

Conduite de projet : La qualité du logiciel

Aldric Degorre

(IRIF, U-Paris) – adegorre@irif.fr

(transparentes originaux de Yann Régis-Gianas et contributions de Mo Foughali)

13 octobre 2023

Plan

Retour sur les deux semaines précédentes

La qualité logicielle

Le développement dirigé par les tests

Intégration continue

En cours

- ▶ Scrum
- ▶ git

En travaux dirigés

- ▶ Bilan : de 0 à 21 tickets fermés.
- ▶ Souvent moins de 5... (dans la plupart des équipes, au moins un étudiant ne travaille pas!)

En travaux dirigés

- ▶ Bilan : de 0 à 21 tickets fermés.
- ▶ Souvent moins de 5... (dans la plupart des équipes, au moins un étudiant ne travaille pas!)
- ▶ Groupes problématiques :
 - ▶ Cas 1 : Aucun ticket, aucune PR, rien! (Au bout de 2 sprints...)
 - ▶ Cas 2 : Des discussions, des choses faites MAIS rien sur Gitlab!
 - ▶ Cas 3 : Seule une minorité des membres travaille.

En travaux dirigés

- ▶ Bilan : de 0 à 21 tickets fermés.
- ▶ Souvent moins de 5... (dans la plupart des équipes, au moins un étudiant ne travaille pas!)
- ▶ Groupes problématiques :
 - ▶ Cas 1 : Aucun ticket, aucune PR, rien! (Au bout de 2 sprints...)
 - ▶ Cas 2 : Des discussions, des choses faites MAIS rien sur Gitlab!
 - ▶ Cas 3 : Seule une minorité des membres travaille.
- ▶ Membres problématiques :
 - ▶ Cas 1 : Absent. (Cas facile.)
 - ▶ Cas 2 : Figuration. (Cas facile, aussi.)
 - ▶ Cas 3 : Des tâches commencées et jamais finies. (Ca ne peut plus durer!)
 - ▶ Cas 4 : Travaille mais ne collabore pas.

En travaux dirigés

- ▶ Bilan : de 0 à 21 tickets fermés.
- ▶ Souvent moins de 5... (dans la plupart des équipes, au moins un étudiant ne travaille pas!)
- ▶ Groupes problématiques :
 - ▶ Cas 1 : Aucun ticket, aucune PR, rien! (Au bout de 2 sprints...)
 - ▶ Cas 2 : Des discussions, des choses faites MAIS rien sur Gitlab!
 - ▶ Cas 3 : Seule une minorité des membres travaille.
- ▶ Membres problématiques :
 - ▶ Cas 1 : Absent. (Cas facile.)
 - ▶ Cas 2 : Figuration. (Cas facile, aussi.)
 - ▶ Cas 3 : Des tâches commencées et jamais finies. (Ca ne peut plus durer!)
 - ▶ Cas 4 : Travaille mais ne collabore pas.
- ▶ Mais comment font les groupes qui réussissent?

En travaux dirigés

- ▶ Bilan : de 0 à 21 tickets fermés.
- ▶ Souvent moins de 5... (dans la plupart des équipes, au moins un étudiant ne travaille pas!)
- ▶ Groupes problématiques :
 - ▶ Cas 1 : Aucun ticket, aucune PR, rien! (Au bout de 2 sprints...)
 - ▶ Cas 2 : Des discussions, des choses faites MAIS rien sur Gitlab!
 - ▶ Cas 3 : Seule une minorité des membres travaille.
- ▶ Membres problématiques :
 - ▶ Cas 1 : Absent. (Cas facile.)
 - ▶ Cas 2 : Figuration. (Cas facile, aussi.)
 - ▶ Cas 3 : Des tâches commencées et jamais finies. (Ca ne peut plus durer!)
 - ▶ Cas 4 : Travaille mais ne collabore pas.
- ▶ Mais comment font les groupes qui réussissent?
 - ▶ Première raison : ils travaillent **et** jouent le jeu!

En travaux dirigés

- ▶ Bilan : de 0 à 21 tickets fermés.
- ▶ Souvent moins de 5... (dans la plupart des équipes, au moins un étudiant ne travaille pas!)
- ▶ Groupes problématiques :
 - ▶ Cas 1 : Aucun ticket, aucune PR, rien! (Au bout de 2 sprints...)
 - ▶ Cas 2 : Des discussions, des choses faites MAIS rien sur Gitlab!
 - ▶ Cas 3 : Seule une minorité des membres travaille.
- ▶ Membres problématiques :
 - ▶ Cas 1 : Absent. (Cas facile.)
 - ▶ Cas 2 : Figuration. (Cas facile, aussi.)
 - ▶ Cas 3 : Des tâches commencées et jamais finies. (Ca ne peut plus durer!)
 - ▶ Cas 4 : Travaille mais ne collabore pas.
- ▶ Mais comment font les groupes qui réussissent?
 - ▶ Première raison : ils travaillent **et** jouent le jeu!
 - ▶ Une affectation claire des tâches et de leurs validations.
 - ▶ Utilisation de gitlab et git. (Rien en dehors!)
 - ▶ Un correspondant "communication" avec l'enseignant.

Séance de cours d'aujourd'hui

1. Qu'est-ce qu'un logiciel de qualité?
2. Comment tester un logiciel?
3. L'intégration et le déploiement continus.

Plan

Retour sur les deux semaines précédentes

La qualité logicielle

Le développement dirigé par les tests

Intégration continue

Selon vous, quels sont les critères qui font
qu'un logiciel est de bonne qualité?

La norme ISO/CEI 9126 fixe les critères suivants :

- ▶ La capacité fonctionnelle :
le logiciel répond aux exigences de fonctionnalités.
- ▶ La fiabilité :
le logiciel fonctionne sous les conditions d'usage normales établies.
- ▶ L'utilisabilité :
le logiciel est adapté à ses utilisateurs.
- ▶ L'efficacité :
le logiciel utilise raisonnablement les ressources allouées à son fonctionnement.
- ▶ La maintenabilité :
le logiciel peut évoluer.
- ▶ La portabilité :
le logiciel peut être transféré d'un environnement à un autre.

