

Documentation de validation

Projet GL



ELHAJ-LAHSEN Hugo
GONDET Matthias
LAVAL Jean
MAZOUZ Mahmoud
NADJA Mohamad

Table des matières

Descriptif des tests	3
Organisation des Tests	3
Stratégie des tests d'intégration	4
Implémentation de l'oracle	6
Format des tests et description	7
Stratégie pour les tests unitaires	7
Scripts de tests	8
Comment faire passer tous les tests	8
Résultats de Jacoco	9
Autres méthodes de validation que le test	9
Environnement de CI	9
Utilisation du code review	11
Gestion des risques et gestion des rendus	12
Release management process	15

Descriptif des tests

A la fin de notre projet, nous avons 400 tests au total.

- 320 Integration Tests
- 80 Unitary Tests
- Un test automatisé : Test_location pour les tests de localisation automatiques

Organisation des Tests

Afin de rendre notre vision plus claire lors des tests, nous avons suivi les divisions qui nous ont été données pour ces tests en y ajoutant des mini-divisions pour les différentes étapes que nous avons parcourues tout en testant le tout depuis le tout début.

Alors que les 3 types principaux des Tests étaient :

- Syntax
- Context
- Codegen

Nous avons divisé dans ces 3 types de tests en deux sections :

- Valid
- Invalid

Après cela, nous avons divisé nos tests, qu'ils soient valides ou invalides, en plusieurs étapes que nous avons traversées tout au long du projet.

Examples of increment

- First goals:
 - ▶ Compile the empty program
 - ▶ Compile a hello-world
- Without objects:
 - ▶ Simple expressions (2+2, 2-2, ...)
 - ▶ Variables (int, float)
 - ▶ Control-structures (if/while)
- Objects:
 - ▶ Objects without methods
 - ▶ Methods (definitions and calls)

Planning should be driven by **language subsets**,
not by stage/passes (B1, B2, B3, C1, C2)

Notre organisation pour les tests a été inspirée par la manière dont nous avons implémenté notre code en suivant cette image au dessus et par cela on a obtenu le résultat suivant :

