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

Filière Informatique 2020-2021

Intégration des nombres complexes et des Unums en Java avec COJAC

Classe I3

Rapport

Travail de bachelor

Version : 1.0

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
0.4	20.06.2021	Cédric Tâche	Conception des nombres complexes
0.5	29.06.2021	Cédric Tâche	Implémentation et tests des nombres complexes
0.6	06.07.2021	Cédric Tâche	Analyse et Conception des unums
0.7	11.07.2021	Cédric Tâche	Description d'une nouvelle alternative pour l'implémentation des unums
1.0	16.07.2021	Cédric Tâche	Finalisation de l'intégration des unums Ajout d'un glossaire, d'une liste des versions et d'une conclusion Amélioration de la mise en page

Table des matières

1 Introduction **1**

1.1 Contexte 1

1.2 Objectifs 2

1.2.1 Intégration des nombres complexes 2

1.2.2 Intégration des Unums 2

1.2.3 Démonstration des deux fonctionnalités 3

1.3 Objectifs secondaires 3

1.3.1 Mise à jour des librairies 3

1.3.2 Tests de performance 3

1.3.3 Documentation et promotion 3

1.3.4 Comparaison des approches Wrappers et Behaviours 3

1.3.5 Améliorations diverses 4

1.4 Structure du rapport 4

2 COJAC **5**

2.1 Agent Java 5

2.1.1 Agent statique 6

2.1.2 Agent dynamique 7

2.1.3 Synthèse 8

2.2 Bytecode Java 9
2.3 Intégration 10
2.3.1 Ajout d'un wrapper 11
2.3.2 Limitations 11
2.3.3 Méthodes magiques 12
2.4 Maven 12

3 Intégration des nombres complexes 15

3.1 Nombres complexes 15
3.1.1 Avantages 15
3.1.2 Désavantages 16
3.1.3 Représentations 16
3.2 Spécifications 17
3.3 Démonstration 17
3.4 Conception 18
3.4.1 Approche 18
3.4.2 Comparaison 19
3.4.3 ToString and fromString 19
3.4.4 Modes 20
3.4.5 Librairie disponible 22
3.4.6 Diagramme de classes 22
3.5 Implémentation 23
3.5.1 Wrapper 23
3.5.2 Remplacement de méthodes 24
3.5.3 Ajout de l'option 25
3.5.4 *toString* et *fromString* 26

3.6 Tests . . . 27
3.6.1 Tests unitaires . . . 27
3.6.2 Tests d'intégration . . . 27
3.7 Documentation . . . 27
3.8 Résultats . . . 28

4 Intégration des Unums **29**

4.1 Format de stockage des nombres réels . . . 29
4.1.1 Virgule fixe . . . 29
4.1.2 Virgule flottante . . . 30
4.1.3 Unums . . . 30
4.2 Spécifications . . . 30
4.3 Démonstration . . . 31
4.4 Conception . . . 32
4.4.1 Librairie . . . 32
4.4.2 Approche . . . 32
4.4.3 Passerelle vers le code natif . . . 34
4.5 Implémentation . . . 34
4.5.1 Passerelle JNI . . . 35
4.5.2 Build de la librairie . . . 37
4.5.3 Chargement de la librairie . . . 38
4.5.4 Autres . . . 39
4.6 Tests . . . 40
4.6.1 Tests unitaires . . . 40
4.6.2 Tests d'intégration . . . 40
4.7 Résultats . . . 40

5 GitLab CI **41**

6 Problèmes rencontrés **43**

6.1 Surefire provoque une exception lors de l'exécution des tests par Maven 43

6.1.1 Problème 43

6.1.2 Solution temporaire : ne pas exécuter les tests 44

6.1.3 Cause 44

6.1.4 Solution 44

6.2 Erreur lors de l'instrumentation d'une application par COJAC 45

6.2.1 Problème 45

6.2.2 Solution temporaire : compiler avec Java 8 45

6.2.3 Cause 45

6.3 Erreur des tests unitaires 45

6.3.1 Problème 46

6.3.2 Cause 46

6.3.3 Solution 46

6.3.4 Remarques 47

6.4 Erreur de compilation de *SoftPosit* 47

6.4.1 Problème 47

6.4.2 Cause 48

6.4.3 Solution 48

6.5 Inclure *SoftPosit* lors de la compilation de la librairie native 49

6.5.1 Problème 49

6.5.2 Cause 49

6.5.3 Solution 49

7 Versions **51**

7.1 Conclusion 51
7.1.1 Atteinte des objectifs 51
7.1.2 Perspectives d'amélioration 52
7.1.3 Conclusion personnelle 53
7.1.4 Déclaration d'honneur 53

8 Glossaire **56**

Chapitre 1

Introduction

La plupart des langages de programmation offrent des capacités similaires pour stocker des nombres et effectuer des calculs. Ces langages permettent, entre autres, d'utiliser des nombres réels.

Ces nombres réels ont tout de même des limitations. Ils sont limités au domaine du réel. De plus, les nombres à virgule flottante sont souvent inexacts. Par conséquent, lorsque les erreurs s'accumulent, le résultat d'un calcul peut être très éloigné de la réponse exacte.

Pourtant, d'autres alternatives existent. Les nombres complexes sont couramment utilisés en mathématique et en physique. Ils peuvent être utilisés pour simplifier des réponses utilisant des racines de nombres négatifs ou pour combiner deux quantités réelles telles que la tension et l'intensité en électricité. Quant à eux, les nombres réels peuvent aussi être représentés différemment. Les **universal numbers** (**Unums**) sont une représentation alternative possible dont la dernière version est expliquée dans l'article *Beating Floating Point at its Own Game : Posit Arithmetic* [9].

1.1 Contexte

En Java, aucun type ni aucune classe dans le JDK permet de gérer des nombres complexes ou des Unums, mais il est possible de l'implémenter soi-même. Des librairies pouvant gérer ces éléments peuvent déjà exister.

La première solution pour ajouter ces fonctionnalités et de créer de nouvelles classes : une classe pour les nombres complexes et une classe pour les Unums. Ensuite, il faut écrire le code en utilisant directement ces classes. Pour changer le type de calculs effectué (ex : nombre complexe → nombre réel), il faut changer le code source de l'application.

COJAC [3] est une librairie Java permettant de modifier les capacités arithmétiques d'un programme Java sans en modifier le code. Elle utilise l'API d'instrumentation et peut trans-

former les classes et méthodes au runtime pour changer le type de calcul effectué. Ainsi, pour changer le type de calculs effectué (ex : nombre réel → nombre complexe), il faut seulement changer l'argument donné à COJAC lors du démarrage de l'application. Ceci ne demande aucune modification dans l'application de l'utilisateur.

1.2 Objectifs

Le but de ce projet est d'ajouter deux nouvelles fonctionnalités à COJAC. COJAC devra permettre de remplacer automatiquement les nombres à virgules flottantes par deux nouveaux types numériques.

1.2.1 Intégration des nombres complexes

Une option de COJAC permettra de changer le comportement des calculs dans l'application de l'utilisateur. Les nombres à virgules flottantes (*float* et *double*) seront remplacés, au runtime, par des nombres complexes. Les opérations arithmétiques telles que l'addition, la soustration, etc. devront être adaptées. De plus, les méthodes souvent utilisées de la librairie standard devront également pouvoir fonctionner. Par exemple, la méthode *Math.sqrt* devra permettre de retourner la racine carrée d'un nombre négatif.

Voici un exemple sans les nombres complexes :

```
double val = Math.sqrt(-1); // = NaN
val = val * val; // = NaN;
```

Avec les nombres complexes, on obtient le résultat suivant :

```
double val = Math.sqrt(-1); // = i
val = val * val; // = -1;
```

1.2.2 Intégration des Unums

Une option de COJAC permettra de changer le format de stockage et de calculs des nombres réels. Les Unums seront utilisés à la place de la virgule flottante. Par conséquent, les opérations arithmétiques devront être redéfinies pour fonctionner avec ce nouveau format de stockage. Il faudra probablement utiliser JNI pour accéder à une librairie C/C++ permettant d'utiliser les Unums, mais d'autres approches restent possibles.

1.2.3 Démonstration des deux fonctionnalités

Des programmes de démonstrations seront réalisés pour montrer ces deux fonctionnalités. Ces démonstrations doivent montrer l'utilité et les avantages de cette approche.

1.3 Objectifs secondaires

D'autres ajouts de fonctionnalités ou modifications permettraient d'améliorer ce projet.

1.3.1 Mise à jour des librairies

COJAC utilise plusieurs librairies dont les versions sont désormais obsolètes. Il vaut mieux mettre à jour les versions avant de rencontrer des problèmes à cause de versions trop anciennes. Cependant, COJAC devra garantir une compatibilité pour Java 8+.

1.3.2 Tests de performance

Lorsque les fonctionnalités de remplacement des nombres à virgule flottante par des nombres complexes et des Unums, il restera encore un aspect inconnu qui est pourtant important pour décider de l'utilité de cette fonctionnalité : les performances. Pour cette raison, des tests de performance peuvent aussi être ajoutés pour tester l'efficacité de l'implémentation.

1.3.3 Documentation et promotion

COJAC possède une documentation pour l'utiliser et des vidéos pour expliquer l'utilité de certaines fonctionnalités. Il serait possible de documenter les nouvelles fonctionnalités ajoutées, de réaliser une vidéo pour montrer l'utilité de ces vidéos ou encore de compléter la documentation actuelle.

1.3.4 Comparaison des approches Wrappers et Behaviours

Deux approches sont possibles pour implémenter de nouvelles fonctionnalités dans COJAC :

— 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.

1.3.5 Améliorations diverses

Toute autre amélioration à la base de code existante est aussi le bienvenu. Voici quelques exemples d'améliorations qui pourraient être effectuées :

— Ajout d'un CI sur GitLab pour vérifier les tests et compiler le JAR.
— Ajout d'un logger pour améliorer la gestion des logs.
— Améliorer l'architecture du projet.
— Améliorer la documentation du code.
— Faire les TODO présents dans le code.

1.4 Structure du rapport

Ce rapport décrit le déroulement de ce projet ainsi que les problèmes, les décisions et les évolution de ce dernier. Les différentes parties de ce rapport sont décrites ci-dessous :

— Introduction : Ce chapitre explique le contexte et définit les objectifs de ce projet.
— COJAC : Ce chapitre décrit les notions nécessaires à la compréhension du fonctionnement de COJAC.
— Intégration des nombres complexes : Ce chapitre explique l'intégration complète des nombres complexes. Il décrit les nombres complexes, les décisions prises, l'implémentation, les tests, etc.
— Intégration des Unums : Ce chapitre explique l'intégration complète des Unums. Il décrit les Unums, les choix effectués pour l'implémenter, des tests, etc.
— Versions : Ce chapitre liste la version des librairies utilisées.
— Conclusion : Ce chapitre résume l'état du projet par rapport à ses objectifs ainsi que ses possibilités d'amélioration.
— Références : Ce chapitre liste les sources sur lesquels se basent ce projet.
— Glossaire : Définit certains termes utilisés dans le rapport.

