



APACHE MAVEN 3

Durée : 1 jours



Modules

- **Module 1 : Introduction à Apache Maven**
- **Module 2 : Mise en Place d'un premier projet**
- **Module 3 : Test Unitaire avec Maven**
- **Module 4 : Couverture de test avec maven**
- **Module 5 : POM / Life Cycles**
- **Module 6 :**
- **Module 7 :**



MODULE 1

Introduction à Scala

Définition

Maven est un outil de gestion de projet qui comprend :

- un **modèle objet** pour définir un projet,
- un **ensemble de standards**,
- un **cycle de vie**,
- et un **système de gestion des dépendances**.

Le site web officiel est <http://maven.apache.org>

De quoi s'agit-il ?

- **Un outil de construction d'application**

- Génère une application « déployable » à partir d'un code source
- Compile
- Exécute des tests

- **Un outil de gestion de développement**

- Documentation & Rapports
- Site web
- ...

Motivations de Maven 1/2

- **Modèle abstrait de projet (POM)**
 - Orienté objet, héritage
 - Séparation de préoccupations
- **Cycle de vie standard**
 - Séquencement d'états (goal) standards
 - Action des plugins en fonction des états
- **Structure « standard » de projet**
 - Nommage standard des variables (src.dir, ...)

Motivations de Maven 2/2

- **Gestion automatique des dépendances avec d'autres projets**
 - Chargement des MAJ
- **Dépôts des projets**
 - publiques ou privés, local ou distants
 - caching et proxy
- **Extensible via l'ajout des plugins**
 - Eux même des projets Maven

Les différentes versions de Maven

Plusieurs versions majeures de Maven ont été diffusées :

- version 1.0, juillet 2004
- version 2.0, octobre 2005
- version 3.0, octobre 2010
- version 3.1, juillet 2013

Maven 2 est très différent de Maven 1.

Maven 3 est compatible avec Maven 2 et apporte de meilleures performances.

Comparaison de Maven et de Ant

Rappel sur ANT

- Remplaçant de make (pour les développements Java)
- Séquenceur de tâches (regroupées en cible)
- Très grand nombre de tâches développées

Défaut

- Pas de structure « standard » de projet
- Pas de cycle de vie « standard » d'un projet
- Pas de métadonnées « standard » sur les projets Pas de séparation de préoccupation
- Libs externes à récupérer pour le projet et pour les tâches problème de la MAJ des versions

Simple fichier build.xml pour Ant

```
1 <project name="my-project" default="dist" basedir=".">
2   <description> simple example build file</description>
3
4   <!-- set global properties for this build -->
5   <property name="src" location="src/main/java"/>
6   <property name="build" location="target/classes"/>
7   <property name="dist" location="target"/>
8
9   <target name="init">
10    <!-- Create the time stamp -->
11    <tstamp/>
12    <!-- Create the build directory structure used by compile -->
13    <mkdir dir="${build}"/>
14  </target>
15
16  <target name="compile" depends="init" description="compile the source " >
17    <!-- Compile the java code from ${src} into ${build} -->
18    <javac srcdir="${src}" destdir="${build}"/>
19  </target>
20
21  <target name="dist" depends="compile" description="generate the distribution" >
22    <!-- Create the distribution directory -->
23    <mkdir dir="${dist}/lib"/>
24    <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->
25    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
26  </target>
27
28  <target name="clean" description="clean up" >
29    <!-- Delete the ${build} and ${dist} directory trees -->
30    <delete dir="${build}"/>
31    <delete dir="${dist}"/>
32  </target>
33 </project>
```

Simple fichier pom.xml pour Maven

```
1 <project>
2   <modelVersion>4.0.0</modelVersion>
3   <groupId>com.formation.maven</groupId>
4   <artifactId>my-project</artifactId>
5   <version>1.0</version>
6 </project>
```

Installation de Maven

- Vérifier votre installation de Java
- Téléchargement de Maven
- Installer Maven
- Tester une installation Maven
- Configuration et dépôt spécifiques à l'utilisateur
- Obtenir de l'aide avec Maven

Installation de Maven

Vérifier votre installation de Java

Nous allons supposer dans ce chapitre est que vous avez déjà installé un JDK (Java Development Kit) approprié.

```
C:\Users\mbeng>java -version
java version "1.8.0_131"
Java(TM) SE Runtime Environment (build 1.8.0_131-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.131-b11, mixed mode)
```

Positionner les variables d'environnement

- JAVA_HOME=c:\prog\java1.8
- PATH=%JAVA_HOME%\bin

Installation de Maven

Téléchargement de Maven

- [http:// maven.apache.org/download.html](http://maven.apache.org/download.html)

Installer Maven

- Dézipper dans un répertoire ex: c:\prog

Positionner les variables d'environnement

- MAVEN_HOME=C:\prog\tools\maven-3
- %MAVEN_HOME%\bin

Installation de Maven

Tester une installation Maven

```
mvn -v
```

```
Apache Maven 3.5.0 (ff8f5e7444045639af65f6095c62210b5713f426;2017-04-03T21:39:06+02)
```

```
Maven home: C:\prog\tools\maven-35\bin\..
```

```
Java version: 1.8.0_131, vendor: Oracle Corporation
```

```
Java home: C:\prog\language\java8\jre
```

```
Default locale: fr_FR, platform encoding: Cp1252
```

```
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

Configuration et dépôt spécifiques à l'utilisateur

Configurer ~/.m2/settings.xml

- repositories, plugins repositories, proxies, ...

Installation de Maven

- Obtenir de l'aide avec Maven : **mvn -help**

```
C:\Users\mbeng>mvn -help

usage: mvn [options] [<goal(s)>] [<phase(s)>]

Options:
  -am,--also-make                If project list is specified, also
                                build projects required by the
                                list
  -amd,--also-make-dependents    If project list is specified, also
                                build projects that depend on
                                projects on the list
  -B,--batch-mode                Run in non-interactive (batch)
                                mode (disables output color)
  -b,--builder <arg>            The id of the build strategy to
                                use
  -C,--strict-checksums          Fail the build if checksums don't
                                match
  -c,--lax-checksums             Warn if checksums don't match
  -cpu,--check-plugin-updates    Ineffective, only kept for
                                backward compatibility
  -D,--define <arg>             Define a system property
  -e,--errors                    Produce execution error messages
  -emp,--encrypt-master-password <arg> Encrypt master security password
  -ep,--encrypt-password <arg>    Encrypt server password
  -f,--file <arg>               Force the use of an alternate POM
                                file (or directory with pom.xml)
  -fae,--fail-at-end             Only fail the build afterwards;
                                allow all non-impacted builds to
                                continue
  -ff,--fail-fast                Stop at first failure in
                                reactorized builds
  -fn,--fail-never               NEVER fail the build, regardless
                                of project result
  -gs,--global-settings <arg>    Alternate path for the global
                                settings file
  -gt,--global-toolchains <arg>  Alternate path for the global
```


Disposition de titre et de contenu avec liste

Question ?

Disposition de titre et de contenu avec liste

TP

Installation



MODULE 2

Mise en Place d'un premier projet

Maven : choix sous-jacents

- **Privilégier la standardisation à la liberté (Convention over Configuration)**

- Structure standard des répertoires d'une application
- Cycle de développement standard d'une application
- → Maven se débrouille souvent tout seul !

- **Factoriser les efforts**

- Un dépôt global regroupant les ressources/bibliothèques communes
- Des dépôts locaux
- Un dépôt personnel (~/.m2)

- **Multiplier les possibilités**

- Une application légère
- De nombreux plugins, chargés automatiquement au besoin

Maven : vocabulaire

- **POM** : Project Object Model (descripteur de projet maven)
- **Artefact** : Fichier résultat du build d'un projet
- **Groupe** : Espace de nom permettant le regroupement d'artefacts
- **Version** : Identification d'une version d'un artefact
- **Repository** : Référentiel (dépôt) d'artefacts
- **Archétype** : Modèle (prototype) de projet

Maven : vocabulaire

- **Plugin**

- Extension de l'application de base, proposant un ensemble de buts

- **But (Goal)**

- Tâche proposée par un plugin permettant de lancer un certain nombre d'action lorsqu'il est invoqué par `mvn plugin:but`.

- **Phase (Maven Lifecycle Phase)**

- Phase du cycle de développement d'un logiciel, généralement associée à des buts et exécutée par `mvn phase`

- **POM (Project Object Model)**

- Fichier xml décrivant les spécificités du projets (par rapport à Build.xml, décrit non pas tout, mais juste le « non standard »)

C'est quoi POM(P_{roject} O_{bject} M_{odel})?

Contient les informations nécessaires à Maven pour traiter le projet tel que:

- nom du projet
- numéro de version
- dépendances vers d'autres projets
- bibliothèques nécessaires à la compilation

Description d'un projet indépendante des actions à accomplir

- Orienté objet héritage du modèle

Le modèle de projet (pom.xml)

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  ...
</project>
```

