TP Industrialisation

Maven / GIT / Jenkins / SONAR / …:

Introduction Maven, git et Integration Continue

**Objectifs du TP**

**Comprendre le fonctionnement de maven.**

**Utiliser les artifacts**

**Configurer un projet eclipse avec maven.**

**Créer son propre MOJO**

**Générer des rapports maven**

**Utiliser Git pour sauvegarder le code source de votre projet**

**Utiliser un système d’Intégration Continue**

**Liens utiles**

* Site de Maven : <http://maven.apache.org/>
* Plugin Checkstyle : <http://maven.apache.org/plugins/maven-checkstyle-plugin/>
* FAQ MAVEN developpez.com : <http://java.developpez.com/faq/maven/>

**Environnement**

Pour ce TP, nous vous avons préparé un eclipse qui se trouve sur e: ou dans /Extras. Le nom est eclipse4Java, c’est comme son nom l’indique un simple eclipse pour Java Developper contenant le plugin maven et les outils pour faire du développement en équipe à l’aide de git.

Commencez par configurer dans les préférences le M2\_REPO pour le faire pointer sur w: plutôt que sur votre compte local.

Pour ceux qui n’aurait pas accès à w:, allez dans ws.istic.univ-rennes1.fr pour vous créer un accès sur w:. Vous devez vous déloguez et vous reloguez pour monter le disque w:

Pour ce faire, utilisez le fichier settings.xml suivant

<?xml version="1.0" encoding="UTF-8"?>  
<settings>  
 <localRepository>w:\mavenrepository</localRepository>  
 <offline>false</offline>  
</settings>

il se trouve aussi sur /share

**Partie 1 : Utilisation de maven**

Création d’une application basique.

Pour initialiser un projet Java, vous pouvez utiliser l’archetype maven *maven-archetype-quickstart.* Vous avez juste à fournir un groupid et un artefactid.

**Dans eclipse.**

new -> other -> maven -> maven project. Vous devrez sélectionner l’archetype,  l’artifactId et le groupId

**En ligne de commande (non nécessaire si vous l’avez fait depuis eclipse)**

mvn archetype:generate

 -DgroupId=fr.istic.master1.sir

 -DartifactId=tp1maven

 -DarchetypeArtifactId=maven-archetype-quickstart

Ou simplement,

mvn archetype:create

 -DgroupId=[your project's group id]

 -DartifactId=[your project's artifact id]

Vous obtenez la structure de projet jointe

|-- src

|   |-- main

|   |   `-- java

|   |       `-- [your project's package]

|   |           `-- App.java

|   `-- test

|       `-- java

|           `-- [your project's package]

|               `-- AppTest.java

 `-- pom.xml

Par exemple si vous exécutez la commande

mvn archetype:create

 -DgroupId=fr.istic.master1.sir

 -DartifactId=tpmaven

Vous obtiendrez l’architecture suivante.

|-- src

|   |-- main

|   |   `-- java

|   |       `-- fr

|   |           `-- istic

|   |               `-- master1

|   | --sir

|   |                       `-- App.java

|   `-- test

|       `-- java

|           `-- fr

|               `-- istic

|              `-- master1

| --sir

|                           ` -- AppTest.java

 `-- pom.xml

**Partie 2 : Configuration d’eclipse**

**Pour eclipse 4.X**

Dépuis eclipse 4.X, le support de maven s’est amélioré. Pour importer votre projet. File -> import -> maven -> existing maven project.

Votre projet est configuré.

**Partie 3: Gestion des dépendances**

Intégrer à votre code source le fichier suivant

<https://raw.githubusercontent.com/barais/swingx/master/JXBusyLabelTest>

Vous verrez que le code ne compile pas car il manque une dépendances.

Intégrez maintenant la dépendance à swingx.

<!-- https://mvnrepository.com/artifact/org.swinglabs/swingx -->

<dependency>

   <groupId>org.swinglabs</groupId>

   <artifactId>swingx</artifactId>

   <version>1.6.1</version>

</dependency>

Votre IDE va downloader la dépendance et la mettre automatiquement dans votre classpath. Dans ce sens, cela permet de ne mettre dans votre gestionnaire de source que le code source et le descripteur de projet (pom.xml).

**Partie 4: Spécialisation du process de build.**

Imaginons que vous souhaitiez ajouter une t^ache dans le processus de build. Par exemple, compilez votre code source avec la version Java 1.7

Ajoutez la section suivant à votre fichier pom.xml

<build>

   <plugins>

     <plugin>

       <groupId>org.apache.maven.plugins</groupId>

       <artifactId>maven-compiler-plugin</artifactId>

       <configuration>

         <!-- or whatever version you use -->

         <source>1.7</source>

         <target>1.7</target>

       </configuration>

     </plugin>

   </plugins>

 </build>

Vous pouvez ajouter de nombreux plugin dans cette section.

<https://maven.apache.org/plugins/>

**Partie 5 : Génération de rapports**

Générer la javadoc

Ajoutez des commentaires à votre code projet tp1

Ajouter le code suivant dans le pom.xml de votre projet

<reporting>

<plugins>  
  <plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-javadoc-plugin</artifactId>

</plugin>

</plugins>  
</reporting>

Puis lancez : mvn site

Par défaut, le site généré contient un certain nombre de pages accessibles par le menu de gauche.

La partie « Project Info » regroupe trois pages : la mailing liste, la liste des développeurs et les dépendances du projet.

La partie « Project report » permet d'avoir accès à des comptes rendus d'exécution de certaines tâches : javadoc, tests unitaires, ... Certaines de ces pages ne sont générées qu'en fonction des différents éléments produits par Maven.