Chapitre 2

COJAC

COJAC [3] est un programme Java permettant de modifier les capacités arithmétiques d'une application Java cible sans en modifier le code. Elle utilise l'API d'instrumentation et peut transformer les classes et méthodes au runtime pour changer le type de calcul effectué. Ainsi, pour changer le type de calculs effectué (ex : nombre réel → nombre complexe), il faut seulement changer l'argument donné à COJAC lors du démarrage de l'application. Ceci ne demande aucune modification dans l'application cible.

Ce chapitre détaille tout d'abord les agents Java, car ils sont essentiels au fonctionnement de COJAC. Ensuite, le Bytecode Java sera expliqué parce que c'est là-dessus que fonctionne COJAC. Finalement, la dernière section explique comment les fonctionnalités supplémentaires désirées peuvent être ajoutés dans COJAC.

2.1 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. C'est sur ce principe fondamental que repose COJAC. Comme montré dans la Figure 2.1, l'agent Java Java intercepte les fichiers .class qui contiennent du Bytecode et peut y apporter des modifications. Le Bytecode correspond à l'assembleur de Java. Ce concept est détaillé dans la section 2.2 suivante.

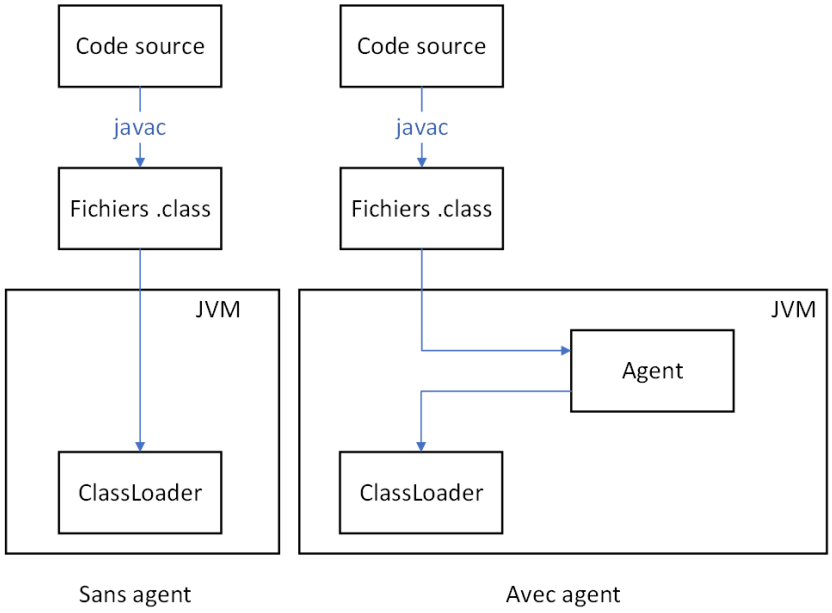

FIGURE 2.1 – Chargement d'une classe

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.

La majorité des informations dans cette section provient d'une conférence de Rafael Winterhalter [20]. La première moitié de la conférence se focalise sur la base des agents Java et des problèmes qui y sont liés. Alors que la seconde moitié se focalise sur Byte Buddy [19], une librairie pour simplifier la transformation de classes.

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

2.1.1 Agent statique

Un agent statique est un agent Java qui est spécifié au démarrage du programme comme c'est le cas avec COJAC. Ainsi, lors du démarrage de l'application, l'agent et la JVM interagissent ensemble conformément à la Figure 2.2. Cette interaction se déroule en plusieurs étapes :

1. La JVM appelle l'agent Java 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 d'octets 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.

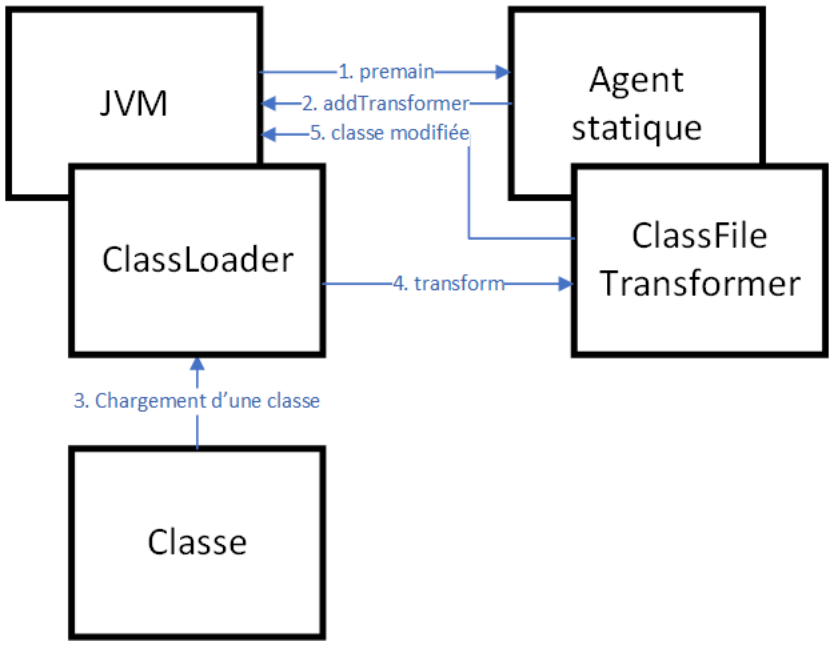

FIGURE 2.2 – Fonctionnement d'un agent statique

2.1.2 Agent dynamique

Un agent dynamique est un agent Java 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 2.3 :

1. Une application pour ajouter l'agent doit être créée. Cette application doit s'attacher à la JVM du processus dans lequel l'agent doit être inséré. Cette opération ne 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 Java 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 qui sont listées dans la documentation officielle de cette méthode [11]. 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.

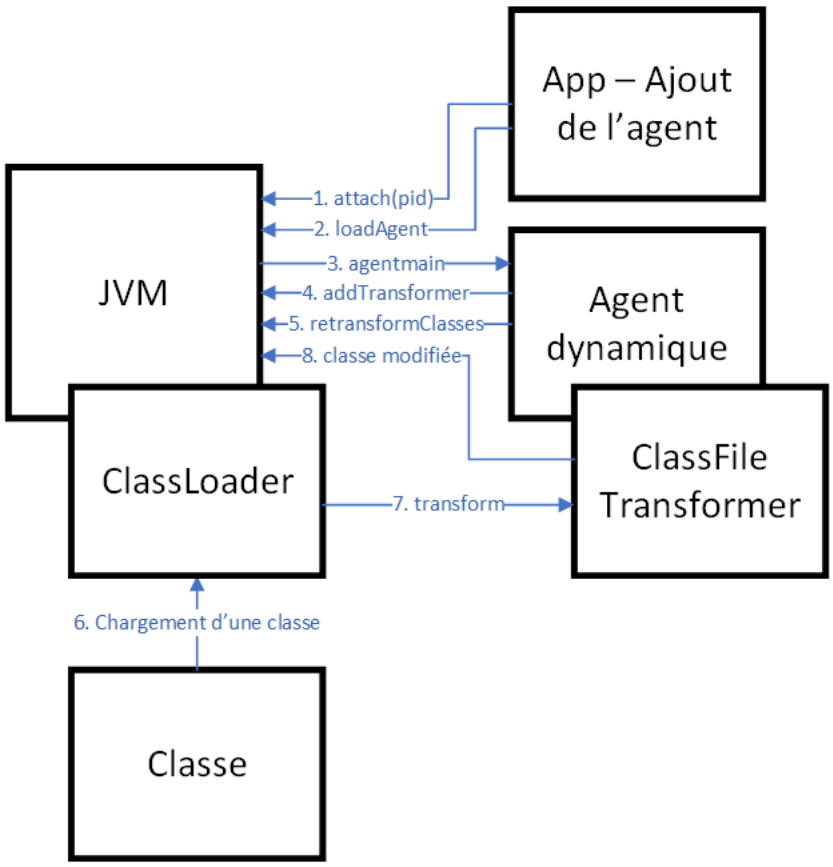

FIGURE 2.3 – Fonctionnement d'un agent dynamique

2.1.3 Synthèse

Un agent Java 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 limitations. Celles-ci sont définies dans la documentation officielle de la méthode correspondante [11].
— 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.

2.2 Bytecode Java

Tel que brièvement mentionné dans la section 2.1 précédente sur les agents 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 ajoutent 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 2.4. 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.

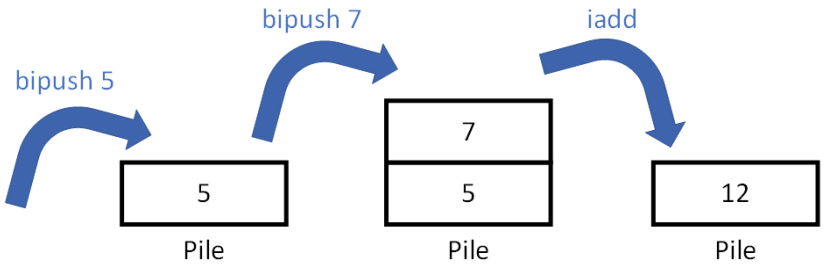

FIGURE 2.4 – Addition de deux nombres en Bytecode Java

Des informations plus complètes sont disponibles dans une conférence intitulée "Java Bytecode Crash Course" de David Buck, un ingénieur chez Oracle [1].

Les spécifications de chaque instruction sont aussi définies dans la documentation officielle d'Oracle [2].

2.3 Intégration

COJAC 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.

2.3.1 Ajout d'un wrapper

Pour ajouter un nouveau Wrapper dans COJAC, il faut créer une nouvelle classe dans le package *com.github.cojac.models.wrappers*. Cette nouvelle classe doit hériter de la classe abstraite *ACojacWrapper*. Il faut ensuite implémenter toutes les méthodes.

