

Projet Génie Logiciel : Document de validation

Groupe 7 Equipe 34

BESSET Noé
FALL Rokhaya Yvette
GILLOT Aurélie
LESPINE Marilou
ROCHE Faustine

24 janvier 2024

Tables des matières

I - Descriptif des tests.....	2
a - Type des tests pour chaque étape/passe.....	2
b - Organisation des tests.....	2
c - Objectifs des tests.....	3
II - Script de tests.....	4
III - Gestion des risques et gestion des rendus.....	5
IV - Résultats de Jacoco.....	7
V - Autres méthodes de validation.....	8
VI - Annexes.....	9
Annexe 1 : Extrait du tableau des correspondances règles/tests.....	9

I - Descriptif des tests

a - Type des tests pour chaque étape/passe

Concernant l'étape B, qui consiste à vérifier les règles de la syntaxe contextuelle, nous avons listé toutes les règles. Puis, nous avons associé un test valide et un test invalide à chacune d'elles. Ceci nous permet de vérifier que notre code couvre bien toutes les règles contextuelles. En annexe 1, est inclus un extrait du tableau des correspondances règles/tests. Celui-ci nous permet de vérifier que l'on teste toutes les règles.

b - Organisation des tests

L'architecture des tests étant déjà établie, nous avons conservé celle-ci. Nous avons enrichi les répertoires *syntax*, *context* et *codegen*. Dans ces derniers, nous avons ajouté les tests dans le répertoire *valid* ou *invalid* selon le résultat attendu.

Un fichier test se présente comme ci-dessous :

```
// Description:
// Programme minimaliste qui test le Greater
//
// Resultats:
// test que Greater fonctionne : OK
//
// Historique:
// cree le 09/01/2024

{
    boolean x;
    x = 2>2;

    print ("test que Greater fonctionne : ");

    if (x==false){
        print ("OK");
    }
    else {
        print("ERRONE");
    }
}
```

Resultats indique le résultat attendu lorsqu'on exécute le fichier produit par la commande **decac**. Si le test doit retourner une erreur, alors dans la section **Resultats** se trouvera la ligne de l'erreur et l'exception qu'elle doit lever.

c - Objectifs des tests

L'objectif des tests est de vérifier que la spécification du langage Deca est bien respectée, et de tester le code que nous avons implémenté. Il faut donc écrire le maximum de tests afin d'exhiber le plus grand nombre d'erreurs possibles, et donc d'obtenir un livrable fiable. Pour cela, nous avons défini une organisation stricte à propos de la rédaction des tests :

- Tout d'abord, on s'assure d'écrire les tests de telle sorte que chaque exception soit levée au moins une fois. Pour ce faire, nous nous aidons de l'outil Jacoco qui nous indique si nos tests .deca couvrent bien les parties de codes que nous souhaitons couvrir. Ces tests-là sont des tests unitaires qui se concentrent sur une petite partie du code implémenté.
- Ensuite, nous écrivons des tests plus généraux qui couvrent plusieurs parties du code (voire la majorité), et nous nous assurons que l'arbre généré est celui attendu.

La plupart des tests sont écrits avant l'implémentation du code (à partir d'une liste de tests que nous pensons à faire), mais également pendant l'implémentation lorsqu'un nouveau scénario nous vient en codant, auquel cas nous l'écrivons sur le moment.

II - Script de tests

Afin de gagner du temps, nous avons choisi d'écrire des scripts (similaires à ceux déjà fournis) qui exécutent tous les tests présents (valides ou invalides) pour chaque étape du compilateur et nous les avons ajoutés au fichier pom.xml.

Pour la partie A, nous avons écrit le script **exec_SyntTests.sh** qui lance le programme **test_synt** sur chacun des fichiers que l'on retrouve dans le dossier de test **syntax**. Sur les tests valides, on vérifie si l'exécution s'est terminée sans erreur et à l'inverse, sur les tests invalides, on vérifie si l'exécution s'est terminée par une exception. Pour chacun, si l'exécution s'est déroulée comme escompté, le message du script est coloré en vert :

```
Succes attendu pour test_synt sur hello-many-instr.deca
-----
Succes attendu pour test_synt sur ifthenelse-simple.deca
-----
Echec attendu pour test_synt sur print1.deca
-----
Echec attendu pour test_synt sur print2.deca
```

Concernant la partie B, nous avons écrit le script **exec_ContTests.sh** qui va avoir le même comportement que le script précédent à la différence près qu'il lance **test_context** sur chacun des fichiers que l'on retrouve dans le dossier de test **context**.

