Université de Franche-comté

Rapport de projet

Développement de machines virtuelles en méthodes agiles

Léo Leuthardt
Malak Majdoub
Hugo Rougetet
Ali Mohamed Chehem
Vital Focheux
Younes Benhamroura

1er Septembre 2024 — Décembre 2024

Table des matières

Introduction

Durant notre semestre 7 du Master informatique à l'UFR Sciences et Techniques de Besançon, nous avons eu l'opportunité de réaliser un projet de développement en appliquant différentes méthodes agiles de travail. Ce rapport a pour objectif de détailler les techniques de développement mises en œuvre au cours de ces quatre mois, ainsi que les technologies utilisées pour mener à bien ce projet.

Dans un premier temps, nous aborderons le sujet du projet, ses contraintes et les technologies que nous avons choisi d'utiliser.

Par la suite, nous décrirons en détail le travail accompli sur les différentes parties attendues du projet, ainsi que des tests réalisés.

Pour finir, nous présenterons les méthodes de travail utilisées, ce qu'elles nous ont apportés, et la manière dont nous les avons mises en place.

1 Sujet

Le projet de développement agile de machines virtuelles s'inscrit dans l'objectif d'acquérir des compétences à la fois dans le domaine de la compilation et dans les différentes méthodes de travail en collaboration.
En effet, ce projet vise à valider les compétences pratiques en lien avec les connaissances théoriques acquises dans ces deux domaines. Pour cela, nous devions concevoir un compilateur capable de transformer un langage donné en un langage dit de bas niveau, tout en l'intégrant dans un environnement de développement que nous devions également créer nous-mêmes.

1.1 Contraintes

Plusieurs contraintes nous étaient imposées pour la réalisation du projet :

1.1.1 Contraintes de développement

Afin de travailler dans de bonnes conditions, le sujet nous impose certaines façons de travailler ainsi que certains éléments de travail.

- Un langage de haut niveau, le *MiniJaja*, une version simplifiée et modifiée du Java. Le second langage, le*JajaCode*, représente un langage de bas niveau, comparable au Bytecode du Java. Ces deux langages seront les sources principales sur lequel se portera le développement.
- Le développement incrémentale de l'application, on retrouve notamment deux phases : *Release 1* et *Release 2*, mais aussi l'utilisation de *User Stories* pour guider et évaluer l'avancement du projet.
- Utilisation des différents outils mis à disposition par l'université pour implémenter l'intégration et le déploiement continus (CI/CD), ainsi que pour gérer les différents aspects des méthodes agiles tel que les User Stories, tickets, etc.

1.1.2 Contraintes fonctionnelles

Afin de garantir un logiciel de qualité en fin de production, nous avons reçu une liste de contraintes fonctionnelles que doit respecter le produit final.

- Compiler tous les programmes MiniJaja corrects et bien typés, vérification faites selon la grammaire MiniJaja qui nous est donné.
- Comparer les résultats d'interprétation du MiniJaja avec sa contrepartie compilée, le JajaCode.
- Compilation des programmes édités grâce à un éditeur de texte sous différents systèmes d'exploitation.
- Observer le JajaCode issu de la compilation.
- Interprétation pas à pas du MiniJaja et du JajaCode.
- Exécution d'un programme munis de points d'arrêt.

Ces contraintes, représentant une partie conséquente du travail, nous avons dû prendre un temps pour réfléchir aux technologies que nous allions utiliser afin de les satisfaire.

1.2 Technologies utilisées

Pour répondre aux contraintes du projet, nous avons utilisés diverses technologies. Ces outils nous ont permis de structurer efficacement notre travail et de livrer un produit conforme aux attentes. Nous détaillerons ci-dessous les principales technologies utilisées.

1.2.1 Technologies de développement

Tout d'abord, l'intégralité du développement à été réalisée en Java 17, un langage riche de son API et adapté aux exigences du projet, notamment pour l'implémentation du compilateur et des outils associés. Plus précisément, nous avons créé un projet Maven qui permet la création de modules, la gestion des dépendances mais aussi de tester les différents modules.

Pour l'analyse syntaxique, nous avons utilisés JavaCC/JJtree. Ces technologies ont été choisies pour leur rôle de parseur et de générateur d'arbres syntaxiques. Leur facilité d'implémentation, combinée à leur capacité à générer des arbres de syntaxe abstraite (AST), nous a permis de progresser rapidement dans le développement de l'interpréteur et du compilateur.

Afin de garantir une collaboration synchronisée entre tous les membres de l'équipe, nous avons utilisé Gitlab, un gestionnaire de versions, avec une version mise à disposition par l'Université. En complément, nous avons aussi eu l'opportunité de travailler avec Nexus, un outil de dépôt distant, et SonarQube, un logiciel d'analyse de code.
Ces deux outils, également fournis par l'Université, nous ont permis de garantir la qualité du code produit et la qualité du produit final.

1.2.2 Technologies de travail en méthodes agiles

Pour assurer une organisation efficace et une communication fluide au sein du groupe, nous avons utilisé deux outils : Jira et Discord.

Jira a été notre principal outil de gestion de projet. Grâce à lui, nous avons pu suivre l'avancement du travail grâce aux tickets que nous pouvions créer, attribuer et suivre tout au long des sprints. Ces tickets étaient basés sur des User Stories, qui décrivaient les fonctionnalités attendues du point de vue utilisateur.

En parallèle, Discord a été notre principal canal de communication. Cet outil nous a permis d'organiser des réunions ainsi que de discuter rapidement des différents problèmes. La création de canaux dédiés (développement, tests, réunion, etc) a renforcé la collaboration et a permis de structurer efficacement nos échanges.

En combinant ces outils, nous avons réussi à établir un mode de travail collaboratif et agile, essentiel pour mener à bien ce projet.

2 Réalisation des machines virtuelles

Ce projet, axé sur la réalisation de machines virtuelles, se décomposait en plusieurs parties ou modules :

— Analyseur syntaxique : Ce module permet d'analyser les différents langages et de générer leurs arbres de syntaxe abstraite.

— Compilateur : Chargé de transformer le code MiniJaja en JajaCode.

— Interpréteur : Module qui aura pour objectif d'interpréter les différents codes, que ce soit en MiniJaja ou en JajaCode.

— Mémoire : Permet la gestion du tas et de la pile.

— IHM (ou UI en anglais) : Le module de l'interface Homme-Machine permet à l'utilisateur d'utiliser le produit final de façon intuitive. De plus il joue un rôle central, faisant office de point de liaison entre les différents modules.

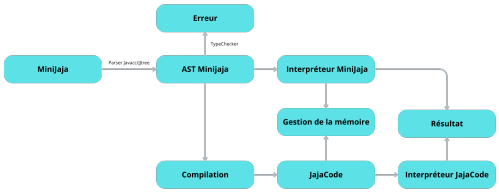


FIGURE 2 – Récapitulatif modules AVM

La figure 2 représente un schéma récapitulant ces modules et leurs connexions. Nous détaillerons ci-dessous les différents modules, les technologies employées pour leur réalisation et leur mode de fonctionnement.

2.1 Analyseur syntaxique

L'analyseur syntaxique comprend 3 parties principales. La création de la liste de lexèmes, le parseur utilisant les lexèmes pour valider les règles de grammaire, et enfin la création de l'arbre de syntaxe abstraite. En utilisant Javacc, la définition des lexèmes et des règles de grammaire est très intuitive. De plus sa doc étant accessible et facilement compréhensible, cette partie est très facilement réalisable. En revanche, avant de commencer à utiliser JavaCC il faut s'assurer que la grammaire qui nous est donné (MiniJaja) est LL(1). Cela veut dire que l'on doit éliminer toutes ambiguïtés et cela comprend notamment l'entièreté des récursions à gauche.
Une fois cette étape complété, nous avons pu passer à la définition des lexèmes comme nous pouvons le voir sur la figure 3.

```
<DEFAULT> TOKEN :
{
    <CLASS: "class">     |
    <MAIN: "main">       |
    <VOID: "void">       |
    <INT: "int">         |
    <BOOLEAN: "boolean"> |
    <FINAL: "final">     |
    <TRUE: "true">       |
    <FALSE: "false">     |
    <LENGTH: "length">   |
    <WHILE: "while">     |
    <RETURN: "return">   |
    <WRITE: "write">     |
    <WRITELN: "writeln"> |
    <IF : "if">          |
    <ELSE : "else">
}
```

FIGURE 3 – Définition des lexèmes

Les lexèmes étant maintenant définis, nous devions implémenter les règles de grammaire en JavaCC. En plus de ces règles de grammaires nous devions aussi marquer celles qui se devaient de produire des noeuds pour l'AST. La figure 4 montre un exemple de ces règles de grammaire.

```
private void decl() : {}
{
    typemeth() ident() decl1()          |
    <FINAL> type() ident() vexp() #Cst(3)
}

private void decl1() : {}
{
    <LPAR> entetes() <RPAR> <LCBRACKET> vars() instrs() <RCBRACKET> #Methode(5) |
    var1()
}
```

FIGURE 4 – Définition des règles de grammaire et des noeuds de l'AST pour cst et méthode

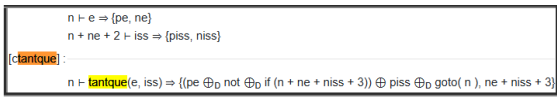
L'ensemble de ces règles et lexèmes combiné à la commande `javacc MiniJaja.jjt` permettront à JavaCC de créer les tokens nécessaires à la construction de l'arbre de syntaxe abstraite. Enfin, c'est au tour de l'outil JJTree de faire son entrée afin de créer cet AST automatiquement en utilisant la commande `jjtree MiniJaja.java`

2.2 Compilateur

Le compilateur représente la partie centrale du produit final. C'est en effet autour de ce module que gravite le projet. Pour sa mise en œuvre, nous avons adopté plusieurs approches.

