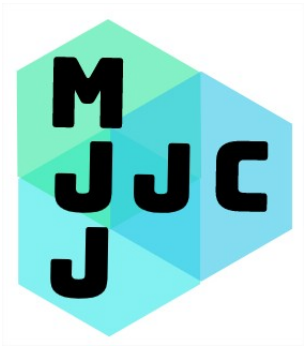
UFR Sciences et Techniques de Besançon

Projet de machines virtuelles
Master Informatique - 1re année

IDE MiniJaja



Étudiants :

Clément Aubry
Adrien Avenia
Corentin Borde
Alexandre Dillon
Loïc Grandperrin
Cynthia Maillard

Encadrants :

Fabrice Bouquet
Aymeric Cretin
Bruno Legeard

Année universitaire 2019-2020

Remerciements

Pour la réalisation de ce projet, nous tenons à remercier notre encadrant de TP, Monsieur Aymeric Cretin, ainsi que nos enseignants-chercheurs Monsieur Fabrice Bouquet et Monsieur Bruno Legeard sans qui ce projet n'aurait pas pu être réalisé.

Table des matières

Table des figures

Introduction

Dans le cadre de la première année en Master Informatique à l'*UFR*[1] *Sciences et Techniques* de Besançon, les étudiants doivent réaliser un projet de développement agile de machine virtuelle en équipe. L'objectif principal de ce projet est de concevoir un outil d'interprétation et de compilation d'un langage *Java* réduit appelé *MiniJaja* avec une organisation agile.

Ce présent rapport détaille le projet que nous avons réalisé, en équipe de six étudiants ainsi que notre organisation, utilisant la méthode agile *SCRUM* dans l'objectif d'appréhender les notions vues en cours et de mieux nous organiser tout au long de ce projet. La méthode *SCRUM* est basée sur des itérations de durées fixes nommées sprints. Lors d'un sprint, les membres de l'équipe choisissent des tâches à réaliser.

La première section de ce rapport présente le cahier des charges avec les besoins fonctionnels et non fonctionnels ainsi que la structuration de notre projet avec les choix de réalisation. La deuxième section décrit notre organisation lors du développement de ce projet. La troisième section présente la réalisation de nos différents modules. La quatrième section présente nos démarches dans une quête d'assurance qualité. Enfin, une section décrit les résultats obtenus et le rapport se conclue par une rétrospective de conduite du projet.

1. Unité de Formation et de Recherche

1 Présentation et structuration du projet de machines virtuelles

Dans cette section, nous présentons le projet avec un rappel du cahier des charges et des besoins demandés fonctionnels et non fonctionnels. Puis, nous décrivons le découpage du projet avec les différents modules le structurant ainsi que le modèle que nous avons utilisé afin d'établir un lien entre chaque module.

1.1 Cahier des charges

L'objectif attendu de ce projet est de pouvoir compiler le code *MiniJaja* et le transformer en code de bas niveau appelé *JajaCode*. L'utilisateur doit pouvoir interpréter l'un comme l'autre tout en visualisant l'état mémoire. Cet environnement de développement doit donc disposer d'une interface utilisateur, appelée *IHM*[2] permettant aux utilisateurs de gérer les différents fichiers *MiniJaja* ainsi que les différentes fonctionnalités associées. De plus, un affichage de la mémoire doit être présent lors de l'interprétation ainsi qu'une console permettant à l'utilisateur d'avoir un retour sur son code par l'affichage de variables ou d'éventuelles erreurs rencontrées.

Pour cela, plusieurs exigences ont été données dans ce sujet, décrivant à la fois des besoins fonctionnels et non fonctionnels.

Besoins fonctionnels

- Pouvoir compiler tous les programmes *MiniJaja* corrects et bien typés ;
- Pouvoir comparer les résultats de l'interprétation d'un programme *MiniJaja* et du programme *JajaCode* obtenu par compilation ;
- Pouvoir faire la compilation des programmes édités grâce à un éditeur de texte standard sous différents systèmes d'exploitation (Unix, Windows) ;
- Pouvoir observer le *JajaCode* résultant de la compilation ;
- Pouvoir observer l'interprétation pas à pas des programmes *MiniJaja* et *JajaCode* ;
- Pouvoir observer l'exécution d'un programme munis de points d'arrêt.

Besoins non fonctionnels

- La compilation et l'interprétation du langage *MiniJaja* doivent respecter les règles décrites dans les notes de cours disponible en annexe A ;
- Les messages d'erreurs doivent être les plus clairs possibles. C'est à dire qu'ils doivent localiser le plus précisément possible le lieu de l'erreur relativement au texte source. Ils doivent donner un message le plus explicite possible. Par conséquent, les erreurs doivent être détectées le plus tôt possible ;
- Les structures de données mémoire la pile et le tas doivent être mises en œuvre le plus efficacement possible. La réalisation des opérations doit être basée sur l'accès direct. Une pile avec des entrées par hachage est la structure conseillée

2. Interface Homme-Machine

pour la mémoire et une système de ramasse miette pour le tas. L'utilisation des techniques de hachage permet de satisfaire la contrainte d'accès direct au mieux ;

— La réalisation du projet doit conserver la structure des règles des notes de cours. En utilisant, les classes (*Java*) pour décrire la syntaxe abstraite, il est possible d'attacher à chaque classe d'arbre de syntaxe abstraite des opérations (interpréter, compiler, ...) qui réalisent les règles. La notion d'héritage doit permettre de partager les opérations identiques par le plus grand nombre de types d'arbres possibles. La hiérarchie des classes doit en tenir compte ;
— On doit pouvoir observer les états mémoire de fin d'exécution des programmes afin de les comparer ;
— L'ensemble du projet doit être réalisé dans un même environnement cohérent.

La figure 1 présente le processus fonctionnel attendu pour ce projet. Il faudra ainsi effectuer une analyse lexicale, puis syntaxique du code *MiniJaja* proposé par l'utilisateur pour réaliser une interprétation *JajaCode* et la compilation en *JajaCode* du programme. Enfin, l'interprétation du programme *JajaCode* résultant de la compilation avec l'affichage d'erreurs grâce à un contrôleur de type.

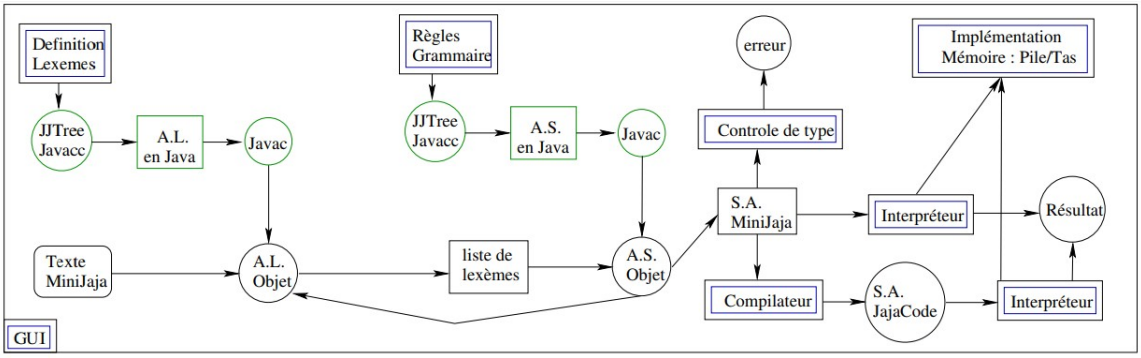


FIGURE 1 – Schéma du fonctionnement attendu

Pour réaliser ce projet, tous les membres de l'équipe ont utilisé l'*IDE*[3] *IntelliJ IDEA* avec *Java* 8 afin de développer dans un même environnement cohérent. Nous avons ainsi découpé notre projet en plusieurs modules et créé des dépendances entre celles-ci grâce à *Maven* pour séparer les différentes fonctionnalités.

1.2 Découpage du projet

Notre projet est constitué de cinq modules principaux qui sont *Lexer_Parser*, *AST*, *IHM*, *Interpreter* et *Memory*.

Chaque module est composé de deux répertoires séparant les fichiers principaux des fichiers de tests comme nous pouvons le voir sur la figure 2. Le module possède également un fichier de configuration pom.xml permettant de gérer la version de

3. Integrated Development Environment

celui-ci et d'établir des dépendances avec d'autres modules ou librairies.

