Ingénierie du logiciel : cours 4

Frédéric Gava

Master ISIN, Université de Paris-Est Créteil

Cours Ingénierie du logiciel du M2 ISIN

Plan

Plan

Plan

Déroulement du cours

Introduction

Pour quoi faire ?

- utilisé par le programme make pour arriver à une cible
- une cible étant elle-même dépendante d'autres cibles
- P. ex., créer de gros programmes C++ :
  - Rassembler les différents objets (.o) qui le composent
  - Ces objets peuvent dépendre de fichiers sources
- Génération de documents Latex
- Traîter des fichiers *etc.*

Syntaxe et utilisation

Un fichier "Makefile" puis commande "make cible". Syntaxe :

```
CIBLE: DEPENDANCES
  <TAB> COMMANDE1
  <TAB> COMMANDE2
  <TAB> COMMANDE2
```

Les commandes exécuter après les dépendances !

Fonctionnement

En vrac

- Calcul des dépendances : Une cible est à jour quand make regarde la date de modification du fichier portant le nom de la dépendance ou de la cible.
- Sur plusieurs lignes avec “\” et “@” commande invisible
- Règle “all” en début de fichier et
- Règle “clean” sans dépendance pour effacer

Exemple

```
all: edit
edit : main.o command.o
    cc -o edit main.o command.o
main.o: main.c
    cc -c main.c
command.o: command.c
    cc -c command.c
clean:
    rm -f edit main.o command.o
```

Variables (1)

Utilisation

CC=gcc
CFLAGS=−02 −Wall
prog: ...
    $(**CC**) −o ... $(**CFLAGS**)
VPATH pour des dépendances d'un dir : VPATH=src:../headers

Génériques

Symbole	Signification
$?	dépendances modifiées (pas à jours)
$^	toutes les dépendances
$+	idem sans doublons
$@	la cible courante
$<	première dépendance

Variables (2)

Ensemble de fichiers

% pour traîter tout les fichiers C :

```
%.o: %.c
    gcc -c $^
#ou
%.o: %.c
    gcc -c $*.c
```

Règles génériques

D'abord donner les suffixes qui sont traités par des règles. Puis avoir des règles pour ces suffixes. Exemple :

```
.SUFFIXES: .cpp .c
.cpp.o:
  $(CC) -c $(CFLAGS) $<
.c.o:
  $(CC) -c $(CFLAGS) $<
```

On trouve les commandes pour transformer le fichier d'extension d'origine en fichier d'extension cible.

Fonctions (1)

Cas particuliers : les fonctions

clean:

rm ∗.o

est correct mais dans les cibles ∗.o≡∗.o. Pour cela on utilise une fonction : $(wildcard ∗.o). Les fonctions commencent par $(et se terminent par). La fonction wildcard va faire ce que l'on cherche : elle va regarder les fichiers qui correspondent au motif (ici *.o) que l'on passe en paramètre.

D'autres fonctions

- Filter une liste de nom. $(filter %.o,$(noms)) correspond à tous les noms contenus dans $(noms) répondant à %.o
- Substituer $(subst FROM,TO,TEXT) (ou sur une liste de mots)

Fonctions (2)

Exemples

comma:= ,
empty:=
space:= $(empty) $(empty)
foo:= a b c
bar:= $(**subst** $(space),$(comma),$(foo))
 # bar is now 'a,b,c'.

La fonction $(patsubst PATTERN,REPLACEMENT,TEXT) permet de faire une substitution dans une liste de mots (des fichiers par exemple) :

SRC:= $(**wildcard** *.c)
OBJ:= $(**patsubst** %.c,%.o,$OBJ)

Gestion des makefiles (1)

Lancer un make depuis un make

Utile si on a un projet réparti dans plusieurs dossiers avec un make par dossier. Faire make −C subdir ou (cd subdir; make) (attention aux parenthèse sinon cela fait 2 commandes séparées)

Inclure un fichier

Utile quand on veut modifier plusieurs Makefile. Exemple :

−−−−−−−−−−−−Makefile.global−−−−−−−−−−−−

```
.cpp.o:
  $(CC) -c $(CFLAGS) $<
```

−−−−−−−−−−−−−−Makefile−−−−−−−−−−−−−−

```
include Makefile.global
OBJ= main.o
all: $(OBJ)
    for i in $(SUBDIRS); do (cd $$i; $(MAKE) all); done
    $(MAKE) ciblePrincipale
clean:
  rm -f $(OBJ) core *~
ciblePrincipale:
  @echo 'Ce que l'on veut'
```

Gestion des makefiles (2)

Créer des dépendances

La commande makedepend permet de déterminer les dépendances d'un fichier C ou C++ et les ajouter au makefile. Exemple : makedepend −− $(CFLAGS) −− $(SRC). Généralement on crée une règle dans le makefile qui va créer les dépendances, avant de compiler. Exemple :

```
SUBDIR= core ihm annexes
CC=gcc
CFLAGS=-I/usr/local/include -Wall -pipe
LDFLAGS=-lMesaGL -L/usr/local/lib
SRC=main.c install.c lancement.c
OBJ=$(subst .c,.o,$(SRC))
depend:
    makedepend -- $(CFLAGS) -- $(SRC)
```

Remarque :configure et automake (autoconf)

