Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Filière Informatique 2020-2021

Intégration des nombres complexes et des Unum en Java avec COJAC

Classe I3

Rapport

Travail de bachelor

Version : 0.3

Student :	Cédric Tâche
Professor :	Frédéric Bapst
Expert :	Baptiste Wicht

Table des versions

Version	Date	Author	Description
0.1	03.06.2021	Cédric Tâche	Création de la structure
0.2	07.06.2021	Cédric Tâche	Description de Maven et du problème de compilation
0.3	15.06.2021	Cédric Tâche	Complétion de l'analyse de COJAC Analyse des nombres complexes Spécification des nombres complexes

Table des matières

Chapitre 1

Introduction

Chapitre 2

COJAC

2.1 Bytecode JAVA

La **machine virtuelle JAVA** (**Java Virtual Machine** ou **JVM**) utilise un seul langage : le Bytecode JAVA. Bien qu'il soit possible de programmer en JAVA, en Kotlin ou encore en C pour nommer quelques exemples, la JVM n'utilise que du Bytecode.

Le nom de Bytecode provient du fait que chaque code d'opération fait exactement 1 byte. Ceci limite fortement le nombre de codes d'opération disponibles à 256. Certains codes d'opérations peuvent être suivis de paramètres précisant l'opération à effectuer. Voici deux exemples de Bytecode dont leur fonctionnement sera illustré plus loin. La première opération prend un byte en paramètre alors que la deuxième opération n'en prend pas.

```
bipush 5
iadd
```

Contrairement aux processeurs habituels, la JVM utilise une pile au lieu de registres. Certaines opérations permettent de charger des informations sur la pile ou d'y retirer un élément pour le stocker dans une variable. Toutes les autres opérations prennent comme entrée les valeurs sur la pile et ajoute le résultat sur cette même pile.

Voici un exemple de Bytecode qui permet de mettre deux integers sur la pile et de les additionner :

```
bipush 5
bipush 7
iadd
```

L'exécution de ce Bytecode est illustrée sur la figure 2.1. Pour cet exemple, la pile est vide avant d'exécuter ces commandes. L'exécution se déroule en plusieurs étapes :

1. Le nombre 5 est ajouté sur la pile.
2. Le nombre 7 est ajouté sur la pile.
3. L'addition consomme les deux nombres et ajoute le résultat sur la pile.

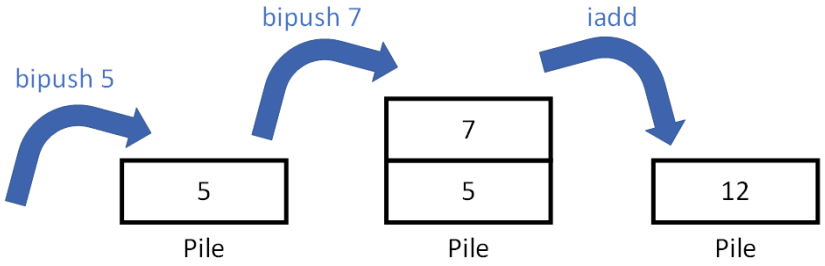

FIGURE 2.1 – Addition de deux nombres en Bytecode JAVA

2.2 Agent JAVA

Un agent Java est un programme Java spécial conçu pour modifier le comportement d'un autre programme sans en modifier le coude source. Comme montré dans la figure 2.2, l'agent JAVA intercepte les fichiers .class qui contiennent du Bytecode et peut y apporter des modifications.