mohamad najda | January 25, 2022

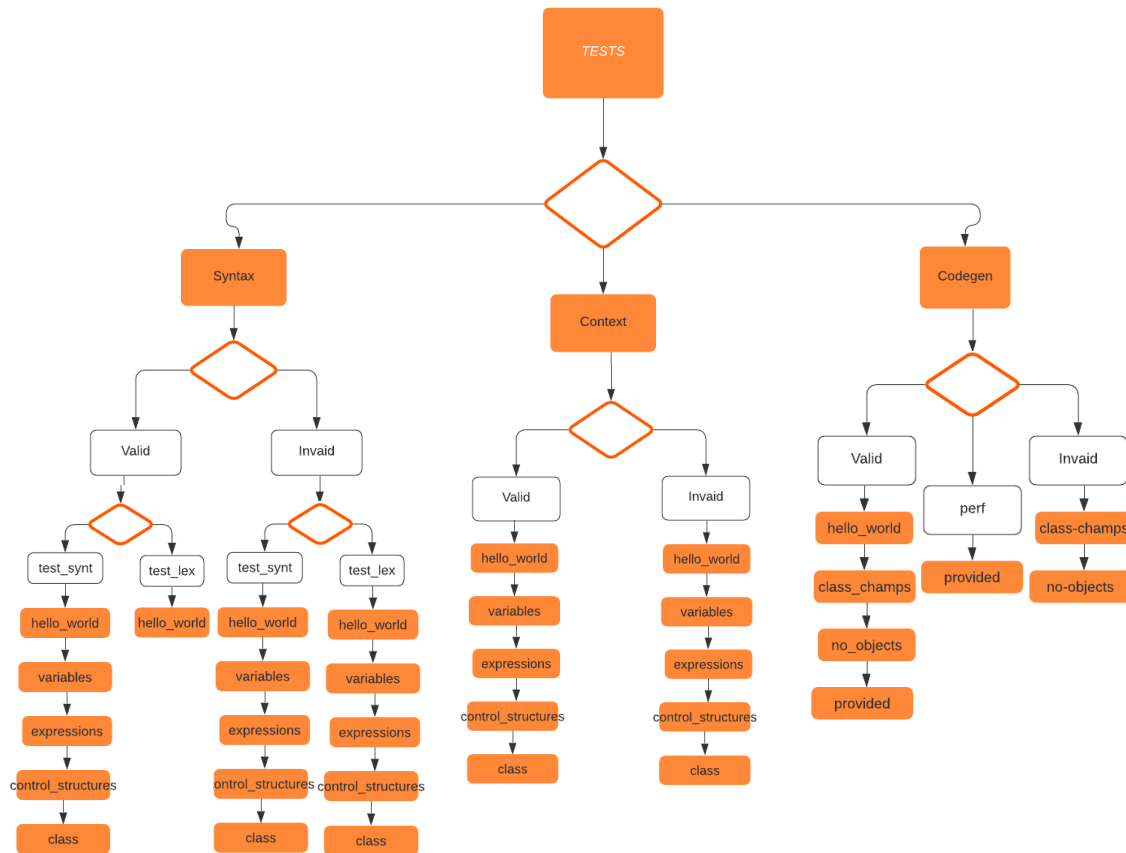


Figure 1: Organisation des tests

Stratégie des tests d'intégration

Au tout début, nous faisons ces tests de manière assez aléatoire, mais nous nous sommes rendu compte que nous ne couvrons pas une grande partie du code que nous écrivons, c'est pourquoi nous sommes passés à une stratégie qui a permis de couvrir plus de code et de savoir quelles erreurs nous avons dans notre compilateur.

Après les tests des deux premières semaines, nous avons suivi les erreurs possibles décrites dans le poly. Notre stratégie était une stratégie de couverture maximale, avec un nombre de tests limité mais pertinent.

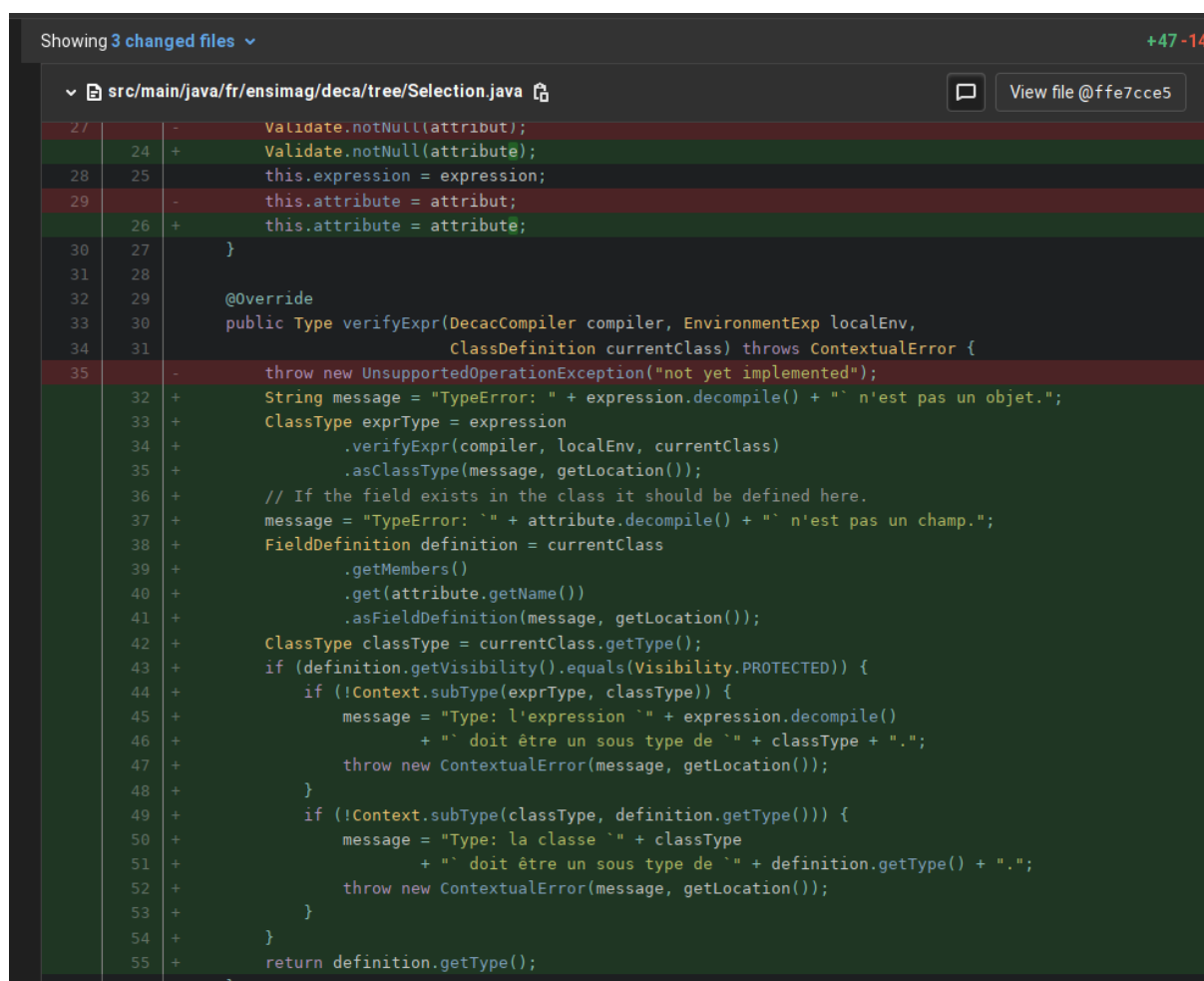
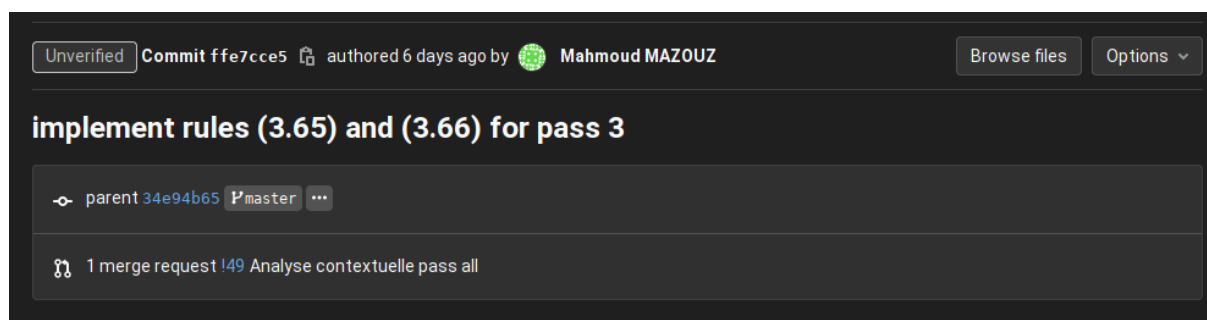