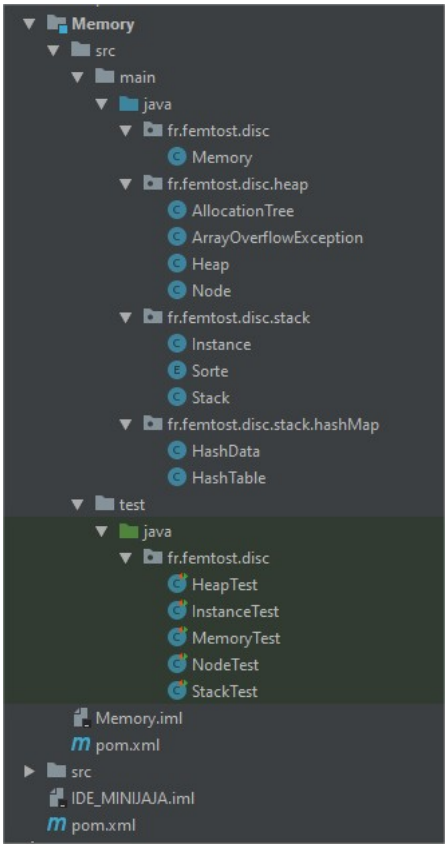


FIGURE 2 – Module mémoire du projet

Voici le rôle de chacun de ces modules :

— *Lexer-parser* : Le *lexer*, aussi appelé analyseur lexical, permet de lire le code *JajaCode* saisi et de convertir ces chaînes de caractères en une liste de symboles appelée *tokens* qui sont reconnus par le *parser*, analyseur syntaxique, qui crée les nœuds de l'*AST* dans un fichier généré automatiquement dans le répertoire *target* ;

— *AST*[4] : C'est un arbre de syntaxe abstraite qui est constitué des différents nœuds correspondants au code écrit par l'utilisateur en *MiniJaja*. Il est utilisé pour l'*interpreter*. Ce module est décomposé en plusieurs packages (AST, Compilation, TypeController et Visuals) ;

— *Interpreter* : Il visite l'*AST* pour appeler les différentes fonctions afin d'exécuter le code. Ce module est décomposé en plusieurs packages (*JajaCode* et *MiniJaja*) ;

— *Memory* : Ce module permet de stocker des objets ainsi que leurs valeurs associées. Ce module est décomposé en plusieurs packages (Heap, Stack, HashMap) ;

— *IHM* : Application visuelle permettant à l'utilisateur d'utiliser les différentes fonctionnalités. Ce module est également décomposé en plusieurs packages

4. Abstract Syntax Tree

pour séparer les vues, l'application et les contrôleurs (View, Controller et Application) et dispose également d'un dossier ressources pour utiliser des feuilles de styles.

Pour communiquer entre ces modules, nous sommes dans une configuration *MVC*[5] avec une approche du modèle *MVP*[6] en réutilisant les fonctionnalités développées dans les autres modules du projet.

MVC est un modèle utilisé pour séparer un projet en trois parties distinctes.
— Modèle : Le modèle est le code source de l'application, dans notre cas, les modules *AST, Interpreter, Lexer_Parser* et *Memory* englobent ce rôle.
— Vue : La vue est constituée de plusieurs fichiers *FXML* utilisant *JavaFX* qui est un *framework* et une bibliothèque d'interface utilisateur ;
— Contrôleur : Il va agir en fonction des actions réalisées par l'utilisateur en appelant les différentes fonctionnalités présentes dans chaque module.

1.3 Choix de réalisation

Ce projet abordant de nouvelles notions dans notre cursus, nous avons eu tout d'abord une phase de recherche afin d'aboutir aux choix exposés dans cette partie, notamment pour le lexer-parser et la construction de l'interface utilisateur.

Lexer-Parser

Pour notre module *Lexer_Parser*, plusieurs outils étaient à notre disposition. Chacun ayant ses avantages et ses inconvénients :

— *Antlr4*[7] : il reconnaît les récursivités à gauche et les supprime automatiquement. De plus, il génère l'*AST* en un seul fichier. Un des avantages d'*Antrl4* est de pouvoir utiliser la grammaire sans enlever les récursivités à gauche facilitant ainsi l'agrandissement de la grammaire ;
— *JavaCC*[8] qui a comme principal inconvénient de ne pas supporter les récursivités à gauche ;
— *JBison* et *JFlex* : ont l'inconvénient d'être séparés entre le lexer (*JBison*) et le parser (*JFlex*), de plus *JBison* ne crée pas automatiquement un *AST*.

Nous avons donc choisi d'utiliser *Antlr4*, correspondant aux besoins demandés pour ce projet.

IHM

Concernant l'*IHM*, il a été décidé de s'inspirer de celle que l'on peut retrouver au sein des différents IDE ou éditeurs de texte existants, en divisant l'affichage en

5. Model View Controler
6. Model View Presenter
7. ANother Tool for Language Recognition
8. Java Compiler Compiler

différentes zones :

- Une barre d'action pour ouvrir un fichier, sauvegarder un fichier, etc ;
- Une zone de texte à gauche pour l'écriture et l'affichage du code *MiniJaja* ;
- Une zone de texte au centre pour l'affichage du code compilé *JajaCode* ;
- Une zone de bouton d'action pour la zone de code *MiniJaja* et *JajaCode* pour compiler, interpréter, utiliser le pas à pas ;
- Une zone d'affichage destinée à l'état de la mémoire englobant tout l'espace latéral droit ;
- Une zone de texte dans la partie basse de l'interface pour l'affichage de la console.

La figure 3 représente l'idée générale que nous avons convenu pour l'organisation de l'*IHM*. La figure 4 représente les différents cas d'utilisation de l'*IHM*.

FIGURE 3 – Choix de la vue générale et simplifiée de l'*IHM*

Pour réaliser notre interface, nous avons choisi d'utiliser la bibliothèque *JavaFX*, offrant de nombreux éléments pré-conçus pour la création d'un *IDE*. Cet outil a notamment permis l'intégration d'une interface personnalisable avec la possibilité de modifier dynamiquement la taille des différentes zones. De plus, *JavaFX* nous offre la possibilité d'utiliser *SceneBuilder* qui est un outil de conception d'interface et qui est directement intégré à notre *IDE*.

1.4 Outils utilisés

Dans cette section, nous allons détailler les différents outils de développement agile que nous avons utilisé au cours de notre projet.

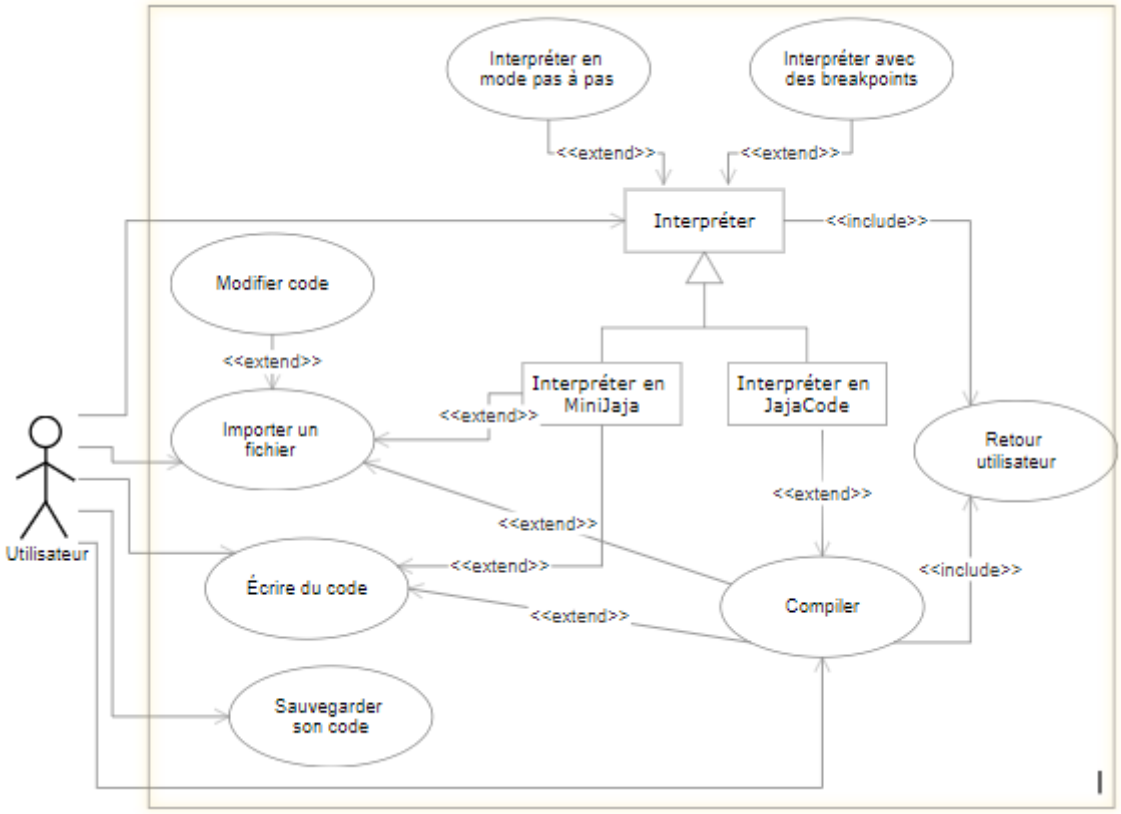


FIGURE 4 – Diagramme de cas d'utilisation de l'*IHM*

Gestion des tâches du projet : *Trello*

Dans l'objectif de suivre les activités de développement - les tâches en cours - nous avons utilisé l'outil de gestion de projet *Trello*. Il est basé sur une organisation de projets en colonnes répertoriant des fiches permettant de suivre leur état d'avancement. Nous avons également ajouté une extension permettant d'estimer nos tâches avec le concept de *Planning Poker*.

