15/03/2023
Version 1

DEVOPS

JÉRÉMY PERROUAULT

INTRODUCTION

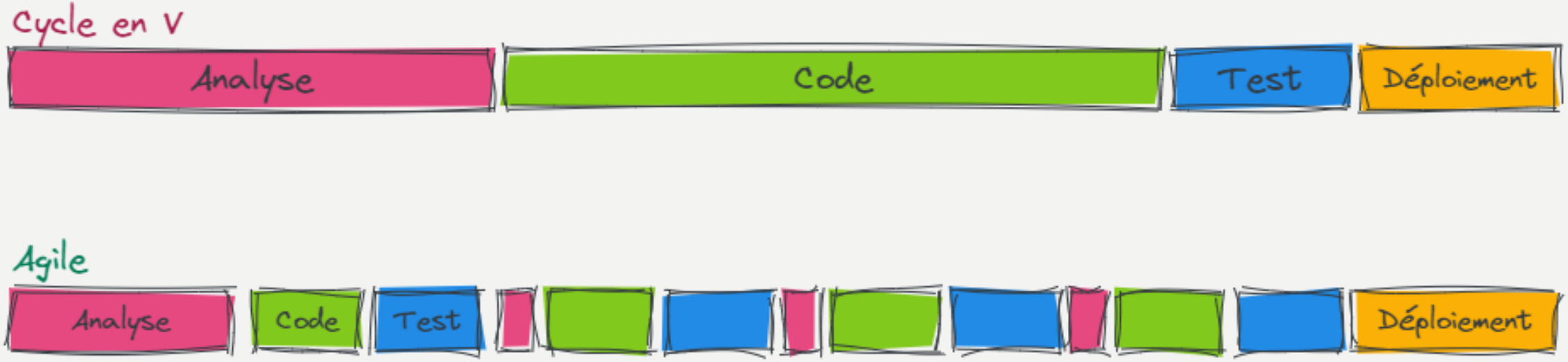
INTRODUCTION

PROBLÉMATIQUE

- Les projets cycles en V
  - 32% des projets sont réussis
  - 84% des projets dépassent le délai
  - 64% des fonctionnalités développées ne sont pas utilisées

- Ca s'explique …
  - Difficile de reproduire la demande client « à la lettre »
  - Les équipes (ou les personnes) ne discutent pas nécessairement entre elles, ou peu
  - Cette méthodologie en particulier ne permet pas de revenir en arrière

