Génie Logiciel

Mise en place votre écosystème logiciel pour le projet GL avec Gradle

Objectifs :

- installer tous les composants nécessaires pour votre projet GL (au moins une partie)
- et comprendre (un minimum) comment ils interagissent entre eux.

Prérequis :

- disposer d'un JDK. Le « 15 » des fichiers fournis veut dire « Java 15 ». Adaptez à votre cas !
- disposer d'un IDE (Eclipse ou **IntelliJ** par exemple)
- disposer d'un compte ***Gitlab*** : https://gitlab.enssat.fr/
- avoir installé **Gradle** (c'est un package d'IntelliJ) : https://gradle.org/install/

Pour une introduction à Git : https://www.cristal.univ-lille.fr/TPGIT/

Pour une introduction à Gradle : https://docs.gradle.org/current/userguide/what_is_gradle.html

Comptabilité des versions Gradle/Java : https://docs.gradle.org/current/userguide/compatibility.html

1. Création du projet dans IntelliJ et premier commit

Dans IntelliJ, créez un nouveau projet via le menu File → new project → gradle → java

Créez une classe *Money* dans un package *demo* (à positionner dans le répertoire *src/main/java*).

Maintenant que vous avez un début de classe, il est temps (et oui déjà) de faire votre premier commit. Attention néanmoins à ne pas commiter tout le répertoire ! Il contient de nombreux fichiers de configuration liés à votre IDE, qui sont donc propres à *votre* machine. Vous ne souhaitez pas imposer ces fichiers à vos collègues, ni récupérer les leurs lorsque vous collaborerez à plusieurs.

A l'aide de votre IDE, ajoutez un nouveau fichier à la racine de votre répertoire de travail intitulé *.gitignore* dans lequel vous allez préciser les fichiers et dossier que vous ne souhaitez pas commiter (le répertoire *.idea* si vous utilisez IntelliJ par exemple, ou encore *.gradle*).

Configurez un système de versionnage via le menu VCS → import into version control → share project on Gitlab (une extension peut être nécessaire pour utiliser Gitlab).

Indexez les fichiers restants (si votre IDE ne l'a pas fait pour vous) puis commitez (et pushez).

2. Amélioration de Money et compilation avec Gradle

Etoffons la classe *Money* qui représente un montant *amount* entier dans une devise *currency* donnée. Ajoutez à l'aide de votre IDE un constructeur ainsi qu'une méthode *toString().*

Créez une nouvelle classe *Main* dans le répertoire *src/main/java* dans laquelle vous allez créer une méthode *main* avec un instance de la classe *Money* représentant 42€ (42, ''EUR'') et l'afficher avec *System.out.println()*. Exécutez votre méthode *Main.main*.

Vous pourrez constater le résultat dans la sortie de la console de l'IDE. Retournez maintenant dans votre terminal et lancez la commande *gradle build* pour construire l'exécutable automatiquement. En éditant le fichier *build.gradle*, ajoutez l'extrait suivant tout en bas.

```
jar {
    manifest {
        attributes "Main-Class": "Main"
    }
}
```

L'exécutable .jar est généré dans le répertoire *build/libs*. Vous pourrez vérifier, en le lançant à l'aide de la commande *java -jar chemin/fichier.jar*, que vous retrouvez bien le même résultat qu'en exécutant la méthode *Main.main*. (Si vous rencontrez des difficultés pour générer ce fichier jar à l'aide de la commande *gradle build*, vous pouvez toujours utiliser la commande *gradle jar*)

Le fichier .jar généré n'a pas vocation à être mis en ligne sur votre répo, mais vous pouvez désormais à nouveau commiter le reste de votre travail.

3. Mon premier test

Créez un répertoire *test/java* dans le répertoire *src* de votre projet.

