
DOCUMENTATION DE VALIDATION

EQUIPE 20

GROUPE 4



Table des matières

1	Objectifs des tests	2
2	Les scripts de tests	3
2.1	Règle de de mise en place	3
2.2	Appel automatisé	4
3	Gestion des risques et gestion des rendus	7
4	Résultats de Jacoco	8
5	Méthodes de validation utilisées autres que le test	9

Objectifs des tests

Nous n'avons pas mis en place de tests unitaires à proprement parler. Les tests unitaires ont finalement été combinés dans des tests fonctionnels incluant possiblement plusieurs opérations arithmétiques. Notre organisation a été tournée vers un test par fonction donnée.

Pour atteindre un objectif de taux de couverture des tests important il a fallu utiliser Jacoco couplé avec des règles de définition des résultats attendus. Ces points sont détaillés dans la suite des parties.

De nombreuses itérations nous ont permis de récupérer tous les tests non fonctionnels et de les traiter un par un afin de valider l'intégralité du projet.

Les scripts de tests

Dans cette partie nous allons détailler comment nous avons pu créer toute une série de scripts de tests automatisés. Au vu du nombre de tests importants que nous avons dû mettre en place, il a semblé indispensable d'automatiser le lancement des batteries de tests.

Cela a été essentiel pour nous permettre de voir très rapidement si certains d'entre eux n'étaient pas bien exécutés. La mise en place de script de tests nous a aussi permis d'automatiser le balayage de tous les fichiers et de vérifier qu'aucun oubli n'a été mis en avant par Jacoco.

2.1 Règle de de mise en place

Afin de rendre possible l'automatisation des scripts de tests, nous avons dû définir des règles de conception sur chaque fichier. Si les règles de conception sont mal mises en place, cela est détecté lors du lancement de la batterie de test.

Nous pouvons donc ensuite, au cas par cas, analyser si l'erreur qui ressort est plutôt liée à un problème de respect de la règle ou bien si il s'agit d'un problème algorithmique dans notre fonction.

Voici un exemple de mise en place de règle dans le fichier `DebordementArithFlot.deca`

```
1 // Description:
2 //   Erreur du 11.1 : débordement arithmétique flottant
3 //
4 // Resultats:
5 // Error: overflow_error
```

Cette mise en place nous a permis par la suite de savoir où étaient situés les résultats attendus. En l'occurrence, il s'agissait de la ligne 5, après trois caractères. Avec la commande :

```
head -n 5 cond0.deca | tail -n 1 | cut -c4-
```

Nous pouvons récupérer la chaîne de caractère "Error:overflow_error" qui sera utile ensuite pour la comparaison du résultat de sortie.

2.2 Appel automatisé

Afin d'automatiser l'appel de chaque script, nous avons du lister en amont toutes les commandes utiles que nous allions devoir appeler successivement : Les commandes globales de maven pour lancer un nettoyage / compilation ou vérification sur le projet entier.

```
mvn clean
mvn compile
mvn verify
```

Les commandes spécifiques pour lancer des fichiers bien précis

```
rm [Nomfichier].ass pour supprimer des fichiers compiler
decac [Nomfichier].decac pour recompiler un fichier deca et avoir une sortie .ass
ima [Nomfichier].ass pour exécuter le fichier compilé et analyser les sorties
```

Toutes ces commandes shell étant à automatiser, nous avons choisis d'utiliser python pour pouvoir avoir une rapidité de mise en place et une facilité de récupération des informations utiles. Il y a donc eu deux librairies importantes qui nous ont permis de lancer ces commandes shell sur l'ensemble des fichiers que nous visions :

```
import glob
import subprocess
```

La librairie glob permet d'assurer la recherche de chemin type. Cela nous permet de

parcourir tout le projet à la recherche des fichiers .deca. Une fois ces fichiers repérés, leurs chemins relatifs sont stockés dans un tableau.

Exemple pour chercher tous les fichiers .ass situés dans les sous répertoires de deca, nommés valid

```
files = glob.glob('./src/test/deca/**/*.ass', recursive = True)
```

Sur la base de ce tableau, nous pouvons lancer les commandes shell de compilation (decac) grâce à la librairie subprocess.

Exemple pour lancer une commande decac sur un fichier donné avec subprocess

```
cmd = subprocess.run(['decac', file], capture_output=True)
```

Exemple pour lancer une commande ima sur un fichier donné avec subprocess

```
cmd = subprocess.run(['ima', file], capture_output=True)
```

L'attribut capture_output=True est essentiel afin de récupérer les sorties shell. Cela va nous permettre par la suite de comparer la sortie obtenue, avec la sortie prévue dans la règle de mise en place que nous avons détaillé plus haut. Une fois que cmd contient toute l'information shell, il faut extraire l'output qui nous intéresse grâce à :

```
sortie_shell = cmd.stdout.decode('ascii')
```

C'est cette commande qui nous permet d'obtenir les résultats type de la règle de conception comme par exemple :

```
Error: overflow_error
```