Tout d'abord, nous avons fait le choix de retourner une liste d'instructions plutôt que de passer par un fichier pour stocker le résultat de la compilation. C'est un choix motivé par considération par les performances et par la facilité d'implémentation. En effet, devoir écrire dans un fichier, lire ce fichier, le parser puis reconstruire son arbre abstrait semblait long et coûteux au moment de l'implémenter.

Passons maintenant à la technique. Le compilateur se construit à partir de l'AST du MiniJaja. En parcourant ce dernier, nous appliquons pour chaque nœud nécessaire les règles de compilation qui nous sont fournis dans le cours de compilation. Ces règles permettent d'assurer la cohérence entre le MiniJaja et la JajaCode. De plus, elles fournissent le nécessaire afin que le JajaCode produit soit fonctionnel. Comme le démontre la figure 5b en comparaison avec la figure 5a nous observons que le code suit précisément la règle de compilation associée. On y retrouve les différentes instructions JajaCode (comme not,if,goto,etc) ainsi que l'instruction de compilation des expressions en entête, suivie de la compilation des instructions du TantQue.
La valeur retournée par chaque fonction du compilateur correspond au nombre de lignes ajoutées dans le code JajaCode, ce qui nous permet d'assurer la bonne organisation du code final.



(a) Règle de compilation du TantQue

```java
@Override
public Object visit(ASTTantQue node, Object data) {
    DataModel dm = (DataModel) data;
    int n = (Integer) dm.data[0];
    Mode m = (Mode) dm.data[1];

    if (m == Mode.DEFAULT) {
        int ne = (int) node.jjtGetChild(0).jjtAccept(this, data);

        fr.m1comp5.jjc.generated.ASTNot nNot = new fr.m1comp5.jjc.generated.ASTNot(JJTNOT);
        instrs.add(nNot);

        ASTIf nIf = new ASTIf(JJTIF);
        instrs.add(nIf);


        int niss = (int) node.jjtGetChild(1).jjtAccept(this, new DataModel(n+ne+2, Mode.DEFAULT));

        ASTJcNbre ifNbre = new ASTJcNbre(JJTJCNBRE);
        ifNbre.jjtSetValue(n+ne+niss+3);
        nIf.jjtAddChild(ifNbre,0);

        ASTGoTo nGoTo = new ASTGoTo(JJTGOTO);
        ASTJcNbre addrGoto = new ASTJcNbre(JJTJCNBRE);
        addrGoto.jjtSetValue(n);
        nGoTo.jjtAddChild(addrGoto,0);

        instrs.add(nGoTo);

        return ne+niss+3;
    }

    return 0;
}
```

(b) Code de compilation du TantQue

Enfin, nous pouvons observer la présence de la classe ***DataModel***. Cette classe représente un tableau de données que l'on peut remplir pour transmettre des informations aux nœuds suivants de l'arbre. Nous l'utilisons notamment pour indiquer aux nœuds si l'exécution se déroule en mode retrait des variables ou en mode d'exécution normal. De plus, ce tableau stocke également le nombre de lignes JajaCode générées jusqu'à présent dans le programme, ce qui permet de garder la cohérence au niveau du JajaCode.

2.3 Interprétation

L'interprétation d'un langage est une étape primordiale pour son exécution. En Java, par exemple, le code est compilé en fichiers *.class*, qui sont ensuite interprétés par la JVM (Java Virtual Machine). Dans notre cas, nous avons développé deux types d'interprétation censés produire les mêmes résultats : l'interprétation MiniJaja et l'interprétation JajaCode. Nous commencerons par détailler l'interprétation MiniJaja avant d'aborder celle de JajaCode. Nous

expliquerons également nos choix de conception pour chaque approche. Enfin, nous conclurons par une synthèse récapitulative, soulignant les fonctionnalités opérationnelles.

2.3.1 Interprétation MiniJaja

Commençons par l'interprétation MiniJaja, qui est la première méthode permettant d'exécuter un programme écrit en MiniJaja. Le processus est le suivant :

- Le *parser* génère un arbre de syntaxe abstraite (AST).
- Les nœuds de cet arbre sont créés automatiquement par le préprocesseur JJTree. Chaque nœud peut être visité, et c'est l'implémentation de la méthode visit associée à ce nœud qui réalise les traitements spécifiques.

L'essentiel du travail consiste donc à implémenter correctement les méthodes des visiteurs. Par exemple, pour la règle d'affectation :

- Si le membre gauche de l'affectation est un indice de tableau, on applique une règle spécifique (*affectationT*).
- Sinon, on applique la règle d'affectation classique.

```
public Object visit(ASTAffectation node, Object data) throws VisitorException {
    checkDebugNode(node);
    try {
        Object val = node.jjtGetChild( i: 1).jjtAccept( visitor: this, data);
        if (node.jjtGetChild( i: 0) instanceof ASTTab tab)
        {
            String id = (String) ((ASTIdent) tab.jjtGetChild( i: 0)).jjtGetValue();
            int idx = (int) tab.jjtGetChild( i: 1).jjtAccept( visitor: this, data);
            if (idx < 0 || idx >= memory.getHeap().getArraySize((int) memory.getSymbolTable().get(id).getValue()))
            {
                throw new ArrayIndexOutOfBoundsException();
            }
            memory.assignValueArray(id, idx, val);
        }
        else
        {
            String id = (String) ((ASTIdent) node.jjtGetChild( i: 0)).jjtGetValue();
            memory.assignValue(id, val);
        }
    } catch (Exception e) {
        throw new InterpreterException(e.getMessage(), node.getLine(), node.getColumn(), callStack);
    }
    return null;
}
```

FIGURE 6 – Exemple de gestion de la règle d'affectation

Cette logique est généralisée à toutes les règles d'interprétation définies dans le langage MiniJaja. Nous avons soigneusement implémenté chaque règle dans les visiteurs appropriés de l'AST, en respectant les spécifications fournies.

Pour la gestion des *scopes*, nous avons adopté le mécanisme suivant :

- Un *scope* global est créé lors de la déclaration de la classe.
- Un *scope* spécifique est défini pour le *main*.
- Chaque fonction appelée possède son propre *scope*, tout en ayant accès aux variables du *scope* global.

Ce mécanisme garantit une organisation claire et cohérente, rappelant le fonctionnement de certains langages comme Java.

Aussi, le traitement des nœuds suit une progression logique :

— La visite commence par le nœud *classe*.
— Ce nœud poursuit en visitant le *main* en deux étapes :
  — Une première visite en mode "défaut", durant laquelle les actions principales sont exécutées (déclaration des variables, appels des fonctions, etc.).
  — Une seconde visite en mode "retrait", qui permet de gérer le nettoyage des déclarations effectuées lors de la première visite.

Après cette présentation du fonctionnement de l'interprétation MiniJaja, passons à l'interprétation JajaCode.

2.3.2 *Interprétation JajaCode*

Cette section aborde l'interprétation JajaCode, qui repose sur un principe similaire à celui de MiniJaja. Comme pour MiniJaja, une implémentation est fournie pour chaque visiteur de l'AST JajaCode. La différence principale réside dans la structure des données traitées. En effet, pour JajaCode :

— Au lieu de traiter un nœud de type *classe*, nous manipulons une liste d'instructions.
— Chaque instruction est associée à une adresse.

