R5.A8.D7 : Qualité de Développement
Feuille TD-TP n° 1

Révisions sur les tests unitaires
Java, IntelliJ, Gradle, JUnit, JAssert, jacoco, mockito

Objectifs :

1.- *Reprise en main de l'outil de développement utilisé : IntelliJ + Java*
2.- *Révision des test unitaires vus dans la ressource R4.02-Qualité de développement*

Sujet :

Il s'agit de développer un sous-ensemble d'une classe simple, nommée `Calculator`, réalisant quelques opérations basiques sur des éléments de type primitifs (par exemple, int, double) et personnalisés (par exemple rationnels).

Le développement de la classe sera accompagné des tests unitaires associés.

Cette feuille de TD-TP a pour but, non pas de développer cette classe de manière exhaustive, mais de se remémorer les acquis du module R4.02-Qualité de développement consacré aux tests unitaires et Intégration Continue utilisant JUnit, JAssert, Jacoco et Mockito, Gradle, Git/Github lors d'un développement en Java avec l'IDE IntelliJ.

Ressources à votre disposition :

- L'archive **`junit5-jupiter-starter-gradle.zip`**

  C'est un projet 'Modèle' minimal pour le développement en Java avec IntelliJ et gradle. La fonction main() de son unique clase Main affiche « Hello world ».

  Il devra être configuré pour le développement demandé.

- Le fichier **`build.yml`**, pour l'automatisation de tests sous la forme d'action Github. Il sera complété lors de la séance.

Préparation du travail

1.- Création du dossier consacré aux TDs et TPs de cette ressource (R5.A.08 – R5.D.07)

Dans le dossier de votre espace réseau destiné à vos travaux pratiques, créer un sous-dossier destiné à recevoir tous les TPs de la présente ressource. Nous l'appellerons, par exemple : **`r5.A08.D07`**

Dans ce dossier, créer un dossier **`tdtp1`**.

2.- Installation des fichiers

a. Télécharger l'archive **`junit5-jupiter-starter-gradle.zip`** disponible sur eLearn. Décompresser l'archive.
b. Déposer le dossier décompressé dans **`r5.A08.D07\tdtp1`** puis supprimer l'archive **`.zip`** téléchargée.
c. Changer le nom du dossier/projet → **`Calculator`**
d. Lancer IntelliJ, ouvrir le projet **`Calculator`**
e. Compiler, puis exécuter **`main()`**

Vous êtes prêt à passer à l'étape suivante.

3.- Vérifier les paramétrages du projet IntelliJ – Configurer la variable d'environnement `JAVA_HOME`

a. Dans IntelliJ, à partir du **`Main menu`**, et dans le menu **`Project Structure/Project Settings/Project`** identifier la version du JDK présente dans la rubrique **`SDK`**.
Exemple : 11 (11.0.16)

Il s'agit probablement de la version de JDK installée par défaut sur les bureaux virtuels de l'IUT. Consulter la version du JDK installée sur **`C:\Program Files\Microsoft\`**
Exemple : **`C:\Program Files\Microsoft\jdk-11.0.16.101-hotspot`**

Si aucun SDK n'est associé au projet, télécharger la version de votre choix (nous vous conseillons Amazon coretto 18.0.2 – Langage level 18). Noter le chemin du dossier d'installation.

b. Quitter IntelliJ

c. A partir du Panneau de Configuration de Windows, créer une variable d'environnement **`JAVA_HOME`** liée à votre compte et préciser le chemin d'accès au dossier d'installation du JDK consulté au point précédent.

d. Lancer IntelliJ

e. Dans IntelliJ, à partir du **`Main menu`**, puis dans le menu **`Settings /Build, Execution, Deployment/Build Tools/Gradle`**, renseigner la rubrique **`Gradle JVM`** avec la valeur **`JAVA_HOME`**.

f. Sauvegarder.

Vous êtes prêt à passer à l'étape suivante.

4.- Préparer la structure du code (main et test)

a. Dans **`src/main/java`**, créer un package java nommé **`calculator`**[1]

b. Déplacer la classe **`Main`** dans le package **`calculator`**

Dans le package **`com.nomChoisi.calculator`**

c. Créer une classe **`Calculator`**

d. Créer la classe de test associée (**`CalculatorTest`**)[2]. Utiliser JUnit5.

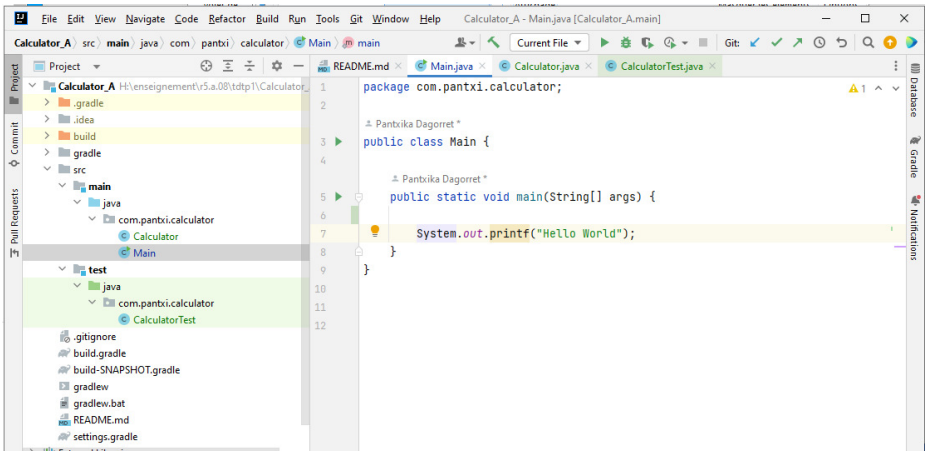


Figure 1 : Projet Calculator structuré, minimal

[1] Pour rappel, la structure du code et noms des packages en Java est la suivante :
- Le code de l'application se trouve dans le dossier src/main/java
- Le code des tests se trouve dans le dossier src/test/java
- La pratique de nommage des **packages** est la suivante :
  `com.nomEntreprise.nomPackage` ou bien **`com.nomProgrammeur.nomPackage`**
  Par exemple : **`com.pantxi.calculator`**