Identifiant (unique) du projet :
Identifiant de l'artifact produit

type du projet:
pom, jar, war, ear, bundle, ...

dépendances du projet envers
d'autres projets (artifact)
constitue le `$CLASSPATH`

id d'une dépendance
version peut être un intervalle

portée de la dépendance par rapport au cycle
de vie (compile, provided, runtime, test)

la suite bientôt ...

Coordonnées maven

Identifiant

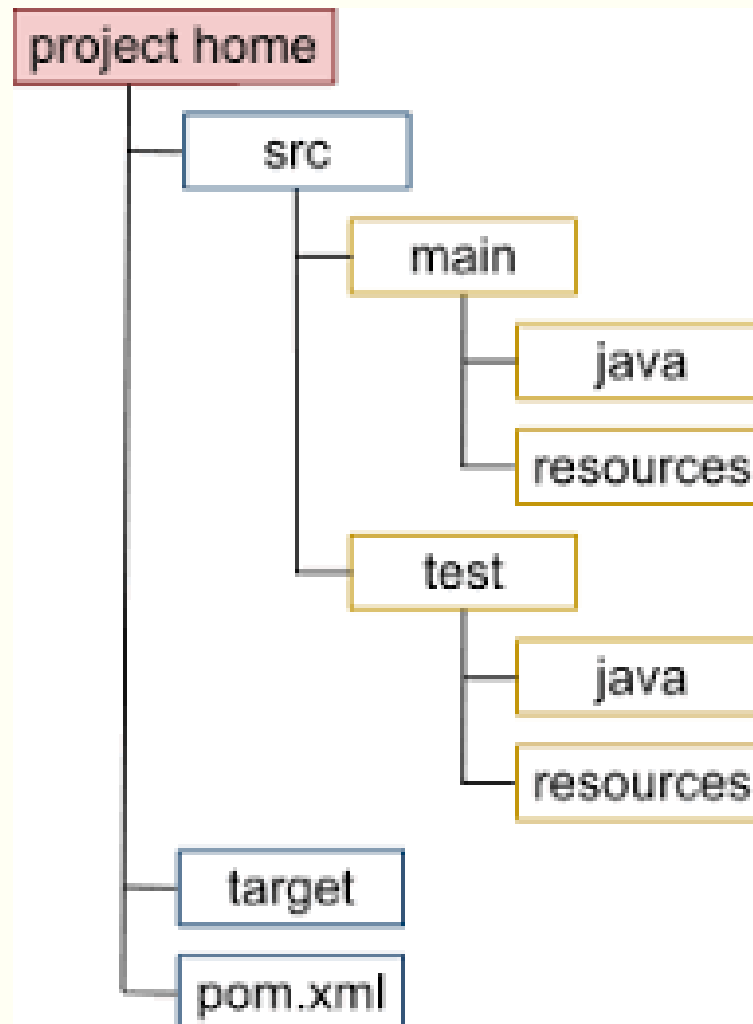
- **GroupId** : En général, le nom du domaine à l'envers
- **ArtifactId** : En général, le nom du projet
- **Version**
 - major.minor.micro-qualifier
 - Permet de spécifier une version minimale lors d'une dépendance avec order <num,num,num,alpha>
 - Si contient SNAPSHOT, remplacé par dateHeureUTC lors de la génération du package
 - Non importé par défaut
- **Packaging**
 - Type de packaging à produire (jar, war, ear, etc.)

Portée des dépendances

5 portées possibles par rapport aux classpaths du projet

- **Compile** : nécessaire à la compilation, et inclus dans le paquetage
- **Provided** : nécessaire à la compilation, non packagé (ex : Servlets)
- **Runtime** : nécessaire pour exécution et test, mais pas compilation (ex : driver jdbc)
- **Test** : nécessaire uniquement pour les tests
- **System** : comme provided + chemin à préciser, mais à éviter

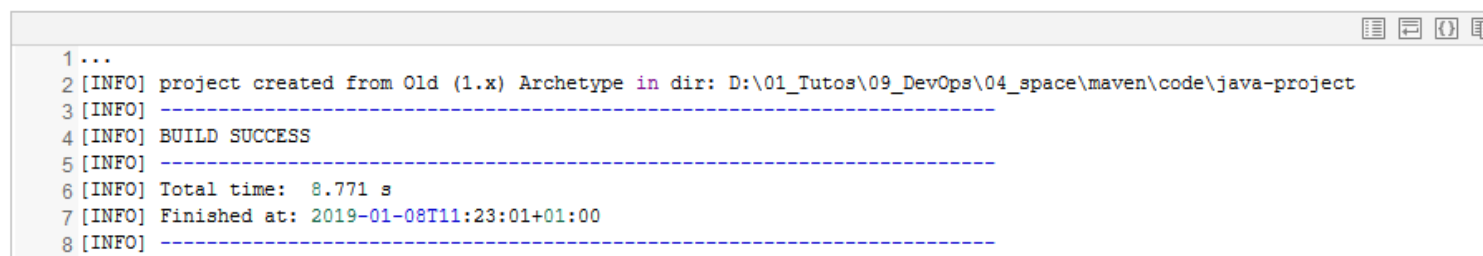
Structure des répertoires



Création d'un projet Maven Plugin archetype

Suggestion

- `mvn archetype:generate \`
 - DgroupId=com.formation \
 - DartifactId=java-project \
 - DarchetypeArtifactId=maven-archetype-quickstart \
 - DinteractiveMode=false