Le processus consiste à parcourir cette liste, numérotée de 0 à *adresse - 1*, en exécutant pour chaque élément la règle correspondante à son adresse. Ainsi, à chaque itération, on applique la règle associée au nœud correspondant à l'adresse actuelle dans la liste. Ce parcours se poursuit jusqu'à ce que la fin de la liste soit atteinte.

```
public String interpret() throws VisitorException
{
    System.out.println("Activating debugger...");
    visitorJcc.ActiverDebugger( flag: true);
    int addr = 1;
    while (addr < instrs.size()) {
        instrs.get(addr - 1).jjtAccept(visitorJcc,  data: null);
        addr = visitorJcc.getAddr();
    }
    instrs.get(instrs.size()-1).jjtAccept(visitorJcc,  data: null);

    if (debugger != null) {
        debugger.triggerEventHandler( isFinished: true, root);}

    return visitorJcc.toString();
}
```

FIGURE 7 – Principe interprétation JajaCode

À l'exception de cette organisation, l'interprétation JajaCode reste très proche de celle de MiniJaja. Nous avons suivi de manière rigoureuse les règles d'interprétation JajaCode fournies.

Pour la gestion des *scopes*, nous avons appliqué les mêmes choix que ceux définis pour l'interprétation MiniJaja :

— Un *scope* global est utilisé pour les variables accessibles partout.
— Chaque fonction ou procédure dispose de son propre *scope*, tout en pouvant accéder aux variables du *scope* global si nécessaire.

2.3.3 Synthèse des interprétations MiniJaja et JajaCode

Pour résumer, nos interprétations MiniJaja et JajaCode sont entièrement fonctionnelles. Elles prennent en charge :

- La gestion des tableaux (synonymie, affectation d'indices).
- La gestion des fonctions (appels récursifs, gestion des paramètres).
- L'affectation des variables.
- Les expressions arithmétiques.
- Et d'autres aspects définis par la grammaire MiniJaja.

Nos choix de conception, notamment pour la gestion des *scopes*, visent à garantir une interprétation proche de celle de langages comme Java. En conclusion, les deux interprétations permettent de traiter tous les éléments de la grammaire MiniJaja avec succès.

2.4 TypeChecker

Le contrôle de type constitue une étape essentielle pour garantir le succès de l'interprétation et de la compilation du code source. La classe TypeChecker implémente l'interface MiniJajaVisitor, permettant de parcourir et de valider les nœuds de l'arbre syntaxique abstrait (AST) généré par le parseur.
La classe utilise une table des symboles (SymbolTable) pour gérer les portées et définitions des variables, constantes, tableaux et méthodes. Chaque entrée dans cette table est représentée par un objet mémoire (MemoryObject).

Le contrôleur de type valide les déclarations des variables et des méthodes avant leur utilisation. Il assure que :

- Les variables et méthodes sont préalablement déclarées et atteignable (vérification de la portée).
- Les signatures des méthodes respectent le nombre et le type des paramètres définis.

Il vérifie que les types des variables, constantes, expressions et instructions respectent les règles suivantes :

- Les opérateurs spécifiques (+, -, &&, ||, +=, etc) sont utilisés uniquement sur des types compatibles, tels que les entiers et les booléens.
- Les valeurs des expressions sont correctement évaluées dans les contextes appropriés et retournent le bon typage, notamment pour les instructions *return* et les conditions (if, while).

En outre, le contrôleur de type effectue des vérifications approfondies sur des structures particulières, notamment :

- Les tableaux, pour s'assurer que leurs indices sont bien des entiers.
- Les appels de méthode, en validant l'existence des méthodes invoquées et la correspondance entre arguments et paramètres.

— Les structures conditionnelles et itératives (if, while), en s'assurant que les conditions évaluées sont de type booléen.

Pour finir, il consigne à l'aide du AppLogger, les erreurs et avertissements rencontrés. Cela inclut les incohérences dans les types, les références non déclarées, et les incompatibilités d'opérateurs ou de paramètres.

2.5 Mémoire

La gestion de la mémoire est un aspect essentiel de notre machine virtuelle destinée à exécuter le langage MiniJaja. Elle a pour rôle de stocker diverses données telles que les fonctions et les variables, en vue de leur utilisation ultérieure. Dans MiniJaja, la gestion de la mémoire est implicite, comme en Java avec le garbage collector. Il n'est donc pas nécessaire d'appeler explicitement des fonctions pour allouer dans le tas, contrairement à un langage comme C.

Dans cette section, nous détaillons l'implémentation de la mémoire pour MiniJaja. Nous commencerons par une vue d'ensemble des classes qui composent notre module Memory, en décrivant brièvement leur utilité. Ensuite, nous présenterons nos choix de conception, en mettant l'accent sur la table des symboles et le tas. Enfin, nous explorerons en détail la classe Memory, qui orchestre nos différentes structures de données.

2.5.1 Vue d'ensemble des classes

Comme dit précédemment, le module mémoire comprend plusieurs classes, chacune ayant un rôle précis :

— **ObjectNature** : Une énumération représentant la nature d'un quadruplet (modélisé par la classe MemoryObject). Elle comporte cinq valeurs possibles :
  — *VAR* : Variable
  — *METH* : Méthode
  — *VCST* : Constante non initialisée
  — *CST* : Constante
  — *TAB* : Tableau

— **ObjectType** : Une énumération désignant le type d'un quadruplet, avec quatre valeurs possibles :
  — *INT* : Entier
  — *BOOLEAN* : Booléen
  — *VOID* : Aucun type assigné
  — *OMEGA* : Type indéfini

— **MemoryObject** : Cette classe représente un quadruplet sous la forme $QUAD = ID \times VAL \times OBJ \times SORTE$, où :
  — *ID* est l'identifiant du quadruplet,
  — *VAL* sa valeur,
  — *OBj* sa nature (*ObjectNature*,
  — et *SORTE* son type (*ObjectType*).
  Elle inclut des *getters* et *setters* pour la manipulation de ses données.

- **Stack** : Représente une pile de quadruplets. Elle propose des opérations classiques (telles que *push* et *pop*) et d'autres plus avancées, comme :
  - *getObjectFromTheTop(int n)* : Renvoie le quadruplet situé à la n-ième position en partant du sommet de la pile.
  - *searchVariableFromTop(String id)* : Recherche un quadruplet à partir de son identifiant.
- **StackException** : Exception personnalisée liée à la gestion de la pile.
- **HashTable** : Une table de hachage classique utilisant l'algorithme *FNV-1 Hash*. Elle se re-hache lorsque le facteur de charge (load factor) dépasse 0,5. Cette classe offre des opérations usuelles (*put, get, remove*).
- **SymbolTable** : Implémente la table des symboles, qui associe une table de hachage à chaque nouvelle portée (*scope*). Lorsqu'une portée devient inutile, elle est supprimée, ainsi que sa table associée. Si un symbole n'est pas trouvé dans la portée actuelle, il est recherché dans la portée globale.
- **SymbolTableException** : Exception spécifique à la table des symboles.
- **Heap** : Cette classe représente un tas organisé en blocs de mémoire dont les tailles sont des puissances de 2. Elle implémente toutes les opérations classiques de gestion mémoire, telles que l'allocation, l'accès à un emplacement mémoire via une adresse, ou encore la modification d'une valeur à une adresse donnée. Les stratégies spécifiques utilisées pour l'implémentation de ce tas seront détaillées dans la section suivante.
- **HeapBlock** : Cette classe décrit un bloc individuel dans le tas. Chaque bloc est défini par son adresse de début, sa taille, et son état (utilisé ou libre). À noter qu'un bloc peut être marqué comme libre une fois désalloué.
- **HeapElement** : Cette classe représente un élément alloué dans le tas. Elle contient les informations suivantes : l'adresse de l'élément, la taille réellement allouée (différente de la taille en puissance de 2), le nombre de références pointant vers cet élément, ainsi que le type des valeurs qu'il contient.
- **Memory** : Cette classe joue le rôle de chef d'orchestre dans la gestion de la mémoire lors de l'interprétation. Elle coordonne l'utilisation de la table des symboles, du tas, et de la pile. De plus, elle intègre les axiomes nécessaires à la gestion des allocations mémoire. Un examen approfondi de cette classe sera présenté dans les sections suivantes.

2.5.2 Choix de conceptions

Après avoir présenté les différentes classes composant notre module Memory, nous allons maintenant expliquer certains choix de conception majeurs. Cette section se concentre sur deux aspects essentiels : l'implémentation de la table des symboles et les décisions relatives au fonctionnement du tas.

Table des symboles

Pour optimiser l'accès aux variables, nous avons choisi d'implémenter une table des symboles. Une approche alternative aurait consisté à stocker les variables dans la pile et à les rechercher en parcourant celle-ci. Cependant, cette méthode aurait été inefficace en termes de performances.