Gestionnaire de version : *Subversion*

Pour gérer les différentes versions du projet et assembler les différentes parties de développement de chacun des membres, l'utilisation d'un serveur de gestionnaire de version nous a été imposé avec *Apache Subversion*.

Un dépôt contenant les artefacts du projet se trouve sur ce serveur où tous les membres de l'équipe possèdent des copies de ces fichiers avec un accès, qu'ils peuvent éditer et envoyer sur le serveur. Chaque membre de l'équipe dispose d'un logiciel sous forme d'exécutable stand-alone ou de plug-in comme *TortoiseSVN*.

Le dépôt est constitué de deux répertoires, un permettant de travailler sur le projet appelé trunk et un autre répertoire contenant les différentes versions du projet nommé tags.

Outils d'intégration : *Jenkins* et *Nexus*

Durant le développement de notre projet, le département informatique nous a mis à disposition un serveur *Jenkins* qui est un outil open-source permettant de faire de l'intégration continue. Ce logiciel s'associe avec *Subversion* et *Maven*. Ainsi, *Jenkins* nous a permis de faire des builds automatiques à chaque commit sur le serveur *Subversion*. Cela nous permettait notamment d'avoir un aperçu de la stabilité de chaque module du projet avec les tests en réussite ou en échec.

Nous avons également utilisé *Nexus* qui est un gestionnaire de dépôt *Maven*, pour assurer la disponibilité de nos librairies aux autres développeurs/utilisateurs. Il contient notamment les fichiers .jar de nos deux releases.

Qualité du code : *SonarQube*

Pour mesurer la qualité du code, nous avons utilisé le logiciel libre *SonarQube* qui nous était mis à disposition également. Ce logiciel effectue des rapports à différents niveaux, comme :

- la détection de bugs ;
- la couverture des tests de composants sur le code ;
- la duplication du code ;
- les vulnérabilités.

Ce logiciel a été utilisé de manière entièrement automatisé grâce à l'outil *Jenkins*.

Communication au sein du groupe : *Discord*

Pour faciliter les échanges entre les membres de l'équipe à distance, nous avons utilisé le logiciel *Discord* comme outil de communication qui nous a permis de créer un serveur dédié à notre projet avec plusieurs canaux textuels et vocaux. Ces canaux permettaient à chaque membre de l'équipe de faire part de ses difficultés rencontrées, et de faire des points de situation sur les différents modules en développement grâce à leurs canaux dédiés.

On remarque sur la figure 5 les différents canaux de communication de notre serveur *Discord*.

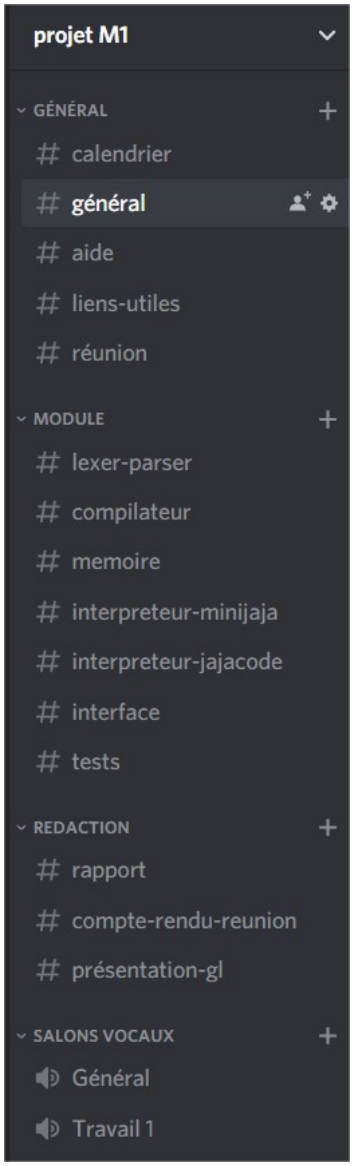


FIGURE 5 – Serveur *Discord*

2 Organisation agile de l'équipe et sprints

2.1 Organisation agile

Afin de mener à bien notre projet, nous avons travaillé avec une philosophie agile et plus précisément avec la méthode *SCRUM*.

Notre organisation SCRUM

Tout d'abord, le projet à été découpé en six phases appelées *sprints* et deux livraisons appelées *releases* placées au milieu et à la fin du projet comme présenté dans la figure 6 représentant notre calendrier de projet.

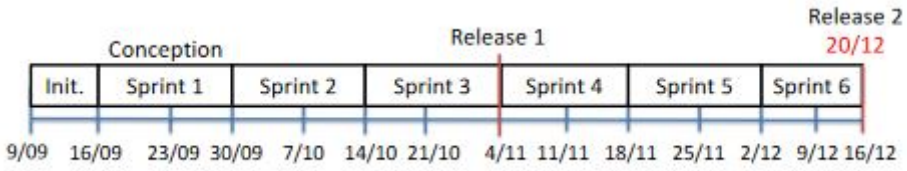


FIGURE 6 – Calendrier des *sprints*

La méthode *SCRUM* fonctionne avec plusieurs cérémonials que nous avons planifié à l'échelle d'un projet scolaire.

Pour chaque *sprint*, nous faisions une planification en début d'itération et une revue une fois le *sprint* terminé.

Nous regroupions ces deux cérémonials en une seule réunion entre deux *sprints*. Nous commencions par la revue du sprint précédent pour discuter de la réalisation des tâches de chacun des binômes et nous soulevions les problèmes rencontrés. Ensuite, nous définissons les tâches à effectuer pendant le prochain *sprint* ainsi que la répartition des différents binômes.

À notre échelle, les mêlées quotidiennes des équipes agiles travaillant avec la méthode *SCRUM* étaient réalisées une à deux fois par semaine afin de connaître l'avancée de chacun des membres de l'équipe et de s'assurer que personne ne restait en échec en proposant des solutions aux problèmes rencontrés.

L'organisation en binômes au sein de l'équipe a permis un travail efficace en *pair-programming*. La répartition des binômes était réalisée afin que chaque membre de l'équipe travaille au moins une fois sur chaque module et une fois avec chacun des autres membres du groupe.

Enfin, les objectifs à atteindre pour chacune de nos *releases* étaient représentés sous forme de *User-stories*.

Trello

L'outil *Trello* s'est avéré particulièrement efficace pour la répartition des tâches pour chaque *sprint* ainsi que pour les *backlogs* de produit.

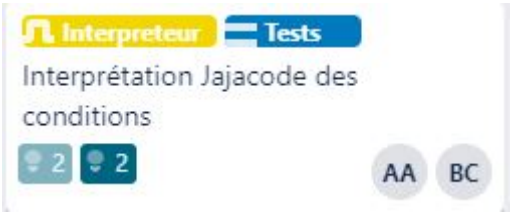


FIGURE 7 – Exemple d'une fiche *Trello*

Les fiches *Trello* permettent de représenter chacune des tâches à effectuer dans

le tableau de *sprint*. L'exemple de la fiche de la figure 7 nous permet de voir les différentes informations représentant une tâche. Chacune des fiches se compose d'une description, des membres affectés, des étiquettes qui correspondent aux modules, le temps prévisionnel et le temps effectué. Ces deux temps ont été ajoutés à l'aide d'un *plugin BurndownForTrello*[9] qui va générer à partir de ces temps un *Burndown chart*.

