# Software Engineering

Philippe Collet

Master 1 IFI International
2012-2013
http://deptinfo.unice.fr/twiki/bin/view/Minfo/SoftEng1213

# Objectives

- Mastering software engineering techniques, focusing on object and component based approaches
- Principles and tools for automatic building and execution
- Testing
- Introspection, reflexivity and dynamic loading
- Choices and limitations of inheritance, compositon and constraint genericity mechanisms
- Micro-architectures and design patterns
- First steps in software architecture

# Agenda

- Construction (ant tool)
- Introspection, reflexivity
- Object V&V, unit testing
- Genericity
- Inheritance
- Design Patterns
- Dynamic loading
- (dependency injection)

# Evaluation

- 1 mark with continuous assessment during labs (presence, work, participation) (25%)
- 1 mark with a team project (35%)
- 1 final test (40%)

# Automated building in Java : ANT

- From Richard Grin's course
- http://ant.apache.org/

# Introduction

- Very rich syntax and options
- In this lecture
  - Version 1.5 (similar to 1.8.x)
  - No complete syntax and options
  - Ant is always distributed with its manual
- Open source project (Apache foundation)
- A reference for automatic and **portable** building of Java applications
- Written itself in Java

# Principles

- Similar to the *make* command
  - A project
  - Some targets (compile, jar, javadoc,…)
  - The description of targets and dependencies between them are described in a file
  - *XML file, named by default build.xml*
- Extensible : one can add his/her own tasks

# build.xml : example

```
<project name="hello" default="compile">

  <target name="prepare">
    <mkdir dir="./classes" />
  </target>

  <target name="compile" depends="prepare">
    <javac srcdir="./src"
           destdir="./classes" />
  </target>
</project>
```

# Building script: structure

- An XML header (with some optional DTD spec)
- A **`project`** entry that contains
  - Optionally, some **`property`** entries
  - Optionally, some **`path`** or **`classpath`** entries
  - One or several entries **`target`**
  - Optionally, an entry **`description`**
    - Informal description of the project
    - ```
      <description>
      This project enables one . . .
      . . .
      </description>
      ```
      ```

# Entry `project`

- Each file contains one and only one entry project
- This entry can have 3 attributes:
  - **`Name`** : project name
  - **`default`** : default target (required)
  - **`basedir`** : the base directory when dealing with relative paths
    - Can be overriden by the property **`basedir`**
    - By default the directory where the building file is

# Targets

- A target (**`target`**)
  - Corresponds to an action described in the file
  - Can depend on other targets (attribute **`depends`**)
- Each type of target can have its own attribute
- Attributes common to all targets:
  - **`name`** : name of the target (mandatory)
  - **`description`** : if it appears, it allows for displaying a description of the target with the option **`-projecthelp`** when calling **`ant`**
  - **`depends`** : indicates other targets that the current target depends upon

# Target dependencies

- One target can have several dependent targets (e.g. **`depends A,B,C`**)
  - Targets are executed in the order of the **`depends`** declaration (from left ro right)
- When handling dependencies, tasks are executed only once:

```
<target name="A"/>
<target name="B" depends="A"/>
<target name="C" depends="A,B"/>
```

A will be executed once

# Behavior on error

- Usually, an error in a task stops the building of the corresponding target
  - A class does not compile, the target building the jar stops
- Some tasks do not cause a stop
  - The attribute « **`failonerror`** » set to **`true`** can be used to force stop
  - Example : the « **`java`** » task

# Tasks

- A task is an atomic execution unit to realize a target
- Possible attributes:
  - id gives an unique ident to the taks. This id can be used in the rest of the file to refer to the task
  - taskname gives another name to the task. This name will be used in the execution reports
  - Description describes the tasks with unformated text
- Optional tasks
  - Needs an additional library to be exeucted (jar file to be installed in the appropriate directory)

# Tasks (java)

- Ant provides XML tags for most common tasks in Java:
  - javac, java, rmic, javadoc, jar, unjar, war, unwar, ear

```
<javadoc
   packagenames=« com.bigmoney.pack.*"
  sourcepath="${src}"
   destdir="${doc}/api"
   use="true"/>
```

# The task **javac**

- Used compiler: property **build.compiler**
  - By default, the JDK that runs Ant
- Compile recursively all java file in the source directory
  - Use last modification dates to determine whether a class needs to be recompiled
- Very large number of attributes: **srcdir** (required), **classpath**, **debug**, **optimize**, **source**, **fork**…

```
<javac srcdir="${src}“ destdir="${build}"
  classpath="xyz.jar" debug="on" />
```

# The task **java**

- Launches execution of a java program
- Attributes :
  - **classname** or **jar** to indicate which class to execute
  - **classpath**, **fork**, **failonerror**, **output**, **append**…

```
<java jar="dist/test.jar" fork="true"
        failonerror="true" maxmemory="128m">
    <arg value="-h"/>
    <classpath>
      <pathelement location="dist/test.jar"/>
      <pathelement path="${java.class.path}"/>
    </classpath>
</java>
```

# Other tasks

- System:
  - mkdir, delete, copy, move, chmod, touch, get, zip, unzip, tar, untar, gzip, gunzip
- Properties:
  - property gives the value of a property

```
<property name="jaxp.jar"
          value="./lib/jaxp11/jaxp.jar"/>
```

  - available initialises a property if a resource can be reached (file, directory, resource of the JVM)

```
<available classname="fr.unice.Classe"
           property="Class.present"/>
```