[2] Utiliser le raccourci : **`Ctrl+Shift+T`** après avoir positionné le curseur à l'intérieur de la classe. La liste des raccourcis de l'IDE IntelliJ est disponible sur : **`https://www.jetbrains.com/help/idea/reference-keymap-win-default.html#top_shortcuts`**

5.- Préparer la gestion des versions de votre projet (à l'aide de git et github)

a. Depuis l'explorateur de fichiers de Windows, à la racine du dossier du projet Calculator, à l'aide de gitBash, créer un dépôt local (**`git`**)
b. Créer un dépôt distant associé (**`github`**)
c. Sur le dépôt local (**`git`**), engager le projet minimal (**`git add…`** puis **`git commit…`**) puis le pousser sur le dépôt distant (**`git push origin main`**)

Vous êtes prêt à travailler !

Travail à faire

6.- Coder et tester quelques opérations simples sur des types primitifs

a. *Pour chacune des méthodes* dont les signatures suivent :

```
public int add(int opG, int opD) ;
public int divide(int opG, int opD) ;
```

- écrire sa définition, dans un premier temps sans tenir compte des possibles erreurs d'exécution pouvant advenir en raison de paramètres inapropriés
- écrire le/les tes(s)t associé(s) en utilisant la bibliothèque **`jAssert`** [3,4,5]
- exécuter le(s) test(s) et visualiser le résultat sur la fenêtre dédiée de l'IDE
- le résultat est également sauvegardé, au format xml dans nomDuProjet/build/test-results/test/, et au format html dans le dossier nomDuProjet/build/reports/tests/
- isoler l'ajout de la méthode dans une nouvelle version, dans le dépôt local puis pousser vers le dépôt distant.

Rappel : Test individuel – Batterie de tests

Les tests peuvent être lancés de manière :

- individuelle, via le menu contextuel associé à chaque méthode de test (Figure 2) :

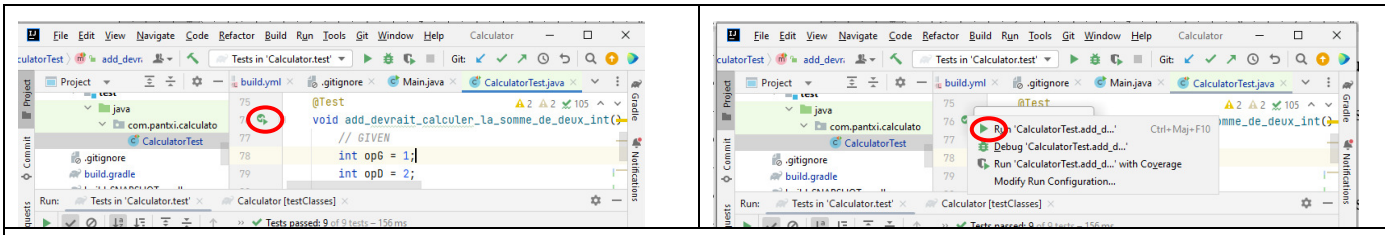


Figure 2 : Exécution d'un test individuel

- ou collective (tous simultanément)

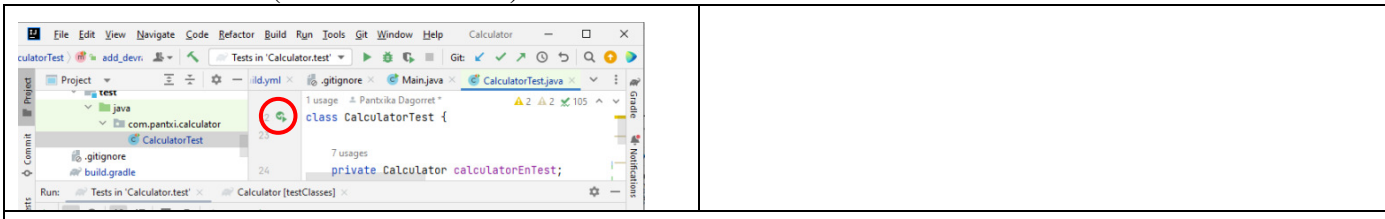


Figure 3 : Exécution collective de tous les tests de la classe (1)

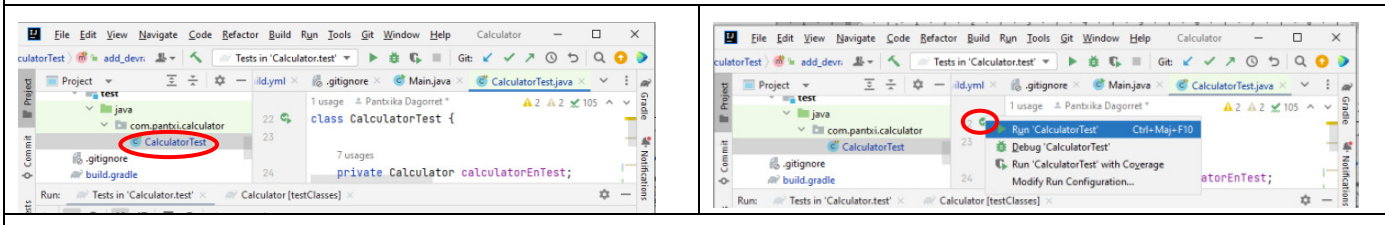


Figure 4 : Exécution collective de tous les tests de la classe (2)

---

[3] Etendre les **dépendances** du fichier **`build.gradle`** avec la dernière version (3.26.3) de assert-core, à partir de l'url : **`https://mvnrepository.com/artifact/org.assertj/assertj-core`**