PROBLÉMATIQUE

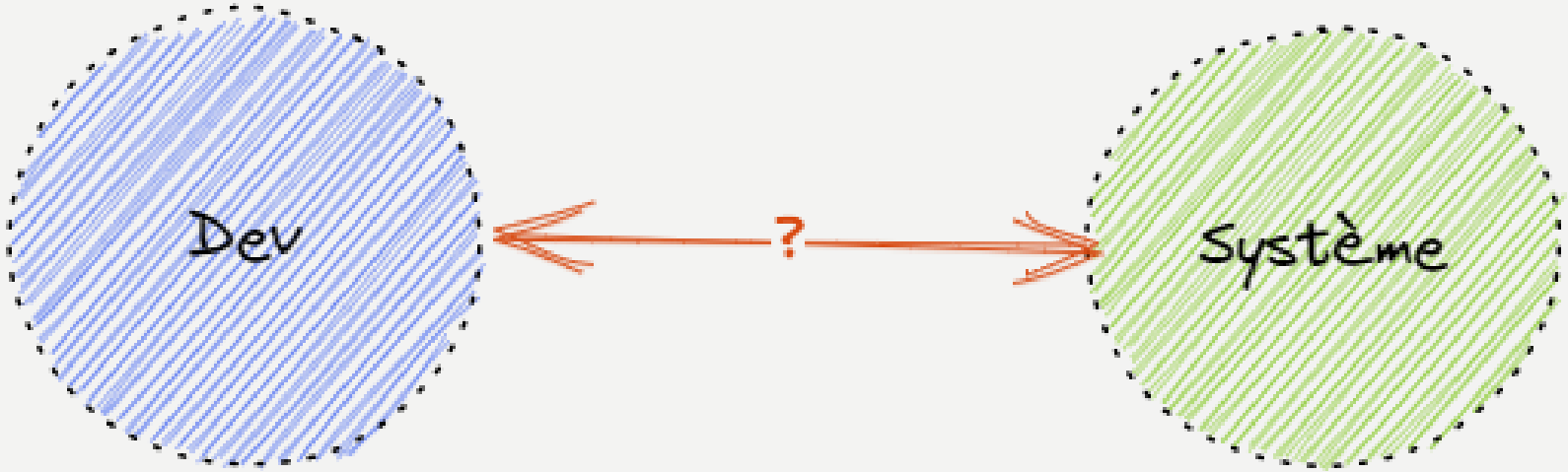


PROBLÉMATIQUE

- Le code
  - Qui fonctionne en dév mais pas en production
  - Qui utilise une version spécifique d'une dépendance
  - Qui ne teste pas / ou teste peu
  - Le logiciel a son cycle de vie, cycle en V
    - Analyse, développement, tests, déploiement
- Le système
  - Sur lequel le logiciel ne fonctionne pas ou n'est pas stable
  - Qui utilise une version spécifique d'une dépendance
  - Qui ne monitor pas / ou peu
  - Qui a son cycle de vie, la maintenance du système

PROBLÉMATIQUE

- Deux objectifs distincts
  - Les développeurs
    - Faire évoluer, innover, tester
  - Les opérateurs système
    - Garantir la stabilité et la fiabilité de l'application et du système en production

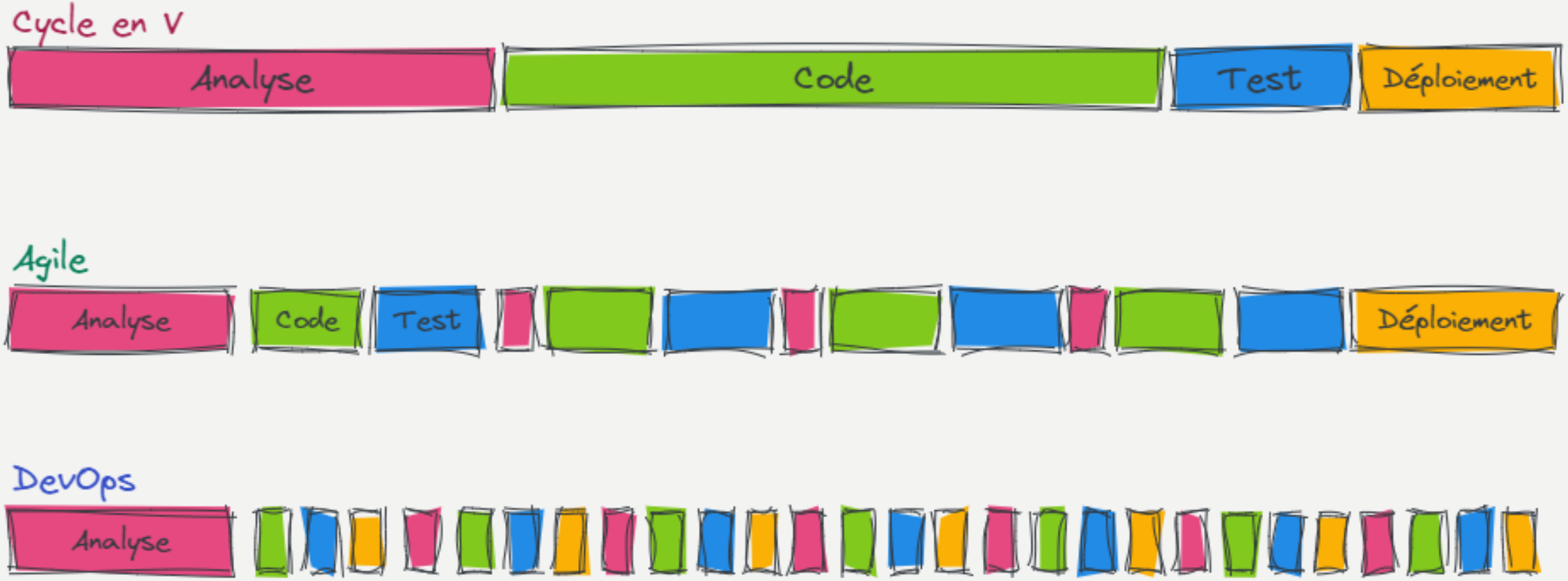


SOLUTION

- DevOps
  - Terme lancé par Patrick Debois (Belgique) en 2009
    - Idée de cette approche bien avant ça
  - Dev pour développeur
  - Ops pour opérations systèmes, les opérateurs