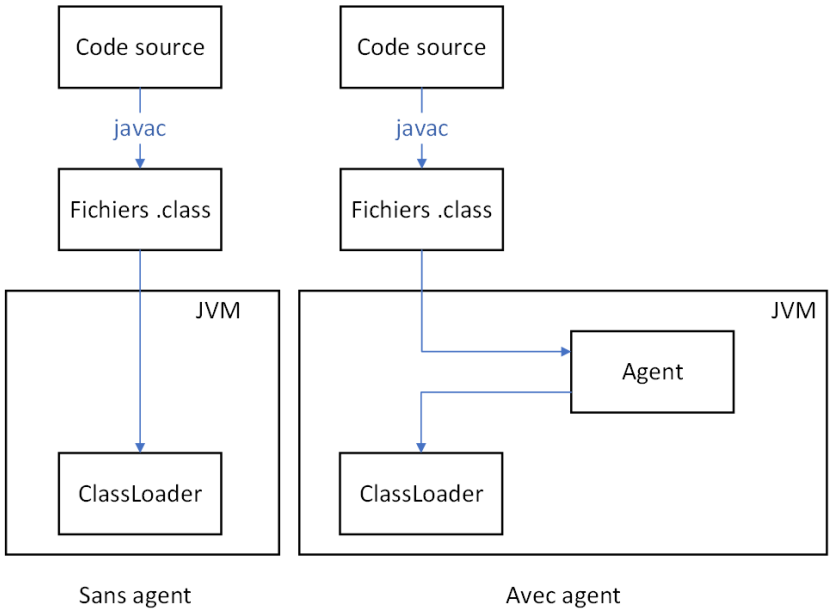

FIGURE 2.2 – Chargement d'une classe

Cette fonctionnalité est particulièrement utile pour des programmes pouvant fonctionner avec beaucoup d'autres applications. Ils peuvent offrir, par exemple, du monitoring, du profilage, etc.

Il existe deux types d'agents qui seront appelés, dans ce document, agent statique et agent dynamique.

2.2.1 Agent statique

Un agent statique est un agent qui est spécifié au démarrage du programme. Ainsi, lors du démarrage de l'application, l'agent et la JVM intéragissent ensemble conformément à la figure 2.3. Cette intéraction se déroule en plusieurs étapes :

1. La JVM appelle l'agent en lui donnant le paramètre spécifié (un string) lors du démarrage de l'application. La méthode appelée s'appelle *premain* parce qu'elle est exécutée avant le *main* de l'application cible.
2. Dans cette méthode *premain*, l'agent doit créer un *ClassFileTransformer*. Cet objet sera utilisé plus tard pour modifié les classes. Cet objet est ensuite donné à la JVM grâce à la méthode *Instrumentation.addTransformer*.
3. Ensuite à chaque fois que la JVM veut charger une nouvelle classe, le *ClassLoader* lira le fichier *.class* correspondant.
4. Le *ClassLoader* enverra ensuite le tableau de bytes qu'il vient de lire au *ClassFileTransformer* qui pourra y apporter les modifications désirées.
5. Une fois que le *ClassFileTransformer* a modifié la classe, il la retournera la classe à la JVM.