Skelette d'un makefile (1)

```
CC=gcc
CFLAGS=-I/usr/local/include -Wall -pipe
LDFLAGS=-lMesaGL -L/usr/local/lib
RM=/bin/rm
MAKE=/usr/bin/make
MAKEDEPEND=/usr/X11R6/bin/makedepend
SRC= a.c \
     b.c \
     c.c
OBJ=$(subst .c,.o,$(SRC))

SUBDIR= paf pof

.SUFFIXES: .c

.c.o:
  $(CC) -c $(CFLAGS) $<

all:
  for i in $(SUBDIRS); do (cd $$i; $(MAKE) all); done
  $(MAKE) monProgramme
```

Skelette d'un makefile (2)

```
monProgramme: $(OBJ)
    $(CC) -o $@ $(LDFLAGS) $^

clean:
    $(RM) -f $(OBJ) core *~
    for i in $(SUBDIRS); do (cd $$i; $(MAKE) clean); done

depend:
    $(MAKEDEPEND) -- $(CFLAGS) -- $(SRC)
    for i in $(SUBDIRS); do (cd $$i; $(MAKE) depend); done
```

Déroulement du cours

Introduction (1)

Qu'est-ce ?

ANT (Another Neat Tool) est un projet de l'Apache Software Foundation pour Windows et Linux. Un outil pour l'élaboration de projets avec un intégration dand l'IDE Eclipse et l'installation d'une interface de développement Antelope.

Pourquoi utiliser ANT (1)

- Ant peut être défini comme un gestionnaire de construction de projets basés sur le langage Java ainsi que le standard XML (un gestionnaire de "build") :
  - Il permet d'automatiser les tâches fastidieuses du processus de construction de projets.
  - Totalement open source, il présente des similitudes fonctionnelles avec le fameux make du monde Unix/Linux tout en ajoutant la portabilité du langage Java.
- L'exécution des tâches est déclenchée par un simple appel à la commande ant, avec ou sans arguments

Introduction (2)

Pourquoi utiliser ANT (2)

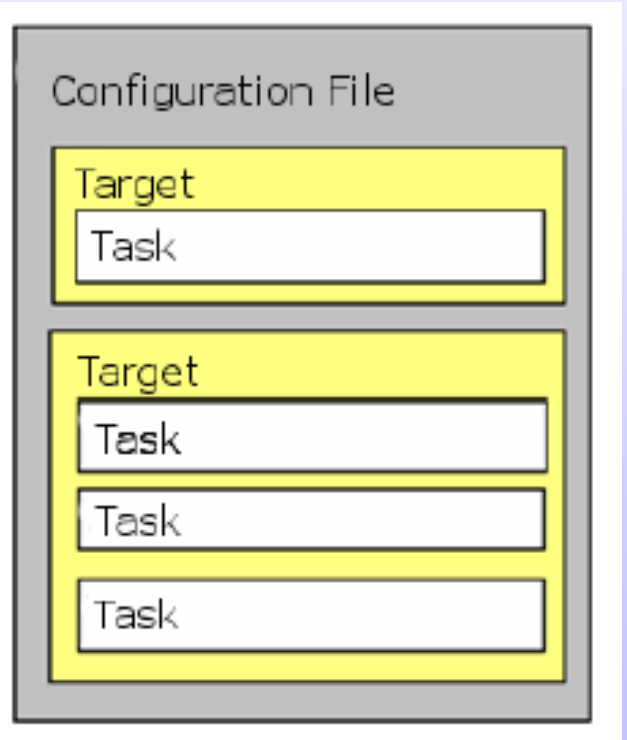
- L'automatisation de tâches définies directement par l'utilisateur dans un fichier de configuration évite un travail long et répétitif :
  - L'exécution des mêmes tâches scriptées garantit une fiabilité du processus de construction du projet ; Cela est d'autant plus appréciable lorsque l'on réalise un processus de test des fichiers.
  - L'automatisation des tâches oblige le ou les programmeurs à établir une hiérarchie et une arborescence de projet précise et à s'y tenir. Cela évite également les problèmes de version de fichiers.
- Génération des logs de construction du projet afin de bâtir une historique
- Moins intuitif/puissant que Maven mais plus léger

Automatisation des tâches (1)

- Ant a pour mission de compiler et de regrouper automatiquement divers composants d'un projet Java en tenant compte des dépendances entre ces derniers.
- Comme avec make, un projet Ant est constitué d'une collection de cibles correspondant aux différentes étapes de la construction du programme pour lesquelles il est possible de définir des dépendances afin de contrôler l'ordre dans lequel elles s'exécuteront
- Make est la solution idéale pour compiler des projets en C ou C++, mais inadapté à Java . Ant $\Rightarrow$ portabilité

Automatisation des tâches (2)

- Les différentes tâches ne sont toutefois pas indiquées sous la forme de commandes mais par le biais de balises XML prédéfinies
- C'est Ant qui choisira la commande à invoquer pour chacune d'entre elles.
- Bien évidemment, ce système est multi-plateforme.
- Exemples de tâches :
  - Compilation
  - Gestion des fichiers
  - Générer automatique de la documentation
  - Générer des paquetages



Configuration

- Linux : Récupérez l'archive pré compilée et, décompressez-la. Copier le dossier obtenu dans le répertoire choisi (par exemple /usr/local/ant) et régler les variables $ANT_HOME et $PATH (dans .bashrc) :

```
export ANT_HOME=/usr/local/ant
export PATH=$PATH:$ANT_HOME/bin
```

- Windobe : Il faut aller dans Démarrer -> Panneau de configuration -> Système -> Avancé -> Variables d'environnement ; Possibilité de redéfinir les variables système/ou utilisateur