- C'est avant tout une philosophie, une culture, une approche

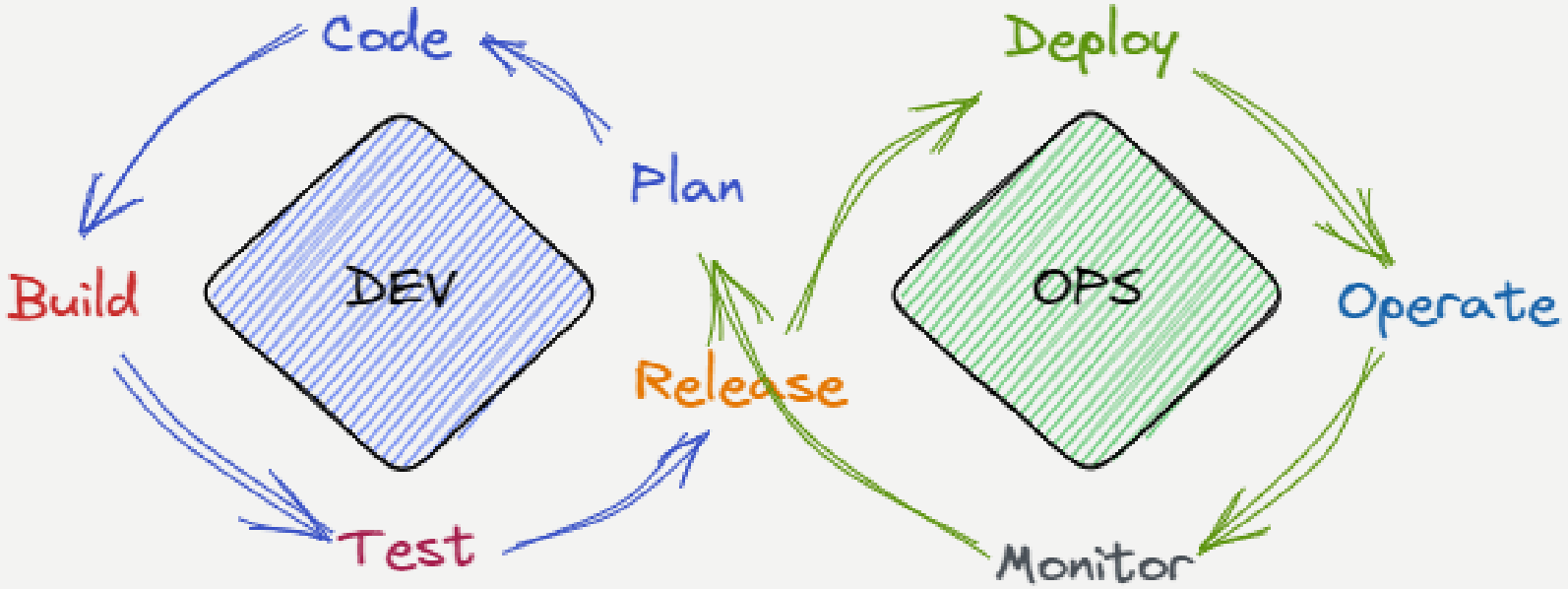
SOLUTION


Cycle en V
Analyse
Code
Test
Déploiement
Agile
Analyse
Code
Test
Déploiement
DevOps
Analyse

SOLUTION

- Les objectifs du DevOps
  - Améliorer la **coopération** entre `Dev` et `Ops`
    - Co-construction de l'application entre les deux domaines
  - Améliorer la **livraison** du produit
    - Il faut réactif, tout en garantissant la fiabilité et la stabilité, en évitant les régressions
    - C'est comme ça que Netflix, Facebook, etc. arrivent à sortir des mises à jour tous les jours ou presque
  - Fluidifier l'**élaboration** du produit
    - La partie Dev et Ops ne doivent pas se ralentir l'une l'autre
      - Permet plus de liberté sur les innovations

