TP 4 : Outils pour le développement logiciel

Ce TP présente plusieurs outils couramment utilisés pour développer des applications Java. Il existe bien sûr des équivalents pour d'autres langages de programmation.

1 Organisations des fichiers sources Java

On a vu lors des précédents TPs que Java disposait d'une API de base assez riche, et qu'elle était organisée en *package*. On a notamment utilisé les package `java.io` pour les entrées-sorties et `java.util` pour les structures de données. Pour utiliser les classes regroupées dans ces package, on peut utiliser la directive `import` ou bien spécifier explicitement le nom du package lors de la déclaration d'un objet.

```
import java.io.*; // On importe tout le package
import java.util.Random; // On importe une classe précise

public class MyClass {
  public static void myMethod (){
    java.math.BigInteger big; // On précise le paquet qui contient la classe
  }
}
```

Pour faire appel à une classe d'un package, il faut forcément utiliser une de ces méthodes, sauf pour le package `java.lang` qui est directement importé dans le langage.

Il est possible d'utiliser le mécanisme de package pour son propre code. L'organisation par paquet permet naturellement de regrouper les classes par groupes de tâches, et d'assurer une certaine sécurité de manipulation aux développeurs en jouant sur la visibilité des classes, attributs et méthodes. Elle est aussi très utile pour la modularisation et la réutilisation du code. On peut ainsi diffuser ses fichiers .class et fournir les interfaces d'utilisation à d'autres développeurs, qui n'ont plus qu'à inclure les packages ou classes qui les intéressent.

Pour signifier qu'une classe appartient à un package précis, il suffit d'utiliser le mot-clef `package` suivi du nom de paquet voulu au début du fichier source, avant toute autre déclaration ou `import`. Par exemple, on peut avoir un fichier MyClass.java

```
package mypackage;

public class MyClass{
  public void myMethod() {}
}
```

La classe, une fois compilée est donc dans le package *mypackage*, lequel se manipule comme les autres. Ainsi, dans un fichier Main.java, on peut avoir le code suivant :

```
import mypackage.MyClass;

public class Main{
  public static void main(String[] args){
    MyClass c = new MyClass();
    c.myMethod();
  }
}
```

Naturellement, on souhaite organiser les fichiers d'un même paquet dans un même répertoire. Le compilateur et la machine virtuelle Java permettent ce comportemnet. On peut en effet regrouper l'ensemble des fichiers sources d'un même paquet dans un répertoire du même nom. En compilant depuis la racine, le compilateur java trouvera seul les autres paquets et saura gérer les dépendances. Ainsi, si on a deux répertoires *firstpackage* et *secondpackage* dans le répertoire courant. Dans le premier, on a des fichiers sources qui appartiennent tous au même paquet :

```
package firstpackage;
```

Maintenant, imaginons que nous utilisons des classes du premier package dans le second, on aurait alors des fichiers avec comme en-tête :

```
package secondpackage;
import fistpackage.*;
```

On peut alors compiler un tel fichier dans le second paquet de la manière suivante :

```
javac secondpackage/MyClass.java
```

Il saura trouver les fichiers .class du firstpackage dans le répertoire du même nom, où compilera les fichiers sources si les fichiers .class n'existent pas. Les fichiers .class de chaque paquet seront mis dans les répertoires correspondant. On peut alors exécuter la méthode `main` d'une classe depuis le repértoire courant de cette manière :

```
java secondpackage.MyClass
```

Il s'agit bien d'un point et non du séparateur de fichier classique. D'autre part, on peut vouloir stocker les fichiers sources et les fichiers objets (.class) dans des endroits différents. Il suffit alors d'avoir la même arborescence de répertoires correspondants aux package dans les deux endroits, et d'utiliser les bonnes options du compilateur et de l'interpréteur :

```
javac -classpath /import/login/path1/classes -d /import/login/path1/classes \
      -sourcepath /import/login/path2/sources secondpackage/MyClass.java
java --classpath /import/login/path1/classes
```

L'option *classpath* indique le répertoire où le compilateur et/ou la machine virtuelle vont chercher la hiérarchie des fichiers .class pour les réutiliser. L'option *-d* indique l'endroit de base où seront écrits les fichiers .class générés à la compilation, et est généralement inclus dans le classpath. Enfin, l'option *-sourcepath* indique le répertoire où le compilateur va chercher l'arborescence des fichiers sources. Aussi, ces deux commandes peuvent être exécutées depuis n'importe quel répertoire. Notez que vous pouvez définir le classpath dans une variable système :

```
export CLASSPATH=/import/login/path1/classes
```

Exercice 1. Reprendre les fichiers du premier TP et mettre le code dans le package `fr.ens_cachan.dptinfo.chess`. Compiler et exécuter le code.

