



LOG8371- Ingénierie de la qualité en logiciel

Hiver 2023

Travaille pratique #1: Plan et Test, Maintenabilité et Fiabilité

Groupe Peel

2088821 – Laurent Quoc Hoa Nguyen

1995039 - Labib Bashir-Choudhury

1956802- Dawut Esse

1954607 - Victor Kim

2085085 - Yasser Kadimi

Soumis à : Armstrong Tita Foundjem

Date de Remise : 15 février 2023

Table des matières

1. Plan d'assurance qualité	2
1.1. Introduction	2
1.2. Critères de qualité couverts	3
1.3. Stratégies de validation	5
1.3.1. Tests unitaires	5
1.3.2. Revues par les pairs	5
1.3.3. Tests en boîte noire	5
2. Stratégie de testing	6
2.1. Plan des tests	6
2.2. Description des tests	7
2.2.1. Critère fiabilité	7
2.2.2. Critère fonctionnalité	7
2.2.3. Critère maintenabilité	7
2.3. Rapport des tests	8
2.3.1. Critère fiabilité	8
2.3.1.1. Sous-critère maturité	8
2.3.1.2. Sous-critère récupérabilité	8
2.3.2. Critère fonctionnalité	8
2.3.2.1. Sous-critère complétude	8
2.3.2.2. Sous-critère exactitude	9
2.3.3. Critère maintenabilité	9
2.3.3.1. Sous-critère testabilité	9
2.3.3.2. Sous-critère modifiabilité	10
2.3.4. Conclusion	11
3. Intégration continue	12
3.1. Plan pour l'intégration continu	12
3.2. Nouvel algorithme	17
3.3. Garantir la qualité du système après les modifications	18
3.4. Mise à jour du plan de qualité	19
3.5. Vidéo sur l'intégration continue	19
4. Références	20

1. Plan d'assurance qualité

1.1. Introduction

Le logiciel Massive Online Analysis, communément appelé MOA, est un logiciel open source pour les données massives. Il inclut une collection d'algorithmes d'apprentissage comme la classification, la régression et le clustering et des outils d'évaluation.

Il est important que le logiciel de MOA soit de bonne qualité, car il est le plus populaire cadre open-source pour l'extraction de données (data mining). En effet, beaucoup de personnes l'utilisent, notamment pour des projets d'envergure ou dans le domaine médical. Sa qualité doit alors être bonne pour satisfaire ses utilisateurs. Ainsi, pour bien évaluer la qualité, il faut utiliser les critères de l'assurance qualité d'un logiciel. Ces critères sont principalement la maintenabilité, la fiabilité, la fonctionnalité, l'efficacité et la sécurité. Nous utilisons le modèle ISO/IEC 25010:2011.

- **Maintenabilité** : La maintenabilité est le degré d'efficacité avec lequel un développeur peut modifier un produit ou un système peut être modifié. Les sous-critères associés à ce critère sont l'analysabilité, la changeabilité, la stabilité et la testabilité.
- **Fiabilité** : La fiabilité est la mesure de la capacité d'un système d'opérer sans échec pour une période de temps spécifiée. Ainsi, un utilisateur doit être capable d'exécuter des fonctions d'un composant sans erreur pendant une période donnée. Les sous-critères sont la maturité, la tolérance aux pannes et la récupérabilité.
- **Fonctionnalité** : Le critère fonctionnalité ou la pertinence fonctionnelle est la mesure dans laquelle les fonctionnalités d'un logiciel répondant aux besoins. Les sous-critères sont la complétude et l'exactitude.
- **Efficacité** : L'efficacité est la mesure de la bonne utilisation des ressources. Les sous-critères l'utilisation du temps et l'utilisation des ressources.
- **Sécurité** : La sécurité est la mesure de protection d'un utilisateur. Les sous-critères sont l'intégrité, la confidentialité, la responsabilité, l'authenticité et la non-répudiation.

Les critères pour la qualité sont pris du modèle ISO/IEC 25010:2011. Pour le cadre de ce TP, seuls les critères maintenabilité, fonctionnalité et fiabilité sont considérés ainsi que deux sous-critères pour chacun. Les sous-critères évalueront des algorithmes qui appartiennent au module « Classifiers ». Les algorithmes mis à l'étude sont les suivants: Naive Bayes Theorem, Drift, Lazy and Hoeffding Trees.