Premier example

À la main

Compiler puis exécuter ou faire une archive (jar) :

```
mkdir build\classes                          #oata est la nom d'un package
javac -sourcepath src -d build\classes src\oata\HelloWorld.java
java -cp build\classes oata.HelloWorld
=======
echo Main-Class: oata.HelloWorld>myManifest
mkdir build\jar
jar cfm build\jar\HelloWorld.jar myManifest -C build\classes .
java -jar build\jar\HelloWorld.jar
```

Avec Ant

Fichier "build.xml". Puis executer "ant compile jar run"

```
<project>
    <target name="clean"> <delete dir="build"/>  </target>
    <target name="compile">
        <mkdir dir="build/classes"/>
        <javac srcdir="src" destdir="build/classes"/>
    </target>
    <target name="jar">
        <mkdir dir="build/jar"/>
        <jar destfile="build/jar/HelloWorld.jar" basedir="build/classes">
            <manifest> <attribute name="Main-Class" value="oata.HelloWorld"/> </manifest>
        </jar>
    </target>
    <target name="run"> <java jar="build/jar/HelloWorld.jar" fork="true"/> </target>
</project>
```

Quelques détails (1)

La balise “project”

- L'ensemble des éléments du fichier de configuration se voit inclut au sein d'une paire de balises project
- Les attributs name et basedir de celle-ci sont optionnels et contiennent respectivement le nom du projet et le répertoire à partir duquel seront définis tous les chemins relatifs
- L'attribut default est obligatoire et il a pour fonction de spécifier le nom de la cible qui sera exécutée par défaut lorsque Ant est appelé sans argument

La balise “project”

- Les commandes property servent à définir des variables qui peuvent ensuite se voir insérées avec `${variable}`.
- Si le projet se trouve dans le répertoire projet placé à la racine du compte toto, la deuxième clause property attribuera donc la valeur `/home/.../projet/src` à la variable `${src.dir}`

Quelques détails (2)

```
<target name="init" description="Initialize the project ">
 <tstamp/>
   <mkdir dir="${build.dir}" />
   <mkdir dir="${doc.dir}" />
   <mkdir dir="${dist.dir}/lib"/>
</target>
```

tstamp a pour effet d'assigner la date et l'heure actuelle aux variables DSTAMP, TSTAMP et TODAY, respectivement sous la forme "aaaammjj", "hhmm" et "mois jour année".


```
<target name="init" description="Initialize the project ">
 <tstamp/>
   <mkdir dir="${build.dir}" />
   <mkdir dir="${doc.dir}" />
   <mkdir dir="${dist.dir}/lib"/>
</target>
```


L'attribut "depends", a pour fonction de spécifier les étapes devant être effectuées avant d'exécuter les clauses de cette cible.

Propriétés

Quand on a régulièrement les mêmes noms de dossiers/fichiers :

```
<project name="HelloWorld" basedir="." default="main">
  <property name="src.dir"     value="src"/>
  <property name="build.dir"   value="build"/>
  <property name="classes.dir" value="${build.dir}/classes"/>
  <property name="jar.dir"     value="${build.dir}/jar"/>
  <property name="main-class"  value="oata.HelloWorld"/>

 <target name="clean">  <delete dir="${build.dir}"/>  </target>

 <target name="compile">
        <mkdir dir="${classes.dir}"/>
        <javac srcdir="${src.dir}" destdir="${classes.dir}"/>
 </target>

 <target name="jar" depends="compile">
    <mkdir dir="${jar.dir}"/>
    <jar destfile="${jar.dir}/${ant.project.name}.jar" basedir="${classes.dir}">
      <manifest>  <attribute name="Main-Class" value="${main-class}"/>  </manifest>
    </jar>
 </target>

 <target name="run" depends="jar">
   <java jar="${jar.dir}/${ant.project.name}.jar" fork="true"/>
  </target>

 <target name="clean-build" depends="clean,jar"/>
 <target name="main" depends="clean,run"/>
</project>
```

Tester une classe avec Junit (1)

Supposons le petit code suivant :

```
public class HelloWorldTest extends junit.framework.TestCase {
  public void testNothing() {}
  public void testWillAlwaysFail() {fail("An error message");}
```

Pour tester :

```
<path id="application" location="${jar.dir}/${ant.project.name}.jar"/>
  <target name="run" depends="jar">
        <java fork="true" classname="${main-class}">
            <classpath>
                <path refid="classpath"/>
                <path refid="application"/>
            </classpath>
        </java>
    </target>

  <target name="junit" depends="jar">
        <junit printsummary="yes">
            <classpath>
                <path refid="classpath"/>
                <path refid="application"/>
            </classpath>

            <batchtest fork="yes">
                <fileset dir="${src.dir}" includes="*Test.java"/>
            </batchtest>
        </junit>
    </target>
```

Tester une classe avec Junit (2)

Et produire un report :

```
...
   <property name="report.dir"  value="${build.dir}/junitreport"/>
   ...
   <target name="junit" depends="jar">
       <mkdir dir="${report.dir}"/>
       <junit printsummary="yes">
           <classpath>
               <path refid="classpath"/>
               <path refid="application"/>
           </classpath>

           <formatter type="xml"/>

           <batchtest fork="yes" todir="${report.dir}">
               <fileset dir="${src.dir}" includes="*Test.java"/>
           </batchtest>
       </junit>
   </target>

   <target name="junitreport">
       <junitreport todir="${report.dir}">
           <fileset dir="${report.dir}" includes="TEST-*.xml"/>
           <report todir="${report.dir}"/>
       </junitreport>
   </target>
```

