**Cours du 15 janvier**

**Qualite & Qualimétrie .**

Professeur : Sébastien

Astuce :  
- ALT + flèche dans eclipse, pour bouger une ligne dans la direction.  
- ne pas mettre de PAUSE dans un .bat qu’on execute automatiquement par programme.   
- CTRL + D : efface (delete) dans Eclipse  
- ALT + SHIFT + A : selection par colonne  
- ALT + SHIFT + Z : surround try / catch

Mesure de la qualité 🡪 code

Au niveau de l’indentation :  
  
En Java 🡪 4 espaces  
En web (HTML/JS/CSS) : 2 espaces

Pas de tab !!

Au niveau du nommage (en Java) :  
classes 🡪 PascalCase  
variables & méthodes : camelCase  
pacckages : domaine.societe.projet..  
propriétés : nom 🡪 getUnePropriete et setUnePropriete

Nombre de lignes/méthode : doit se voir à l’écran  
Nombre de caractères par ligne.

Nombre de paramètres limités

Complexité cyclomatique

Documentation : obligatoire sur la classe, les packages, les variables, les méthodes.

[Javadoc](https://openclassrooms.com/courses/presentation-de-la-javadoc) 🡪 elle peut être exportée et diffusée, en html et en jar.

Pour vérifier que l’on a bien écrit le code, on branche checkStyle qui vérifie au fur et à mesure de l’écriture du code le respect des conventions de nommage.

Qualité logicielle :  
Tests :  
Côté développeur :  
Un test unitaire : test une méthode **sans ses dépendances**  
Test d’intégration : teste une méthode **avec ses dépendances**

Côté MOA :  
Tests fonctionnels : IHM & bdd.  
Test d’intégration : déploiement & environnement

Couverture de test : elle doit être au minimum 80%

En Java, pour les tests on utilise **JUnit**.  
Pour tester les IHM web : tests avec **Slenenium** qui marche par scripts et peut être écrit en java.  
Utilisation aussi de **Sahi** (moins utilisé)  
Pour les bases de données : **DBUnit**

SCM : source control management pour le **versioning**

On a Git, SVN…

Qui est basé sur les tests ? 🡪 MAVEN  
Quand on fait de l’AGILE, on se base sur les tests. On part d’un CDC incomplet ; on doit pouvoir tout casser pour faire face à de nouveaux besoins qu’on ne connaissait pas, et modifier / ajouter des fonctionnalités.  
On doit donc faire des tests, et se baser dans notre démarche sur ces derniers.

MAVEN : Apache Maven  
construction d’un projet 🡪 cycle de build  
Différentes phases, avant de packager Maven a une phase de tests.  
On doit faire les tests, première étape. Sinon blocage par Maven.

Complie 🡪 Tests 🡪 package 🡪 install 🡪 deploy.  
Si une étape foire, maven stop, le deploy échoue.

Le package est ici la création des Jarchives (.jar), les Warchives (.war), le EArchive (.ear)

Deux choses seulement peuvent être déployées avec Java : projets web, et EJB (Entreprise JavaBean)

Toute archive Java contient **obligatoirement** un **dossier META-INF comprenant un fichier MANIFESTE.MF** qui définit l’auteur, la version, le MainClass, le ClassPath.

Ce fichier est généré automatiquement, il est généra par Java ou avec son utilisation, par Maven.

En fait, Maven sert à construire un projet.  
Avant Maven, l’outil qui le faisait très bien, c’est ANT.  
Maven gère aussi les dépendances d’un projet (bibliothèques de classes).

Ant ne gérait juste pas les dépendances, en comparaison avec Maven.  
Maven gère la construction **et** les dépendances.

Projet : retrouve les sources.  
Maven inclus les dépendances au moment du packaging de l’application.  
Pas besoin de les télécharger, il va télécharger les dépendances pour nous.

Pour tester le projet dans sa globalité, et pas juste ce que crée un dév, on a des outils d’intégration continue (Maven).  
On fait des Maven install sur la globalité d’un projet.  
Maven crée un reporting (« hier à 14h, ça compilait »).

Jenkins | Hudson  
  
Jenkins : gère un cycle de déploiement complet, et le reporting qualité.  
Il peut injecter tout ce qui est métrique et qualité, dans SonarCube.

SonarCube : reporting de qualité.

Le chef de projet, ou l’architecte responsable du projet s’assure de la qualité du code pour assurer qu’il est toujours propre, et s’assurer de l’intégration continue du projet.

Repository d’artifact : Nexus et Artifactory.  
Ce sont des serveurs web dans lesquels on dépose plein de fichiers, et l’on peut ensuite faire des get dessus au besoin.  
Dans Nexus, juste des artifact maven.  
Sert juste à diffusion du projet.

Il reste une grosse étape avant d’arriver au testeur.

Pour tester l’IHM, le testeur a besoin d’un OS : dans ce système d’exploitation, on a besoin d’un serveur d’application qui doit pouvoir aller chercher la config entre autre. Ainsi, il peut tester les IHM.  
Il peut aussi avoir besoin d’une bdd ; pour tester la BDD, le testeur a besoin de scripts

Testeur peut tester le scripting des IHM.

L’OS ne gère pas le bash.  
Pour gérer les OS, il faut deux choses : pour automatiser l’install d’un OS, il faut des images.  
**Virtualisation** ou **containering** (depuis 64 bits, Linux et Windows 7).  
Le containering, c’est un OS qui tourne dans un OS de manière totalement autonome ; peut exploiter directement toutes les ressources matérielles sans passer par l’OS de base.  
Virtualisation : virtual box, VMWare.

Le containering a toujours existé sous Lunix. Sous Windows 10, on a un unbuntu qui fonctionne justement grace a du containering !  
HyperV est le composant Windows qui le permet.  
Docker

Le développeur peut écrire son code, et finalement tout derrière est bien géré pour que les testeurs MOA puissent direct faire leurs tests.  
  
Le testeur utilise Jenkins pour lancer des scripts Docker, pour gérer le déploiement d’un serveur, d’un OS, etc.

DEVOPS connait le dév de l’application, l’intégration continue, le déploiement continue,etc.

De notre côté, on va se concentrer sur les outils JUnit et Maven, mise en place donc du checkstyle, versionning..  
En AL, on apprendra à faire de l’intégration continue côté développement, et Docker Jenkins … En Dévops

Maven est toujours organisé ainsi :  
Projet 🡪 sources src 🡪 main & tests

Dans le main, les java à complier, et un dossier ressources pour les classes à ne pas compiler.  
Idem pour les tests : deux dossiers, java et ressources

Fichier Target aussi, dans le maven.  
Et pom.xml : Project Object Model qui décrit le projet.

Une identité pour le projet : elle est définie par trois choses :  
- un groupId  
- un artifactId (Nom)  
- version

Maven cherche automatiquement le projet dans un dossier : fr.afcepf.ai102.  
L’artifactId permet de chercher un nom de projet : UnProjet  
La version : dossier 1.0, 1.1…

Lorsqu’on a besoin de dépendances, l’arborescence permet a Maven de bien rechercher.

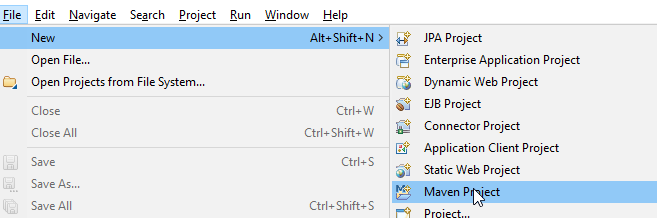
Plusieurs packaging :  
- jar (lib | executable)  
- war  
- ear  
… ejb  
  
Système de dépendances, pour dire quelles sont celles du projet.  
Maven les lit.  
Une dépendance de manière unitaire est définie toujours de la même façon, on a besoin de :  
- son groupId  
- son artifactId  
- sa version  
Maven sait ensuite facilement la télécharger.  
  
Une dépendance peut être utilisée dans différents buts, et de différentes façons.  
Elle peut être nécessaire à la compilation : scope compile  
Elle peut être utilisée uniquement pour les tests : scope test  
Scope provided : utilisée pour la compilation, mais pas pour l’exécution (la bibliothèque de classe est fournie par l’IDE).  
Dépendance system.

Système de propriétés :  
-maven.compiler.source  
-maven.compiler.target  
-project.build.sourceEncoding : UTF-8

Quand une classe java est compilée, elle l’est avec son **numéro de version**, sinon la compilation est impossible.

Lorsqu’on importe un Maven, le projet Maven contient avant tout un src, et un pom. Pour tout ce qui est dépendances, Maven les télécharge uniquement quand elle en a besoin.