Nous avons donc opté pour l'algorithme de hachage performant FNV-1 Hash, reconnu pour sa rapidité et son efficacité. Voici une description de cet algorithme :

```
algorithm fnv-1 is
    hash := FNV_offset_basis

    for each byte_of_data to be hashed do
        hash := hash x FNV_prime
        hash := hash XOR byte_of_data

    return hash
```

Chaque *scope* déclaré se voit attribuer une nouvelle table de hachage pour éviter les conflits. Par exemple, lorsqu'une fonction est appelée, un nouveau *scope* est créé. À la fin de la fonction, ce *scope* ainsi que la table de hachage associée sont supprimés. Si une variable n'est pas trouvée dans le *scope* courant, la recherche est automatiquement effectuée dans le *scope* global, correspondant au premier *scope* déclaré. Ce mécanisme garantit des recherches rapides, ce qui est particulièrement important dans les programmes contenant un grand nombre de variables.

Gestion du tas

En ce qui concerne le tas, nous avons adopté une stratégie de découpage des blocs "au plus juste". Par exemple, si un utilisateur souhaite allouer un tableau de 10 éléments, un bloc de 16 (la plus proche puissance de 2) lui sera attribué. Une fois un bloc libéré, nous avons mis en place une stratégie de reconstruction pour maximiser l'efficacité :

- Les blocs adjacents de même taille sont fusionnés jusqu'à ce qu'il n'existe plus de blocs contigus de tailles identiques disponibles.
- Si l'utilisateur demande une allocation plus grande que le plus gros bloc disponible, nous réassemblons d'abord tous les blocs libres possibles. Si cela reste insuffisant, nous agrandissons dynamiquement la taille du tas, tout en conservant les adresses des blocs déjà alloués.

Voici une illustration de l'agrandissement du tas :

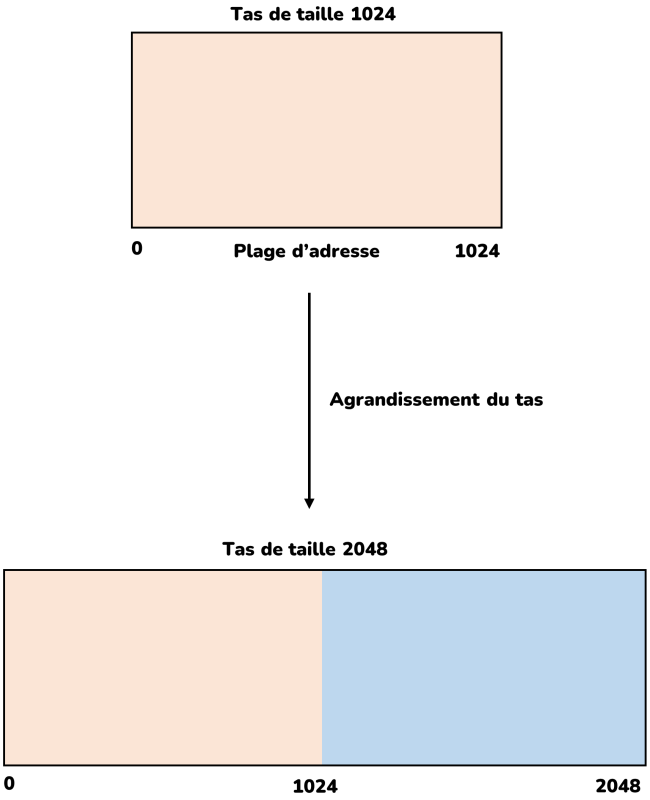


FIGURE 8 – Agrandissement du tas

Pour gérer efficacement les blocs alloués, nous avons adopté une solution interne simple : une liste contenant l'adresse de chaque bloc alloué ainsi que son nombre de références. Ce système permet de libérer un bloc au bon moment tout en facilitant l'accès rapide aux éléments spécifiques du tas.

Ces choix de conception posent ainsi les bases solides pour une gestion efficace de la mémoire. Nous allons à présent nous pencher sur la classe Memory, qui orchestre l'ensemble des interactions entre ces composants et joue un rôle central dans la gestion de la mémoire du module

2.5.3 Focus sur la classe Memory

Après avoir présenté les différentes classes du module Memory et expliqué les choix de conception majeurs, concentrons-nous à présent sur la classe centrale de ce module : la classe Memory. Celle-ci agit comme un véritable chef d'orchestre, coordonnant les interactions entre les différentes structures.

Tout d'abord, il convient de souligner que cette classe implémente les axiomes nécessaires à la gestion de la mémoire pour l'interprétation et la compilation. Voici une partie de ses axiomes :

Axiome de Gestion de la pile :

- $Empiler(q, m) = q.m$
- $Depiler(q.m) = m$
- $Echanger(q_1.q_2.m) = q_2.q_1.m$
- $DeclVar(i, v, t, m) = < i, v, var, t > .m$
- $IdentVal(i, t, [], s) = []$
- $IdentVal(i, t, < i_1, v_1, o_1, t_1 > .m, s) =$ **Si** $s == 0$ **alors** $< i, v_1, var, t > .m$ **sinon** $< i_1, v_1, o_1, t_1 >.IdentVal(i, t, m, s - 1)$
- $DeclCst(i, v, t, m) =$ **Si** $v == w$ **alors** $< i, v, vcst, t > .m$ **sinon** $< i, v, cst, t > .m$
- $DeclTab(i, v, t, m) = < i,$ CréerTas$(v, t, m), tab, t > .m$
- $DeclMeth(i, v, t, m) = < i, v, meth, t > .m$

FIGURE 9 – Axiome mémoire

La classe Memory offre également un accès direct aux instances de la pile, du tas et de la table des symboles, permettant des manipulations manuelles lorsque cela est nécessaire. Cette flexibilité peut s'avérer précieuse dans certains cas spécifiques.

De manière générale, la classe Memory gère automatiquement le tas, tout en intégrant la table des symboles et la pile de manière cohérente. Cependant, certaines fonctionnalités, comme la déclaration automatique d'un nouveau scope pour la table des symboles, ne sont pas encore prises en charge. Cette limitation constitue une piste d'amélioration dans un futur proche.

Également, certaines méthodes, telles que *expParam*, ont été volontairement exclues de cette classe pour éviter des dépendances cycliques avec d'autres modules du projet.

Ainsi, la classe Memory est le cœur du module, regroupant toutes les composantes nécessaires pour garantir l'exécution correcte des programmes dans le langage MiniJaja.

2.6 Interface Homme-Machine

L'interface Homme-Machine permet de faire le lien entre l'utilisateur et l'outil de compilation. Elle doit permettre une expérience plaisante et intuitive qui donne envie à l'utilisateur d'y retourner. C'est avec cette optique que nous avons développé notre interface.
Pour se faire nous avons recouru à plusieurs options.

2.6.1 Choix des technologies

Pour le développement de l'interface, nous avons fait le choix d'utiliser JavaFX comme technologie principale. Ce choix s'appuie sur plusieurs raisons techniques et pratiques

— **JavaFX** : Cette bibliothèque graphique de Java offre des capacités avancées pour créer des interfaces utilisateur riches et réactives. Son architecture Modèle-Vue-Contrôleur (MVC), facilite la maintenance et l'évolution du code. De plus, sa compatibilité native avec Java, le langage principal de notre projet, assure une intégration cohérente avec les autres modules.

— **SceneBuilder** : Cet outil visuel de conception d'interfaces JavaFX nous a permis de concevoir rapidement des layouts complexes et de les modifier facilement. On peut notamment citer la possibilité d'ouvrir ou fermer l'explorateur de fichier dans l'interface (voir figure 21).
— **CSS** : L'utilisation du CSS pour la personnalisation de l'interface permet une grande flexibilité dans la gestion des thèmes et des styles.
— **RichTextFX** : Cette bibliothèque spécialisée pour JavaFX a été choisie pour l'éditeur de code car elle offre des fonctionnalités essentielles comme la coloration syntaxique, la numérotation des lignes et une performance optimale même avec des fichiers plus lourds.

2.6.2 Composants de l'interface

L'architecture de la fenêtre principale s'organise autour de plusieurs panneaux stratégiquement positionnés :

— **Barre de menu et outils** : Positionnée au sommet, elle centralise l'accès aux fonctionnalités essentielles de l'IHM.
— **Explorateur de fichiers** : Situé à gauche, il facilite la navigation dans les projets.
— **Editeur central** : Zone principale intégrant un système d'onglets pour afficher plusieurs fichiers.
— **Console** : ¨Panneau inférieur affichant les résultats et messages.
— **Panneau JajaCode** : Zone optionnelle à droite pour visualiser le code JajaCode généré par la compilation.
— **Panneau visualisation de l'état de la mémoire** : Affichage optionnel de l'état de la mémoire en temps réel.