Organiser un projet Ant (proposition)

- Le répertoire “src” renferme tous les fichiers source (extension .java)
- build ou classes pour accueillir des classes compilées (.class)
- dist contiendra la distribution (les fichiers jar ; l'archive jar de notre application prête à être déployée.
- lib pour contenir les archives utiles au projet telle que ant.jar
- le répertoire doc pour servir à stocker la documentation des classes (
- Certains projets peuvent également contenir
  - conf pour les fichiers de configuration,
  - res contenant les différentes ressources nécessaires au programme (images, traductions, sons, ...),
  - un dossier etc pour les fichiers qui n'appartiennent à aucune des catégories précitées.

les tâches Java(c)

Tout d'un coup

```
<javac srcdir="${src.dir}" destdir="${build.dir}"/>
```

Compiler tout ce qui est dans srcdir et de placer les classes compilées dans destdir. Cette opération est récursive. A l'instarde make, Ant compilera chaque classe uniquement s'il n'existe pas déjà un fichier compilé dans le répertoire de destination ou si la date d'enregistrement d'un fichier éventuellement déjà présent dans ce dernier est plus ancienne que le fichier source.

Démarrer et un petit fork

```
<java dir="${build.dir}" classname="com.example.magicalSquare.TestSquare" fork="true"/>
```

Pour démarrer notre application. 3 arguments vont être définis :

1. "classname" définit la classe de démarrage de l'application
2. "dir", permettant de désigner le répertoire racine de l'application. On y définit le répertoire build.
3. fork indique qu'une nouvelle machine virtuelle sera utilisée. Cette option est nécessaire afin de changer le répertoire d'exécution (attribut dir).

Quelques détails (3)

Le paramètre classpath

- Elle est imbriquée dans la tâche `<java>` afin de définir la variable classpath
- Celle-ci contient le chemin dans le système de fichier vers les librairies jar nécessaire à l'application
- Le contenu de l'attribut name nécessite une explication : `**/*.jar`. Cela signifie que l'on charge tous les fichiers terminant par .jar, dans tous les sous-répertoires de `${lib.dir}` :

```
<java ...> <classpath>
  <fileset dir="${lib.dir}"> <include name="**/*.jar"/> </fileset>
</classpath> </java>
```

- Le paramètre fileset permet de lire l'ensemble dans fichier jar dans le répertoire lib. Le paramètre include permet d'inclure dans le classpath les fichiers correspondant à l'expression régulière donnés par l'attribut name.

Quelques détails (4)

Écrire la cible documentation

```
<javadoc packagenames="com.example.magicalSquare" sourcepath="${src.dir}" destdir="${doc.dir}"/>
```

La tâche javadoc de la cible documentation appelle le programme homonyme qui permet de générer automatiquement des pages d'aide en HTML à partir du code source de vos classes. Contrairement à javac, cette tâche génère systématiquement les fichiers de documentation sans vérifier si ces derniers existent déjà ou sont à jour. Il est donc conseillé de ne l'appeler que de manière ponctuelle. Nous fournissons ici 3 attributs :

1. destdir indique le dossier où seront stockées les pages HTML,
2. sourcepath le répertoire racine de vos sources
3. packagenames : liste des paquetages à documentés.

Il est possible de compiler uniquement certains fichiers en omettant ces 2 derniers paramètres et en précisant leur chemin à l'aide de l'attribut sourcefiles.

Quelques détails (5)

Écrire la cible dist

```
<target name="dist" depends="compile" description="Generate a jar file">
  <jar jarfile="${dist.dir}/magical_Square-${DSTAMP}.jar" basedir="${build.dir}"/>
</target>
```

- Elle sert à créer une archive jar qui permettra de distribuer facilement les classes compilées.
- Remarque : l'attribut jarfile, la variable `${DSTAMP}` (créé par l'appel de la tâche `<tstamp/>`) afin d'inclure la date de compilation dans le nom d'archive. Un nouveau fichier apparaîtra ainsi chaque jour, permettant de disposer automatiquement d'un historique de notre projet.
- Cette pratique est extrêmement courante dans le contexte de projets OpenSource, où l'on procède quotidiennement à des "nightly builds", des compilations systématiques du code source d'un projet.

Quelques détails (6)

Écrire la cible clean

- La cible clean est similaire à celle utilisée dans make et a pour fonction de nettoyer notre projet de tous les répertoires créés par Ant.

```
<target name="clean">
  <delete dir="${build.dir}"/>
  <delete dir="${doc.dir}"/>
  <delete dir="${dist.dir}"/>
</target>
```

- la tâche delete et son attribut dir efface récursivement un répertoire. Pour effacer une liste de fichiers : à l'attribut file.