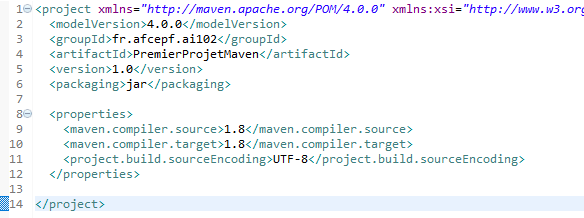
On crée toujours un projet Maven en UTF-8, sur java 1.8



On create toujours un simple project



Ecriture à la mano des propriétés



Du coup on va toujours écrire ça :

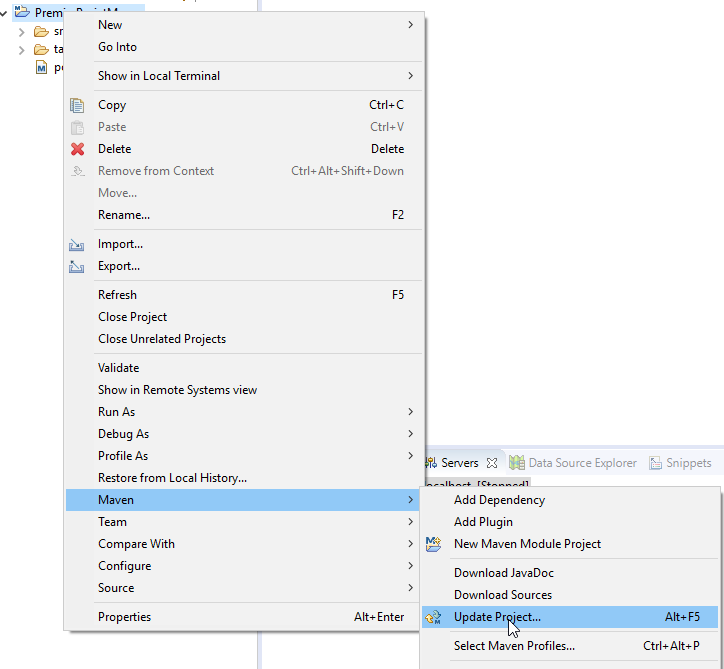
<properties>

<maven.compiler.source>1.8</maven.compiler.source>

<maven.compiler.target>1.8</maven.compiler.target>

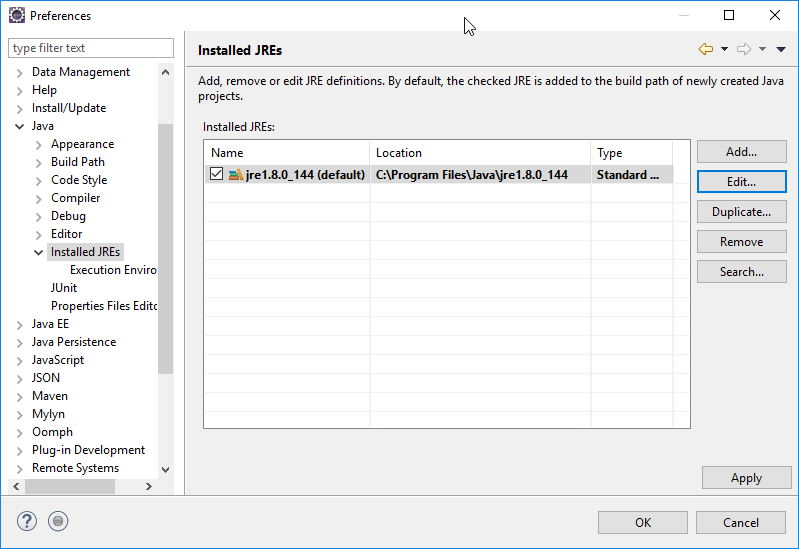
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

Clic droit et :  


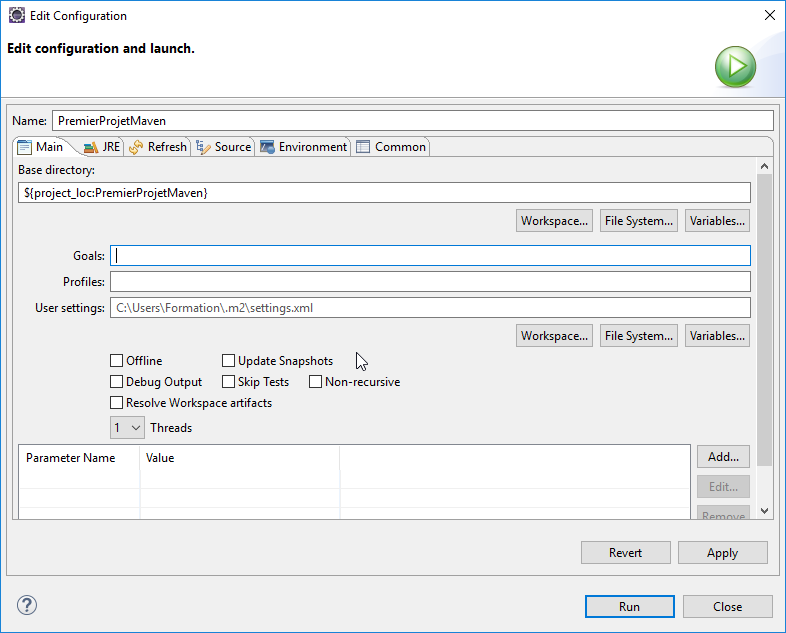
On le fait souvent, pour MAJ du pom (ALT + F5 donc).

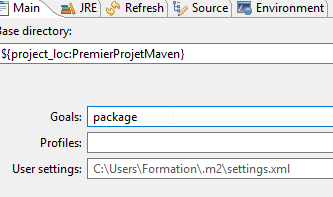
Mvn 🡪 compile, install (il passe par compile, test, package PUIS si tout a marché l’install).

Dans les préférences, on doit éditer dans Installed JREs :  
On paramètre le jdk :  
  


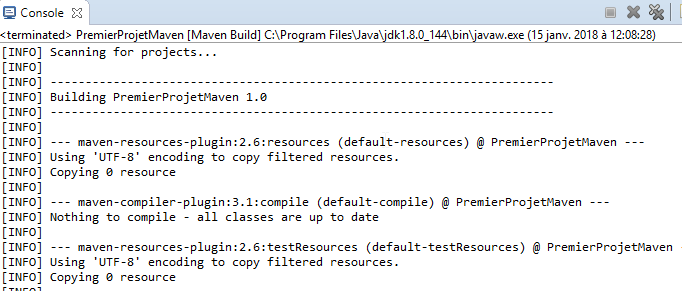
On édite en mettant le jdk (et pas jre)  


En lançant avec **run as** 🡪 **Maven Built**



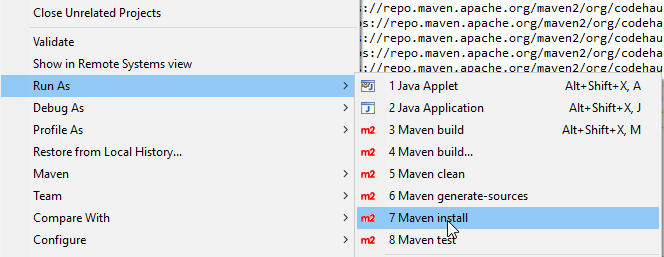


Et hop, ça télécharge plein de trucs utiles.



Automatiquement maven va chercher en local qund il a besoin d’un fichier,  
Sinon il télécharge puis il stocke en local (au fur et à mesure du temps, il a donc de moins en moins besoin de dL).

Maven install :



Lorsqu’on a un soucis avec maven, on va chercher sur le net :  
maven connector mysql  
On tombe toujours sur le site maven repository : <https://mvnrepository.com/>

On va ainsi aller chercher une install pour les logs.

Cours de JMDoudoux sur le LOGGIN : [clic](https://www.jmdoudoux.fr/java/dej/chap-logging.htm)  
  
Le login (le journal de bord, quoi) :  
Plusieurs niveaux : level 🡪 debug, info, fatar, warm …  
log.info(’’…’’)  
log.debug(’’…’’) ..  
  
On ne fait pas de system.out(’’…’’) !!

Permet d’avoir une trace complète de ce qu’il s’est passé.  
On l’utilise toujours dans les **catch** pour remonter les infos, entre autre.

En java, celui qui marche très bien est **log4j**.

On va donc aller chercher log4j.

On rajoute à la mano les lignes suivantes au pom :

<dependencies>

<dependency>

<groupId>Log4j</groupId>

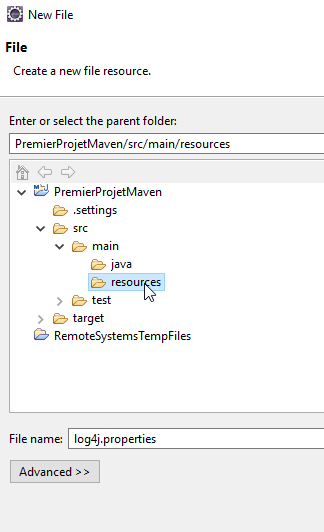
<artifactId>log4j</artifactId>

<version>1.2.16</version>

<scope>compile</scope>

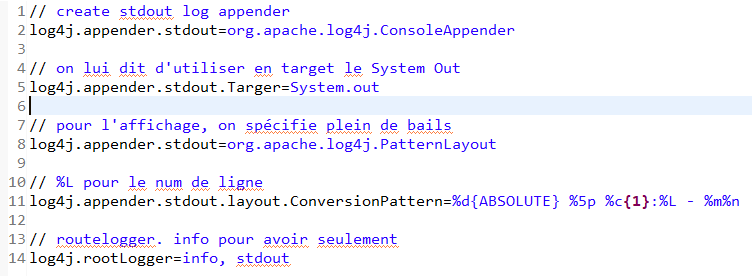
</dependency>

</dependencies>



On a ainsi créé un nouveau fichier, dans le src 🡪 resources.  
Quand on aura ce fichier, on l’utilisera à chaque fois, plus besoin de le réécrire.