On peut ensuite tester le nouveau Wrapper en suivant ces deux étapes :

1. Créer le JAR de COJAC
2. Démarrer l'application cible en donnant le JAR créé précédemment comme agent Java et lui donner le Wrapper à utiliser avec l'option *-W*. Pour démarrer une application avec un Wrapper nommé *WrapperComplexNumber*, la commande suivante sera utilisée :

```
java -javaagent:cojac.jar="-W cojac.WrapperComplexNumber"
↪  demo.HelloComplexNumber
```

2.3.2 Limitations

La documentation de COJAC [4] prévient que les Wrappers de COJAC sont encore expérimentaux et qu'ils possèdent les limitations suivantes :

— Il y a des problèmes dans les transitions entre le code utilisateur et la librairie standard Java. Par exemple, lors de l'utilisation de tableaux de nombres.
— Les "callbacks" de la bibliothèque Java vers le code utilisateur lorsque des nombres à virgule flottante sont transmis ne sont pas supportés.
— L'instruction Bytecode *invokedynamic* devrait fonctionner pour Java 8, mais cela n'est pas garanti. De plus, les nouvelles utilisations de cette instruction tel que mentionné dans la section 6.2 ne sont pas supportés.
— L'utilisation de la *Java reflection* n'est pas supportée.
— La conversion des types primitifs (float/double) et de leur Wrapper original (Float/ Double) provoquent certains problèmes tels que des conflits de signatures de méthodes ou des erreurs de comparaison.
— L'implémentation des modèles n'est pas optimale. Par exemple, les appels à la librairie standard *Math.** ne calculent pas les valeurs aussi précisément que demandées.
— Il y a également un ralentissement important.

2.3.3 Méthodes magiques

COJAC offre aussi un mécanisme de méthodes magiques. Ces méthodes permettent à l'application cible d'utiliser du code de COJAC. Le principe qui se cache derrière se mécanisme est bien documenté dans la section 5.4 d'un rapport précédent sur les nombres enrichis dans COJAC [12].

2.4 Maven

COJAC utilise Maven [14] pour créer le JAR. Maven 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*.

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.

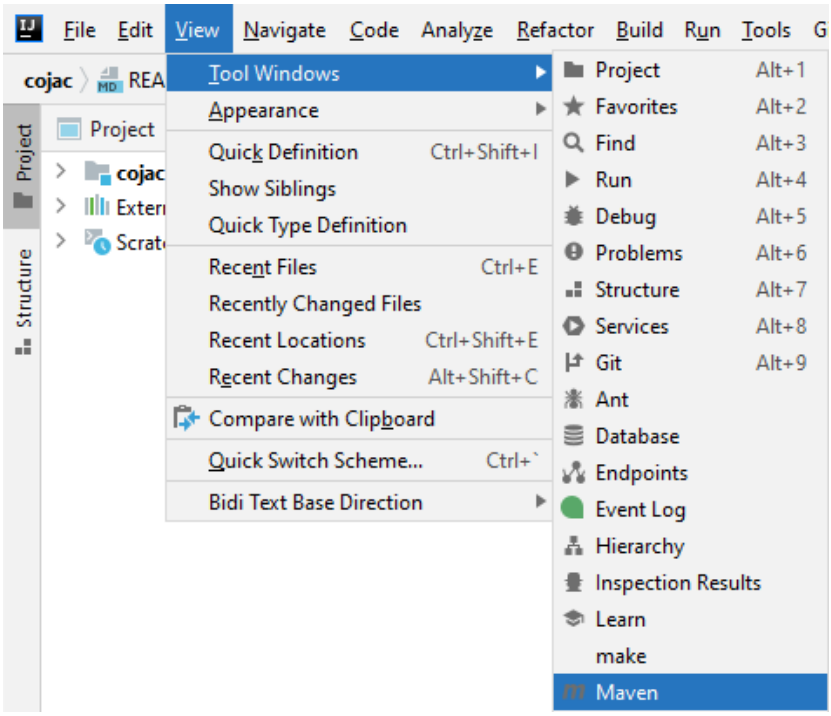

FIGURE 2.5 – Bouton d'ouverture de la fenêtre Maven

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.

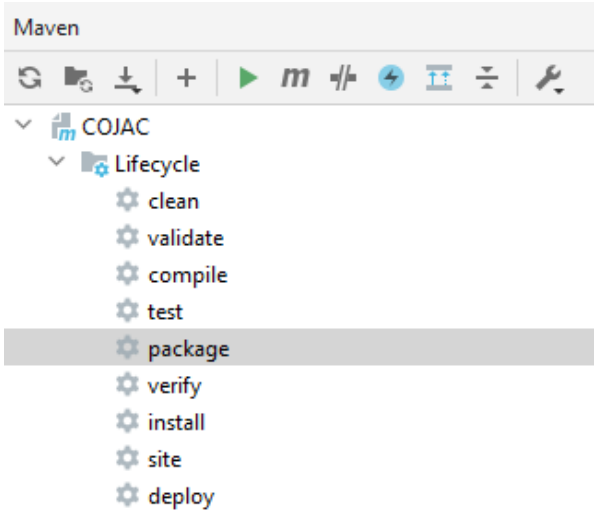

FIGURE 2.6 – Fenêtre Maven

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. Cette option est visible sur la Figure 2.7. L'option en ligne de commande équivalente se nomme *--debug*.

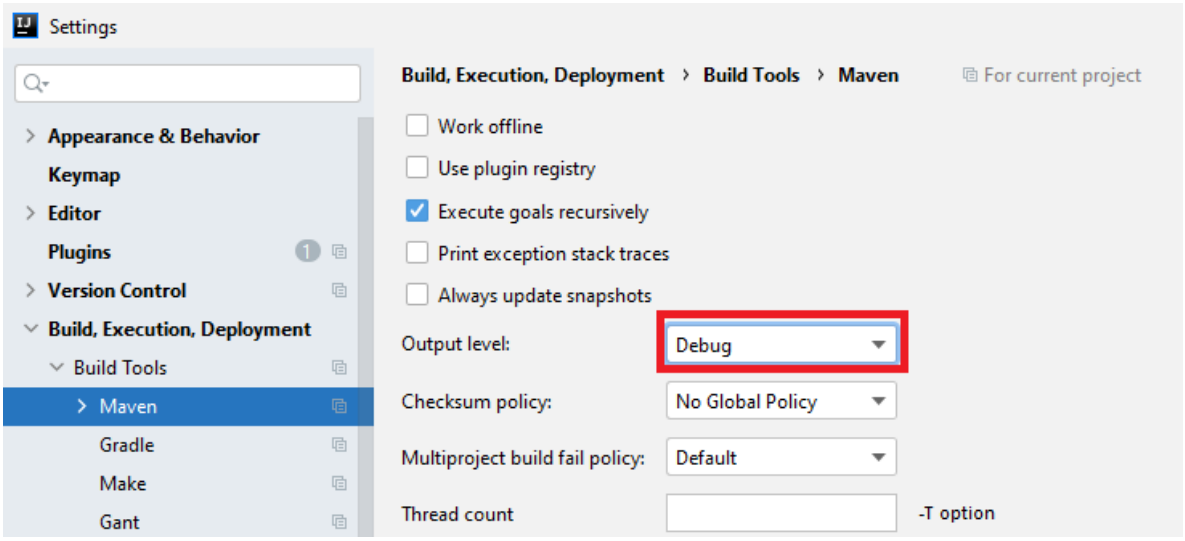

FIGURE 2.7 – Configuration d'affichage des messages de debug

Chapitre 3

Intégration des nombres complexes

3.1 Nombres complexes

Lorsque l'humanité à créer les nombres, ils 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 suffisants pour décrire une grande majorité des lois et phénomènes physiques.

Une raison majeure de la nécessité d'intégrer dans l'arsenal mathématique le corps des nombres complexes tient au fait qu'il 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 [6].

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 : $3 - 2i$
- Trigonométrique : $r(cos(\theta) + i \cdot sin(\theta))$
- Exponentielle : $re^{i\theta}$
- Polaire : (r, θ)

Voici l'exemple du nombre $3 - 2i$ visible sur la Figure 3.1.

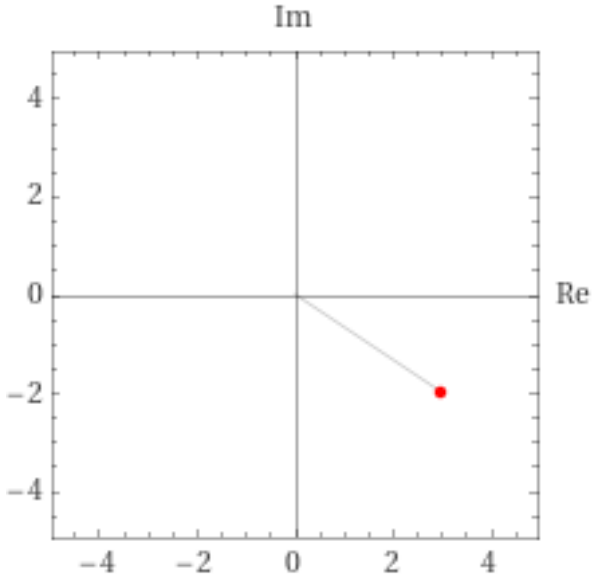

FIGURE 3.1 – Le nombre $3 - 2i$ dans l'espace complexe

Voici les différentes notations correspondant toutes au nombre $3 - 2i$:

- Algébrique : $3 - 2i$
- Trigonométrique : $\sqrt{13} \cdot (cos(-tan^{-1}(\frac{2}{3})) + i \cdot sin(-tan^{-1}(\frac{2}{3})))$
- Exponentielle : $\sqrt{13} \cdot e^{-i \cdot tan^{-1}(\frac{2}{3})}$
- Polaire : $(\sqrt{13}, -tan^{-1}(\frac{2}{3}))$

Dans chacune de ces formes, on connaît soit la partie réelle et la partie imaginaire, soit l'angle et le module (ou rayon) du nombre complexe. On peut passer d'une forme à l'autre en utilisant les formules de Pythagore et de trigonométrie.