Qualité de la version initiale de `pacman`

Comment évaluez-vous la version initiale du projet vis-à-vis de ces critères ?

- ▶ La capacité fonctionnelle :
- ▶ La fiabilité :
- ▶ L'utilisabilité :
- ▶ L'efficacité :
- ▶ La maintenabilité :
- ▶ La portabilité :

Qualité de la version initiale de `pacman`

Comment évaluez-vous la version initiale du projet
vis-à-vis de ces critères ?

- ▶ La capacité fonctionnelle :
Le jeu fourni permet juste de déplacer Pac-Man. Les fantômes ne bougent même pas ; on ne peut pas ramasser les Pac-Gommes ou des bonus (on est loin du jeu d'origine, sans parler d'ajouts amusants...).
- ▶ La fiabilité :
- ▶ L'utilisabilité :
- ▶ L'efficacité :
- ▶ La maintenabilité :
- ▶ La portabilité :

Qualité de la version initiale de `pacman`

Comment évaluez-vous la version initiale du projet vis-à-vis de ces critères ?

- ▶ La capacité fonctionnelle :
- ▶ La fiabilité :
Le code manque de tests. Mais assez vite on rencontre des bugs dans le déplacement de Pac-Man.
- ▶ L'utilisabilité :
- ▶ L'efficacité :
- ▶ La maintenabilité :
- ▶ La portabilité :

Qualité de la version initiale de `pacman`

Comment évaluez-vous la version initiale du projet
vis-à-vis de ces critères ?

- ▶ La capacité fonctionnelle :
- ▶ La fiabilité :
- ▶ L'utilisabilité :
Les contrôles sont simples. À voir ce que l'utilisabilité donnerait avec plus de fonctionnalités.
Pac-man mériterait un installateur digne de ce nom.
- ▶ L'efficacité :
- ▶ La maintenabilité :
- ▶ La portabilité :

Qualité de la version initiale de `pacman`

Comment évaluez-vous la version initiale du projet
vis-à-vis de ces critères ?

- ▶ La capacité fonctionnelle :
- ▶ La fiabilité :
- ▶ L'utilisabilité :
- ▶ L'efficacité :
Le jeu est tout à fait jouable sur une machine actuelle. Sans aucun doute il le sera encore avec l'IA des fantômes.
- ▶ La maintenabilité :
- ▶ La portabilité :

Qualité de la version initiale de `pacman`

Comment évaluez-vous la version initiale du projet vis-à-vis de ces critères ?

- ▶ La capacité fonctionnelle :
- ▶ La fiabilité :
- ▶ L'utilisabilité :
- ▶ L'efficacité :
- ▶ La maintenabilité :
Le code sépare vue et modèle. Cependant, il manque quelques abstractions (interfaces), côté modèle.
- ▶ La portabilité :

Qualité de la version initiale de `pacman`

Comment évaluez-vous la version initiale du projet vis-à-vis de ces critères ?

- ▶ La capacité fonctionnelle :
- ▶ La fiabilité :
- ▶ L'utilisabilité :
- ▶ L'efficacité :
- ▶ La maintenabilité :
- ▶ La portabilité :

Oui, car Java est portable et les versions des dépendances sont clairement fixées.

Les FIXMEs du code de pong

Les FIXMEs sont les “fenêtres cassées” d'un code source

⇒ Il faut les supprimer au plus vite.

```
aldric@myPC:~/Documents/Enseignement/2023-2024/Conduite de projet/pacman$ rgrep -Hn FIXME *
src/main/java/model/MazeState.java:54:      // FIXME: too many things in this method. Maybe some responsibilities can be delegated t
src/main/java/model/MazeState.java:96:      // FIXME Pac-Man rules should somehow be in Pacman class
src/main/java/model/MazeState.java:121:     // FIXME: this should be displayed in the JavaFX view, not in the console
src/main/java/model/MazeState.java:126:     // FIXME: this should be displayed in the JavaFX view, not in the console. A game over
src/main/java/model/PacMan.java:6: * Implements Pac-Man character using singleton pattern. FIXME: check whether singleton is really a
src/main/java/config/Cell.java:5:      // FIXME: all these factories are convenient, but it is not very "economic" to have so many metho
```

(Et certains TODOs sont parfois des FIXME cachés...)

```
aldric@myPC:~/Documents/Enseignement/2023-2024/Conduite de projet/pacman$ rgrep -Hn TODO *
src/main/java/model/PacMan.java:48:      // TODO handle timeout!
src/main/java/model/Ghost.java:7:      // TODO: implement a different AI for each ghost, according to the description in Wikipedia's pag
src/main/java/config/MazeConfig.java:60:    // TODO: mazes should be loaded from a text file
```

Comment améliorer la qualité d'un logiciel ?

- ▶ La capacité fonctionnelle :
- ▶ La fiabilité :
- ▶ L'utilisabilité :
- ▶ L'efficacité :
- ▶ La maintenabilité :
- ▶ La portabilité :

Comment améliorer la qualité d'un logiciel ?

- ▶ La capacité fonctionnelle :
 - ⇒ Des tests, des preuves, de la documentation.
- ▶ La fiabilité :
- ▶ L'utilisabilité :
- ▶ L'efficacité :
- ▶ La maintenabilité :
- ▶ La portabilité :

Comment améliorer la qualité d'un logiciel ?

- ▶ La capacité fonctionnelle :
- ▶ La fiabilité :
 - ⇒ Des contrôles de sécurité, utiliser des langages de programmation apportant des garanties sur l'exécution du programme, bonnes pratiques de programmation, ...
- ▶ L'utilisabilité :
- ▶ L'efficacité :
- ▶ La maintenabilité :
- ▶ La portabilité :

Comment améliorer la qualité d'un logiciel ?