Ant Graphique et Eclipse

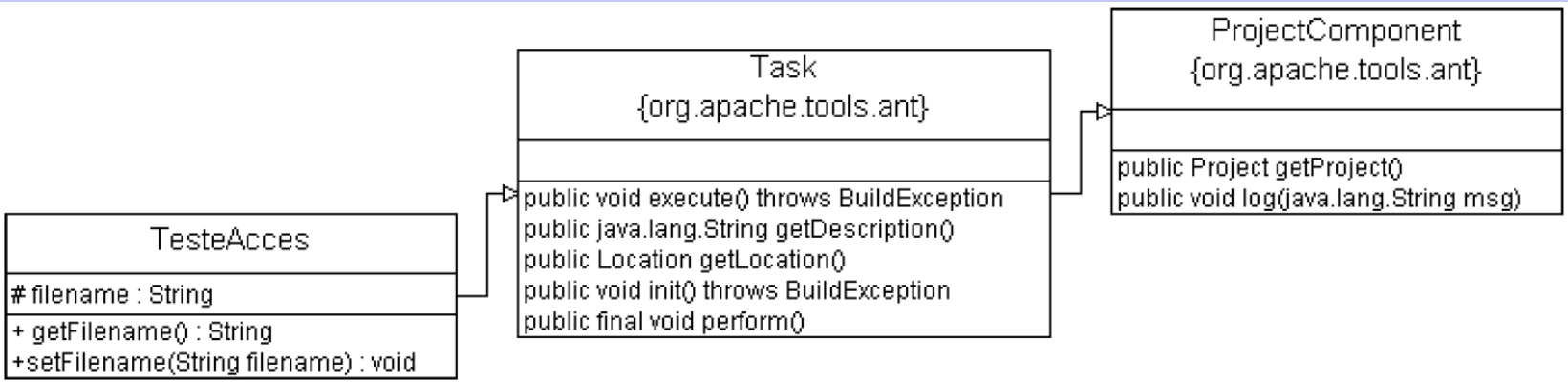
- Il existe un plugin. L'éditeur inclut les fonctionnalités suivantes : Syntaxe colorée ; Assistance d'écrire du code ; vision particulière des lib, packages et classes.
- Antelope : Interfaces de ANT ou plugin pour JEdit
- On trouve d'autres trucs : AntView plugin, *etc.*

Création d'une tâche personnalisée (1)

Edition

Une tâche qui testera si un fichier est éditable :

- Cela se justifie par le fait que des commandes comme replace ne s'exécutent que si un fichier est autorisé en écriture.
- Développer une tâche, qui définisse un attribut myFile, associé à canWrite lorsque le fichier spécifié avec l'attribut filename est autorisé en écriture :



Création d'une tâche personnalisée (2)

```
/** Cette tâche teste si le fichier est accessible en écriture */
import java.io.File ;
import org.apache.tools.ant.BuildException ;
import org.apache.tools.ant.Task ;
public class TesteAcces extends Task {
 private String filename ; // le fichier qui sera utilisé
 // setter pour l'attribut filename, il est appelé automatiquement par ant
 public void setFilename(String filename) {this.filename = filename;}
 /** Réécriture de la méthode execute()
 * @exception BuildException, si un problème se produit, utile au debug. */
 public void execute() throws BuildException {
  // accès au fichier
  File myFile = new File(filename);
  // contrôle de l'écriture
  if(myFile.canWrite()){project.setUserProperty("myFile", "canWrite");}
```

Dans le code Java, il faut prendre en considération ceci :

- La nouvelle tâche doit dériver de la classe abstraite org.apache.tools.ant.Task.
- Il faut implémenter la méthode execute(), appelée à l'exécution de la tâche. La méthode execute() doit pouvoir lever une exception BuildException.
- Pour chaque attribut (uniquement filename ici), un accesseur doit être écrit.

Création d'une tâche personnalisée (3)

Pour utiliser une tâche, il faut qu'elle soit compilée, déclarée dans taskdef et qu'elle se trouve dans le CLASSPATH :

```
<taskdef name="accesOk" classname="TestAccess" classpath="${build.dir}" />
 <target name="main" depends="TestAccess" if="myFile">
    <replace File="labo.txt" Token="test" Value="java" summary="true" />
 </target>
 <target name="TestAccess">
    <accesOk filename="${test.file.path}" />
    <echo message="The file ${test.file.path} is writeable = ${myFile}" />
 </target>
```

- On pourra réaliser les initialisations avec la méthode init().
- Deux champs publics : project et location permettent de modifier l'environnement dans lequel la tâche s'exécute
- Une dépendance de la cible TestAcces est reconnue et exécutée. Si la permission en écriture est à vrai, la propriété myFile est définie et envoyé à la sortie avec un echo. Sans droit d'écriture, la propriété n'est pas définie et l'attribut if de la cible main veille à ce que les tâches de la cible main ne soient pas exécutées. Mais si l'attribut est défini et qu'il existe des permissions en écriture, un remplacement de texte, par exemple dans la cible main, peut être effectué sur le fichier.

Annexe : project

Structure de build.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default="compile"
name=" Typical ANT Build ">
<!-- [définitions de property] -->
<!--[définitions de path et patternset ] -->
<!--[définitions de target ] -->
</project>
```

Annexe : property

```
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default="compile" name=" Typical
ANT Build ">
<!-- [définitions de property ] -->
<property name="src.dir" value="./src"/>
<property name="build.dir" value="./build"/>
<property name="doc.dir" value="./doc"/>
<property name="apidoc.dir" value="${doc.dir}/api"/>
<property file="project.properties"/>
<property environment="env"/>
<property name="lib.dir" value="${env.LIBDIR}"/>
<!-- [définitions de path et patternset ] -->
<!-- [définitions de target ] -->
</project>
```

En plus des propriétés définies par l'utilisateur, vous disposez de toutes les propriétés système (voir la description de la méthode getProperties() de la classe java.lang.System dans la documentation Java API) et de quelques propriétés spécifiques à Ant (basedir, ant.file, ant.version, project.name, et ant.java.version). Les variables DSTAMP et TSTAMP sont également intéressantes afin d'insérer la date dans le nom du fichier jar créé, par exemple.

Annexe : path