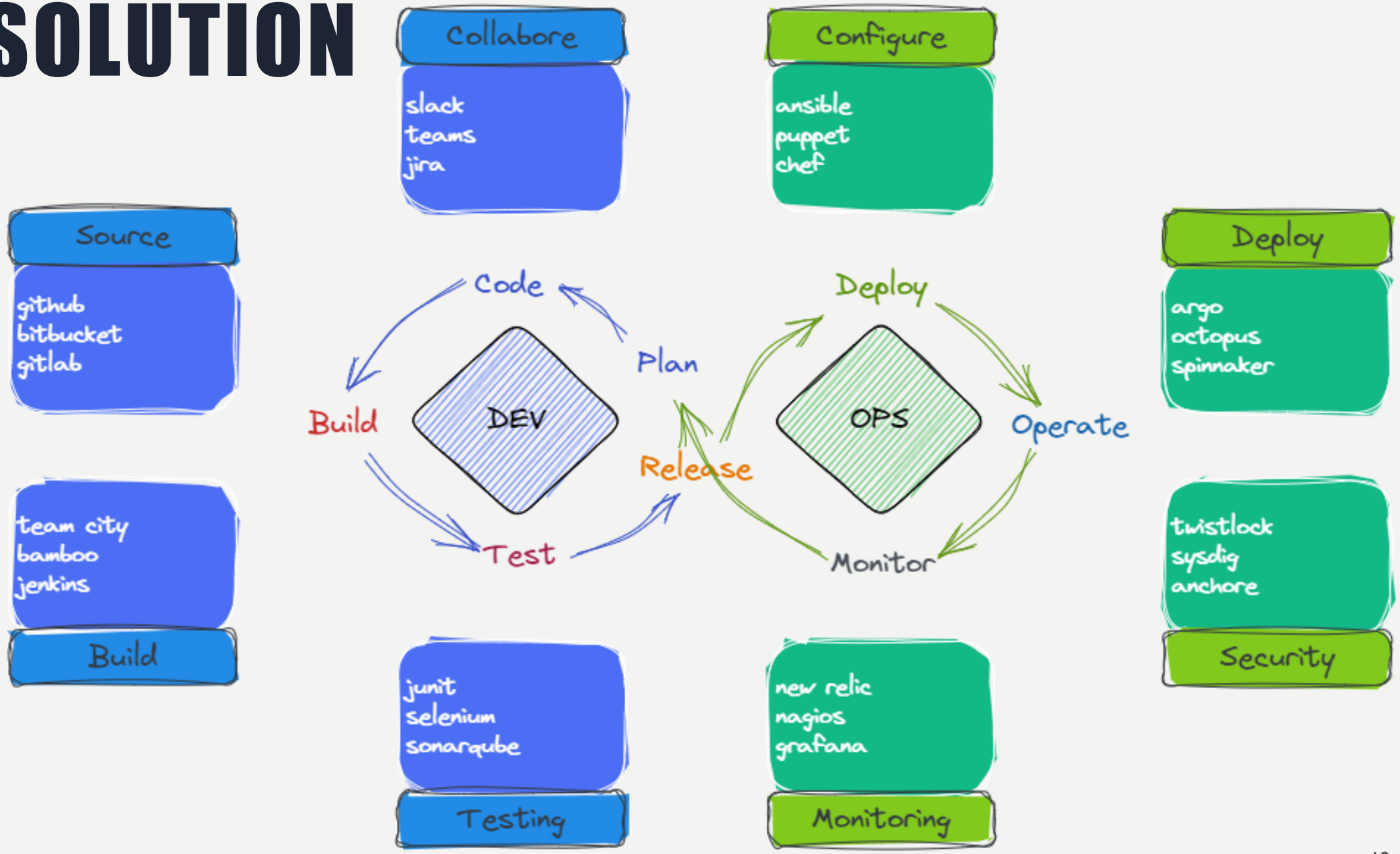
SOLUTION


Code
Plan
Build
DEV
Test
Release
Deploy
OPS
Operate
Monitor

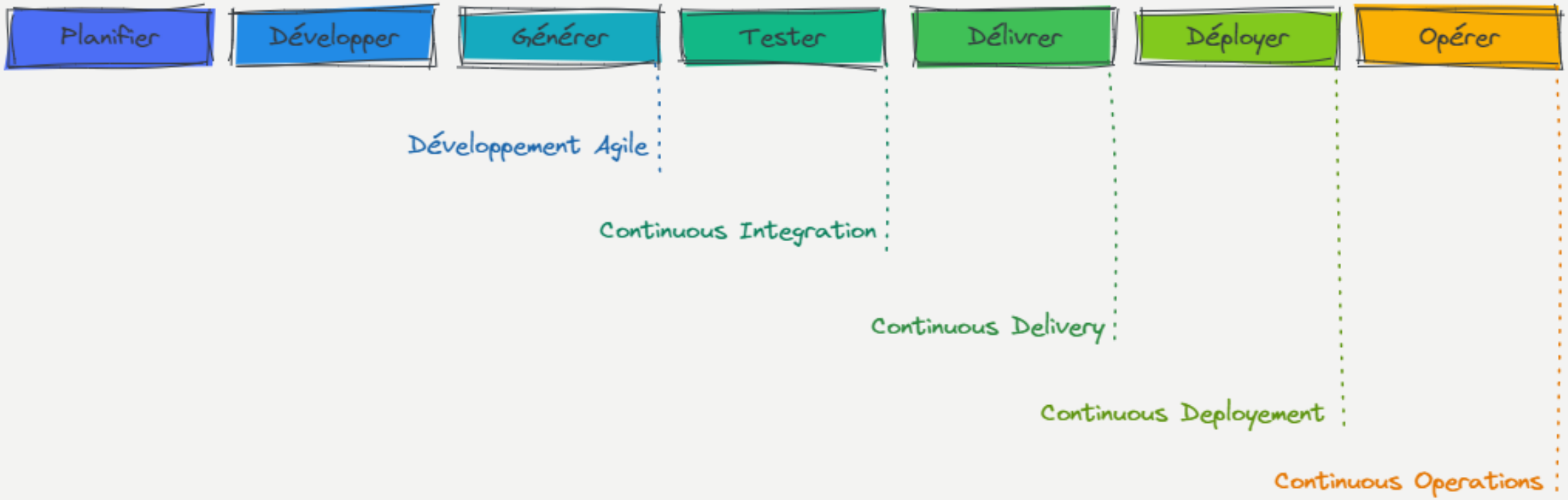
SOLUTION

- DevOps est une approche centrée sur l'automatisation et l'agilité
- Mais attention, il ne s'agit pas seulement d'utiliser les bonnes méthodes et les bons outils
- Il faut penser bien en amont l'architecture du code et du système !
  - C'est pour cette raison que c'est une philosophie avant tout

SOLUTION


Collabore
slack
teams
jira
Configure
ansible
puppet
chef
Source
github
bitbucket
gitlab
Deploy
argo
octopus
spinnaker
Code
Plan
Build
DEV
Test
Release
Deploy
OPS
Operate
Monitor
team city
bamboo
jenkins
Build
twistlock
sysdig
anchore
Security
junit
selenium
sonarqube
Testing
new relic
nagios
grafana
Monitoring

SOLUTION


Planifier
Développer
Générer
Tester
Délivrer
Déployer
Opérer
Développement Agile
Continuous Integration
Continuous Delivery
Continuous Deployement
Continuous Operations

PRATIQUES DÉV

BONNES PRATIQUES

BONNES PRATIQUES

- Respecter des normes de développement et bonnes pratiques
  - Faciliter la maintenance et l'évolution
  - Détecter plus facilement les problématiques
- Utiliser un outil de gestion de versions (**Git** par exemple)
  - Travailler de façon collaborative
  - Revenir sur un travail antérieur

BONNES PRATIQUES

- Style de code
  - Indentations espaces ou tabulations (mais garder le même style partout !)
  - Accolades et leur positionnement
  - lowerCamelCase, CamelCase (ou PascalCase), snake_case, kebab-case
- Métriques
  - Longueur de code (nombre de lignes)
    - Classes, Méthodes
  - Nombre de paramètres / d'attributs / de méthodes
  - Complexité cyclomatique / NPath (chemins de sortie)

BONNES PRATIQUES

- Pour une meilleure maintenabilité