- ▶ La capacité fonctionnelle :
- ▶ La fiabilité :
- ▶ L'utilisabilité :
Utiliser des bêta-testeurs.
- ▶ L'efficacité :
- ▶ La maintenabilité :
- ▶ La portabilité :

Comment améliorer la qualité d'un logiciel ?

- ▶ La capacité fonctionnelle :
- ▶ La fiabilité :
- ▶ L'utilisabilité :
- ▶ L'efficacité :
Des tests de montée charge.
- ▶ La maintenabilité :
- ▶ La portabilité :

Comment améliorer la qualité d'un logiciel ?

- ▶ La capacité fonctionnelle :
- ▶ La fiabilité :
- ▶ L'utilisabilité :
- ▶ L'efficacité :
- ▶ La maintenabilité :
Ecrire le code pour qu'il soit compréhensible. Réduire sa complexité. Intégrer une batterie de tests.
- ▶ La portabilité :

Comment améliorer la qualité d'un logiciel ?

- ▶ La capacité fonctionnelle :
- ▶ La fiabilité :
- ▶ L'utilisabilité :
- ▶ L'efficacité :
- ▶ La maintenabilité :
- ▶ La portabilité :
Utiliser des technologies portables. Bien isoler les parties du système qui sont dépendantes de l'environnement.

Plan

Retour sur les deux semaines précédentes

La qualité logicielle

Le développement dirigé par les tests

Intégration continue

Qu'est-ce qu'un test ?

Test

Un **test** est un programme qui vérifie une propriété d'un autre programme sur une entrée fixée.

- ▶ Un test n'est donc pas une preuve (mais les preuves coûtent cher).
- ▶ Un test est donc un composant à part entière du logiciel.
- ▶ On appelle **batterie de tests** l'ensemble des tests d'un logiciel.
- ▶ Un test permet d'améliorer la qualité d'un logiciel en explicitant et en exerçant des **chemins d'exécution critiques**.
- ▶ Une batterie de tests permet la **non-regression** de la qualité du logiciel.
- ▶ Un test peut aussi servir à **reproduire un bug**. \Rightarrow Quand un utilisateur rapporte un bug, il faut d'abord essayer de le reproduire sous la forme d'un test.
- ▶ Pour résumer, un test est une **connaissance exécutable sur le logiciel**.

Classification des tests

Classification des tests

- ▶ **test unitaire** : vise à exercer le fonctionnement d'une unité logique du logiciel. Au moins un par fonction ou méthode.
- ▶ **test d'intégration** : vise à exercer la bonne interaction entre les composants du logiciel.
- ▶ **test système** : vise à exercer le fonctionnement du logiciel dans son ensemble. On peut vérifier beaucoup de choses : fonctionnalités, performances, gestion des exceptions, montée en charge, utilisabilité, accessibilité ...
- ▶ **test d'acceptation** (ou recette) : vise à assurer que le déploiement chez le client se passera bien. On distingue les tests d'acceptation "en usine" et ceux fait chez l'utilisateur.

Classification des tests

Classification des tests

- ▶ **test unitaire** : vise à exercer le fonctionnement d'une unité logique du logiciel. Au moins un par fonction ou méthode.
Exemple : Dans votre projet, créer une classe de test **MazeStateTest** dans `src/test/java/model` dont les méthodes testent celles de la classe **MazeState**.
- ▶ **test d'intégration** : vise à exercer la bonne interaction entre les composants du logiciel.
- ▶ **test système** : vise à exercer le fonctionnement du logiciel dans son ensemble. On peut vérifier beaucoup de choses : fonctionnalités, performances, gestion des exceptions, montée en charge, utilisabilité, accessibilité ...
- ▶ **test d'acceptation** (ou recette) : vise à assurer que le déploiement chez le client se passera bien. On distingue les tests d'acceptation "en usine" et ceux fait chez l'utilisateur.

Classification des tests

Classification des tests

- ▶ **test unitaire** : vise à exercer le fonctionnement d'une unité logique du logiciel. Au moins un par fonction ou méthode.
- ▶ **test d'intégration** : vise à exercer la bonne interaction entre les composants du logiciel.
Exemple : Dans votre projet, la vérification du bon fonctionnement entre les classes du package `gui` et celles du package `model`, dont il dépend.
- ▶ **test système** : vise à exercer le fonctionnement du logiciel dans son ensemble. On peut vérifier beaucoup de choses : fonctionnalités, performances, gestion des exceptions, montée en charge, utilisabilité, accessibilité ...
- ▶ **test d'acceptation** (ou recette) : vise à assurer que le déploiement chez le client se passera bien. On distingue les tests d'acceptation "en usine" et ceux fait chez l'utilisateur.

Classification des tests

Classification des tests

- ▶ **test unitaire** : vise à exercer le fonctionnement d'une unité logique du logiciel. Au moins un par fonction ou méthode.
- ▶ **test d'intégration** : vise à exercer la bonne interaction entre les composants du logiciel.
- ▶ **test système** : vise à exercer le fonctionnement du logiciel dans son ensemble. On peut vérifier beaucoup de choses : fonctionnalités, performances, gestion des exceptions, montée en charge, utilisabilité, accessibilité ...

Exemple : Dans votre projet, il s'agirait d'écrire un script (pas Java) qui simule des entrées utilisateur et de vérifier que graphiquement le programme se comporte comme prévu (difficile mais possible!).

- ▶ **test d'acceptation** (ou recette) : vise à assurer que le déploiement chez le client se passera bien. On distingue les tests d'acceptation "en usine" et ceux fait chez l'utilisateur.

Classification des tests

Classification des tests

- ▶ **test unitaire** : vise à exercer le fonctionnement d'une unité logique du logiciel. Au moins un par fonction ou méthode.
- ▶ **test d'intégration** : vise à exercer la bonne interaction entre les composants du logiciel.
- ▶ **test système** : vise à exercer le fonctionnement du logiciel dans son ensemble. On peut vérifier beaucoup de choses : fonctionnalités, performances, gestion des exceptions, montée en charge, utilisabilité, accessibilité ...
- ▶ **test d'acceptation** (ou recette) : vise à assurer que le déploiement chez le client se passera bien. On distingue les tests d'acceptation "en usine" et ceux fait chez l'utilisateur.
Exemple : Dans votre projet, il faut vérifier que le script de compilation génère un paquet facilement installable et exécutable sur les PC des joueurs.