2 Archives Jar

Pour diffuser largement des classes Java qui ont été développées, et faciliter leur utilisation, il est possible de créer des archives Jar. Une archive Jar est un fichier qui contient des fichiers .class, éventuellement d'autres fichiers comme des données, le tout de manière compressée. Pour créer une archive Jar contenant des fichiers il suffit de taper la commande :

```
jar cvf file.jar firstpackage secondpackage
```

Le 'c' spécifie qu'on crée un fichier, le 'v' que l'on est dans un mode verbeux qui détaille les opérations, et le 'f' que l'on travaille sur un fichier (ce qui sera toujours le cas). Cette commande va créer un fichier file.jar qui contient l'intégralité des répertoires firstpackage et secondpackage.

Quand on récupère un fichier jar, on peut savoir la liste des fichiers qu'il contient avec la commande suivante :

```
jar tf file.jar
```

Avec 'tvf' au lieu de 'tf' on obtient plus d'infos sur les fichiers contenus, comme les droits de lectures/écritures. On peut aussi extraire les fichiers contenus dans le .jar :

```
jar xvf file.tar
```

Cependant, il est préférable d'utiliser directement le fichier .jar sans le décompresser si on ne compte pas modifier les classes qui y sont contenues. Aussi on peut mettre indiquer le chemin d'un .jar dans un classpath pour pouvoir compiler un fichier source qui utilise les classes qui y sont définies. On pourrait ici utiliser les paquets *firstpackage* et *secondpackage* en passant le chemin de file.jar à l'option *-classpath* du compilateur.

D'autre part, il est possible de lancer un exécutable depuis un .jar. Il faut pour cela spécifier dans un fichier spécial, le Manifest, la classe dont la méthode main sera exécutée. Le manifest contient plus d'informations que celà, mais il suffit d'écrire un fichier texte, par exemple Manifest.txt, qui contiendra une seule ligne qui sera ajoutée au manifest de l'archive :

```
Main-Class: secondpackage.MyClass
```

On ajoutera alors une option et un argument à la création de l'archive :

```
tar cmvf Manifest.txt file.jar firstpackage secondpackage
```

et on pourra l'exécuter avec l'option -jar de l'intepréteur :

```
java -jar file.jar
```

Exercice 2. Créer une archive Jar de votre code pour le jeu d'échecs, et exécuter le programme principal directement depuis l'archive.

3 Javadoc

Vous avez vu lors des TPs précédents que les commentaires en Java ont plusieurs notations possible. Ils peuvent notamment être écrits en style C/C++ :

```
// This is a single-line comment
/* This is another single-line comment */
/*
   This is a multi-line
   comment
 */
/*
 * This is another
 * multi-line comment
 */
```

D'autre part, vous avez pu remarquer des commentaires avec des notations particulières assez explicites :

```
/**
 * Multiplies two int
 *
 * @param x first int to multiply
 * @param y second int to multiply
 * @return the result of x times y
 *
 */
public static int multiply (int x, int y) {
  return x * y;
}
```

Il ne s'agit pas seulement d'une convention de notations qui force à la rigueur. Elles permettent surtout de générer des fichiers HTML de description d'API grâce à la commande

javadoc. Cet utilitaire va analyser les commentaires, et grâce à certaines notations explicites, va générer des pages HTML pour décrire les classes, leurs attributs, leurs méthodes, et s'occupera de générer lui-même des liens entre différentes méthodes en fonction des commentaires ou de l'organisation.