Les caractéristiques principales de l'éditeur de code sont les suivantes :
— Coloration syntaxique spécifique au MiniJaja
— Numérotation des lignes
— Support multi-fichiers via les onglets
— Retour en arrière et en avant (undo/redo)

Ces fonctionnalités permettent aux développeurs d'avoir une expérience agréable en développant sur du MiniJaja.
De nombreuses autres fonctionnalités sont disponible à travers l'interface. Pour n'en citer que quelques unes on y retrouve :

— Différents boutons d'exécution (compilation, interprétation)
— Raccourcis clavier pour différentes tâches
— Mode sombre/clair
— Exécution pas-à-pas
— . . .

D'un point de vue fonctionnel, l'interface répond aux besoins spécifiques des développeurs MiniJaja. L'ergonomie ayant été prise en compte, nous livrons une interface agréable

et intuitive qui permet aux développeurs de tous niveaux de la prendre en main.

3 Les tests

Dans cette partie, nous allons parler en détail des tests qui ont été réalisés tout au long du développement de ce projet.

3.1 Les tests des différents modules

Le projet a été divisé en plusieurs modules qui sont :
— Analyzer
— Compiler
— Interpreter
— Memory
— UI (for User Interface)

Les cinq premiers modules sont testés via différents types de tests qui sont les tests unitaires et d'acceptations que nous détaillerons par la suite. Le module de l'UI quand à lui n'est pas tester via du code mais avec des tests statiques.

3.1.1 Tests unitaires

Parmi les cinq modules, un seul sera réellement testé via des tests unitaires,le module Memory. De plus, un deuxième sera testé partiellement avec des tests unitaires, le module Interpreter.
Cela s'explique par le fait que les autres modules ne donne pas la possibilité de tester unitairement (ex : UI), tandis que les autres correspondent à des parcours d'arbres (Ex : Compiler) ou bien utilise en majeure parti des fichiers auto-générés (Ex : Analyzer).

Parmi les classes du module Memory certaines seront testées en boite blanche, certaines en boite noire et d'autres en boite grise.
Parmi les tests en boite noire nous pouvons évoquer les classes HashTable et Stack. Ce sont des classes qui effectuent les mêmes opération que celle implanter dans la bibliothèque standard de Java, à la différence qu'elles ont été développées selon les besoin de notre projet. Par conséquent, nous connaissons ce qu'elles doivent donner comme résultat lorsque nous utilisons leurs méthodes.
Tandis que certaines classes comme Memory, SymbolTable et d'autres vont être tester en boite blanche pour essayer de couvrir le plus de lignes de codes, mais aussi essayer de vérifier le bon fonctionnement du code en regardant si les méthodes effectuent bien ce qu'on attends d'elles avec certains paramètres. Par exemple avec le code figure 10, nous savons que cela doit lever une erreur, en revanche nous ne connaissons pas les raisons directement, il faut donc exécuter le test afin de savoir quelle erreur ce code lève.

Enfin, étant donné que pour certaines classes comme Memory, on utilises des attributs qui sont dans le module Memory comme Heap, Stack, HashTable. . .Nous faisons des tests boite grise car nous connaissons ce que produit les classes Heap, Stack et HashTable mais nous ne savons pas forcément avec quelles méthodes elles sont appelées ni à quels moments. La classe

```
Memory mem = new Memory();
mem.getSymbolTable().newScope();
mem.declVar("x", null, ObjectType.OMEGA);
mem.getVal("x");
```

FIGURE 10 – Exemple de code Java qui doit lever une erreur

Interpreter n'est quant à elle pas tester entièrement avec des tests unitaires mais les classes pour le contrôleur de type ainsi que le débugueur sont elles testées de cette manière. Tous les tests ont été réalisés en boite noire car ce qui importe c'est d'être sûr que le test doit lever une exception, que ce soit dû à une mauvaise assignation de type, une erreur syntaxique,. . .

3.1.2 Tests paramétriques

Pour les modules Analyzer, Interpreter et Compiler nous avons effectué des tests d'acceptations grâce aux tests paramétriques. Les tests paramétriques sont des types de tests disponible avec la bibliothèque de JUnit qui permettent de créer un seul test qui prend en source les arguments renvoyés par une méthode, cela permet ainsi d'avoir un seul test avec une méthode qui permet de récupérer toutes les sources plutôt que d'avoir plus d'une centaines de tests écrit, qui se ressemble tous à quelques lignes/variables près.
Nous avons alors écrit un grand nombre de programme MiniJaja (voir annexe test) qui nous permettrons de tester ces trois modules.

Tout d'abord, pour le module Analyzer, nous avons la possibilité de créer l'AST à partir d'un fichier, grâce au programme généré par JavaCC, qui va nous permettre de tester si les programmes MiniJaja lèvent ou non les erreurs que l'on attends de lui pour le parseur. Pour le parser/lexer du JajaCode, nous utilisons le même principe.
En revanche, pour écrire les programmes JajaCode qui ne doivent pas lever d'erreurs nous utiliserons les résultats du compilateur, tandis que pour les programmes qui devant lever des erreurs nous modifierons les programmes générés. Pour le module Interpreter, nous allons utiliser la même démarche que ce soit pour le MiniJaja ou le JajaCode. Nous aurons aussi des programmes que le TypeChecker se doit de vérifier. Pour le module Compiler, nous compilons les programmes MiniJaja et nous utilisons le code JajaCode résultant en tant que code source pour l'interpréteur JajaCode.

3.2 Résultats des tests fournies

Grâce à tous ces tests nous pouvons voir le pourcentage de couverture pour chaque module que nous avons tester :

— Analyzer → 51%
— Compiler → 88%

— Interpreter → 46%
— Memory → > 90%

Au total le projet correspond à 6663 lignes à couvrir et 3703 lignes ont été couvertes grâce aux tests ce qui correspond à environ 54% de couvertures, tout cela pour un total de 1000 tests.

À noter que le code coverage ne concerne que les classes que nous avons développées et que nous pouvons tester. Malheureusement, un pourcentage important de ces classes reste non testable, car il s'agit de fichiers auto-générés par JavaCC et JJTree.

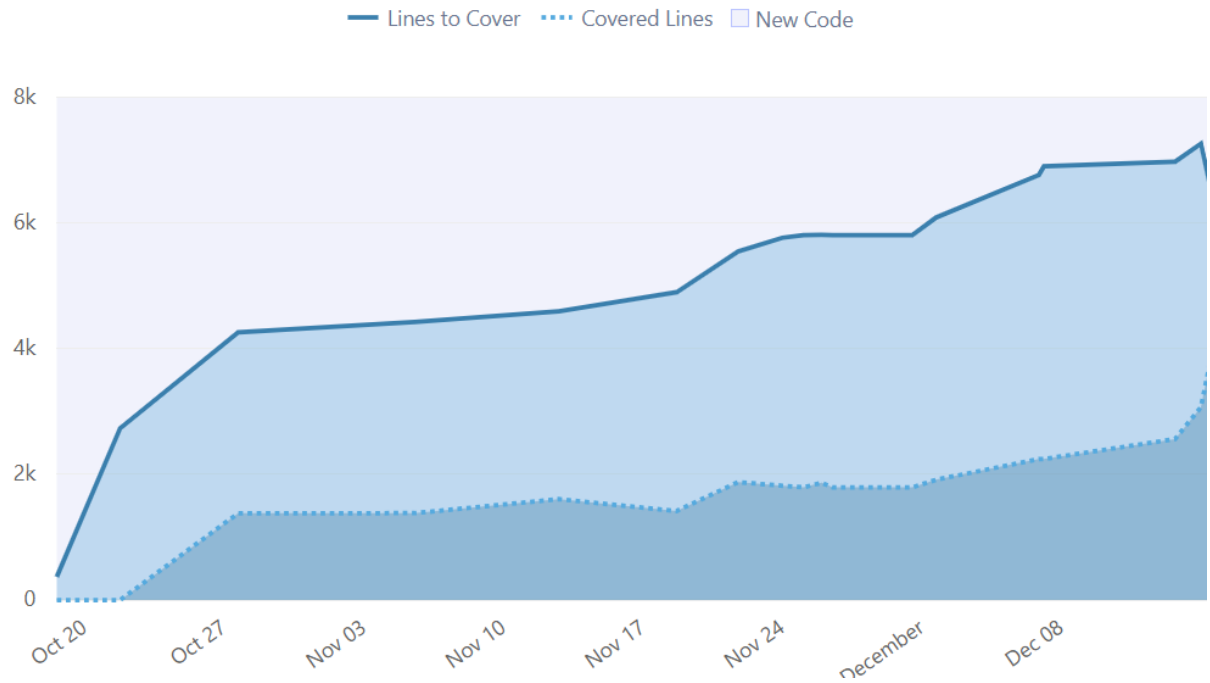


FIGURE 11 – Coverage du projet sur SonarQube

4 Applications des méthodes agiles

Dans le cadre de ce projet, nous avons travaillé avec des méthodes agiles pour structurer le travail, favoriser la collaboration et assurer une progression incrémentale vers les objectifs définis. Jira est l'outil principal que nous avons utilisé pour travailler de manière agile.
Nous expliquerons en quoi consistent les méthodes agiles et comment les appliquer avec Jira.