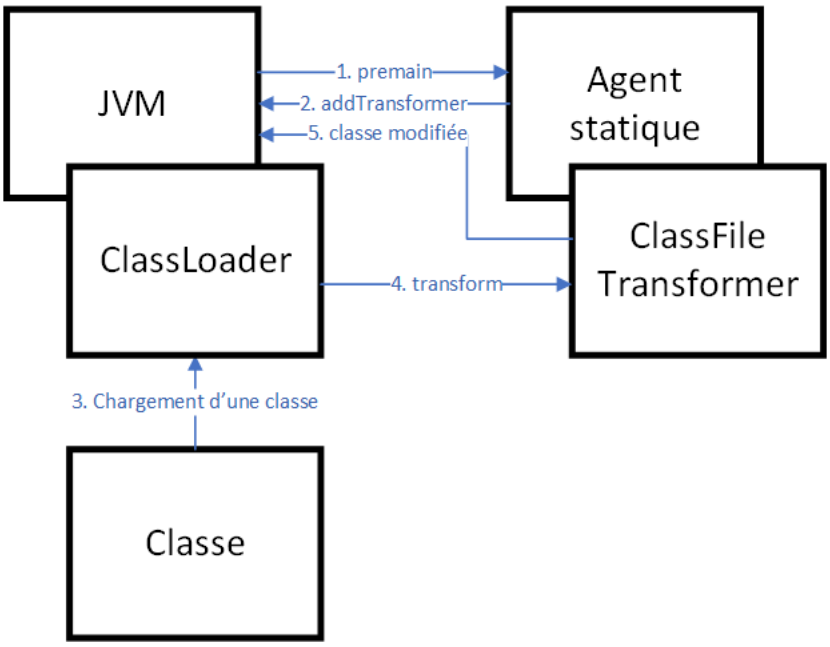

FIGURE 2.3 – Fonctionnement d'un agent statique

2.2.2 Agent dynamique

Un agent dynamique est un agent qui est ajouté après le démarrage de l'application cible. Ceci rend la modification du programme plus complexe et nécessite une autre application pour insérer l'agent. L'ajout de l'agent se déroule en plusieurs étapes comme montré sur la figure 2.4 :

1. Une application pour ajouter l'agent doit être créé. Cette application doit s'attacher à la JVM du processus dans lequel l'agent doit être inséré. Cette opération peut se faire que si l'application cible et l'application source sont sur des processus possédés par le même utilisateur pour des raisons de sécurité.
2. L'application source charge ensuite l'agent dans l'application cible.
3. Contrairement à l'agent statique, la JVM appelle une autre méthode nommée *agentmain*.
4. L'agent enregistre ensuite son *ClassFileTransformer*.
5. Cette étape est facultative. Cependant, comme l'application cible a démarrée avant l'agent, elle a déjà chargée certaines classes. La méthode *Instrumentation.retransformClasses* permet de transformer les classes déjà chargées. Cependant, il y a quelques restrictions. Pour chaque classe à retransformer, les étapes 6 à 9 seront à nouveau exécutées.
6. Ensuite à chaque fois que la JVM veut charger une nouvelle classe, le *ClassLoader* lira le fichier *.class* correspondant.

7. Le *ClassLoader* enverra ensuite le tableau de bytes qu'il vient de lire au *ClassFileTransformer* qui pourra y apporter les modifications désirées.
8. Une fois que le *ClassFileTransformer* a modifié la classe, il la retournera la classe à la JVM.