Distinction “boîte noire” et “boîte blanche”

- ▶ Un test **boîte noire** ne suppose rien sur le fonctionnement interne du composant logiciel testé : il n'utilise que son interface.
- ▶ Un test **boîte blanche** s'autorise à s'appuyer sur le fonctionnement interne du composant.
- ▶ Boîte noire : robuste mais moins précis que boîte blanche.
- ▶ Boîte blanche : plus difficile à maintenir mais généralement nécessaire pour initialiser efficacement des contextes de test ou pour reproduire un bug.

Spécification

- ▶ Une **spécification** décrit les propriétés du logiciel.
- ▶ On distingue **spécification fonctionnelle** et **spécification non fonctionnelle**.
- ▶ Spécification fonctionnelle : ce que calcule le programme.
- ▶ Spécification non fonctionnelle : comment calcule le programme.
- ▶ Parmi les propriétés fonctionnelles des logiciels, on trouve :
 - ▶ les **préconditions** décrivent les hypothèses sur les entrées.
 - ▶ les **postconditions** décrivent les garanties sur les sorties.
 - ▶ les **invariants** décrivent des propriétés toujours vraies qui font que le système fonctionne bien.

Lien entre test et spécification

- ▶ Un test (fonctionnel) peut servir à valider la bonne implémentation d'une spécification sur une entrée fixée.
- ▶ Le test doit fournir des entrées validant la précondition.
- ▶ Le test doit vérifier que la sortie valide la postcondition.
- ▶ Un test en boîte blanche peut aussi vérifier que les invariants internes du composant restent valides entre chaque interaction avec ce dernier.

Les tests unitaires avec JUnit

- ▶ JUnit est un cadriciel (framework) pour écrire des tests unitaires en Java.
- ▶ Pour activer JUnit5 dans Gradle, mettre dans **build.gradle** :

```
dependencies {  
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.10.0'  
    testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.10.0'  
}
```

```
test {  
    useJUnitPlatform()  
}
```

- ▶ Documentation complète ici :

<https://junit.org/junit5/docs/current/user-guide/>

Les bases de JUnit : les annotations

- ▶ Convention de nommage : la classe de test de `A.java` est `ATest.java`.
- ▶ Les "imports":¹
`import org.junit.jupiter.api.Test;`
- ▶ Les méthodes de tests sont préfixées par l'annotation `@Test`.
- ▶ `@Timeout(time)` permet de s'assurer que le test termine.
- ▶ `@BeforeEach` indique la méthode doit être exécutée avant chaque test.
- ▶ `@AfterEach` indique la méthode doit être exécutée après chaque test.
- ▶ `@DisplayName("Description")` description du test (pour affichage dans l'IDE notamment).

`@Timeout(time)` et `@DisplayName("Description")` s'ajoutent à `@Test`.

1. À adapter si vous voulez utiliser d'autres annotations que `@Test`.

Les bases de JUnit : les assertions

Méthodes statiques de la classe `org.junit.jupiter.api.Assertions`, à insérer dans les méthodes de test :

- ▶ `fail()` fait échouer le test.
- ▶ `assertTrue(test)` fait échouer si la condition est fausse.
- ▶ `assertFalse(test)` fait échouer si la condition est vraie.
- ▶ `assertEquals(expected, actual)` fait échouer si les deux arguments ne sont pas égaux (au sens de `.equals`).
- ▶ `assertThrows(UnexpectedException.class, () -> { unAppelATester(); })` ; fait échouer si pas d'exception levée.
- ▶ Dans tous les cas, on peut ajouter un argument de type `String` pour décrire le problème.

Pour les utiliser sans le préfixe, faire des “imports statiques”. Exemple :

```
import static org.junit.jupiter.api.Assertions.assertEquals;
```

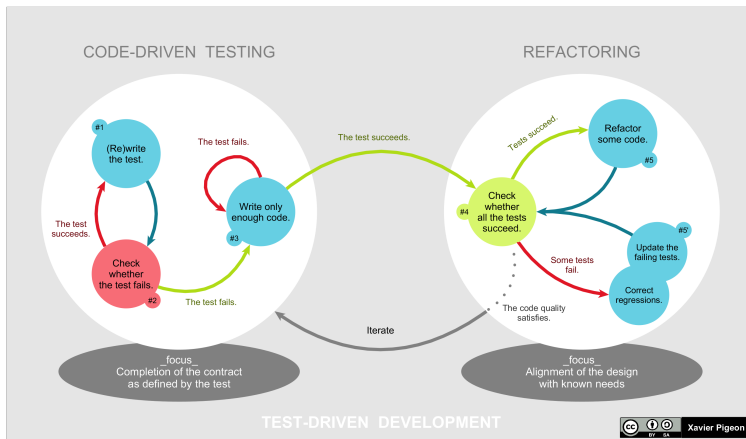
Remarque : un test qui retourne sans échouer réussit !

Comment écrire de bons tests ?

- ▶ Un test est indépendant des autres tests².
- ▶ On ne peut pas tout tester !
- ▶ Il faut traiter les cas standard mais surtout les cas limite.
- ▶ Il faut aussi traiter les cas d'erreurs.
- ▶ Il faut essayer d'écrire de petits tests.
- ▶ Les tests doivent être "simples", facile à comprendre.

2. penser à **BeforeEach** et **AfterEach** pour le code commun

La programmation dirigée par les tests



Comment écrire de bons tests ?

Choix de tests par partitionnement.

Comment écrire de bons tests ?

- ▶ Partitionner le domaine des entrées en sous-domaines.
- ▶ Choisir une entrée par sous-domaine.
- ▶ Tester sur chaque entrée choisie.