On y inscrit :



// create stdout log appender

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

// on lui dit d'utiliser en target le System Out

log4j.appender.stdout.Targer=System.out

// pour l'affichage, on spécifie plein de bails

log4j.appender.stdout=org.apache.log4j.PatternLayout

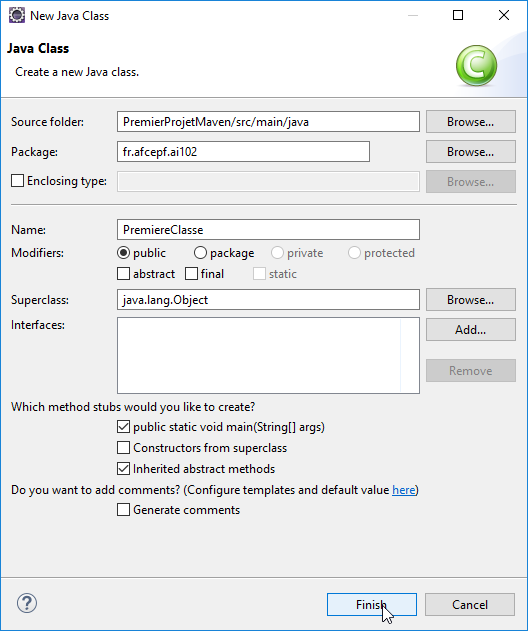
// %L pour le num de ligne

log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c**{1}**:%L - %m%n

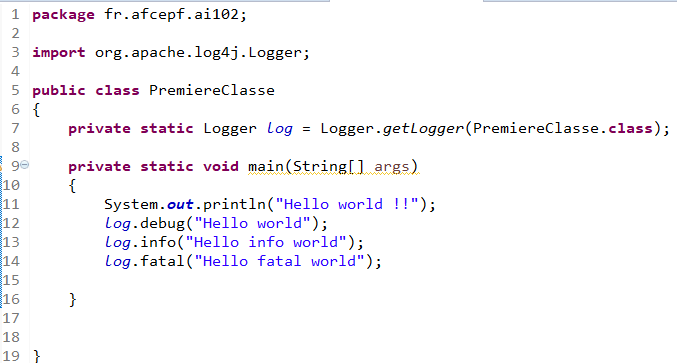
// routelogger. info pour avoir seulement

log4j.rootLogger=info, stdout

Création d’une classe (avec nouveau package à nommer)

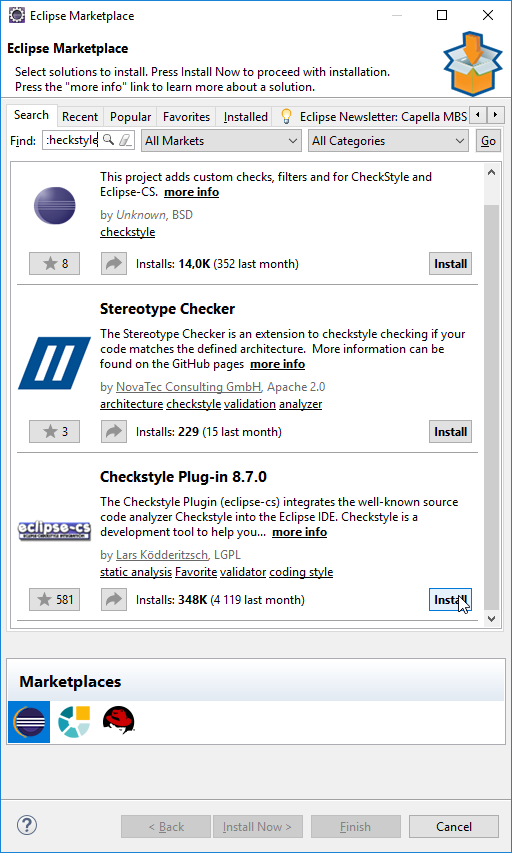


On y écrit ça :

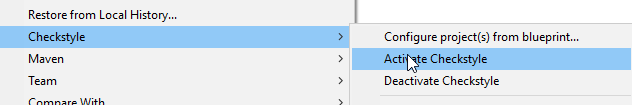


On a ainsi configuré le « nouveau syso », le loggin.

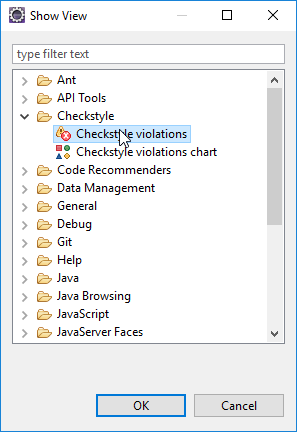
Maintenant on install le checkStyle :  
Onglet Help, Marketplace on choisit ce plugin :

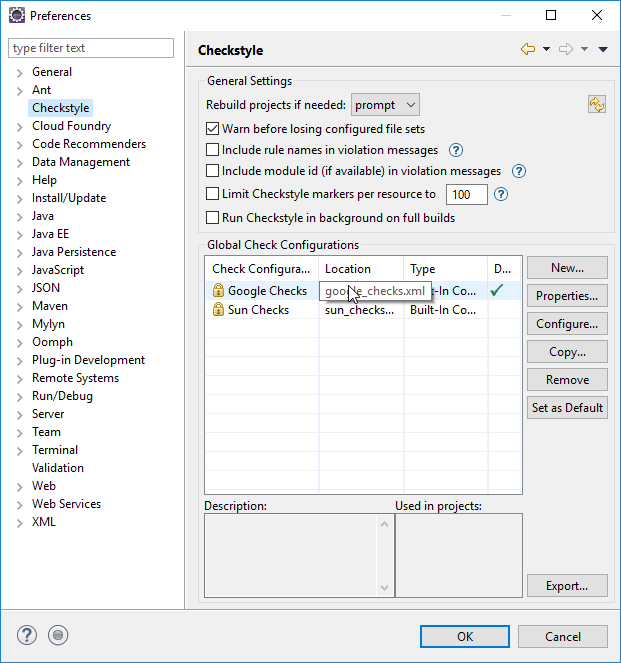


On crée désormais une classe tout bête, avec :  
- deux variables (i int et s string)  
- deux constructeurs un vide un plein  
- get/set  
- deux méthodes : 0 args et retoune une string, une avec 1 arg retourne un int.

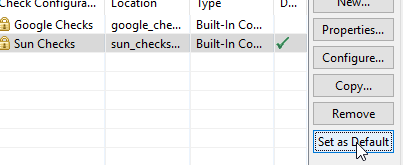


Ajout de la windows



Par défaut, le compte Google est ajouté (car ils font beaucoup de dév web)  


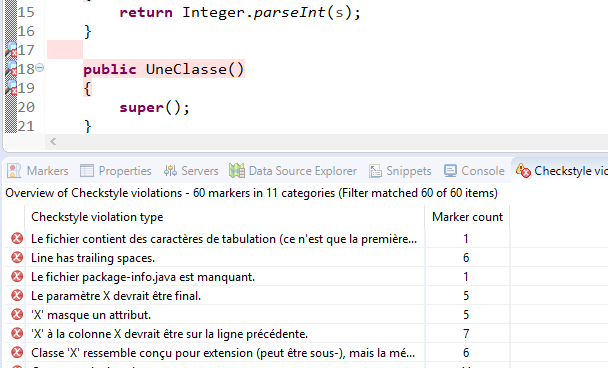
On set as default le compte Sun



Ensuite, on fait checkstyle suite à clic droit, pour lancer le check :



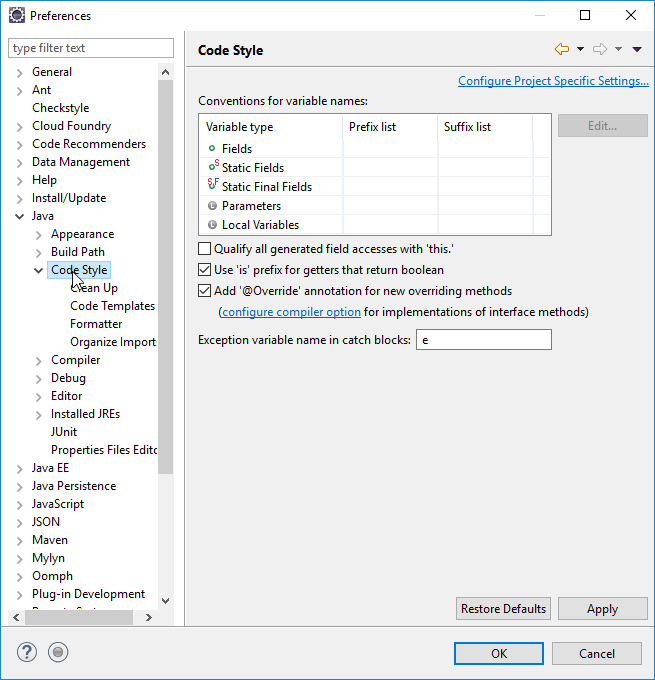
On arrive a un maaaax d’erreurs, donc ça compilera pas !!