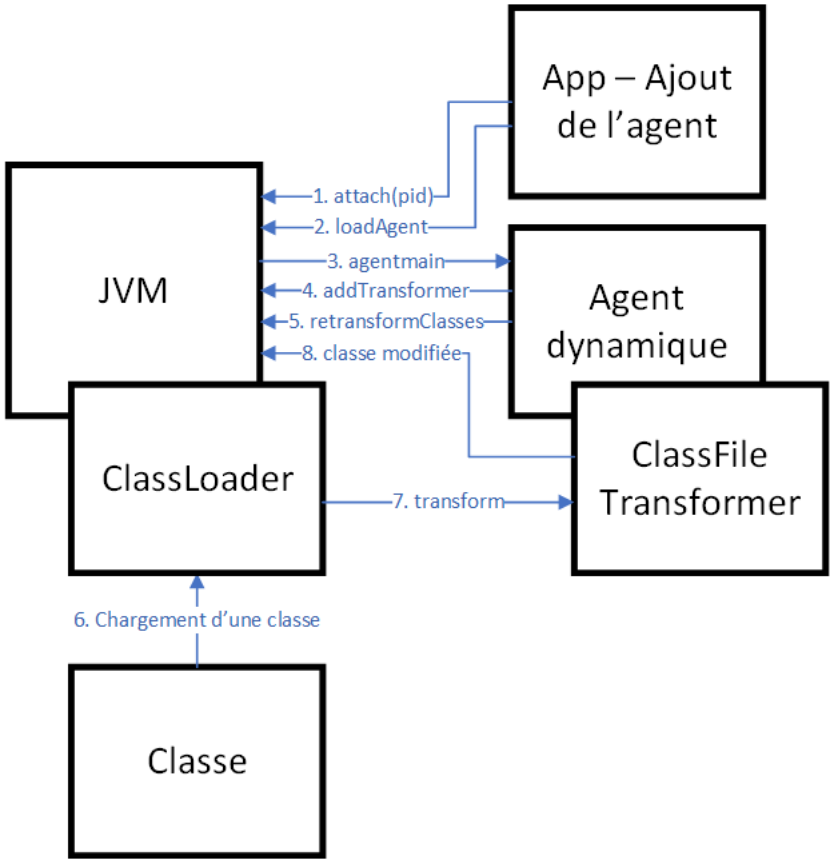

FIGURE 2.4 – Fonctionnement d'un agent dynamique

2.2.3 Synthèse

Un agent permet de modifier le comportement de l'application cible sans en modifier le code source. Cette fonctionnalité permet de faire du profilage, de surveiller des applications ou de modifier le comportement d'une libraire propriétaire.

Les agents statiques et dynamiques ont des objectifs différents. Voici leurs avantages et inconvénients respectifs.

Les agents statiques possèdent les caractéristiques suivantes :
- Le changement de l'agent statique nécessite de redémarrer l'application cible.

Les agents dynamiques possèdent les caractéristiques suivantes :
- La retransformation de classes a d'importantes limites jusqu'en Java 14.

— L'agent peut être ajouté après que l'application ait démarré. Cette fonctionnalité permet, par exemple, de pouvoir obtenir des informations sur une application ayant un bug rare, inconnu ou qui se produit après un certain temps d'activité.

2.3 Intégration

COJAC [1] propose deux manières d'intégrer une nouvelle fonctionnalité :

— **Behaviour** : les opérations sur les floats et les doubles sont simplement remplacées par un appel de méthode. Ainsi, il est possible de changer le comportement de ceux-ci. Dans ce projet, il serait possible de mettre la partie réelle et la partie imaginaire dans un double et de modifier les opérations qui les utilisent.

— **Wrapper** : les floats et les doubles peuvent être remplacés par un objet (wrapper). Ce qui permet d'ajouter plus d'éléments dans le wrapper.

Le Behaviour a les caractéristiques suivantes :

— Le stockage est limité en taille (8 octets pour un double).

— Moins de modifications sont nécessaires. Toutes les opérations sur les doubles et les appels de la méthodes de la librairie standard doivent être modifiés. Il est également possible de convertir les floats en doubles. A ce moment, il y a beaucoup plus de modifications.

Le Wrapper a les caractéristiques suivantes :

— Le stockage est illimité.

— Beaucoup de modifications doivent être effectuées. Toutes les constantes, signatures de méthode, opérations, etc. doivent être adaptées parce qu'un type de base et un objet sont radicalement différents.

2.4 Maven

COJAC [1] utilise Maven [2] pour créer le JAR. Maven [2] est un outil d'automatisation pour gérer les dépendances et produire une application. Cet outil peut télécharger les dépendances, compiler le projet, exécuter les tests unitaires et d'intégration et déployer le projet. Maven et Gradle sont les deux outils les plus souvent utilisés pour gérer les projets Java. Toute la configuration se trouve dans le fichier *pom.xml*.

2.4.1 Debug

Lorsqu'il y a un problème avec Maven, il y a plusieurs paramètres qui permettent d'obtenir plus d'informations. Cette section montre comment configurer ces paramètres sous IntelliJ et donne l'argument équivalent pour appeler Maven en ligne de commande.

Ouvrir la fenêtre Maven

Toutes les configurations effectuées ci-dessous seront effectuées depuis la fenêtre Maven dans IntelliJ. Le bouton pour l'ouvrir se situe sous **View** → **Tool Windows** → **Maven** comme montré sur la figure 2.5.