4.1 Organisation du groupe

Afin d'organiser le groupe, nous avons décidés très tôt des rôle. Premièrement, nous avons choisis un Scrum Master/Responsable de projet qui a pour rôle de s'assurer de l'avancement du projet, de la communication entre les différentes parties mais aussi de s'assurer de l'application des méthodes agiles. Pour le Scrum Master nous avons testé deux méthodes différentes. Au départ du projet, nous avons désigné une personne. Au sprint 3, sous les conseils de Mr Anthony Dugois, enseignant-chercheur à l'université, nous avons fait en sorte de choisir un nouveau Scrum Master au hasard parmi les développeurs toutes les semaines. Au vu du peu de résultats positifs que cette méthode rendait, nous sommes revenu à la première pour le sprint 5.

Enfin, nous avons choisis un des développeurs en tant que testeur. Sa mission consistait donc à organiser et effectuer les différents tests possible sur l'ensemble des modules. A noter que nous avons désigné un développeur pour ce rôle dû au fait que nous étions six dans le groupe, ce qui représentait un avantage par rapport aux autres groupes. Nous avons donc estimé que les cinq autres développeurs pouvait alors implémenter l'ensemble des fonctionnalités en plus de nous donner cet avantage du testeur.

4.2 Création des tâches

Afin de pouvoir travailler à plusieurs sur le projet, nous avons dû découper toutes nos missions en plusieurs tâches, aussi appelées tickets. Ces tickets font partie d'un ensemble appelé une User Story.

4.2.1 Création des User Stories

Les User Stories sont des phrases simples du type "*En tant que ... je voudrais ... afin de ...*" qui permettent de décrire le chemin parcouru par un utilisateur ou bien les fonctionnalités disponibles pour les utilisateurs. Pour ce projet, nous avons défini un certain nombre de User Stories. En voici quelques-unes :

- En tant qu'utilisateur, je voudrais pouvoir écrire du code MiniJaja (version réduite aux opérations arithmétiques et booléennes) afin de l'évaluer sur le terminal de l'interface
- En tant qu'utilisateur je voudrais pouvoir importer/sauvegarder des fichiers MiniJaja (UI) afin de sauvegarder mon code

Dans ces deux exemples, nous pouvons voir typiquement les différentes parties que peuvent toucher les User Stories. La première portant plutôt sur une face "cachée" du logiciel, tandis que la deuxième porte sur la partie graphique du logiciel, par conséquent bien plus visible.
C'est avec ces User Stories que nous avons pu planifier nos tâches et ainsi nous répartir les tickets au sein du groupe.

4.2.2 Planification des tâches

Les tickets Jira nous permettent de séparer le travail en plusieurs petites tâches à effectuer afin de faire avancer le projet. Notre stratégie au départ a été de rendre les tâches les plus petites possible afin de pouvoir les compléter en un ou deux jours au maximum. Comme nous pouvons l'observer sur la figure 12, ce ticket concerne la tâche de créer un module, ce qui prend tout au plus une dizaine de minutes. Ce ticket sera par exemple, complété par un autre qui sera de créer les dépendances inter-module, qui lui aussi sera assez court.

FIGURE 12 – Ticket création de module

Ce fut notre stratégie tout le long de la *Release 1*. En revanche, en arrivant à la *Release 2*, nous nous sommes rendus compte du peu de temps que nous avions pour travailler sur le projet. Nous avons donc décidé de créer des tickets prenant plus de temps et, ce faisant, le nombre de tickets créés a diminué.

La prochaine étape après la création des tickets est tout aussi importante. Elle concerne l'estimation des tickets. C'est une partie important car elle définira le nombre de points à valider lors du sprint. Ces points permettent de calculer l'avancement de l'équipe lors des sprints.
Pour effectuer ces estimations, nous nous basons sur une estimation théorique sur le temps que prend le ticket à être validé, en suivant la suite de Fibonacci, 1 étant un ticket très court et 8 un ticket très long à valider.

Pour ce qui est de l'attribution de ces tickets, nous choisissions lors de la planification des sprints lesquels nous étions susceptible de faire. Cette stratégie à résulté en l'affectation des mêmes tâches aux mêmes développeurs la plupart du temps. Typiquement, un développeur qui a fait le premier ticket de compilation s'est souvent occupé des autres tickets de compilation, souvent par soucis de temps ou de compréhension des mécanismes de développement.

4.3 Les sprints

Les sprints sont des cycles de travail courts et itératifs, d'environ deux semaines. Durant ces sprints, le groupe devait se concentrer sur un ensemble de tâches pour atteindre un objectif fixé en amont.

4.3.1 Planification des sprints

Pour la planification des différents, nous faisions des réunions au début de cycle. Ces sessions nous permettaient de définir l'objectif principal du sprint, qui devait être réalisable dans le temps imparti.
Une fois l'objectif établi, nous identifions les tâches prioritaires à effectuer pour atteindre cet objectif et effectuons la répartition entre les membres. Voici une liste des différents objectifs de sprints que nous avons établis :

- **Sprint 1** : Changement de la grammaire et établir le parseur.
- **Sprint 2** : Générer l'AST du MiniJaja, Avoir un éditeur de texte minimum, Interprétation MiniJaja pour les opérations arithmétiques.
- **Sprint 3** : Compilation pour les opérations arithmétiques et interprétation JajaCode pour les opérations arithmétiques et implémentation de la pile.
- **Sprint 4** : Implémentation du contrôleur de type, Correction des erreurs sur les interpréteurs pour les opérations arithmétiques.
- **Sprint 5** : Implémentation du tas, interprétations pour les méthodes et tableaux en MiniJaja, Implémenter le restant de la compilation.
- **Sprint 6** : Corriger les bugs, implémenter le pas-a-pas.

Il est important de noter que malgré le fait d'avoir établi des objectifs sur les sprints, certains n'étaient pas forcément atteints. Nous en discuterons dans la prochaine partie portant sur les rétrospectives des sprints.

4.3.2 Rétrospectives et analyses des sprints

Une fois les sprints terminés, nous avons organisé des revues. Ces revues nous ont permis de discuter du sprint précédent, d'échanger sur les fonctionnalités implémentées, celles qui restent à implémenter et de partager nos avis sur différents aspects du projet. Ainsi voici les différentes revues de sprint du projet :

Sprint 1 :
Objectifs réalisés :

- Première version de la pile
- Modification de la grammaire en LL(1)
- Écriture des lexèmes
- Écriture des règles de grammaire
- Première version de l'éditeur de texte
- Écriture des Tests du parser/lexer

Tâches restantes ou à planifier pour le prochain sprint :

- Créer l'AST
- Afficher l'AST

— Sauvegarder/importer fichiers (UI)
— Implémentation de la pile
— Commencer la compilation des opérations arithmétiques en JajaCode
— Implémentation de la CI/CD

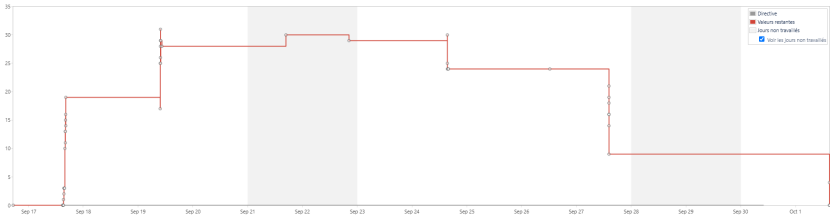


FIGURE 13 – Burndown Chart Sprint 1

Comme nous pouvons l'observer sur la figure 13, le sprint 1 a été un sprint bien planifié. En effet nous pouvons observer qu'à la fin du sprint nous avons pu valider tous les tickets et objectifs. En revanche, étant le premier on remarque une inexpérience sur la création des tickets au départ du sprint, ce qui explique cette montée des story points sur la première semaine.