Dans le répertoire *test/java* ajoutez un package *demo*, et une classe *MoneyTest* dedans. Intégrez une méthode de test *testAddEuros()* vérifiant qu'il est possible de bien additionner deux instances de la classe *Money* qui ont bien dans la même devise, comme suit :

```
public class MoneyTest {

    @Test
    public void testAddEuros() {
        Money m1 = new Money(20,"EUR");
        Money m2 = new Money(10,"EUR");

        Money expected = new Money(30,"EUR");

        assertEquals(expected,m1.add(m2));
    }
}
```

La méthode *assertEquals()* doit permettre de vérifier que vous arrivez bien à des instances identiques, pour valider le test. Votre IDE vous proposera de l'importer d'un paquet JUnit. Remarquez également la présence de l'annotation « @Test » précédant la méthode. Il reste à ajouter la méthode *add* pour la classe *Money*.

```
public Money add(Money that) {
    return new Money(this.amount+that.amount,this.currency);
}
```

Il n'y a plus qu'à lancer l'exécution du test. Mais cela suppose néanmoins de pouvoir comparer deux instances de la classe *Money* ! Cela se fait à l'aide d'une méthode *equals()*. Demandez à votre IDE de définir cette méthode pour vous : dans IntelliJ par exemple, suivez le menu Code → Generate → equals() and hashcode().

Votre premier test devrait désormais s'exécuter correctement dans l'IDE, ainsi qu'avec Gradle (commande *gradle test*). Commitez, et mettez à jour votre répo en ligne avec un *git push*.

Par ailleurs, remarquez la présence d'un répertoire *build/reports/tests/test* dans votre arborescence, et ouvrez le fichier *index.html* dans votre navigateur favori.

4. Est-ce que Gitlab peut confirmer que mon test passe bien de son côté aussi ? (Premier pas vers l'intégration continue)

Ajoutez le fichier *gitlab-ci.yml* à la racine de votre répertoire de travail, dans lequel vous allez indiquer à Gitlab quel comportement adopter (fichier à télécharger directement sur l'ENT).

Commitez ce nouveau fichier puis allez sur la page de votre répo dans Gitlab. Allez voir ce qui se passe dans le menu CI/CD : vous y retrouverez votre dernier commit, qui est en cours de traitement : Gitlab recompile votre exécutable et refait tous vos tests pour vous !

Variante Github

Créez un dossier *.github/workflows* à la racine de votre répertoire de travail, et ajoutez dans ce nouveau dossier un fichier *java.yml* dans lequel vous allez indiquer à Github quel comportement adopter (fichier à télécharger directement sur l'ENT).

Commitez ce nouveau fichier puis allez sur la page de votre répo dans Github. Remarquez la présence de l'onglet « Actions » et allez dessus. Vous y retrouverez votre dernier commit, qui est en cours de traitement : Github recompile votre exécutable et refait tous vos tests pour vous !

5. Et UML dans tout ça ?

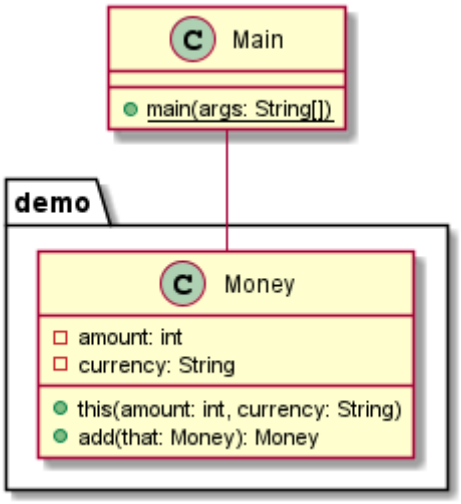
Le plugin *PlantUML integration* dans IntelliJ (PlantUML tout court dans Eclipse) vous permettra de maintenir à jour vos schémas de classe.

L'outil Graphviz (version 2.44.1) devra peut-être être téléchargé en complément de PlantUML pour permettre l'affichage en direct de votre modèle. Plus d'informations ici : https://plantuml.com/fr/graphviz-dot

Créez un nouveau fichier class_diag.puml à la racine de votre répertoire, et ajoutez-y le contenu suivant :

```
@startuml
package demo {
    class Money {
     - amount: int
     - currency: String
     + this(amount: int, currency: String)
     + add(that: Money): Money
    }
}
class Main {
 + {static} main(args: String[])
}
Main -- Money
@enduml
```

Cela devrait vous permettre d'obtenir le diagramme suivant dans la fenêtre de visualisation PlantUML.