Comment écrire de bons tests?

Exemple : `BigInteger.multiply()`

```
/**  
 * @param val another BigInteger  
 * @return a BigInteger whose value is (this * val).  
 */  
public BigInteger multiply(BigInteger val)
```

...

```
BigInteger a = ...;  
BigInteger b = ...;  
BigInteger ab = a.multiply(b);
```

`multiply` : `BigInteger × BigInteger → BigInteger`

Comment écrire de bons tests ?

Exemple : `BigInteger.multiply()`

Que pensez-vous de cette batterie de test :

- ▶ un certain a négatif, un certain b négatif
- ▶ un certain a négatif, un certain b positif
- ▶ un certain a positif, un certain b négatif
- ▶ un certain a positif, un certain b positif

Comment écrire de bons tests ?

Exemple : `BigInteger.multiply()`

Toujours inclure les cas limites (frontières des partitions) a (ou b) est :

- ▶ positif
- ▶ négatif
- ▶ “petit entier” (small integer, représentable en `int`)
- ▶ “grand entier” (big integer, non représentable en `int`)
- ▶ égal à zéro
- ▶ égal à 1
- ▶ égal à -1

On a donc en tout une partition de $7 \times 7 = 49$ éléments.

Plan

Retour sur les deux semaines précédentes

La qualité logicielle

Le développement dirigé par les tests

Intégration continue

Intégration continue

Qu'est-ce ?

La CI/Continuous Integration/Intégration continue est la pratique consistant à faire en sorte que les changements apportés au dépôt de développement soient immédiatement intégrables/utilisables en production.

En pratique, il s'agit de fusionner plusieurs fois par jour vers la branche principale tout en s'assurant de maintenir les critères de qualité.

Déploiement continu

Qu'est-ce ?

On veut aussi aller plus loin : si la qualité est assurée, alors on déploie aussitôt l'artefact produit par le projet :

- ▶ installation sur le serveur de production
- ▶ ou bien publication d'un artefact/paquet sur un dépôt centralisé comme Maven Central, JCenter, NVM, FlatHub, voire Docker Hub... (cela dépend de la nature de l'artefact)

Cette partie du CI s'appelle le déploiement continu/continuous deployment.³

3. On le distingue de la livraison continue/continuous delivery, consistant juste à s'assurer que l'artefact est livrable, sans pour autant le déployer automatiquement.

Intégration continue

Comment faire?

Problème : Les tâches permettant d'assurer la qualité ou de faire le déploiement sont très répétitives (plusieurs commandes à répéter à chaque mise à jour du projet).

Intégration continue

Comment faire ?

Problème : Les tâches permettant d'assurer la qualité ou de faire le déploiement sont très répétitives (plusieurs commandes à répéter à chaque mise à jour du projet).

Tâches à répéter :

- ▶ La compilation (un minimum).
- ▶ La vérification de la qualité du code (« linter »).
- ▶ L'exécution de la suite de tests.
- ▶ La création de packages ou d'une image déployable (cf. Docker).
- ▶ Publication de l'artefact sur un serveur central.
- ▶ Installation de l'artefact sur le serveur de production.
- ▶ Lancement de la nouvelle version sur le serveur de production.

Intégration continue

Comment faire ?

→ On voudrait que toutes ces tâches soient automatiquement exécutées à chaque changement sur le git...

... or c'est justement possible dans GitLab⁴, sous le lien « Intégration et livraison continue » (abbrégé en CI/CD).

4. Et les principales plateformes concurrentes

CI/CD dans GitLab

Le fichier `.gitlab-ci.yml`

Pour configurer la CI/CD dans un projet gitlab, il suffit de créer un fichier `.gitlab-ci.yml` dans la racine du projet.

Ce fichier suit une syntaxe définie ici : <https://docs.gitlab.com/ee/ci/yaml/>

Exemple en ce qui nous concerne (page d'après) :

Le fichier .gitlab-ci.yml

```
image: gradle:8-jdk17-alpine

variables:
  GRADLE_OPTS: "-Dorg.gradle.daemon=false -Dhttps.proxyHost=194.254.199.96 -Dhttps.proxyPort=3128"

before_script:
  - export GRADLE_USER_HOME=`pwd`/.gradle
stages:
  - build
  - test

build:
  stage: build
  script: gradle --build-cache assemble
  cache:
    key: "$CI_COMMIT_REF_NAME"
    policy: push
    paths:
      - build
      - .gradle

test:
  stage: test
  script: gradle check
  cache:
    key: "$CI_COMMIT_REF_NAME"
    policy: pull
    paths:
      - build
      - .gradle
```


Le fichier `.gitlab-ci.yml`

Explications

```
image: gradle:8-jdk17-alpine
```

Choix d'une image docker.

Ici il s'agit d'une image Alpine Linux contenant gradle version 8 (celle qui est utilisée par **gradlew** dans Pac-Man) et Java 17.

Cette image sera téléchargée par GitLab sur la plateforme Docker Hub.

Le fichier `.gitlab-ci.yml`

Explications

```
variables:  
  GRADLE_OPTS: "-Dorg.gradle.daemon=false -Dhttp.proxyHost=193.254.199.96  
    -Dhttp.proxyPort=3128 -Dhttps.proxyHost=194.254.199.96 -Dhttps.proxyPort=3128"
```

La section **variables** permet d'indiquer des valeurs pour certaines variables d'environnement, permettant aux commandes définies dans les autres sections de s'exécuter.

En l'occurrence nous définissons l'adresse du proxy qui sera utilisée par la machine exécutant la CI/CD.

[**gaufre** délègue cela à **ginette**, qui est aussi une machine de l'UFR d'informatique, qui doit passer par le proxy de l'UFR pour télécharger les dépendances définies pour gradle].

Le fichier `.gitlab-ci.yml`

Explications

```
before_script:  
  - export GRADLE_USER_HOME=`pwd`/.gradle
```

La section **before_script** liste les commandes à exécuter avant d'exécuter l'un des jobs définis.