Comme nous pouvons le voir sur la figure 8, toutes ces fiches sont réparties dans différentes colonnes qui sont :

— À faire
— En cours
— À tester/valider
— Fait
— À reporter/terminer

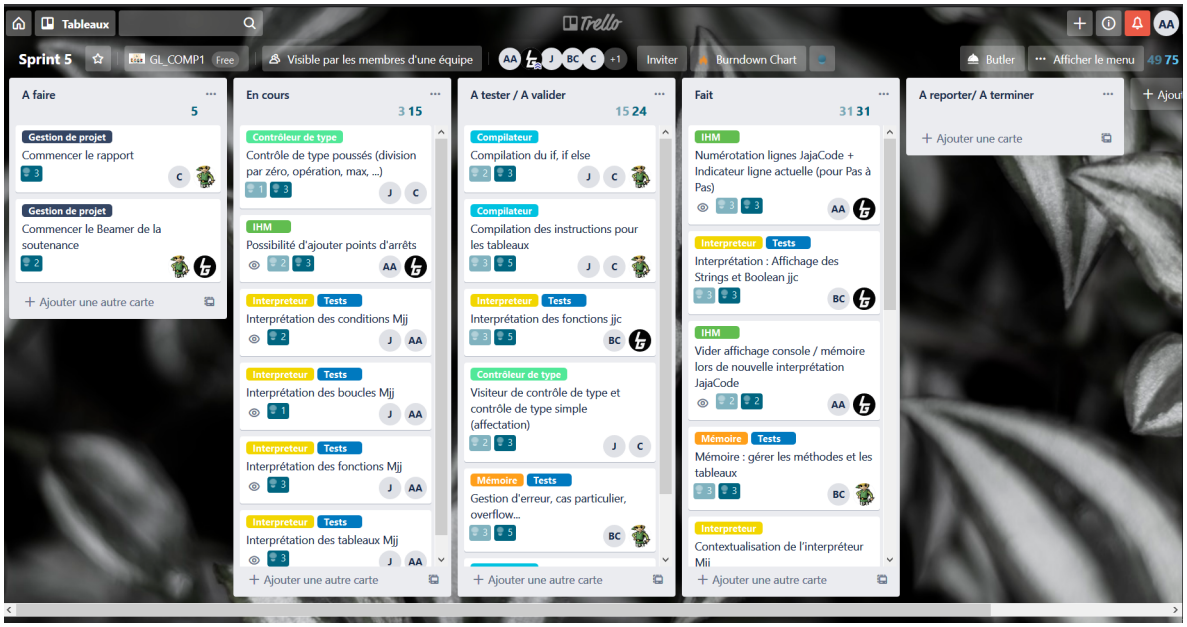


FIGURE 8 – Tableau *Trello* du *sprint* 5

Burndown Chart

La mesure de l'avancement des *sprints* est un élément très important dans la méthode *SCRUM* et est réalisée grâce à un *Burndown chart*.

Sur la figure 9, nous pouvons voir les différents temps pour un sprint :
— Jaune : le temps idéal qui serait réalisé ;
— Bleu : le temps qu'il reste à réaliser ;
— Rouge : le temps déjà réalisé.

9. `https://trello.com/power-ups/569f8a70a115d18c5ea9af05/burndown-for-trello`

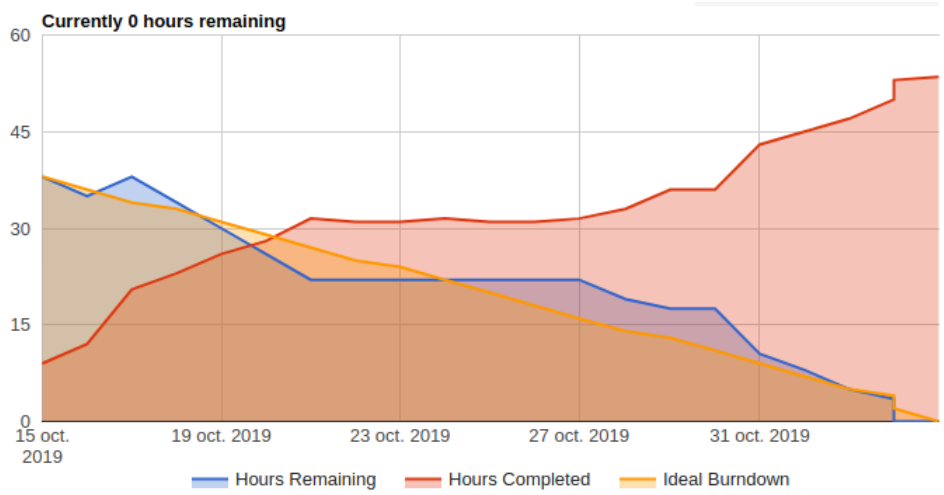


FIGURE 9 – Burndown Chart du *sprint* 3

Grâce à cet outil, on peut rapidement voir si des problèmes ont été rencontrés au cours d'un *sprint* et ajuster notre méthode de travail pour les prochains *sprints*.

2.2 Sprints et Releases

Afin de suivre la philosophie agile en développant de manière itérative et incrémentale, nous avons agrandit la grammaire au fur et à mesure de nos *sprints*.

Sprint 1

Lors de la planification du *sprint* 1 les objectifs principaux étaient de créer le projet *Maven* dans *InteliJ Idea* et de configurer les différents outils relatifs à la méthode agile.

Nous devions aussi mettre en place notre *lexer-parser* afin de commencer l'écriture de la grammaire.

Pour ce *sprint* la grammaire devait permettre :
— La déclaration des variables ;
— Les types ;
— Les identifiants ;
— La déclaration du main.
À partir de cette grammaire nous devions créer l'AST.

Pour le module *IHM*, les différentes zones devaient apparaître et nous devions faire l'implémentation de l'importation et de la sauvegarde d'un fichier *MiniJaja*. Enfin, nous devions commencer à nous intéresser au fonctionnement de la mémoire et des structures de la mémoire (la pile et le tas) ainsi que la fonction de hachage.

En conclusion de ce sprint, tous les objectifs ont été atteints malgré des difficultés rencontrées au niveau de la fonction de hachage pour la pile.

Sprint 2

Lors de ce *sprint* nous n'avons pas ajouté d'éléments à notre grammaire afin de nous concentrer sur le fonctionnement de tous les modules sur la grammaire actuelle.

Nous nous sommes focalisé sur les 3 éléments suivants :
— La mémoire ;
— L'AST ;
— La compilation.
Nous avons commencé à créer les visiteurs pour l'AST ainsi que la contextualisation de nos éléments à ajouter dans la mémoire.

Pour la compilation, un début de gestion d'erreur a été effectué pour la déclaration d'une variable déjà déclarée, en plus de la génération du *JajaCode* sur la grammaire *MiniJaja* actuelle. Enfin, la fonction de hachage de la pile a été améliorée.

Sprint 3 et Release 1

Le *sprint* 3 clôturait le développement du projet pour la première livraison lors de la *release* 1.

Voici les différents objectifs que nous nous étions fixé pour cette *release* :
— Importer et sauvegarder un fichier ;
— Écrire du code *MiniJaja* ;
— Déclarer une classe, un *main* et des variables ;
— Effectuer des opérations simples sur les variables (addition, soustraction...) ;
— Faire des affichages dans la console ;
— Compiler le code *MiniJaja* de la grammaire actuelle pour obtenir le code *JajaCode* correspondant ;
— Voir le code *JajaCode* généré ;
— Interpréter le code *MiniJaja* sur la grammaire actuelle ;
— Interpréter le code *JajaCode* obtenu lors de la compilation.

Pour le *sprint* 3 nous avons dû étendre la grammaire pour la gestion des classes, des opérations simples, des affectations et du *write* permettant l'affichage dans la console.

Nous avons commencé à travailler sur l'interprétation *MiniJaja* et *JajaCode* pour tous les éléments de la grammaire actuelle, modules dans lesquels nous n'avions encore rien développé jusqu'ici.

Dans l'*IHM* nous avons ajouté les différents boutons pour effectuer la compilation et l'interprétation.

Les différentes fonctions liées à l'ajout des nouveaux éléments dans la grammaire ont été créées pour la gestion mémoire. De même, le visiteur pour la compilation a été développé pour gérer ces nouveaux éléments.

Malgré un petit problème pour l'affichage des chaînes de caractères et des booléens lors de l'interprétation *JajaCode*, nous avons atteint tous les autres objectifs fixés pour la *release* 1.

Sprint 4

Dans le *sprint* 4, nous devions continuer d'agrandir la grammaire afin d'avoir l'incrément, les conditions, les boucles et les tableaux.

Nous avions à implémenter ces changements de grammaire pour la compilation et l'interprétation *MiniJaja* et *JajaCode*. Au niveau mémoire, nous avions à utiliser le tas et à l'améliorer pour l'utilisation des tableaux. Enfin, nous devions améliorer notre gestion des erreurs.

Lors de la revue de ce *sprint*, nous avons dû reporter au *sprint* suivant certains éléments qui n'avaient pas pu être traités : l'interprétation *MiniJaja* sur la grammaire actuelle et l'ajout des conditions dans la grammaire.

Sprint 5

En plus des tâches reportées du *sprint* 4 vers le *sprint* 5 nous avons ajouté les fonctions dans la grammaire *MiniJaja*.

Nous avions donc à finir l'interprétation *MiniJaja* du *sprint* et y ajouter l'interprétation des fonctions. Pour l'interprétation *JajaCode* nous avions à implémenter l'interprétation sur les nouveaux éléments.

De plus, nous devions commencer à développer le mécanisme de pas à pas au niveau de l'*IHM* pour l'interprétation *JajaCode*. Enfin, le contrôleur de type a été ajouté afin de gérer les erreurs manquantes.

Quelques problèmes ont été soulevés comme un décalage pour la compilation des fonctions ou la gestion des paramètres des fonctions pour l'interprétation. Ils seront à résoudre lors du *sprint* 6.

