to "plugins" and "pluginManagement".
- **reporting**
  Are special Maven plugins used for site generation. They join the Maven lifecycle like plugins referenced in previously shown "plugins" element.
- **properties**
  Are name-value-pairs that can be used to simplify configuration.

Concepts (12) – Maven lifecycles

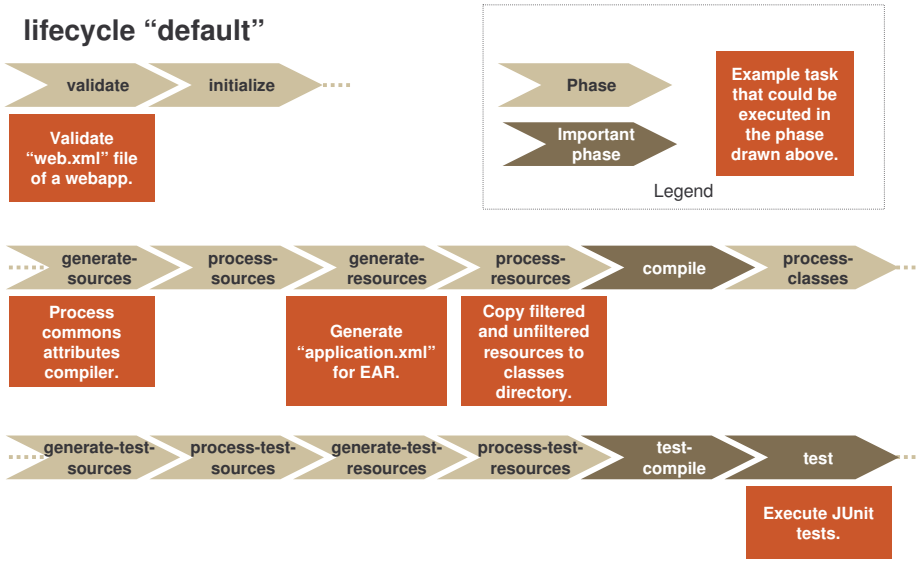
Maven knows by default the following three *lifecycles*

- **default**
  Is used for most activities on artifacts like performing a traditional build.
- **clean**
  Is mostly used to delete generated parts.
- **site**
  Is used to generate a website for the current artifact.

A lifecycle has one or more *phases*, and a plugin can *join* a phase. Typically, when phases of the lifecycles above are activated, some predefined plugin-goals are automatically executed.
More about this on the next slides…

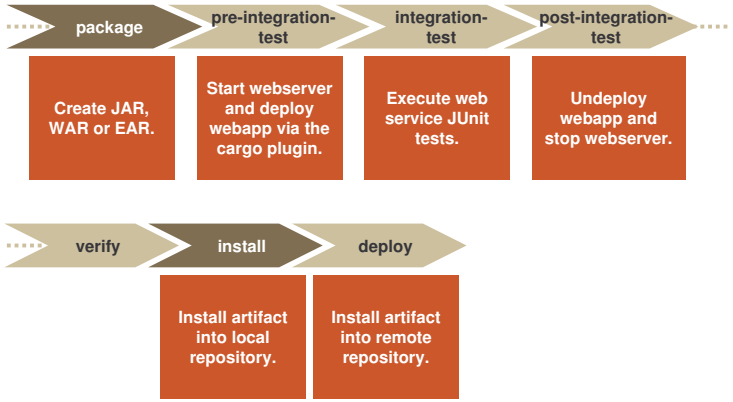
Concepts (13) – Maven lifecycles

lifecycle "default"



Concepts (14) – Maven lifecycles

lifecycle "default" (2)



Concepts (15) – Maven lifecycles

lifecycle “clean”

Concepts (16) – Maven lifecycles

lifecycle “site”

Using mvn on the command line

How to launch maven on the command line:

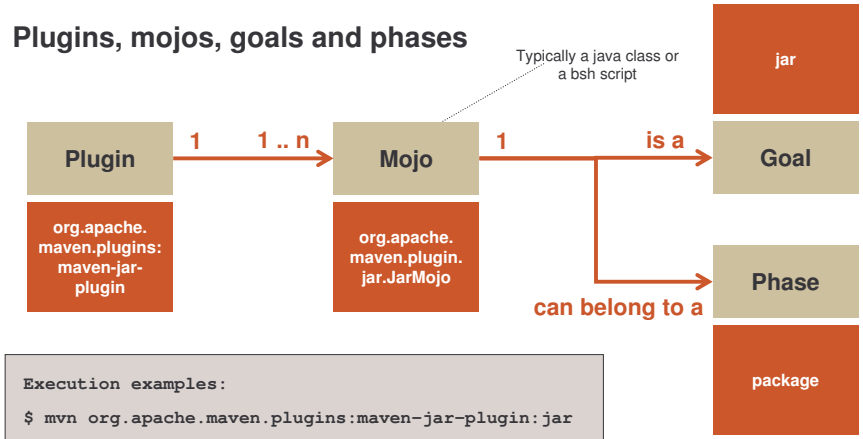
`mvn <goal>` : execute the goal `<goal>`