3.1 Écrire des commentaires Javadoc

Javadoc ne va analyser que les commentaires qui commencent par */*** et finissent par **/*. Les étoiles au début de ligne ne sont pas obligatoire, mais c'est une convention largement adoptée. Un commentaire Javadoc est censé décrire une classe, un attribut ou une méthode. Aussi il est censé se trouver directement au-dessus de la déclaration d'une classe, d'un attribut ou d'une méthode. Un commentaire qui serait placé ailleurs ne sera pas analysé par javadoc.

```
/**
 * A class for points with discrete coordinates
 */
public class DiscretePoint {

  /**
   * The horizontal coordinate of the point
   */
  private int x;

  /**
   * This is NOT a valid javadoc comment
   */

  /**
   * The vertical coordinate of the point
   */
  private int y;

  /**
   * Moves the point
   */
  public void move (int dx, int dy) {
    /**
     * This is NOT a valid javadoc comment
     */
    x += dx;
    y+ = dy;
  }
}
```

Le texte présent dans les commentaires valides sera entièrement repris dans les fichiers HTML générés lors de l'exécution de Javadoc. En plus d'une description, il est possible, et même fortement recommandé, d'utiliser des indications précises (les *tags*) pour spécifier les paramètres, les valeurs de retour ...

```
/**
 * Creates a new DiscretePoint
 *
 * @param x the horizontal coordinate of the point
 * @param y the vertical coordinate of the point
 */
public DiscretePoint (int x, int y) {
  this.x = x;
  this.y = y;
}

/**
 * Creates a new point at the given relative distance
 *
 * @param dx the horizontal relative distance of the new point
 * @param dy the vertical vertical distance of the new point
 * @return the new point
 */
public DiscretePoint relativePoint (int dx, int dy){
  return new DiscretePoint (x + dx, y + dy);
}
```

On peut vouloir spécifier un peu plus que les paramètres ou les valeurs de retour des méthodes et constructeurs. Voici une liste des tags Javadoc que vous voudriez normalement utiliser. Ils sont indiqués dans l'ordre conventionnel si vous voulez en spécifier plusieurs :

1. `@param`
2. `@return`
3. `@exception`
4. `@author`
5. `@see`

Le tag `@param` permet de décrire les arguments des constructeurs et méthodes, mais également les types génériques des constructeurs, méthodes et classes paramétrées. Il est recommandé de toujours spécifier tous les arguments puis tous les types génériques :

```
/**
 * Returns the greatest of two Comparable object of the same class.
 *
```

```
 * @param u1 the first object
 * @param u2 the second object
 * @param U  the type of the objects
 * @return the greatest of u1 and u2
 */
public static <U extends comparable<U>> U getGreater (U u1, U u2){
  if (u1.compareTo(u2) > 0)
    return u1;
  else
    return u2;
}
```

Le tag `@return` doit décrire la valeur retournée par toute méthode dont le type de retour n'est pas `void`. Elle ne doit pas être utilisée pour un constructeur.

Le tag `@exception` décrit les exceptions qui peuvent être soulevées par une méthode.

```
import java.io.*;

/**
 * Write a given String through a FileWriter
 *
 * @param writer          the FileWriter where to write
 * @param data            the String to write
 * @exception IOException if the writing encounters a problem
 */
public static void writeInFile (FileWriter writer, String data)
                                                  throws IOException {
   writer.write (data, 0, data.length());
}
```

Le tag `@author` permet de spécifier le ou les auteurs d'une méthode ou d'une classe. Ce qui permet de s'y retrouver lors d'un projet avec de nombreux développeurs.

Enfin, `@see` permet de faire des liens vers des attributs ou des méthodes de la classe courante ou d'une autre classe, ou simplement d'indiquer de la documentation à consulter

```
/**
 * Sets the horizontal coordinate
 *
 * @param x the new horizontal coordinate
 * @see #x
 * @see SomeOtherClass#someMethod(char, int) someMethod
 * @see "The Absolute Guide To Everything"
 */
public void setX (int x){
  this.x = x;
}
```

Enfin, il est possible d'utiliser du code html dans les commentaires javadoc

```
/**
 * Do something dangerous.
 * <br/>
 * <b>USE WITH CAUTION</b>
 */
public static void riskyMethod (){ /* ... */}
```

De manière générale, il est recommandé de commenter tous les attributs, méthodes, classes et interfaces, ou au moins toutes celles qui sont déclarées `public`. Évidemment, tous les arguments, les valeurs de retours, et les exceptions soulevées doivent faire l'objet d'un tag correspondant.