Sprint 6 et Release 2

Le *sprint* 6 est le dernier *sprint* du projet. Voici les différents objectifs à atteindre afin de rendre un projet terminé lors de la livraison pour la *release* 2 :

— Reconnaître toute la grammaire *MiniJaja* ;
— Interpréter en *MiniJaja* et en *JajaCode* toute la grammaire ;
— Compiler tous les éléments de la grammaire ;

— Utiliser l'interprétation pas à pas en *MiniJaja* et en *JajaCode* ;
— Utiliser des points d'arrêts lors de l'interprétation *MiniJaja* et *JajaCode* ;
— Afficher les erreurs dans la console ;
— Visualiser l'état de la mémoire.

Pour ce *sprint* nous devons donc développer le pas à pas et les points d'arrêts pour l'interprétation *JajaCode* et *JajaCode*. De plus, les problèmes du *sprint* 5 sont à résoudre afin de terminer le projet.

2.3 Difficultés rencontrées

Travailler en équipe agile sur un tel projet fût une véritable découverte pour tous les membres de l'équipe et certaines difficultés ont été présentes.

La première difficulté était d'attribuer les tâches à faire dans les *sprint* et d'y évaluer le temps requis. Cette difficulté a vite été surmontée grâce à l'expérience que nous avons acquise et nous avons été plus efficace au cours des différents sprints.

Ce projet étant réalisé dans un cadre scolaire, différent d'un projet en cadre professionnel, le temps consacré à d'autres unités d'enseignement pouvait parfois perturber le bon déroulement de nos *sprints*. De plus, le partage des tâches dans une équipe de 6 a parfois été difficile à organiser pour des travaux en binôme lors de précédents projets mais le travail en *pair-programming* était en revanche une méthode que nous connaissions.

Enfin, la prise en main des différents outils et la création du serveur d'intégration continue *Jenkins* nous était aussi inconnue et nous a demandé un temps d'adaptation. Certains de ces outils (*Jenkins* et *SonarQube*) ont également été en panne à plusieurs reprises perturbant le bon déroulement de notre développement.

3 Implémentation

En s'appuyant sur le cahier des charges, sur la structure générale de notre projet ainsi que sur nos choix de réalisation présentés précédemment, nous allons maintenant détailler la réalisation de chaque partie du projet.

3.1 Lexer_Parser

L'analyse lexicale et syntaxique au sein du module *Lexer_Parser* a pu être automatisée avec *Antlr4*, et s'est effectuée sur l'intégralité de la grammaire *MiniJaja* que l'on nous a fourni. En résultat, nous avons obtenu une génération automatique de l'*AST*, avec des visiteurs associés pour chaque nœud.

3.2 Compilation

La compilation permet de transformer le code de haut niveau nommé *MiniJaja* en code de bas niveau appelé *JajaCode*. Pour réaliser cette transformation, nous utilisons un arbre de syntaxe abstraite (*AST*) que nous parcourons à l'aide d'un visiteur fourni par *Antlr4* nous permettant ainsi de construire le *JajaCode*.

Dans cette partie du projet, la classe `EvalVisitor` va se charger de visiter tous les nœuds de l'arbre et retourner une instance de l'objet
`CompilationResult`. Cet objet contient une liste d'instructions *JajaCode*, une liste d'erreurs et l'index de la ligne de la méthode *main*, qui permet de savoir où il se situe dans la liste d'instructions *JajaCode*.

Pour chaque nœud visité, si c'est une feuille de l'arbre, alors une instruction *JajaCode* va pouvoir être créée et ajoutée à la liste d'instructions, sinon le visiteur va continuer de parcourir les nœuds fils du nœud visité.

3.3 Mémoire

La mémoire, représentée par la classe *Memory*, se compose de deux structures globales : la pile et le tas ayant chacun des fonctionnalités différentes. La figure 10 est un diagramme de classe représentant l'ensemble des classes utilisées par la mémoire.

La pile

La pile est la structure de données se chargeant du stockage des variables et de leur valeur, ainsi que de tout stockage de valeurs temporaires, telles que nécessaires lors de l'appel de méthode, ou de la réalisation de différents calculs. Elle est représentée par la classe *Stack*, directement contenue dans la classe *Memory*.

Il a été décidé d'encapsuler les valeurs stockées dans des *Instances*, permettant notamment de retenir le type de la valeur et sa nature (valeur d'une variable, d'une constante ou valeur temporaire au sein d'un calcul).

Le principe usuel d'une pile est assurée par des références au sommet et au bas de la pile, et par la présence de nombreuses opérations de manipulation standard : push, pop, load et swap. Aussi, les instances ont été implémentées en respectant le principe d'une liste chaînée, chacune ayant une référence vers la prochaine dans la liste.

Une table de hachage (représentée par la classe *HashTable*) a été mise en place pour le stockage des variables (au sein d'*Instance*), permettant d'accélérer les performances lors d'un accès. Nous avons choisi d'implémenter au sein de cette table la fonction de hachage *CRC32*, qui a pour particularité de minimiser le nombre de collisions. La contextualisation réalisée lors de la création de l'*AST* a d'autant plus permis cette réduction de collisions, grâce à l'utilisation de déclarations uniques.

Le tas

Le tas est une structure de données qui permet d'allouer et de stocker les valeurs contenues dans un tableau, ainsi que de gérer la place occupée par chacun. Il est représenté par la classe *Heap*, directement contenue dans la classe *Memory*.

Cette classe se décompose en deux structures principales : un tableau *memory*, et une classe *AllocationTree*, représentant un arbre binaire.

Concernant le tableau *memory*, il permet de stocker les valeurs contenues dans chacun des tableaux alloués, en associant l'adresse du tableau et l'index interne d'une de ses valeurs, à un index au sein de *memory*. En parallèle, l'utilisation d'un arbre binaire permet de gérer efficacement l'espace utilisé par chaque tableau, et d'optimiser l'adressage. L'arbre binaire a notamment une taille limitée, et permet d'informer lorsque l'allocation d'un tableau trop grand n'est plus possible.

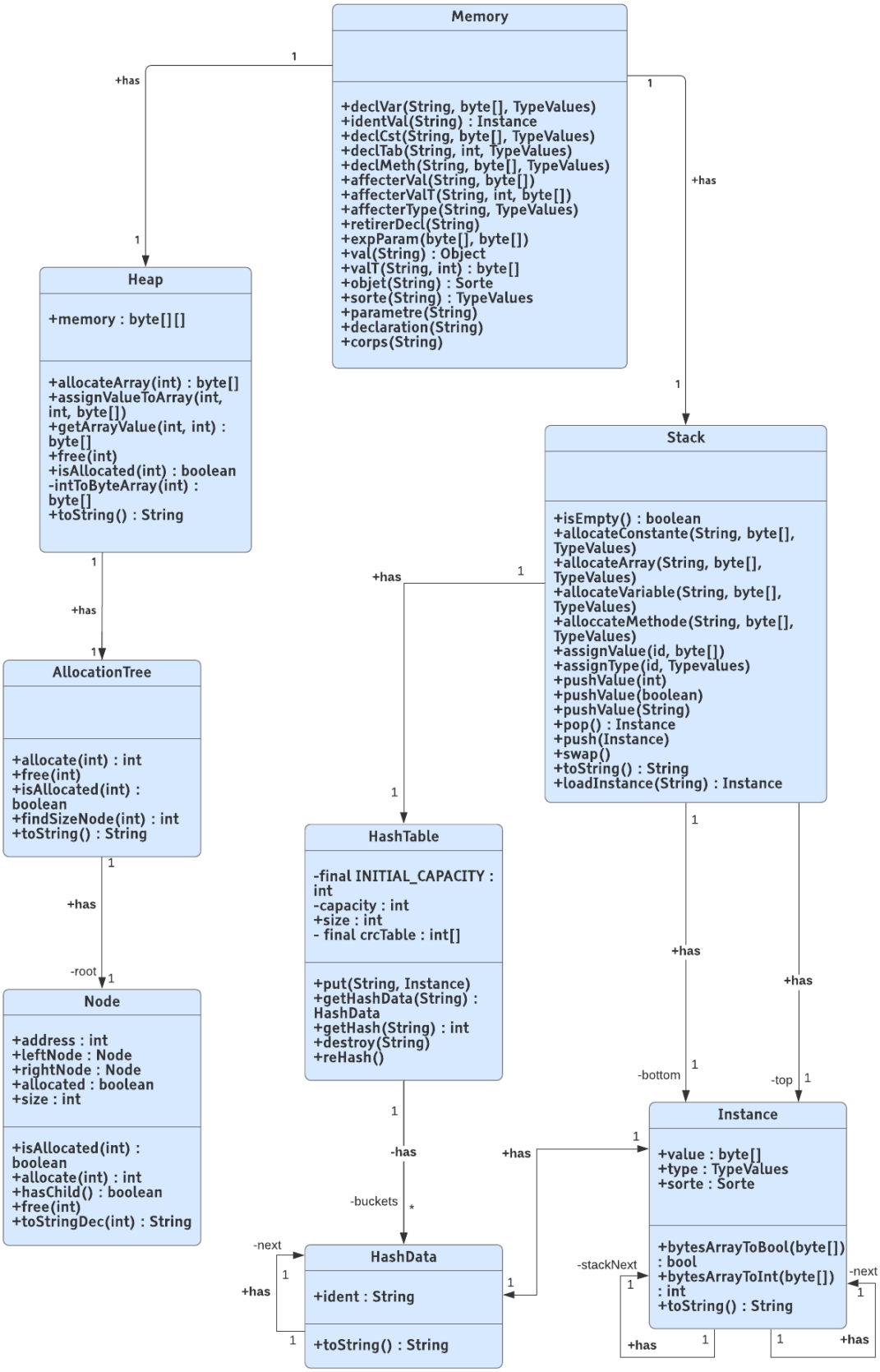


FIGURE 10 – Diagramme de classe du module *Memory*

3.4 Interprétation

MiniJaja

L'interprétation du *MiniJaja* est réalisée avec un visiteur qui va parcourir récursivement les nœuds de l'*AST* créé à partir du code.