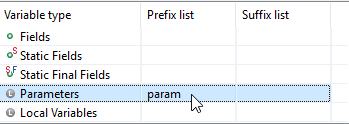
On va devoir paramétrer entre autre :  
- le fait qu’on ne fait pas une tabulation, mais bien 4 espaces pour indentation.  
- finalParameter : vient de checkStyle, c’est là qu’on configure ça.  
- X masque un attribut : c’est dans Eclipse à propos des « this », car on est censé être précis, et le this ne l’est pas)  
- « la classe n’est pas prévue pour être dérivée » : vient de checkStyle

Wikipédia : bom.

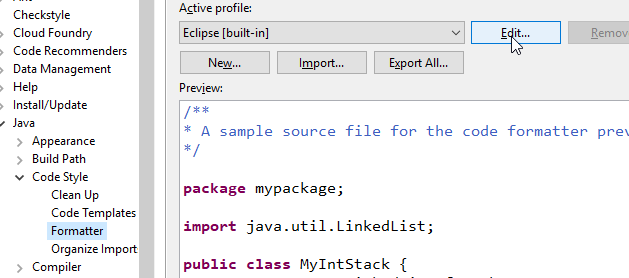
On change les paramtères pour adéquation :



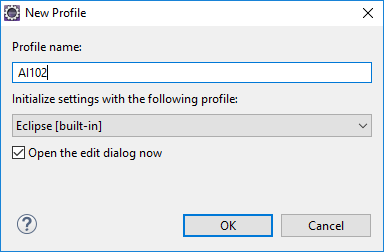
Ajout de param :



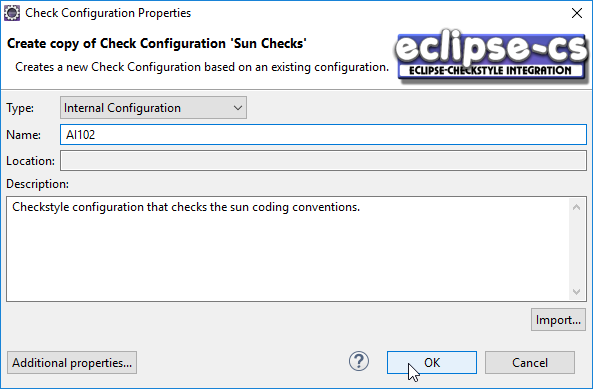
Ensuite on a ici EDIT :



Nouveau profil :



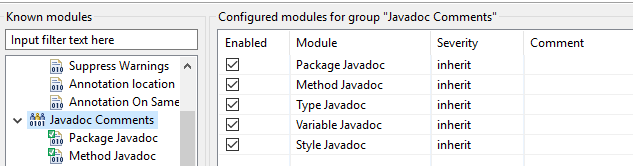
Paramétrage d’un nouveau profil :



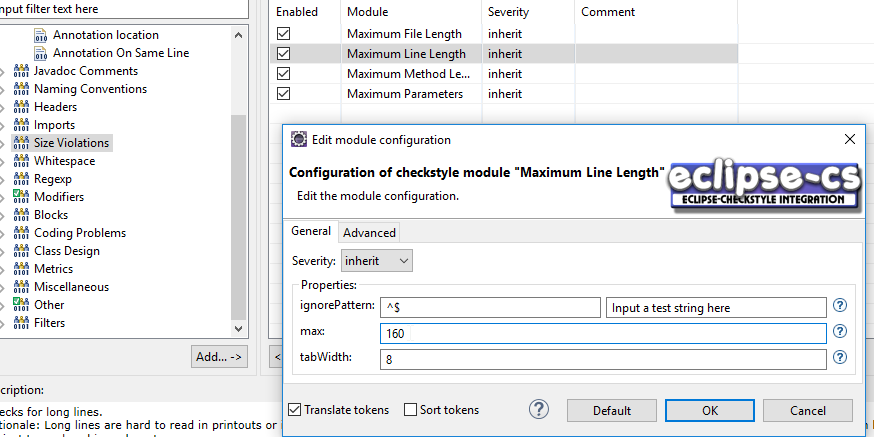
On le setAsDefault

On le paramètre (configurations) :

Javadoc, on touche pas c’est nickel chrome

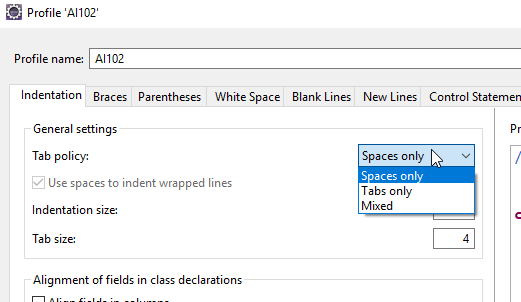


On passe les lignes à 160 :

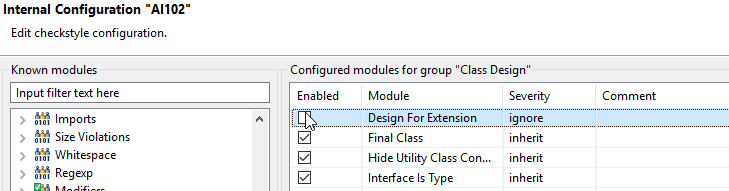


Y’avait aussi l’indentation dans les paramts, décocher UTILISER TAB et indentation :

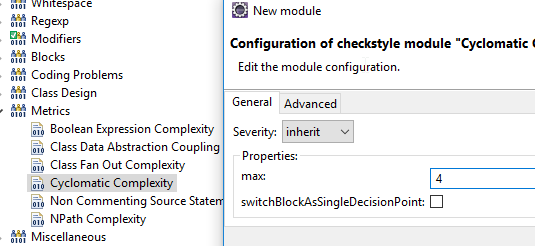
Pour réindenter et appliquer l’indentation, on CTRL A + CTRL I



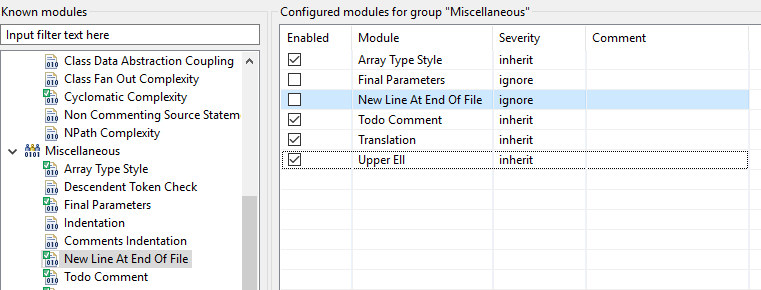
En class design on peut enlver le Design for Extension qui oblige à passer en FINAL



Complexité cyclomatique, on oblige à ne pas dépasser 4 :

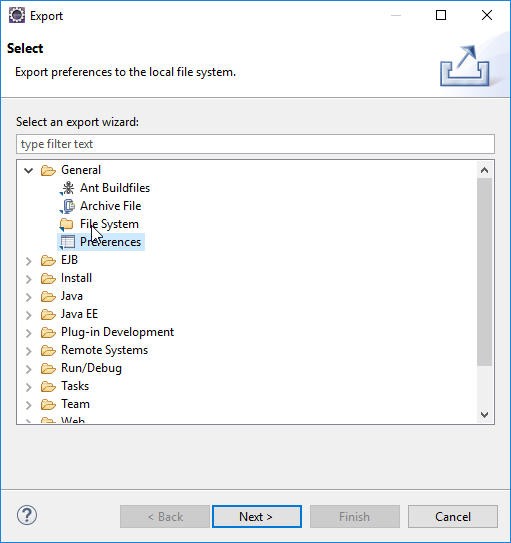


Dans Miscellaneous, on décoche un ou deux trucs :

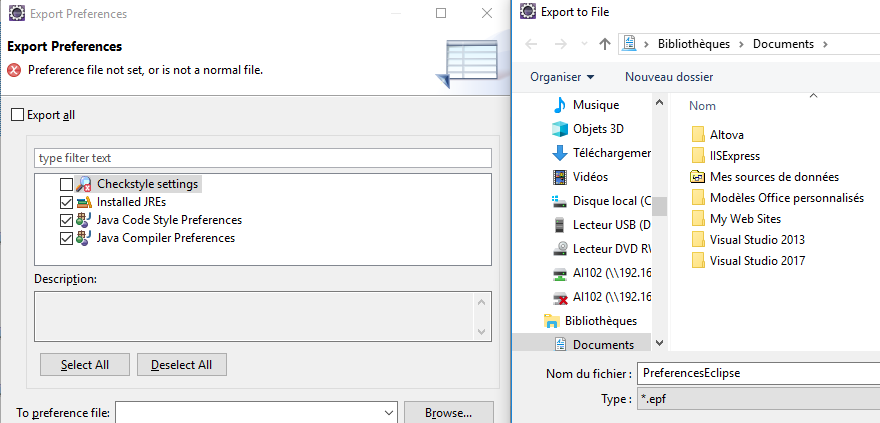


A l’avenir, on peut exporter les préférences !