3.2 Génerer la documentation javadoc

La commande *javadoc* s'utilise facilement :

```
javadoc DiscretePoint.java
```

Il existe de nombreuses options pour améliorer le comportement de javadoc :

1. `-encoding` *encodage* : spécifie l'encodage des fichiers sources. Vous aurez sûrement besoin de spécifier l'encodage "iso-8859-1" ou "utf-8".
2. `-d` *répertoire* : indique dans quel répertoire créer les fichier HTML
3. `-sourcepath` *répertoire* : indique dans quel répertoire chercher les fichiers java à analyser
4. `-author` : inclus les commentaires des tags `@author` dans les fichiers (absents sinon)
5. `-doctitle` *titre* : affichera le titre donné en argument en haut de la page principal de la documentation
6. `-header` *en-tête* : affichera un en-tête en haut de chaque page générée
7. `-bottom` *"bas de page"* : affichera un bas-de-page en bas de chaque page générée
8. `-public` : affichera uniquement les données déclarées `public`
9. `-protected` : affichera uniquement le données déclarées `public` et `protected`
10. `-private` : affichera toutes les données
11. `-link` *URL* : ajoute des liens vers une javadoc externe

D'autre part, il est possible de préciser la liste des options utilisées dans un fichier, et la liste des fichiers java à traiter dans un autre. Ainsi, on pourrait avoir un fichier options :

```
-encoding "iso-8859-1"
-d doc
-public
-bottom "<p>Projet Java 2009</p>"
```

Ainsi qu'un fichier sources

```
MyFile1.java
MyFile2.java
MyFile3.java
```

On compile alors de cette manière

```
javadoc @options @sources
```

Exercice 3. Essayez ainsi de compiler les sources commentées en Javadoc du premier TP. Ouvrez ensuite un navigateur, et visitez la page `file:///import/votrelogin/chemin/vers/doc/index.html`. Essayez de rajouter des tags javadoc pertinents, ou créez quelques classes jouets pour manipuler les tags javadoc, faire des liens entre méthodes de différents classes, etc.

4 Tests unitaires avec JUnit

Lors du développement d'un logiciel de grande envergure, il est recommandé de mettre en place des batteries de tests unitaires. Les tests unitaires sont des tests simples qui s'assurent du bon comportement des méthodes de base des classes. L'interêt, au-delà de prévenir des erreurs simplistes lors de la création d'une classe, est de pouvoir les lancer suite à n'importe quelle modification de fichiers qui peuvent interférer avec des classes écrites précédemment, pour vérifier que le comportement reste correct. Un *framework* qui contient quelques paquets et classes supplémentaires ainsi que de nouvelles notations permet de faciliter l'écriture et l'exécution automatique de tests unitaires : **JUnit**.

La principale utilisation de JUnit consiste à écrire des fonctions de test. Pour cela, on rajoute la notation `@Test` devant une fonction devant nécessairement retourner `void`. D'autre part, pour vérifier que des conditions sont satisfaites, on utilise des méthodes spécifiques d'assertions : `assertTrue(boolean)`, `assertFalse(boolean)`, `assertNotNull(Object)`, `assertEquals(int, int)` ...

```
import org.junit.*;

public class MyCLass {
  public int x;
  public int y;
  public int sum;

  public MyClass(int x, int y){
    this.x = x;
    this.y = y;
    this.sum = x + y;
  }
  @Test public void check () {
```

```
    MyClass c = new MyClass(2, 5);
    Assert.assertEquals(c.sum, 7);
  }
}
```

Pour lancer toutes les fonctions de test définies dans la description d'une classe, il suffit de lancer la commande :

```
org.junit.runner.JUnitCore.runClasses(MyClass.class);
```

Elle lancera automatiquement toutes les méthodes annotées `@Test` dans la classe `MyClass`. Idéalement, cette méthode sera appelée depuis le main d'un fichier `Test.java`.

D'autres annotations permettent de manipuler des tests. Ainsi, si on veut initialiser des champs avant chaque test, on peut annoter une méthode qui le fera avec `@Before`, et elle sera appelée avant chaque test. Si on veut libérer des données après chaque test, une méthode annotée `@After` s'en occupera. Enfin, si on veut ne pas exécuter un test temporairement, on peut rajouter un `@Ignore` devant le `@Test`.