Si vous souhaitez visualiser ce diagramme directement dans l'interface de Gitlab, il va falloir tricher un petit peu (en attendant une mise à jour côté Gitlab, le ticket est ouvert pour cette fonctionnalité). L'extension Kroki (déjà en place sur le Gitlab de l'Enssat) permet de générer des images de diagrammes (+ d'infos ici pour votre curiosité : https://kroki.io/).

Pour arriver à nos fins, il vous faudra simplement ajouter l'extension .md à votre schéma (qui avait sûrement une extension .puml ou .plantuml jusqu'à présent), puis la ligne ```*plantuml* en début de fichier, et terminer le fichier avec la ligne ``` (il s'agit d'un accent grave). L'extension PlantUML d'IntelliJ reste ouverte d'esprit avec ces modifications et reste complètement fonctionnelle.

Variante Github

On va maintenant demander à Github de nous générer une image vectorielle du modèle à chaque mise à jour de ce dernier. Donc dans le dossier *.github/workflows* on va ajouter un nouveau fichier *plantuml.yml* dans lequel on va décrire les actions à effectuer pour cela. Récupérez ce fichier sur l'ENT.

Il n'y a plus qu'à commit et push, et vous obtiendrez votre image vectorielle à la racine de votre répo sur Github. Vous pouvez également spécifier dans quel répertoire l'image devrait s'afficher en éditant le fichier *plantuml.yml* pour organiser vos fichiers de modélisation.

6. Premiers pas vers la couverture de code

Vous avez pu écrire plus tôt votre premier test pour vérifier s'il était possible d'additionner deux instances de la classe *Money*. Nous allons vérifier si ce test est vraiment suffisant vis-à-vis de tout le code que vous avez déjà écrit en analysant la couverture du code de votre classe de test. Votre IDE vous propose certainement un outil natif pour cela (IntelliJ propose « Coverage » par exemple, sinon tous les IDE modernes permettent d'utiliser le plugin JaCoCo pour le langage Java par exemple).

Dans le menu « Run » cliquez sur l'option « Show Code Coverage Data ». Trois métriques de couvertures sont proposées par défaut : classes, méthodes, et lignes. Analysez vos résultats, puis améliorez vos tests en conséquence pour améliorer ces métriques. Vous pouvez également exporter ces résultats sous forme de tableaux HTML à l'aide de l'option « Generate Coverage Report » pour les consulter à l'aide de votre navigateur Internet.

Devra-t-on toujours créer autant de répertoires à la main, alors que c'est classique dans chaque projet java (ou autre) ? Non, en fait Gradle et votre IDE permettent de faire tout cela automatiquement… Mais c'est bien de faire des choses à la main pour découvrir aussi.

7. En bonus…

Au-delà de l'addition, quelles autres opérations pourriez-vous imaginer effectuer à l'aide de cette classe *Money* ? Comment les tester ? Testez !

Ajoutez une nouvelle classe *MoneyBag* qui prend en compte plusieurs devises différentes et offre une méthode de normalisation du *MoneyBag* (un seul couple (montant, devise) pour chaque devise). Proposez des méthodes pour ajouter et retirer des instances de *Money* du *MoneyBag* puis testez-les et enfin vérifiez vos taux de couverture.

Dans un contexte international vous devrez également être en mesure d'échanger un montant *a* d'une devise *b* contre un montant *c* d'une devise *d* (cela implique de gérer des taux de change quelque part…) au sein d'un *MoneyBag*.

Plus généralement vous disposez soudainement de divers moyens de paiement : vous obtenez ainsi une carte bancaire débit, une carte bancaire crédit, et un chéquier (oui ça existe encore). Ces différents moyens de paiement (chacun associé à une seule devise) existent au sein d'un *Wallet*. Décrivez des méthodes de transfert d'argent entre ces moyens de paiement et un *MoneyBag* (on veillera à ne pas s'enrichir en prétextant une erreur informatique en testant les différents comportements attendus).