

Document de validation

Groupe 17

GUILLAUME Laure

LUU Duc Anh

EL RHATRIF Mohammed Amine

CHAKIR Sami

FARID Othmane

27 janvier 2017

1 Descriptif des tests

Nous avons tout d'abord fait des tests sans objet. Nous avons décidé que les tests de la partie objets seraient fait par la suite. Nous avons aussi bien séparé les tests des trois parties dans les dossiers attitrés et nous avons fait le choix de ne pas copier des tests d'une partie dans une autre. Ainsi les tests utilisés dans chaque partie sont différents. Nous avons fait le choix que les tests de la partie B exécutent aussi la partie A et de même pour la partie C où les tests exécutent aussi les deux parties précédentes. En procédant ainsi nous espérons couvrir un maximum de code avec nos tests et être capable de détecter des erreurs supplémentaires que les tests des parties précédentes auraient laissé passer.

1.1 Tests de la partie sans objet

Au début nous avons préféré de faire principalement des tests unitaires. Ces tests s'appliquent sur une partie précise de notre code, ce qui simplifie grandement la correction d'erreur. En effet nous avons implémenté les tests en parallèle avec les implémentations. Ainsi à chaque partie implémentée correspond un certain nombre de tests unitaires qui permettent à celui qui implémente cette partie de vérifier son bon fonctionnement. Nous avons créé de nombreux tests unitaires pour chaque partie du code afin de s'assurer que toutes les configurations possibles étaient prises en compte. Par exemple, nous avons vérifié que l'opérateur "-" fonctionnait avec deux registres, un registre à droite et Dval à gauche et inversement et ainsi de suite. Nous avons donc créé une multitude de tests unitaires.

Par exemple, le test unitaire du *if* ressemble à ceci :

```
{
  int x=3;
  if (x >= 0) {
    print("x est positif");
  }
}
```

Nous avons aussi effectué des tests plus complexes. Une fois les tests unitaires validés, on peut alors tester les fonctions réalisées avec des tests exécutant une plus grande partie de code. Par exemple, une fois les tests unitaires du *if* effectués nous avons fait des tests plus complexes avec des *or*, des booléens et encore des *ou*.

1.2 Tests de la partie avec objet

Nous avons fait les tests de la partie avec objet une fois que ceux de la partie sans objet fonctionnaient. Nous avons procédé de manière similaire que dans la partie sans objet. On a donc commencé par des tests simples et unitaires. Ainsi nous avons tout d'abord fait une classe vide, puis une fois cela testé une classe avec des champs, des méthodes vides,... Ainsi nous avons fait des tests de moins en moins simples et impliquant de plus en plus de code. Pour au final faire des tests plus globaux sur les classes avec des classes possédant des champs et des méthodes avec des champs, des héritages, ...

En fonctionnant ainsi nous avons pu implémenter et faire fonctionner des fichiers avec des classes simples quand d'autres parties du code n'étaient pas encore fonctionnelles, ce qui a permis de déboguer notre code petit à petit. Puis vérifier que le tout marchait bien.

1.3 Liste des erreurs contextuelles et les tests

Pour proposer des tests ciblés pour la partie B, on a essayé tout d'abord de dégager toutes les erreurs contextuelles possibles qu'on peut avoir pour chaque règle de notre grammaire attribuée, puis on a utilisé cette liste pour à la fois coder notre analyseur contextuel et pour aussi faire des tests dessus, pour vérifier que notre compilateur se comportait comme voulu et que les bons messages d'erreur soit levés.

Liste des erreurs possible pour chaque règle :

Règles	Description	Messages d'erreurs
Passe 1
1.3 (decl_class -> ...)	class_super de la classe reconnue n'est pas dans Env_type Super n'est pas associée à une définition de class	erreur Contextuelle:la class mère est non encore déclaré
1.3 (decl_class -> ...)	deux classes ont le même nom	erreur Contextuelle:cette class a déjà été déclarée
2.5 (decl_field -> ...)	un champs de la classe qui existe dans les super classes.	variable is already a field or a method in a <u>super class</u>
2.5 (decl_field -> ...)	le champs est de type void	in a field type shouldn't be void
2.7 (decl_metho		this method does not have the same signature as the

FIGURE 1 – un fragment de la liste d'erreur utilisée pour construire les tests

1.4 Conclusion

L'objectif de ces tests était surtout de vérifier la bonne implémentation de notre code en essayant de confronter notre code à tous les cas possibles. Il faut qu'à l'issue de chaque test notre programme renvoie un résultat conforme aux spécifications du client et en cas contraire qu'ils nous permettent d'identifier la source de l'erreur rapidement et de la corriger.