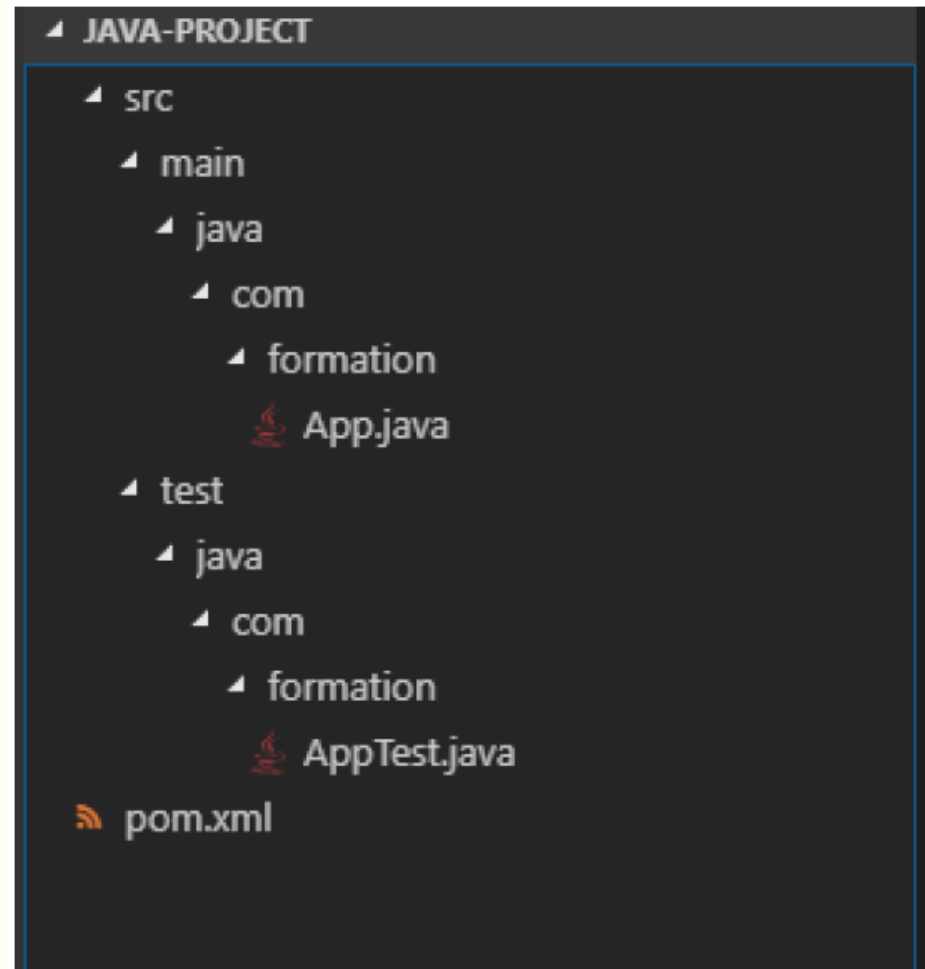


```
1 ...
2 [INFO] project created from Old (1.x) Archetype in dir: D:\01_Tutos\09_DevOps\04_space\maven\code\java-project
3 [INFO] -----
4 [INFO] BUILD SUCCESS
5 [INFO] -----
6 [INFO] Total time:  8.771 s
7 [INFO] Finished at: 2019-01-08T11:23:01+01:00
8 [INFO] -----
```

Remarques

- Les arguments non renseignés seront demandés à l'exécution

Structure d'un projet nouvellement créé



Fichiers de base générés : pom.xml

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
3   <modelVersion>4.0.0</modelVersion>
4   <groupId>com.formation</groupId>
5   <artifactId>java-project</artifactId>
6   <packaging>jar</packaging>
7   <version>1.0-SNAPSHOT</version>
8   <name>java-project</name>
9   <url>http://maven.apache.org</url>
10  <dependencies>
11    <dependency>
12      <groupId>junit</groupId>
13      <artifactId>junit</artifactId>
14      <version>3.8.1</version>
15      <scope>test</scope>
16    </dependency>
17  </dependencies>
18 </project>
```

Fichiers de base générés : App.java

```
package com.mycompany;

/**
 * Hello world!
 *
 */
public class App {
    public static void main( String[] args ) {
        System.out.println( "Hello World!" );
    }
}
```

Fichiers de base générés : AppTest.java

```
package com.mycompany;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

/**
 * Unit test for simple App.
 */
public class AppTest extends TestCase
{
    /**
     * Create the test case
     *
     * @param testName name of the test case
     */
```

```
public AppTest( String testName ){
    super( testName );
}
/**
 * @return the suite of tests being tested
 */
public static Test suite() {
    return new TestSuite( AppTest.class );
}
/**
 * Rigorous Test :-)
 */
public void testApp(){
    assertTrue( true );
}
}
```


Utilisation de base du projet - Compilation

Préambule

- Exécuter toutes les commandes depuis le répertoire contenant le fichier pom.xml

Compilation

- Commande
 - **mvn compile**
- Bilan
 - **my-app/target/classes/com/fmycompany/App.class**
- Exécution (à des fins de test)
 - **mvn exec:java -Dexec.mainClass=com.mycompany.App**

Utilisation de base du projet - Packaging et Installation

Packaging

- Commande : **mvn package**
- Bilan
 - Ne recompile pas le code, mais compile la classe de test
 - Exécution avec succès du seul test unitaire
 - Création de
 - – my-app/target/test-classes
 - – my-app/target/surefire-reports
 - – my-app/target/my-app-1.0-SNAPSHOT.jar

Installation

- Commande : **mvn install**
- Bilan
 - Ré-exécution des tests (*pour éviter cela, **mvn install -Dmaven.test.skip=true**)
 - Création de \$HOME/.m2/repository/com/mycompany/my-app/1.0-/my-app-1.0.jar, ...

Disposition de titre et de contenu avec liste

Question ?

Disposition de titre et de contenu avec liste

TP

Projet Java



MODULE 3

Test Unitaire avec maven

JUnit

- La référence du tests unitaires en Java
- Trois des avantages de l'eXtreme Programming appliqués aux tests :
 - Comme les tests unitaires utilisent l'interface de l'unité à tester, ils amènent le développeur à réfléchir à l'utilisation de cette interface tôt dans l'implémentation
 - Ils permettent aux développeurs de détecter tôt des cas aberrants
 - En fournissant un degré de correction documenté, ils permettent au développeur de modifier l'architecture du code en confiance

Junit - Exemple

```
1 package com.formation;
2
3 public class Anagramme {
4
5     public static String getReverse(String mot) {
6         StringBuilder newMot = new StringBuilder(mot);
7         return newMot.reverse().toString();
8     }
9
10    public static int getSize(String mot) {
11        return mot.length();
12    }
13
14 }
15
```

Junit - Exemple