`mvn <phase>` : execute up to phase `<phase>`

You can combine multiple goals or phases on the command line such as `mvn clean install`

Concepts (15) – Maven plugins

Plugins, mojos, goals and phases




```
Execution examples:
$ mvn org.apache.maven.plugins:maven-jar-plugin:jar
$ mvn jar:jar
$ mvn package
➔ Attention: All goals of mojos bound to
phase <= "package" will be executed!
```

Concepts (16) – Maven plugins

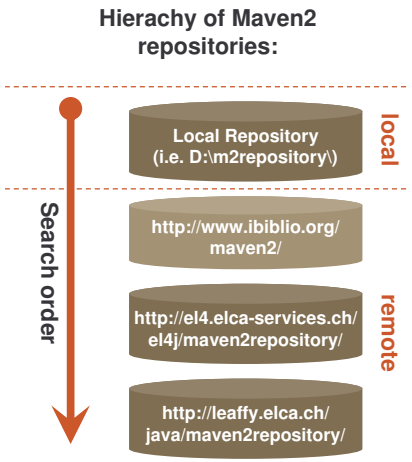
Short plugin list	
Plugin	**Description**
antrun	Run a set of ant tasks from a phase of the build.
assembly	Build an assembly (distribution) of sources and binaries.
checkstyle	Generate a checkstyle report.
clean	Clean up after the build.
compiler	Compiles Java sources.
deploy	Deploy the built artifact to the remote repository.
ear	Generate an EAR from the current project.
eclipse	Generate an Eclipse project file for the current project.
ejb	Build an EJB (and optional client) from the current project.
help	Get information about the working environment for the project.
install	Install the built artifact into the local repository.
jar	Build a JAR from the current project.
javadoc	Generate Javadoc for the project.
jxr	Generate a source cross reference (analog to javadoc).
resources	Copy the resources to the output directory for including in the JAR.
site	Generate a site for the current project.
source	Build a JAR of sources for use in IDEs and distribution to the repository.
surefire	Run the Junit tests in an isolated classloader.
war	Build a WAR from the current project.

Concepts (17) – Artifact lookup

If an artifact has a dependency to another artifact or a plugin, Maven will go through the given repositories until it finds the requested artifact.

As shown in Maven model we can have separate repositories for plugins and their dependencies and separate repositories for all other dependencies.

In EL4J we use the “ibiblio” repository only as “pluginRepository” to prevent having unexpected dependencies.



Maven 2 – The powerful build system

1. Installation
2. Concepts
3. Where to find …
4. Daily usage
5. Advanced usage
6. (Plug-in development)

Where to find … (1)

- Documentation
  - Better Builds with Maven – The How-To guide for Maven 2
    - http://www.mergere.com/m2book_download.jsp
      ➔ Free book in PDF format from
  - Getting started guide
    - http://maven.apache.org/guides/getting-started/index.html
  - Available plugins from Apache
    - http://maven.apache.org/plugins/index.html
  - Available plugins from Codehaus
    - http://mojo.codehaus.org/
- Issue Management
  - Maven Components
    - http://jira.codehaus.org/browse/MNG
  - Other Maven Technologies and Maven Plugins
    - http://jira.codehaus.org/secure/BrowseProjects.jspa
      ➔ Go to categories “Maven Technologies” and “Maven 2 plugins”

Where to find … (2)

- Plugins
  - Use Google to find out available plugin versions
    http://www.google.ch/search?q=site:ibiblio.org/maven2+MY SEARCH QUERY
- Help
  - Subscribe to the very active Maven user mailing list (users@maven.apache.org).
  - Use Google to find help in Maven user mailing list
    http://www.google.ch/search?q=site:http://mail-archives.apache.org/mod_mbox/maven-users+MY SEARCH QUERY
    - To only get messages from 2006 just modify the URL a bit
      http://www.google.ch/search?q=site:http://mail-archives.apache.org/mod_mbox/maven-users/2006+MY SEARCH QUERY

Where to find what is inside Maven 2

- Plexus Container
  - http://plexus.codehaus.org/
  - Plexus is similar to other *inversion-of-control* (IoC) or *dependency injection* frameworks such as the Spring Framework (http://www.springframework.org).
  - Used for configuration.
- Maven Wagon
  - http://maven.apache.org/wagon/
  - Maven Wagon is a transport abstraction that is used in Maven's artifact and repository handling code.
  - Used to down- and upload artifacts.
  - Protocols file, http, https, ftp, sftp and scp are available.