L'interpréteur *MiniJaja* se compose d'une classe *InterMjjVisitor* qui hérite de la classe *MiniJajaBaseVisitor* qui est générée automatiquement par le plugin ANTLR4 du lexer-parser.

Elle retourne une instance de l'objet *InterpretationMjjResult*. Cet objet contient une liste d'erreurs et une chaîne de caractères représentant les résultats de l'exécution.

C'est dans la classe *InterMjjVisitor* que le visiteur parcours l'ensemble des nœuds de l'*AST*. Une instance de la classe *Memory* est initialisée. Lors de l'interprétation des feuilles de l'arbre, la classe *InterMjjVisitor* fait appel aux différentes fonctionnalités de la mémoire, en l'occurrence au tas et à la pile.

JajaCode

L'interprétation du *JajaCode* se fait de manière linéaire : chaque instruction est interprétée l'une après l'autre ou renvoie vers une instruction ciblée par un numéro de ligne. Ainsi, pour chaque instruction, on appelle les fonctions présentes dans la mémoire.

La classe *InterpreterJC* permet de réaliser l'interprétation du *JajaCode*. Les instructions *JajaCode* sont récupérées par l'*IHM* après la compilation et envoyées à l'interpréteur sous forme d'une chaîne de caractères que l'on décompose en une liste d'instructions sous forme de tableau. Une instance de la classe *Memory* est initialisée. L'interpréteur va donc pouvoir traiter chaque ligne une à une de la liste d'instructions et pouvoir faire les différentes opérations nécessaires sur la mémoire. L'interprétation en mode pas à pas du *JajaCode* a été intégré facilement puisqu'il suffit de s'arrêter à chaque instruction ou retourner à une instruction ciblée par un numéro de ligne lorsque l'utilisateur passe à l'instruction suivante. L'interprétation avec des points d'arrêts a été réalisé de la même manière.

Si des erreurs sont levées pendant l'interprétation, l'exécution s'arrête et les erreurs sont ajoutées dans une liste qui sera affichée dans la console de l'interface.

3.5 Contrôleur de type

Pour le contrôleur de type nous avons choisi d'utiliser deux visiteurs qui sont utilisés à chaque modification du code *MiniJaJa* afin de prévenir au plus tôt l'utilisateur des erreurs de type et de valeur. En effet, un premier visiteur a pour charge de répertorier les différentes déclarations par leur nom et leur type. Cela permet ensuite, lors d'autres déclarations, de vérifier si le nom n'est pas déjà utilisé, ou

lors des instructions, de vérifier si le typage est respecté. Un second visiteur permet quand à lui de vérifier les valeurs, c'est-à-dire qu'il vérifie si des valeurs interdites sont utilisées (division par zéro et valeur max) ou si des variables sont utilisées sans avoir été initialisées. Pour cela nous devons simuler une mémoire pour réaliser des calculs simples sur les valeurs.

En plus des vérifications de déclaration et de typage, le premier visiteur intègre la contextualisation.

3.6 Contextualisation

La contextualisation est un choix d'implémentation qui nous permet de gérer les scopes et les surcharges de fonctions sur l'ensemble du projet. Il s'agit d'un renommage des variables, constantes et méthodes en fonction de leur contexte. Un *context* est défini par le niveau de la position dans le code *MiniJaJa* et permet de gérer, à la compilation, les scopes et supprime ainsi le problème de plusieurs instances par un identificateur dans la pile. Le contexte est défini par le nom de classe, la position ou non dans le *main*, et la position ou non dans une fonction. Les méthodes profitent en plus de cela , d'un renommage en fonction des types de leurs entêtes, ce qui permet de gérer la surchage de fonction à la compilation. Le contrôleur de type qui intègre ces fonctionnalités nous permet donc de réécrire le code *MiniJaJa* avec des variables contextualisées et des méthodes surchargées.

3.7 Gestion d'erreurs

La gestion d'erreurs est réalisée dans tous les modules du projet. Une grande partie des erreurs sont détectées dans le contrôleur de type car le visiteur nous permet de les détecter facilement. Mais certaines erreurs ne sont pas détectables dans cette partie, notamment les erreurs pendant l'interprétation, par exemple les *overflow*. Ces erreurs sont donc gérées dans les modules *Interpreter* et *Memory* et sont levées pendant l'exécution contrairement aux erreurs prises en charge par le contrôleur de type, levées au cours de la compilation.

3.8 IHM

Le module *IHM* est composé de vues, de contrôleurs et d'un *main* de l'application.

Les vues sont les fichiers *XML* générés par l'outil *scene builder* et utilisés par le *framework FXML*. Les contrôleurs reprennent les différents élément *FXML* et définissent leur comportement suivant l'action de l'utilisateur. Enfin, le *main* de l'application permet de charger la vue et d'initialiser les paramètres de base.

L'interface utilisateur se décompose en plusieurs zones. Chacune de ces zones a ses propres fonctionnalités.

La barre de menu

En haut de notre application, nous avons une barre d'action qui correspond à notre menu.

On y retrouve plusieurs onglets :

— *File* : Permet d'ouvrir un fichier existant ou de sauvegarder la saisie en cours dans un fichier *.mjj* ;

— *Debug* : Permet d'utiliser le mode pas à pas pour l'interprétation *MiniJaja* et *JajaCode*.

La zone *MiniJaja*

La zone correspondante au *MiniJaja* se sépare en deux parties :

La partie haute avec les boutons d'actions :

— Compiler : Permet de compiler le programme écrit dans la zone *MiniJaja* et de générer le code *JajaCode* correspondant s'il n'y a pas d'erreurs. Le cas contraire, un message d'erreur sera affiché dans la console.

— Interpréter : Permet d'interpréter directement le code *MiniJaja* sans avoir besoin de le compiler. Les affichages indiqués dans le code ou les erreurs d'interprétation sont affichées dans la console.

— Boutons pas à pas : Les 3 boutons de la seconde ligne sont utilisés pour faire le débogage en pas à pas ou avec des points d'arrêt. C'est une alternative à l'onglet dans la barre de menu.

La seconde partie correspond à la zone de saisie.

Le code saisi est reconnu et une coloration syntaxique est faite. Les lignes sont numérotées et il est possible de placer des points d'arrêt à gauche de la colonne des numéros de ligne.

La zone *JajaCode*

La zone correspondante au *JajaCode* se sépare elle aussi en deux parties similaires au *MiniJaja* :

La partie haute avec les boutons d'actions :

— Interpréter : Permet d'interpréter le code *JajaCode* contenu dans la zone de texte. Les affichages indiqués dans le code ou les erreurs d'interprétations sont affichées dans la console.

— Boutons pas à pas : Les 3 boutons de la seconde ligne sont utilisés pour faire le débogage en pas à pas ou avec des points d'arrêt. C'est une alternative à l'onglet dans la barre de menu.

La seconde partie est la zone de texte *JajaCode*. Le code écrit dans cette zone ne peut pas être édité par l'utilisateur car il est généré par la compilation du code *MiniJaja*.

On retrouve aussi les numéros de ligne et la zone pour pouvoir ajouter des points d'arrêt pour le débogage.

La console

En dessous des zones de texte *MiniJaja* et *JajaCode* nous avons la console.

Dans cette zone nous affichons les résultats de l'interprétation *MiniJaja* et *JajaCode* lors de l'usage de l'élément *write*. En cas d'erreurs dans la compilation ou l'interprétation, les erreurs sont affichées en rouge et aucun résultat n'apparaît.

La mémoire

À droite de l'application se trouve une zone où l'état courant de la mémoire est affiché lors de l'exécution. On retrouve l'état de la pile et du tas.

Elle se met à jour lors de chaque étape du débogage en pas à pas ou lors d'utilisation des points d'arrêt.