Figure 2: Exemple de commit (partie contextuelle) décrivant une règle

Le test étant fait après le commit, nous pouvons vérifier après que toutes les erreurs soient bien touchées dans le code.

Nous avons vérifié les tests avec la grammaire contextuelle page 80 à 89. Pour chaque condition de chaque règle, nous écrivons un test qui teste cette condition et affiche l'erreur.

Une fois que l'implémentation chaque commit dans l'implémentation de la contextuelle décrivait l'implémentation d'une règle, et nous pouvions voir en cliquant dans le commit chaque erreur qu'il affiche.

Enfin, nous avons décrit toutes les erreurs possibles dans le manuel utilisateur, en décrivant pour chaque erreur une description, le message d'erreur et le test associé.

(Rule 3.66)

→ Selection [

```
expr ↓env_types ↓env_exp ↓class ↑type_class(class2 )
      field_ident ↓env_exp2 ↑protected ↑class_field ↑type ]
|
condition (class(__, env_exp2), __) , env_types(class2)
et subtype(env_types,type_class(class2),type_class(class))
et subtype(env_types,type_class(class),type_class(class_f ield))
```

Description: le premier type d'erreur qu'on peut avoir avec les champs et lorsqu'on utilise dans notre main un champ protégé ce qui nous affiche une erreur.

Page 12 sur 15

Charte de travail en équipe

JuNGLE

Message d'erreur: ScopeError: le champ `` + attribute.decompile() + `` est protégé.

Tests: src/test/deca/context/invalid/class/bad_use_protected.deca

Figure 3: Exemple de description de message d'erreur

Implémentation de l'oracle

Nous voulions plus de précision sur les tests que simplement savoir si le programme réussissait ou échouait. Pour cela, nous avons mis en place un oracle qui vérifie l'exécution de chaque programme par rapport à une trace: .lis pour la partie A et B, .res pour la partie C.

Nous ne générons pas les traces manuellement. A la place, nous avons créé un script bash permettant de les générer automatiquement, puis nous vérifions à la main que le résultat était cohérent : bonne erreur, bonne sortie si le test est valide.

```
// Description:
//   Test de déclaration de class avec un field protégé
//                                     ( Rule 3.66 )
//
// Resultats:
```

```
//    Lexeur: pas d'erreur
//    Parseur: pas d'erreur
//    context: en context il ya une erreur car on ne peut pas
//    utiliser
//            un champs proteger directement sans des guetteurs
//
//()
// Historique:
//    cree le 23/01/2022

class coordonnees{
    protected int x = 5;

    int plan(int z){
        return this.x;
    }
}

{
    coordonnees coords = new coordonnees();
    coords.x = 6 ;
}
```

Figure 4: Listing de test pour une condition de contextuelle (bad_use_protected.deca)

Format des tests et description

Nous avons respecté le format fourni sur les tests. Chaque test possède une description, sa sortie exacte que nous mettons également dans la trace.