Valider la qualité du code avec le plugin checkstyle

●  Ajoutez à la section <reporting> du projet client le plugin checkstyle

<reporting>  
   <plugins>

         <plugin>

            <groupId>org.apache.maven.plugins</groupId>

            <artifactId>maven-checkstyle-plugin</artifactId>

         </plugin>

   </plugins>  
</reporting>

Lancez : mvn site

Observez la section 'reporting' ajoutée dans le rapport récupéré dans \target\site\index.html

Quelle est la norme de codage à laquelle se réfère le rapport par défaut ? sun\_checks.xml ruleset  
1/ Comment imposer la norme de codage de Google? Télécharger fichier google\_checks.xml et rajouter dans le pom.xml :

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-checkstyle-plugin</artifactId>

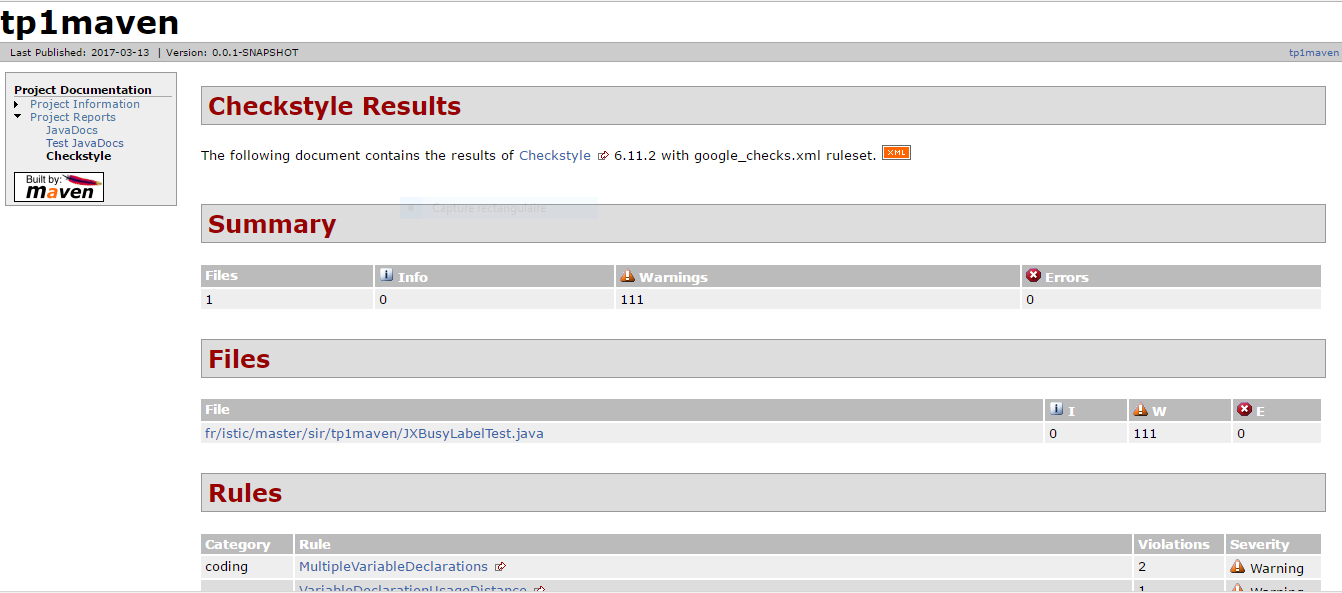
<version>2.17</version>

<configuration>

<configLocation>google\_checks.xml</configLocation>

</configuration>

</plugin>



Fichier de définition [https://raw.githubusercontent.com/checkstyle/checkstyle/master/src/main/resources/google\_checks.xm](https://raw.githubusercontent.com/checkstyle/checkstyle/master/src/main/resources/google_checks.xml)

Changer la version de checkstyle

<https://maven.apache.org/plugins/maven-checkstyle-plugin/examples/upgrading-checkstyle.html>

Changer le fichier de configuration à utiliser

<https://maven.apache.org/plugins/maven-checkstyle-plugin/examples/custom-checker-config.html>

2/ Modifiez votre classe du projet tp1 de façon à diminuer le nb d'erreur ? Liens utiles :

En changeant le fichier de checkstyle par google\_checks.xml, on n’a plus d’erreurs, seulement des warning.

Site de de l'outil CheckStyle : <http://checkstyle.sourceforge.net/>

Site du plugin Maven : <http://maven.apache.org/plugins/maven-checkstyle-plugin/>

Rapport croisé de source

<reporting>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId> <artifactId>maven-jxr-plugin</artifactId>

                                         </plugin>

</plugins>  
</reporting>

Lien utile : <http://maven.apache.org/plugins/maven-jxr-plugin/>

Quelle est la valeur ajoutée de ce plugin ? En particulier, montrez sa complémentarité avec CheckStyle.

## Maven JXR Plugin

Le plugin JXR produit une référence croisée des sources du projet. Les rapports générés facilitent la consultation ou la recherche de lignes de code spécifiques par l'utilisateur. Il est également pratique lorsqu'il est utilisé avec le Plugin PMD pour référencer les erreurs trouvées dans le code.

Désormais vous pouvez passer du rapport CheckStyle au code source en cliquant sur le numéro de ligne associé au commentaire CheckStyle.

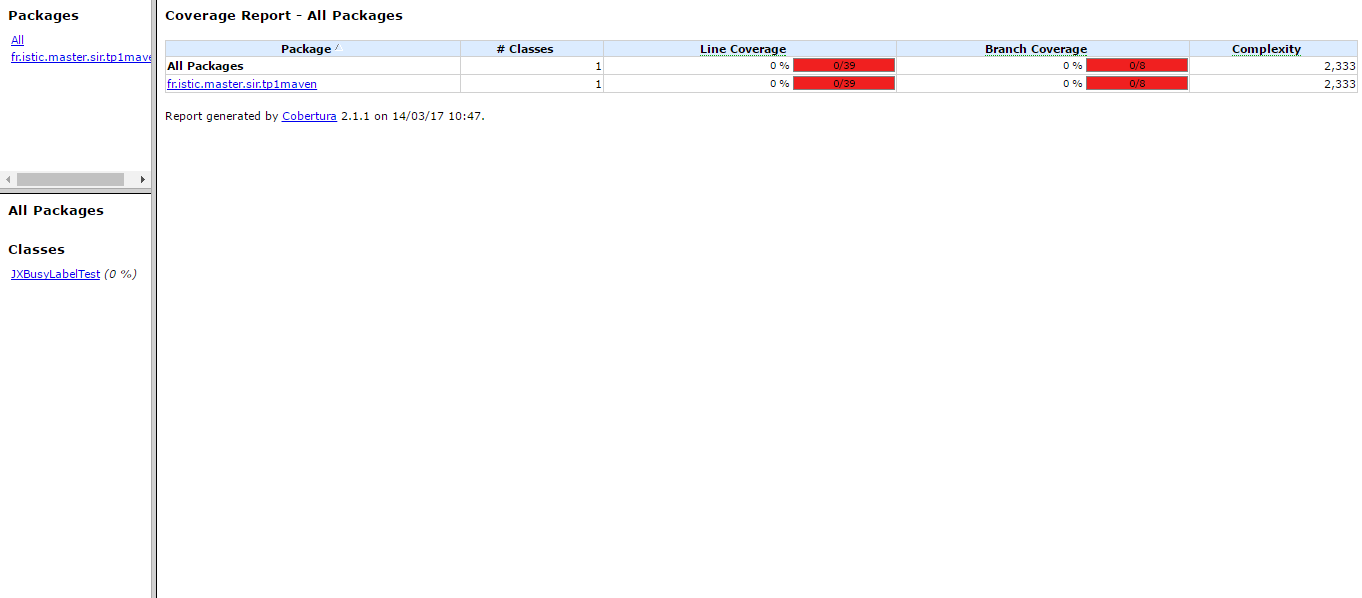
Couverture de test : plugin maven cobertura :