Enfin, pour la partie C, nous avons écrit le script **exec_GenCodeTests.sh** qui va avoir un comportement différent des deux précédent. En effet, lors de l'exécution d'un test pour cette partie, il est nécessaire de faire plus de vérification que précédemment. Ici, les tests valides et invalides doivent pouvoir être exécutés par **decac** sans provoquer d'erreur, et il doit y avoir un fichier **.ass** généré. Ensuite, pour les tests valides, l'exécution de **ima** ne doit pas retourner d'erreur et le résultat de l'exécution doit être le même que celui indiqué dans l'entête des fichiers **.deca** que l'on a récupéré par une ligne de commande :

```
Test de null_dereferencing.deca :
Fichier null_dereferencing.ass généré.
Erreur à l'execution de null_dereferencing.ass
Erreur correct
```

Pour les tests invalides, l'exécution de **ima** doit retourner une erreur et l'erreur retournée doit correspondre à l'erreur indiquée dans l'entête des fichiers **.deca** (aussi récupérée automatiquement) :

```
Test de multiply.deca :
Fichier multiply.ass généré.
Bonne execution de multiply.ass
Resultat correct
```

III - Gestion des risques et gestion des rendus

Il est important de distinguer les conséquences potentielles d'une erreur dans la gestion du projet, la manière de programmer et de tester.

Avant le démarrage du projet, nous avons organisé des sessions de brainstorming pour identifier les risques potentiels liés à chaque étape du projet. Nous avons également effectué une analyse SWOT (forces, faiblesses, opportunités, menaces) pour évaluer les facteurs internes et externes qui pourraient influencer le projet (absence de certains membres pour cause de cours, compétences de certains que d'autres n'auraient pas...).

Afin de mener à bien ce projet, nous avons décidé de créer des branches sur notre git afin d'avoir une branche par partie et d'éviter des problèmes de conflits tout au long du projet. L'organisation des branches est la suivante :

- une branche master, dans laquelle seule l'étape A est mise à jour
- une branche pour l'étape B qui, lorsqu'elle sera implémentée, sera testée avec l'étape A de la branche master
- une branche pour l'étape C qui sera testée lorsque les étapes A et B seront fonctionnelles et testées.

Une fois que les 3 étapes seront mises en commun et testées, tout sera rapatrié dans la branche master, et les autres branches seront supprimées (pour éventuellement en créer des nouvelles).

On *git push* seulement le code testé et fonctionnel. Ainsi, chacun rédige des tests pour son code avant de partager ses modifications sur le git. Mais le risque est alors d'écrire deux tests équivalents. Comme solution, on se focalise alors sur les erreurs liées à notre partie, et pas à une autre. De plus, si un membre a un doute, il communique avec un membre d'une autre partie afin de savoir si le test est déjà traité.

Comme indiqué dans le polycopié (p.151 Section Tests), la commande *mvn test* active le script **common-tests.sh**. Cette commande viendra en complément de nos tests afin de vérifier que nous passons les tests les plus "importants". Cela nous aidera pour savoir si nous traitons les codes les plus communs. Nous gagnerons alors en crédibilité, avec un compilateur utilisable pour la base du langage Deca.

Concernant les délais à respecter, nous nous sommes basés sur les dates indicatives données sur les présentations : le langage "hello-world" doit être fonctionnel en fin de première semaine, et le langage "sans-objet" en fin de deuxième semaine. Ceci demande une communication essentielle entre les membres des différentes parties (A, B et C). Les parties doivent être fusionnées avant la fin des délais afin de régler les éventuelles collisions entre

codes. Nous avons évidemment dressé un diagramme de Gantt et nous tenons un Trello à jour afin de lister les tâches à réaliser pour chaque étape :

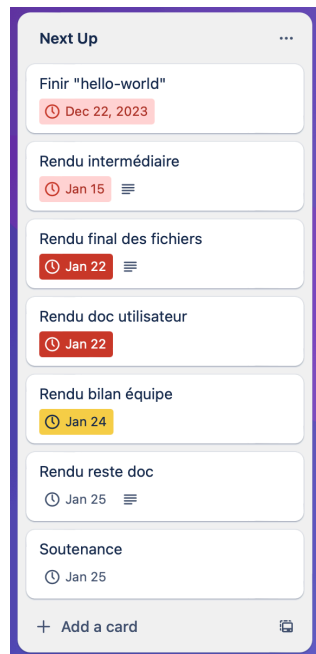


Figure 1 : Liste des deadlines



Figure 2 : Tâches à valider à chaque étape

De cette façon, nous minimisons les risques de manquer un délai, ou une étape dans notre validation.

