

Document de Validation

BOUTALEB Youssef

BOUHAR Mounsef

MALYAH Lina

EL KAZDAM Zakaria

EL ASLI Adnan

I. Descriptif des tests	2
1. Types de tests	2
2. organisation des tests	2
- Format des tests	2
- Hiérarchie	3
3. Objectifs des tests	4
II. Les scripts de tests	5
1. Commandes de lancement des scripts	5
- add-test	5
- execute	6
- complete-lex	6
- complete-synt	6
- complete-context	7
- complete-decac	7
- Autotest-mvn	8
- Exemple	9
III. Gestion des risques et gestion des rendus	9
IV. Résultats de Jacoco	10
V. Méthodes de validation utilisées autres que le test	11

I. Descriptif des tests

1. Types de tests

Pour assurer la robustesse et la fiabilité de notre compilateur Deca, nous avons mis en place une stratégie complète de tests, regroupés en deux principaux types : les tests unitaires simples (simpleSpec) et les tests intégration.

Tests Unitaires simples (simpleSpec): Au début de chaque étape de développement, nous avons conçu des tests unitaires simples, visant à évaluer le bon fonctionnement de chaque spécification isolée du compilateur. Ces tests se concentrent sur des fonctionnalités spécifiques telles que les structures conditionnelles (if), les boucles (while), et d'autres aspects essentiels du langage Deca. Les tests unitaires simples nous ont permis de vérifier rapidement l'implémentation de chaque spécification au fur et à mesure de son développement. De plus, cette méthode a considérablement facilité la détection et la correction rapide de bugs potentiels, garantissant ainsi la stabilité du compilateur tout au long de son développement.

Tests d'intégration conséquents : À mesure que nous avançons dans le développement du compilateur, nous avons introduit des tests d'intégration plus conséquents. Ces tests, élaborés à la fin de chaque sprint ou phase de développement majeure, englobent plusieurs spécifications du compilateur. L'objectif était de vérifier la cohérence et l'interaction des différentes fonctionnalités dans un contexte plus complexe. Les tests d'intégration ont permis de valider notre compilateur dans des scénarios réalistes, mettant en lumière la capacité du système à gérer des combinaisons variées de spécifications.

Cette approche progressive nous a permis d'assurer un contrôle qualité efficace à chaque étape du développement, garantissant une implémentation robuste et fonctionnelle du compilateur Deca.

2. organisation des tests

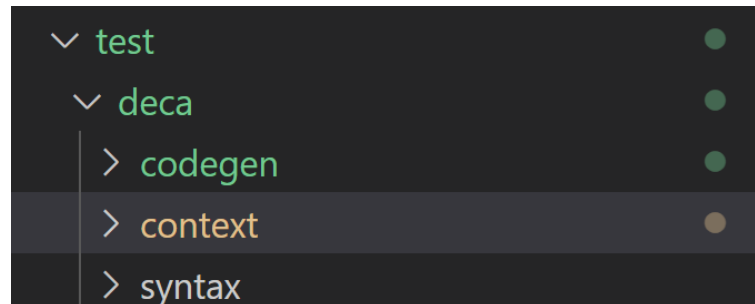
Nous avons mis en place une structure organisée pour nos tests, basée sur une hiérarchie claire qui facilite la gestion et la compréhension des différentes phases de validation du compilateur. Voici comment cette organisation se présente :

- Format des tests

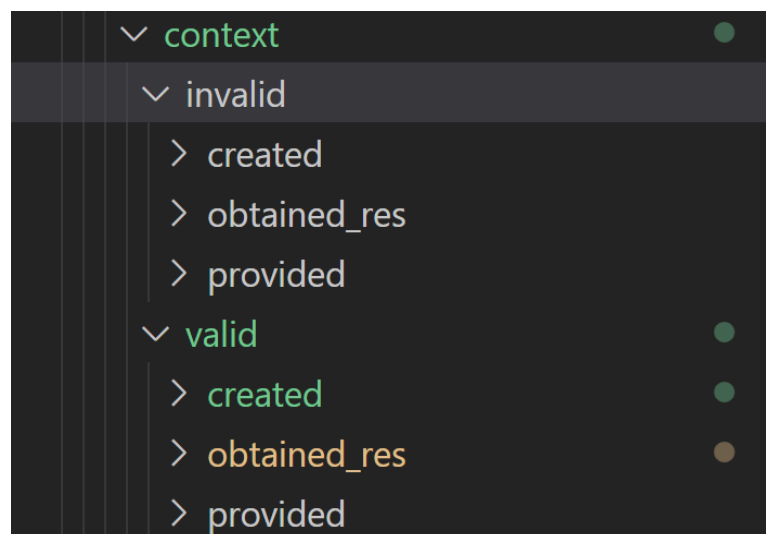
```
// Description :  
//   Test invalide : Assignment incorrecte à des variables de différents types  
//  
// Résultats :  
//   Erreur contextuelle  
//   Ligne 15: Incompatibilité de type lors de l'assignation. Type attendu : int, Type obtenu : Float
```