[4] Les méthodes à importer pour écrire les assertions AssertJ appartiennent à la classe **`Assertions`**. Elles sont dans le package **`org.assertj.core.api.Assertions`** :
https://www.javadoc.io/static/org.assertj/assertj-core/3.26.3/org/assertj/core/api/Assertions.html
https://assertj.github.io/doc/#assertj-core-assertions-guide

[5] Appliquer les bonnes pratiques vues en R4.02 : bien nommer les cas de test, notation snake_case, utiliser des import static pour les méthodes statiques, zones **`GIVEN WHEN THEN`**, …

7.- Simplification d'un test : méthodes de montage / démontage

Les méthodes exécutées avant / après chaque test, et appelées méthodes **de montage** (set up) ou **démontage** (tear down), permettent de diminuer le nombre de lignes de chaque test et donc de le rendre plus lisible.

a. Utiliser les annotations **`@BeforeEach`** et **`@AfterEach`** de JUnit[6] pour alléger chaque test avec les actions d'initialisation / nettoyage communes à tous les tests.

b. Enregistrer, versionner.

8.- Montage et démontage - Cas des méthodes statiques

Il existe des situations où les méthodes écrites / testées sont statiques. Cela pourrait être le cas des méthodes `add()` et `divide()`.

a. Modifier les méthodes et les tests en conséquence.

b. Enregistrer, versionner.

[6] JUnit : https://junit.org/junit5/docs/current/user-guide/#overview-getting-started

9.- Écrire des tests paramétrés

a. Pour la méthode suivante :

```
public int add(int opG, int opD) ;
```

- A l'aide de l'annotation **`@ParameterizedTest`** de JUnit[7], ajouter un (nouveau) test paramétré puis exécuter ce test sur l'ensemble de valeurs simples ci-dessous, listées dans l'ordre opG, opD, ResultatAttendu :

```
0,    1,    1
1,    2,    3
-2,   2,    0
0,    0,    0
-1,   -2,   -3
```

- exécuter le test.

b. Enregistrer, versionner.

[7] JUnit : https://junit.org/junit5/docs/current/user-guide/#overview-getting-started
Utiliser https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests-sources-CsvSource

10.- Gestion d'ensembles

a. Ajouter à la classe **`Calculator`** la méthode suivante :

```
public Set[8]<Integer> ensembleChiffres(int pNombre)
```

qui, étant donné l'entier passé en paramètre, retourne l'ensemble (non ordonné) des chiffres composant ce nombre.

Exemples : `ensembleChiffres(7679)` retourne l'ensemble {6, 7, 9} (listés dans un ordre quelconque)
`ensembleChiffres(-11)` retourne l'ensemble {1}

b. Écrire et exécuter le/les tests associés

c. Enregistrer, versionner.

[8] Interface Set : https://docs.oracle.com/javase/8/docs/api/java/util/Set.html
et Classe HashSet : https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html

11.- Tester les méthodes contenant une levée d'exception

a. Pour chacune des méthodes dont les signatures suivent :

```
public int add(int opG, int opD) ;
public int divide(int opG, int opD) ;
```

- identifier une situation d'erreur donnant lieu à une levée d'exception
- modifier sa définition en vue de prendre en charge cette erreur[9]
- écrire un test associé en utilisant la bibliothèque **`jAssert`**
- exécuter le test.

b. Enregistrer, versionner.

[9] Java, hiérarchie des classes d'exceptions :
https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html
https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html
https://docs.oracle.com/javase/8/docs/api/java/lang/RuntimeException.html
https://docs.oracle.com/javase/8/docs/api/java/lang/ArithmeticException.html

12.- Contrôler la couverture de code du projet

La **couverture de code** est une mesure utilisée pour décrire le taux de code source exécuté d'un programme quand une suite de test est lancée. Un programme avec une haute couverture de code, mesurée en pourcentage, a davantage de code exécuté durant les tests ce qui laisse à penser qu'il a moins de chance de contenir de bugs logiciels non détectés, comparativement à un programme avec une faible couverture de code[10].

a. Rapport de couverture de code généré par IntelliJ & Gradle

- Exécuter les tests simultanément avec l'option 'with Coverage' (clic droit sur src/test/java puis Run 'Tests in Calculator with Covage' - Figure 5)

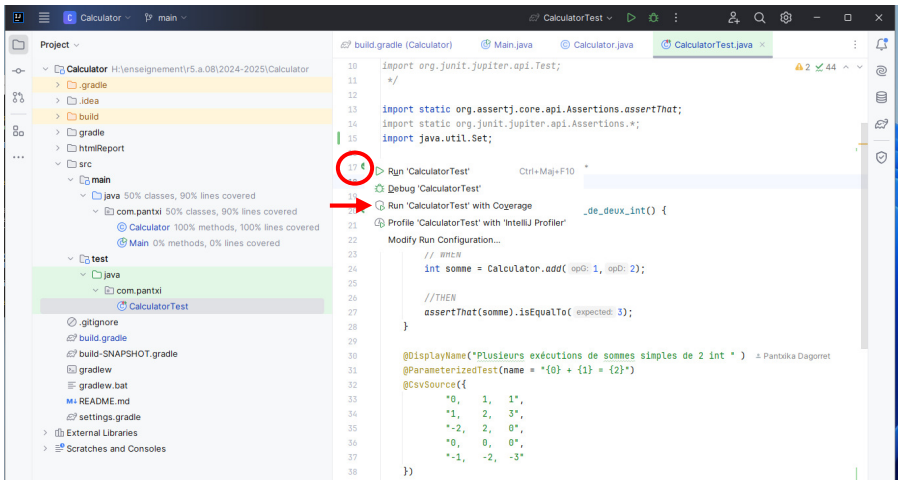


Figure 5 : Lancement d'un un test avec couverture de code

- Analyser les informations fournies par l'EDI : pourcentages de classes, de lignes, de méthodes, zones des fichiers sources couvertes / non couvertes par les tests (Figure 7)