Une chose qui paraît évidente mais qu'il est aussi important de mentionner est que nous avons chacun respecté les horaires de travail et avons décidé de travailler tous en présentiel (lorsqu'on le pouvait), afin de favoriser une bonne communication entre les membres de l'équipe mais aussi travailler dans un environnement similaire à une entreprise. Pour cela, nous nous sommes dès le début du projet fixés des horaires de "journées de travail" auxquelles nous devons nous tenir afin de garder un rythme actif.

IV - Résultats de Jacoco

L'outil Jacoco nous a permis de visualiser la couverture de nos tests sur notre code. Ainsi, nous parcourons la page html générée afin de repérer les lignes que les tests ne couvraient pas. Il est alors possible de créer des tests supplémentaires afin de couvrir ce code. La commande qui génère cette page html est la suivante : **mvn -Djacoco.skip=false verify**.

Lors du rendu final, cette commande donnait une couverture de 78%. Notons que nous n'avions pas créé de script qui teste les options. Or, le seul moyen de parcourir les méthodes **decompile()** est en appliquant l'option **-p**. Cette méthode étant présente dans presque toutes les classes, ceci diminue notre couverture de code.

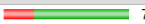
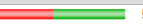










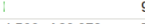
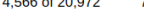
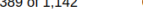
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		75%		56%	456	652	500	1,981	254	369	4	48
fr.ensimag.deca.tree		84%		79%	151	635	298	1,631	85	436	0	87
fr.ensimag.deca		55%		43%	46	82	104	236	11	40	2	5
fr.ensimag.deca.context		85%		78%	30	155	48	297	18	118	0	21
fr.ensimag.deca.codegen		82%		62%	21	77	27	192	7	50	0	5
fr.ensimag.ima.pseudocode		74%		75%	29	87	48	185	24	77	2	26
fr.ensimag.ima.pseudocode.instructions		72%		n/a	18	62	30	111	18	62	13	54
fr.ensimag.deca.tools		94%		100%	1	16	3	39	1	13	0	3
Total	4,566 of 20,972	78%	389 of 1,142	65%	752	1,766	1,058	4,672	418	1,165	21	249

Figure 1 : Résultats de couverture fourni par Jacoco

Lorsqu'on ajoute un script qui appelle l'option **-p**, la couverture augmente de 3% pour atteindre 81%.

V - Autres méthodes de validation

Une méthode de validation utilisée, autre que les tests, sont les affichages à l'aide de traces de débogage, qui nous auront servis notamment pour afficher les environnements, leurs méthodes et leurs champs ainsi que les définitions associées. Cela nous permettait à la fois de déboguer notre code, mais aussi de s'assurer que les héritages entre les classes et les empilements d'environnements se passaient comme nous le voulions.

Pour cela, nous avons utilisé la bibliothèque **log4j**, grâce à laquelle nous pouvons choisir le niveau de debug à afficher : six niveaux classés du moins impactant (**trace**) aux plus importants (**fatal**), en passant par **debug**, **info**, **warn** et **error**. Dans le code, nous utilisons donc une instruction de la forme : ***LOG.debug("debug message");***

On utilise ensuite la fonction `setLevel` du logger ou le fichier de configuration **log4j.properties** pour modifier le niveau de debug souhaité lors de nos compilations, selon les critères à tester.

VI - Annexes

Annexe 1 : Extrait du tableau des correspondances règles/tests

Message	Explication	Type	Fichier/Classe/Méthode	Test valide	Test invalide
Contextual error : the name of the field is already used in superclass, but not as a field name(règle 2.5)	Règle (2.5)	Syntaxe contextuelle	DeciField.java verifyDeciField	test_deci_field_multiple.deca class A {int a;} class B extends A {int a;}	test_class_deci_field_invalid.deca class A { int a; int getA(){ return a; } } class B extends A { int getA; }
Contextual error : field cannot be void type (règle 2.5)	Règle (2.5)	Syntaxe contextuelle	DeciField.java verifyDeciField		test_deci_field_void.deca class A {void a;}
Contextual error : method "method name" already declared (règle 2.7)	Règle (2.7)	Syntaxe contextuelle	DeciMethod.java verifyDeciMethod	test_deci_met.deca class A { int get(); float set(); }	test_deci_meth_multiple.deca class A { int get(); int get(); }