Chaque test est accompagné d'un en-tête en commentaire décrivant le test et spécifiant le résultat attendu pour un test invalide : un message d'erreur à la ligne 15 avec le libellé "Incomp..." ou un "OK" en cas de test valide.

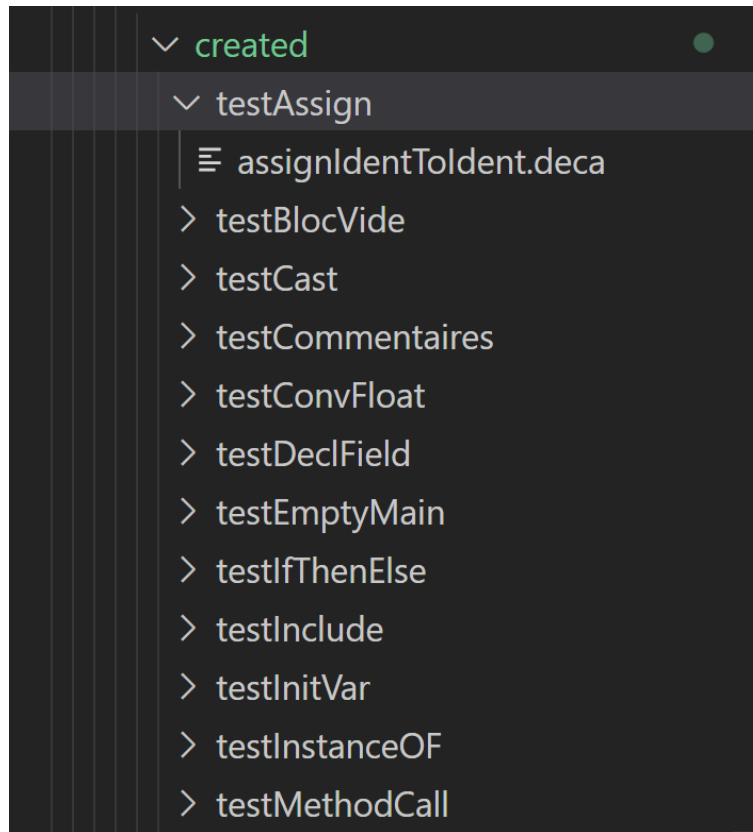
- Hiérarchie



La structure des tests pour le compilateur Deca est initialement divisée en trois branches principales, correspondant respectivement aux étapes A (syntaxe), B (contexte), et C (génération de code).



Chaque spécification du compilateur a son propre sous-dossier dans les répertoires "Valide" et "Invalide". À l'intérieur de ces sous-dossiers, les tests sont séparés en "created" pour les tests que nous avons conçus, "provided" pour les tests fournis, et "obtained_res" pour les résultats respectifs.



De plus, nous avons regroupé les tests dans des sous-répertoires spécifiques, réunissant les tests traitant des mêmes fonctionnalités. Par exemple, un sous-répertoire tel que "testConvFloat" regroupe tous les tests liés à la conversion des nombres flottants. Cette organisation facilitait la navigation et la gestion des tests, en les regroupant de manière logique et cohérente.

3. Objectifs des tests

Les objectifs de nos tests étaient multiples. D'abord, nous avons cherché à valider chaque spécification individuellement, garantissant le bon fonctionnement de chaque composant du compilateur. En parallèle, nous avons évalué le compilateur dans son ensemble en combinant plusieurs spécifications pour tester son intégration et sa capacité à traiter des scénarios complexes.