Nous avons ajouté une syntaxe spéciale `//! IMA_OPTIONS: <options>` et `//! DECAC_OPTIONS: <options>`. Ceci nous permet de lancer certains tests avec des options différentes à la compilation ou à l'exécution. Ceci est utile en particulier dans les tests codegen.

Stratégie pour les tests unitaires

Pour les tests unitaires, nous avons testé les parties du code simple à tester unitairement pour toucher avec Jacoco. La majorité des tests unitaires porte sur la partie contextuelle, et nous avons également des tests pour certaines classes importantes.

L'organisation des tests unitaires suit la structure du code: nous avons une classe de test unitaire par classe du code qui teste les fonctionnalités de chaque classe.

Chaque test utilise extensivement les mock pour éviter de devoir tester d'autres classes que les classes nécessaires, et teste ainsi uniquement la fonctionnalité simulée. Quand nous avons un bug dans ces parties, les tests unitaires nous permettent de voir immédiatement ce qui casse, ce qui est plus dur dans un test d'intégration.

Scripts de tests

Pour automatiser le lancement des tests d'intégration, nous avons fait immédiatement un script en python permettant de lancer tous les tests d'intégration. Ce script est intégré au `mvn test` et le fait échouer si tous les tests ne passent pas.

```
[21/23] REUSSI: codegen/valid/class-champs/assign_object_direct_init.deca
[22/23] REUSSI: codegen/valid/class-champs/assign_object_field.deca
[23/23] AVERTISSEMENT: pas de résultat .res trouvée pour codegen/valid/class-champs/cast_class_to_class.deca
[SUITE] RESULTAT Tests exécution, class-champs: Tests lancés: 23, Echecs: 3
[ECHECS] Liste des tests échoués:
ECHEC codegen/valid/class-champs/methodDeclaration.deca
ECHEC codegen/valid/class-champs/Cast_between_int_and_float.deca
ECHEC codegen/valid/class-champs/instanceof.deca
[SUITE] Tests execution, class-champs (invalid):
[1/2] REUSSI: codegen/invalid/class-champs/field_class_member_recursive_def.deca
[2/2] REUSSI: codegen/invalid/class-champs/field_class_member_no_init.deca
[SUITE] RESULTAT Tests exécution, class-champs: Tests lancés: 2, Echecs: 0

[RAPPORT GLOBAL]: Tests lancés: 289, Echec: 10
Liste des tests échoués:
ECHEC codegen/valid/no-objects/divide_by_float_with_checks.deca
ECHEC codegen/valid/no-objects/ln2.deca
ECHEC codegen/invalid/no-objects/overflow_divide.deca
ECHEC codegen/invalid/no-objects/overflow_sub.deca
```

Figure 5: Exemple d'exécution du `test.py` (en local)

Un test est compatible si il est dans `valid/` et le test réussit, ou si il est dans `invalid/` et le test échoue.

Dans le script de test, trois résultats sont possibles pour chaque test:

- **RÉUSSI**: le test est comparé avec sa trace et il est compatible.
- **AVERTISSEMENT**: le test est compatible, mais pas de trace n'est disponible pour ce test. Ceci ne compte **pas** comme une erreur.
- **ÉCHEC**: le test n'est pas compatible (qu'il ait une trace ou pas).

Comment faire passer tous les tests

A l'exécution, tous les tests sont censés passer automatiquement. Si un test ne passe pas, vous pouvez le désactiver en supprimant le test correspondant. Vous pouvez également supprimer tout un dossier de test en commentant la ligne correspondante à la fin du `test.py`.

Résultats de Jacoco

Notre but principal était l'atteinte de 80% sur Jacoco. Nous n'avons pas réussi à atteindre ce but, atteignant uniquement 77,7%.

Vers la fin du projet, nous avons mis en œuvre un indicateur sur le Jacoco sur chaque commit que nous décrivons dans Autres méthodes de validation utilisées.

En particulier, quand un bout de code n'était pas couvert sur Jacoco, notre processus était le suivant:

1. Est-ce que le bout de code est un assert/une exception DecacInternalError? Nous avons choisi de ne pas tester ces erreurs, car un test qui passe sur ces erreurs indique une erreur sur notre compilateur.
2. Est-ce que nous pouvons supprimer ou refactor le code correspondant? Si oui, nous n'avons pas besoin de tester le code s' il n'existe plus.
3. Est-ce que le code est une erreur utilisateur ? Si oui, nous pouvons écrire un test d'intégration qui couvre ce test.
4. Est-ce que ce test peut être couvert par un test unitaire? Nous préférons les tests unitaires aux tests d'intégration car ils sont plus légers à lancer sur le mvn test.
5. Faire un test d'intégration pour ce cas.