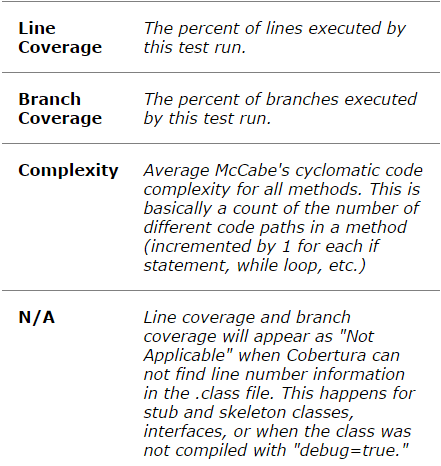
http://www.mojohaus.org/cobertura-maven-plugin/usage.html

Le rapport généré par ce plugin est le résultat de l'exécution de l'outil Cobertura contre vos classes compilées pour vous aider à déterminer dans quelle mesure les tests unitaires et les efforts d'essais d'intégration ont été et peut ensuite être utilisé pour identifier les parties de votre programme Java manquant test couverture.

A quel point les développeurs ont réalisé des tests unitaires ? Quelles parties de l'application n'ont pas été testées ?

Unit test for simple App.





<reporting>

 <plugins>

   <plugin>

     <groupId>org.codehaus.mojo</groupId>

     <artifactId>cobertura-maven-plugin</artifactId>

   </plugin>

 </plugins>  
</reporting>

Lien utile : <http://mojo.codehaus.org/cobertura-maven-plugin/usage.html>

Identifier patterns d'erreur avec PMD

Ajoutez volontairement du code mort à votre code projet tp1 (une méthode pas utilisée par exemple)

Identifiez :

Code mort (Ex : variables ou paramètres non utilisés)

Duplication de code (Code copié/collé = possible bug copié/collé Code 'compliqué' (Ex : trop de if...else)

Ajoutez à la section <reporting> du projet persist le plugin PMD

<project>

<reporting>  
         <plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-pmd-plugin</artifactId>

<version>2.5</version>

</plugin>

</plugins>  
</reporting>

</project>

Quels sont les deux nouveaux rapports générés ? CPD Report et PMD Report.

Qu'est ce que le rapport 'CPD Report' ?

CPD permet de trouver du code dupliqué dans Java, C, C ++, C #, Groovy, PHP, Ruby, Fortran, JavaScript, PLSQL, Apache Velocity, Scala, Objective C, Matlab, Python, Go, Swift et Salesforce.com Apex.

Qu'est ce que le rapport 'PMD Report' ?

PMD est un analyseur de code source. Il trouve des défauts de programmation courants comme des variables inutilisées, des blocs de capture vides, une création d'objet inutile, et ainsi de suite. Il supporte Java, JavaScript, Apex Salesforce.com, PLSQL, Apache Velocity, XML, XSL.

Connaître l'activité du projet

Quels et combien de fichiers modifiés par un développeur ?

Commiter votre projet sur github ou sur la forge de l’istic.

●  Ajoutez à la section <reporting> du projet persist le plugin changelog

<project>

<reporting>  
      <plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-changelog-plugin</artifactId> </plugin>