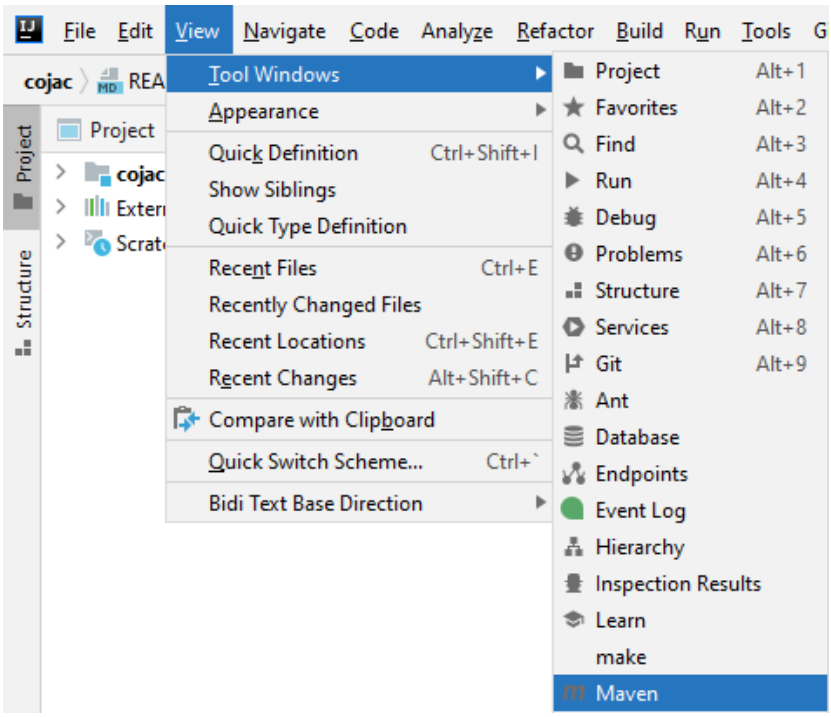

FIGURE 2.5 – Bouton d'ouverture de la fenêtre Maven

Fenêtre Maven

La fenêtre Maven montre le contenu du projet avec les étapes du cycle de vie de la production de l'application comme le montre la figure 2.6. Le bouton encadré sur l'image permet d'ouvrir les paramètres de Maven qui seront utilisés plus tard. On peut également voir les plugins et les dépendances. Cette fenêtre permet aussi d'exécuter les étapes pour produire l'application. L'étape *package* est suffisante pour générer le JAR.

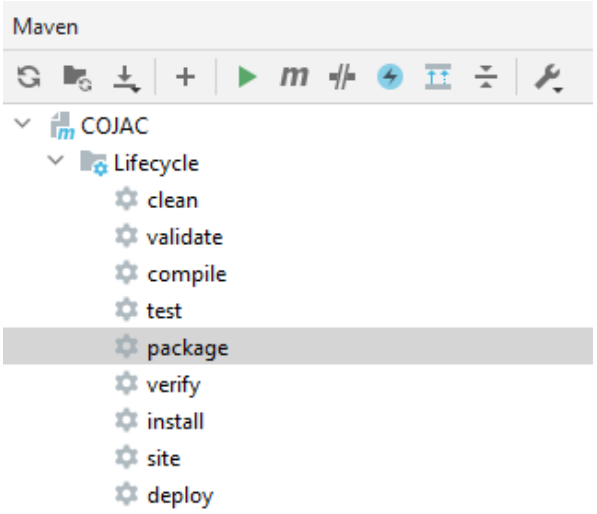

FIGURE 2.6 – Fenêtre Maven

Une option peut être ajoutée pour afficher les messages de debug. Cette option est plus souvent utilisée lors du développement de Maven ou d'un plugin Maven. Cependant, elle peut également être utile pour trouver la source d'un problème difficile. **TODO: image** . L'option en ligne de commande équivalente se nomme *--debug*.

Chapitre 3

Intégration des nombres complexes

3.1 Nombres complexes

Au tout début, les nombres étaient limités aux entiers positifs. L'humanité y a progressivement inclus des nombres négatifs, puis des nombres réels. Cependant, bien que ces nombres suffisent en général, d'autres ensemble de nombres plus larges existent également comme les nombres complexes qui sont le sujet de ce chapitre.