Notre objectif a été pleinement atteint grâce aux tests, qui ont été des outils précieux pour comprendre en profondeur notre implémentation. Ces tests ont mis en lumière des aspects critiques de notre compilateur, nous permettant d'identifier ses forces ainsi que ses limitations. En détectant des bugs que nous aurions pu ne pas remarquer autrement, ils ont joué un rôle essentiel dans l'amélioration continue de notre système, facilitant ainsi un processus de débogage efficace.

II. Les scripts de tests

Les scripts de tests ont joué un rôle essentiel dans la phase de développement en fournissant une approche méthodique et automatisée pour évaluer la robustesse de notre compilateur. Ces outils ont grandement contribué à optimiser notre temps, à encourager une culture de codage axée sur la qualité, et à faciliter l'intégration continue des nouvelles fonctionnalités. En permettant une vérification automatisée à chaque ajout de code, les scripts de tests ont joué un rôle central dans le maintien de la stabilité du compilateur, garantissant que les nouvelles implémentations n'altèrent pas les fonctionnalités existantes. Ils ont ainsi été un pilier essentiel pour assurer la cohérence et la fiabilité de notre projet à chaque étape de son développement, tout en encourageant activement l'équipe à coder des tests exhaustifs pour garantir la pérennité du compilateur.

1. Commandes de lancement des scripts

- **add-test**

Le script `add-test.sh` est une commande shell qui automatise le processus d'ajout de nouveaux tests dans le cadre du développement du compilateur Deca. Voici un résumé des fonctionnalités de ce script :

1. **Options de Test** : Le script prend en charge plusieurs options pour spécifier les types de tests à effectuer, notamment les tests lexicaux (`-l`), syntaxiques (`-s`), contextuels (`-c`), Decac (`-d`), et ima (`-i`). L'utilisateur peut sélectionner plusieurs options simultanément.
2. **Édition avec VSCode** : Le script ouvre le fichier de test correspondant dans l'éditeur de code VSCode pour permettre à l'utilisateur d'ajouter ou de modifier du code.
3. **Copie dans les Répertoires Appropriés** : Le script copie le fichier de test dans les répertoires correspondants en fonction des options sélectionnées. Il prend en charge les répertoires de tests syntaxiques, contextuels, et de génération de code.
4. **Exécution des commandes** : Selon les options choisies, le script exécute les commandes de test lexicales, syntaxiques, contextuelles, Decac, et ima. Il affiche les commandes exécutées pour une meilleure traçabilité.
5. **Suppression des Fichiers Existant** : Avant d'exécuter Decac ou ima, le script vérifie l'existence des fichiers `.ass` et les supprime si nécessaire pour éviter tout conflit.

En résumé, ce script simplifie et automatise le processus d'ajout, d'édition, et de test de nouveaux scénarios dans le cadre du développement du compilateur Deca. Il contribue à accélérer le cycle de développement, à garantir la qualité du code, et à faciliter l'intégration continue du projet.

- **execute**

Le script `execute.sh` est conçu pour simplifier le processus d'exécution du compilateur Deca sur un fichier source en langage Deca. Voici une description détaillée des fonctionnalités de ce script :

1. **Vérification du Fichier Source** : Le script vérifie si l'utilisateur a fourni un nom de fichier source `.deca` en tant qu'argument. S'il manque l'argument, le script affiche un message d'utilisation indiquant comment utiliser le script et quitte avec un code d'erreur (1).
2. **Nom du Fichier Assembleur** : Le script utilise le nom du fichier source fourni pour générer le nom du fichier assembleur correspondant en remplaçant l'extension `.deca` par `.ass`.
3. **Compilation avec Decac** : Le script utilise la commande `decac` pour compiler le fichier source `.deca` en code assembleur. Si la compilation réussit, un fichier assembleur (`.ass`) est généré dans le même répertoire que le fichier source.
4. **Exécution avec Ima** : Le script vérifie si le fichier assembleur a été généré avec succès. Si tel est le cas, il utilise la commande `ima` pour exécuter le fichier assembleur. En cas de réussite, le programme est exécuté. Si le fichier assembleur n'a pas été généré, le script affiche un message d'erreur indiquant qu'il y a eu un problème lors de la compilation avec `decac`.

En résumé, le script `execute.sh` automatise les étapes de compilation et d'exécution du compilateur Deca sur un fichier source, fournissant ainsi un moyen simple d'exécuter des programmes Deca.