</plugins>  
</reporting>

<scm>

<connection>scm:git:git://github.com/CamilleBarbe/maven.git</connection>

<url>http://github.com/CamilleBarbe/maven.git</url>

</scm>

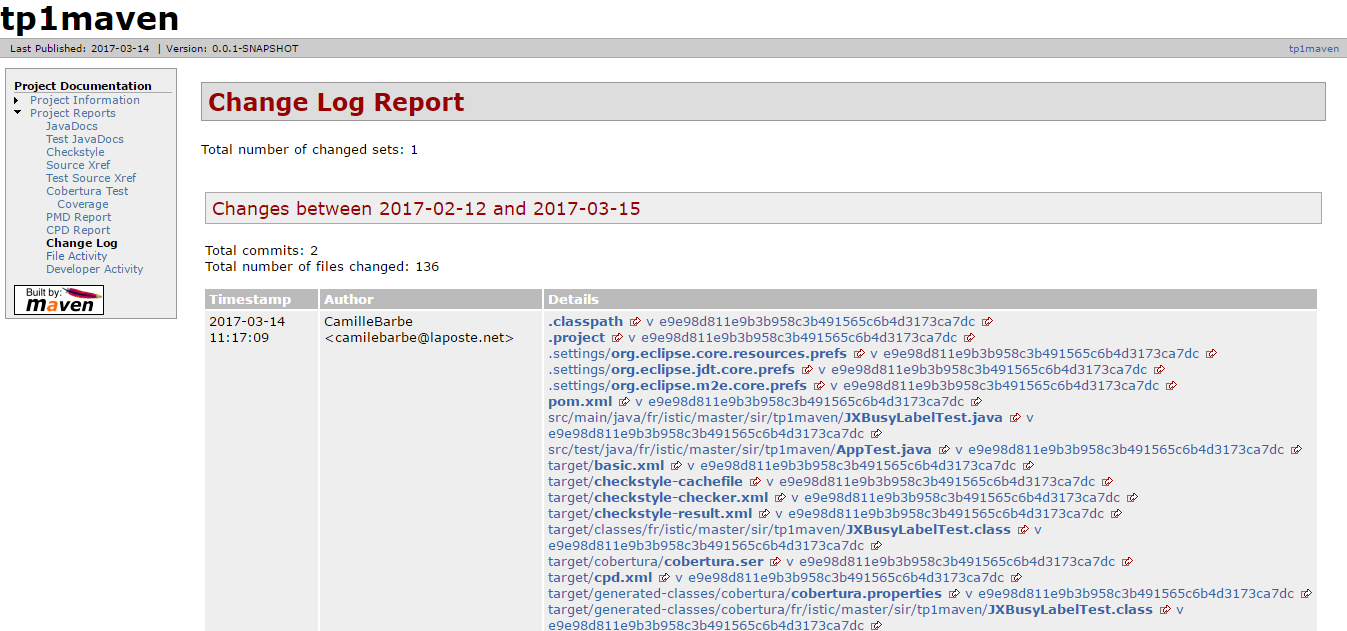
</project>

https://github.com/CamilleBarbe/maven.git

ipserveur doit être remplacé par l’IP de votre repo svn ou git.

●  Lancez : mvn site

○  Que s'est t'il passé ?



Le répertoire /target/site situé dans votre projet contient maintenant trois rapports d'activité :

•  changelog : rapport indiquant toutes les activités sur le SCM.

•  dev-activity : rapport indiquant par développeur le nombre de commits, de fichiers modifiés.

•  file-activity : rapport indiquant les fichiers qui ont été révisés.

Intégration, avec l'outil Sonar

Permet de contrôler la qualité du code produit :

Téléchargez Sonar : <http://sonar.codehaus.org/downloads>Installez Sonar : <http://sonar.codehaus.org/documentation/>

Lancez : mvn sonar:sonar

Télécharger SonarQube : <https://www.sonarqube.org/>

Modifier le sttings.xml de maven et ajouter :

<profile>

<id>sonar</id>

<activation>

<activeByDefault>true</activeByDefault>

</activation>

<properties>

<!-- Example for MySQL-->

<sonar.jdbc.url>

jdbc:mysql://localhost:3306/sonar?useUnicode=true&amp;characterEncoding=utf8

</sonar.jdbc.url>

<sonar.jdbc.username>sonar</sonar.jdbc.username>

<sonar.jdbc.password>sonar</sonar.jdbc.password>

<!-- Optional URL to server. Default value is http://localhost:9000 -->

<sonar.host.url>

http://localhost:9000

</sonar.host.url>

</properties>

</profile>

Ajouter dans le pom.xml :

<plugin>

<groupId>org.sonarsource.scanner.maven</groupId>

<artifactId>sonar-maven-plugin</artifactId>

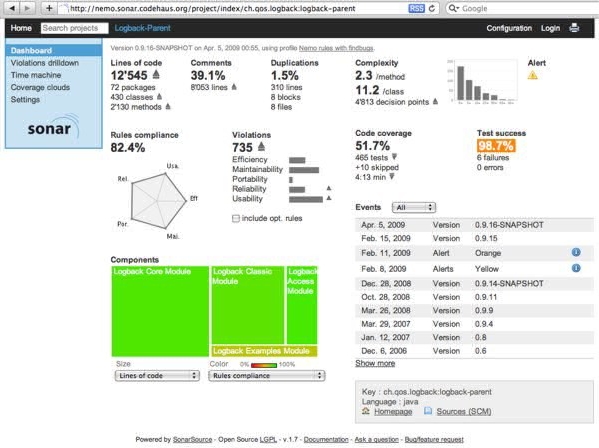
<version>3.2</version>

</plugin>

Exécuter le StartSonar.bat dans sonarqube-6.3\bin\windows-x86-64

Puis mvn sonar :sonar

Puis voir l’analyse : http://localhost:9000



Utilisation dans un contexte d’intégration continue

Nous allons ici observez les bénéfices pour l’utilisation d’un outil d’intégration continue Jenkins.

Commitez votre code sur github (public) ou bitbucket(privé)

**Partie 6 : Intégration Continue**

**Partie 6.1 : Gestion de version (Git)**

Placez votre code sur github ou gitlab.

**Partie 6.2 : Jenkins**

**Téléchargez Jenkins.**

L'intégration, si on doit le résumé en quelques mots, c'est : "voir les problèmes le plus rapidement possible".

**Jenkins** est un outil [open source](https://fr.wikipedia.org/wiki/Open_source) d'[intégration continue](https://fr.wikipedia.org/wiki/Int%C3%A9gration_continue), [fork](https://fr.wikipedia.org/wiki/Fork_(d%C3%A9veloppement_logiciel)) de l'outil [Hudson](https://fr.wikipedia.org/wiki/Hudson_(Informatique)) après les différends entre son auteur, Kohsuke Kawaguchi, et [Oracle](https://fr.wikipedia.org/wiki/Oracle_(entreprise)). Écrit en [Java](https://fr.wikipedia.org/wiki/Java_(langage)), Jenkins fonctionne dans un [conteneur de servlets](https://fr.wikipedia.org/wiki/Conteneur_de_servlets) tel qu’[Apache Tomcat](https://fr.wikipedia.org/wiki/Apache_Tomcat), ou en mode autonome avec son propre [serveur Web](https://fr.wikipedia.org/wiki/Serveur_Web) embarqué.

Il s'interface avec des systèmes de [gestion de versions](https://fr.wikipedia.org/wiki/Gestion_de_versions) tels que [CVS](https://fr.wikipedia.org/wiki/Concurrent_versions_system), [Git](https://fr.wikipedia.org/wiki/Git) et [Subversion](https://fr.wikipedia.org/wiki/Subversion_(logiciel)), et exécute des projets basés sur [Apache Ant](https://fr.wikipedia.org/wiki/Apache_Ant) et [Apache Maven](https://fr.wikipedia.org/wiki/Apache_Maven) aussi bien que des scripts arbitraires en [shell Unix](https://fr.wikipedia.org/wiki/Shell_Unix" \o "Shell Unix) ou [batch Windows](https://fr.wikipedia.org/wiki/.bat).

<http://jenkins-ci.org/>

Prenez la version **Java Web Archive (.war)**

Démarrez jenkins

**> java -jar jenkins.war --httpPort=9900**

Allez dans votre navigateur : <http://localhost:9900>.