```
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default="compile" name=" Typical
ANT Build ">
<!-- [définitions de property ] -->
<!-- [définitions de path et patternset ] -->
<path id="project.classpath">
<pathelement path="${build.dir}"/>
<fileset dir="${lib.dir}">
<include name="**/*.jar"/>
</fileset>
</path>
<!-- [définitions de target ] -->
</project>
```

Les jeux de fichiers (marqueurs ¡fileset¿) sont utilisés pour spécifier des ensembles de fichiers. Ces marqueurs sont normalement des marqueurs internes. Ils peuvent être définis, le cas échéant, à l'aide des éléments include, exclude, etc., par l'intermédiaire d'un jeu de modèles ou de références. Certaines tâches prédéfinies offrent ces marqueurs directement sous forme d'attribut (mais avec un "s" à la fin, comme ceci : includes). Cette balise s'applique à : apply, chmod, copy, delete, dependset, ear, exec, filter, javac, javadoc, move, touch, uptodate, war, zip, etc. Dans l'exemple suivant, la tâche copy est utilisée pour copier récursivement, à partir du répertoire défini par "src" tous les fichiers source Java, dans le nom desquels ne figure pas "42", et cela vers un répertoire "dest". L'expression **/ englobe tous les sous-répertoires.

```
<copy todir="dest">
<fileset dir="src" includes="**/*.java" excludes="**/*42*"/>
</copy>
```

Annexe : target

```
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default="compile" name=" Typical
ANT Build ">
<!-- [définitions de property ] -->
<!-- [définitions de path et patternset ] -->
<!-- [définitions de target ] -->
<target name="init">
<mkdir dir="${build.dir}"/>
</target>
<target name="compile" depends="init">
<javac srcdir="${src.dir}"
destdir="${build.dir}"
classpathref="project.classpath"/>
</target>
</project>
```

Les “cibles” regroupent les phases de travail individualisées. L’attribut depends permet de définir des dépendances, c’est-à-dire, l’ordre dans lequel les cibles seront exécutées. Les cibles peuvent dépendre de plusieurs autres cibles, séparées par des virgules dans la liste associée à l’attribut (depends = “B, C, D”).

Annexe : lancement

```
ant [options] [target [target2 [target3] ...]]
Options:
-help print this message
-projecthelp print project help information
-buildfile <file> use given build file (-file, -f)
-find [<file>] search for build.xml, or file, towards the root of the filesystem
-D<property>=<value> use value for given property
-propertyfile <file> load all properties from file (with -D taking precedence)
-version print the version information and exit
-quiet be extra quiet (-q)
-verbose be extra verbose
-debug print debugging information
-emacs produce logging information without adornments
-logfile <file> write logging output to given file (-l)
-logger <classname> the class that is to perform logging
-listener <classname> add an instance of classname as a project listener
-inputhandler <class> the class that will handle input requests
```

Annexe : initialisation avec vérification des propriétés

```
<target name="init">
<available property="server.ok"
classname="com.mycomp.server.HTTPDServer"
classpath="${server.classpath}"/>
<available property="setup.done"
file="${server.conf.dir}/conf/server.xml"/>
</target>
<target name="check server" unless="server.ok">
<fail message="${line.separator}Configure the server
classpath."/>
</target>
<target name="check setup" unless="setup.ok">
<fail message="${line.separator}Setup your server.xml
configuration file."/>
</target>
<target name="run" depends="init, check server, check setup">
...
</target>
```

Annexe : structure de contrôle

```
<target name="confirm.deletion">
<input message="All data is going to be deleted (y/n)?"
validargs="y,n" addproperty="do.delete" />
<condition property="do.abort">
<equals arg1="n" arg2="${do.delete}" />
</condition>
<fail if="do.abort">Build aborted by user.</fail>
</target>
```

Annexe : FTP