Autres méthodes de validation que le test

Environnement de CI

Pour avoir un environnement consistant pour chaque personne de l'équipe, et vérifier que les tests ne passent pas juste sur une machine, nous avons mis en place un environnement de CI (une pipeline sur GitLab).

Cet environnement, pour chaque commit sur chaque branche, récupère le projet et les tests, fait le mvn compile et le mvn verify, et nous donne un statut sur le Git nous indiquant à la fois si les tests passent, et le pourcentage de Jacoco.

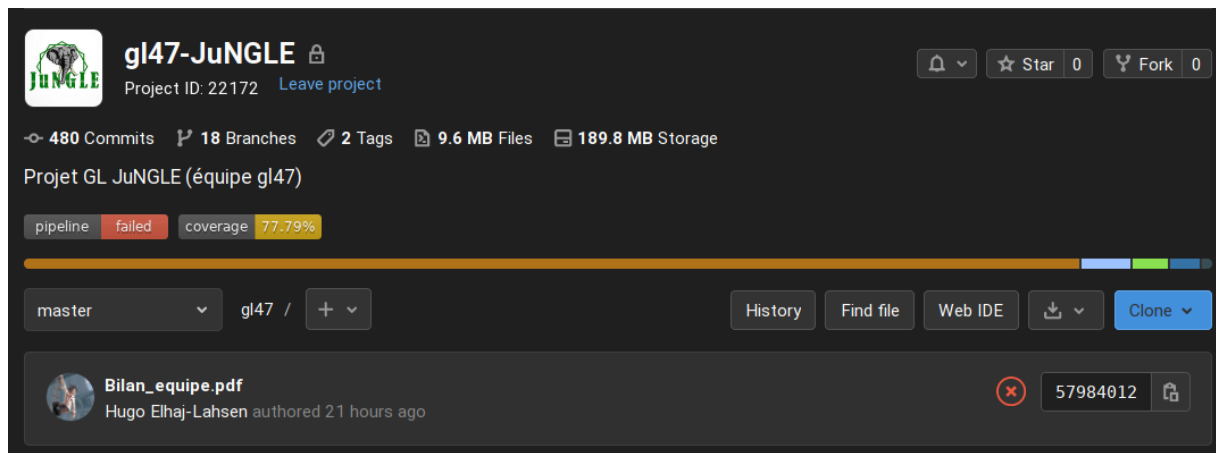


Figure 6: Page de garde du projet, montrant le badge “pipeline” et le coverage

Chaque pipeline à deux phases: une phase compile et une phase test. Si une pipeline échoue, nous savons dans quelle étape elle échoue.