**Configurez jenkins pour compiler et exécuter votre projet.**

Par défaut, jenkins ne contient pas le plugin pour gérer des repository Git, Il vous faut installer le plugin “Git Plugin”. De plus, vous devez configurer Maven (voir Configure System)

Ensuite créer un job, définissez les sources en indiquant l’url du repository git que vous avez préalablement créé sur github ou bitbucket et enfin définissez les goals maven pour le build.

La configuration d’un job peut prendre de nombreux paramètres. Par exemple, vous pouvez définir la condition de déclenchement d’un build (“Build Triggers”), vous pouvez limiter la sauvegarde des anciens builds (“Discard old builds”), vous pouvez aussi envoyer des notifications lorsque le build échoue (“E-mail notification” qui nécessite une configuration dans “Configure System”).

À ce point, vous devez avoir un jenkins sur lequel il y a un job défini. Lancer un Build pour vérifier que vous avez bien configuré votre projet.

Maintenant nous allons ajouter sur le jenkins, la possibilité d’utiliser sonar et cobertura ainsi que pmd.

<https://wiki.jenkins-ci.org/display/JENKINS/Sonar+plugin>

<https://wiki.jenkins-ci.org/display/JENKINS/Cobertura+Plugin>

<https://wiki.jenkins-ci.org/display/JENKINS/PMD+Plugin>

Attention pour Cobertura, vous avez besoin de définir le format de sortie en xml.

Pour cela, il existe deux solutions:

- la première consiste à ajouter une option dans la définition du build maven: “-Dcobertura.report.format=xml”

- la deuxième consiste à modifier la configuration dans votre pom et d’ajouter l’option de configuration  appropriée (voir sur la page de Cobertura plugin)

!!!! À la fin du TP, il est préférable que vous supprimiez le dossier ~/.hudson sauf si vous souhaitez conserver la configuration de votre jenkins

**Partie 7 : Créez votre propre MOJO**

Il y a des outils qui sont bien pratiques, voire parfois nécessaires à tout développeur qui se respecte. Un logiciel de gestion de build comme Ant ou Maven en fait partie. Outre l’avantage indéniable de permettre de télécharger la moitié de l’internet lors du premier build sur une machine *clean*, Maven permet aussi via son système de plugins d’effectuer tout un tas d’actions allant plus loin que la compilation des classes et leur packaging dans un JAR. Je vous propose de regarder comment créer son premier plugin Maven.

D’abord il faut un besoin

Eh oui, parce que se contenter de créer un plugin qui affiche “Hello World” à chaque build, ce n’est pas très utile… Alors tant qu’à faire, mettons nous en situation réelle. Prenons un plugin qui compte le nombre de classes et d’attributs/méthodes par classe dans notre code.

Création du squelette de plugin Maven

En ligne de commande.

mvn archetype:generate  -DgroupId=fr.sir.tpmaven -DartifactId=classcounter-maven-plugin  -DarchetypeArtifactId=maven-archetype-mojo

Puis import dans eclipse

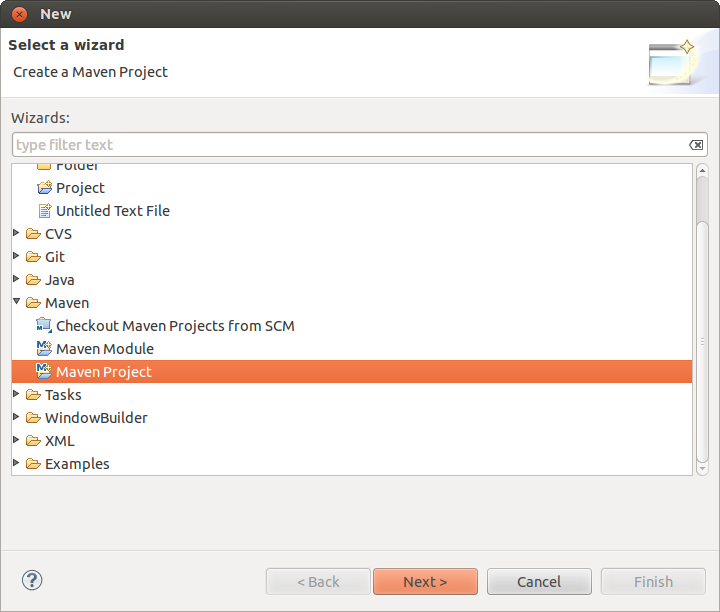
File -> import -> maven -> import existing maven project -> select the generatedproject.

Sous Eclipse

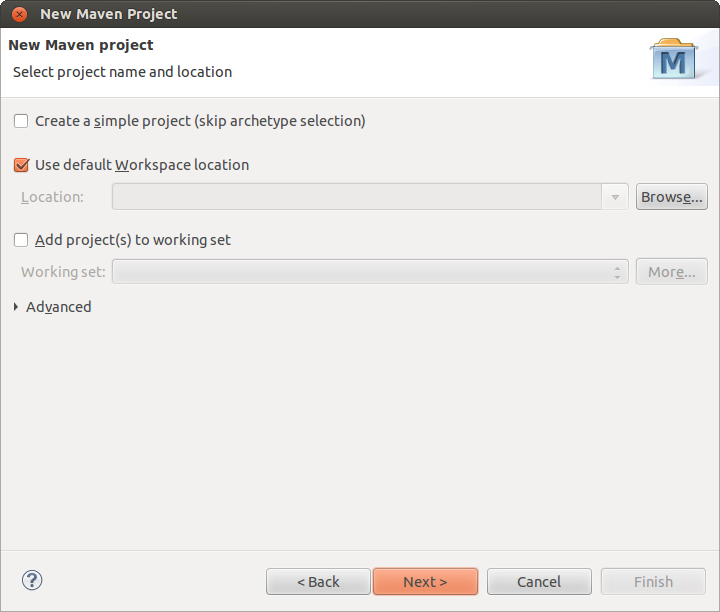
Un plugin Maven n’est rien d’autre qu’un projet Maven avec un *packaging* de type **maven-plugin** :