```
<project name="Mon Site Web Perso" default="transfert" basedir=".">
<property name="src.dir" value="."/>
<property name="remote.dir" value="/public_html"/>
<target name="init">
<tstamp><format property="TODAY" pattern="yyMMdd"/></tstamp>
<echo file="${src.dir}/LAST_SITE_LOADING">${TODAY}</echo>
<input message="Please enter username:" addproperty="userid"/>
<input message="Please enter password:" addproperty="password"/>
</target>
<target name="transfert" depends="init,help">
<ftp server="ftp.monhebergeur.fr" remotedir="${remote.dir}"
userid="${userid}" password="${password}"
depends="yes" binary="yes">
<fileset dir="${src.dir}"/>
</ftp>
...
<mail from="me" tolist="you" subject="Results of transfer" files="build.log" />
</target>
```

Déroulement du cours

Introduction

Pourquoi Maven ? (http ://maven.apache.org)

Maven est un outil “open-source” de Apache Jakarta. Il permet de faciliter et d'automatiser la gestion et la construction d'un projet Java. Il permet notamment :

- d'automatiser la compilation, les tests unitaires et le déploiement des applications du projet (jar)
- de gérer les dépendances des bibliothèques du projet
- de générer des documentations du projet : rapport de compilation et des tests unitaires, javadoc, *etc.* un site web complet du projet
- d'utiliser une large bibliothèques de goals prédéfinis (les plugins)
- de faciliter la création d'extensions grâce au langage Jelly (à base de XML)

ANT vs Maven ; Caractéristiques des deux logiciels

	ANT	Maven
Installation	Très simple	Très simple (similaire à Ant)
Temps de démarrage d'un projet	5 minutes	15 minutes
Temps pour ajouter une fonctionnalité	10 minutes pour ajouter une cible	2 minutes pour utiliser un nouveau goal
Temps d'apprentissage	30 minutes. Très simple à comprendre, et très bien documenté	2 heure, car il peut être difficile à appréhender
Standardisation des projets	Non (ce qui est bien, on peut faire ce que l'on veut)	Oui (ce qui est bien, car tous les projets se ressemblent)
Génération de documentation	Aucun standard, mais il y a plusieurs outils disponibles	oui
Intégration dans les environnements de développement	moyen+	moyen-

Composition du fichier “project.xml”

En général, 3 principales parties composent le fichier project.xml :

1. La partie “gestion du projet” inclut les informations telles que l’organisation du projet, la liste des développeurs, la localisation du code source, et l’URL du système pour déceler les bugs.
2. La partie “dépendance” du projet inclut les informations concernant les dépendance du projet.
3. La partie de “build et de documentation” (rapports) contient les informations du build telles que le répertoire du code source, des tests unitaires, et les rapports à générer définis dans le build.

Exemple de fichier project.xml (1)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- l'élément racine d'un fichier projet -->
 <project>
  <!-- Partie gestion de projet -->
  <!-- groupe id du projet. -->
  <!-- Si présent, l'id est utilisé comme nom de dossier-->
  <!-- dans le repository -->
  <!-- pour regrouper tous les jars du projet -->
  <groupId>labosun.maven_session</groupId>

  <!-- Unique id du projet. -->
  <!-- Il est aussi utilisé pour générer des noms de fichier -->
  <!-- i.e, le fichier JAR est nommé : <id>-<version> -->
  <id>maven_session</id>

  <!-- Le nom court du projet -->
  <name>maven_session</name>

   <!-- numero de version du projet. (Pas de norme à suivre) -->
   <currentVersion>1.2</currentVersion>
```

Exemple de fichier project.xml (2)

```
<!-- Partie dépendances du projet -->
<dependencies>
<dependency>
 <groupId>log4j</groupId>
 <artifactId>log4j</artifactId>
 <version>1.2.8</version>
</dependency>
</dependencies>

<!-- Partie build et documentation du projet -->
<build>
 <sourceDirectory>src/main/java</sourceDirectory>
 <unitTestSourceDirectory>src/test/java</unitTestSourceDirectory>
 <resources>
   <resource>
     <directory>src/main/resources</directory>
   </resource>
 </resources>
 <unitTest>
 <includes>
   <include>**/*Test.java</include>
  </includes>
 </unitTest>
</build>
</project>
```

Les fonctionnalités de Maven

Les tâches exécutées par Maven reposent sur des plugins qui sont des fichiers jar :

- Chaque plugin permet d'effectuer des tâches particulières prédéfinies appelées "goals".
- Les goals sont configurables grâce au fichier : "maven.xml"
- Maven en ligne de commande : `Maven plugin:goal` ; Exemple : `Maven java:compile`
- Si le goal n'est pas spécifié le goal par défaut sera exécuté
- Il faut exécuter Maven dans le répertoire qui contient le fichier "project.xml".
- Si les paramètres fournis ne sont pas corrects, une exception est levée.
- Une liste complète des plugins à disposition de Maven : `maven -g`

Générer un projet et types

Pour débuter, un plugin permet de générer la structure du projet :

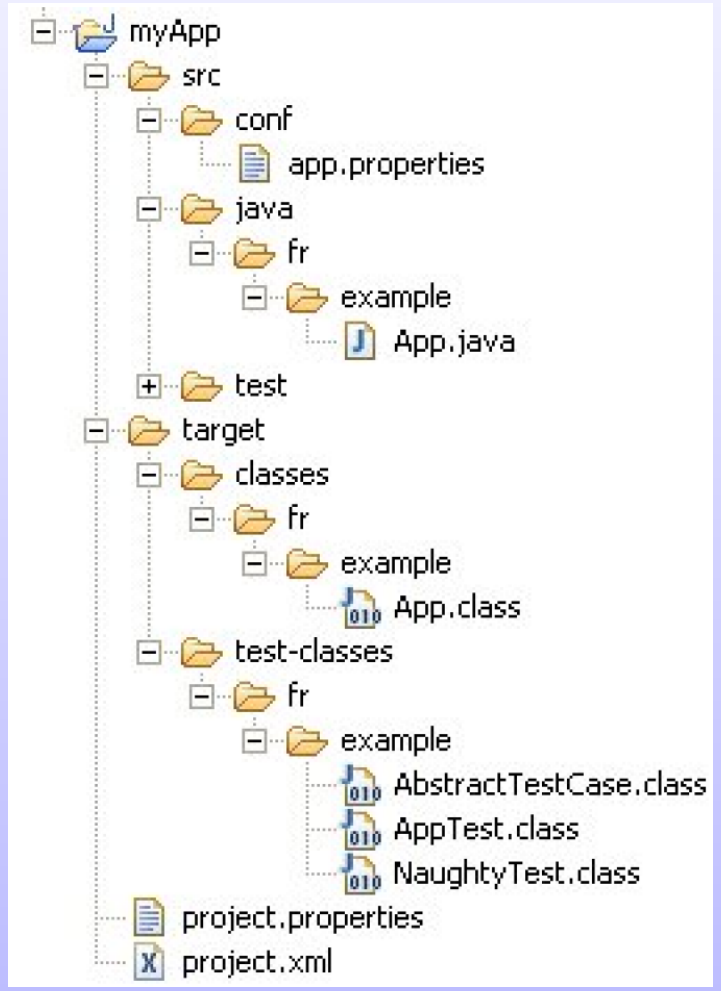
```
$ maven genapp
| \/ |__ _Apache__ ___
| |\/| / _‘ \ V / -_) ’ \ ~ intelligent projects ~
|_| |_\__,_|\_/\___|_||_| v. 1.0
Enter a project template to use: [default]
defaut

Please specify an id for your application: [app]
myApp

Please specify a name for your application: [Example Application]
My Appplication Test

Please specify the package for your application: [example.app]
fr.example
```