2 Scripts des tests

La création de scripts devient nécessaire avec un projet de cette taille. En effet, beaucoup de tests sont nécessaires pour vérifier toutes les implémentations mises en place et il est important de pouvoir les relancer régulièrement après une modification de code, en particulier avant de push une nouvelle version sur git. Les scripts permettent de lancer en une commande, un ensemble de tests de façon rapide et d'en vérifier leur bon fonctionnement.

2.1 Commandes de lancement des scripts

Nous avons mis en place des scripts qui permettent de lancer les tests. On peut lancer tous les tests avec la commande :

```
./src/test/script/launch-all-test.sh
```

On peut aussi lancer les tests de chaque partie séparément en utilisant les commandes suivantes pour chaque partie :

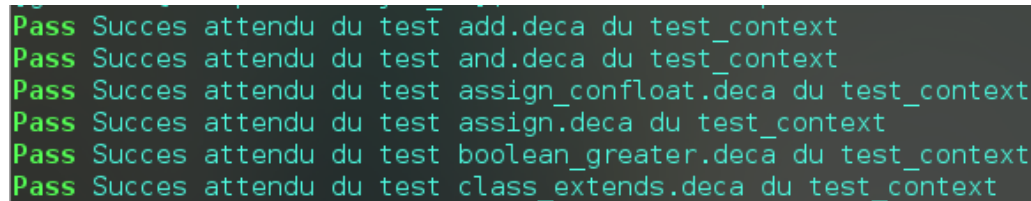
- Partie syntaxique : `./src/test/script/test-syntax-valid.sh` ou `./src/test/script/test-syntax-invalid.sh` en fonction de si on veut lancer les tests valides ou invalides.
- Partie contextuelle : de la même façon on peut lancer `./src/test/script/test-context-valid.sh` ou `./src/test/script/test-context-invalid.sh`.

- Partie génération du code assembleur : `./src/test/script/test-gencode-valid.sh` ou `./src/test/script/gencode-invalid.sh`

On peut aussi lancer la décompilation du code via les scripts avec la commande `./src/test/script/test-decompile.sh` ou effacer les fichiers `.ass` et `.res` produits par les commandes précédentes avec la commande `./src/test/script/removeResult.sh`. Vous pourrez retrouver l'ensemble de ces commandes de lancement des scripts dans le README.txt contenu dans le projet.

2.2 Fonctionnement des scripts

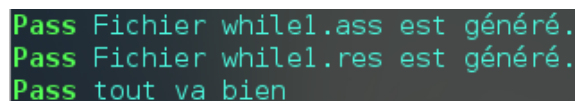
Les scripts qui lancent les tests des parties syntaxique et contextuelle, vérifient le contenu du fichier `.res` généré lors de leur exécution. Ils sont capables de savoir si un arbre a bien été généré ou au contraire, si le test a rendu une erreur. Cependant ils ne sont pas capables de vérifier le contenu de l'arbre. Il est donc nécessaire de vérifier manuellement le contenu du fichier `.res` une fois le test validé. En effet un test validé peut produire un arbre invalide. Si les tests passent avec succès on obtient dans le terminal :



```
Pass Succes attendu du test add.deca du test_context
Pass Succes attendu du test and.deca du test_context
Pass Succes attendu du test assign_confloat.deca du test_context
Pass Succes attendu du test assign.deca du test_context
Pass Succes attendu du test boolean_greater.deca du test_context
Pass Succes attendu du test class_extends.deca du test_context
```

FIGURE 2 – Résultat du lancement du script des parties contexte et syntaxe

Le script qui lance les tests de la génération du code vérifie trois choses : que le fichier `.ass` est bien généré, que le fichier `.res` qui contient le résultat du test est bien généré et que le résultat obtenu est bien le même que le résultat attendu. On obtient donc si tout va bien ceci dans le terminal :



```
Pass Fichier while1.ass est généré.
Pass Fichier while1.res est généré.
Pass tout va bien
```