Sprint 2 :
Objectifs réalisés :

— Création de l'AST MiniJaja
— Interpréteur MiniJaja pour opérations arithmétiques
— Implémentation de la table des symboles
— Écriture de tests levant des Exceptions
— Sauvegarder/importer fichiers (UI)
— Débuter l'implémentation du contrôleur de type

Tâches restantes ou à planifier pour le prochain sprint :

— Intégrer la CI/CD
— Contrôleur de type pour les variables et opérations arithmétiques
— Implémentation de la pile
— Interpréteur JajaCode pour les opérations arithmétiques
— Ouvrir plusieurs fichiers sur l'UI
— Dark Mode (UI)
— Afficher résultat de l'interpréteur (UI)

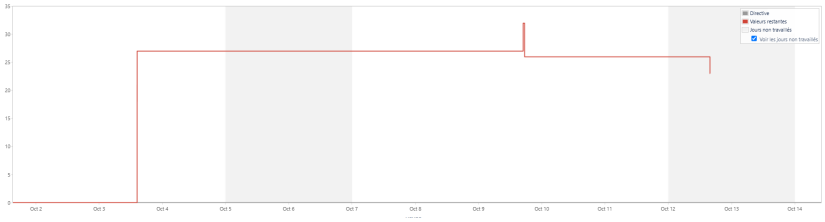


FIGURE 14 – Burndown Chart Sprint 2

Le sprint 2 fut plus compliqué pour le groupe. En effet, l'arrivée des premiers examens, additionné aux tâches plus complexe ont ralenti notre avancée en comparaison au prévisionnel. Il est tout de même important de noter que, malgré le peu de ticket validé, beaucoup étaient en réalité en attente de validation. Ce processus suivant beaucoup d'étapes (exécution des tests, validation par revu de code et approbation de merge sur la branche de développement), il semblait difficile d'organiser ce sprint correctement de notre point de vue.

Sprint 3 :
Objectifs réalisés :

- Interprétation JajaCode pour les opérations arithmétiques et implémentation des variables
- Implémentation de la pile
- Implémentation de la CI/CD
- Correction pour l'interprétation MiniJaja
- protection des branches main et test pour empêcher les push distant

Tâches restantes ou à planifier pour le prochain sprint :

- Planifier la release 2 et le sprint 4

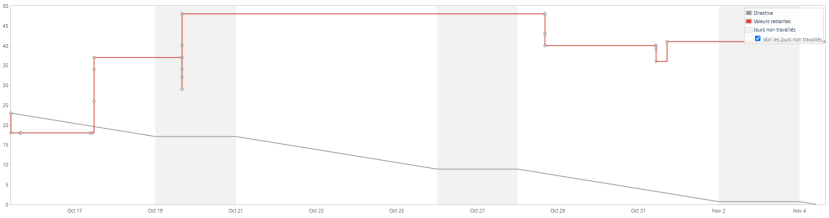


FIGURE 15 – Burndown Chart Sprint 3

Ici, la figure 15, démontre que nous avons su apprendre de nos erreurs au sprint 2. En répartissant et estimant au mieux les tickets, nous avons pu en valider certains pour la release 1. Par ailleurs, un manque de rigueur de notre part fait qu'une partie des tickets n'étaient pas placés en tant que validés. Cela explique le graphique et pourquoi le nombre de story points ne baisse pas. Ayant pu valider l'intégralité de nos objectifs pour la release 1, cela représente une grosse erreur de notre part au niveau de la méthode agile.

Sprint 4 :
Objectifs réalisés :

- Correction des erreurs d'interprétation du MiniJaja
- Correction de la grammaire pour les enchaînements d'opérations (typiquement x + x + x + x)

Tâches restantes ou à planifier pour le prochain sprint :

- implémentation du tas
- Faire le contrôle des types sur les variables et tableaux
- Implémenter l'ensemble du compilateur
- Implémenter l'interprétation MiniJaja pour les tableaux et méthodes

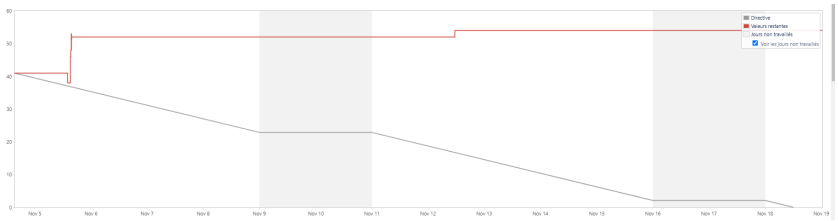


FIGURE 16 – Burndown Chart Sprint 4

Sur la figure 16, on remarque que très peu de tickets ont été validés par l'équipe. En effet, le sprint 4 a été marqué par le rendu imminent d'autres projets, notamment ceux liés aux graphes et au développement mobile. L'ensemble de l'équipe étant concerné, et le projet étant légèrement en avance à ce moment-là, nous avons pris la décision de nous consacrer pleinement aux rendus les plus urgents. Cela explique notre manque d'investissement durant ce sprint.

Sprint 5 :
Objectifs réalisés :

- Implémentation du tas
- Séparation du projet en modules Maven (package précédemment)
- Création des dépendances inter-Modules
- Finition du contrôleur de type
- Refonte de l'interface avec JavaFX
- Implémentation de plusieurs fonctionnalites interface graphique (undo, redo, explorateur de fichiers)

Tâches restantes ou à planifier pour le prochain sprint :

- Finir le compilateur pour les méthodes et tableaux du MiniJaja
- Implémentation des méthodes et tableaux pour JajaCode
- Implémentation du dark Mode

— Ajouter la coloration syntaxique sur l'éditeur de code

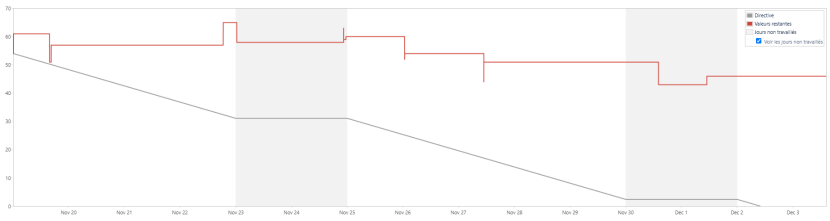


FIGURE 17 – Burndown Chart Sprint 5

Après un sprint 4 en échec, nous avons redoublé d'effort afin de faire de ce sprint 5 une réussite. La stratégie fut payante, car en effet le développemen à bien avancé. En revanche, comme le démontre la figure 17, nous retombons dans nos travers du sprint 3 en validant moins de tickets que le travail fourni ne représente. De plus un manque d'investissement dans le groupe se fait ressentir notamment à l'approche de certains examens.

Sprint 6 :
Objectifs réalisés :

— Complétion de la compilation
— Complétion de l'interprétation JajaCode
— Implémentation d'un logger pour le contrôleur de type
— Ajout de fonctionnalités visuelle pour l'interface (coloration syntaxique, numérotation des lignes, . . .)
— Implémentation du dark mode

Tâches restantes :

— Mise en place du pas-à-pas
— Mise en place des points d'arrêts
— Visualisation de la mémoire sur l'interface
— Redirection des erreurs du contrôleur de type sur un fichier

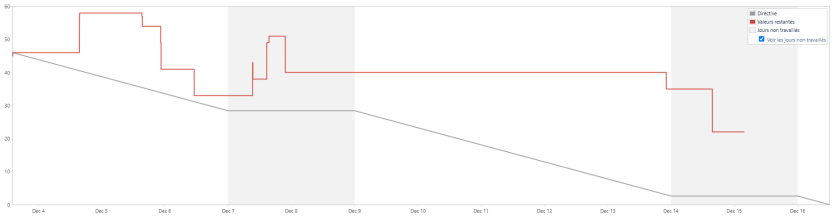


FIGURE 18 – Burndown Chart Sprint 6

Nous avons précédemment évoqué le manque d'investissement d'une partie du groupe. Afin d'y remédier, nous avons organisé une réunion avec M. Fabrice Bouquet, enseignant-

chercheur à l'université et superviseur des projets AVM. Suite à cette réunion, tenue durant la première partie du sprint, nous observons sur la figure 18 un pic de productivité notamment dans la validation des tickets. Une période de révisions pour les multiples examens a suivi, expliquant la stagnation observable au milieu du sprint. Enfin, après les examens, nous avons pu nous concentrer sur les derniers tickets afin de livrer un produit conforme aux attentes minimales pour la release 2.

4.4 Rétrospective du projet

Pour ce projet nous avions défini deux objectifs pour les deux release. La première release devait inclure toutes les étapes pour l'interprétation JajaCode sur les opérations arithmétiques et l'utilisation des variables (hors tableaux). Cela comprend l'implémentation de la pile, l'interprétation MiniJaja et la compilation. Enfin,la deuxième release devait contenir l'ensemble des attentes définis dans le sujet.

4.4.1 Prévisionnel face au réalisé

Release 1

La release 1, comprenant les sprints 1,2 et 3, a été un succès en termes de travail. Bien que nous ayons eu du mal à appliquer pleinement les méthodes agiles, nous avons su organiser notre travail et ainsi valider les tickets nécessaires pour remplir les objectifs fixés. De plus, les réunions de rétrospectives de sprint nous ont permis de mieux suivre l'avancement du projet et de corriger nos erreurs sur le sprint suivant. A la fin de la release 1, nous avions atteint nos objectifs tout en prenant une certaine avance, notamment sur l'implémentation de la mémoire et sur l'interface graphique.