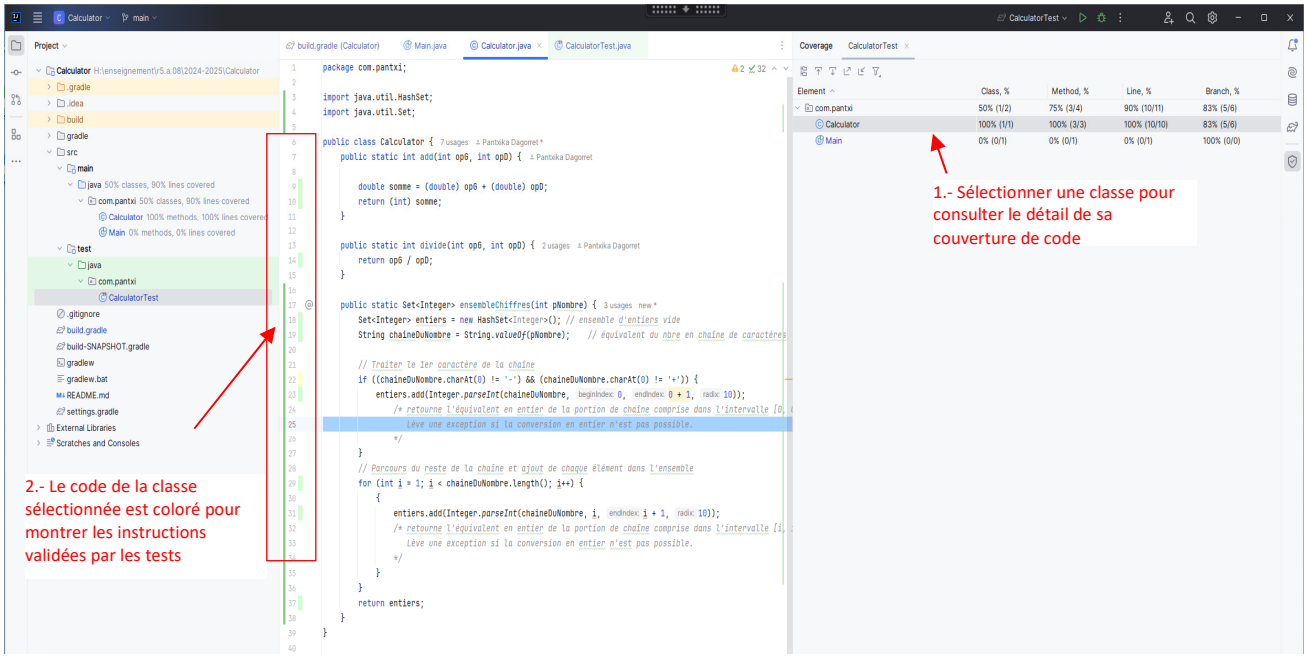


Figure 6 : Couverture de code illustrée directement sur l'EDI

[10] https://fr.wikipedia.org/wiki/Couverture_de_code

– IntelliJ peut générer un rapport de test et couverture de code sous la forme de fichier html. Le nom et le dossier de rangement du rapport sont modifiables (Figure 7, Figure 8)

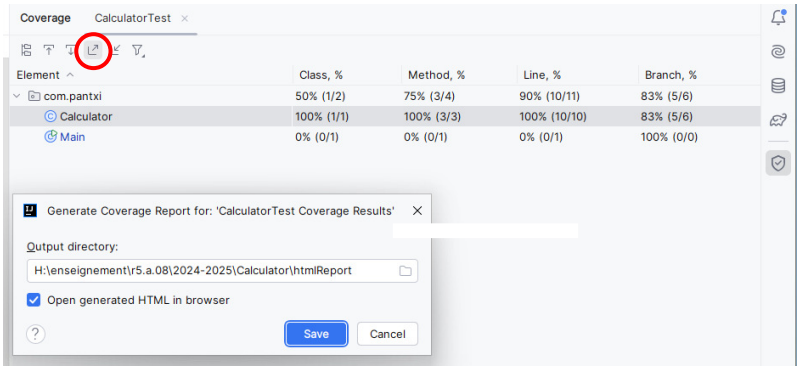


Figure 7 : Couverture de code générée par IntelliJ – reporting exporté au format HTML (1/2)