Maven 2 – The powerful build system

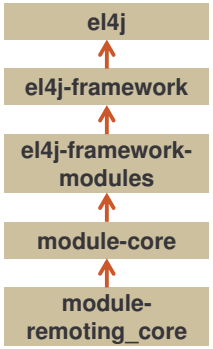
1. Installation
2. Concepts
3. Where to find …
4. Daily usage
5. Advanced usage
6. (Plug-in development)

Daily usage (1)

- `cd EL4J_ROOT/external`
- `mvn -N install`
  - Take a look at your local repository.
- `mvn -N`
- `mvn install -Dmaven.test.skip=true`
- `cd framework/modules/core`
- `mvn clean`
  - Inspect content of current directory.
- `mvn compile`
  - Inspect directory target.
- `mvn test-compile`
  - Inspect directory target.
- `mvn surefire:test`
- `mvn test`
  - What is the difference between "`mvn surefire:test`" and "`mvn test`"?

Daily usage (2)

- **`mvn package`**
  - Inspect directory target.
- Remove directory **`el4j-framework-modules`** from local repository (**`M2_REPO/ch/elca/el4j/modules`**).
- **`mvn clean`**
- **`mvn compile`**
  - What happens? Why?
- **`cd ../remoting_core`**
- **`mvn clean`**
- **`mvn compile`**
  - What happens? Why?
- **`cd ..`**
- **`mvn -N install`**
- **`cd core`**
- **`mvn install -Dmaven.test.skip=true`**
- **`cd ../remoting_core`**
- **`mvn install -Dmaven.test.skip=true`**



Daily usage (2 – answers to questions)

Certain targets of the compile phase require another project to exist. The target `clean` does not require a project, but the target `compile` needs other projects (it needs the compiled code in order to run).

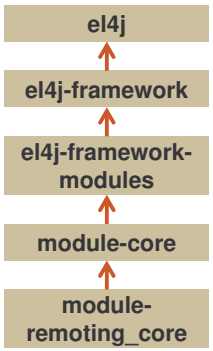
Maven does only build other projects in certain cases! The previous slide illustrates the 2 different cases:

- In the first case it works, because mvn directly looks in the direct directory or pom hierachy (but not in all transitive dependencies!) **This is different from EL4Ant behavior!**
- In the second case it does not work, because the dependency is not in the direct hierarchy of the artifact.

Remark: we would actually prefer the earlier EL4Ant behavior and will look into how to achive it. For now we keep the maven convention as it directly follows from some core maven hypothesis.

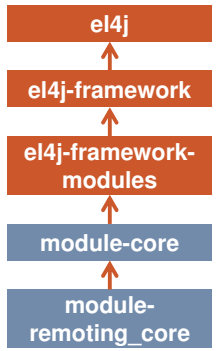
Daily usage (3)

- **`cd ../core`**
- **`mvn site`**
  - Inspect directory target.
  - What is meant with argument **`site`**?
- **`cd ../remoting_core`**
- Eclipse
  - Start Eclipse with workspace **`EL4J_ROOT/external/framework/workspace`**
  - Import your preferences.
  - Close Eclipse.
  - **`mvn -N eclipse:add-maven-repo -Declipse.workspace= "EL4J_ROOT/external/framework/workspace"`**
    - By this command the classpath variable **`M2_REPO`** has been added to the given workspace.
  - Start Eclipse again with the same workspace.



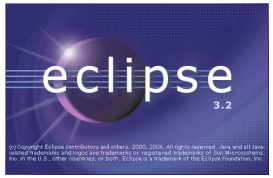
Daily usage (4)

- **`mvn eclipse:clean eclipse:eclipse`**
  - Creates the Eclipse project newly for **`module-remoting_core`**
  - Import this project in opened workspace. Does the project compile in Eclipse?
- **`cd ..`**
- **`mvn eclipse:clean eclipse:eclipse`**
  - Go into Eclipse and refresh project **`module-remoting_core`**
  - Does the project still compile in Eclipse?
  - Import project **`module-core`** in Eclipse and refresh both projects.