Les variables suivantes seront utilisées :

— Partie réelle : a
— Partie imaginaire : b
— Module/rayon : r
— Angle : θ

On peut ensuite définir les conversions entre la forme cartésienne et polaire de la manière suivante :

— $a = rcos(\theta)$
— $b = rsin(\theta)$
— $r = \sqrt{a^2 + b^2}$
— $\theta = tan^{-1}(y/x)$

3.2 Spécifications

L'intégration des nombres complexes doit permettre de pouvoir réaliser toutes les opérations arithmétiques standards comme l'addition ou la multiplication. Les comparaisons doivent pouvoir fonctionner avec les nombres réels. 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 vérifier que l'implémentation des nombres complexes est correcte. 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");
}
```

3.4 Conception

Ce chapitre contient la conception de l'intégration des nombres complexes dans COJAC.

3.4.1 Approche

Comme expliqué dans la section 2.3, l'intégration des nombres complexes peut se faire à l'aide de deux mécanismes différents :

— Behaviour : réinterprétation des *floats* et des *doubles*.
— Wrapper : remplacement des *floats* et des *doubles* par un Wrapper.

L'intégration des nombres complexes se fera en utilisant un Wrapper afin de pouvoir garder la *double precision*.

3.4.2 Comparaison

La comparaison entre les nombres complexes est un problème important. Toutes les comparaisons entre les nombres sont faites à l'aide d'une même opération et il faut aussi que les comparaisons entre des nombres purement réels soient respectés. Ainsi si on compare des nombres nombres complexes (avec une partie imaginaire), il n'y a pas forcément de valeurs de retour adaptées car il n'y a pas d'ordre total avec les nombres complexes. Cette situation peut être montrée avec l'exemple de code suivant :

```
double a = ...;
if (a == 0) {
    System.out.println("a = 0");
} else {
    System.out.println("a != 0");
}
```

Contrairement à ce qu'on peut penser, la vérification de l'égalité provoque, en interne, une comparaison où on ne doit pas seulement dire si c'est égal ou pas, mais également quel nombre est plus petit que l'autre.

Si la variable a est purement réelle, ce code doit fonctionner. Cependant il y a un problème si la variable a est complexe. Pour cet exemple, la valeur de a sera $2i$. Lorsque le remplacement de cette opération est faite, la méthode doit comparer ces deux nombres ($2i$ et 0) et retourner une valeur parmi les trois suivantes :

— $2i < 0$
— $2i = 0$
— $2i > 0$

Cependant, aucune de ces réponses n'est vraie. Il est possible de créer une méthode magique pour gérer les égalités, mais ceci nécessitera que l'application cible soit modifiée pour réaliser la comparaison.

Comme aucun des deux choix n'est complètement satisfaisant, une option supplémentaire pour COJAC sera disponible pour choisir le comportement des comparaisons. Ces deux modes différents sont détaillés dans la section 3.4.4 suivante.

3.4.3 ToString and fromString

Il a été décidé que le *toString* écrirait le nombre sous sa forme complexe. Ceci peut provoquer une erreur si l'application cible utilise un format strict. Il est surtout important que le *toString* du Wrapper puisse être converti en Wrapper à nouveau. Ainsi les méthodes *Float.parseFloat* et *Double.parseDouble* doivent aussi fonctionner avec le résultat du *toString* du Wrapper.

3.4.4 Modes

Il n'y a pas de comportement idéal pour implémenter la comparaison (cf section 3.4.2) et le cast en double. Ainsi, les deux modes différents sont détaillés dans cette section. Une synthèse et des exemples comparent les résultats de ces différentes approches.

Mode normal

Dans le mode normal, la comparaison se fera d'abord avec la partie réelle, puis avec la partie complexe. Ainsi *2 - i* sera considéré comme plus petit que *2*. Le cast d'un nombre complexe en double provoquera la perte de la partie imaginaire.

Ce mode a l'avantage suivant :

— L'application cible n'a pas ou peu besoin de modifications pour fonctionner. Ceci respecte au maximum l'ordre établi entre les nombres réels.
— Les méthodes magiques créées pour le mode strict sont aussi disponibles en mode normal. Ces méthodes sont expliquées dans la section 3.4.4.

Il possède également les inconvénients suivants :

— Un ordre total est défini alors qu'il n'existe pas mathématiquement. Ceci peut conduire à des comparaisons donnant un résultat mathématiquement faux et produire des incohérences.
— La partie imaginaire peut être perdue sans le remarquer car cela ne provoquera aucune erreur.

Mode strict

Quant à lui, le mode strict offrira une comparaison mathématiquement correcte. La comparaison entre deux nombres purement réels donnera toujours un résultat. La comparaison entre deux nombres complexes égaux donnera également un résultat correct. Dans les autres cas, une exception sera générée.

Lors du cast d'un nombre complexe en nombre réel, la présence d'une partie imaginaire générera une erreur. S'il n'y a qu'une partie réelle, le cast fonctionnera.

De plus, une méthode magique permettra de vérifier l'égalité de deux nombres complexes. Elle retournera un booléen même en cas d'inégalité. Deux méthodes magiques permettant de garder seulement la partie réelle ou imaginaire d'un nombre complexe sera aussi disponible et permettra à l'utilisateur d'implémenter lui-même la comparaison entre les nombres complexes.

Ce mode offre les avantages suivants :

— La comparaison est mathématiquement correcte. Ainsi, l'utilisateur est conscient des problèmes qui peuvent se produire lorsqu'il travaille avec les nombres complexes.
— L'usage des méthodes magiques permet une plus grande flexibilité avec les nombres magiques.

Ce mode a aussi l'inconvénient suivant :

— L'application cible doit être écrite en tenant compte des fonctions magiques de COJAC.

Exemples

Voici plusieurs exemples montrant la différence entre le mode normal et le mode strict.

Le premier exemple trie un tableau de nombres complexes, puis calcule leur valeur absolue. Le résultat obtenu est :

— Mode normal : le code s'exécute et le tableau final n'est plus trié.
— Mode strict : une erreur est générée lors de l'appel à la méthode *Arrays.sort*, parce qu'une comparaison est effectuée avec des nombres complexes contenant une partie imaginaire.

```
double[] sorted_positives = ... // ex: new double[]{2 + 3i, 3, 1};
Arrays.sort(sorted_positives); // sorted_positives = [1, 2 + 3i, 3]
double[] sorted_abs = new double[sorted_positives.length];
for (int i = 0; i < sorted_positives.length; i++){
    sorted_abs[i] = Math.abs(sorted_positives[i]);
}
// sorted_abs = [1, 3.6, 3]
```

Le second exemple cast un double en int. Le résultat est le suivant :

— Mode normal : a est égal à la partie imaginaire du nombre (a = 2) ;
— Mode strict : une exception est générée à cause de la perte de la partie imaginaire

```
double d = Math.sqrt(-1) + 2; // 2 + i
int a = (int) d;
```

Avec une méthode magique, le même résultat peut aussi être obtenu avec le mode strict :

— Mode normal ou strict : a est égal à la partie imaginaire du nombre (a = 2) ;

```java
double d = Math.sqrt(-1) + 2; // 2 + i
int a = (int) COJAC_MAGIC_getReal(d);
```

Synthèse

Le mode normal nécessite aucune modification pour démarrer le programme. Cependant, il peut y avoir des erreurs de logique. Le mode strict garanti un comportement mathématiquement correct et limite le nombre d'erreurs. Cependant, il est nécessaire d'utiliser des méthodes magiques et, par conséquent, de modifier le code de l'application cible. Les méthodes magiques sont aussi disponibles en mode normal et il est ainsi possible de faire la transition du mode normal au mode strict par étapes. Ces méthodes magiques peuvent aussi être utilisés pour avoir un comportement plus sûr à un endroit spécifique du code.

3.4.5 Librairie disponible

La librairie *commons-math3* est déjà disponible en Java pour gérer les nombres nombres complexes [5]. Elle possède beaucoup de fonctionnalités et implémente déjà la grande majorité des opérations mathématiques nécessaires. De plus, cette librairie fait déjà partie des dépendances de ce projet. Elle sera ainsi utilisée pour implémenter les nombres nombres complexes.

3.4.6 Diagramme de classes

Toutes les méthodes abstraites du *ACojacWrapper* seront implémentées en plus de plusieurs autres méthodes. De plus, une nouvelle méthode sera ajoutée au *ACojacWrapper* afin de pouvoir supporter la méthode *Math.cbrt*. Les méthodes du Wrapper sont surchargées pour retourner un *WrapperComplexNumber* à chaque fois pour simplifier les tests. Toutes les méthodes de la classe *WrapperComplexNumber* qui sont visibles sur le diagramme de classes ont été implémentées.

FIGURE 3.2 – Diagramme de classes de l'intégration des nombres complexes

3.5 Implémentation

Cette section décrit les aspects principaux de l'implémentation des nombres complexes dans COJAC.

3.5.1 Wrapper

La création d'un nouveau Wrapper se fait en créant une nouvelle classe héritant de *ACojacWrapper*. Comme *ACojacWrapper* est une classe abstraite, le Wrapper ne peut pas étendre d'autres classes.

Il faut également implémenter un constructeur prenant un *ACojacWrapper* en paramètre. Il faut également gérer le cas où ce paramètre est null. Voici le constructeur implémenté pour ce Wrapper :

```java
public WrapperComplexNumber(ACojacWrapper w) {
    // CommonDouble can call this constructor with a null wrapper
    if (w == null) {
        this.complex = new Complex(0, 0);
    } else {
        Complex value = ((WrapperComplexNumber) w).complex;
        this.complex = new Complex(value.getReal(), value.getImaginary());
    }
}
```

3.5.2 Remplacement de méthodes

Seulement une sélection de méthodes de la librairie standard sont remplacées. Deux méthodes supplémentaires sont également remplacées car elles sont utilisées dans la démonstration. Il s'agit de :

- *Math.cbrt(double)*
- *Double.isNaN(double)*

Le remplacement des méthodes de la librairie standard pour les Wrappers se fait dans la classe *com.github.cojac.instrumenters.ReplaceFloatsMethods* dans la méthode *fillMethods*. Les lignes suivantes ont été ajoutées pour remplacer ces deux méthodes supplémentaires :

```java
// String CDW; // signature of wrapper
// String CDW_N; // name of wrapper
invocations.put(new MethodSignature(DL_NAME, "isNaN", "(D)Z"),
new InvokableMethod(CDW_N, "double_isNaN", "(" + CDW + ")Z",
↪  INVOKESTATIC));

invocations.put(new MethodSignature(MATH_NAME, "cbrt", "(D)D"),
new InvokableMethod(CDW_N, "math_cbrt", "(" + CDW + ")" + CDW,
↪  INVOKESTATIC));
```

Il faut également ajouter les deux méthodes déclarées ici : *double_isNaN* and *math_cbrt* dans la classe *com.github.cojac.models.wrappers.CommonDouble* et les faire appeler la méthode correspondante du Wrapper :

```java
public static boolean double_isNaN(CommonDouble a){
    return a.val.isNaN();
}

public static CommonDouble math_cbrt(CommonDouble a){
    return new CommonDouble(a.val.math_cbrt());
}
```

3.5.3 Ajout de l'option

Une nouvelle option doit aussi être ajoutée à COJAC pour que le Wrapper puisse être utilisé et configuré. Tout d'abord, il faut ajouter une valeur dans l'enum *com.github.cojac.Arg*.