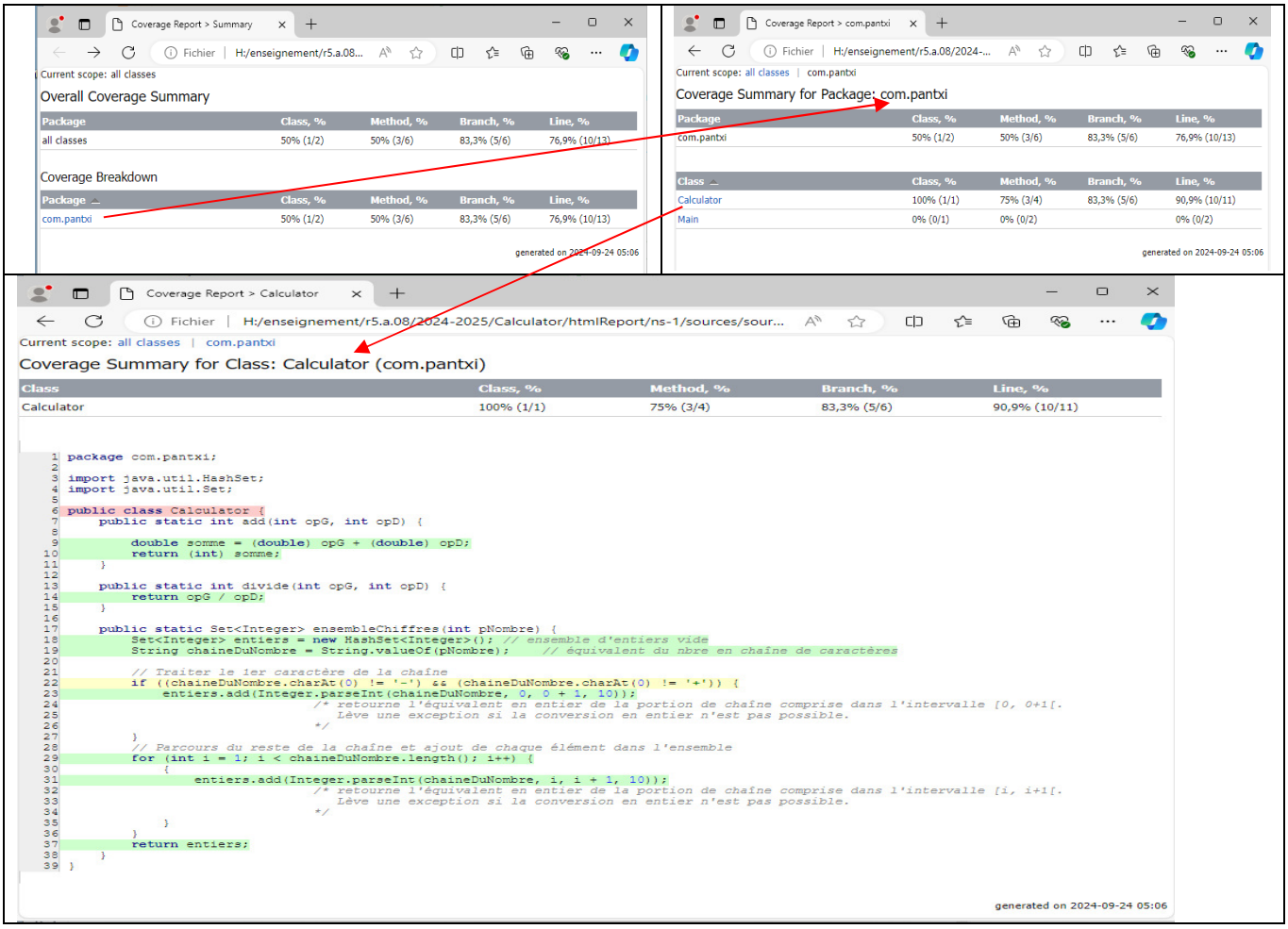


Figure 8 : Couverture de code générée par IntelliJ – reporting exporté au format HTML (1/2) (2/2)

b. Rapport de couverture de code .html généré par Jacoco

Le rapport de couverture de code peut aussi être généré par JaCoCo (Java Code Coverage), un outil spécialisé. Le rapport généré est au format html. Le nom et le dossier de rangement du rapport sont modifiables

- Compléter le fichier **`build.gradle`** avec les directives concernant :
  - Ajout de jacoco comme plugin de reporting (rubrique plugins : ajouter `id 'jacoco'`)
  - Finalisation du test par un rapport de test jacoco (rubrique test : ajouter `finalizedBy jacocoTestReport`)
  - Préciser le dossier où sera généré le rapport :

```
jacocoTestReport {
        dependsOn test
        reports {
                xml.required = true
                //par défaut html.outputLocation=layout.buildDirectory.dir('reports/jacoco/test')
                // mais modifiable, comme par exemple :
        html.outputLocation=layout.projectDirectory.dir('reports/jacoco/test')
        }
}
```

- Depuis l'IDE, exécuter à nouveau les tests de la classe avec l'option 'with Coverage' (clic droit sur src/test/java puis Run 'Tests in Calculator with Covage' - Figure 5).
  L'EDI montre les tests générés par IntelliJ, mais un rapport .html est également généré par jacoco dans le dossier demandé.

- Enregistrer, versionner.

c. Créer un test (script) Jacoco générant un rapport de couverture de code au format .html.

- Sélectionner le menu Run / Edit Configuration(cf. **Figure 9** à **Figure 12**).

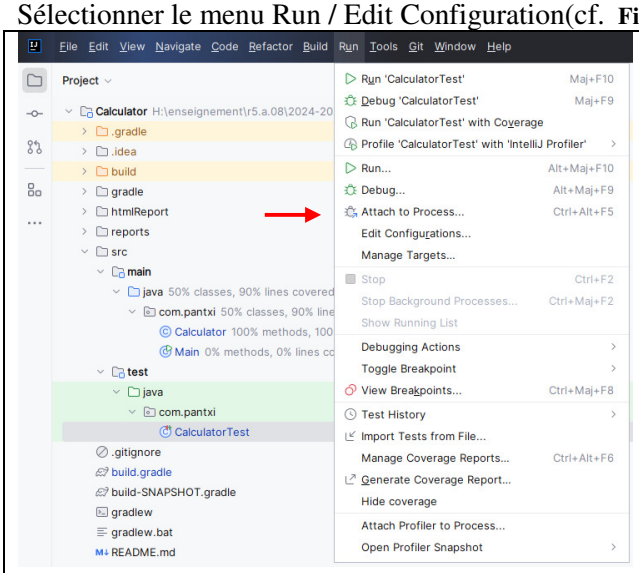


Figure 9 : Création d'un test avec rapport Jacoco (1/4)

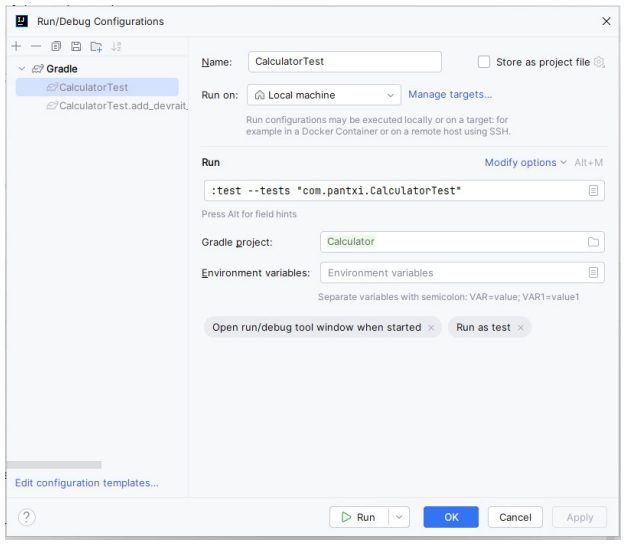


Figure 10 : Création d'un test avec rapport Jacoco (2/4)