Les parties prenantes principales sont les développeurs de MOA qui sont souvent des contributeurs, car le projet est open-source. Les développeurs sont souvent des chercheurs, des enseignants et des étudiants.

1.2. Critères de qualité couverts

Critère	Sous-critère	Objectif	Mesures de validation
Fiabilité	<u>Maturité :</u> La capacité d'un système de pouvoir fonctionner même en cas d'une défaillance, ou de perte de connexion à Internet	Optimiser le processus de développement (taux de défaillances [ISO/IEC FDIS 25023:2015-12 RMa-3-G])	$P = X/Y$ X : Nombre de défaillances détectées pendant une semaine Y : Une semaine d'observation
	<u>Récupérabilité :</u> La vitesse dans laquelle une application peut récupérer ces données et revenir à l'état normal après une panne ou interruption	Diminuer le temps moyen nécessaire pour que le service soit disponible après une panne (temps moyen de récupération [ISO/IEC FDIS 25023:2015-12 RRe-1-G])	$P = \frac{\sum_{i=1}^n X_i}{n}$ Xi : Temps total de récupération par le système pour chaque défaillance nécessitant une récupération Y : Nombre de défaillances i nécessitant une récupération
Fonctionnalité	<u>Complétude :</u> Le degré dans lequel les exigences des utilisateurs sont satisfaites par les fonctionnalités implémentées	De donner à l'utilisateur la possibilité de faire toutes les tâches qu'il a précisées avant la conception de l'application (couverture de l'implémentation fonctionnelle [ISO/IEC FDIS	$P = 1-X/Y$ X : Nombre de fonctions manquantes ou non exécutables Y : Nombre de fonctions spécifiées dans le présent document, les spécifications de

		25023:2015-12 FCp-1-G))	conception ou les guides utilisateurs.
	<u>Exactitude :</u> Le degré dans lequel les valeurs générées par les fonctions de l'application sont égales aux valeurs attendues	S'assurer que les valeurs de l'algorithme sont les mêmes que les résultats attendus (exactitude fonctionnelle [ISO/IEC FDIS 25023:2015-12 FCr-1-G])	$P = 1 - X/Y$ X : Nombre de fonctions non conformes aux exigences du présent document Y : Nombre de fonctions couvertes par des exigences du présent document, excluant les exigences de priorité souhaitable
Maintenabilité	<u>Testabilité:</u> Le degré dans lequel une couverture de tests permet de prévenir toutes les défaillances	S'assurer que les tests unitaires couvrent suffisamment les fonctionnalités (complétude des fonctions de test intégrées (tests automatiques) [ISO/IEC FDIS 25023:2015-12 MTe-1-G])	$P = X/Y$ X : Nombre de fonctions de test intégrées (tests automatiques) requises, selon les exigences du présent document, et qui ont été mises en œuvre Y : Nombre de fonctions de test intégrées requises
	<u>Modifiabilité :</u> La mesure de la performance d'une application après avoir ajouté une nouvelle fonctionnalité. Les autres fonctionnalités ne sont pas affectées par cette modification	Il ne doit y avoir aucune défaillance du système quand il y a une mise à jour du système (capacité de modification ([ISO/IEC FDIS 25023:2015-12 MMd-3-S])	$P = X/Y$ X : Nombre d'éléments modifiés pendant une semaine Y : Nombre d'éléments qui doivent être modifiés pendant une semaine

1.3. Stratégies de validation

En considérant que l'application est open-source, nous pouvons considérer seulement les stratégies revues personnelles et revues par les pairs. Les stratégies de revues de conceptions formelles et audit sont formelles et s'appliquent souvent en entreprise et non dans un projet open-source où il peut avoir de la contribution des développeurs anonymes. D'autres stratégies de validation peuvent être utilisées, telles que les tests unitaires et les tests en boîte noire.

1.3.1. Tests unitaires

Chaque développeur a la responsabilité de faire des tests unitaires après avoir complété une nouvelle fonctionnalité pour vérifier que le code ajouté fait bien ce qui est demandé dans les exigences et valider que le code fonctionne correctement.