# Other tasks

- Programming:
  - fail stops the complete process
  - ant executes another ant file (useful for subprojects)
  - antcall executes another targets in the same file
  - apply, exec executes shellscripts and external programs
  - echo displays a message on System.out
  - mail sends an email
  - sql excutes a SQL query using a JDBC source
  - ftp establishes a FTP client to transmit files
  - junit adds tasks associated to the Junit framework (optional)

# Running Ant

- « **`ant`** » runs Ant using
  - The **`build.xml`** file of the current directory
  - The default target
    - Another target can be given as command argument
- Options
  - **`-buildfile`** to use another file than **`build.xml`**
  - **`-D`*`propriété=valeur`*** to give the value of a property
  - **`-help`** display available options
  - **`-projecthelp`** displays a project description, with all targets that have a description

# Main datatypes

- **`property`** : to parameterize the building process
- **`filelist`** : a list of files (no pattern « * » allowed)
- **`dirset`** : same as **`fileset`** with directories
- **`fileset`** : a list of files with more power than **`filelist`**, e.g. with **`patternset`**
- **`patternset`** : uses patterns in include/exclude clauses, used in **`fileset`** or **`dirset`**
- **`filterset`** : to replace tokens by values
- **`path`**, **`classpath`** : to give ordered list of PATH or CLASSPATH

# Properties

- Each project can have a set of properties, which are used as variables in the tasks' attributes
- **`${prop}`** gives the value of the property ***`prop`***
- Properties can be either local or global (out of any target)
- The name of a property is of the form **`project.name`** ou **`build.dir`**, following the convention of Java properties
- It it case sensitive
- 3 ways to give value to a property:
  - Task **`property`**
  - Task **`available`**
  - At Ant launch, using option –D :
    **`ant -D`***`property`*`=`*`value`*…

# Task **`property`**

- Several ways to give value to a property/set of properties
- For a single property:
  - **`name`** and **`value`**
  - **`name`** and **`refid`**
  - **`name`** and **`location`**
- For several properties:
  - **`file`** (name of a file containing Java formatted properties)
  - **`resource`** (same as **`file`** but looking in the *classpath*)

# Examples

- ```
  <available classname="fr.unice.MyClass"
      property="Class.present"/>
  ```

- ```
  <property name="jaxp.jar"
      value="./lib/jaxp11/jaxp.jar"/>
  <available file="${jaxp.jar}"
      property="jaxp.jar.present"/>
  ```

- ```
  <available file="/usr/local/lib" type="dir"
      property="local.lib.present"/>
  ```

- ```
  <property file="build.properties"/>
  ```

# Base properties

- One can use all Java system properties given by **`System.getProperties()`** as well as internal properties in **`ant`** :
  - **`basedir`** : the absolute path to the project root (set by the attribute « **`basedir`** » in « **`project`** »)
  - **`ant.file`** : absolute path of the building file
  - **`ant.version`** : Ant version
  - **`ant.java.version`** : JVM version

# `path` and `classpath`

- Special properties **`path`** and **`classpath`** are retained for filenames or lists of filenames
- They share the same syntax
- They can be
  - Included in a target definition
  - Or at the same level as global properties
    - A id is setup for each one that can then be used in several targets

# Classpath (or path)

- Refers to a Java *classpath* :

```
<classpath>
   <pathelement path="${classpath}"/>
   <pathelement location="lib/helper.jar"/>
</classpath>
```

May have several entries

Can only contain one entry

- Elements are defined by **`pathelement`** or **`fileset`**

# How to do better? Maven

- Handles tasks such as :
  - Reporting
  - Dependencies (automatic retrieve from repositories)
  - Configurations
  - Releases
  - Distribution process
- http://maven.apache.org/

# Maven : principles

- Creation of a project

```
mvn archetype:create -DgroupId=com.mycompany.app -DartifactId=my-app
```

- Default structure:

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

# Maven : pom.xml

- The file central to all configurations
  - Contains the main information of the project
  - Becomes very long and complex (generated and modified by GUI)

```
<project xmlns="http://maven.apache.org/POM/4.0.0" ... >
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 ...">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

# Maven phases

```
mvn archetype:create -DgroupId=com.mycompany.app -DartifactId=my-app
```

- archetype:create
  - archetype : plugin name => organisation with plugins and their dependencies
  - create : goal, similar to ant tasks

```
$ mvn package
...
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Thu Oct 05 21:16:04 CDT 2006
[INFO] Final Memory: 3M/6M
[INFO] ------------------------------------------------------------------------
```

- package
  - Is a phase : a step in the building cycle
  - The building cycle is a ordored list of phases
  - Execution of a phase exectutes all preceeding phases in order

# Maven phases

```
mvn compile
```

- Execute the following phases
  1. validate
  2. generate-sources
  3. process-sources
  4. generate-resources
  5. process-resources
  6. compile
- Default phases :
  - **validate**: validation of the  projet and of all necessary information (dependencies)
  - **compile**: compilation of the soure code
  - **test**: test of the compiled source using a tsting framework (declared). These tests should not require that the code is packaged or deployed
  - **package**: creation of a distributable package (e.g. Jar) from the compiled code
  - **integration-test**: deployment of the package in a environment where the integration tests are going to be passed
  - **verify**: checking on the package validity and some quality criteria
  - **install**: installation of the package int the local mvn repository,, so to be used as dependencies by other local projects
  - **deploy**: copy of the package on a remote site, for distribution
- Other useful phases:
  - **clean**: complete cleaning!
  - **site**: generation of the web site of the project

# TODO

- Preparing lab 1:
  - Install :
    - JDK 6 update 35
    - Eclipse IDE for Java developers : 4.2 Juno
    - Ant 1.8
  - Some tutorials. Stay tuned on
    - http://deptinfo.unice.fr/twiki/bin/view/Minfo/SeLab1