- **complete-lex**

Le script `complete-lex.sh` a pour objectif de réaliser les tests lexicaux sur l'ensemble des fichiers Deca valides présents dans le répertoire dédié aux tests lexicaux. Son rôle principal est d'évaluer la conformité lexicale du compilateur Deca. Voici une explication détaillée :

1. **Évaluation des résultats** : Pour chaque fichier, le script extrait un commentaire au début du fichier `.deca` contenant les résultats lexicaux attendus. Il compare ensuite ces résultats avec ceux obtenus par l'exécution des tests lexicaux.
2. **Affichage des Résultats** : Le script affiche clairement le résultat de chaque test en indiquant s'il a réussi ([OK]) ou échoué ([FAILED]). À la fin de l'exécution, un récapitulatif des résultats est présenté, comprenant le nombre total de tests réussis et échoués.

- **complete-synt**

Le script `complete-synt.sh` vise à réaliser les tests syntaxiques sur l'ensemble des fichiers Deca valides situés dans le répertoire dédié aux tests syntaxiques. Son rôle principal est

d'évaluer la conformité syntaxique du compilateur Deca. Voici une explication détaillée de son fonctionnement :

Évaluation des Résultats : Le script parcourt chaque fichier et exécute le test syntaxique en utilisant le script `test_synt`. Pour chaque fichier valide, il compare les résultats obtenus avec ceux attendus, en recherchant les indications de ligne d'erreur dans le commentaire situé au début du fichier `.deca`.

Affichage des Résultats : Pour chaque test, le script affiche clairement le résultat, indiquant s'il a réussi ([OK]) ou échoué ([FAILED]). À la fin de l'exécution, un récapitulatif des résultats est présenté, comprenant le nombre total de tests réussis et échoués.

Tests Invalides : Le script réalise également des tests syntaxiques sur les fichiers invalides. Pour chaque fichier invalide, il vérifie si le message d'erreur généré correspond à celui spécifié dans le commentaire du fichier. Cela signifie que tant la ligne d'erreur que le message d'erreur attendu doivent être identiques.

- `complete-context`

Le script `complete-context.sh` a pour objectif de réaliser les tests contextuels sur l'ensemble des fichiers Deca valides et invalides, présents dans les répertoires dédiés aux tests contextuels. Son rôle principal est d'évaluer la conformité contextuelle du compilateur Deca. Voici une explication détaillée :

Évaluation des résultats : Pour chaque fichier, le script extrait un commentaire au début du fichier `.deca` contenant les résultats contextuels attendus. Il compare ensuite ces résultats avec ceux obtenus par l'exécution des tests contextuels.

Affichage des Résultats : Le script affiche clairement le résultat de chaque test en indiquant s'il a réussi ([OK]) ou échoué ([FAILED]). À la fin de l'exécution, un récapitulatif des résultats est présenté, comprenant le nombre total de tests réussis et échoués pour les fichiers valides et invalides respectivement.

La comparaison des résultats contextuels est effectuée en vérifiant la présence d'erreurs, de numéros de ligne et de messages d'erreur. Les fichiers valides doivent être exempts d'erreurs, tandis que les fichiers invalides doivent générer des erreurs contextuelles conformes aux attentes définies dans les commentaires des fichiers de test.

- `complete-decac`

Le script `complete-deca.sh` a pour objectif de réaliser les tests Deca sur l'ensemble des fichiers Deca valides et invalides, présents dans les répertoires dédiés aux tests de génération de code. Son rôle principal est d'évaluer la conformité de la génération de code du compilateur Deca. Voici une explication détaillée :

Tests Valides :

1. Suppression des fichiers .ass existants : Le script commence par supprimer tous les fichiers .ass déjà présents dans le répertoire et ses sous-répertoires.
2. Compilation avec decac -P : Le script exécute la commande decac -P sur tous les fichiers .deca dans le répertoire et ses sous-répertoires. Dans le but d'évaluer l'efficacité de l'option -P et d'optimiser le temps d'exécution en utilisant une exécution parallèle.
3. Comparaison avec les résultats attendus : Le script compare ensuite les résultats obtenus avec les résultats attendus. Chaque fichier .ass est comparé avec son équivalent .expected pour évaluer la génération de code.

Affichage des Résultats : Le script affiche clairement le résultat de chaque test en indiquant s'il a réussi ([OK]) ou échoué ([FAILED]). À la fin de l'exécution, un récapitulatif des résultats est présenté, comprenant le nombre total de tests réussis et échoués pour les fichiers valides et invalides respectivement.