Le fichier `.gitlab-ci.yml`

Explications

```
build:
  stage: build
  script: gradle --build-cache assemble
  cache:
    key: "$CI_COMMIT_REF_NAME"
    policy: push
    paths:
      - build
      - .gradle
```

La section **build** donne les instructions pour compiler le projet.

- ▶ **stage** : étape du pipeline à laquelle appartient ce job
- ▶ **script** : le ou les commande(s) à exécuter pour effectuer le job
- ▶ **cache** : politique de rétention des fichiers temporaires entre les différentes exécutions du job.

Le fichier `.gitlab-ci.yml`

Explications

```
test:
  stage: test
  script: gradle check
  cache:
    key: "$CI_COMMIT_REF_NAME"
    policy: pull
    paths:
      - build
      - .gradle
```

La section **test** donne les instructions pour lancer les tests définis dans le projet.

La CI/CD sous GitLab

Étapes

Les étapes sont habituellement, dans cet ordre :

1. **build** : compilation du projet
2. **test** : exécution des tests
3. **deploy** : déploiement (sans objet pour Pac-Man)

Mais cela est configurable dans la sections **stages** de `.gitlab-ci.yml`.

À noter que si une étape échoue, les suivantes ne sont pas exécutées.

La CI/CD sous GitLab

Ajouter une étape

Par exemple, supposons qu'avant de compiler on veuille vérifier la qualité du code source ("linter"), on pourrait alors insérer :

stages:

- lint
- build
- test
- deploy

La CI/CD sous GitLab

Ajouter une étape

Pour l'étape **lint**, on peut par exemple demander d'exécuter le fameux contrôleur de style **checkstyle** (via plugin de gradle) :

```
styleMain:
  stage: lint
  script: gradle checkstyleMain
  cache:
    key: "$CI_COMMIT_REF_NAME"
    policy: pull
    paths:
      - build
      - .gradle
```

```
styleTest:
  stage: lint
  script: gradle checkstyleTest
  cache:
    key: "$CI_COMMIT_REF_NAME"
    policy: pull
    paths:
      - build
      - .gradle
```

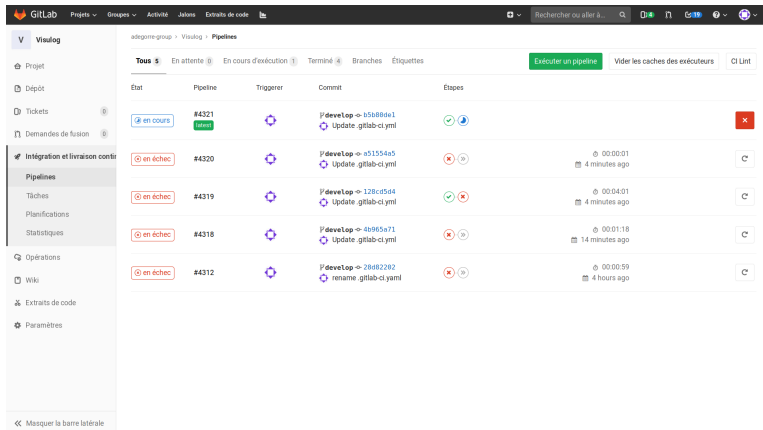
Il faudra encore configurer gradle (ajouter **id:checkstyle** à la section **plugins** de **build.gradle**)

... et configurer checkstyle (ajouter un fichier **checkstyle.xml**⁵ avec les options qui vous conviennent).

5. Exemples ici : <https://github.com/checkstyle/checkstyle/tree/master/src/main/resources>

Contrôler l'exécution de la CI/CD

L'onglet Pipelines



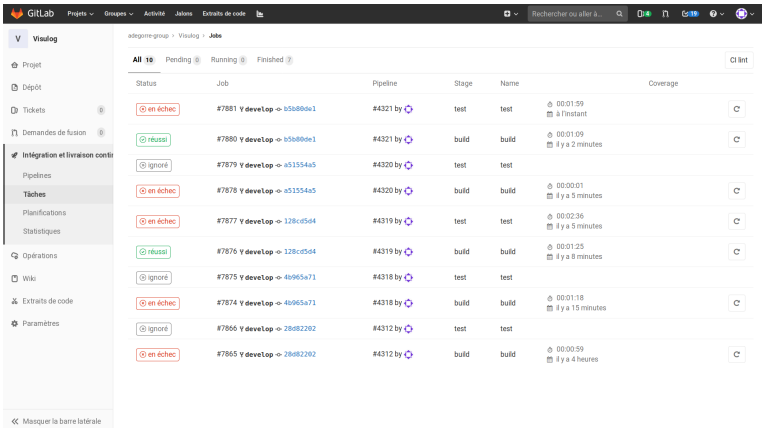
The screenshot shows the GitLab interface for the 'adegore-group' project, specifically the 'Pipelines' tab. The left sidebar contains navigation links: Visulog, Projet, Dépôt, Tickets, Demandes de fusion, Intégration et livraison continue (selected), Pipelines, Tâches, Planifications, Statistiques, Opérations, Wiki, Extraits de code, and Paramètres. The main content area displays a table of pipeline runs. The table has columns for 'État', 'Pipeline', 'Triggerer', 'Commit', and 'Étapes'. The first row shows a pipeline in 'en cours' (in progress) status with ID #4321, triggered by 'P'develop -> b5b88de1', and commit 'Update .gitlab-ci.yml'. The subsequent rows show pipelines in 'en échec' (failed) status with IDs #4320, #4319, #4318, and #4312, each with a duration and a timestamp. The 'Étapes' column shows the status of individual steps, with green checkmarks for successful steps and red X's for failed steps. A red 'X' icon is visible in the 'Étapes' column for the first pipeline.