**Sous eclipse 4.X**

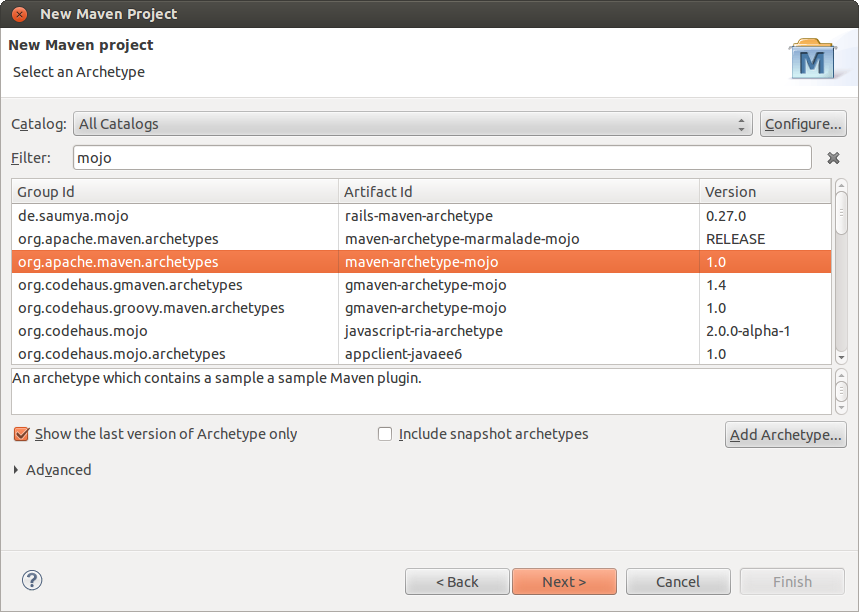
Créez un projet Maven.



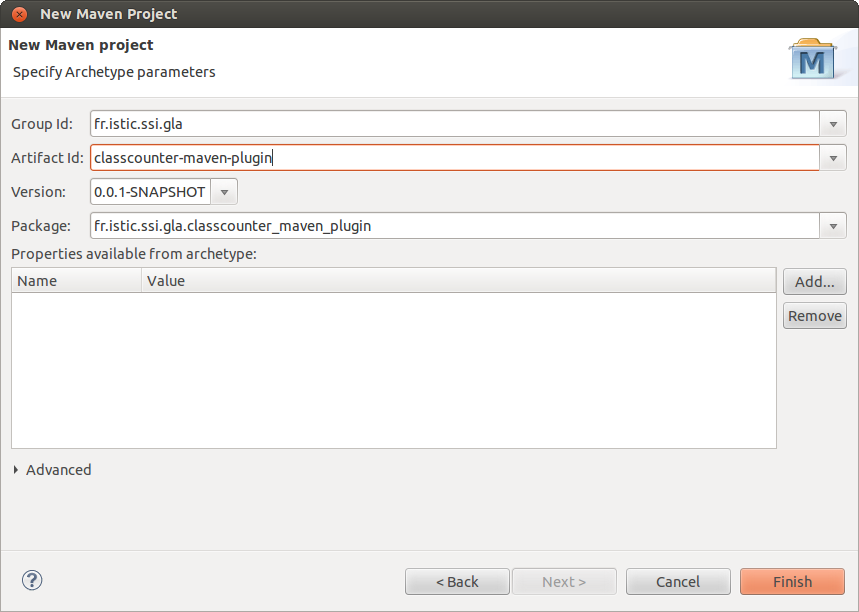
à l’écran suivant, vous pouvez laisser les options par défaut.



Ensuite choisissez le mojo du projet (template de création du projet). Pour créez un projet de plugin pour maven il faut selectionnez l’archetype org.apache.maven.archetypes:maven.archetype.mojo



Ensuite renseignez le nom du projet (group id et artifact id).



**Pour eclipse 4.X**

La convention veut que les plugins soient nommés *xyz-maven-plugin*, ce qui permet d’appeler les goals de la manière suivante lors d’un build :

|  |
| --- |
| $ mvn classcounter:some-goal |

Notez qu’il n’est pas nécessaire de spécifier le nom complet du plugin, Maven va compléter avec “-maven-plugin”, ça fait quelques caractères en moins à écrire https://lh3.googleusercontent.com/xRzOPEiSVuVmrL_KYgw01aSyfUecPfw2ZPudIvA8W0rYHZ6sLxIsoL-utkzT6XQnRb4KXtYDQI4pIXfw5-GIGhYbKznZuUrjanL1WFKct7Yp3xFHExg6diHcPtiZI-G7sBDUBoY . Si la convention de nommage n’est pas suivie, il faut impérativement spécifier le nom complet du plugin dans la ligne de commande.

Le projet du MOJO contient le POM suivant.

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">      <groupId>fr.istic.master1.sir</groupId>     <artifactId>classcounter-maven-plugin</artifactId>     <version>1.0-SNAPSHOT</version>     <packaging>maven-plugin</packaging> </project> |

Ensuite, un plugin contient un ou plusieurs MOJO, qui vont exécuter le “vrai” code du plugin. Un MOJO correspond en fait à un *goal*. Par exemple, dans le plugin dependency-maven-plugin, on retrouve les goals “[tree](http://svn.apache.org/viewvc/maven/plugins/tags/maven-dependency-plugin-2.4/src/main/java/org/apache/maven/plugin/dependency/TreeMojo.java?view=markup)” (mvn dependency:tree) et “[resolve](http://svn.apache.org/viewvc/maven/plugins/tags/maven-dependency-plugin-2.4/src/main/java/org/apache/maven/plugin/dependency/resolvers/ResolveDependenciesMojo.java?view=markup)” (mvn dependency:resolve)

Créons donc un squelette de MOJO pour notre générateur de code :

|  |
| --- |
| package *fr.istic.master1.sir.plugin.classcounter*;  import *org.apache.maven.plugin.AbstractMojo*; import *org.apache.maven.plugin.MojoExecutionException*; import *org.apache.maven.plugin.MojoFailureException*;  */\*\** *\* A goal to generate code.* *\** *\* @goal count* *\* @phase compile* *\*/* public class ClassCounterMojo extends AbstractMojo {     public void execute() throws MojoExecutionException, MojoFailureException {         getLog().info("Hello World!");     } } |

Notre MOJO étend AbstractMojo, qui fournit toute l’infrastructure utile à la création d’un goal. La méthode abstraite execute() contient le code à exécuter, c’est donc à nous de l’implémenter. Pour l’instant elle ajoute une ligne de log saluant le monde (je sais, j’ai dit tout à l’heure qu’on ne se contenterait pas de faire un hello world…). La méthodegetLog() est un exemple de facilités fournies par l’AbstractMojo.