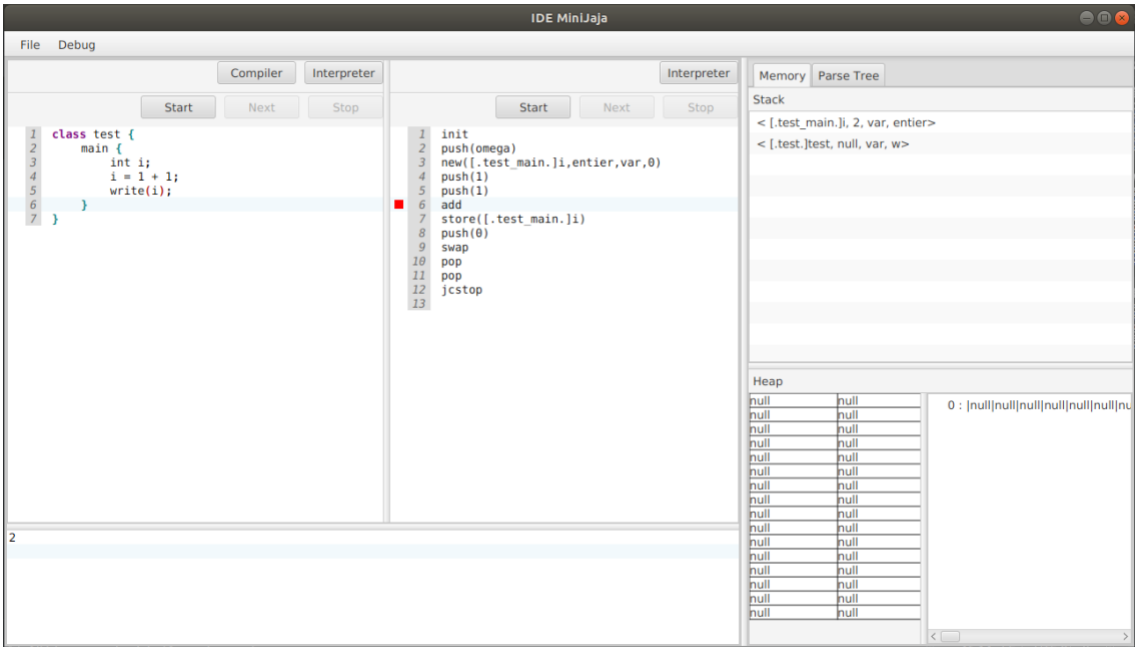


FIGURE 11 – Capture d'écran de l'interface

4 Assurance qualité

Tout au long du projet, des tests ont été réalisés par les membres de l'équipe en parallèle du développement. Ils nous ont permis de détecter rapidement les erreurs pour les corriger. Ces tests sont réalisés à plusieurs niveaux :

— Tests unitaires/composants permettant de vérifier le bon fonctionnement d'une partie précise d'une portion d'un programme ;
— Tests d'acceptation visant à assurer formellement que le produit est conforme aux spécifications à l'aide de fichiers en *MiniJaja* (.mjj) ;
— Tests d'intégration permettant de s'assurer que chaque module communique correctement ensemble.

Notre objectif de couverture avec les tests unitaires en début de projet est de **80%** en prenant en compte les fonctionnalités non testables avec l'outil *JUnit*.

Les résultats présentés ci-dessous ne correspondent pas aux résultats définitifs puisqu'ils évolueront pendant la dernière semaine de sprint pour atteindre l'objectif fixé et s'assurer du bon fonctionnement du logiciel.

Dans cette section, nous allons décrire les différents tests réalisés à plusieurs niveaux, leurs résultats ainsi que la qualité du code.

4.1 Tests unitaires

Les tests unitaires ont été réalisés dans les modules suivants :
— AST
— Memory
— Interpreter

Voici les résultats obtenus pour chacun des modules à la révision 217 :

Module	Coverage	Uncovered Lines	Uncovered Cond.
ALL	78.3	257	167
AST	73.2	115	68
MEMORY	79.2	92	36
INTERPRETER	82.7	50	63

L'ensemble de ces statistiques a été obtenu à partir de 185 tests unitaires.

Nous estimons que les tests réalisés présentent des résultats acceptables, mais sont tout de même à renforcer par la suite pour s'assurer de la bonne qualité du projet.

Les modules *IHM* et *Lexer_Parser* n'ont pas de tests unitaires puisque nous ne disposions pas des ressources nécessaires pour pouvoir le faire étant donné qu'il

n'existe pas d'outils pour tester les fichiers générés par *JavaFx* et *Antlr4*.

4.2 Tests d'acceptation

Pour réaliser les tests d'acceptation, nous utilisons une bibliothèque de plus d'une trentaine de fichiers *MiniJaja*. Ces fichiers nous permettent de s'assurer que les différents modules communiquent correctement ensemble et que le fonctionnement suit bien le processus décrit pas le schéma 1.

La figure 12 présente un exemple de test d'acceptation utilisant plusieurs fonctionnalités comme les fonctions, boucles, tableaux, opérations.

```
class c {
    int x = 0;
    int t[4];
    int fct(int max){
        int y = 5;
        while(max > 0){
            y+= max;
            max = max -1;
        };
        return y;
    };
    main{
        while(4>x){
            t[x] = fct(1);
            x++;
        };
    }
}
```

FIGURE 12 – Exemple de test d'acceptation

4.3 Tests d'intégration

Les tests d'intégration ont été réalisés au travers de tests via l'interface de notre logiciel permettant de montrer que les différents modules communiquent correctement entre eux, comme pour la fonctionnalité du mode pas à pas.

4.4 Résultats sur la batterie de tests fournie

Une batterie de 5 tests d'acceptation en fichier *MiniJaja* nous a été fournie.

Actuellement, sur ces 5 fichiers, nous obtenons les résultats suivants :
— Passable(s) : 4
— En échec(s) : 1

4.5 Qualité du code

L'outil *SonarQube* permet d'obtenir des statistiques générales du projet et concernant le code. La figure 13 présente ces résultats à la révision 217. Nous pouvons observer que ces résultats sont satisfaisants avec une qualité du code passable, aucun bug et aucune vulnérabilité.

Nous pouvons également voir le nombre de points d'amélioration concernant des standards de programmation qui est de 177, la couverture du projet avec les tests unitaires qui est de 78.3%, ainsi que le taux de duplication de code qui est de 0.6%. Le nombre de ligne de code s'élève à 2600 environ.

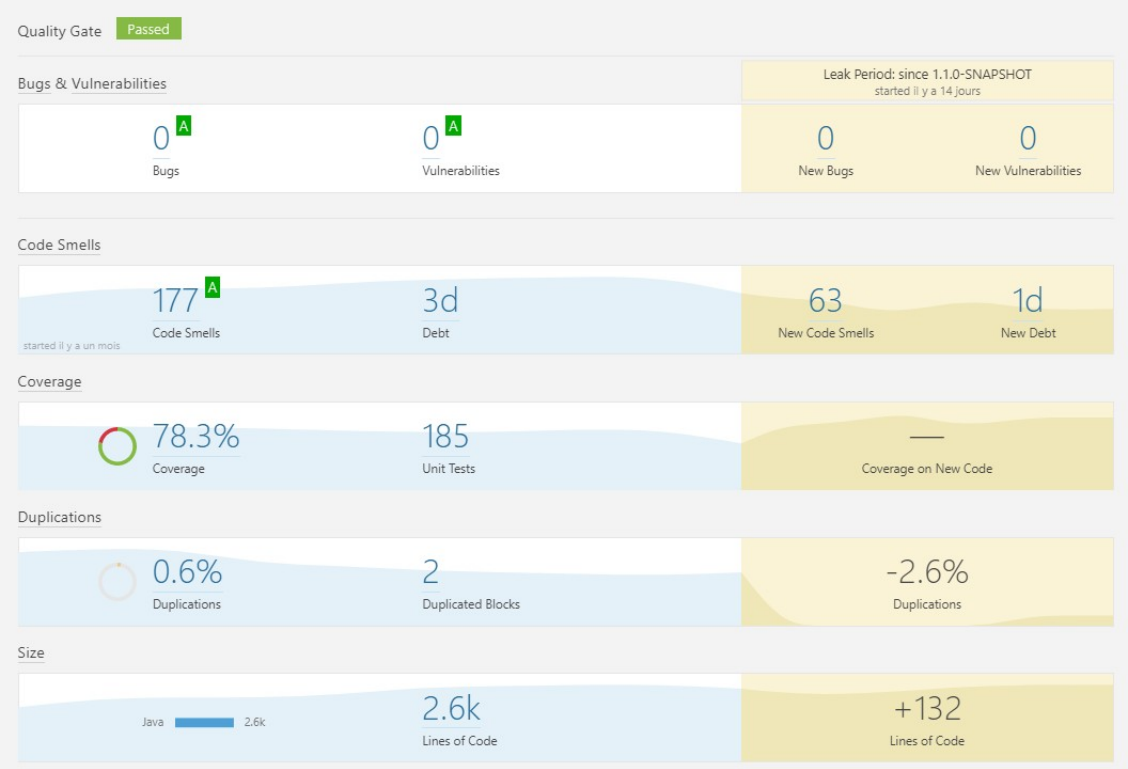


FIGURE 13 – Qualité globale du code

5 Résultats obtenus

Suite aux différentes phases de développement qui ont été réalisées pour concevoir l'application demandée, nous allons à présent détailler les résultats obtenus en énumérant les fonctionnalités implémentées et non implémentées au moment de l'écriture de ce présent rapport.