FIGURE 3 – Résultat du lancement du script de la partie génération du code

3 Gestion des risques et gestion des rendus

3.1 Gestion des risques

Il est important dans un projet de prévoir les problèmes auxquels on risque de faire face et de proposer des solutions pour éviter qu'il y ait de graves répercussions sur le reste du projet. On peut diviser ces risques en trois grands types : les problèmes dus au fonctionnement de l'équipe, ceux concernant le développement du programme et au développement des tests. Il est important de connaître la gravité des risques pour savoir à quel point il est important de réagir vite et si il est important de passer beaucoup de temps à prévoir ces problèmes. En effet, si on perd plus de temps à les prévenir que le temps qu'ils nous ferait perdre cela ne vaut pas le coût. Il faut aussi connaître les impacts de ces risques pour savoir quelles seront les

conséquences et si cela est important.

A chaque problème le groupe doit définir une action permettant de régler le problème en fonction de sa gravité.

3.1.1 Fonctionnement de l'équipe

Oubli ou retard pour une date de rendu

Gravité : Importante

Impact : Non rendu d'un produit à temps qui va décrédibiliser l'équipe et implique des pénalités de retard.

Action : Vérifier régulièrement sur le planning où figure les dates de rendu et de suivi.

En cas de maladie ou d'absence d'un membre de l'équipe

Gravité : moyenne

Impact : Cela implique un retard sur le planning prévu et risque si il n'y a pas beaucoup de marge de provoquer un retard pour le rendu.

Action : Prévenir à l'avance en cas d'absence pour permettre une revue du planning et l'adaptation de l'équipe et la redistribution des tâches.

Mauvaise répartition du travail

Gravité : importante

Impact : Membres de l'équipe qui peuvent se retrouver à ne pas savoir quoi faire et donc à perdre du temps ce qui peut impliquer une prise de retard sur le planning

Action : Régulièrement faire la point sur ce qu'il y a à faire et revoir le planning pour répartir les tâches.

Si un membre bloque sur une partie

Gravité : Importante

Impact : La partie n'avance pas et le projet prend du retard et risque de ne pas être fini à temps.

Action : Signaler quand on prend du retard et que l'on bloque et demander de l'aide aux autres membres.

En cas de conflit

Gravité : moyenne

Impact : Risque de ralentir le travail en équipe et donc de retarder le planning et dans le pire des cas provoquer un retard dans les rendus.

Action : En discuter systématiquement avec le chef de projet ou en réunion pour que le conflit soit réglé au plus vite.

Différence entre l'attente du client et le produit

Gravité : Importante

Impact : Un produit qui ne correspond pas aux attentes provoque une mauvaise satisfaction du client.

Action : Parler régulièrement au client et ne pas hésiter à poser des questions lors des suivis.

3.1.2 Risques logiciels

Mauvaise compréhension de la grammaire

Gravité : Importante

Impact : Le compilateur ne se comporte pas comme prévu et le produit ne correspond alors pas aux attentes du client.

Action : Relecture du code et tests effectués par une personne autre que celle qui a fait le code.

Non respect des spécifications

Gravité : Importante

Impact : Le projet n'est pas conforme aux demandes du client.

Action : Relire le document fourni par le client avant de commencer une partie et vérifier à chaque fin d'étape que le produit suit bien les spécifications du client.

Problème de redondance

Gravité : Importante

Impact : Plusieurs membres du groupe font la même partie du code ce qui fait perdre du temps à l'équipe et il y a perte de travail.

Action : Chaque jour il faut faire le point sur ce que chacun fait pour que tous aient quelque chose à faire de différent.

Mauvaise factorisation du code

Gravité : Faible

Impact : Code pas optimisé.

Action : À la fin d'une partie revoir la structure du code pour vérifier si on peut le factoriser.

3.1.3 Risque liés aux tests

Mauvaise couverture des tests

Gravité : Importante

Impact : On passe à côté d'erreurs et de problèmes dans le code.

Action : Utiliser Cobertura pour voir la portion du code parcourue avec les tests effectués et voir si on a englobé tous les cas possibles.