Notez également la Javadoc, qui contient deux tags particuliers. @goal spécifie le nom du goal correspondant à notre MOJO, ce tag est obligatoire. @phase permet de dire à Maven que notre plugin intervient durant la phase de génération des sources du [lifecycle Maven](http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html)(avant la compilation des sources, donc).

Pour utiliser notre plugin depuis un autre projet, il faut d’abord l’installer dans notre dépôt local :

$ mvn install

Notre plugin est prêt à être exécuté. Configurons le pom.xml de notre projet *tpmaven* pour permettre l’exécution du plugin lors du build :

|  |
| --- |
| <build>         <plugins>             <plugin>                 <groupId>fr.istic.master1.sir</groupId>                 <artifactId>classcounter-maven-plugin</artifactId>                 <version>1.0-SNAPSHOT</version>             </plugin>         </plugins>     </build> |

Puis lançons un build Maven :

|  |
| --- |
| $ mvn classcounter:count [INFO] Scanning for projects... [INFO]                                                                          [INFO] ------------------------------------------------------------------------ [INFO] Building classcounter-maven-plugin 1.0-SNAPSHOT [INFO] ------------------------------------------------------------------------ [INFO]  [INFO] --- classcounter-maven-plugin:1.0-SNAPSHOT:count (default-cli) @ classcounter-maven-plugin --- [INFO] Hello World! [INFO] ------------------------------------------------------------------------ [INFO] BUILD SUCCESS [INFO] ------------------------------------------------------------------------ [INFO] Total time: 0.322s [INFO] Finished at: Mon May 14 14:47:54 CEST 2014 [INFO] Final Memory: 2M/81M [INFO] ------------------------------------------------------------------------ |

Ajout de paramètres au plugin

Notre cahier des charges requiert la possibilité de configurer certaines parties du générateur. Pour cela, ajoutons des paramètres à notre MOJO:

|  |
| --- |
| public class ClassCounterMojo extends AbstractMojo {            /\*\*  \* Location of the file.  \* **@parameter** expression="${project.build.directory}"  \* **@required**  \*/  **private** File outputDirectory;  /\*\*  \* Message language  \* **@parameter** default-value="french"  \* **@required**  \*/  **private** String language;              …... } |

Notre attribut utilise également des tags Javadoc pour indiquer que c’est un paramètre du plugin. Ce paramètre est obligatoire et prend une valeur par défaut.

L’attribut “alias” de @parameter permet de définir le nom du paramètre s’il est différent du nom de l’attribut Java. Ainsi, par défaut il y aura un paramètre nommé “language”, puisqu’on ne lui a pas donné d’alias. Une liste plus exhaustive des tags Javadoc est disponible sur le [site de Sonatype](http://www.sonatype.com/books/mvnref-book/reference/writing-plugins-sect-mojo-params.html).

Ouvrons à nouveau le pom.xml de notre projet pour ajouter la configuration du plugin :

|  |
| --- |
| <build>         <plugins>             <plugin>                 <groupId>fr.istic.master1.sir</groupId>                 <artifactId>classcounter-maven-plugin</artifactId>                 <version>1.0-SNAPSHOT</version>                 <configuration>                     <language>english</language>                  </configuration>             </plugin>         </plugins>     </build> |

Pour vérifier que la configuration est bien injectée au runtime, ajoutons un log :

|  |
| --- |
| getLog().info("Indication will be given in the following language " + language); |
| mbp:GeneratorMavenPlugin bastien$ mvn classcounter:count ... [INFO] --- classcounter-maven-plugin:1.0-SNAPSHOT:count (default-cli) @ classcounter-maven-plugin --- [INFO] Hello World! [INFO] Indication will be given in the following language English  [INFO] ------------------------------------------------------------------------ [INFO] BUILD SUCCESS ... |

Maintenant que nous savons créer, configurer et exécuter un plugin Maven, il ne reste plus qu’à implémenter la méthode execute() pour appeler les vrais services.

XDoclet c’est mal !!

Comme vous l’avez vu, la configuration du Mojo se fait par défaut dans des tags Javadoc (des sortes de [XDoclet](http://xdoclet.sourceforge.net/xdoclet/index.html)). Ce n’est pas terrible dans la mesure où il est facile de faire une faute de frappe (ex @paramter au lieu de @parameter) qui ne sera pas détectée par le compilateur ni par Maven. Pour pallier ce problème, depuis maven 3, il existe un *set* d’annotations Java 5 (<https://maven.apache.org/plugin-tools/maven-plugin-plugin/examples/using-annotations.html#POM_configuration>). Après avoir mis à jour les dépendances de votre pom, il est possible de tout annoter :

|  |
| --- |
| <dependencies>   <dependency>     <groupId>org.apache.maven</groupId>     <artifactId>maven-plugin-api</artifactId>     <version>3.0</version>    </dependency>    <dependency>     <groupId>org.apache.maven.plugin-tools</groupId>     <artifactId>maven-plugin-annotations</artifactId>     <version>3.4</version>     <scope>provided</scope><!-- annotations are needed only to build the plugin -->   </dependency>   <dependency>  <groupId>org.apache.maven</groupId>  <artifactId>maven-core</artifactId>  <version>3.3.9</version>         <scope>provided</scope><!-- annotations are needed only to build the plugin -->   </dependency>  </dependencies>  <build>   <plugins>     <plugin>         <groupId>org.apache.maven.plugins</groupId>         <artifactId>maven-plugin-plugin</artifactId>         <version>3.4</version>         <executions>           <execution>             <id>default-descriptor</id>             <phase>process-classes</phase>           </execution>         </executions>       </plugin>     </plugins>  </build> |
| **package** fr.istic.sir.classcounter\_maven\_plugin;  **import** org.apache.maven.plugin.AbstractMojo;  **import** org.apache.maven.plugin.MojoExecutionException;  **import** org.apache.maven.plugins.annotations.Execute;  **import** org.apache.maven.plugins.annotations.LifecyclePhase;  **import** org.apache.maven.plugins.annotations.Mojo;  **import** org.apache.maven.plugins.annotations.Parameter;  @Mojo( name = "count")  @Execute( goal = "count",  phase = LifecyclePhase.***COMPILE***)  **public** **class** ClassCounterMojo  **extends** AbstractMojo  {  @Parameter(defaultValue="french")  String language;  **public** **void** execute() **throws** MojoExecutionException  {  getLog().info("Hello World! " + language);  }  } |

Passons au travail de compter les classes, les attributs par classes et les opérations par classes.