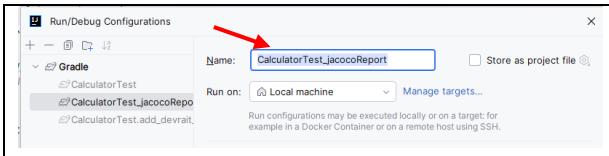


Figure 11 : Création d'un test avec rapport Jacoco (3/4)

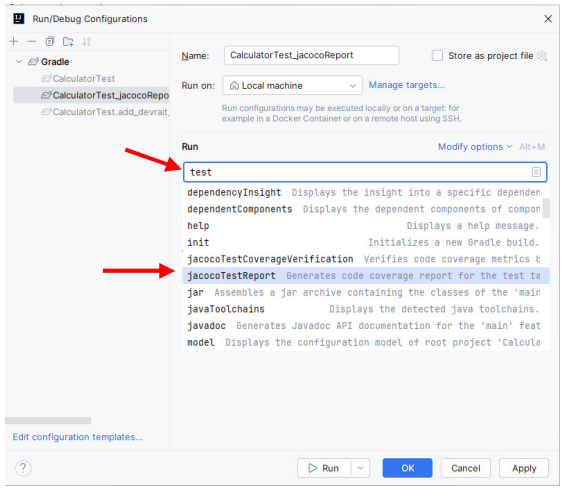


Figure 12 : Création d'un test avec rapport Jacoco (4/4)

– Exécuter le test avec rapport de couverture Jacoco créé (cf.)

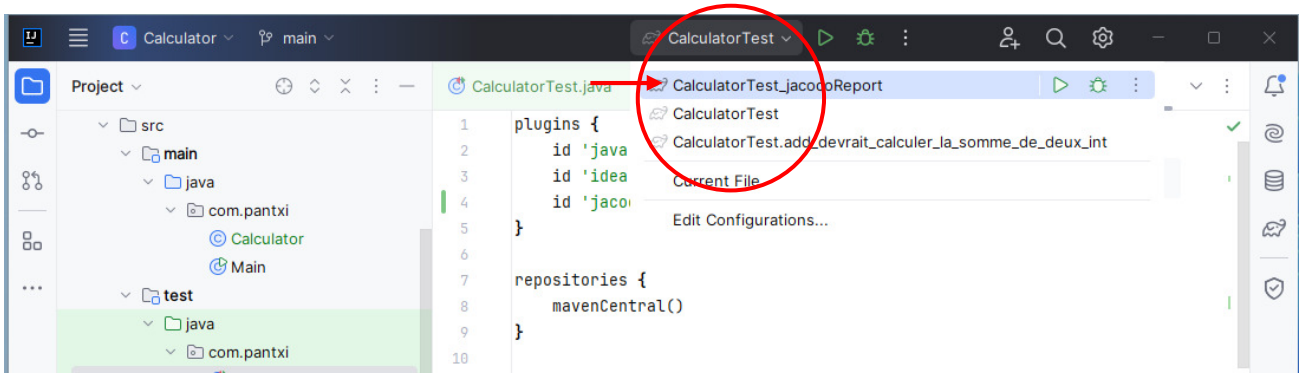


Figure 13 : Lancement du test créé

– Consulter les rapports générés par Jacoco dans le dossier nomDuProjet/reports (**Figure 14**)

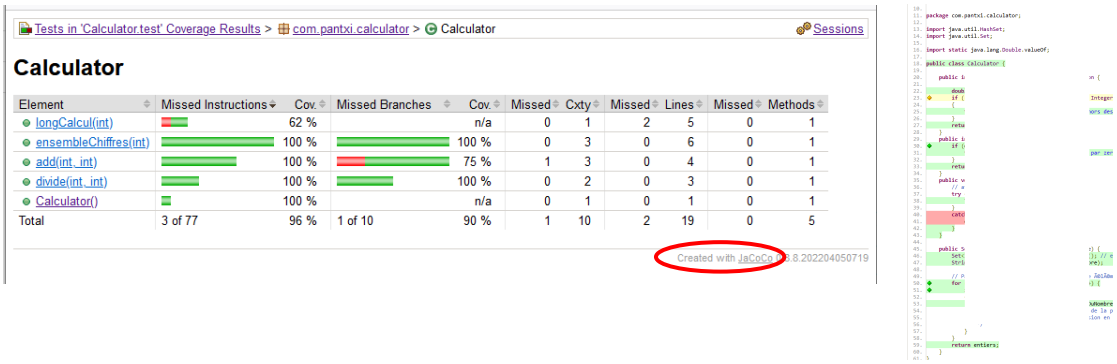


Figure 14 : Test avec Couverture de test – reporting fait par jacoco

– Enregistrer, versionner.

13.- Automatiser la construction du projet et l'exécution des tests lors du push : action Github

L'exécution fréquente des tests est nécessaire afin de s'assurer que tout nouvel ajout de fonctionnalité n'altère pas la validité du code déjà produit.

Afin de décharger le programmeur de cette tâche répétitive, il convient de l'automatiser. Cette pratique relève de ce que l'on appelle **Intégration Continue**[11].

Gradle est le **moteur de production**[12] utilisé dans notre projet par IntelliJ. Il organise toutes les étapes de construction du projet, dont l'exécution des tests et les mesures de couverture de code.