```
1 package com.formation.test;
2
3 import static org.junit.jupiter.api.Assertions.assertEquals;
4
5 import org.junit.jupiter.api.Test;
6
7 import com.formation.Anagramme;
8
9 public class TestAnagramme {
10
11     @Test
12     public void testMot() {
13         assertEquals("otoT", Anagramme.getReverse("Toto"));
14     }
15
16     @Test
17     public void testSize() {
18         assertEquals(4, Anagramme.getSize("Toto"));
19     }
20 }
```


Les cas de test

- Ecrire des classes quelconques
- Définir à l'intérieur un nombre quelconque de méthodes annotés @Test
- Pour vérifier les résultats attendus (écrire des oracles !), il faut appeler une des nombreuses variantes de méthodes assertXXX() fournies
 - `assertTrue(String message, boolean test), assertFalse(...)`
 - `assertEquals(...)` : test d'égalité avec equals
 - `assertSame(...), assertNotSame(...)` : tests d'égalité de référence
 - `assertNull(...), assertNotNull(...)`
 - `Fail(...)` : pour lever directement une `AssertionFailedError`
 - *Surcharge sur certaines méthodes pour les différents types de base*
 - **Faire un « `import static org.junit.Assert.*` » pour les rendre toutes disponibles**

Fixture : contexte commun

- Des classes qui comprennent plusieurs méthodes de test peuvent utiliser les annotations `@Before` et `@After` sur des méthodes pour initialiser, resp. nettoyer, le contexte commun aux tests (= *fixture*)
 - Chaque test s'exécute dans le contexte de sa propre installation, en appelant la méthode `@Before` avant et la méthode `@After` après chacune des méthodes de test
 - Pour deux méthodes, exécution équivalente à :
 - `@Before-method ; @Test1-method(); @After-method();`
 - `@Before-method ; @Test2-method(); @After-method();`
 - Cela doit assurer qu'il n'y ait pas d'effet de bord entre les exécutions de tests
 - Le contexte est défini par des attributs de la classe de test

Fixture au niveau de la classe

- **@BeforeClass**
 - 1 seule annotation par classe
 - Evaluée une seule fois pour la classe de test, avant tout autre initialisation **@Before**
 - Finalement équivalent à un constructeur...
- **@AfterClass**
 - 1 seule annotation par classe aussi
 - Evaluée une seule fois une fois tous les tests passés, après le dernier **@After**
 - Utile pour effectivement nettoyer un environnement (fermeture de fichier, effet de bord de manière générale)

Suite : organisation des tests

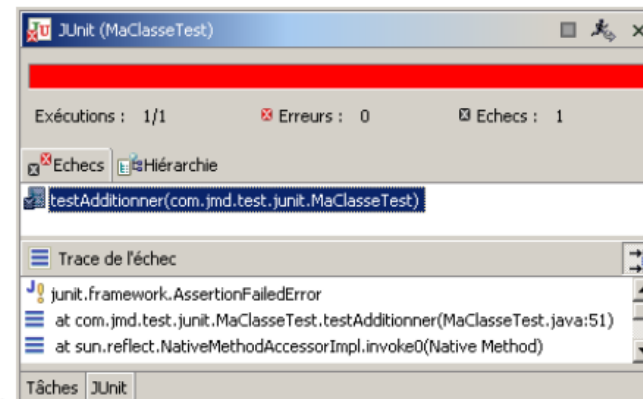
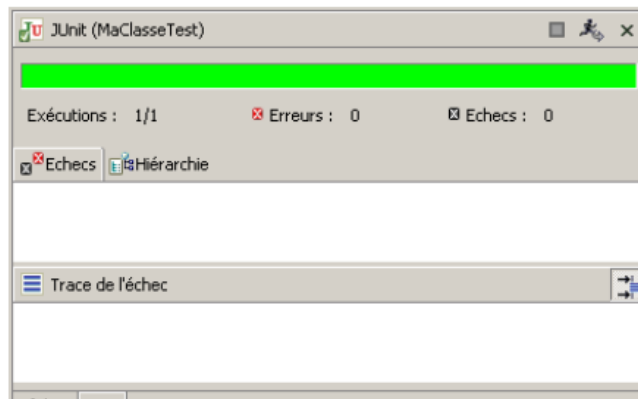
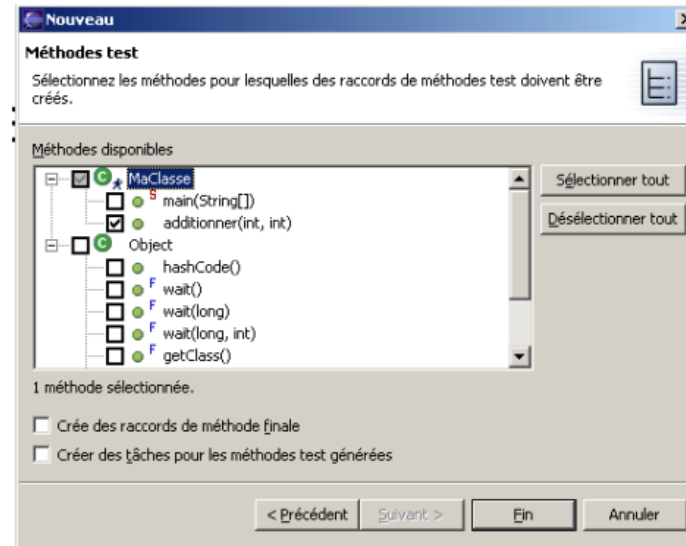
- Des classes de test peuvent être organisées en hiérarchies de « Suite »
 - Elles appellent automatiquement toutes les méthodes `@Test` de chaque classe de test
 - Une « Suite » est formée de classes de test ou de suites de test
 - Les tests peuvent être assemblés dans une hiérarchie de niveau quelconque, tous les tests seront toujours appelés automatiquement et uniformément en une seule passe.