Tests Invalides :

1. Exécution de decac et ima : Le script exécute la commande decac sur les fichiers invalides pour générer les fichiers .ass, puis exécute ima sur ces fichiers .ass pour évaluer les erreurs générées.
2. Comparaison avec les messages d'erreur attendus : Le script compare les messages d'erreur générés avec les messages d'erreur attendus, permettant d'évaluer la gestion des erreurs lors de la génération de code.

Ce dernier permet également d'afficher le résultat comme ([MISSING EXPECTED]) si quelqu'un a implémenté un test sans spécifier le résultat attendu.

- Autotest-mvn

Dans le cadre de notre gestion proactive des risques, nous avons également introduit une commande essentielle, **autotest-mvn**, qui représente un alias de **mvn test -Dmaven.test.failure.ignore**. Cette commande, intégrée dans notre fichier **pom.xml**, offre un moyen efficace de lancer tous nos scripts précédents ainsi que les tests déjà fournis. Elle permet d'automatiser et de simplifier le processus de validation en ignorant les échecs de tests temporaires, facilitant ainsi l'exécution globale des tests sans interruption. Cette amélioration renforce notre approche méthodique en assurant une validation continue tout en facilitant l'intégration des tests dans notre processus de développement.

- Exemple

```
----- INVALIDES DECA TESTS -----  
[ OK ] dereferencementNullField.deca  
[ OK ] dereferencementNullVar.deca  
[ OK ] invalidcast.deca  
[ OK ] noReturn.deca  
[ OK ] Dereferencementnull.deca  
[ FAILED ] addOverFlow.deca  
[ FAILED ] divisionzero.deca  
/mnt/c/Users/AdMiN/Desktop/ENSIMAG/2A/Projet_GL/GL/./src/test/deca  
he identifier was not declared before use.  
cat: ./src/test/deca/codegen/invalid/obtained_res/instanceofNotCl  
[ FAILED ] instanceofNotClass.deca  
/mnt/c/Users/AdMiN/Desktop/ENSIMAG/2A/Projet_GL/GL/./src/test/deca  
identifier was not declared before use.  
cat: ./src/test/deca/codegen/invalid/obtained_res/instanceofObject  
[ FAILED ] instanceofObject.deca  
[ FAILED ] minusOverFlow.deca  
[ FAILED ] moduloZero.deca  
[ FAILED ] multiplyOverlFlow.deca  
[ OK ] noReturn.deca  
/mnt/c/Users/AdMiN/Desktop/ENSIMAG/2A/Projet_GL/GL/./src/test/deca  
The identifier was not declared before use.  
cat: ./src/test/deca/codegen/invalid/obtained_res/notAnObjectInst  
[ FAILED ] notAnObjectInstance.deca  
  
----- RESULTS : INVALIDE DECA TESTS -----  
  
PASSED TESTS : 6  
FAILED TESTS : 8  
RESULTS : 6/14  
  
----- END -----
```

```
[ MISSING EXPECTED ] Equals.deca : Aucun fichier .expected trouvé
```

III. Gestion des risques et gestion des rendus

La gestion des risques dans notre projet de développement d'un compilateur Deca demeure une composante essentielle pour garantir la qualité du produit final. Nous avons adopté des mesures concrètes pour anticiper et atténuer les risques identifiés, tout en mettant en place des procédures de gestion des rendus.

Dans le domaine de la programmation, notre approche proactive inclut l'automatisation des tests subtils. Un membre de l'équipe exécute banque de tests automatisée, représentative de divers cas de programmes Deca, après chaque pull d'un sprint. L'intégration de la commande

`autotest-mvn` dans notre processus de tests automatisés permet d'identifier et de résoudre rapidement les erreurs simples mais critiques avant chaque rendu.

Pour éviter tout oubli de dates des rendus, nous avons établi un calendrier partagé avec des rappels automatiques. Sur GitLab, des tâches sont créées pour chaque aspect du projet, avec des deadlines claires. Chaque membre peut prendre en charge une tâche et mettre à jour son suivi dans les commentaires.

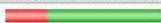












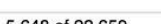
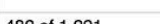
Afin de garantir la crédibilité du compilateur, une attention particulière est accordée à la validation des fonctionnalités cruciales, avec l'organisation de sessions régulières de revue de code. Cela assure la qualité du code source et permet de résoudre rapidement tout problème potentiel.