D'autre part, on peut vouloir limiter le temps d'exécution d'un test : on le fait de cette manière

```
@Test(timeout=100) public void ...
```

Enfin on peut spécifier qu'on attend qu'un test renvoie une exception de cette manière :

```
@Test(expected=IndexOutOfBoundsException.class) public void ...
```

JUnit signalera alors si l'exception attendue est bien soulevée ou non.

5 Déploiement avec Ant

Comme vous avez pu le voir dans les sections précédentes, il existe de nombreux outils d'aide au développement d'applications Java, depuis le compilateur `javac` lui-même jusqu'au packaging `jar` et les tests JUnit. L'outil `ant` permet de faire cohabiter ces nombreux outils et d'automatiser leur exécution.

`ant` utilise un dialecte XML pour déclarer quelles actions sont possibles sur le code Java. Le fichier lu par défaut s'appelle `build.xml`, et ressemble typiquement à :

```
<?xml version="1.0"?>
<project name="chess" default="compile">

    <property name="src.dir" value="src/main"/>
    <property name="build.dir" value="build/classes"/>

    <target name="init">
        <mkdir dir="${build.dir}"/>
    </target>
```

```
    <target name="compile" depends="init">
        <!-- Compile the java code -->

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

Un appel à `ant` dans le répertoire courant va compiler les fichiers présents dans `src/main` et mettre les fichiers `.class` correspondants dans le répertoire `build/classes`.

5.1 Propriétés

Les propriétés `ant` servent à déclarer des constantes, comme `src.dir` et `build.dir` dans l'exemple précédent. Les valeurs sont ensuite référencées par `${src.dir}` et `${build.dir}`.

5.2 Cibles

Le cœur d'un fichier `build.xml` est constitué de cibles pour `ant`. Celui-ci effectue les tâches précisées à l'intérieur de la cible (par exemple la compilation du code par la tâche `javac`), après avoir effectué toutes les cibles prérequises (comme la cible `init` qui crée le répertoire `build`).

5.3 Chemins

Un des aspects désagréables de Java est le besoin de spécifier les chemins vers les classes déjà compilées. On peut spécifier de tels chemins dans `ant` à l'aide de l'élément `classpath` :

```
<target name="run" depends="build">
  <java classname="${main.class}">
    <classpath id="build.path.id">
      <pathelement location="${build.dir}" />
    </classpath>
  </java>
</target>
```

En donnant l'identifiant `build.path.id` à ce classpath, on peut le réutiliser ensuite dans des tâches qui dépendent de la tâche `build` par la syntaxe :

```
    <classpath>
      <path refid="build.path.id" />
      <!-- other paths -->
    </classpath>
```

5.4 Tâches

Les tâches `ant` couvrent la plupart des actions que l'on peut souhaiter faire sur du code Java : `javac`, `java`, `jar`, `javac`, `junit`, ... La liste complète des tâches pré-implémentées dans `ant` est disponible depuis la page `http://ant.apache.org/manual/tasksoverview.html`. Il est de plus possible d'implémenter de nouvelles tâches relativement facilement en Java.

Exercice 4. En vous aidant de la documentation des tâches `ant`, organiser un répertoire qui sépare le code source des fichiers compilés et de la documentation. Écrire un fichier `build.xml` qui

1. compile votre code,
2. permet de l'exécuter,
3. permet de générer un fichier Jar,
4. permet de générer sa documentation,
5. permet de supprimer tous les fichiers qui peuvent être générés depuis `ant`.

6 Gestion des dépendances avec Ivy

`ivy` est un gestionnaire de dépendances qui s'intègre avec `ant` pour télécharger des bibliothèques Java ou les trouver dans des repertoires partagés.

On peut par exemple vouloir utiliser JUnit dans un fichier `build.xml`, mais que faire si JUnit n'est pas installé ? En déclarant JUnit comme une dépendance de notre programme, `ivy` va se charger ensuite de trouver une version adéquate et de l'intégrer à l'environnement de compilation.

6.1 Installation automatique

S'il est possible d'installer `ivy` manuellement, il est plus simple d'écrire une tâche `ant` dans `build.xml` qui se charge de le télécharger et de l'installer (noter l'ajout de l'espace de noms d'`ivy` à la racine du fichier `build.xml`) :