Mauvaise écriture des tests

Gravité : Importante

Impact : Le test risque de ne pas relever une erreur ou inversement relever une erreur alors que le code est correct.

Action : Relecture du test par un autre membre que celui qui l'a rédigé.

Mauvaise écriture des scripts

Gravité : Importante

Impact : Une mauvaise écriture de scripts peut avoir la même conséquence qu'une mauvaise écriture des tests décrite ci-dessus.

Action : Vérifier si les tests fonctionnent bien individuellement une première fois. Si le script affiche qu'un test ne marche pas lancer le test à part.

Correction du code qui se répercute

Gravité : Moyenne

Impact : Tests précédemment validés deviennent invalides ce qui provoque une perte de temps car il faut revoir ces tests.

Action : Lancer l'ensemble de tests à chaque fois que l'on modifie une partie du code.

3.2 Gestion des rendus

Etant donné le volume de travail à fournir pour ce projet et la diversité des tâches, la gestion des rendus s'avère être indispensable. En effet elle permet aux membres du groupe de bien s'organiser, de se répartir les missions.

Nous avons tout d'abord mis en place des rendus presque quotidiens pour que chaque membre rende compte de son travail quotidien, pour suivre notre évolution et voir si celle-ci concordait avec le planning mis en place en début de projet. Chacun disait alors ce qu'il avait fait le jour même et ce qu'il prévoyait de faire les jours suivants. Ainsi en fonction de notre évolution, nous savions si un changement de rythme s'imposait, ou si tout se déroulait comme prévu et que le projet serait fini à temps. Cela nous permettait aussi de vérifier la répartition du travail et de parler des problèmes rencontrés.

Après chaque grande modification de portion de code, nous devions toujours faire la commande *mvn test*, et vérifier régulièrement ceci étant l'outil avec lequel on a automatisé notre base de tests. Le but de cette commande est de savoir si ces modifications n'ont pas influencé sur les tâches antérieures. Et si par malchance on trouve un souci de compilation, soit on trouve le problème le plus rapidement possible, soit on revient à la dernière version valide.

Après avoir vérifié que notre base de tests fonctionnait bien, l'utilisateur ayant réalisé la dernière modification met à jour le dépôt Git. Pour cela, avant de faire la commande *git push*, il fait un *git pull* pour vérifier que sa version est bien mise à jour. Une fois la modification effectuée on vérifie sur une autre machine que le projet fonctionne bien. Cela nous permet de voir si l'on a pas oublié de faire un *git add*.

On a appris aussi, à part les commandes basiques de git comme le push, pull et commit, d'autres options avancées du git comme créer des branches locales et de sauter entre celles-ci et la branche master et vis-versa, avec la commande *git branch*. De plus, on a appris une bonne habitude pendant ce projet de faire des *git status* pour comparer les changements effectués après le dernier enregistrement (*git commit*), et aussi de passer sur l'historique des push faites par les membres de l'équipe. Le checkout nous a été d'autant plus important où on l'utilise pour effacer des modifications ne servant plus à rien ou dans le cas où on abandonne l'idée pour une autre.