```
@RunWith(value=Suite.class)
@SuiteClasses(value={CalculatorTest.class,
AnotherTest.class})
public class AllTests {
    ...
}
```

- Une « suite » peut avoir ses propres méthodes annotées `@BeforeClass` et `@AfterClass`, qui seront évaluées 1 seule fois avant et après l'exécution de toute la suite

JUnit dans Eclipse

- Assistants pour :
 - Créer des cas de test
- TestRunner intégré à l'IDE



Disposition de titre et de contenu avec liste

TP

Maven Junit 5



MODULE 4

Couverture de test avec maven

Concepts

- La couverture de code est une mesure importante car elle indique la qualité des tests et peut également jouer le rôle de condition d'arrêt des tests.
- L'utilisation d'un outil d'analyse de la couverture de code permet de savoir les parties du code qui n'ont pas encore été testées et pour lesquelles des tests additionnels devront être écrits.
- Il existe plusieurs outils d'analyse avec des fonctionnalités différentes: **Emma**, **Corbetura**, **Jacoco**.
- Jacoco est une librairie d'analyse de couverture de code pour Java.
- Cette librairie, gratuite et open source, est simple d'utilisation et flexible.
- Jacoco est développé sous le projet eclEmma. Jacoco permet de mesurer la couverture des lignes et des branches.
- Elle offre une visualisation graphique de la couverture du code et fournit des rapports détaillés de l'analyse de la couverture.

Glossaire

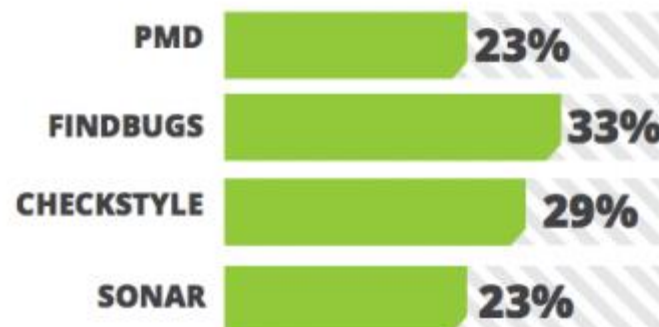
Mot	Définition	Exemple
Métrique	Une métrique est une caractéristique (ou une propriété) d'une application.	Nombre de lignes de code, pourcentage de couverture de code par les tests unitaires, nombre de violations de règles de codage, etc.
Mesure	Valeur d'une métrique à un moment précis de la vie de l'application.	Au 5 juin, l'application comptait 42.000 lignes de code, 317 violations et avait une couverture de code de 78 %.
Règle de codage	Définition d'une bonne pratique à respecter dans son code. Cela peut également concerner des mauvaises pratiques à éviter, car souvent source d'erreur.	Redéfinir la méthode hashCode lorsque l'on redéfinit equals, etc.