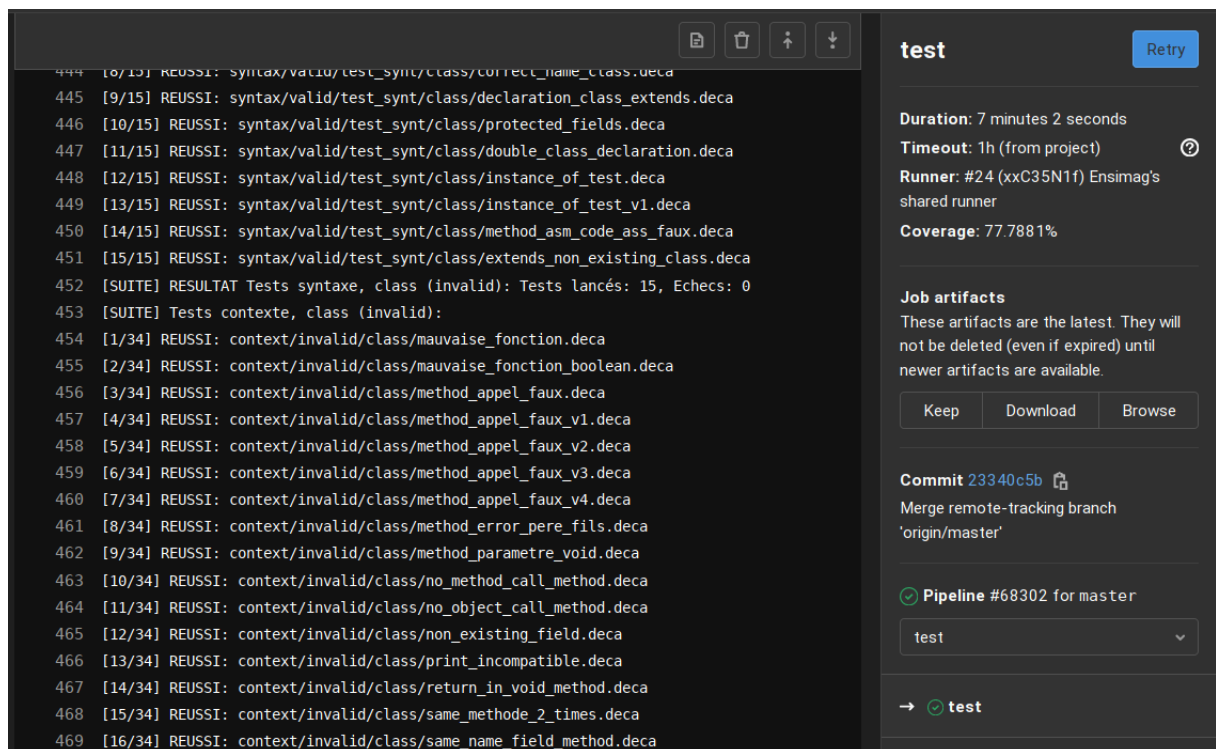


Figure 7: Exemple d'exécution de pipeline: affichant le résultat et la couverture

Pour n'importe quelle pipeline, il est possible de voir le Jacoco correspondant en allant dans Job Artefacts > Browse > target > site et cliquer sur index.html.

Pour modifier la pipeline, editez le fichier gitlab-ci.yml.

https://gl2022.pages.ensimag.fr/-/jobs/238882/artifacts/target/site/index.html

130 %

imag

jsho

draw

kanji

jp

JuNGLE

proj

GL

Deca Compiler

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax	<div><div></div></div>	77%	<div><div></div></div>	53%	675	873	484	2,102	245	365	1	48
fr.ensimag.deca.tree	<div><div></div></div>	81%	<div><div></div></div>	75%	174	708	350	1,974	100	523	0	86
fr.ensimag.arm.pseudocode	<div><div></div></div>	0%	<div><div></div></div>	0%	39	39	109	109	34	34	13	13
fr.ensimag.deca	<div><div></div></div>	75%	<div><div></div></div>	72%	24	70	57	199	5	31	2	5
fr.ensimag.ima.pseudocode	<div><div></div></div>	84%	<div><div></div></div>	77%	31	112	37	235	22	92	2	26
fr.ensimag.deca.context	<div><div></div></div>	87%	<div><div></div></div>	78%	29	161	39	268	18	129	0	24
fr.ensimag.ima.pseudocode.instructions	<div><div></div></div>	71%	<div><div></div></div>	n/a	19	63	33	112	19	63	13	55
fr.ensimag.arm.pseudocode.instructions	<div><div></div></div>	0%	<div><div></div></div>	n/a	7	7	14	14	7	7	3	3
fr.ensimag.arm.pseudocode.syscalls	<div><div></div></div>	0%	<div><div></div></div>	n/a	6	6	8	8	6	6	2	2
fr.ensimag.deca.tools	<div><div></div></div>	93%	<div><div></div></div>	100%	1	16	3	37	1	13	0	3
Total	5,005 of 22,533	77%	593 of 1,520	60%	1,005	2,055	1,134	5,058	457	1,263	36	265