Daily usage (5)

- Eclipse issues
  - Eclipse is just a helper for Maven, it is not a replacement!
    - Examples: Commons Attributes code generation, filtering of environment files.
  - Executed tests in Eclipse can have different results than executed tests with Maven. The relevant results are the one from Maven. There can be various causes for this behaviour:
    - Eclipse projects don't separate compile and test scope but Maven does. Can be dangerous i.e. if dir test resources contains Spring bean xml files in directory "mandatory"!
    - Maven does always have dependent artifacts as jar files in classpath. In Eclipse, depending to execution level/directory of the "mvn eclipse:eclipse" command, some dependencies are in classpath as jar and some directly as directory with its classes. The test classes itself are always via directory in classpath.
    - Eclipse has its own compiler. There are some cases (specially Java 5 syntax) tests work if classes compiled with Eclipse compiler and don't work if classes are compiled with Sun's compiler. The relevant compiler is the one from Sun.



Daily usage (6) – Problem solving

- `mvn -N help:describe -DgroupId=... -DartifactId=... -Dfull=true`
  - Describs all goals of the given plugin (groupId & artifactId). Example:
    ```
    mvn -N help:describe -DgroupId=ch.elca.el4j.plugins
        -DartifactId=maven-env-support-plugin -Dfull=true
    ```
- `mvn -N help:describe -DgroupId=... -DartifactId=... -Dmojo=... -Dfull=true`
  - Describs the given goal (aka mojo) of the given plugin. Example:
    ```
    mvn -N help:describe -DgroupId=ch.elca.el4j.plugins
        -DartifactId=maven-env-support-plugin
        -Dmojo=resources -Dfull=true
    ```
  - Instead off `groupId` & `artifactId` you can use parameter plugin with format `groupId:artifactId` and you can even use the plugin prefix. Examples:
    ```
    mvn -N help:describe -Dplugin=repohelper
        -Dmojo=deploy-libraries -Dfull=true
    mvn -N help:describe -Dplugin=jar
        -Dmojo=sign -Dfull=true
    ```


Daily usage (7) – Problem solving

- `mvn -N help:active-profiles (-P...|-D...)`
  - To test what profiles of the current artifact are currently active. In addition you can set profiles (-P) or system properties (-D) on the command line to see what profiles would be active in that case.
- `mvn -N help:effective-pom`
  - Prints the effective pom on console. You can define the parameter *output* to get the effective pom in a file. Example:
    ```
    mvn -N help:effective-pom -Doutput=effective-pom.xml
    ```
- `mvn -N help:effective-settings`
  - Prints the effective settings on the console. You can define the parameter *output* to get the effective settings in a file. Example:
    ```
    mvn -N help:effective-settings -Doutput=effective-settings.xml
    ```

➔ The settings file in the directory `~/.m2/` does override settings configured in directory `M2_HOME/conf/`

Daily usage (8) – Missing third party artifacts

Sometimes you create an artifact and this artifact must have a dependency to a third party jar like "spring-2.0.jar". With the repository helper from EL4J you have the possibility to easily install this new artifact in your local repository and directly in a remote repository. The jar file must have the following name:

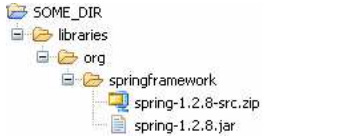
name-version.jar

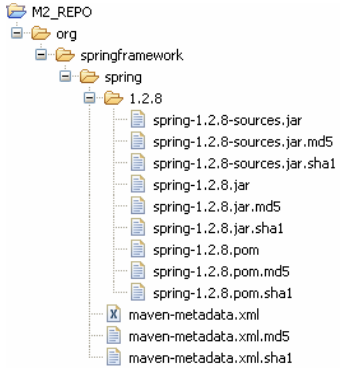
If you have a zip that contains the java source as well, this zip must have the following name:

name-version-src.zip

The *name* will be the artifactId. The groupId of the artifact will be determined by taking the delta between your given library path and the path of where these files are located. Slashes or backslashes in this delta are replaced by dots. No leading/trailing dots are permitted. The next slide shows an example of this.

Daily usage (9) – Missing third party artifacts





In the example on the left the following task has been executed in SOME_DIR.

```
mvn repohelper:install-libraries
  -DlibraryDirectory=libraries
```

The artifact org.springframework:spring:1.2.8 is now installed in the local repository and ready for local use.

To deploy libraries to a remote server just use the goal **`deploy-libraries`** instead of **`install-libraries`** and with additional parameter **`repositoryId`**. If the repository with this id is not defined in your **`pom.xml`** (see element **`distributionManagement`**) you must in additionally add the parameter **`repositoryUrl`** or **`repositoryDirectory`** that points to the remote repository. BTW, the username and password can be saved in the **`settings.xml`** file.