1.3.2. Revues par les pairs

Chaque fois qu'un développeur veut intégrer du nouveau code, il doit en premier permettre à un autre développeur de vérifier et de confirmer que le code est d'une bonne qualité. Cette revue doit être documentée, avec toutes les défaillances et tous les bogues présents ou potentiels. Tout cela pourra être documenté durant un pull request. Si les pairs pensent que le code peut être bien intégré dans le projet, celui qui occupe la position de reviewer peut accepter le pull-request.

1.3.3. Tests en boîte noire

Après chaque nouvelle fonctionnalité ajoutée. Il faut s'assurer que la nouvelle fonctionnalité fonctionne dans les mains d'un utilisateur. On teste le système avec des entrées données et on compare les résultats avec les sorties attendues. Le testeur doit tester toutes les différentes possibilités en utilisant des classes d'équivalences pour tester tous les extrêmes. Si le nouveau code n'est pas une fonctionnalité, il faut tester les fonctionnalités qui sont affectées.

2. Stratégie de testing

2.1. Plan des tests

Critère	Sous-critère	Objectifs	Test ou diagnostique	Algorithme(s) testé(s)
Fiabilité	Maturité	L'indicateur de maturité P doit être égal à 0.10 ($[P = X/Y] = 0.10$)	Tests unitaires	Bayes et Drift
	Récupérabilité	L'indicateur de récupérabilité P doit être au maximum égal à 1 heure ($[P = X/Y] = 1$)	Tests d'échec et de récupération	Tous
Fonctionnalité	Complétude	L'indicateur de complétude P doit être au minimum égal à 0.85 ($[P = X/Y] = 0.85$)	Revue	Bayes
	Exactitude	L'indicateur d'exactitude P doit être égal à 0.99 ($[P = X/Y] = 0.99$)	Tests en boîte noire	Drift
Maintenabilité	Testabilité	Il faut avoir une couverture de code de 90% pour les tests unitaires	Tests unitaires	Lazy
	Modifiabilité	L'indicateur de modifiabilité P doit être égal ou inférieur à 1 ($[P = X/Y] \leq 1$)	Tests de régression	Hoeffding Trees

2.2. Description des tests

2.2.1. Critère fiabilité

Dans l'objectif de réduire les risques associés aux pannes ou aux défaillances afin de permettre une meilleure fiabilité du système, les deux sous-critères choisis sont la maturité et la récupérabilité. Pour le sous-critère de maturité, l'outil JUnit sera utilisé. Les tests unitaires sont un type de tests efficace qui permet de révéler les défauts dans le code source. Ainsi, la stratégie de tests pour ce sous-critère consiste à mesurer le nombre de défauts, grâce aux tests unitaires, sur une semaine d'observation du système pour ensuite les corriger dans le but d'avoir le maximum de tests réussis.

Pour le sous-critère de récupérabilité, des tests d'échec et de récupération seront faits. La stratégie adoptée sera la simulation d'une panne ou d'une défaillance. Par exemple, nous pouvons débrancher le câble d'alimentation de la machine qui est en train de rouler MOA, puis rebrancher le câble et ensuite redémarrer la machine.

2.2.2. Critère fonctionnalité

Afin de tester que le système répond aux besoins et aux exigences, les sous-critères de complétude et d'exactitude seront à l'étude. Pour le sous-critère de complétude, une revue par les pairs sera faite. Même si la revue par les pairs est une stratégie de validation moins formelle, elle permet toutefois de détecter des erreurs de conception.

Pour le sous-critère d'exactitude, des tests en boîte noire seront faits sur le système. Des cas de tests seront préalablement montés à partir des classes d'équivalence et serviront de paramètres aux tests. De cette manière, nous pourrions comparer les résultats générés avec les résultats attendus et en tirer des conclusions quant à l'exactitude du système. Il est intéressant de noter que les tests en boîte noire pourront aussi servir de tests pour le sous-critère de complétude, car ils permettent de détecter des fonctionnalités manquantes.

2.2.3. Critère maintenabilité

Les sous-critères choisis pour couvrir le critère de maintenabilité sont la testabilité et la modifiabilité du système. Pour la testabilité, la stratégie adoptée est les tests unitaires. Dans le but de nous assurer que le système soit testé, des tests unitaires doivent être implémentés par les développeurs lorsqu'ils implémentent de nouvelles fonctionnalités. De cette manière, nous nous assurons que les fonctionnalités seront minimalement testées et couvertes par les tests unitaires.