Figure 8: Exemple de Jacoco pour cette pipeline

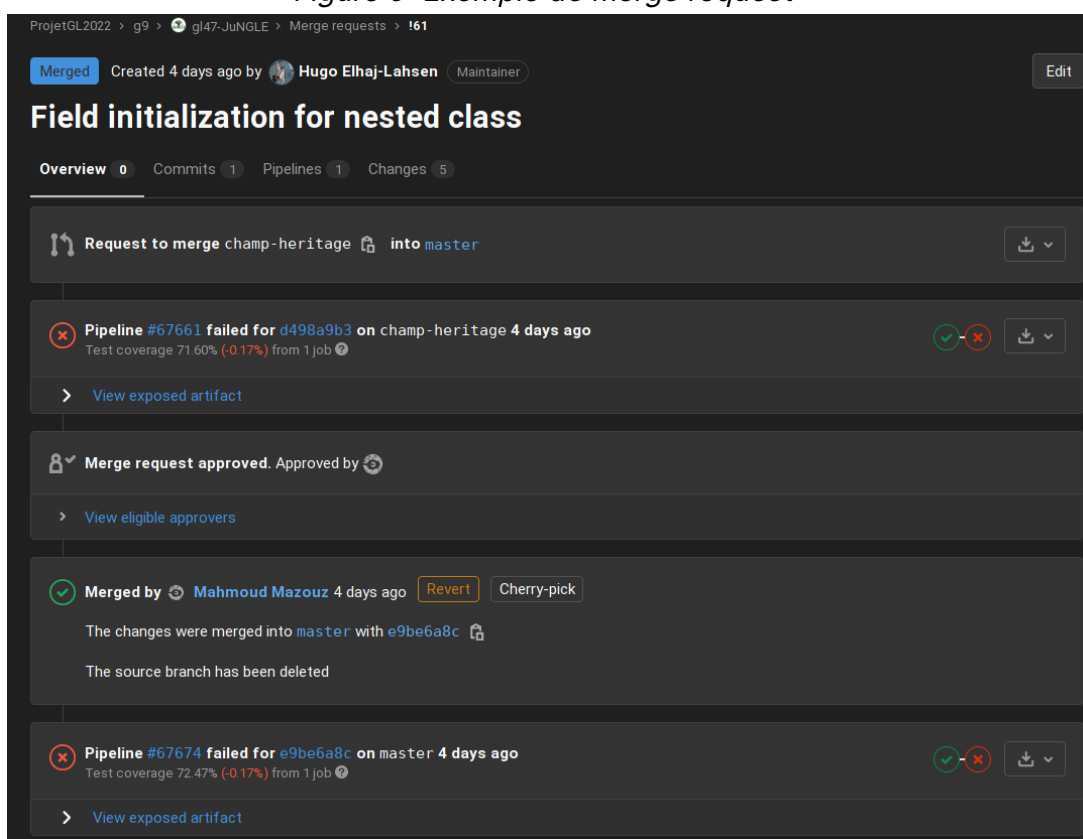
Utilisation du code review

Nous avons mis en place un processus de code review avec les Merge Request de GitLab, qui est intégré à notre système de CI.

Aucune personne ne devrait push directement sur master. Pour pouvoir avoir ses changements sur master, une personne devait d'abord demander une review a un autre membre de l'équipe, qui devait revoir ses changement et voir si tout est bon ou s' il y a des problèmes.

De plus, le pipeline exécuté sur la branche du merge request nous laisse savoir si des régressions sont apparues, nous laissant développer sans avoir nécessairement la discipline de lancer les tests à chaque commit.

Figure 9: Exemple de merge request



ProjetGL2022 > g9 > gl47-JuNGLE > Merge requests > #61

Merged Created 4 days ago by **Hugo Elhaj-Lahsen** (Maintainer) Edit

Field initialization for nested class

Overview 0 Commits 1 Pipelines 1 Changes 5

Request to merge champ-heritage into master

Pipeline #67661 failed for d498a9b3 on champ-heritage 4 days ago
Test coverage 71.60% (-0.17%) from 1 job

> View exposed artifact

Merge request approved. Approved by

> View eligible approvers

Merged by Mahmoud Mazouz 4 days ago Revert Cherry-pick

The changes were merged into master with e9be6a8c

The source branch has been deleted

Pipeline #67674 failed for e9be6a8c on master 4 days ago
Test coverage 72.47% (-0.17%) from 1 job

> View exposed artifact

Gestion des risques et gestion des rendus

Dans cette partie, nous décrivons les risques sous forme de Risk Matrix. Cette matrice décrit chaque risque, son impact sur le projet et sa probabilité.

Pour chaque risque, nous expliquerons sa probabilité et les actions que nous avons prises pour minimiser sa probabilité/son impact.

		Impact			
		Acceptable (peu d'effet)	Tolérable (effets sont sentis, mais non critique)	Inacceptable (impact sérieux sur le reste du projet)	Dévastateur (effet dévastateur sur le projet)
Proba	Improbable	1	2	3	4
	Possible	5	6	7	8
	Probable	9	10	11	12

Improbable, inacceptable (3):

- Le document de conception d'extension n'est pas complet
- Le document de l'analyse de l'impact énergétique n'est pas complet
- Le document de validation n'est pas complet

Ces documents sont à rendre après le rendu final, et nous avons une marge de temps assez importante pour les peaufiner après le rendu final. De plus, ils ne seront pas négligés pendant le projet, une attention étant à préciser sur eux pendant les rendus.

- Les tests ne peuvent pas être lancés par le compilateur du prof

Pour mitiger ce risque, malgré notre travail important sur la structure des tests, en particulier pour séparer les tests en sous-langages granulaires, nous avons toujours respecté la convention des répertoires de test¹.