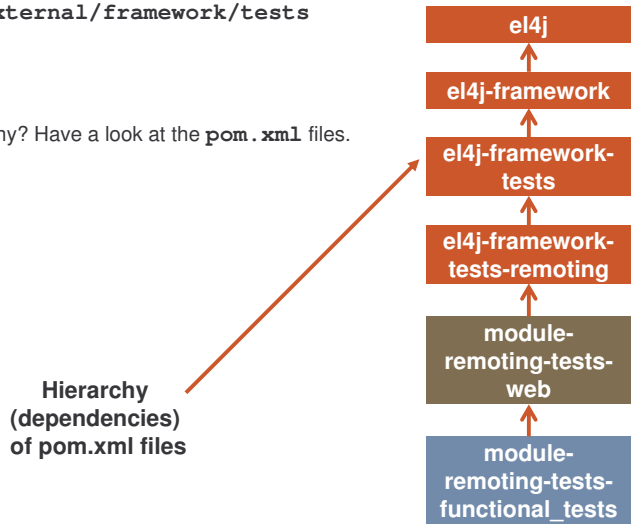
In **`EL4J_ROOT/external/helpers/upload`** there are two helper artifacts to install/deploy libraries in the external and internal repository. Example: Just put your libraries in **`EL4J_ROOT/external/helpers/upload/external/libraries`** and execute the specific goal without any parameters in **`EL4J_ROOT/external/helpers/upload/external`**.

Maven 2 – The powerful build system

1. Installation
2. Concepts
3. Where to find …
4. Daily usage
5. Advanced usage
6. (Plug-in development)

Advanced usage (1)

- **`cd EL4J_ROOT/external/framework/tests`**
- **`mvn -N`**
- **`cd remoting`**
- **`mvn install`**
  - What happens? Why? Have a look at the **`pom.xml`** files.



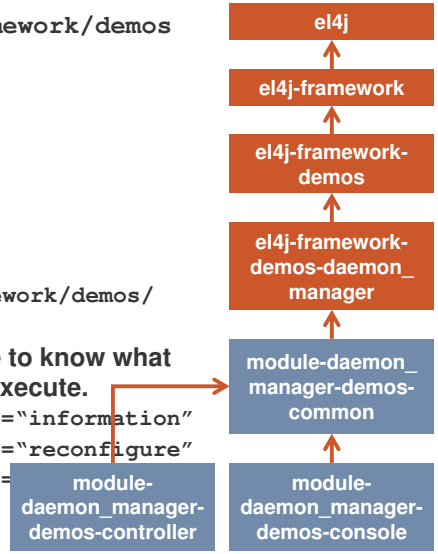
Advanced usage (1 – answers to questions)

It executes the functional tests:

- Create jars, wars, start tomcat, deploy the war, execute functional tests, undeploy war, stop tomcat
- In case tomcat does not yet exist, it is automatically downloaded (by default in the external-tools directory)

Advanced usage (2)

- `cd EL4J_ROOT/external/framework/demos`
- `mvn -N`
- `cd daemon_manager`
- `mvn install`
- `cd controller`
- `mvn exec:java`
  - **Which class will be executed?**
- **Open another command line**
  - `cd EL4J_ROOT/external/framework/demos/daemon_manager/console`
  - **Take a look in the pom.xml file to know what the following commands will execute.**
    - `mvn exec:java -Dexec.args="information"`
    - `mvn exec:java -Dexec.args="reconfigure"`
    - `mvn exec:java -Dexec.args=`



Advanced usage (2 - solution)

It executes the daemon manager (= the controller) on the console. You can then access the controller from remote (via the console (see the 3 actions from the previous slide))

Maven 2 – The powerful build system

1. Installation
2. Concepts
3. Where to find …
4. Daily usage
5. Advanced usage
6. (Plug-in development)

Plug-in development

- A plugin artefact is like a jar artifact.
- Packaging of its pom must be set to `maven-plugin`.
- Mojos can be annotated with Commons Attributes, so no plug-in descriptor must be written.
- A class needs to implement the interface `org.apache.maven.plugin.Mojo` to be a mojo.
- Plugins of EL4J are in the directory `EL4J_ROOT/external/framework/plugins`
  - `maven-checkclipse-helper-plugin`
  - `maven-env-support-plugin`
  - `maven-manifest-decorator-plugin`
  - `maven-repohelper-plugin`

Thank you for your attention

For further information please contact:

Martin Zeltner
Software Engineer
martin.zeltner <at> elca.ch

ELCA

Steinstrasse 21
CH-8036 Zürich
+41 (0)44 456 32 11

Lausanne I Zürich I Bern I Genf I London I Paris I Ho Chi Minh City