```
COMPLEX_NUMBER ("Rc"),
```

Cette enum a aussi une méthode *createOptions*. Il faut aussi y ajouter une option. Ceci permettra de rendre l'option publique et utilisable comme argument et permettra aussi d'afficher l'aide à propos de cette option. Voici le code pour ajouter une option avec un argument facultatif :

```
options.addOption(OptionBuilder
    .withArgName("strict")
    .hasOptionalArg()
    .withDescription("Use complex number wrapping. Strict mode
    ↪  generates an exception when the imaginary " +
            "part is lost or when a comparison between two different
            ↪  complex numbers is made.")
    .create(COMPLEX_NUMBER.shortOpt()));
```

Il faut aussi détecter quand l'option est utilisée et la traiter en conséquence. Ainsi, il faut ajouter une condition dans la classe *com.github.cojac.CojacReferences.CojacReferencesBuilder* à l'intérieur de la méthode *build*. Ce code doit être ajouté suffisamment haut dans la méthode car elle définit l'option *Arg.NG_WRAPPER* qui est aussi traitée dans la même méthode. Voici l'extrait de code qui permet de traiter cette option :

```
if (args.isSpecified(Arg.COMPLEX_NUMBER)) {
    args.setValue(Arg.NG_WRAPPER,
    ↪  "com.github.cojac.models.wrappers.WrapperComplexNumber");
    WrapperComplexNumber.setStrictMode(
    ↪  args.getValue(Arg.COMPLEX_NUMBER) != null);
}
```

3.5.4 *toString* et *fromString*

La méthode *toString* a été implémentée pour avoir trois formats possibles d'après le type de nombres complexes.

— Nombre réel : uniquement la partie réelle (ex : 2.25).
— Nombre imaginaire : uniquement la partie réelle avec un i à la fin (ex : -4.2i).
— Nombre complexe : les deux parties sont affichées (ex : -4.25 + 4.0i ou 1.25 - 1.5i).

Ces deux méthodes se basent sur l'affichage du double de Java. Ainsi, *1.401298464324817E-45 - 1.0i* est une représentation possible d'un nombre complexe et il est aussi possible de convertir cette chaîne de caractères en un nombre complexe.

La méthode *fromString* est capable de lire et de créer un nombre complexe à partir de tous les exemples de String valides donnés précédemment.

Cependant, des modifications ont dû être effectuées parce que les méthodes *Float.parseFloat* et *Double.parseDouble* appellent les méthodes des classes *CommonFloat* et *CommonDouble*. Cependant, ces classes font ensuite elles-même un appel aux vraies méthodes *Float.parseFloat* et *Double.parseDouble*.

Voici l'ancien code du *CommonDouble* :

```
public CommonDouble(String v) {
    this(Double.valueOf(v));
}

public static CommonDouble fromString(String a){
    return fromDouble(Double.valueOf(a));
}
```

Et voici le nouveau code qui fait l'appel à une nouvelle méthode du *ACojacWrapper* :

```
public CommonDouble(String v) {
    val = newInstance(null).fromString(v, false);
}

public static CommonDouble fromString(String a){
    return new CommonDouble(a);
}
```

3.6 Tests

Cette section décrit l'ensemble des tests unitaires et d'intégration réalisés pour vérifier le fonctionnement du Wrapper. Tous les tests écrits se déroulent avec succès.

3.6.1 Tests unitaires

Toutes les méthodes ont été testées avec au moins un cas général. Les cas particuliers n'ont pas été testés. Ceci inclut les grands nombres, les nombres tout petits et autres valeurs spéciales (NaN, null, etc.). Quelques tests plus précis ont été ajoutés pour éviter que certains bugs, qui ont été corrigés depuis, se reproduisent.

55 tests unitaires ont été réalisés. De plus, ces tests couvrent 100% de la classe *WrapperComplexNumber*. Les détails sont visibles dans le tableau suivant :

Élément	**Classe**	**Méthodes**	**Lignes**
WrapperComplexNumber	100% (1/1)	100% (47/47)	100% (118/118)

3.6.2 Tests d'intégration

Des tests d'intégrations ont aussi été réalisés. Une classe de test est instrumentée et le retour de ces méthodes sont vérifiées. 13 tests ont été réalisés en mode normal et 14 tests en mode strict.

3.7 Documentation

Quelques commentaires ont été ajoutés dans le code à quelques endroits spécifiques pour expliquer les choix réalisés. Comme les tests d'intégration sont séparés en deux classes, le retour attendu a aussi été documenté car la vérification des valeurs se fait uniquement dans

la deuxième classe. Aucun autre commentaire n'a été ajouté parce qu'ils n'apportaient que peu de valeur parce que les méthodes sont généralement courtes et explicites.

Une nouvelle section dans le wiki décrit désormais aussi le Wrapper des nombres complexes. Cette section détaille également les deux modes disponibles et montre également un code de démonstration.

3.8 Résultats

L'implémentation des nombres complexes fonctionnent correctement. Des tests unitaires et d'intégration assurent également la fiabilité de cette implémentation. La démonstration donne également le résultat attendu et montre l'intérêt de cette fonctionnalité. Cette fonctionnalité est aussi documentée dans le wiki.

Chapitre 4

Intégration des Unums

4.1 Format de stockage des nombres réels

Il y a deux types principaux de nombres à utiliser : les nombres entiers et les nombres réels. Les nombres entiers sont simplement stockés en base binaire. Quant-à-eux, les nombres réels peuvent être stockés sous plusieurs formes.

Un format peut être évalué selon plusieurs critères :

- Nombre de bits utilisés
- Précision de la représentation d'un nombre
- Gamme dynamique (rapport entre les valeurs maximales et minimales qui peuvent être représentées avec ce format)
- Précision des calculs
- Complexité de l'implémentation hardware

Quelques formats sont expliqués dans cette section, cependant d'autres formats existent également.

4.1.1 Virgule fixe

Le principe des nombres à virgule fixe est de stocker les nombres en tant qu'entiers et d'y appliquer un facteur d'ajustement constant. Ainsi, on peut décider que le nombre représenté est égal à l'entier divisé par 10'000. Lorsqu'on veut avoir plus de chiffres après la virgule, on réduit également la valeur maximale que peut stocker ce type. Ce format a ainsi des limites importantes. La virgule fixe est utilisée lorsque le processeur ne possède pas de d'unité de calcul pour la virgule flottante.

4.1.2 Virgule flottante

Le principal problème de la virgule fixe est qu'il est impossible de stocker à la fois un nombre très grand et un nombre très petit dans le même format. Il est cependant possible d'utiliser les bits à disposition de façon plus intelligente. Les nombres réels peuvent aussi être stockées sous la forme d'une notation scientifique tel que : $-2.568 * 10^{-6}$. Ainsi, les nombres peuvent être représentés sous la forme suivante : $signe * mantisse * base^{exposant}$. Parmi ces quatre éléments, la base n'est pas stockée. La base est souvent égale à 2.

Les ordinateurs actuels utilisent tous les nombres à virgule flottante et la spécification a été révisée en 2019 [10].

4.1.3 Unums

Les **universal numbers** (**Unums**) sont un nouveau format de stockage dont la première version a été publié en 2015 par M. John L. Gustafson. Durant les années suivantes, les Unums ont été modifiés a deux reprises. La liste des versions des Unums sont visibles dans l'introduction de l'arithmétique des posits (Unum III) [7].

Le créateur des Unums comparent les différents versions des Unums avec la virgule flottante et affirme qu'ils sont meilleurs dans tous les cas dans sa présentation lors d'un séminaire à Stanford [8]. D'après lui, les Unums offrent, entre autres, :

- Une meilleure précision
- Une gamme dynamique plus large
- Meilleures réponses avec le même nombre de bits
- Réduction de la consommation d'énergie et de la latence

Beaucoup plus d'informations sur les Unums sont également disponibles sur le site `posithub.org` [18].

4.2 Spécifications

Les calculs seront réalisé avec les Unums autant que possible. Les *floats* et les *doubles* auront le même comportement pour les autres méthodes (*toString*, *parseDouble*, etc.). Les Unums doivent fonctionner au moins sur une plateforme. Si seulement quelques plateformes sont supportés, il est nécessaire de décrire la procédure pour porter cette fonctionnalité sur une nouvelle plateforme.

4.3 Démonstration

La démonstration se base sur deux exemples donnés par M. John L. Gustafson.

En utilisant les formules suivantes : $E(0) = 1, E(z) = \frac{e^z - 1}{z}, Q(x) = |x - \sqrt{x^2 + 1}| - \frac{1}{x + \sqrt{x^2 + 1}}$, la fonction suivante $H(x) = E(Q(x)^2)$ évaluée avec les valeurs 15, 16, 17 et 9999 doit valoir 1.

Lorsque ce calcul est réalisé en *double precision*, le résultat est toujours 0. Le code utilisé est le suivant :

```java
private static double func1E(double a) {
    if (a == 0) return 1;
    return (Math.exp(a) - 1) / a;
}

private static double func1Q(double a) {
    double sqrt = Math.sqrt(a * a + 1);
    return Math.abs(a - sqrt) - 1 / (a + sqrt);
}

public static double func1H(double a) {
    double q = func1Q(a);
    return func1E(q * q);
}

public static void main(String[] args) {
    // example from https://youtu.be/jN9L7TpMxeA?t=1994
    double[] inputs = new double[]{15, 16, 17, 9999};
    for (double input : inputs) {
        double result = func1H(input);
        System.out.println(result + " should be 1.0");
    }
}
```

Le deuxième exemple est un produit scalaire de 2 vecteurs particuliers. Lors des tests, le *double precision* est suffisant pour obtenir le bon résultat contrairement aux résultats obtenus par M. John L. Gustafson. Ceci peut s'expliquer par le fait que les calculs ne sont pas forcément identiques entre des machines différentes et que la norme IEEE-754 [10] a de nombreuses options facultatives.

Le produit scalaire est implémenté de la manière suivante :