Clic droit :



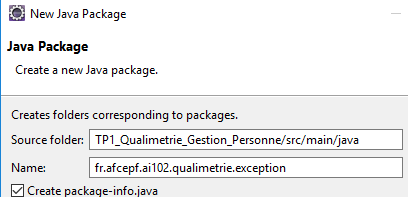
On donne un nom, ex PreferencesEclipse et on enregistre.



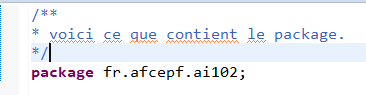
Création d’un fichier :

package-info.java

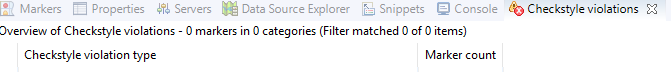
Mieux, le faire en automatisation à la création d’un package : clic sur Create package-info.java



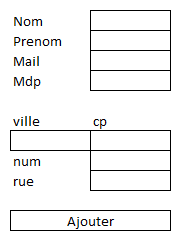
On y écrit juste, après l’avoir mis dans le package adéquat (fr.afcepf.ai102 donc) :



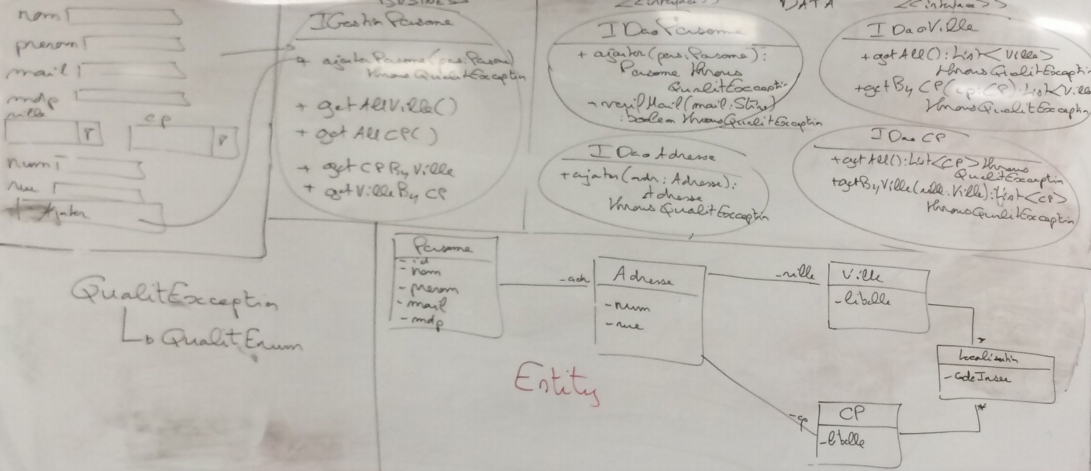
On arrive ENFIN à :



A partir de là, l’exercice :  
Création d’un formulaire, et d’exceptions QualityException 🡪 QualityEnum (de code d’erreur).







On met les constantes dans les Enums,  
Mais avant c’était dans les interfaces.

Dans le DAO on utilise des interfaces ! Elle fournit des méthodes, et on les implémante … Donc si l’on change par exemple de ArrayList à Vector, ça compilera !

Les DAO permettent de conserver quelque chose qui marche, alors qu’on change l’implémentation.

* Lire le GRASP !

**TESTS UNITAIRES**

TDD :  
1- Ecrire les tests : cas nominal, cas alternatif, cas d’échec  
2- Vérifier les NON SUCCESS  
3- Implémentation  
4- On vérifie que tous les tests fonctionnent

Une fois les tests écrits, on implémente et l’on passe l’implémentation en test pour vérifier que tout se passe comme désiré.

Assertions : assert null(obj) pour s’assurer qu’un objet est bien nul,  
assertNotNull(obj) pour le contraire,  
AssertEquals(…,…) pour égalité, etc.  
  
Test en echec si assert revient FALSE.  
Success / failure.

On va créer une classe de tests, utilisant JUnit (JUnit 4)

@Test  
Public void TestUnTruc() {  
 // Ecriture du verifMail par ex  
}

@Before // Execution avant chaque test  
@After // Execution après chaque test  
  
@BeforeClass // Execution avant tous les tests  
@AfterClass // Execution après tous les tests

Pour les tests, on ajoute au pom la dépendance

<dependency>

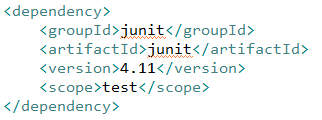
<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

<scope>test</scope>

</dependency>



Ajout pour la connexion et le driver, en scope : compile (compilation / exécution) de :

<dependency>

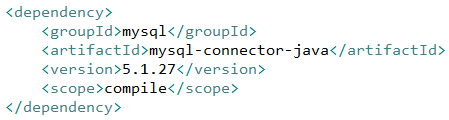
<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.27</version>

<scope>compile</scope>

</dependency>



Java Naming Directory Interface  
Il définit trois grandes choses : comment ? Ou (est l’objet appelé) ? et qui (est a appeler) ?

*Exemple :****jdbc :mySql****://127.0.0.1 :3306/nom\_bdd  
(****comment****://ou/qui)*

Java est arrivé pour :  
- supprimer les pointeurs  
- objets distribués : CORBA

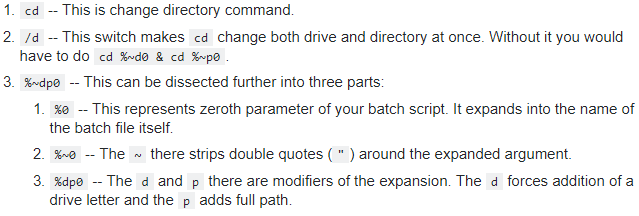
A PROPOS DE LA JVM  
**Espace mémoire de la JVM** qui gère les pointeurs vers la mémoire RAM.  
**Table de référence :** enregistre les noms qui pointent vers la mémoire JVM, qui elle-même pointe vers la mémoire RAM.  
**Classloader**: tout ce qui est static est chargé, le main est alors vivant dans le programme.

Lorsqu’on a du traitement dans une classe static, ce traitement est lancé lors du ClassLoader.  
En général, on utilise ça pour la configuration avant que le programme ne démarre.

Le classloader permet d’embarquer les images dans un jar, entre autre.  
La classe Thread.currentThread() qui possède un .getContextClassLoader(), on peut faire un getRessources(’’relativePath’’)

Nb : constructeur static possible en C# (aberration)

Cd /d %~dp0 🡪 permet de récupérer le chemin



Nb : java**.io** 🡪 **i**nput **o**utput = ce sont les flux !  
Disposent donc toujours d’une méthode .close().

Pour l’ajout de la personne, combien va-t-on avoir de tests ?  
- cas nominal  
- cas d’echec : nom, mail, prénom, mdp trop long …  
 nom null, mail …  
 adresse n’existe pas : c’est une **foreign key**

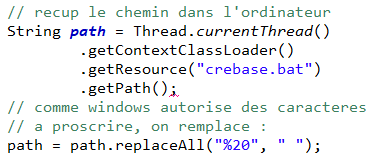
On aura besoin de :  
- Dao : IDaoPersonne  
- adresseNominale : Adresse  
- personneNominal : Personne  
- PersonneExcpected : Personne 🡪 même personne avec un ID de valeur 4 (car trois personnes dans la base)  
- une adresseEchec : Adresse (qui n’existe pas)  
- personneEchecNomTropLong  
- personneEchecNomNull  
- personneEchecPasAdresse

Avant cchaque test, il faut regénérer la base ;   
On va le faire dans un @Before :

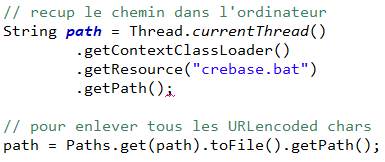
@Before   
// re-génération de la bdd

On lance avec l’OS une JRE, qui lance une instance de JVM dans un process ;  
Via la classe Runtime du JRE, on a une méthode .exe qui permet de lancer des commandes, qui permettent de lancer un .bat

Récupérer le chemin :



Ou mieux :

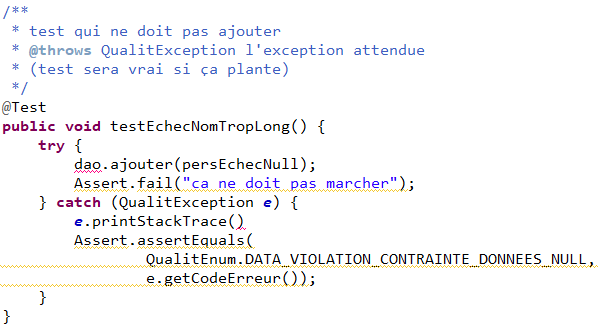


Dans notre modèle, on a fait des tests avec des dépendances, ce qui n’a pas lieu d’être en tests unitaires : on est alors en test d’intégration.