Release 2

La release 2, comprend les sprints 4,5 et 6. Comme mentionné précédemment, le sprint 4 a été un échec en termes des objectifs. Le manque de temps, combiné aux échéances d'autres projets, nous a ralentis durant ce sprint. En revanche, nous avons su rattraper ce retard sur les sprints 5 et 6, en validant un maximum de tickets. Pour cela, nous avons dû faire des choix difficile, notamment renoncer à certaines fonctionnalités telles que l'interprétation pas-à-pas et l'exécution avec points d'arrêts. Cette release 2 est marqué par l'abandon des ces fonctionnalités afin de terminer au mieux les fonctionnalités minimum requises : interprétation MiniJaja, compilation et interprétation JajaCode.

4.4.2 Avantages et inconvénients des méthodes agiles

Avantages :

— Permet le travail collaboratif.

— Division du travail en tâches courtes.
— visualisation de l'avancement du projet avec Jira.
— Avancée incrémentale du projet.
— Permet de communiquer plus rapidement sur les problèmes et, par conséquent, permet de les régler plus rapidement

Inconvénients :

— Dépend de l'investissement de toutes les personnes du groupe.
— Demande une planification rigoureuse au début de chaque sprint / release.
— Coût en temps pour les réunions, gestion des tickets, . . .

5 Les outils de développement

Dans le cadre de ce projet, nous avons utilisé divers outils tels que Git, GitLab CI/CD, SonarQube et Nexus. Ces outils ont joué un rôle clé dans le contexte de développement agile dans lequel nous avons évolué tout au long de ce projet.

Dans un premier temps, nous expliquerons comment nous avons articulé l'utilisation de Git avec l'intégration continue pour garantir un code fonctionnel à tout moment. Ensuite, nous aborderons en détail le processus de déploiement sur Nexus, ainsi que l'utilisation de SonarQube et son utilité dans le projet. Enfin, nous conclurons avec une discussion sur la phase de release et son importance.

5.1 Intégration continue

Commençons par examiner l'intégration continue dans notre projet. Cependant, il est essentiel de comprendre le rôle de Git avant de détailler son interaction avec l'intégration continue.

Tout d'abord Git est un outil de gestion de version. Dans ce projet, il a été particulièrement utile pour versionner les différentes itérations de notre code. Nous avons adopté une utilisation structurée et collaborative de Git :

- Chaque membre de l'équipe travaillait sur sa propre branche, qui était une copie de test.
- Une fois qu'un membre avait terminé une fonctionnalité, il effectuait une merge request vers la branche de test commune.
- À ce stade, l'intégration continue jouait un rôle clé : un pipeline CI/CD était déclenché pour vérifier que les modifications proposées n'introduisaient pas de régressions ou d'erreurs de build.
- Si le pipeline était validé, un autre membre de l'équipe examinait les changements et approuvait la *merge request*.

Grâce à ce fonctionnement, nous avions en permanence un code fonctionnel et à jour sur la branche de test.

L'intégration continue nous a donc permis de garantir la stabilité de la branche de test en détectant rapidement d'éventuelles régressions. Nous avons utilisé GitLab CI/CD pour gérer l'intégration continue. Pour ce faire, nous avons défini un fichier .gitlab-ci.yml décrivant les différentes étapes des pipelines.

Les étapes de notre pipeline CI/CD étaient les suivantes :

```
stages:
  - build
  - test
  - deploy
```

FIGURE 19 – Étapes du pipeline CI/CD

Par exemple, l'étape de build était exécutée uniquement lorsqu'une merge request était réalisée :

```yaml
build-job:
  stage: build
  tags:
    - avm_groupe5
  rules:
    - if: $CI_PIPELINE_SOURCE == 'merge_request_event'
    - if: $CI_COMMIT_BRANCH == 'test'
  script:
    - mvn clean compile
```

FIGURE 20 – Étape de build

5.2 Déploiement

Après avoir décrit précédemment l'utilisation de l'intégration continue avec GitLab CI/CD et Git dans un contexte agile, nous allons maintenant nous concentrer sur le rôle de Nexus et SonarQube dans notre projet, en particulier pour le déploiement et l'analyse de la qualité du code.

Nexus a été un outil essentiel pour la gestion des versions des artefacts générés lors du processus de build, permettant ainsi un stockage centralisé et une gestion optimale des versions. Quant à SonarQube, il a joué un rôle clé en analysant notre code de manière approfondie, et en calculant, entre autres, le *code coverage* (le pourcentage de code couvert par des tests unitaires). Plus précisément, SonarQube a permis de détecter :

- Des incohérences dans le code : telles que des pratiques de codage non conformes aux standards, des duplications inutiles ou des dépendances redondantes.
- Des problèmes potentiels : comme des lignes de code susceptibles de générer des bugs ou des vulnérabilités de sécurité.
- Des métriques générales : telles que la complexité cyclomatique et la dette technique, nous aidant ainsi à mieux prioriser les actions correctives à entreprendre.

En intégrant SonarQube dans notre pipeline CI/CD, nous avons pu obtenir des retours immédiats sur la qualité de chaque merge request. Cela nous a fourni des informations précieuses pour améliorer la robustesse, la fiabilité et la propreté de notre code.

Conclusion

Pour conclure ce rapport, voici un récapitulatif du projet :

— Prototyper un compilateur et un interpréteur
— Réaliser un travail d'équipe
— Développer une interface utilisateur
— Tester le bon fonctionnement du logiciel et la véracité du compilateur et de l'interpréteur

Au vu des besoins de départ, nous dressons ici la liste des objectifs que nous avons pu réaliser au moment de l'écriture de ce rapport :

— Création de l'AST pour le langage minijaja
— Interprétation complète du MiniJaja
— Compilation complète du MiniJaja
— interprétation complète du JajaCode
— Contrôleur de type sur l'ensemble du MiniJaja
— IHM intuitif et agréable à utiliser comprenant de nombreuses fonctionnalités

Au travers des ces quatre mois de développement en équipe, nous avons énormément appris, tant sur le plan technique qu'humain. Ce projet nous a permis de développer nos compétences en programmation, en travail collaboratif, en communication, ainsi qu'en planification et gestion du temps.

Ce fut une expérience enrichissante sur beaucoup d'aspects. Comme nous avons pu le constater, le travail en méthode agile demande un investissement constant et une certaine rigueur. Cependant, c'est ce qui rend cette méthode efficace lorsqu'elle est correctement appliqué.

Nous vous remercions sincèrement d'avoir pris le temps de lire ce rapport et espérons que les résultats présentés répondront à vos attentes.

Annexe

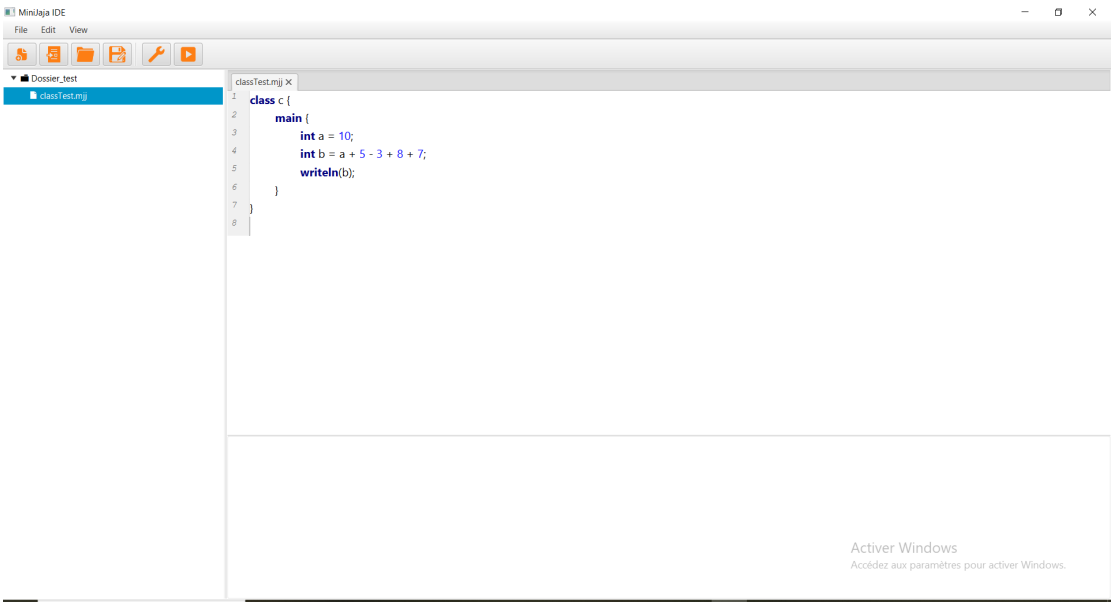


FIGURE 21 – Interface avec explorateur de fichier