Pour la modifiabilité, les tests de régression seront mis à profit. Cela peut se faire de deux manières. La première façon serait d'utiliser un outil d'intégration continue tel que Jenkins. Avec des « pipelines » que Jenkins permet de configurer, les développeurs pourront développer des tests de régression qui seront automatisés. Cela vérifie donc que les nouvelles fonctionnalités

implémentées n'altèrent pas les autres fonctionnalités déjà présentes. La deuxième façon serait d'utiliser les tests de régressions proposés par les développeurs de MOA. En effet, la plupart des fonctionnalités du système viennent avec des tests de régression.

2.3. Rapport des tests

2.3.1. Critère fiabilité

2.3.1.1. Sous-critère maturité

Figure 1 : Résultats des tests unitaires de l'algorithme Bayes

✓ bayes (moa.classifiers)	907 ms
✓ NaiveBayesMultinomialTest	554 ms
✓ testRegression	540 ms
✓ testSerializable	14 ms
✓ NaiveBayesTest	353 ms
✓ testRegression	341 ms
✓ testSerializable	12 ms

Figure 2 : Résultats des tests unitaires de l'algorithme Drift

✓ drift (moa.classifiers)	1sec 64 ms
✓ DriftDetectionMethodClassifierTest	609 ms
✓ testRegression	595 ms
✓ testSerializable	14 ms
✓ SingleClassifierDriftTest	455 ms
✓ testRegression	440 ms
✓ testSerializable	15 ms

Pour mesurer la maturité du système, des tests unitaires ont été utilisés. Ces tests permettent de détecter rapidement les défaillances du système. Ici, les développeurs de MOA ont pris soin d'implémenter des tests unitaires pour les algorithmes de Bayes et Drift. Comme nous pouvons le voir dans les captures d'écran ci-haut, des tests nommés « testSerializable » ont été exécutés à l'aide du cadriciel JUnit. Les résultats concluants permettent de croire à première vue que ces algorithmes ne révèlent aucune défaillance, car l'indicateur $P = X/Y = 0$.

2.3.1.2. Sous-critère récupérabilité

Le système n'offre aucun test afin de vérifier et de valider le sous-critère de récupérabilité. Nous proposons alors de faire des tests d'échec et de récupération. Pour y arriver, les testeurs devront manuellement simuler une panne ou interruption logicielle. Ils peuvent par exemple débrancher le câble d'alimentation de la machine sur laquelle le système à l'étude est en train de rouler. Les testeurs devront au préalable noter l'état du système avant l'interruption. Une fois débranché, le câble devra ensuite être rebranché. Les testeurs devront enfin rallumer la machine, rouvrir le système et noter l'état. Une comparaison pourra être faite entre les données d'avant et après l'interruption. De cette manière, les testeurs pourront tester le temps moyen de récupération du système suite à une panne ou interruption et la comparer au seuil défini dans le plan de test.

2.3.2. Critère fonctionnalité

2.3.2.1. Sous-critère complétude

Une façon de tester la complétude du système serait de faire des revues par les pairs. Pour y arriver, les développeurs peuvent faire des « pull requests ». Les pull requests permettent aux contributeurs du projet de vérifier et valider que le code à être fusionné avec la version principale

est de bonne qualité, qu'il passe les tests et qu'il n'a pas d'erreur. Ainsi, les testeurs et développeurs seront en mesure de voir si le système manque ou non des fonctionnalités.

2.3.2.2. Sous-critère exactitude

Une façon de tester l'exactitude du système serait de faire des tests en boîte noire. Cela peut se faire en implémentant des tests unitaires qui ne font que vérifier que les fonctionnalités retournent bien le résultat attendu selon les paramètres en entrée. Les développeurs devront au préalable rédiger des classes d'équivalence et des cas de tests afin de pouvoir comparer les paramètres en entrée et les sorties attendues.