Violation (d'une règle de codage)	Non-respect, dans une partie du code précise, d'une règle de codage particulière.	

Qualité de Code: Outils disponibles sur le marché

- Il existe une multitude d'outils disponibles pour mesurer la qualité de code, souvent centrés sur un seul langage et possédant chacun ses propres spécificités
 - CodePro Analytix
 - **PMD**
 - **FindBugs**
 - **Cobertura**
 - Emma
 - **Checkstyle**
 - JBoss Tattletale
 - ...
- Cependant, pour des raisons pratiques, on ne peut implémenter tous ces outils.
- Il nous fallait ainsi trouver un outil pouvant regrouper si possible les précédents outils et proposer un support complet pour des projets pouvant énormément différer (différences de langages, différences d'outils d'intégration continue...).

SonarQube

- On peut au premier abord penser que Sonar est moins utilisé que ses confrères. Cependant, Sonar se base sur les outils PMD, Findbugs, Checkstyle et Squid.
- Par raccourci, utiliser Sonar revient à utiliser les technologies les plus utilisées actuellement sur le marché ce qui est un avantage non négligeable.
- Les avantages de Sonar sont :
 - Il supporte notamment plus de 25 langages
 - Il se combine avec Maven ou Ant ou Gradle (voir la partie Moteur de Production).
 - Il possède une forte communauté.



Checkstyle

- **Checkstyle** est une aide précieuse pour les développeurs afin de leur faire respecter des standards de codage précis.
- On y retrouve une liste assez complète de règles paramétrables (environ 130 règles) permettant de valider à peu près n'importe quel type de standard.
- On pourra ainsi vérifier que les lignes n'excèdent pas une certaine longueur, que la Javadoc est bien présente, que les standards de nommage sont bien respectés, etc...
- Checkstyle permet aussi d'améliorer l'écriture et la qualité de son propre code, en indiquant par exemple quelles expressions peuvent être simplifiées, quels blocs peuvent être supprimés, quelle classe doit être déclarée finale, etc.

PMD

PMD va scanner le code Java à la recherche de problèmes potentiels, tels que :