État	Pipeline	Triggerer	Commit	Étapes
en cours	#4321	P'develop -> b5b88de1	Update .gitlab-ci.yml	
en échec	#4320	P'develop -> a51554a5	Update .gitlab-ci.yml	00:00:01 4 minutes ago
en échec	#4319	P'develop -> 128cd5d4	Update .gitlab-ci.yml	00:04:01 4 minutes ago
en échec	#4318	P'develop -> 4b965a71	Update .gitlab-ci.yml	00:01:18 14 minutes ago
en échec	#4312	P'develop -> 28d92202	rename .gitlab-ci.yml	00:00:59 4 hours ago

Une entrée est ajoutée à chaque changement sur le git. Le pipeline consiste en une séquence contenant une instance de chaque job défini dans `.gitlab-ci.yml`.

Contrôler l'exécution de la CI/CD

L'onglet Tâches



The screenshot shows the GitLab interface for the 'adegore-group' project, specifically the 'Jobs' tab under 'Visuallog'. The left sidebar contains navigation links for 'Projet', 'Dépôt', 'Tickets', 'Demandes de fusion', 'Intégration et livraison continue' (selected), 'Opérations', 'Wiki', 'Extraits de code', and 'Paramètres'. The 'Intégration et livraison continue' section is expanded, showing 'Pipelines', 'Tâches' (selected), 'Planifications', and 'Statistiques'. The main content area displays a table of jobs with the following columns: Status, Job, Pipeline, Stage, Name, and Coverage. The table shows 10 jobs, with 3 failed, 3 successful, and 4 ignored. Each job entry includes a status icon, a job ID, a job name, a pipeline ID, a stage, a name, a duration, and a 'C' icon for coverage.

Status	Job	Pipeline	Stage	Name	Coverage
en échec	#7881 Y develop -> b5b80de1	#4321 by	test	test	00:01:59 à l'instant
réussi	#7880 Y develop -> b5b80de1	#4321 by	build	build	00:01:09 il y a 2 minutes
ignoré	#7879 Y develop -> a51554a5	#4320 by	test	test	
en échec	#7878 Y develop -> a51554a5	#4320 by	build	build	00:00:01 il y a 5 minutes
en échec	#7877 Y develop -> 128cd5d4	#4319 by	test	test	00:02:36 il y a 5 minutes
réussi	#7876 Y develop -> 128cd5d4	#4319 by	build	build	00:01:25 il y a 8 minutes
ignoré	#7875 Y develop -> 4b965a71	#4318 by	test	test	
en échec	#7874 Y develop -> 4b965a71	#4318 by	build	build	00:01:18 il y a 15 minutes
ignoré	#7866 Y develop -> 28d82292	#4312 by	test	test	
en échec	#7865 Y develop -> 28d82292	#4312 by	build	build	00:00:59 il y a 4 heures

Cette vue a une granularité plus fine : chaque job est détaillé.

Contrôler l'exécution de la CI/CD

Vue de la console

The screenshot displays the GitLab web interface. On the left, a sidebar contains navigation links: 'Visuel', 'Projet', 'Dépôt', 'Tickets', 'Demandes de fusion', 'Intégration et livraison continue' (selected), 'Opérations', 'Wiki', 'Extraits de code', and 'Paramètres'. The 'Intégration et livraison continue' section is expanded, showing 'Pipeline', 'Tâches', 'Planifications', and 'Statistiques'. The main area shows the console output of a pipeline named 'test'. The output text includes: 'From https://gaufre.informatique.univ-paris-diderot.fr/adegorre-group/visulog', 'Checking out 128cd5d4 as develop...', 'Removing .gradle/', 'Skipping Git submodules setup', 'Checking cache for develop...', 'No URL provided, cache will not be downloaded from shared cache server. Instead a local version of cache will be extracted.', 'Successfully extracted cache', '\$ export GRADLE_USER_HOME=\$PWD/.gradle', '\$ gradle check', 'Welcome to Gradle 6.6.1!', 'Here are the highlights of this release:', '- Experimental build configuration caching', '- Built-in conventions for handling credentials', '- Java compilation supports --release flag', 'For more details see https://docs.gradle.org/6.6.1/release-notes.html', 'To honour the JVM settings for this build a new JVM will be forked. Please consider using the daemon: https://docs.gradle.org/6.6.1/userguide/gradle_daemon.html', 'Daemon will be stopped at the end of the build stopping after processing', and a list of tasks: 'Task :compileJava NO-SOURCE', 'Task :processResources NO-SOURCE', 'Task :classes UP-TO-DATE', 'Task :compileTestJava NO-SOURCE', 'Task :processTestResources NO-SOURCE', 'Task :testClasses UP-TO-DATE', 'Task :test NO-SOURCE', 'Task :check UP-TO-DATE', 'Task :config:compileJava', 'Task :gitrawdata:compileJava', 'Task :analyzer:compileJava', 'Task :analyzer:processResources NO-SOURCE', 'Task :analyzer:classes', and 'Task :analyzer:compileTestJava'. The pipeline status is 'test' with a duration of 2 minutes 13 seconds, a timeout of 1h, and a runner of 'gigi (r8)'. The commit is '128cd5d4' and the pipeline is '#4319 for develop'.

En cliquant sur un job, on peut regarder sa console d'exécution (utile pour comprendre une éventuelle erreur).

Contrôler l'exécution de la CI/CD

Planification

The screenshot shows the GitLab web interface for scheduling a new pipeline. The left sidebar contains navigation links: Visulog, Projet, Dépôt, Tickets, Demandes de fusion, Intégration et livraison continue (selected), Pipelines, Tâches, Planifications (active), Statistiques, Opérations, Wiki, Extraits de code, and Paramètres. The main content area is titled 'Planifier un nouveau pipeline' and includes the following fields:

- Description:** A text input field with the placeholder 'Indiquez une courte description'.
- Modèle d'intervalle:** Radio buttons for scheduling options: 'Personnalisé (Syntaxe de la planification cron)' (selected), 'Chaque jour (à 4 h du matin)', 'Chaque semaine (dimanche à 4 h du matin)', and 'Chaque mois (le 1^{er} à 4 h du matin)'.
- Fuseau horaire des tâches planifiées cron:** A dropdown menu currently set to 'UTC'.
- Branche cible:** A dropdown menu currently set to 'develop'.
- Variables:** A section with a 'Variable' dropdown, a 'Nom de la variable' input, and a 'Valeur de la variable' input.
- Activé:** A checkbox labeled 'Actif' which is checked.