```
<?xml version="1.0"?>
<project name="chess" default="compile"
         xmlns:ivy="antlib:org.apache.ivy.ant">

  <!-- ... -->

  <!-- Here is the version of ivy we will use. Change this property to
       try a newer version if you want -->
  <property name="ivy.install.version" value="2.0.0" />
  <property name="ivy.jar.dir" value="${lib.dir}" />
  <property name="ivy.jar.file" value="${ivy.jar.dir}/ivy.jar" />
```

```
<!-- Path to shared jars at ENS Cachan. -->
<property name="ivy.shared.default.root" value="/usr/share/java"/>
<property name="ivy.shared.default.artifact.pattern"
          value="[artifact]-[revision].[ext]"/>

<!-- Download ivy from the maven repository. -->
<target name="download-ivy" unless="offline">
  <mkdir dir="${ivy.jar.dir}"/>
  <get
    src="http://repo1.maven.org/maven2/org/apache/ivy/ivy/\
         ${ivy.install.version}/ivy-${ivy.install.version}.jar"
       dest="${ivy.jar.file}" usetimestamp="true"/>
  <!-- alternative source:
  http://www.apache.org/dist/ant/ivy/${ivy.install.version}/ivy.jar -->
</target>

<!-- Init Ivy. -->
<target name="init-ivy" depends="download-ivy">
  <path id="ivy.lib.path">
    <fileset dir="${ivy.jar.dir}" includes="ivy*.jar"/>
  </path>
  <taskdef resource="org/apache/ivy/ant/antlib.xml"
           uri="antlib:org.apache.ivy.ant"
           classpathref="ivy.lib.path"/>
</target>

<!-- Clean Ivy. -->
<target name="clean-cache" depends="init-ivy">
  <ivy:cleancache />
</target>
```

6.2 Utilisation d'`ivy`

Il existe en fait deux façons d'utiliser `ivy` depuis `ant` :

retrieve pour ajouter des dépendances dans un répertoire cible, habituellement `lib`,

cachepath pour manipuler les chemins d'accès pour que les tâches `ant` trouvent les dépendances.

On peut bien sûr mélanger les deux utilisations, même si cela risque de rendre le fichier `build.xml` assez illisible.

6.2.1 Mode `retrieve`

On fournit dans cette utilisation un fichier `ivy.xml` qui détaille les dépendences de notre application :

```
<?xml version="1.0" encoding="UTF-8"?>
<ivy-module version="2.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://ant.apache.org/ivy/schemas/ivy.xsd">
  <info organisation="ens-cachan" module="chess" />
  <dependencies>
    <dependency org="junit" name="junit" rev="latest.integration"
                conf="default"/>
    <!-- other dependencies -->
    <!-- ... -->
  </dependencies>
</ivy-module>
```

On peut spécifier comment nommer les fichiers téléchargés par `ivy` :

```
  <!-- Pattern for retrieved files. -->
  <property name="ivy.retrieve.pattern"
            value="${ivy.lib.dir}/[artifact].[ext]" />
```

La résolution des dépendences se fait alors dans le fichier `build.xml` par une tâche

```
  <target name="resolve" depends="init-ivy">
    <ivy:retrieve />
  </target>
```

qui va chercher JUnit sur Internet et dans les fichiers partagés sur la machine, et copier le fichier `junit.jar` dans `lib`. Les tâches qui utilisent JUnit n'ont plus qu'à ajouter une dépendence vers la tâche `resolve`, et à ajouter `lib/junit.jar` à leur classpath.

6.2.2 Mode `cachepath`

Dans cette utilisation, `ivy` va télécharger les dépendences dans un cache local et ajouter directement les chemins utiles à un classpath `ant` :

```
  <target name="junit" depends="init-ivy">
    <ivy:cachepath organisation="junit" module="junit"
                   pathid="junit.path.id" inline="true" type="jar"/>
  </target>
```

Les tâches `ant` qui utilisent JUnit vont ajouter la tâche `junit` à leurs prérequis, et pouvoir faire référence au chemin `junit.path.id`.

Exercice 5. Intégrer le lancement des tests unitaires à votre fichier `build.xml` en utilisant `ivy` pour récupérer JUnit. Séparer proprement les fichiers qui servent aux tests (`src/test` et `build/test-classes`) et les rapports d'erreurs (`build/test-reports`) des autres.