2.3.3. Critère maintenabilité

2.3.3.1. Sous-critère testabilité

Figure 3 : Résultats des tests unitaires de l'algorithme Lazy

```
> • No tests were found /Users/laurentnguyen/Library/Java/JavaVirtualMachines/openjdk-19.0.2/Contents/Home/bin/java ...
0 test classes found in package 'moa.classifiers.lazy'

Process finished with exit code 254
```

Figure 4 : Résultats de la couverture des tests unitaires pour tous les algorithmes

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	42,4% (53/125)	26,5% (295/1115)	22,4% (1564/6983)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
moa.classifiers.bayes	100% (2/2)	66,7% (16/24)	55,9% (80/143)
moa.classifiers.drift	100% (2/2)	58,3% (7/12)	72,9% (43/59)
moa.classifiers.lazy	0% (0/4)	0% (0/54)	0% (0/404)
moa.classifiers.lazy.neighboursearch	0% (0/9)	0% (0/140)	0% (0/727)
moa.classifiers.lazy.neighboursearch.kdtrees	0% (0/6)	0% (0/39)	0% (0/288)
moa.classifiers.trees	70% (49/70)	50,8% (272/535)	48,4% (1441/2978)
moa.classifiers.trees.iadem	0% (0/32)	0% (0/311)	0% (0/2384)

Nous avons expliqué plus haut que les tests unitaires efficaces pour prévenir les défaillances. Or, nous pouvons constater avec la figure 3 intitulée « Résultats des tests unitaires de l'algorithme Lazy » que des tests unitaires n'ont pas été conçus pour l'algorithme Lazy du module Classifiers. Avec l'absence de tests unitaires, il devient très difficile de corriger les erreurs, ce qui augmente le nombre de défaillances.

La capture d'écran intitulée « Résultats de la couverture des tests unitaires pour tous les algorithmes » permet de voir le pourcentage de lignes que les tests unitaires implémentés ont couvertes. Nous observons rapidement qu'aucun algorithme n'est couvert à 90%. Nous observons également que beaucoup de méthodes de classe ne sont pas testées. Cela est donc

problématique au point de vue de la testabilité du système, car des bogues peuvent exister sans jamais être détectés. Pour remédier à la situation, il faudrait que les développeurs implémentent les tests unitaires manquants pour tester les algorithmes.

2.3.3.2. Sous-critère modifiabilité

Figure 5 : Résultats des tests unitaires de l'algorithme Hoeffding Trees

✖ trees (moa.classifiers)	6 sec 665 ms
✖ AdaHoeffdingOptionTreeTest	1 sec 56 ms
✖ testRegression	1 sec 43 ms
✓ testSerializable	13 ms
✓ ASHoeffdingTreeTest	740 ms
✓ testRegression	728 ms
✓ testSerializable	12 ms
✓ DecisionStumpTest	243 ms
✓ testRegression	231 ms
✓ testSerializable	12 ms
✓ EFDTest	1 sec 132 ms
✓ testRegression	1 sec 120 ms
✓ testSerializable	12 ms
✓ FIMTDDTest	56 ms
✓ testRegression	55 ms
✓ testSerializable	1 ms
✓ HoeffdingAdaptiveTreeTest	1 sec 49 ms
✓ testRegression	1 sec 37 ms
✓ testSerializable	12 ms
✓ HoeffdingOptionTreeTest	892 ms
✓ testRegression	881 ms
✓ testSerializable	11 ms
✓ HoeffdingTreeTest	1 sec 13 ms
✓ testRegression	994 ms
✓ testSerializable	19 ms
✓ LimAttHoeffdingTreeTest	199 ms
✓ testRegression	188 ms
✓ testSerializable	11 ms
> ORTOTest	33 ms
✓ RandomHoeffdingTreeTest	252 ms
✓ testRegression	241 ms
✓ testSerializable	11 ms

Pour tester la modifiabilité du système, une des façons était de faire des tests de régression. Dans ce cas d'étude, les tests ont été réalisés sur l'algorithme Hoeffding Trees. Nous pouvons observer la figure 5 intitulée « Résultats des tests unitaires de l'algorithme Hoeffding Trees » que des tests de régression (« testRegressions ») ont été faits. Nous pouvons vite remarquer qu'un test de régression n'a pas passé sur cette version de l'algorithme. Ainsi, cela signifie que les modifications apportées à cette version ont engendré des erreurs sur la dernière version du système fonctionnel en production.