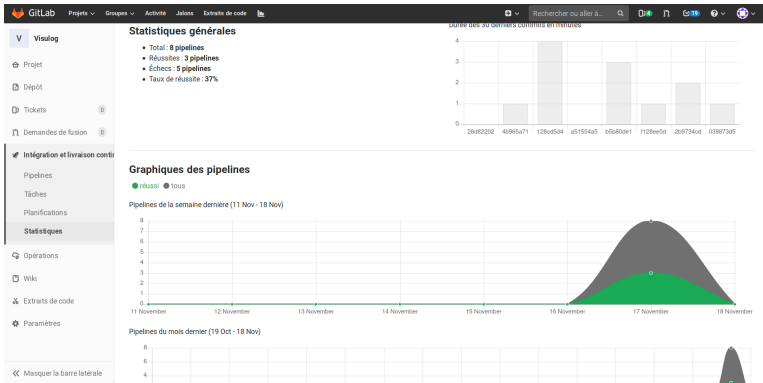
At the bottom of the form is a 'Masquer la barre latérale' (Hide sidebar) button.

Les pipelines sont par défaut exécutés automatiquement à chaque changement sur le git, mais cela est paramétrable (y compris désactivable).

On peut décider en plus, ou à la place, d'exécuter un pipeline à heure fixe, ou à jour fixe dans la semaine ou dans le mois.

Contrôler l'exécution de la CI/CD

L'onglet Statistiques



Dans cet onglet, il est possible de contrôler de niveau d'activité passé (important pour répartir la charge : en effet les ressources du serveur exécutant la CI/CD peuvent être limitées, voire payantes).

Bonne pratique par rapport à la CI

- ▶ Par défaut, le pipeline s'exécute à chaque changement sur git.
- ▶ Cela veut dire que ça ne concerne pas seulement **main/master** ou **develop**, mais aussi les feature branches.
- La bonne pratique : refuser de fusionner toute branche qui ne passe pas la CI.

Mais au fait, qu'est-ce que Docker ?

- ▶ Logiciel permettant de lancer des applications dans des conteneurs logiciels.
- ▶ Conteneur : environnement isolé au sein d'une session d'un système d'exploitation pour y exécuter un logiciel sans interférer avec le reste du système.
- ▶ Notamment, le conteneur contient toutes les dépendances nécessaires (qui peuvent ou non exister dans l'OS de base, pas forcément dans la même version).

→ très utile pour déployer un logiciel de façon fiable sur des cibles diverses, notamment quand on n'a pas la main pour modifier l'environnement installé.

Docker

Distribution et déploiement

- ▶ Les conteneurs sont initialisés à l'aide de fichiers distribuables, contenant le système de fichier du conteneur : les "images".
- ▶ Dans le cas de Docker, les images peuvent être envoyées par les développeurs sur le site <https://hub.docker.com/>.
- ▶ Les machines de production peuvent donc récupérer les images sur ce site. Cela peut être manuel, mais c'est souvent automatisé (dans le cadre du continuous deployment, par une règle dans le stage **deploy**).

Docker

En ce qui nous concerne :

- ▶ Pac-Man n'a pas vocation à être déployé sur des serveurs distants (c'est juste un jeu que l'utilisateur va typiquement installer lui-même sur sa propre machine).
De plus, il ne nécessite pas tout un environnement particulier pour s'exécuter (juste une JVM et une dépendance, JavaFX, qui peut être packagée avec le logiciel).
→ peu d'intérêt d'ajouter un job consistant à créer une image Docker du projet et de la déployer
(un packaging classique sous forme de `.jar` suffit)
 - ▶ Mais le CI/CD de GitLab fonctionne, lui-même, en utilisant des outils packagés dans des images Docker (en l'occurrence, pour nous : Gradle).
- pour GitLab, Docker est non seulement un **objectif** du CD/CI (parfois), mais aussi surtout un **moyen** (incontournable).

Lint

On appelle linter⁶ tout outil permettant de contrôler la qualité d'un code source de façon statique.⁷

Un tel outil peut signaler :

- ▶ des erreurs de programmation évidentes (par exemple en Java, faire un cast sans `instanceof`)
- ▶ des bugs (ex : usage de tableau avec indice connu comme négatif dès la compilation)
- ▶ des problèmes de style (ex : noms de classe en minuscule, mauvaise indentation, ...)
- ▶ des constructions suspectes (ex : liste de `if` imbriqués plutôt qu'un `switch`, ...)
- ▶ ...

6. Cela vient de la commande UNIX `lint`, un des premiers outils du genre, servant à analyser du code C.

7. C'est à dire en analysant le code sans l'exécuter. Cela s'oppose donc au test.

Lint

En Java, vous pouvez notamment⁸ utiliser (outre le compilateur `javac` avec l'option `-Xlint` et les outils intégrés à votre IDE) :

- ▶ SpotBugs : ... pour trouver des bugs!
- ▶ PMD : plutôt pour trouver les constructions suspectes (variables non utilisées, blocks `catch` vides, ...)
- ▶ Checkstyle : pour vérifier les conventions de codage (indentations, retours à la ligne, JavaDoc, ...)

8. Liste énorme ici :

Les linters peuvent généralement être configurés dans les systèmes de build comme Gradle ou Maven (via plugin) :

- ▶ cela ajoute automatiquement des tâches supplémentaires (comme **checkstyleMain** et **checkstyleTest**, pour Checkstyle sous Gradle)
- ▶ et peut déclarer les nouvelles tâches comme dépendances de tâches existantes (notamment la tâche **check** de Gradle, qui regroupe les tâches de vérification, dont la tâche **test**)