```java
public static float scalarProduct(float[] a, float[] b) {
    assert (a.length == b.length);
    float result = 0f;
    for (int i = 0; i < a.length; i++) {
        result = Math.fma(a[i], b[i], result);
    }
    return result;
}

public static void main(String[] args) {
    // example from https://youtu.be/aP0Y1uAA-2Y?t=104
    float[] a = new float[]{3.2e7f, 1, -1, 8.0e7f};
    float[] b = new float[]{4.0e7f, 1, -1, -1.6e7f};
    float result = scalarProduct(a, b);
    System.out.println(result + " should be 2.0");
}
```

La longueur minimale pour que ces exemples fonctionnent n'est pas connue.

4.4 Conception

Ce chapitre contient la conception de l'intégration des Unums dans COJAC.

4.4.1 Librairie

Plusieurs librairies existantes permettent de gérer des Unums. Cependant, aucune librairie n'a été trouvée pour Java.

Deux librairies avancées différentes ont été trouvées :
- ***Universal*** [17], une librairie C++
- ***SoftPosit*** [15], une librairie C

4.4.2 Approche

Deux approches principales sont disponibles :
- Créer une passerelle vers une librairie native.
- Implémenter les Unums en Java.

Ces approches sont très différentes et possèdent chacune des avantages et inconvénients différents.

Passerelle vers la librairie native *universal*

Avantages :
- La librairie est complète et testée.

Inconvénients :
- Cette option ne fonctionnera que sur certaines plateformes.
- Il faut également écrire du code natif pour faire la passerelle.
- Il est aussi nécessaire de libérer la mémoire lorsqu'il n'y a plus besoin de la classe.
- Des problèmes peuvent survenir en cas de concurrence.

Passerelle vers la librairie native *SoftPosit*

Avantages :
- Les Posits 8 et 16 bits ont beaucoup de tests. Le Posit 32 bits est aussi testé, mais il peut encore y avoir des problèmes.
- Pas besoin d'alouer et libérer de la mémoire

Inconvénients :
- Cette option ne fonctionnera que sur certaines plateformes.
- Il faut également écrire du code natif pour faire la passerelle.
- La librairie ne contient pas de Posit 64 bits.

Implémenter les Unums en Java

Avantages :
- Le support de toutes les plateformes est gardée.

Inconvénients :
- La librairie doit être entièrement implémentée, mais la librairie native peut servir de modèle.
- La librairie doit être maintenue. Il est possible de créer un dépôt public séparé afin de laisser d'autres personnes y contribuer.

Choix

La passerelle vers la librairie native a deux défauts importants :
- Il n'est pas possible de supporter toutes les plateformes. Cependant, le plugin NAR [16] peut aider à supporter plus de plateformes.
- La librairie *universal* fonctionne avec une classe C++. Ainsi, il est nécessaire de créer un objet et de garder cette instance. Cependant, la libération de la mémoire est un problème parce que la méthode *finalize* qui pourrait être utilisée est dépréciée depuis Java 9 [13]. Il est aussi possible de recréer une classe à chaque fois, mais cela réduira largement les performances.
- La librairie *SoftPosit*

Bien que la librairie *SoftPosit* ne contienne pas de Posit 64 bits, elle sera tout de même utilisé pour plusieurs raisons :

- L'implémentation a été testée et les bugs devraient être rares.
- Une meilleure implémentation avec ses tests nécessiterait plus de temps qu'il reste d'ici la fin du projet.
- Si une implémentation 64 bits est ajoutée dans la librairie, elle pourra facilement être intégrée à COJAC.
- En cas de bugs ou de modifications de la spécification, il est assez probable que d'autres personnes mettent à jour la librairie
- La précision des Posits 32 bits pourront tout de même être comparés avec les *floats*.

4.4.3 Passerelle vers le code natif

Plusieurs alternatives existent pour faire la passerelle depuis le code Java vers le code natif en C :

- JNI
- JNA
- SWIG

JNI sera utilisé pour effectuer la passerelle vers le code natif pour plusieurs raisons :

- L'invocation des méthodes est plus lente avec JNA. Vu le nombre extrêmement important d'appels devant être effectués, les performances de l'invocation sont critiques.
- SWIG possède peu d'analyse, de documentation et d'exemples pour Java. Il est ainsi difficile d'évaluer l'efficacité de cette alternative.

4.5 Implémentation

Cette section détaille les points importants nécessaires à l'implémentation des Posits dans COJAC.

4.5.1 Passerelle JNI

Tout d'abord, la classe *Posit32Utils* a été écrite avec tous les appels de méthodes natifs.

```
/* ====================
 * Arithmetic methods
 * ====================
 *
 * All the methods below take one or multiple posit32 as inputs.
 *
 * Use #toPosit to convert a float in a posit.
 */
public native static float add(float a, float b);

public native static float substract(float a, float b);
```

Ensuite, il faut générer le header correspondant avec la commande suivante :

```
src\main\java> javac -h
↪  ..\resources\native-libraries\posits\src\include
↪  com\github\cojac\utils\Posit32Utils.java
```

Il faut garder le fichier généré tel quel. Le nom des méthodes est important pour pouvoir réaliser la passerelle entre Java et le code natif.

Dans le fichier source qui contient l'implémentation des méthodes déclarées dans le fichier header généré précédemment. Une ligne a été ajoutée pour redéfinir un type qui n'est pas connu de tous les compilateurs :

```
#define __int64 long long

#include "com_github_cojac_utils_Posit32Utils.h"
```

Deux macros et une union ont également été écrits afin de pouvoir changer le type des données :

```
#define F2P(VALUE) (((float_posit32_t) {.jfloat = (VALUE)}).posit)
#define P2F(VALUE) (((float_posit32_t) {.posit = (VALUE)}).jfloat)

typedef union float_posit32 {
    jfloat jfloat;
    posit32_t posit;
} float_posit32_t;
```

Voici un exemple qui montre l'implémentation d'une méthode et son utilisation de ces macros.

```
JNIEXPORT jfloat JNICALL Java_com_github_cojac_utils_Posit32Utils_add
        (JNIEnv *env, jclass positClass, jfloat a, jfloat b) {
    return P2F(p32_add(F2P(a), F2P(b)));
}
```

4.5.2 Build de la librairie

Des fichiers de configurations sont disponibles pour pouvoir créer la librairie. La librairie *SoftPosit* possède déjà un *Makefile* même si des modifications ont dû être effectuées comme mentionné dans les sections 6.4 et 6.5. Un CMakeLists.txt a été créé pour la compilation de la librairie qui fait la liaison avec JNI.

```
1  cmake_minimum_required(VERSION 3.12)
2
3  # We are only interested in finding jni.h: we do not care about
   ↪  extended JVM
4  # functionality or the AWT library.
5  set(JAVA_AWT_LIBRARY NotNeeded)
6  set(JAVA_JVM_LIBRARY NotNeeded)
7  set(JAVA_AWT_INCLUDE_PATH NotNeeded)
8
9  find_package(Java REQUIRED)
10 find_package(JNI REQUIRED)
11 include(UseJava)
12
13 include_directories(${JNI_INCLUDE_DIRS} ${JAVA_INCLUDE_PATH2})
14
15 project(native)
16
17 file(GLOB SOFT_POSIT libraries/SoftPosit/*.c)
18
19 add_library(posits_jni SHARED
20         src/com_github_cojac_utils_Posit32Utils.c
21         ${SOFT_POSIT}
22         )
23
24 target_include_directories(posits_jni PRIVATE include
   ↪  ${JNI_INCLUDE_DIRS} ${JAVA_INCLUDE_PATH2}
   ↪  libraries/SoftPosit/source/include)
25
26 TARGET_LINK_LIBRARIES (posits_jni ${CMAKE_SOURCE_DIR}/libraries/Sof
   ↪  tPosit/build/Linux-x86_64-GCC/softposit.a)
```

Les lignes importantes sont décrites ci-dessous :

— Lignes 5-7 : Cmake peut ne pas réussir à trouver certains éléments de Java et JNI, qui ne sont pas nécessaires pour ce projet. Ainsi, ces lignes indiquent qu'il n'a pas besoin de les chercher.
— Lignes 9-10 : Il faut trouver Java, puis JNI afin de pouvoir inclure les headers nécessaires pour la compilation de la librairie.
— Lignes 13 et 24 : Ces lignes indiquent les dossiers qui contiennent les headers nécessaire à JNI.
— Ligne 19 : Cette ligne spécifie qu'une librairie doit être créée.
— Ligne 26 : Cette ligne permet d'ajouter la librairie *SoftPosit* à l'intérieur de la librairie générée.

Tout d'abord, il faut compiler la librairie *SoftPosit* avec les commandes suivantes :

```
cd libraries/SoftPosit/build/Linux-x86_64-GCC
make
cd ../../../..
```

Ensuite, les commande suivantes permettent de compiler la librairie qui sert de passerelle avec JNI :

```
cd ..
cmake src
make
```

Windows 10 avec mingw64 a été utilisé pour compiler la librairie pour Windows 64 bits (.dll). L'application Ubuntu 20.04 sur Windows 10 a été utilisée avec le compilateur gcc pour compiler pour Linux 64 bits (.so).

4.5.3 Chargement de la librairie

Le chargement de la librairie est effectuée en se basant sur la classe *ConversionBehaviour* qui doit déjà charger une librairie native. Ainsi, les librairies natives sont compilées et ajoutées dans le JAR. Ensuite, une méthode est créée pour charger la librairie correspondant à la plateforme sur laquelle le JAR est exécuté :

```java
public static void loadLibrary() {
    String libRoot = "/native-libraries/posits/";
    String winLib64 = libRoot + "posits_jni.dll";
    String linLib64 = libRoot + "libposits_jni.so";
    String OSName = System.getProperty("os.name");
    int arch =
    ↪  Integer.parseInt(System.getProperty("sun.arch.data.model"));
    try {
        if (OSName.startsWith("Windows")) {
            if (arch == 64) {
                NativeUtils.loadLibraryFromJar(winLib64);
            }
        } else if (OSName.startsWith("Linux")) {
            if (arch == 64) {
                NativeUtils.loadLibraryFromJar(linLib64);
            }
        } else {
            throw new UnsupportedOperationException("This operation is
            ↪  not supported on this platform");
        }
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

Cette méthode est chargée lorsque l'option indiquant l'utilisation des Posits est traitée afin de charger la librairie uniquement si elle est nécessaire.

4.5.4 Autres

Un nouveau Wrapper a été implémenté pour les Posits 32 de manière similaire aux nombres complexes. Cette étape est décrite dans la section 3.5.1.

Une nouvelle méthode la librairie standard, *Math.fma()*, est désormais aussi remplacée par une méthode du Wrapper. Le processus utilisé est identique à celui détaillé dans la section 3.5.2 sur les nombres complexes.

4.6 Tests

Cette section décrit l'ensemble des tests unitaires et d'intégration réalisés pour vérifier le fonctionnement du Wrapper et de la passerelle JNI. Tous les tests écrits se terminent avec succès.

4.6.1 Tests unitaires

Toutes les méthodes ont été testées avec au moins un cas général. Les cas particuliers n'ont pas été testés. Ceci inclut les grands nombres, les nombres tout petits et autres valeurs spéciales (NaN, null, etc.).

49 tests unitaires ont été réalisés. De plus, ces tests couvrent 100% de la classe *WrapperPosit32*. Les détails sont visibles dans le tableau suivant :

Élément	**Classe**	**Méthodes**	**Lignes**
WrapperPosit32	100% (1/1)	100% (40/40)	100% (55/55)
Posit32Utils	100% (1/1)	100% (1/1)	62% (10/16)

Les 11 méthodes natives de la librairie, appelées par Posit32Utils ne s'affichent pas dans la couverture. Les lignes de la classe *Posit32Utils* qui n'ont pas été couvertes par les tests sont des lignes qui ne s'exécutent que sur d'autres plateformes. Cependant, les tests fonctionnent en local (Windows) et sur le GitLab CI (Linux) qui correspondent à l'ensemble des plateformes supportées actuellement.

4.6.2 Tests d'intégration

Des tests d'intégrations ont aussi été réalisés. Une classe de test est instrumentée et le retour de ces méthodes sont vérifiées. 9 tests vérifient que l'instrumentation du code donne le résultat exact donné par la librairie.