La plupart des fonctionnalités demandées par le client sont à ce jour présentes et fonctionnelles. L'utilisateur peut importer et sauvegarder depuis l'interface des fichiers *MiniJaja*. De plus, la grammaire décrite en annexe A est entièrement couverte. Il est donc possible de compiler des programmes *MiniJaja* en *JajaCode*. L'interprétation du langage *MiniJaja* n'est pas entièrement terminée, nous devons achever l'interprétation des méthodes qui sera développée pendant la dernière semaine du projet. L'interprétation du *JajaCode* quand à elle est complète. Le mode pas à pas avec l'utilisation des points d'arrêt est terminé et fonctionnel pour le langage de bas niveau *JajaCode*. En revanche, nous devons couvrir cette exigence lors de la dernière semaine de sprint pour le langage *MiniJaja*.

L'état actuel de la mémoire pendant l'exécution et les erreurs soulevées lors de la compilation ou l'interprétation sont affichées dans la console de l'*IHM*.

Nous pensons également améliorer l'agencement de l'interface utilisateur au cours de la dernière semaine pour que celle-ci soit plus confortable visuellement et permettre une expérience optimale à l'utilisateur et également revoir l'affichage de la partie mémoire avec le tas et la pile dans l'*IHM*.

6 Rétrospective de conduite du projet

À présent, nous allons réaliser une rétrospective sur notre conduite de projet depuis le lancement.

Suite à notre réalisation, nous en tirons des points positifs, notamment avec la méthode agile. Par conséquent, nous sommes très satisfait de notre organisation avec les compétences qui ont été vues en cours, qui nous ont permis :

- Une bonne cohésion de groupe ;
- Un renforcement de la communication au sein de l'équipe permettant de pallier aux difficultés rencontrées ;
- Une utilisation d'une application de gestion de projet tel que *Trello* nous permettant de mieux diviser le travail et de voir l'état actuel du projet en attribuant les tâches à plusieurs catégories (À faire, En cours, À tester/À valider, Fait ou À reporter). De plus, nous pouvions voir l'avancement des tâches avec un système de points accordés.
- Des Stand-up meetings réguliers ;
- Une inspection du code par les autres membres de l'équipe afin d'avoir un retour sur la qualité du code et corriger les erreurs potentielles ;
- Une bonne estimation des tâches à réaliser nous permettant de bien gérer notre temps et avoir peu de tâche à reporter ;
- Accomplir les objectifs fixés pour chaque sprint et release.

En ce qui concerne les difficultés rencontrées de cette conduite de projet, nous pouvons énumérer les problèmes suivants :

- Une mauvaise gestion du Sprint 5 dû aux activités universitaires annexes ;
- Le manque d'informations à la création du projet, notamment l'organisation de dépôt *Subversion* ;
- Quelques problèmes de compatibilité entre les différents appareils des membres de l'équipe.

Conclusion

D'un point de vue général, ce projet a été enrichissant pour l'ensemble du groupe autant en termes professionnels, grâce à la mise en place des pratiques agiles, qu'en termes techniques, par l'apprentissage des différents outils (*Antlr4*, Lexer-Parser, *Java*) pour créer un environnement de développement.

Aussi, nous avons pu mettre en avant autant nos connaissances acquises durant ce semestre de master (au sein des modules *Génie Logiciel* et *Compilation et Interprétations*), qu'autant durant tout le cours de notre licence (notamment au sein du module *Analyse Syntaxique* et de l'apprentissage continue du langage *Java*).

Différents problèmes ont pu être surmontés, particulièrement lors de l'intégration des outils de développement continu.

Au final, le projet s'est correctement déroulé, en respectant le principe de développement incrémental et itératif. Le projet est grandement abouti et fonctionnel. Les dernières fonctionnalités seront terminées durant la dernière semaine.

Annexe A - Fiche récapitulative du langage *Mini-Jaja*

Le langage MiniJaja

Nous définissons le langage MiniJaja sous la forme d'une grammaire non contextuelle de Backus-Naur avec les notations suivantes :

- les *symboles terminaux* sont soit des identificateurs soulignés (vide exprimant la chaîne vide), soit des caractères entre guillements tels que "{", "(", "+"
- les *symboles non terminaux* sont tous les identificateurs non soulignés
- les symboles →,[,] et | sont des symboles du méta-langage exprimant respectivement la dérivation, la présence facultative et le choix entre les règles.

classe →	class ident "{" decls methmain"}"	*classe($2,$4,$5)*
ident →	identificateur	*ident($1)*
decls →	decl ";" decls | vide	*decls($1,$3)* | *vnil*
decl →	var | methode	*$1* | *$1*
vars →	var ";" vars | vide	*vars($1,$3)* | *vnil*
var →	typemeth ident vexp |	*var($1, $2, $3)* |
	typemeth ident "[" exp "]" |	*tableau($1, $2, $4)* |
	final type ident vexp	*cst($2, $3, $4)*
vexp →	"=" exp | vide	*$2* | *omega*
methode →	typemeth ident "("entêtes")" "{" vars instrs "}"	*méthode($1, $2, $4, $7, $8)*
methmain →	main "{" vars instrs "}"	*main($3,$4)*
entêtes →	entête "," entêtes |	*entêtes($1,$3)* |
	entête | vide	*entêtes($1, enil)* | *enil*
entête →	type ident	*entête($1,$2)*
instrs →	instr ";" instrs | vide	*instrs($1, $3)* | *inil*
instr →	ident1 "=" exp |	*affectation($1,$3)* |
	ident1 "+=" exp |	*somme($1,$3)* |
	ident1"++" |	*incrément($1)* |
	ident"("listexp")" |	*appelI($1,$3)* |
	return exp |	*retour($2)* |
	write "(" (ident | "string") ")"	*ecrire($3)* | *ecrire(chaine($3))* |
	writeln "(" (ident | "string") ")"	*ecrireln($3)* | *ecrireln(chaine($3))* |
	if exp "{" instrs "}" [else "{" instrs "}"] |	*si($2, $4, $8)* |
	while "(" exp ")" "{" instrs "}"	*tantque($3, $6)*
listexp →	exp "," listexp |	*listexp($1,$3)* |
	exp | vide	*listexp($1,exnil)* | *exnil*
exp →	"!" exp1 |	*non($2)* |
	exp "&&" exp1 |	*et($1, $3)* |
	exp "||" exp1 | exp1	*ou($1, $3)* | *$1*
exp1 →	exp1 "==" exp2 |	*==($1,$3)* |
	exp1 ">" exp2 | exp2	*>($1,$3)* | *$1*
exp2 →	exp2 "+" terme |	*+($1,$3)* |
	exp2 "-" terme | "-" terme | terme	*-($1,$3)* | *moins($2)* | *$1*
terme →	terme "*" fact |	**($1,$3)* |
	terme "/" fact | fact	*/($1,$3)* | *$1*
fact →	ident1 |	*$1* |
	ident"("listexp")" |	*appelE($1,$3)* |
	true | false | nombre |	*vrai* | *faux* | *nbre($1)* |
	"("exp")"	*$2*
ident1 →	ident | ident"["exp"]"	*$1* | *tab($1,$3)*
typemeth →	void | type	*rien* | *$1*
type →	int | boolean	*entier* | *booléen*

Résumé

Ce rapport rend compte de la réalisation d'un projet de machine virtuelle utilisant les méthodes agiles *SCRUM* au sein d'une équipe de six développeurs, réalisé dans le cadre de la première année de Master en informatique à l'Université de Franche-Comté.

Ce projet implémente un environnement de développement permettant de reconnaître un langage de haut niveau appelé *MiniJaja*. Ce langage compilé permet d'obtenir un langage de bas niveau *JajaCode*. Lors de l'interprétation d'un de ces deux langages, l'environnement utilise différentes fonctions telles que la mémoire afin d'obtenir un résultat.

Mots clés

Master informatique, développement agile, Java, machine virtuelle, grammaire, compilation, mémoire, interpréteur

Abstract

This document reports on the realization of a virtual machine project using the *SCRUM* agile methods in a team of six developers, as part of the first year of Master in computer science of the University of Franche-Comté.

This project implements a development environment allowing to recognize a high level language called *MiniJaja*. This compiled language provides a low level *JajaCode* language. When interpreting one of these languages, the environment uses different functions such as memory to get a result.

Key-words

Master of computer science, agile development, Java, virtual machine, grammar, compilation, memory, interpreter