Pour s'assurer que ce genre d'erreur puisse être détecté avant la mise en production, les développeurs pourraient mettre en place un outil d'intégration continue. Cela peut être fait à l'aide de l'outil open source Jenkins.

2.3.4. Conclusion

Objectifs/tests associés	Résultats des cas de tests correspondants	Contre-mesure
Maturité (Bayes et Drift)	Succès	--
Récupérabilité (Tous)	Non implémenté	Simulation de pannes ou interruptions
Complétude (Bayes)	Non implémenté	Revue par les pairs
Exactitude (Drift)	Non implémenté	Rédiger des cas de tests et implémenter les tests unitaires correspondants
Testabilité (Lazy)	Échec	Implémentation des tests manquants
Modifiabilité (Hoeffding Trees)	Échec	CI avec Jenkins

3. Intégration continue

3.1. Plan pour l'intégration continue

Notre équipe a décidé d'opter pour l'outil d'intégration Jenkins, car il s'agit d'un serveur d'intégration très fiable qui automatise l'ensemble du processus de développement de logiciels. Jenkins nous fournit une plateforme robuste et évolutive qui peut être facilement personnalisée pour automatiser notre plan de test. Dans notre cas, le « pipeline » d'automatisation sera lancé à chaque fois qu'il y a un « push » sur la branche master.

Pour exécuter Jenkins, nous lançons la commande montrée à la figure 6 dans un terminal.

Figure 6 : Commande pour exécuter Jenkins

```
PS C:\Program Files\Jenkins> java -jar jenkins.war
```

Jenkins installe ensuite les différents plugins nécessaires au projet.

Figure 7 : Installation des différents plugins(1)

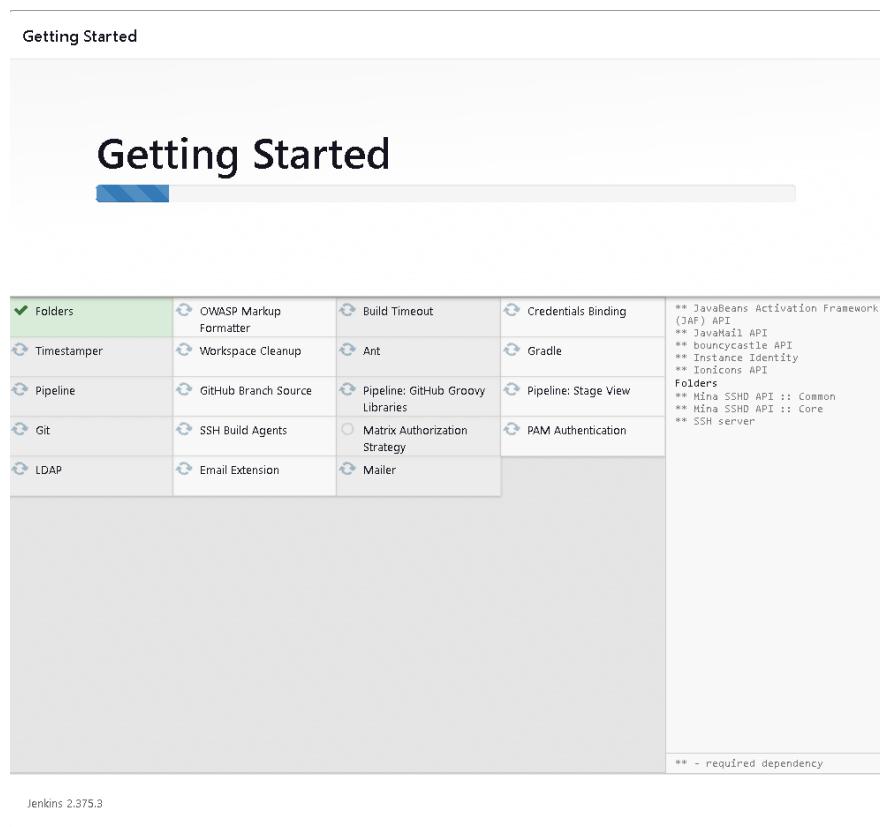


Figure 8 : Installation des différents plugins(2)

Start building your software project

Create a job



Enter an item name

LOG8371_TP1

» Required field



Freestyle project

This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.



Pipeline

Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.