3.1.1 Avantages

Bien que ces nombres complexes étaient considérés comme une astuce pour résoudre des problèmes ayant des racines carrés de nombres négatifs, ces nombres sont désormais très souvent utilisés en mathématique, en physique et en ingénierie. Ils permettent de simplifier l'écriture de nombreuses formules et sont suffisant pour décrire une grande majorité des lois et phénomènes physiques.

Une autre raison de leur usage est que le corps des nombres complexes est algébriquement clos. C'est-à-dire que chaque polynôme complexe de degré 1 ou supérieur a au moins une racine complexe. Ainsi, il n'y a pas les mêmes risques que pour les nombres réels où chaque calcul de racine doit être fait avec précautions sans quoi, il pourrait n'exister aucune solution dans le domaine réel. Avec les nombres complexes, il est garanti qu'une racine existera nécessairement.

Les nombres complexes ont eux aussi des limites, même si on ne les rencontre que rarement. C'est seulement dans des cas spécifiques, tel que la physique quantique avec un spin, que les nombres complexes se révèlent insuffisants.

3.1.2 Désavantages

Les nombres complexes ont aussi un désavantage : la perte de comparaison. Comme les nombres réels peuvent être représentés sur un axe, il est toujours possible de définir si un nombre est plus petit ou plus grand qu'un autre. Cette même comparaison n'a plus lieu d'être avec les nombres complexes, car il ne s'agit plus d'un axe, mais d'un espace à 2 dimensions. Selon les situations, les nombres complexes peuvent être comparés par leur partie réelle, par leur partie imaginaire, par leur module ("la taille" de ce nombre), etc.

3.1.3 Représentations

Les nombres complexes peuvent être écrits sous différentes formes :

- Algébrique : $a + bi$ où a est la partie réelle et bi est la partie imaginaire. Ex : $2 + 3i$
- Trigonométrique : $r(cos(\theta) + i * sin(\theta))$
- Exponentielle : $re^{i\theta}$
- Polaire : (r, θ)

3.2 Spécifications

L'intégration des nombres complexes doit permettre de pouvoir réaliser toutes les opérations arithmétiques standards comme l'addition ou la multiplication. **TODO: Opération de comparaison ?** . Il y aura deux méthodes magiques pour obtenir la partie réelle et imaginaire des nombres complexes. Lorsqu'un nombre complexe sera converti en string, il sera affichée sous sa forme algébrique. Les opérations communes de la librairie standard telles que *Math.sqrt*, *Math.sin*, etc. devront aussi être adaptées.

3.3 Démonstration

Le code suivant sera utilisé pour montrer l'intérêt des nombres complexes. Il permettra aussi de les valider. Le code suivant permet de trouver une racine d'un polynôme du 3e degré. Les 2 polynômes ont des racines réelles et pourtant, le polynôme $x^3 - 2x^2 - 13x - 10$ produit un *NaN* (not a number) lors du calcul de la racine. Si le calcul est effectué avec des nombres complexes, une des racines devrait être trouvée.

```java
// Find a root of a cubic equation of the form ax^3 + bx^2 + cx + d = 0
↪  with the general cubic formula
// This formula can be found on wikipedia:
↪  https://en.wikipedia.org/wiki/Cubic_equation#General_cubic_formula
static double findRootOfCubicEquation(double a, double b, double c,
↪  double d) {
    double det0 = b * b - 3 * a * c;
    double det1 = 2 * b * b * b - 9 * a * b * c + 27 * a * a * d;

    double sqrt = Math.sqrt(det1 * det1 - 4 * det0 * det0 * det0);
    // the root can't be calculated if this value is not available.
    if (Double.isNaN(sqrt)) return Double.NaN;

    double coef = Math.cbrt((det1 + sqrt) / 2);

    if (coef == 0) coef = Math.cbrt((det1 - sqrt) / 2);
    if (coef == 0) return -b / (3 * a);
    return -(b + coef + det0 / coef) / (3 * a);
}

public static void main(String[] args) {
    System.out.println(findRootOfCubicEquation(2, 1, 3, 1) + " should
    ↪  be ~0.66666...");
    System.out.println(findRootOfCubicEquation(1, -2, -13, -10) + "
    ↪  should be -1, -2 or 5");
}
```

Chapitre 4

Problèmes rencontrés

4.1 Surefire provoque une exception lors de l'exécution des tests par Maven

Surefire provoquait une erreur lors de l'exécution des tests lorsqu'un espace était présent dans le chemin d'accès.