On doit gérer champs par champs dans les tests unitaires !

Actuellement, avant chaque ajout on relance le setup.

Test de nom null (ne doit pas réussir à être ajouté), il faut qu’il récupère l’exception :





**. GIT .**

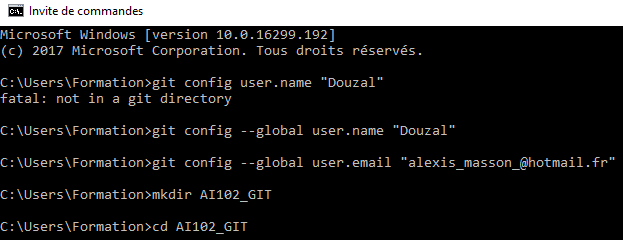
Permet le **versioning**.  
Cela passe par la création d’un clone, mise en local du projet,  
On travaille sur les fichiers, puis quand on a terminé de bosser sur le projet on **comit** (validation) ou on **reset** (rollback) une modification qu’on a faite.

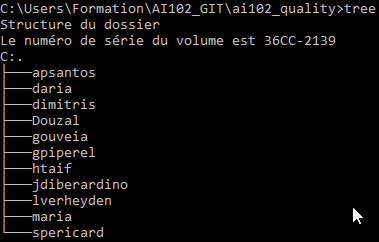
Lorsqu’on a terminé nos modifications locales, et que l’on veut les envoyer au serveur, on les y envoie ; création automatisée d’une nouvelle version.  
  
A tout moment, on peut revenir à une version précédente en chargeant dans l’historique.

Lors de l’ajout de nouveaux fichiers dans la copie locale, on fait du **add**.  
Ensuite, on le **push**, et on **update.**

**Git impose** un username et un mail lors d’un pull.

On installe Git,  
En ligne de commande :  
git config –global user.name = ’’Douzal’’





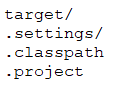
On écrit un fichier  toto.txt avec un « blabla »



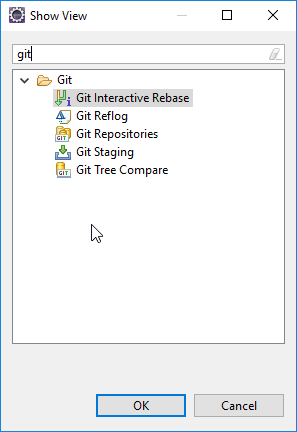
**git commit -m « message »** 🡪 commit avant de **git pull** et **git push**Le **-m** permet de choisir un message pour le commit : on DOIT ecrire un message pour permettre d’assimiler la new version à son apport / précédentes.

On ne commit JAMAIS les configs IDE et le BIN.  
Ainsi, on doit ignorer les fichiers :  
- target  
- settings  
- classpath  
- project  
- les bin

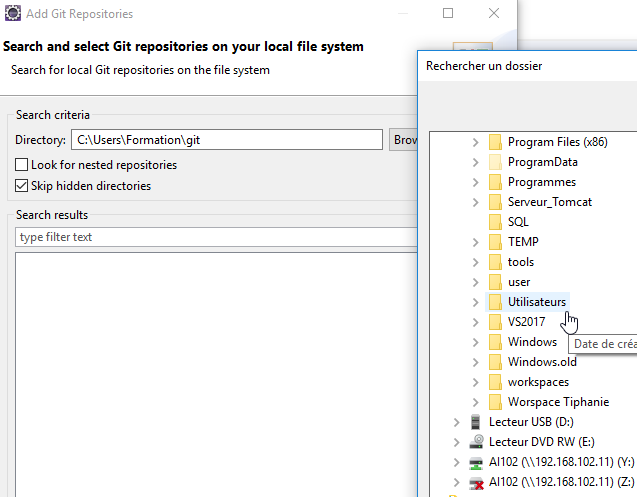
Création d’un fichier nommé .gitignore



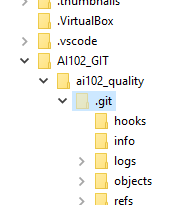
On retrouve notre IDE préféré,  
Window > view > others > Git > Git repositories et choix du bon



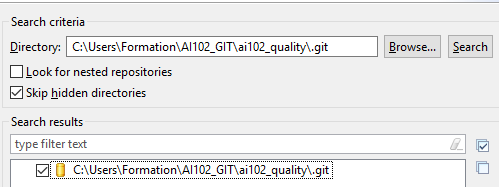
On clic Add an existing local…



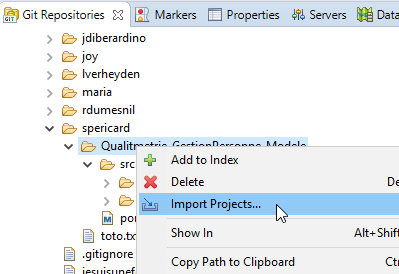
Choix du dossier : .git



Finish



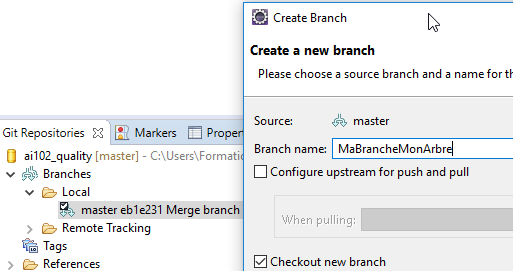
On import le projet de Seb :



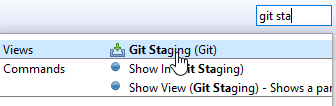
Au niveau de l’intégration des branches de travail, il faut suivre la procédure :  
On pull la **master**  
On **checkout** la master  
On **merge** notre branch

Plus simple que **un dév** s’occupe de la gestion des branches : il pull la master, tout le monde récupère la branche et dév à partir d’elle.

Création d’une de mes branches



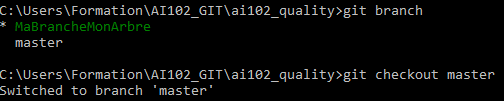
View git staging

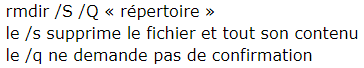


A ne jamais faire :  
On a une copie locale ; il ne faut jamais prendre un dossier physique avec son OS, et le mettre à la corbeille.  
Il faut dire à git de la supprimer !  
En effet, sinon git n’est pas au courant : fichier sera corrompu, on ne pourra ni pull ni push.

Git rm -r dossier

Se foutre sur le branch **master** :





Pour nos tests,  
On va avoir besoin de couper les dépendances.  
Pour ce faire, on crée des mock.

MockDaoAdresse (par exemple) : son but est de couper les dépendances, et les émuler. On teste ainsi unitairement la méthode !

On utilise les classes anonymes : redéfinir à la volée.

DEGIN PATTERN :  
  
C’est un ensemble de classes, permettant de répondre à un besoin de :  
- comportement spécifique  
- de création spécifique  
- de structuration spécifique  
- architecture : MVC, MVVM ; DAO ; IOC (inversion of control), …

Framework : ensemble de classe préprogrammées, basé sur un pattern et proposant des outils permettant de manipuler ce pattern d’architecture, et peut proposer des boites à outil.

Injecter des dépendances : on utilise la réflexion (introspection)  
Classe 🡪 Field 🡪 Methode 🡪 @annotation

Les annotations impliquent un traitement spécifique.  
On peut ensuite faire du traitement, invoquer, manipuler n’importe quoi dans la classe.