4.7 Résultats

L'implémentation des Unums fonctionnent correctement, mais de nombreuses méthodes sont encore implémentées avec les virgules flottantes. Des tests unitaires et d'intégration assurent également la fiabilité de cette implémentation. Cependant, la démonstration donne des résultats mitigés. Une démonstration montre des résultats identiques alors que l'autre montre que les Posits sont moins précis que les *floats*. Une nouvelle démonstration devant fonctionner avec les Posits 32 a été ajoutée, et cette démonstration montre les Posits peuvent être avantageux. Enfin, cette fonctionnalité n'est pas encore documentée dans le wiki.

Chapitre 5

GitLab CI

Un fichier *.gitlab-ci.yml* a été ajouté pour configurer le CI sur GitLab. Ce fichier permet d'exécuter les tests et de compiler le JAR. Il est affiché ci-dessous :

```yaml
image: maven:3-openjdk-11

variables:
  MAVEN_OPTS: "-Dhttps.protocols=TLSv1.2
  ↪ -Dmaven.repo.local=$CI_PROJECT_DIR/.m2/repository
  ↪ -Dorg.slf4j.simpleLogger.log.org.apache.maven.cli.transfer.Sl
  ↪ f4jMavenTransferListener=WARN
  ↪ -Dorg.slf4j.simpleLogger.showDateTime=true
  ↪ -Djava.awt.headless=true"

  MAVEN_CLI_OPTS: "--batch-mode --errors --fail-at-end
  ↪ --show-version -DinstallAtEnd=true -DdeployAtEnd=true"

cache:
  paths:
    - .m2/repository/

stages:
  - test

```

```
15 maven:verify:
16   stage: test
17   script:
18     - 'mvn $MAVEN_CLI_OPTS verify'
19   artifacts:
20     when: always
21     paths:
22       - target/*.jar
23     reports:
24       junit:
25         - target/surefire-reports/TEST-*.xml
26         - target/failsafe-reports/TEST-*.xml
```

Les lignes importantes sont expliquées ci-dessous :

— Ligne 1 : Openjdk 11 et Maven sont déjà installés dans cette image.
— Lignes 3 à 6 : Ces variables sont écrites par défaut lors de la création de ce fichier via l'interface de GitLab pour un projet Maven.
— Lignes 8 à 10 : Ces lignes permettent de garder les plugins en cache et d'éviter de tous les télécharger à chaque fois.
— Ligne 12 : Il n'y a qu'un seul stage exécuté actuellement.
— Lignes 17 à 18 : Cette commande va exécuter toutes les étapes du cycle de vie de Maven jusqu'au *verify*. Ceci inclus, entre autres, l'exécution des tests, la compilation du JAR et la vérification de celui-ci.
— Lignes 21 à 22 : Les JAR générés sont gardés et peuvent être accédés depuis GitLab.
— Lignes 23 à 36 : Les rapports de Surefire sont également gardés. Ils peuvent être utilisés pour comprendre les erreurs lorsque l'exécution des tests échouent.

Chapitre 6

Problèmes rencontrés

6.1 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.

6.1.1 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.
```

6.1.2 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. Pour ce faire, il faut cliquer sur le bouton dédié dans la fenêtre Maven qui est détaillée dans la section 2.4. Ce bouton est visible sur la Figure 6.1.

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.

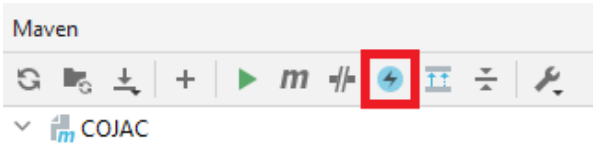

FIGURE 6.1 – Bouton pour ignorer les tests

6.1.3 Cause

Dans un des fichiers de log produits, qui sont mentionnées dans la section 6.1.1, 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 6.1.1. 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.

6.1.4 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.

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

Ce changement corrige le problème.

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

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

6.2.1 Problème

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

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

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

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

6.2.2 Solution temporaire : compiler avec Java 8

COJAC 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
```

6.2.3 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 à l'heure actuelle.

6.3 Erreur des tests unitaires

Lors de la création du JAR, un test peut parfois échouer.

6.3.1 Problème

Un test déjà présent dans le projet et qui agit sur une autre partie du projet peut parfois créer une erreur. Lors de la création du JAR, l'erreur suivante peut se produire :

```
[ERROR] Failures:
[ERROR]   Double2FloatTest.testDouble2FloatConversion:79 On
↪  "testNextUp", Got: 3.0000000000000004, Expected: 3.000000238418579
```

Ce test agit sur les **Behaviours** alors que ce projet se focalise sur les **Wrappers**. Ces deux concepts distincts sont expliqués dans la section 2.3. Cette erreur est devenue un problème particulièrement important lorsqu'elle est devenue récurrente sur le GitLab CI alors qu'elle est invisible sur la machine de développement. Cette erreur complique la détection des autres erreurs sur le GitLab CI.

6.3.2 Cause

Lorsque l'instrumentation est effectuée, elle doit aussi remplacer certaines méthodes des librairies standards telles que *Math.nextUp*. Cependant, il y a une condition qui peut éviter les méthodes d'être instrumentées qui peut être vue dans l'extrait de code suivant :

```
for(Method m:behaviorClass(i).getMethods()){
    if (m.isAnnotationPresent(UtilityMethod.class)){
        break;
    }
```

Lorsqu'il y a une méthode qui est annotée, cette méthode et toutes les méthodes suivantes ne sont pas instrumentées.

6.3.3 Solution

Il semble plus logique de ne pas instrumenter la méthode possédant l'annotation uniquement. Pour ce faire, le code a été changé pour correspondre à l'extrait de code suivant :

```
for(Method m:behaviorClass(i).getMethods()){
    if (m.isAnnotationPresent(UtilityMethod.class)){
        continue;
    }
```

Face à la difficulté de comprendre ce code, il est difficile de savoir si c'est vraiment la bonne correction. Cependant, l'erreur n'est plus réapparue depuis.

6.3.4 Remarques

Cette erreur était très difficile à trouver car elle ne se produisait pas sur la machine locale. Il a fallu ajouter des logs pour détecter la différence de comportement entre les deux environnements. De plus, le code est long et peu explicite. Il utilise également certaines mauvaises pratiques. Ainsi, cette erreur a permis de mettre en avant certains problèmes :

— La classe *com.github.cojac.instrumenters.BehaviourInstrumenter* peut et devrait être améliorée. Il y a beaucoup de code commenté, certaines méthodes sont très longues, il y a jusqu'à 6 niveaux d'indentations et l'apparition d'exceptions est considérée comme un cas habituel et n'est pas signalé.
— La classe *com.github.cojac.instrumenters.BehaviourInstrumenter* contient d'autres *break* suspicieux.
— Les tests du profiler produisent une grande quantité de texte sur la sortie standard. Ce texte n'a pourtant que peu d'intérêt.

6.4 Erreur de compilation de *SoftPosit*

Après le téléchargement du dépôt de *SoftPosit*, la compilation de la librairie provoquait une erreur.

6.4.1 Problème

La compilation de l'erreur provoque de nombreuses erreurs dans le fichier *softposit.h* dont les premières sont disponibles ci-dessous.

```
In file included from ../../source/include/internals.h:50,
                 from ../../source/s_addMagsP8.c:38:
../../source/include/softposit.h:176:5: error: expected '=', ',', ';',
↪  'asm' or '__attribute__' before '.' token
   uA.ui = ((uA.ui + mask) ^ mask)&0xFF;
     ^
../../source/include/softposit.h:177:5: error: expected '=', ',', ';',
↪  'asm' or '__attribute__' before '.' token
   uA.p; \
     ^
../../source/include/softposit.h:178:1: error: expected identifier or
↪  '(' before '}' token
 })
```

6.4.2 Cause

Ces erreurs se produisent à cause de certaines macros qui contiennent des lignes qui ne se terminent pas par le caractère \. Ce caractère est indispensable car il permet de dire que le compilateur doit ignorer le retour à la ligne. Ceci permet ainsi d'indiquer que ces nouvelles lignes font partie de la macro.

```
#define absP8(a)({\
                union ui8_p8 uA;\
                uA.p = (a);\
                int const mask = uA.ui >> 7;
                uA.ui = ((uA.ui + mask) ^ mask)&0xFF;
                uA.p; \
})
```

6.4.3 Solution

Il suffit d'ajouter des \ à la fin de ces lignes pour corriger le problème.

```
#define absP8(a)({\
                union ui8_p8 uA;\
                uA.p = (a);\
                int const mask = uA.ui >> 7; \
                uA.ui = ((uA.ui + mask) ^ mask)&0xFF; \
                uA.p; \
})
```

Un fork du projet a été créé pour corriger ce problème. Une merge request (`https://gitlab.com/cerlane/SoftPosit/-/merge_requests/7`) est actuellement en attente pour corriger ce problème sur le dépôt principal.

6.5 Inclure *SoftPosit* lors de la compilation de la librairie native

Lors de la compilation de la librairie avec Ubuntu, une erreur était générée lors de l'ajout de la librairie *SoftPosit*.

6.5.1 Problème

Lors de la compilation de la librairie qui fait la passerelle entre JNI et *SoftPosit*, l'erreur suivante était produite :

```
/usr/bin/ld: src/libraries/SoftPosit/build/Linux-x86_64-GCC/softposit.
↪  a(p32_sqrt.o): relocation R_X86_64_PC32 against symbol
↪  `softposit_approxRecipSqrt1' can not be used when making a shared
↪  object; recompile with -fPIC
/usr/bin/ld: final link failed: bad value
```

6.5.2 Cause

Cette erreur signifie que *SoftPosit* a été compilée pour être à un endroit précis. Cependant, lorsque *SoftPosit* est ajoutée dans la librairie, le compilateur a besoin de changer les adresses.

6.5.3 Solution

Il faut modifier le *Makefile* de *SoftPosit*. Dans l'ancien *Makefile*, il y avait cette ligne :

```
COMPILE_C = \
  $(COMPILER) -c -Werror-implicit-function-declaration
  ↪  -DSOFTPOSIT_FAST_INT64 \
    $(SOFTPOSIT_OPTS) $(C_INCLUDES) $(OPTIMISATION) \
    -o $@
```

Il faut ajouter l'option *-fPIC* afin d'obtenir la ligne suivante :

```
COMPILE_C = \
  $(COMPILER) -fPIC -c -Werror-implicit-function-declaration
  ↪  -DSOFTPOSIT_FAST_INT64 \
    $(SOFTPOSIT_OPTS) $(C_INCLUDES) $(OPTIMISATION) \
    -o $@
```

Chapitre 7

Versions

Les librairies suivantes, déjà utilisées par COJAC précédemment, ont été mises à jour :

Librairie	**Version précédente**	**Version actuelle**
org.apache.maven.plugins :maven-surefire-plugin	2.12.4	3.0.0-M1
org.apache.maven.plugins :maven-pmd-plugin	2.5	3.14.0
org.codehaus.mojo :findbugs-maven-plugin	2.0.1	3.0.5
org.apache.maven.plugins :maven-checkstyle-plugin	2.5	3.1.2
org.ow2.asm :asm	7.2	9.1
org.ow2.asm :asm-commons	7.2	9.1
org.ow2.asm :asm-tree	7.2	9.1
org.ow2.asm :analysis	7.2	9.1
org.ow2.asm :asm-util	7.2	9.1

La librairie *SoftPosit* [15] est nouvellement utilisée par COJAC. Sa licence permet d'utiliser et de distribuer le code tant que la licence est gardée.