Pour cela nous allons utiliser la librairie clapper. Ajoutons clapper en dépendance à notre projet classcounter.

     <dependency>

           <groupId>org.clapper</groupId>

           <artifactId>classutil\_2.9.2</artifactId>

           <version>0.4.6</version>

       </dependency>

Exemple de code pour la méthode *execute* du Mojo de notre plugin Maven.

|  |
| --- |
| **package** fr.istic.sir.classcounter\_maven\_plugin;  **import** java.io.File;  **import** java.util.ArrayList;  **import** java.util.List;  **import** org.apache.maven.plugin.AbstractMojo;  **import** org.apache.maven.plugin.MojoExecutionException;  **import** org.apache.maven.plugins.annotations.Execute;  **import** org.apache.maven.plugins.annotations.LifecyclePhase;  **import** org.apache.maven.plugins.annotations.Mojo;  **import** org.apache.maven.plugins.annotations.Parameter;  **import** org.clapper.classutil.ClassFinder;  **import** org.clapper.classutil.ClassInfo;  @Mojo( name = "count")  @Execute( goal = "count",  phase = LifecyclePhase.***COMPILE***)  **public** **class** ClassCounterMojo  **extends** AbstractMojo  {  @Parameter(defaultValue="french")  String language;    @Parameter( defaultValue = "${project.build.directory}", readonly = **true** )  **private** File outputDirectory;  **public** **void** execute()  **throws** MojoExecutionException  {  File f = outputDirectory;  File f1 = **new** File(outputDirectory.getAbsolutePath(),"classes");  List<File> files = **new** ArrayList<File>();  files.add(f1);  ClassFinder finder = **new** ClassFinder(scala.collection.JavaConversions.*asScalaBuffer*(files));  **if** ("french".equals(language))  **this**.getLog().info("nombre de classe " + finder.getClasses().size());  **else**  **this**.getLog().info("number of classe " + finder.getClasses().size());  scala.collection.Iterator<ClassInfo> it = finder.getClasses();  **while** (it.hasNext()){  ClassInfo c = it.next();  **if** ("french".equals(language))  {  **this**.getLog().info("\t Pour la classe " + c.name());  **this**.getLog().info("\t \t Nbre attributs " + c.fields().size());  **this**.getLog().info("\t \t Nbre methodes " + c.methods().size());  }**else**{  **this**.getLog().info("\t For the class named " + c.name());  **this**.getLog().info("\t \t Number of filed " + c.fields().size());  **this**.getLog().info("\t \t Number of methods " + c.methods().size());  }  } }  } |

Comme vous avez pu le voir, créer un plugin Maven n’est pas très compliqué. Les plugins sont suffisamment souples et configurables pour pouvoir exécuter du code déjà existant (par exemple remplacer un main() par un plugin Maven) sans trop de difficultés. La palette de plugins déjà disponibles est assez vaste, mais si un jour vous avez un besoin spécifique vous saurez que le coût de création d’un plugin n’est pas très élevé.

Enfin, il existe plein de paramètre au niveau des annotations. Voici un exemple de Mojo plus riche.

**package** fr.istic.sir.classcounter\_maven\_plugin;

**import** java.io.File;

**import** org.apache.maven.execution.MavenSession;

**import** org.apache.maven.plugin.AbstractMojo;

**import** org.apache.maven.plugin.MojoExecution;

**import** org.apache.maven.plugin.descriptor.PluginDescriptor;

**import** org.apache.maven.plugins.annotations.Execute;

**import** org.apache.maven.plugins.annotations.InstantiationStrategy;

**import** org.apache.maven.plugins.annotations.LifecyclePhase;

**import** org.apache.maven.plugins.annotations.Mojo;

**import** org.apache.maven.plugins.annotations.Parameter;

**import** org.apache.maven.plugins.annotations.ResolutionScope;

**import** org.apache.maven.project.MavenProject;

**import** org.apache.maven.settings.Settings;

@Mojo( name = "MyMojo",

aggregator = **true**,

executionStrategy = "always",

inheritByDefault = **true**,

instantiationStrategy = InstantiationStrategy.***SINGLETON***,

defaultPhase = LifecyclePhase.***COMPILE***,

requiresDependencyResolution = ResolutionScope.***NONE***,

requiresDependencyCollection = ResolutionScope.***NONE***, // (since Maven 3.0)

requiresDirectInvocation = **false**,

requiresOnline = **false**,

requiresProject = **true**,

threadSafe = **true** ) // (since Maven 3.0)

@Execute( goal = "test",

phase = LifecyclePhase.***COMPILE***)

**public** **class** MyMojo

**extends** AbstractMojo

{

@Parameter( name = "parameter",

alias = "myAlias",

property = "a.property",

defaultValue = "an expression, possibly with ${variables}",

readonly = **false**,

required = **false** )

**private** String parameter;

// sample objects taken from Maven API through PluginParameterExpressionEvaluator

@Parameter( defaultValue = "${session}", readonly = **true** )

**private** MavenSession session;

@Parameter( defaultValue = "${project}", readonly = **true** )

**private** MavenProject project;

@Parameter( defaultValue = "${mojoExecution}", readonly = **true** )

**private** MojoExecution mojo;

@Parameter( defaultValue = "${plugin}", readonly = **true** ) // Maven 3 only

**private** PluginDescriptor plugin;

@Parameter( defaultValue = "${settings}", readonly = **true** )

**private** Settings settings;

@Parameter( defaultValue = "${project.basedir}", readonly = **true** )

**private** File basedir;

@Parameter( defaultValue = "${project.build.directory}", readonly = **true** )

**private** File target;

**public** **void** execute()

{

}

}

[1] <http://labs.excilys.com/2012/05/14/mon-premier-plugin-maven/>

[2] <http://www.objis.com/formation-java/IMG/pdf/TP8_reporting.pdf>