**.**

**JEE .**

**JEE :**- ensemble de **spécifications** pour le développement d’applications d’entreprise  
- ensemble de **bonnes pratiques** pour le développement d’applications d’entreprise  
- JSR : certifications

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| JEE | JSE | Servlet/JSTL | EJB (JavaBean) | JSF | JPA |
| 5 | 1.5 & 1.6 | 2.5 | 3.0 | 2.1 & 3.0 | 1.2 |  |
| 6 | 1.7 | 3.0 | 3.1 | 2.0 |  |
| 7 | 1.8 | 3.1 | 3.2 | 2.2 | 2.1 |

Utilisation de framework spécifiques :  
Pour la présentation, utilisation de JSF  
Pour l’accès aux données : JPA Java Persistent API

On doit utiliser selon la ligne adéquate :  
pour JEE 7 🡺 JSE 1.8, JSTL 3.1, EJB 3.2, JSF 2.2 et JPA 2.1

DataSet contient des dataTable travaillant avec des TableAdapter (méthodes fill(), update()) ;  
InsertCommand, delete, update, Select.  
Objet command : CommandTest 🡪 Connection : connectionString  
ADO = Access Data Object.

|  |  |
| --- | --- |
| **Désormais : ASP.NET** | **Avant : ASP** |
| HTML CSS JS **C# VB.Net** | HTML CSS JS **VBScript** |

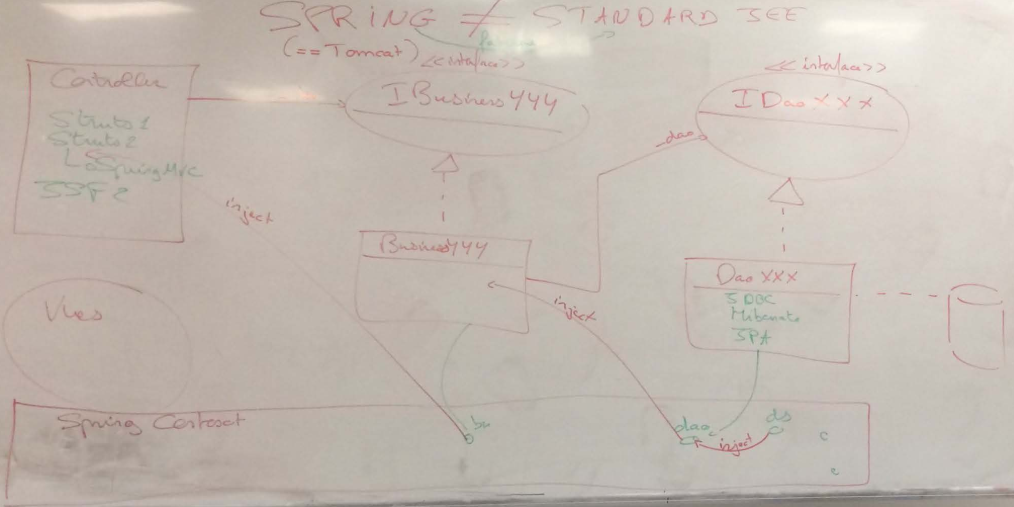
Dans le monde JEE, serveur d’application standard JEE doit posséder tous les composants standard : composants des spécifications, JSF, EJB, JPA, …

Tomcat n’est qu’un conteneur de servlet.

JBOSS **n** a tous les composants du standard Je**n**:  
JBOSS7 a tous les composants du standard JE7.  
JBOSS8 a tous les composants du standard JSE 1.8, donc de JEE 7.

SPRING STANDARD JEE

On travaille sur Tomcat.  
SPRING fabrique les composants, et gère les dépendances entre les couches.  
On a besoin d’importer les bibliothèques de classe



Spring fournit de quoi utiliser AspectJ

Programmation par aspects.

@around : autour du traitement

Depuis JSE 1.6, annotations.  
Ainsi on peut lancer des trucs avec les annotations 🡪 simplicité.  
Ex : @webServlet(’’un-pattern’’)

Classement selon   
Page  
Request  
Session  
Application

**. STRUTS (1.3) .**

MVC2, sorte de giga controller.  
Framework controller 🡪 SRUTS est un servlet !

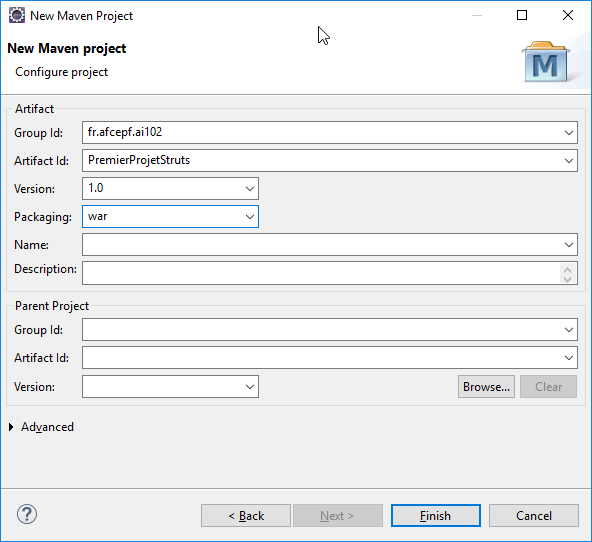
Il automatise les traitements liés au web qu’on avait dans les servlets.

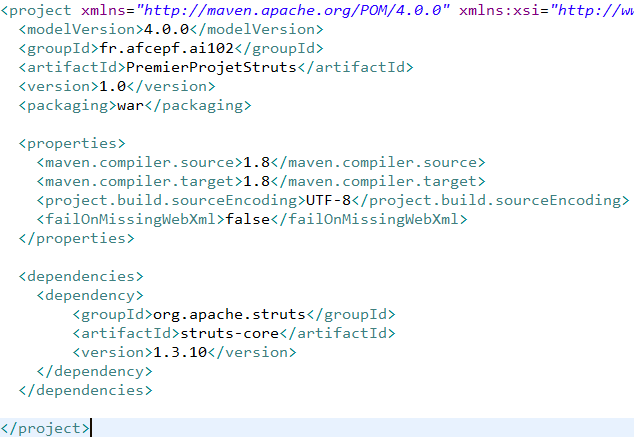
Struts intercepte toutes les actions, il peuple les valeurs des attributs de la classe (avec des getParameters) lors des requêtes qui passent par lui.

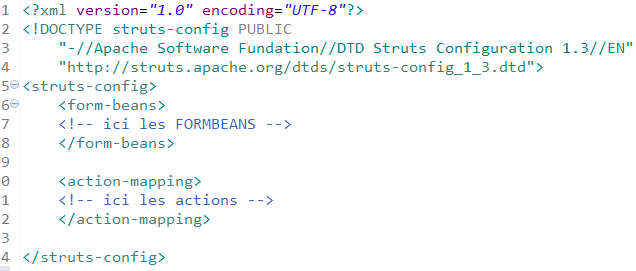
Création projet :  
  
PremierProjetStruts  
🡪 org.apache.struts :struts-core :1.3.10

🡪 paramétrage de struts : web.xml

🡪struts-config.xml





Création du struts-config.xml :  


Les TAGLIB : [clic](http://www.commentcamarche.net/faq/19383-taglibs-partie-1-presentation)

Remote Method Invocation (RMI)

Dialogue entre deux applications / machines, à travers TCP/IP (envoie de strings) et la sérialisation.  
Protocole RPCI (Remote Protocole Communication Invocation)

Pour la communication il faut un port, l’IP,

Le proxy est un design pattern : c’est un objet qui réalise une interface de l’objet concrêt, et appelle les mêmes méthodes.

Marshalling (en programmation) : erreur unmarshalling return 🡪 erreur de sérialisation. L’objet Toto n’est pas sérialisable.

Le numéro de version doit être identique entre les deux objets (celui du local, et celui du client).  
Les objets sérialisables doivent forcément avoir un numéro de version.

getClientHost()

Dans java.policy on peut changer les permissions.  
Système de « grant »

La sécurité se fait par le pare feu (plus pour les administrateurs système), ou par la JVM.

**. Les EJB .**

Permettent à une application d’être contactée par une autre, à distance.  
On aura besoin obligatoirement d’un port, d’un IP, l’interface de l’objet distant, ainsi que les objets manipulés par l’interfaces – ces derniers devant être sérialisables.

Un EJB3 va vivre dans un serveur d’application, qui possède un containeur d’EJB.

Les EJB3 sont arrivés à partir de la version 1.6 du JRE, donc les annotations fonctionnent avec eux.

Un EJB a deux scopes :  
- @Local : toutes les applications hébergées par le même serveur d’application ;  
- @Remote : de n’importe où

L’EJB va aussi définir la façon dont il vie sur le serveur :  
- @Singleton : une instance unique, quelle que soit l’application qui l’appelle (comme le RMI) ;  
- @Stateless : sans état, auquel cas le serveur va gérer un pool pour leur gestion ;  
- @Statefull : même EJB pour un contexte applicatif appelant donné. Le singleton est donc le même pour toutes les applications.

Cycle de vie de l’EJB sur le serveur 🡪 en plus par rapport au RMI.  
On utilisera majoritairement du stateless.

Les annotations définissent comment il est défini, et comment il va vivre.  
Côté client, idem que RMI : un proxy.

Le serveur d’application :  
Depuis le standard JEE7, est imposé au niveau de la sécurité des EJB le serveur d’application : c’est lui qui est en charge de la sécurité.  
Depuis le standard JEE7, les EJB ne passent plus par le RPC (Remote Protocol Command) mais par http, d’où la sécurité au niveau du serveur.

JNDI :  
http-remoting://IP :Port/ + JNDI

En tant que client interne (local), on peut aussi trouver un ejb …://

Un EJB peut être dans :  
- EAR  
- un projet  
- une interface  
- un package

JNDI nommé : Nom\_ear/nom\_Projet/nom\_EJB !interface

http-remoting://IP :Port/**Nom\_ear/nom\_Projet/nom\_EJB !interface**

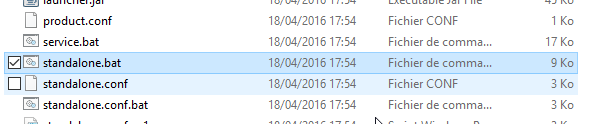
Contrairement au RMI, natif dans la JRE, need d’un driver.  
Les requêtes sont constituées par le driver : c’est la partie qui n’est pas standard.