Improbable, dévastateur (4):

- Un des membres de l'équipe quitte le projet

¹ [Tests], 3.1 Classification des tests "deca", p 151

Cette probabilité est faible mais, comme nous l'avons déjà ressenti, a un impact extrêmement fort sur le projet. Si deux personnes venaient à quitter le projet, l'impact serait trop important pour pouvoir finir le projet gl.

- Le rendu intermédiaire/final ne compile pas

La majorité de notre infra de test nous garantit que le projet compile bien. En effet, nous avons une étape spécifique dans notre CI qui vérifie bien que notre projet compile tout le temps, ce qui minimise la proba de ces risques.

Cependant, il ne faudra pas oublier de réparer le projet quand nous voyons que les tests échouent. Les tests nous sont utiles pour bien diriger nos efforts.

- Le rendu intermédiaire/final ne passe pas les tests
- Le compilateur ne peut pas compiler un "hello world"
- Le code machine généré est invalide

De la même manière, nous avons une étape dans le CI qui fait tourner tous les tests. Ceux-ci vérifient entre-autres qu'on puisse bien lancer le compilateur, compiler un programme, et l'exécuter sur la machine virtuelle ima.

- La ligne de commande du compilateur ne marche pas
- Le compilateur ne passe pas le script test `common-tests.sh`

Notre CI lance ce script et échoue le build s'il ne passe pas.

Possible, tolérable (6):

- Un des membres de l'équipe est temporairement indisponible

Pour mitiger l'impact de ce risque, nous avons des responsabilités que chaque membre de l'équipe comprend: il comprend bien les parties des autres, même s'il peut ne pas comprendre le détail de ce qu'il y a faire.

Possible, inacceptable (7):

- Le document utilisateur n'est pas complet
- Le bilan sur la gestion de projet est incomplet

Ces documents sont à rendre avec la fin du projet. Comparé aux trois autres documents à rendre, il y a un risque plus important car la pression du projet reste lourde.

Possible, dévastateur (8):

- La partie sans objet ne marche pas
- La partie objet ne marche pas
- La partie langage complet ne marche pas

Notre investissement dans les tests nous a permis de minimiser le coût d'échec complet. Cependant, il reste le risque que nous n'arrivions pas à implémenter un

sous-langage. En particulier, le risque est accentué du passage de la partie sans objet à la partie objet, qui demande une implémentation en assembleur beaucoup plus importante, et une partie ARM.

Probable, tolérable (8):

- Nous passons trop de temps sur les tests

Nous avons choisi de diviser l'équipe en deux: une équipe "tests" et une équipe "implantation". Ces deux équipes travaillent largement en parallèle. Cependant, il se peut que l'équipe test se retrouve avec moins de travail que l'équipe implantation.

Le changement de contexte du test à l'implantation peut nous rendre moins efficace que si nous avions choisi de mettre plus de ressources sur l'implémentation, au détriment des tests. Cependant, nous considérons le

Probable, inacceptable (11):

- L'extension est incomplète

Actuellement, nous ne travaillons qu'à quatre dans le projet. Il est compliqué de pouvoir travailler beaucoup l'extension, sachant que nous avons un membre de moins. Malgré cela, nous visons quand même un projet complet avec l'extension finie.

Probable, dévastateur (12):

- Il y a un bug dans le CI (l'infrastructure de tests)

Ce risque est critique et assez probable. En effet, un bug dans l'infra de test pourrait être introduit qui ferait mal fonctionner le CI, ou encore pire, faire passer tous les tests alors qu'ils ne doivent pas marcher, et nous ne nous en rendons compte qu'avant la checklist avant le rendu final.

Les changements sur le CI doivent être faits avec la plus grande précaution. Nous n'avons pas trouvé de moyen de réduire ce risque pour l'instant.

Release management process

Cette partie du document décrit une checklist d'actions à prendre avant de faire une release.

1. Vérifier que le CI passe (voir le Wiki gitlab: Tests et CI)
2. Se mettre sur la branche master et faire `git pull`
3. Activer tous les tests d'intégration dans `test.py`, et ré-exécuter le CI avec tous les tests
4. Compiler le programme avec `mvn compile`
5. Vérifier que les tests passent sur les machines des dev (`mvn verify`)
6. Vérifier que la structure des répertoires de test² est bien conforme

² [Tests], 3.1 Classification des tests "deca", p 151

7. Vérifier que `common-tests.sh` passe
8. Vérifier la couverture des tests sur le CI: voir wiki GitLab