- bogues possibles, blocs de code vides (*try ... catch*, *switch*) ;
- code "mort", non utilisé (essentiellement des méthodes privées, ou des paramètres) ;
- expressions trop lourdes, mauvaises utilisations des opérateurs de boucles ;
- code dupliqué, "erreurs de copier-coller"...

Findbugs

Findbugs est un outil apprécié par les développeurs car il apporte une analyse assez fouillée de programmes Java, et permet de détecter des problèmes assez complexes.

Il pourra ainsi nous informer sur différents aspects, tels que :

- mauvaises pratiques ;
- utilisation de vulnérabilités de certains bouts de code ;
- lister des problèmes épineux ;
- possibles pertes de performances ;
- gestion du *multithreading* ;
- problèmes liés à la sécurité ;
- etc.

Disposition de titre et de contenu avec liste

TP

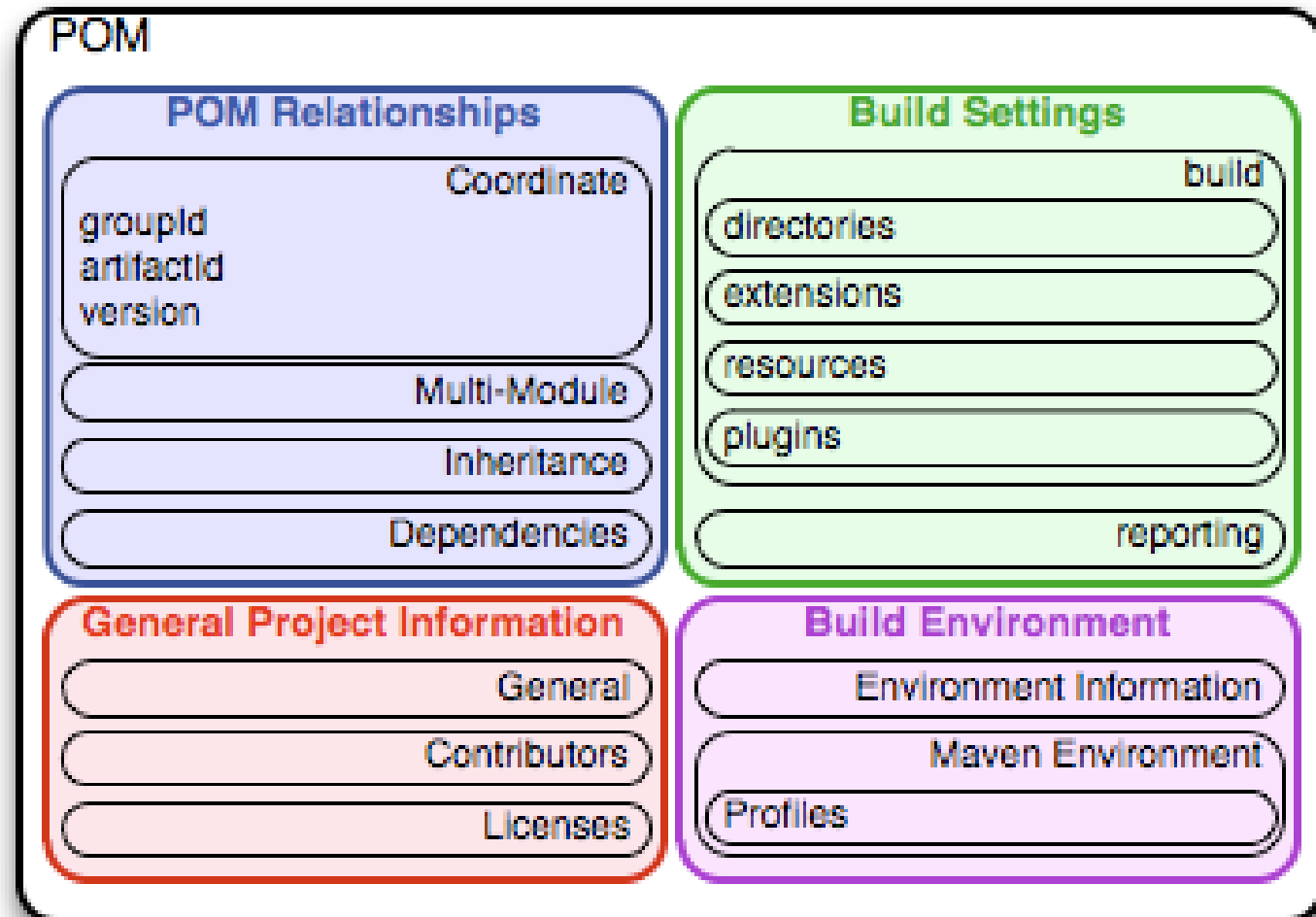
Couverture de test avec maven



MODULE 5

POM / Life Cycles

Le Modèle Objet de Projet (POM)



Le Modèle Objet de Projet (POM)

Le POM se compose de quatre catégories de description et de configuration :

- **Informations générales sur le projet**

Cette catégorie regroupe le nom l'URL et la licence du projet, l'organisation qui produit ce projet, et une liste de développeurs et de contributeurs.

- **Configuration du build**

Dans cette section, nous configurons le build Maven en personnalisant le comportement par défaut. Nous pouvons changer l'endroit où se trouvent les sources et les tests, ajouter de nouveaux plugins, lier des goals de plugins au cycle de vie et personnaliser les paramètres de génération du site web.

Le Modèle Objet de Projet (POM)

▪ Environnement du build

L'environnement du build consiste en un ensemble de profils qui peuvent être activés pour être utilisés dans différents environnements. Par exemple, au cours du développement vous pouvez vouloir déployer sur un serveur qui sera différent de celui sur lequel vous déploierez en production. L'environnement de build adapte la configuration du build pour un environnement spécifique et il s'accompagne souvent d'un fichier `settings.xml` personnalisé dans le répertoire `~/.m2`.