- Utilisation de Gradle depuis l'IDE : onglet Gradle (Figure 15) :

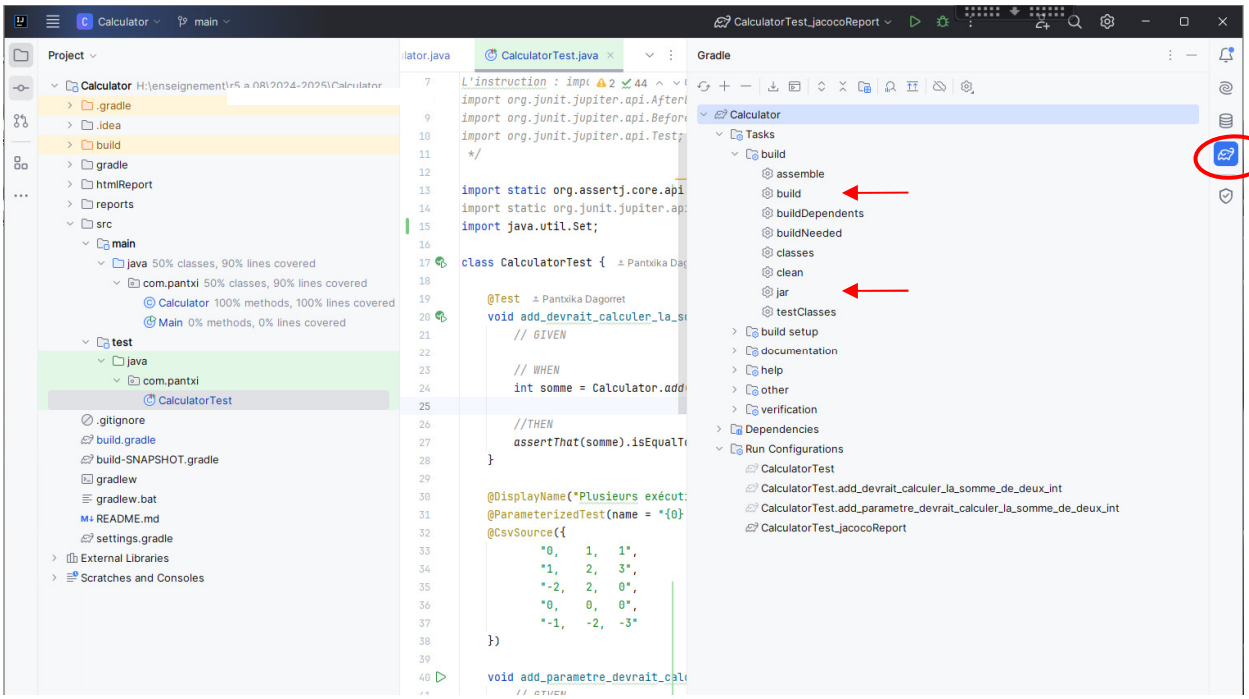


Figure 15 : Gradle sur l'IDE IntelliJ

- Utilisation en mode ligne de commande :

```
C:\Calculator> .\gradlew.bat test
C:\Calculator> .\gradlew.bat build
```

Il est possible de configurer Gradle afin que la construction du projet et exécution des tests soient exécutées **automatiquement lors de chaque push** vers le dépôt distant du projet (Github). Cela se fait grâce aux **actions Github**.

a. S'assurer (en ligne de commande ou via l'interface Gradle de l'IDE) que l'exécution des tests est correcte (les tests passent) et que la construction du projet produit bien un fichier .jar (dans le dossier build/libs/). **Attention** la variable d'environnement JAVA_HOME doit être configurée.

b. Créer une branche sur le dépôt local (par exemple github-action-build-tests-auto)

c. Dans cette branche, créer une action Github pour construire l'application et lancer les tests à chaque dépôt. Pour ce faire :

- Créer un dossier **`.github/workflows`**

[11] https://fr.wikipedia.org/wiki/Int%C3%A9gration_continue
[12] https://fr.wikipedia.org/wiki/Moteur_de_production

- Dans ce dossier, copier le fichier **`build.yml`** qui vous convient fourni sur eLearn en fonction du JDK que vous avez utilisé, en prenant soin de le renommer en **`build.yml`**
- Désactiver l'usage de Jacoco depuis IntelliJ (menu Run/Edit Configurations) → revenir à l'outil de reporting IntelliJ (cf. 12.- Contrôler la couverture de code)
- Pousser sur Github, et vérifier dans le volet Actions que l'action github s'est bien terminée. Noter que le fichier .jar n'est pas présent sur le dépôt distant. Pourquoi ?

d. Fusionner (merge) la branche lorsque l'action github s'est bien terminée.

a. Supprimer la branche. Enregistrer, versionner.

14.- Automatiser jusqu'à la production d'un rapport de tests consultable sur Github

a. Créer une branche sur le dépôt local (par exemple github-action-rapport-test-auto)

b. Ajouter l'étape suivante au fichier build.yml, utilisant l'action **test-reporter**.

```
- name: Rapport de tests
  uses: dorny/test-reporter@v1
  if: success() || failure()
  with:
      name: JUnit Tests
      path: build/test-results/test/TEST-*.xml[13]
      reporter: java-junit
```

c. Sauvegarder et pousser sur GitHub. Vérifier que le rapport de test est bien généré.

d. Fusionner (merge) la branche lorsque l'action github s'est bien terminée

e. Supprimer la branche

f. Enregistrer, versionner.

[13] Le test-reporter se nourrit du fichier résultat de test. Justifier le nom et l'emplacement de ce fichier.