Multi-configuration project

Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.



Folder

Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.



Multibranch Pipeline

Creates a set of Pipeline projects according to detected branches in one SCM repository.



Organization Folder

Creates a set of multibranch project subfolders by scanning for repositories.

La figure 9 montre la configuration git du projet sur lequel le pipeline va rouler.

Figure 9 : Configuration git du projet Jenkins

Git ?

Repositories ?

Repository URL ? ✕

https://github.com/yasskadd/moa.git

Credentials ?

yasskadd/*****

+ Add

Advanced...

Add Repository

Branches to build ?

Branch Specifier (blank for 'any') ? ✕

*/master

Add Branch

Repository browser ?

(Auto)


Nous avons configuré Jenkins pour faire en sorte que chaque push fait sur la branche master du repo git déclenche le build et les tests.

Figure 10 : Hook pour lancer le build et les tests lors d'un push sur la branche master

☒ GitHub hook trigger for GITScm polling ?

Nous avons configuré le build à ce que les tests unitaires soient compilés et exécutés par Maven.

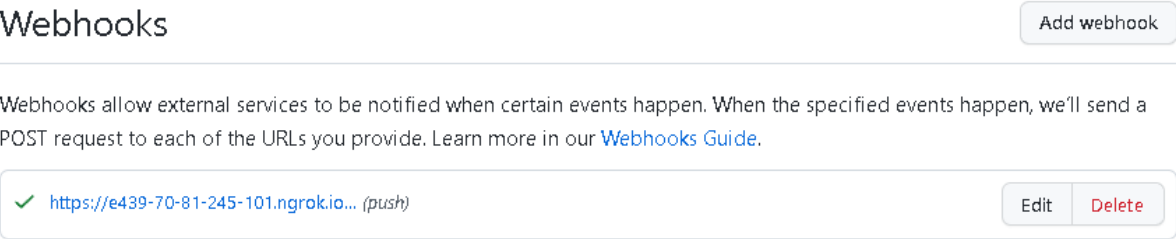
Figure 11 : Configuration Maven pour compiler et exécuter les tests



The screenshot shows the 'Build Steps' section in Jenkins. A step titled 'Invoke top-level Maven targets' is selected. Below the title, there is a 'Maven Version' dropdown menu set to 'Maven_TP1'. Under the 'Goals' section, the text 'clean compile test' is entered in a text field, with a dropdown arrow on the right. Below the goals field is an 'Advanced...' button with a gear icon. At the bottom of the step configuration area is an 'Add build step' button.

Nous avons configuré un « webhook » au repo git du projet vers notre instance de Jenkins afin qu'il puisse envoyer une notification à chaque push fait.

Figure 12 : Webhook github, utilisation du service ngrok pour permettre au port local 8080 d'être exposé à Internet



The screenshot shows the 'Webhooks' section in Jenkins. At the top right is an 'Add webhook' button. Below the section title, there is explanatory text: 'Webhooks allow external services to be notified when certain events happen. When the specified events happen, we'll send a POST request to each of the URLs you provide. Learn more in our [Webhooks Guide](#).' Below this text is a list of configured webhooks. One webhook is shown with a green checkmark, the URL 'https://e439-70-81-245-101.ngrok.io...', and the event '(push)'. To the right of the URL are 'Edit' and 'Delete' buttons.

Figure 13 : Résultat des tests et du build sur Jenkins

```
Tests run: 155, Failures: 2, Errors: 0, Skipped: 0

[INFO] -----
[INFO] BUILD FAILURE
[INFO] -----
[INFO] Total time: 10:52 min
[INFO] Finished at: 2023-02-15T07:22:19-05:00
[INFO] -----
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:2.15:test (default-test) on project moa: There are test failures.
[ERROR]
[ERROR] Please refer to C:\Users\yasse\Documents\GitHub\moa\moa\target\surefire-reports for the individual test results.
[ERROR] -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoFailureException
Build step 'Invoke top-level Maven targets' marked build as failure
Finished: FAILURE
```