7.1 Conclusion

Ce chapitre résume l'état du projet et son déroulement. Il explique si les objectifs ont été atteints, les améliorations possibles et suggère des propositions pour améliorer ce projet.

7.1.1 Atteinte des objectifs

Tous les objectifs primaires ont été atteints :

— Intégration des nombres complexes : Les nombres complexes ont été intégrés, testés, documentés et une démonstration montre également l'intérêt de cette fonctionnalité.

— Intégration des Unums : Les Unums ont aussi été intégrés et testés. Cependant, plus d'opérations pourraient être calculés en utilisant directement les Unums.
— Démonstrations : Des démonstrations montrent l'utilité que ces deux fonctionnalités peuvent offrir.

Certains objectifs secondaires ont aussi été partiellement atteints :

— Mise à jour des librairies : Plusieurs librairies ont été mises à jour, mais ce n'est pas le cas de tous.
— Documentation : Une section a été ajoutée dans le wiki pour décrire les nombres complexes.
— GitLab CI : Un GitLab CI a été ajouté pour vérifier le fonctionnement des tests et compiler le projet.
— Les Wrappers et Behaviours ont été comparés, mais l'analyse peut être approfondies.

7.1.2 Perspectives d'amélioration

Il y a beaucoup de possibilité d'améliorations pour ce projet. Voici la plus importante :

— Vérifier que la licence de la librairie *SoftPosit* est reportée correctement et que ceci ne cause pas de conflit avec la licence de COJAC.

D'autres améliorations seraient également les bienvenues :

— Corriger la méthode *Agent.transform* qui devrait retourner *null* lorsqu'il n'y a pas de changement.
— Mettre à jour les librairies utilisées.
— Implémenter les Posits en Java directement ou ajouter des fonctionnalités dans la librairie *SoftPosit*.
— Le GitLab CI pourrait également tester le fonctionnement sous Windows et tester la compilation de la librairie native.

D'autres points d'amélioration existent également, mais ont moins d'intérêts que ceux énoncés précédemment :

— La classe *com.github.cojac.instrumenters.BehaviourInstrumenter* peut et devrait être améliorée. Il y a beaucoup de code commenté, certaines méthodes sont très longues, il y a jusqu'à 6 niveaux d'indentations et l'apparition d'exceptions est considérée comme un cas habituel et n'est pas signalé.
— Les tests du profiler produisent une grande quantité de texte sur la sortie standard. Ce texte n'a pourtant que peu d'intérêt.
— Ajouter des tests de performance pour les nombres complexes et les Unums.
— Créer une documentation qui détaille le fonctionnement de COJAC pour permettre aux futurs développeurs de prendre le projet en main plus facilement.
— Nettoyer le code : supprimer le code commenté, formater le code, améliorer le code, etc.

— Utiliser un *logger* pour tout le projet.
— Améliorer l'architecture du projet.
— Effectuer les *TODO* présents dans le code.
— Amélioration le nom des *packages* et l'emplacement des tests.
— Créer un dossier pour la librairie *NativeRoundingMode* qui est déjà présente dans le projet.
— Byte Buddy [19] peut potentiellement simplifier l'écriture du code actuel. Il reste possible d'utiliser la librairie actuelle en parallèle de Byte Buddy pendant la transition.

7.1.3 Conclusion personnelle

Ce projet s'est globalement bien déroulé. J'ai planifié et pris un peu trop de temps sur les nombres complexes, même si le retard par rapport à la planification est dû à des problèmes sur le projet. Le temps était ensuite un peu trop limité pour parfaire l'intégration des Unums. Malgré les problèmes qui se sont principalement produits dans la deuxième moitié du projet, j'ai pu réalisés les objectifs primaires et testés également l'implémentation de ceux-ci afin d'assurer une certaine qualité dans le travail effectué. J'aurais voulu faire quelques tâches supplémentaires, même si j'ai dû les abandonner en faveur d'autres tâches plus importantes.

En résumé, même si j'aurais voulu réaliser quelques tâches supplémentaires, j'ai apprécié travailler sur ce projet et je suis satisfait du travail effectué ainsi que des résultats obtenus. Ce projet démontre également une nette amélioration par rapport au projet de semestre précédent.

7.1.4 Déclaration d'honneur

Je, soussigné, Cédric Tâche, déclare sur l'honneur que le travail rendu est le fruit d'un travail personnel. Je certifie ne pas avoir eu recours au plagiat ou à toute autre forme de fraude. Toutes les sources d'information utilisées et les citations d'auteur ont été clairement mentionnées.

Bibliographie

[1] David Buck. *Java Bytecode Crash Course*. Youtube. 2019. URL : https://www.youtube.com/watch?v=e2zmmkc5xI0&t=430s&pp=ugMICgJmchABGAE%5C%3D.

[2] *Chapter 6. The Java Virtual Machine Instruction Set*. Oracle. URL : https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html. (visité le 15.06.2021).

[3] *COJAC - Boosting arithmetic capabilities of Java numbers*. 2019.

[4] *COJAC - Limitations and known issues - Issues with the wrapper*. URL : https://github.com/Cojac/Cojac/wiki#62---issues-with-the-wrapper. (visité le 15.06.2021).

[5] *Complex (Apache Commons Math 3.6 API)*. Apache Commons. URL : https://commons.apache.org/proper/commons-math/javadocs/api-3.6/org/apache/commons/math3/complex/Complex.html. (visité le 20.06.2021).

[6] Frédéric Elie. « Spineurs et algèbre vectorielle en physique quantique ; application à l'équation de Dirac ». In : (oct. 2017).

[7] John L. Gustafson. *Posit Arithmetic*. Juil. 2017. URL : https://posithub.org/docs/Posits4.pdf.

[8] John L. Gustafson. *Stanford Seminar: Beyond Floating Point: Next Generation Computer Arithmetic*. Youtube. 2017. URL : https://www.youtube.com/watch?v=aP0Y1uAA-2Y.

[9] Gustafson et Yonemoto. « Beating Floating Point at Its Own Game: Posit Arithmetic ». In : *Supercomput. Front. Innov.: Int. J.* 4.2 (juin 2017), p. 71-86. ISSN : 2409-6008. DOI : 10.14529/jsfi170206. URL : https://doi.org/10.14529/jsfi170206.

[10] « IEEE Standard for Floating-Point Arithmetic ». In : *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (juil. 2019), p. 1-84. DOI : 10.1109/IEEESTD.2019.8766229.

[11] *Instrumentation (Java Platform SE 8) - retransformClasses*. Oracle. URL : https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/Instrumentation.html#retransformClasses-java.lang.Class...-. (visité le 15.06.2021).

[12] Sylvain Julmy. « COJAC - Nombres enrichis ». 2015.

[13] *Object (Java SE 11 & JDK 11)*. Oracle. URL : https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#finalize(). (visité le 06.07.2021).

[14] Brett Porter, Jason van Zyl et Olivier Lamy. *Welcome to Apache Maven.* URL : `https://maven.apache.org/` (visité le 05/06/2021).

[15] *SoftPosit.* 2021. URL : `https://gitlab.com/cerlane/SoftPosit`.

[16] *The NAR plugin for Maven.* 2020. URL : `https://github.com/maven-nar/nar-maven-plugin`.

[17] *Universal: a header-only C++ template library for universal number arithmetic.* 2019. URL : `https://github.com/stillwater-sc/universal`.

[18] *Unum & posit - next generation arithmetic.* URL : `https://posithub.org/`. (visité le 04.07.2021).

[19] Rafael Winterhalter. *Byte Buddy - runtime code generation for the Java virtual machine.* 2021. URL : `https://bytebuddy.net/`.

[20] Rafael Winterhalter. *The definitive guide to Java agents by Rafael Winterhalter.* Youtube. 2020. URL : `https://www.youtube.com/watch?v=oflzFGONG08`.

Chapitre 8

Glossaire

agent Java Un objet Java qui peut intercepter le chargement d'une classe et modifier son Bytecode. 5–9, 11

Behaviour Un mécanisme de COJAC permettant de modifier les opérations agissant sur les types primitifs. 3, 10, 18, 46, 52

Bytecode Le langage utilisée par la JVM 5, 9–11, 45

COJAC Un outil capable de modifier le comportement d'une application Java à l'aide d'un agent Java. 1–6, 10–12, 18, 19, 21, 23, 25, 32, 34, 45, 52

Java ARchive (JAR) Un fichier regroupant toutes les ressources et toutes les classes utilisées par une application Java. 4, 11–13, 38, 41–46

Java Development Kit (JDK) Un ensemble d'outils utilisés pour développer une application Java. Cela inclus la JVM, le compilateur, etc. 1

Java Native Interface (JNI) JNI permet d'appeler des fonctions natives (généralement écrites en C) depuis un programme Java. 2, 34, 37, 38, 40, 49

Java virtual machine (JVM) Une machine virtuelle qui permet d'exécuter des programmes écrits en Java ainsi qu'avec certains autres langages. Les programmes exécutés doivent préalablement être compilés en Bytecode. 6–9

Makefile Un fichier de configuration composé de règles permettant de compiler un programme. 37, 49

Maven Un outil d'automatisation pour gérer les dépendances et compiler une application. 12–14, 42, 44

nombre complexe Une extension des nombres réels permettant de faire certaines opérations mathématiques impossibles avec les nombres réels. 1–5, 15–23, 26, 28, 39, 51–53

Posit Un nouveau format de stockage spécifié dans les Unums III. 33, 34, 39, 40, 52

SoftPosit Une librairie C permettant de réaliser certains calculs avec des Unums. 32–34, 37, 38, 47, 49, 52

Universal number (Unum) Un nouveau format de stockage pour les nombres réels dont le but est de remplacer les nombres à virgule flottante. 1–4, 30, 32, 33, 40, 52, 53

Wrapper Un mécanisme de COJAC permettant de remplacer les *floats* et *doubles* par un objet. 3, 11, 18, 19, 22–25, 27, 28, 39, 40, 46, 52