4.1.1 Problème

Lors de la création du JAR, l'erreur suivante se produit lors de l'exécution des tests :

```
ExecutionException The forked VM terminated without properly saying
↪  goodbye. VM crash or System.exit called?
```

Le message suivant présent dans les logs indique la localisation des fichiers qui contiennent plus de détails.

```
[ERROR] Please refer to D:\Documents\Documents\HEIA\Semestre 6 -
↪  2021\Bachelor\Projet\cojac\target\surefire-reports for the
↪  individual test results.
```

4.1.2 Solution temporaire : ne pas exécuter les tests

Comme les problèmes ont uniquement lieu durant les tests, il est possible de générer le JAR sans les exécuter. **TODO: lien section Maven** Pour ce faire, il faut cliquer sur le bouton dédié dans la fenêtre Maven. Ce bouton est visible sur la figure 4.1.

Cependant, cette solution ne doit être utilisée que s'il y a besoin du JAR pour d'autres raisons. Il est nécessaire de corriger ce problème pour garantir la qualité du code.

FIGURE 4.1 – Bouton pour ignorer les tests

4.1.3 Cause

Dans un des fichiers de log produits, qui sont mentionnées dans la section 4.1.1, on peut trouver l'erreur suivante :

```
# Created at 2021-06-03T14:09:01.042
Error opening zip file or JAR manifest missing :
↪  D:\Documents\Documents\HEIA\Semestre
```

Ce lien est incomplet par rapport au chemin mentionné dans la section 4.1.1. Le chemin d'accès s'arrête lors du premier espace trouvé. Si le projet est déplacé dans un dossier ne contenant aucun espace, le JAR est correctement généré.

4.1.4 Solution

Il faut modifier une ligne de configuration du plugin Surefire dans le *pom.xml*. Des guillemets ont été ajoutés autour du chemin d'accès du JAR pour obtenir la ligne suivante.

```
1  <argLine>
2      -javaagent:"${basedir}/src/test/resources/cojac-agent-test.jar"
3  </argLine>
```

Ce changement corrige le problème.

4.2 Erreur lors de l'instrumentation d'une application par COJAC

Lors de l'appel de la commande pour instrumenter une démonstration avec COJAC [1], COJAC [1] provoque une erreur.

4.2.1 Problème

L'application peut être démarrée sans COJAC [1] avec la commande suivante :

```
java demo\HelloBigDecimal.java
```

Cependant, COJAC [1] produit une erreur lorsque l'application est démarrée avec COJAC [1]. La commande suivante a été utilisée :

```
java -javaagent:cojac.jar="-Rb 50" demo\HelloBigDecimal.java
```

4.2.2 Solution temporaire : compiler avec Java 8

COJAC [1] doit être compilé avec Java 9 au minimum, mais l'application instrumentée doit être compilée avec Java 8. Les deux commandes suivantes permettent de compiler une démonstration et de l'exécuter :

```
javac -source 8 -target 8 demo\HelloBigDecimal.java
java -javaagent:cojac.jar="-Rb 50" demo.HelloBigDecimal
```

4.2.3 Cause

La source du problème provient de l'instruction Bytecode *Invokedynamic* qui a évolué en Java 9 et qui est désormais utilisé pour de nouvelles opérations qui ne sont pas supportés par COJAC [1] à l'heure actuelle.

Chapitre 5

Références

[1] COJAC. Cojac - boosting arithmetic capabilities of java numbers. `https://github.com/Cojac/Cojac`, 2019.

[2] Brett Porter, Jason van Zyl, and Olivier Lamy. Welcome to apache maven. URL `https://maven.apache.org/`.

Annexes