A cette étape, nous avons donc compilé un fichier, lancé son exécution et récupéré la sortie shell. Il nous reste l'étape de vérification à faire grâce à des fonctions de parcours de fichiers implémentées en python :

```
filin = open(fichier_cible, "r")
```

La variable filin peut ensuite être traitée avec readlines pour séparer chaque ligne :

```
lignes = filin.readlines()
```

Il suffira ensuite de cibler la ligne voulue en considérant "lignes" comme un tableau à 2 dimensions (cases X,Y) :

```
sortie_attendue = lignes[4][3:]
```

Nous avons toutes les informations qui nous permettent de comparer les `sortie_shell` et `sortie_attendue`. Dans le cas où les deux sont identiques, on ne fait rien. Dans le cas où il y a une différence, on affiche sur le terminal les deux sorties ainsi que le chemin du fichier concerné.

Une fois que nous maîtrisons ces commandes, nous avons pu passer à l'étape de généralisation pour avoir un script complètement automatisé.

Étape 1 : supprimer tous les fichiers `.ass` qui pourraient rester depuis les dernières. (utilisation de `glob` pour identifier les fichiers, puis `subprocess` avec la commande `rm` pour la suppression)

Étape 2 : compilation de tous les fichiers `.decac`

Étape 2.1 : pour chaque fichier `decac` situé dans `context/invalid`, print les erreurs et comparer avec le résultat attendu

Étape 3 : exécution de tous les fichiers `.ass`

Étape 3.1 : pour chaque fichiers `ass`, il faut lancer la commande `ima`

Étape 3.2 : pour chaque sortie, il faut comparer par rapport à la sortie attendue et print si une erreur est présente.

Cas particuliers : Certains fichiers se lancent avec des inputs. L'algorithme python va donc automatiquement provoquer une erreur dessus. Le fait d'avoir un visuel sur ces erreurs permet d'analyser les fichiers concernés et d'ajouter des conditions particulières sur les fichiers nécessitant un input. Par exemple :

```
1  #condition listant tous les fichiers spéciaux qui nécessitent des inputs pour être
    testés
2      if file == './src/test/deca/codegen/valid/ReadFloat.ass' :
3          print ("Test avec un Float : ")
4          cmd = subprocess.run(['ima', file], input='1.0', text=True, capture
    \_output=True)
5          print (cmd)
```

Gestion des risques et gestion des rendus

Afin d'éviter toute situation bloquante au niveau du projet l'utilisation de git a été très utile pour :

- Gérer le versioning de fichier
- Travailler en parallèle sur des tests fonctionnels
- Utiliser des branches dev pour l'équipe d'ingénieur, master pour remettre des release stables au client

Le client pouvait donc suivre en temps réel les avancées des ingénieurs afin de valider plusieurs jalons définis ensemble lors du kickoff projet.

Les risques identifiés ont été partagés en amont du lancement et tout au cours du projet si des nouveautés arrivaient. L'objectif a été de montrer au client que la demande pouvait avoir certaines limitations techniques dans le temps imparti. Nous avons donc proposé au client des solutions afin d'être certains que le livrable soit exploitable pour lui même si certaines features ne sont pas atteintes.
















4

SECTION

Résultats de Jacoco

L'objectif a été d'obtenir un pourcentage de couverture le plus élevé possible. Grâce à l'implémentation automatisée détaillée auparavant, nous pouvions vérifier facilement ce critère avec Jacoco.

A ce jour nous obtenu un taux de couverture de 78%

Deca Compiler												
Deca Compiler												
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		71%		50%	698	891	603	2,133	256	372	3	48
fr.ensimag.deca.tree		87%		78%	187	747	268	2,095	99	501	6	91
fr.ensimag.deca		62%		55%	59	126	110	309	10	52	3	6
fr.ensimag.deca.context		75%		69%	52	149	63	262	44	131	1	22
fr.ensimag.ima.pseudocode		86%		80%	19	85	24	182	15	75	1	26
fr.ensimag.ima.pseudocode.instructions		78%	n/a	n/a	15	62	26	111	15	62	11	54
fr.ensimag.deca.codegen		91%		81%	7	44	11	126	3	33	0	2
fr.ensimag.deca.tools		89%		80%	4	20	6	46	2	15	0	3
Total	5,305 of 24,306	78%	674 of 1,697	60%	1,041	2,124	1,111	5,264	444	1,241	25	252

Ce qui n'est pas couvert concerne la programmation défensive.

Il est donc normal de ne pas aller couvrir ces parties car il s'agit de parties de codes mortes : des exceptions dans lesquels nous ne tombons jamais.

Méthodes de validation utilisées autres que le test

La dernière méthode de validation autre que celle utilisée précédemment a été d'implémenter l'extension TRIGO qui utilise nos algorithmes deca.

On remarque que la compilation et l'exécution se fait sans erreur, en revanche on remarque aussi des écarts sur les résultats attendus / obtenus.

Plus de détails sur ces écarts et comment ils ont été traités sont disponibles dans la section dédiée à l'extension.