Figure 14 : Résultats des tests et du build après le push des tests de l'algorithme

```
Tests run: 160, Failures: 2, Errors: 0, Skipped: 0

[INFO] -----
[INFO] BUILD FAILURE
[INFO] -----
[INFO] Total time: 10:30 min
[INFO] Finished at: 2023-02-15T08:28:13-05:00
[INFO] -----
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-surefire-plugin:2.15:test (default-test) on project moa: There are test failures.
[ERROR]
[ERROR] Please refer to C:\Users\yasse\Documents\GitHub\moa\moa\target\surefire-reports for the individual test results.
[ERROR] -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [Help 1] http://cwiki.apache.org/confluence/display/MAVEN/MojoFailureException
Build step 'Invoke top-level Maven targets' marked build as failure
Finished: FAILURE
```

Les figures 13 et 14 montrent l'exécution du build et les résultats des tests qui s'ensuivent. Nous observons que le build a échoué (« build failure »), ce qui est normal. En effet, nous avons vu dans la section « Rapport des tests » que certains tests unitaires échouent, notamment des tests de régression.

3.2. Nouvel algorithme

Figure 15 : Nouvelle classe « newAlgorithme »

```
package main.java.moa.classifiers;

public class newAlgorithme {
    private int a;
    private int b;
    public newAlgorithme(int a, int b){
        this.a = a;
        this.b = b;
    }
    public int sum() {
        return a + b;
    }
    public int subtraction() {
        return a - b;
    }
    public int multiplication(){
        return a*b;
    }
    public int division(){
        return a/b;
    }
    public void setA(int number){
        a = number;
    }
    public void setB(int number){
        b = number;
    }
    public int getA(){
        return this.a;
    }
    public int getB(){
        return this.b;
    }
}
```

Nous avons ajouté une classe nommée « newAlgorithme ». Cette classe est utilisée pour faire des opérations arithmétiques sur des nombres entiers. Cela a été ajouté pour tester l'impact d'une nouvelle classe sur le plan d'assurance qualité.

3.3. Garantir la qualité du système après les modifications

Figure 16 : Tests unitaires ajoutés pour tester la nouvelle classe

```
10 public class newAlgorithmTest {
11     static newAlgorithm algorithm;
12     @BeforeClass
13     public static void setUpBeforeClass() throws Exception {
14         algorithm = new newAlgorithm(a: 2, b: 3);
15     }
16
17     @Test
18     public void testSumFunction() {
19
20         assertEquals(expected: 5, algorithm.sum());
21     }
22     @Test
23     public void testSetAndGet(){
24         algorithm.setA(number: 100);
25         assertEquals(expected: 100, algorithm.getA());
26     }
27     @Test
28     public void testMultiplication(){
29         algorithm.setB(number: 75);
30         algorithm.setA(number: 2);
31         assertEquals(expected: 150, algorithm.multiplication());
32     }
33     @Test
34     public void testDivision(){
35         algorithm.setA(number: 90);
36         algorithm.setB(number: 30);
37         assertEquals(expected: 3, algorithm.division());
38     }
39     @Test
40     public void testSubstraction(){
41         algorithm.setA( number: 20);
42         algorithm.setB(number: 10);
43         assertEquals(expected: 10, algorithm.subtraction());
44     }
45 }
```

Afin de nous assurer de la qualité du système, nous avons ajouté des tests unitaires pour la classe nouvellement ajoutée. La figure 16 montre les tests unitaires que nous avons ajoutés.

3.4. Mise à jour du plan de qualité

Notre plan de qualité couvre assez bien en termes d'assurance qualité et de requis, c'est pour cette raison qu'on le met pas à jour. Comme ce n'est pas une nouvelle fonctionnalité importante pour le système au complet, cela ne nécessite pas des mises à jour au plan du qualité logicielle et les objectifs de qualité demeurent les mêmes.

3.5. Vidéo sur l'intégration continue

Voici le lien d'une vidéo montrant l'intégration continue dans le système :
<https://youtu.be/zTFeB7aILu0>

4. Références

1. ISO 25000 Portal. (s.d.). ISO/IEC 25010.
<https://iso25000.com/index.php/en/iso-25000-standards/iso-25010>
2. Hydro-Québec. (s.d.). Normalisation des exigences.
<https://docs.google.com/spreadsheets/d/0B4cNbsMq3e3bWklBNEtPTkxMMUE/edit?resourcekey=0-drvZT7ONe9z7JlvY6EaV4Q#gid=998574238>