Selon le type de serveur utilisé, la configuration demandée par la bibliothèque de classes du client peut changer.

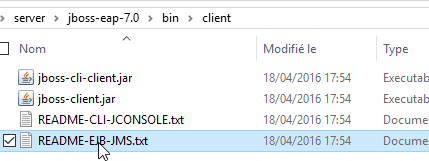
Nous allons utiliser JBOSS EAP 7 (EAP : Entreprise Application Platform)  
Attention, ne pas prendre JBOSS AS (Application Serveur) qui ne sont plus maintenus.

Le serveur JBoss est un serveur écrit en Java ; need le décompresser, need une JavaHome (un chemin vers la jdk) sur le système.

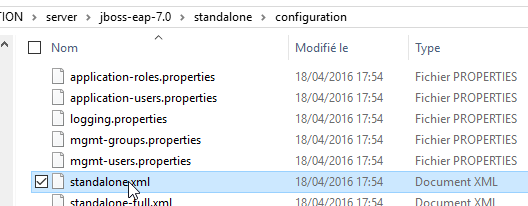
Standalone executable à utiliser sur JBoss :



Et dans le dossier client, les dépendances à utiliser :



On regarde ensuite le dossier :



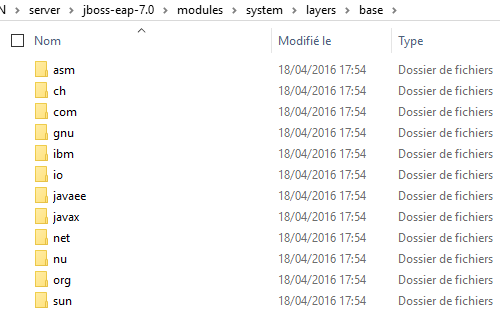
Grappe | Cluster  
Apache communique avec des Tomcat avec ajp.  
Il est ainsi activé dans le JBoss (ce que l’on voit en lisant le Standalone).

On retrouve aussi les ports de managments : le 99.90 (utilisé par NVIDIA).

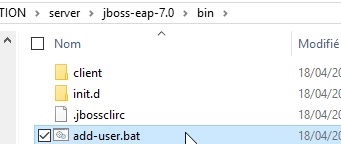
Commande netstat permet de montrer les ports utilisés.

On change donc, en mettant des 1 devant chaque pour être sûr d’être tranquille.

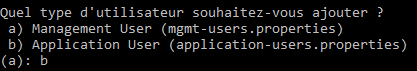
Ensuite à ce chemin d’accès



On change :

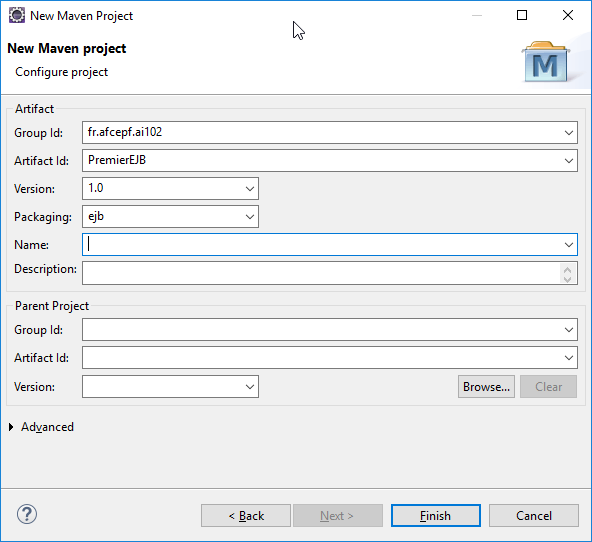


Lancement du .bat

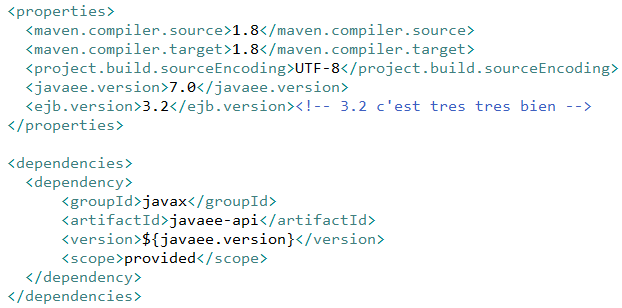


Validation avec b.

Utilisateur : root  
Mdp : afcepf

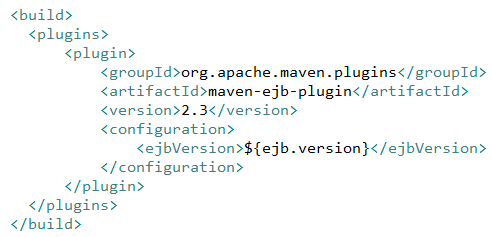
Création du projet :  


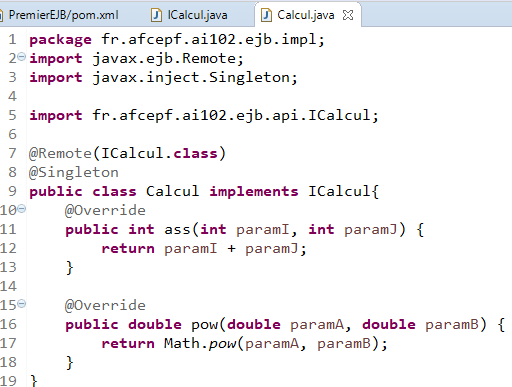
On y fou ces dépendances :

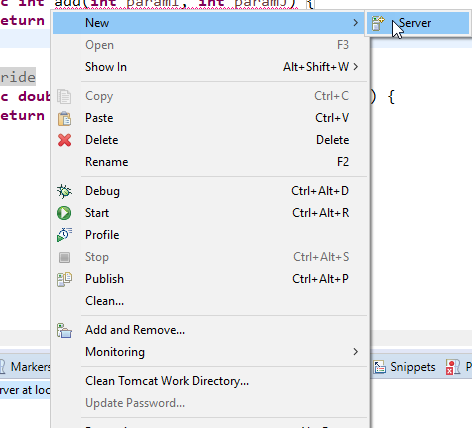


Grosse différence entre les EJB2 et les EJB3, et Java peut gérer les différences.  
Avant la version 3.0 des EJB, dasn le Meta-INF on avait besoin d’un ejb-ar.xml, qui décrivait les bails.  
Depuis, plus besoin : maven en construisant le projet et selon la version du projet qu’il va packager, génère le fichiers de config.

On peut retrouver ça dans le pom.xml : section build permet de définir du comportement pour la conduite du projet.



Voici donc notre premier EJB qui implémente l’interface ICalcul :  




Settings.xml pour la config de maven.

Emplacement des repo red hat dans la config.

On s’écrit cette classe :

**package** fr.afcepf.ai102.ejb.test;

**import** java.util.Hashtable;

**import** java.util.Properties;

**import** javax.naming.Binding;

**import** javax.naming.Context;

**import** javax.naming.InitialContext;

**import** javax.naming.Name;

**import** javax.naming.NameClassPair;

**import** javax.naming.NameParser;

**import** javax.naming.NamingEnumeration;

**import** javax.naming.NamingException;

**import** fr.afcepf.ai102.ejb.api.ICalcul;

**public** **class** CallEJB {

**private** **static** **final** String ***JNDI\_NAME*** =

"PremierEJB/Calcul!fr.afcepf.ai102.ejb.api.ICalcul";

**private** **static** **final** String ***JNDI\_PROVIDER*** =

"http-remoting://127.0.0.0:18080";

**private** **static** **final** String ***USERNAME*** = "root";

**private** **static** **final** String ***PASSWORD*** = "afcepf";

**public** **static** **void** main(String[] args) {

Properties props = **new** Properties();

props.put(Context.***PROVIDER\_URL***, ***JNDI\_PROVIDER***);

props.put(Context.***SECURITY\_PRINCIPAL***, ***USERNAME***);

props.put(Context.***SECURITY\_CREDENTIALS***, ***PASSWORD***);

props.put(Context.***INITIAL\_CONTEXT\_FACTORY***, "org.jboss.naming.remote.client.InitialTextFactory");

props.put(Context.***URL\_PKG\_PREFIXES***, "org.jboss.ejb.client.naming");

props.put("org.jboss.naming.client.ejb.context", **true**);

**try** {

Context ctx = **new** InitialContext(props);

ICalcul proxy = (ICalcul) ctx.lookup(***JNDI\_NAME***);

System.***out***.println(proxy.add(20, 22));

System.***out***.println(proxy.pow(2, 11));

} **catch** (Exception e) {

e.printStackTrace();

}

}

}