▪ Relations entre POM

Un projet est rarement isolé. Il dépend souvent d'autres projets, hérite d'une configuration de POM de projets parent, définit ses propres coordonnées, et peut comporter des sous-modules.

Super POM

(implicit) a-parent inherits from the Super POM

com.sonatype.maven
a-parent
1.0-SNAPSHOT

(explicit) project-a inherits from a-parent

com.sonatype.maven
project-a
1.0-SNAPSHOT

POM + Super POM = Effective POM

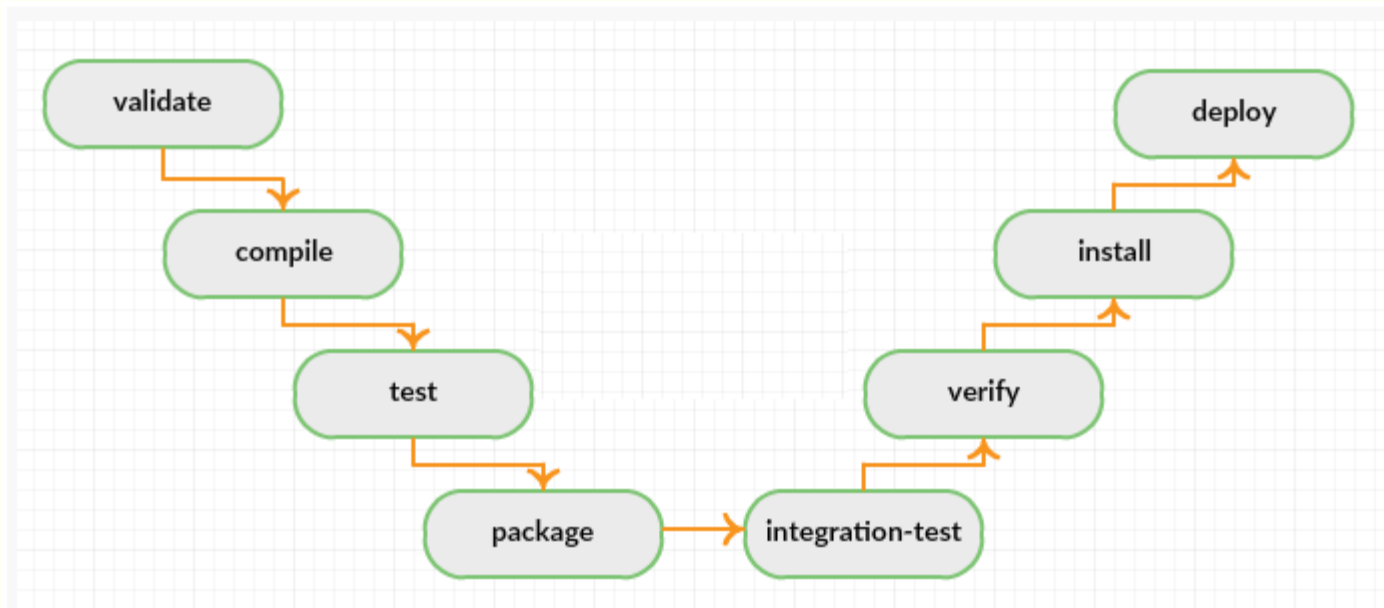
Mbengue Mohamadou

Maven Build Lifecycle

- Maven relies on build lifecycles to define the process of building and distributing artifacts (eg. Jar files, war files)
- There are three built-in build lifecycles
 - **Default** – handles project building and deployment
 - **Clean** – handles project cleaning
 - **Site** – handles project's site generation

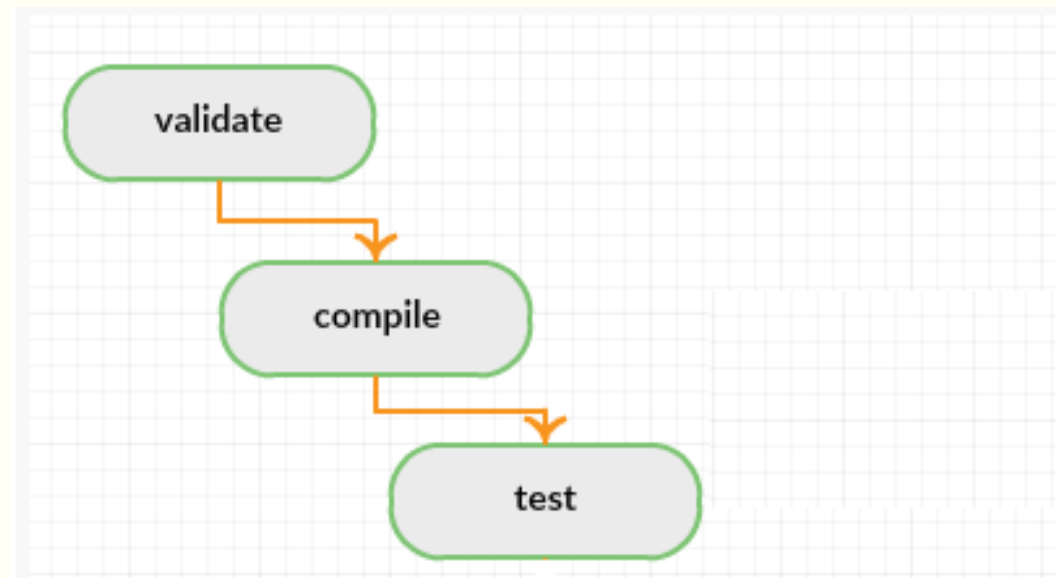
Maven Build Lifecycle

- Each Build Lifecycle is made up of phases
- **Example:** Validate Compile Test Package Install Deploy

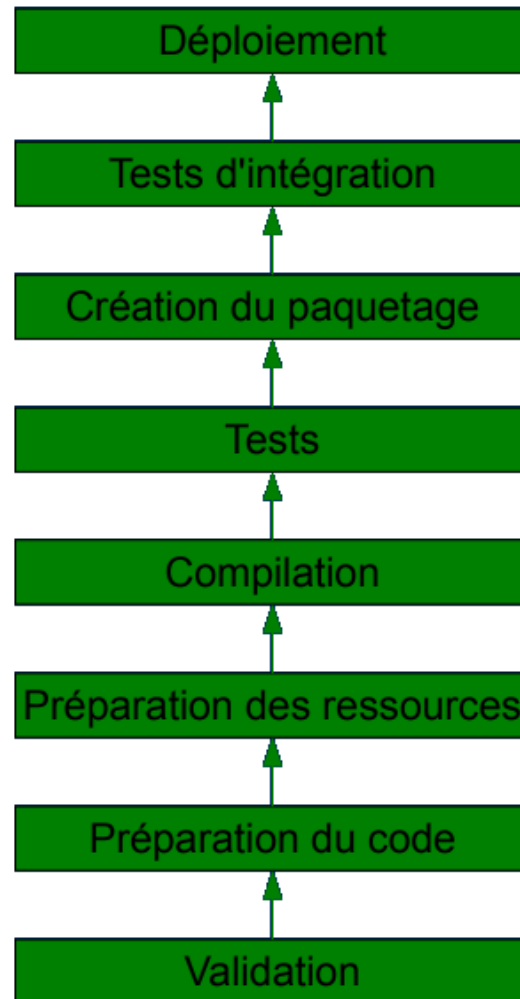


Maven Build Lifecycle

- You can invoke a specific phase on the lifecycle
 - Example: `$ mvn test`
- When you invoke a specific phase, every previous phases run, including the one you specified.
 - If you run: `$ mvn test`



Default Life Cycle



Déploiement du code en local ou sur site distant

Préparation et exécution des tests d'intégration

Génération du paquetage à déployer

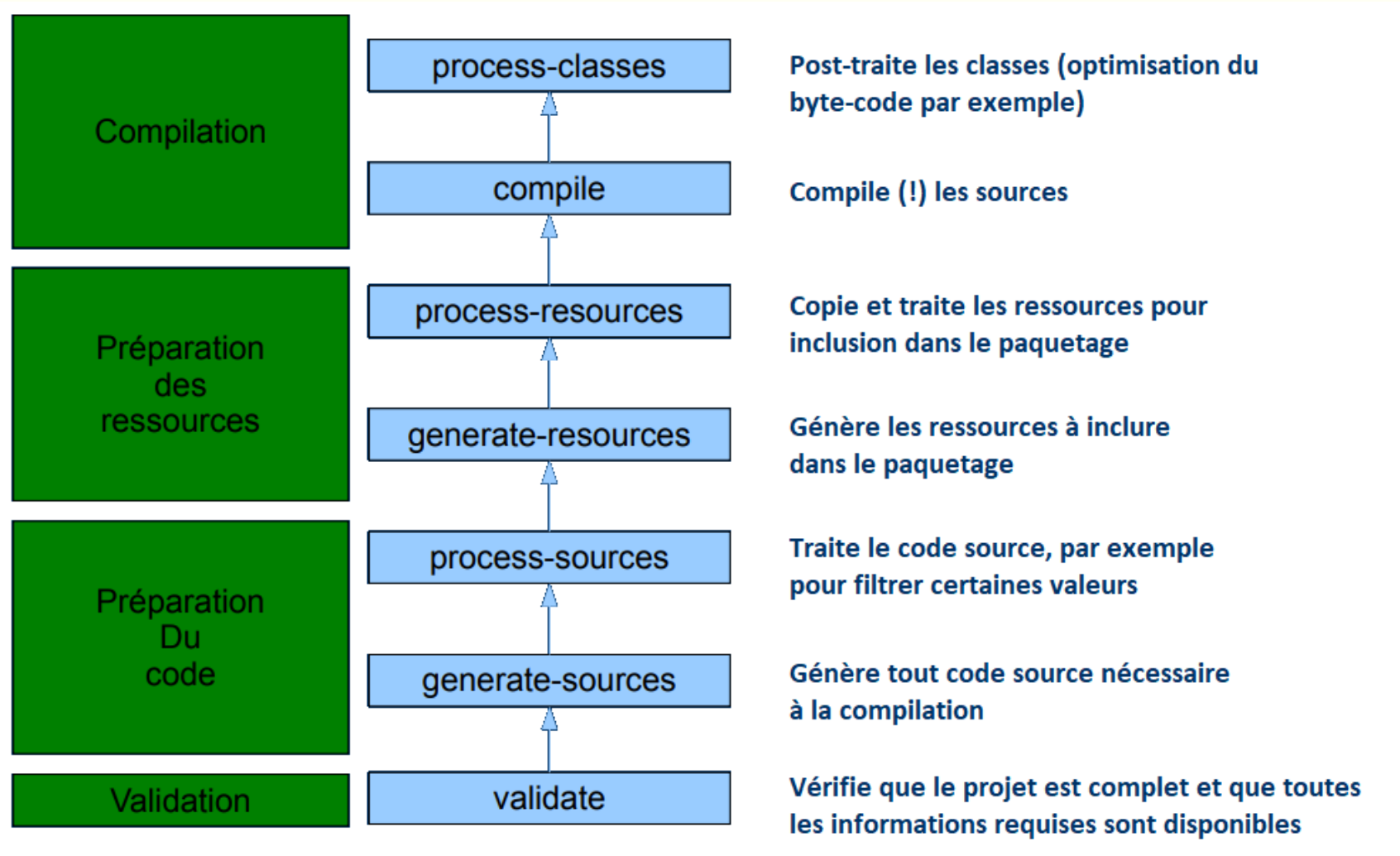
Préparation, compilation et exécution des tests

Compilation du code

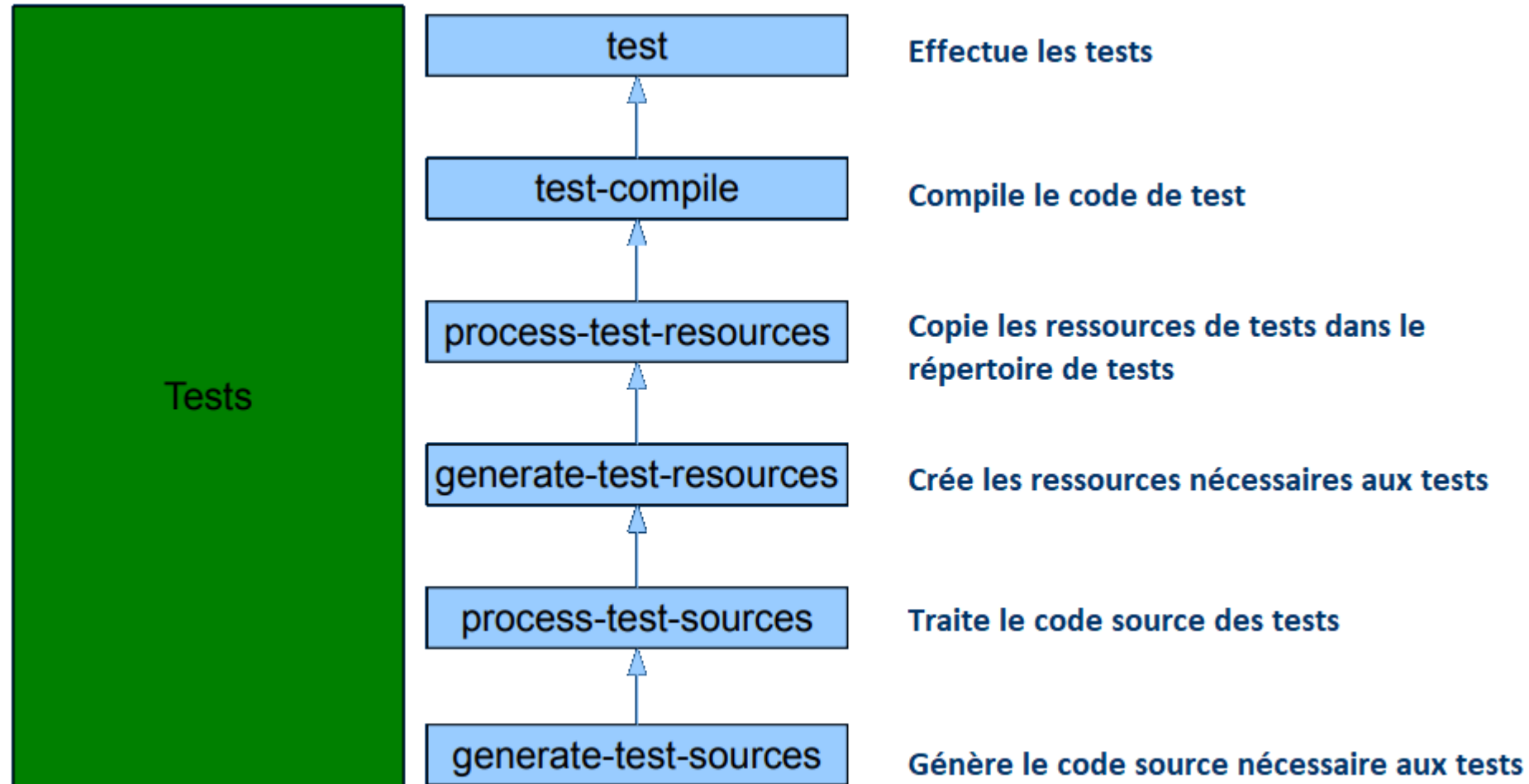
Vérification de la complétude du projet

Mbengue Mohamadou

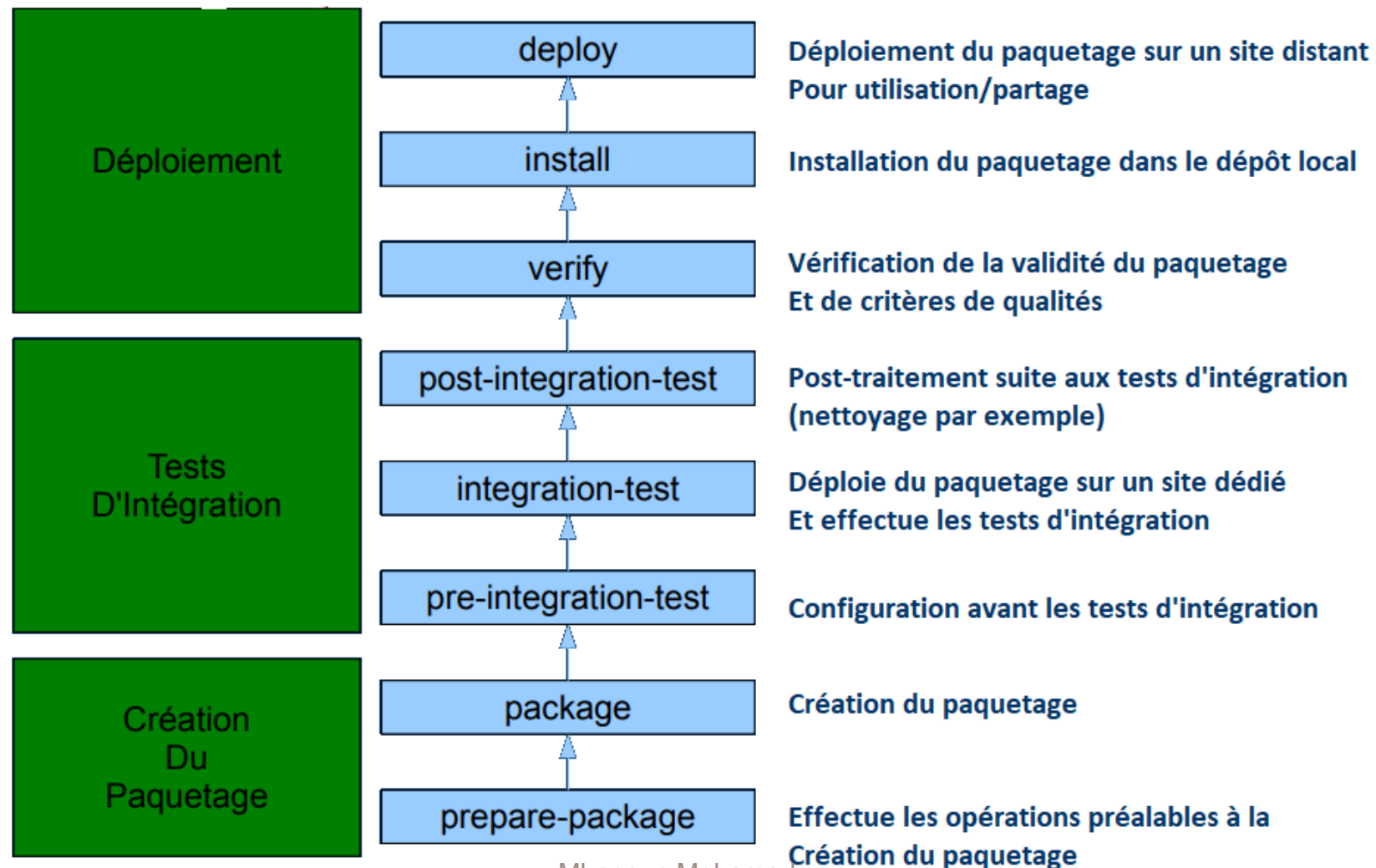
Default Life Cycle - De la Validation à la compilation



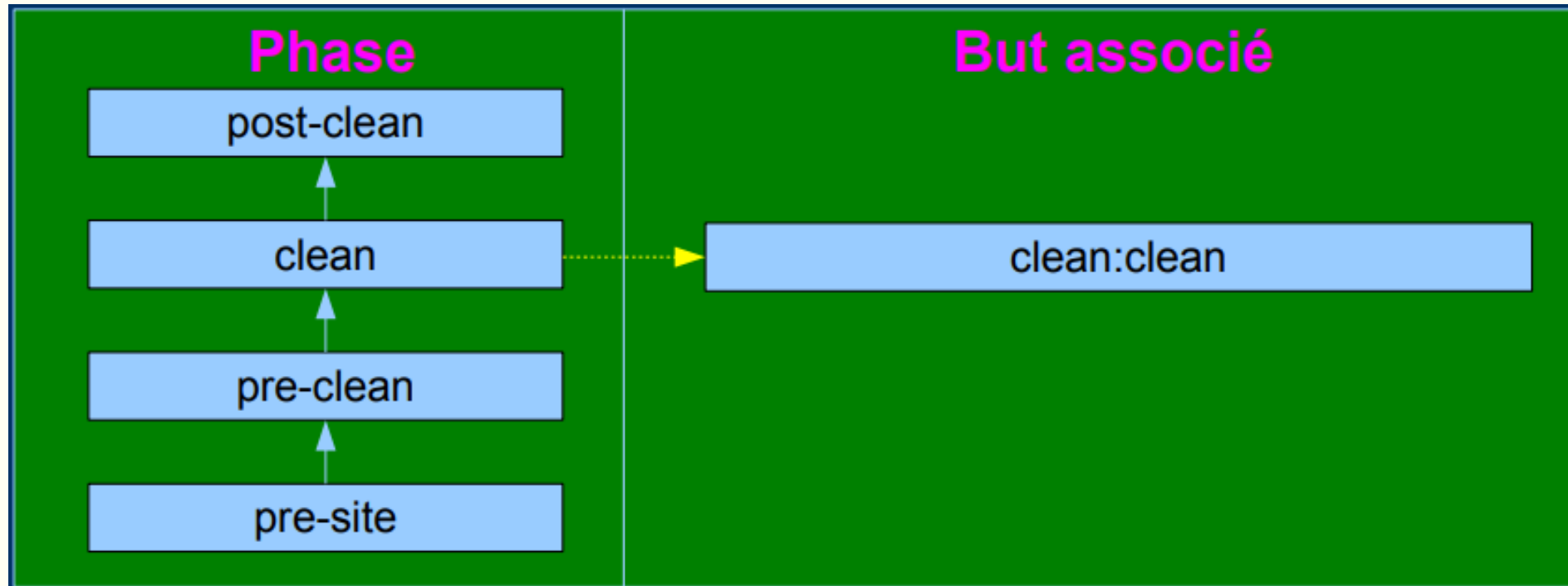
Default Life Cycle - Tests



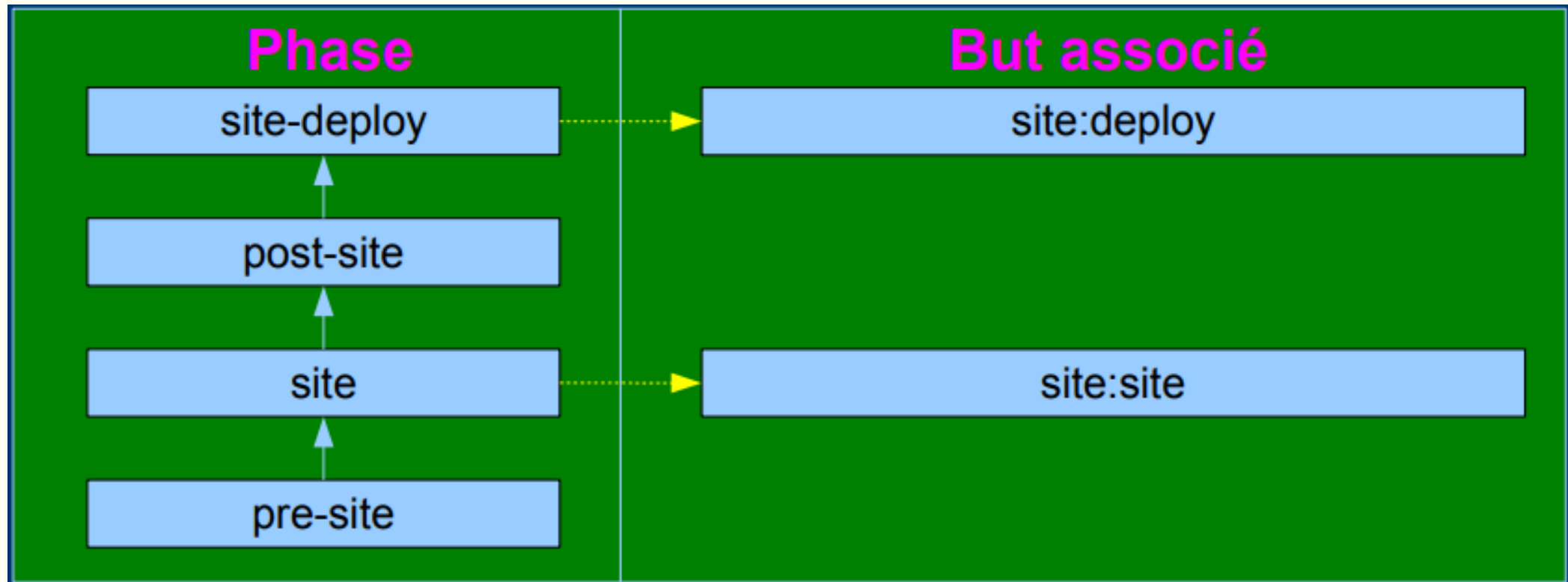
Default Life Cycle - Du packaging au déploiement



Clean lifecycle phases



Site lifecycle phases



Disposition de titre et de contenu avec liste

Question ?

Disposition de titre et de contenu avec liste

TP

Demo : Life Cycle