15.- Automatiser jusqu'à la production d'un rapport de couverture de code (Jacoco) consultable sur Github

La mise en place de cette action comporte deux volets :

- Utilisation du plugin jacoco par Gradle, qui génère des rapports de couverture de code dans divers formats : .html, .xml, … pour être consultés soit par un humain, soit pour être traités automatiquement par un moteur de production. Cela supposera re-configurer le fichier build.gradle.
- Création d'une action github pour récupérer le rapport de couverture (au format .xml) et générer son équivalent au format .html sur Github.

a. Créer une branche sur le dépôt local (par exemple github-action-rapport-jacoco)

b. Modifier le fichier build.gradle[14] :
   - compléter la rubrique **plugin** pour ajouter le plugin jacoco à la liste des plugins installés
   - compléter la rubrique **test** : le test doit se finir par la création d'un rapport de test jacocoTestReport
   - créer une rubrique **jacocoTestReport** décrivant le rapport de test avec les informations suivantes :
     - xml.required = true
     - les rapport de couverture de code doivent être généré dans le dossier nomDuProjet/build/reports/jacoco/test.
   - Mettre à jour le fichier gradle (cliquer sur l'icone 'éléphant' apparaissant dans la zone client de la fenêtre du fichier build.gradle)

c. Dans le fichier build.yml : enlever le commentaire de la ligne : `run: ./gradlew build #JacocoTestReport`

d. Exécuter les tests avec l'option 'with Coverage', consulter le rapport (.html) généré par Jacoco

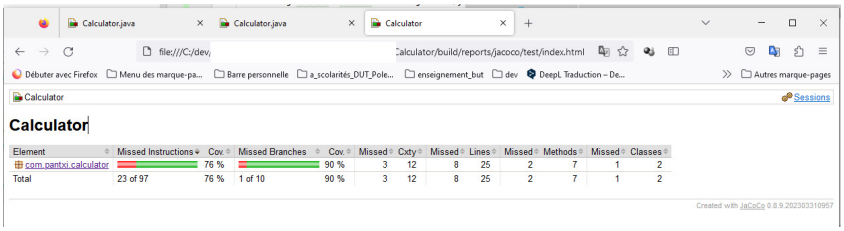
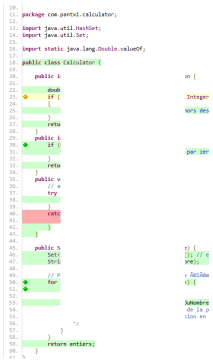


Figure 16 : Test avec Couverture de test – reporting fait par jacoco



e. Ajouter l'étape suivante au fichier build.yml, utilisant l'action **jacoco-reporter**[15].

```
- name: JaCoCo Code Coverage Report
  id: jacoco_reporter
  uses: PavanMudigonda/jacoco-reporter@v4.8
  with:
      coverage_results_path: build/reports/jacoco/test/jacocoTestReport.xml
      coverage_report_name: Coverage
      coverage_report_title: JaCoCo
      github_token: ${{ secrets.GITHUB_TOKEN }}
      skip_check_run: false
      minimum_coverage: 80
      fail_below_threshold: false
      publish_only_summary: false
```

f. Sauvegarder et pousser sur GitHub. Vérifier que le rapport de couverture de code est bien généré.

g. Fusionner (merge) la branche lorsque l'action github s'est bien terminée.

h. Supprimer la branche.

i. Enregistrer, versionner.

[14] https://docs.gradle.org/current/userguide/jacoco_plugin.html

[15] https://docs.gradle.org/current/userguide/jacoco_plugin.html
https://reflectoring.io/jacoco/

16.- Conditionner la construction du projet à la qualité de la couverture de code

La construction du projet est gérée par Gradle. Il faut donc modifier le fichier build.gradle et y ajouter une condition de fin de construction, de sorte à faire échouer la construction du projet si un certain niveau de couverture n'est pas atteint.

Vous pouvez vous inspirer des résultats obtenus au point 12.- précédent, ou bien fixer un seuil arbitraire pour vérifier que cette nouvelle fonctionnalité est opérationnelle. Dans l'illustration ci-dessous, le seuil a été fixé à 0,9.

```
C:\dev\xxx\Calculator>.\gradlew.bat build

> Task :test
CalculatorTest > longCalcul_devrait_durer_moins_d_1_seconde() PASSED
CalculatorTest > divide_devrait_lever_une_exception_quand_diviseur_est_0() PASSED
CalculatorTest > add_devrait_lever_une_exception_si_somme_hors_intervalle_des_int() PASSED
CalculatorTest > add_devrait_calculer_la_somme_de_deux_int() PASSED
CalculatorTest > Plusieurs tests de sommes simples de 2 int > 0 + 1 = 1 PASSED
CalculatorTest > Plusieurs tests de sommes simples de 2 int > 1 + 2 = 3 PASSED
CalculatorTest > Plusieurs tests de sommes simples de 2 int > -2 + 2 = 0 PASSED
CalculatorTest > Plusieurs tests de sommes simples de 2 int > 0 + 0 = 0 PASSED
CalculatorTest > Plusieurs tests de sommes simples de 2 int > -1 + -2 = -3 PASSED

> Task :jacocoTestCoverageVerification FAILED
[ant:jacocoReport] Rule violated for bundle Calculator: instructions covered ratio is 0.4, but expected
minimum is 0.9

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':jacocoTestCoverageVerification'.
> Rule violated for bundle Calculator: instructions covered ratio is 0.4, but expected minimum is 0.9

BUILD FAILED in 19s
6 actionable tasks: 1 executed, 5 up-to-date

C:\dev\xxx\Calculator>
```

a. Créer une branche sur le dépôt local (par exemple github-action-seuil-couverture)

b. Compléter le fichier build.gradle[16] avec une rubrique **jacocoTestCoverageVerification** + le seuil choisi, et une rubrique **check.dependsOn**.
Ne pas oublier de mettre à jour le fichier gradle (cliquer sur l'icone 'éléphant' apparaissant dans la zone client de la fenêtre du fichier build.gradle)

c. Exécuter le build par les moyens de votre choix (ligne de commande, onglet Gradle dans IDE, ou bien en poussant le code sur Github), et vérifier que la construction est bien stoppée.

d. Fusionner (merge) la branche lorsque l'opération est terminée.

e. Supprimer la branche.

f. Enregistrer, versionner.

[16] https://docs.gradle.org/current/userguide/jacoco_plugin.html
https://medium.com/codeops/code-coverage-with-gradle-and-jacoco-8b2e7d580d2a