Il y a plusieurs types de template prédéfinie :

- default : Génère un simple projet Jar.
- Ejb : Génère un simple EJB.
- Struts : Génère une simple application web Struts ; Struts-jstl : Génère une simple application web Struts et JSTL.
- Web : Génère une simple application web Struts ; Web-jstl : Génère une simple application web Struts avec JSTL
- Complex : Génère un projet très complexe avec ear, wars, ejbs, struts, xdoclet.

Générer un projet et types

- Ce plugins génère la structure de l'application selon les paramètres saisis
- Il génère aussi le squelette des fichiers `project.properties` et `project.xml`.
- Il permet de démarrer rapidement sur le développement d'une application.



Générer un site Web

Maven propose une fonctionnalité qui permet de générer automatiquement un site web pour le projet regroupant un certain nombre d'informations utiles le concernant :

- maven site :generate
- un répertoire target/docs est créé contenant les différents éléments du site :
  - La partie Project Info : 3 pages
    1. la mailing list,
    2. la liste des développeurs
    3. les dépendances du projet.
  - La partie Project report : comptes rendus d'exécution de certaines tâches : javadoc, tests unitaires,
- Le contenu du site pourra donc être réactualisé facilement en fonction des différents traitements réalisés par Maven sur le projet.

Plugins et commandes intéressants

Quelques plugins utiles et intéressants

- Java et clean : permettent la compilation et le nettoyage des fichiers générés par la compilation.
- Jar : Créer un fichier jar dans le repertoire build du projet
- Jalopy : utilitaire open source qui permet de formater du code source Java et même de vérifier l'application de normes de codage.
- Site : permet de générer un site très complet du projet

Donc, les commandes

- `java:prepare-filesystem` : créer les répertoires nécessaires à la compilation
- `java:compile` : compile les fichiers codes sources. Le répertoire source est défini entres les tags `<build>` dans project.xml.
- `clean:clean` : Supprime `${maven.build.dir}` et `${basedir}/velocity.log`
- `jar:jar` : créer un fichier jar dans le repertoire build du projet sous la forme `${project.id}-${project.currentVersion}.jar` où id and currentVersion proviennent de project.xml

Plugins et commandes intéressants (2)

Jalopy

Jalopy permet notamment :

- d'indenter le code
- de générer des modèles de commentaires javadoc dynamique en fonction des l'éléments du code à documenter (par exemple générer un tag @param pour chaque paramètre d'une méthode)
- d'organiser l'ordre des clauses import
- d'organiser l'ordre des membres d'une classe selon leur modificateur de vérifier l'application de normes de codage

Pour finir

Le repository

Afin de bien gérer les dépendances, Maven utilise des repositories pour télécharger automatiquement les composants dont il a besoin. Mais pour éviter que les fichiers se téléchargent à chaque reconstruction, Maven stocke automatiquement les dépendances nécessaires dans le repository local, situé dans `$HOME/.maven/repository`

Bilan

- Les grands avantages de Maven sont :
  - Les nombreux plugins qui répondent à nos besoins
  - La diffusion de l'état actuel du projet via la génération de site
- Maven est un outil récent mais il est déjà utilisé par de nombreux projet Jarkarta. Il a donc déjà fait ses preuves et on peut avoir confiance sur la maturité de cet outil.
- Maven est plus simple et plus facile à prendre en main que Ant.
- De plus mevenide (le plugin Eclipse) permet de l'intégrer dans la plupart des outils de développement.

Quelques références

- ANT (avec pléthore de liens externes) : `http://jakarta.apache.org/ant`
- XDOCLET : `http://xdoclet.sourceforge.net`
- Maven : `http://maven.apache.org`
- Antidote (interface graphique pour Ant) : `http://jakarta.apache.org/ant/manual/Integration/Antidote.html`
- AntDoc (Javadoc pour Ant) : `http://mapage.noos.fr/antdoc/example_top.html`
- Nant (Ant version .NET) : `http://nant.sourceforge.net/`
- AntContrib (tâches C++ pour ANT) : `http://sourceforge.net/projects/ant-contrib/`
- CruiseControl (outil d'intégration continue basé sur ANT) : `http://cruisecontrol.sourceforge.net/` (Cette technique permet par exemple de monitorer le code source d'une application pour automatiser la construction d'un projet en cas de changement. Il envoie ainsi un mail avec la liste des modifications effectuées depuis la dernière construction ou génère des rapports en HTML.

A la semaine prochaine