Concernant la gestion des mises en production, nous avons instauré une checklist détaillée d'actions à effectuer avant chaque rendu. De plus, nous travaillons sur une branche de développement, et à la fin de chaque sprint, nous fusionnons avec la branche principale (master) pour conserver l'ancienne implémentation intacte en cas de nouveaux bugs. Lors de chaque fusion, nous vérifions la compilation et le succès des tests. Avant les rendus de production, nous clonons notre branche master sur un nouveau répertoire pour tester et éviter la présence d'éléments non résolus dans nos sessions de travail. Cette approche assure une intégration continue et une stabilité dans le code source du projet. En conclusion, nos actions concrètes démontrent notre engagement envers la gestion proactive des risques et la qualité du projet, en témoignant de notre approche méthodique et de notre volonté de fournir un compilateur Deca robuste et fiable. Ces pratiques sont continuellement discutées et ajustées lors des réunions de suivi pour garantir leur pertinence tout au long du projet.

IV. Résultats de Jacoco

Nous avons exploré l'utilisation de Jacoco dans notre processus de test, bien que son fonctionnement ait parfois été difficile à interpréter. Il arrivait que Jacoco signale une non-couverture de la classe `LOAD`, malgré son utilisation répandue dans la majorité des opérations et dans presque tous les tests. Malgré ces défis, Jacoco s'est avéré être un outil précieux, nous offrant des indications utiles et nous guidant vers des aspects spécifiques de notre implémentation nécessitant une attention particulière. Grâce à Jacoco, nous avons identifié des erreurs dans notre code et avons pu effectuer des ajustements pour améliorer la qualité de notre projet.

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		72 %		54 %	536	722	588	2 063	264	374	8	51
fr.ensimag.deca.tree		78 %		68 %	204	656	424	1 828	108	467	2	87
fr.ensimag.deca.context		73 %		67 %	42	166	61	276	21	126	0	21
fr.ensimag.deca		80 %		65 %	38	91	56	271	9	43	2	5
fr.ensimag.ima.pseudocode		74 %		75 %	29	84	47	180	24	74	2	26
fr.ensimag.deca.tools		36 %		25 %	12	19	29	42	9	15	1	3
fr.ensimag.ima.pseudocode.instructions		70 %	n/a	n/a	20	62	36	111	20	62	16	54
fr.ensimag.deca.codegen		67 %		50 %	5	13	8	26	4	12	0	1
Total	5 648 of 22 650	75 %	482 of 1 221	60 %	886	1 813	1 249	4 797	459	1 173	31	248

V. Méthodes de validation utilisées autres que le test

Dans le cadre de notre méthodologie de gestion d'équipe, nous avons adopté une approche structurée visant à maximiser l'efficacité et la qualité de notre travail sur le projet. Pour chaque spécification à implémenter, nous avons formé des groupes distincts, chacun responsable d'une phase spécifique du processus de développement.

Tout d'abord, un groupe était chargé de la conception, s'assurant que les fonctionnalités étaient correctement conçues et alignées sur les exigences spécifiées. Un autre groupe était dédié au test, garantissant la robustesse et la fiabilité de chaque composant du système. Enfin, un troisième groupe se concentrait sur la validation, examinant attentivement chaque partie du code pour identifier d'éventuelles erreurs ou incohérences.

Cette approche a favorisé une collaboration étroite entre les membres de l'équipe, permettant une compréhension approfondie de chaque aspect du projet. De plus, elle a permis une visibilité accrue sur l'ensemble du projet, avec chaque groupe pouvant examiner et commenter le travail des autres. Les retours constructifs et les échanges d'idées ont été encouragés, contribuant ainsi à l'amélioration continue du projet.

En attribuant des groupes spécifiques à des tâches spécialisées, nous avons pu détecter précocement les problèmes potentiels et assurer une répartition équilibrée des charges de travail. De plus, cette approche a facilité les échanges de connaissances, les membres des équipes partageant leur expertise pour renforcer les compétences globales de l'équipe. Cela a été réalisé tout en observant scrupuleusement les principes et les aspects de la méthodologie Scrum.

En résumé, notre méthodologie de gestion d'équipe a démontré son efficacité en favorisant la collaboration, la détection précoce des problèmes et le partage des connaissances au sein de l'équipe de développement. Ces pratiques ont contribué de manière significative à la qualité et à la réussite de notre projet.