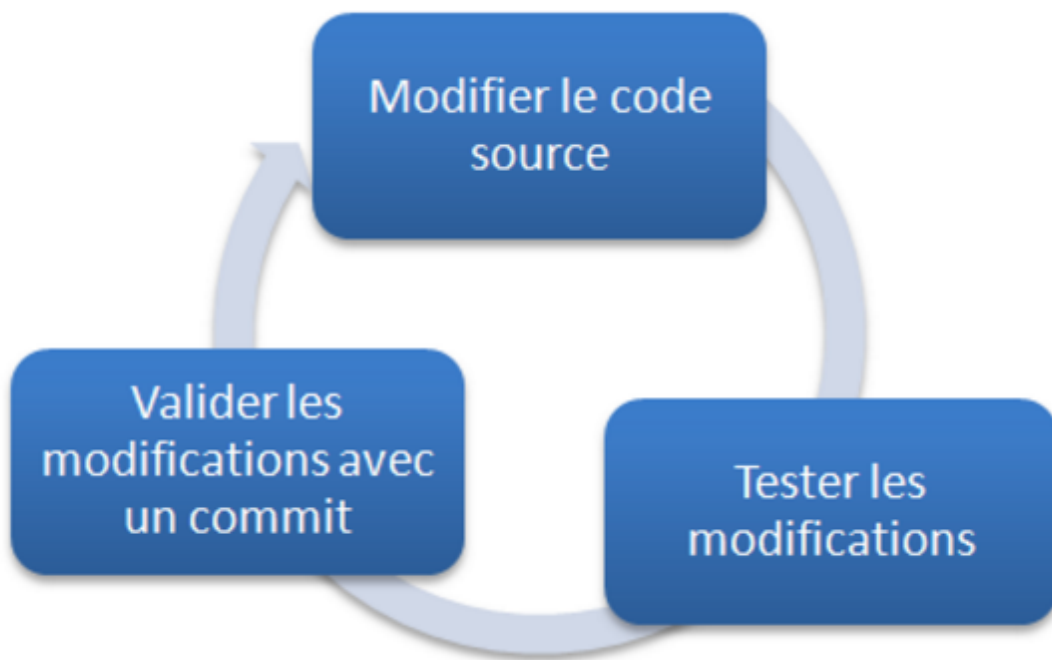


FIGURE 4 – Les étapes essentielles du fonctionnement du git

Nous avons aussi mis en place un système de relecture pour les documents. Après avoir fini l'écriture d'un document à rendre, on signale aux membres de l'équipe que celui-ci nécessite une relecture. Cela permet de diminuer le nombre de fautes de français et de vérifier que le document est complet et contient toutes les informations demandées.

4 Couverture des tests

La mesure de la couverture des tests représente le taux de code testé. Ainsi on considère une portion de code comme couverte si elle a été appelée pendant un test. La couverture des tests est un bon moyen de mesurer la qualité de la base de tests. Pour connaître la couverture de notre base de tests nous avons utilisé l'outil Cobertura qui permet de connaître le taux de code couvert pour chaque partie du projet et de visualiser les parties de code jamais testées.

En utilisant cet outil nous avons pu augmenter petit à petit la couverture en ajoutant des codes ciblés sur les parties non couvertes. Ainsi nous avons pu augmenter notre couverture de code jusqu'à 77% pour l'ensemble du projet. Il aurait idéalement fallu une couverture de 100% cependant nous n'avons pas eu besoin d'utiliser toutes les fonctions assembleur fournies et certaines parties du code sont difficiles à tester.

Voici les résultats détaillés donnés par cobertura après avoir lancé l'ensemble de tests :

Coverage Report - All Packages









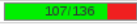
Package ↗	# Classes	Line Coverage
All Packages	248	77% 
fr.ensimag.deca	5	64% 
fr.ensimag.deca.codegen	2	85% 
fr.ensimag.deca.context	21	88% 
fr.ensimag.deca.syntax	50	66% 
fr.ensimag.deca.tools	4	90% 
fr.ensimag.deca.tree	86	87% 
fr.ensimag.ima.pseudocode	26	82% 
fr.ensimag.ima.pseudocode.instructions	54	78% 

FIGURE 5 – Les étapes essentiels du fonctionnement du git

Un exemple qui met en évidence l'importance de cet outil : dans la partie B, faute de quoi, on n'a pas testé l'appel circulaire des fichiers par "include", et on ne l'a pas remarqué et à vrai dire on n'a même pas pensé à le faire que lorsqu'on a lancé cobertura, chose qui montre l'importance de cet outil.

5 Autres méthodes de validation

Pour valider la pertinence de notre code nous avons aussi mis en place un système de re-lecture de code. Des membres du groupes sont chargés de relire le code écrit par les autres membres. Nous n'avons malheureusement pas suffisamment appliqué cette règle sur la fin du projet. En effet si nous l'avions bien appliqué sur toute la longueur du projet cela nous aurait sûrement permis de voir des erreurs plus rapidement.

L'extension nous a aussi permis de valider le code en vérifiant que celle-ci tournait bien avec le code effectué précédemment. En effet, à chaque fois que l'on ajoute une couche de code cela nous permet de vérifier le code réalisé en amont. Ainsi la réalisation de la partie C nous à aussi permis de déboguer la partie B. En effet, le code est construit par couche , il faut que le code en amont soit fonctionnel pour pouvoir faire la suite.