  - Utiliser les constantes (chaines ou nombres) partout où il y a répétition ou lorsque l'information peut évoluer
  - Couplage faible et injection de dépendances
  - Eviter les arguments *null*, préférer surcharger ou utiliser des mécanismes de compensation
  - Ne pas hésiter à utiliser *final* si la valeur ne doit pas changer
  - Ne pas hésiter à utiliser les Enumérateurs à la place des constantes (si applicable)
  - Eviter les attributs publics
  - Eviter les blocs if/else ou switch quand une surcharge est possible
  - Eviter les conditions négatives
  - Mettre en place une journalisation (des loggers)
    - Eviter les écritures dans la console lorsque ce n'est pas nécessaire (c'est rarement le cas)
  - Rédiger des tests

BONNES PRATIQUES

- Respecter des principes de développement
  - KISS Keep It Simple, Stupid
  - DRY Don't Repeat Yourself
  - YAGNI You Ain't Gonna Need It

  - Nom de variables, méthodes, attributs et classes cohérents
  - Données de configuration faciles à changer
- Avec le temps, il y a une tendance à complexifier le code
  - Opérations de refactoring pour restaurer la bonne qualité
  - Régressions possibles dans ce cas

BONNES PRATIQUES

- Concernant les noms
  - Respecter une convention (selon le langage, le projet, l'équipe, etc.)
  - Choisir des noms descriptifs et sans ambiguïté
  - Utiliser des noms qui ont du sens et qui sont recherchables facilement

BONNES PRATIQUES

- Concernant les méthodes / fonctions
  - Ecrire une méthode la plus courte possible
  - La méthode ne fait qu'une seule chose (séparation des responsabilités)
  - Minimiser le nombre d'arguments
  - Minimiser les effets de bord

BONNES PRATIQUES

- Concernant les commentaires
  - Nul besoin de commenter des choses simples

```
i++; // Incrémentation de i
```

  - Ecrire un code expressif, ne nécessitant pas (ou peu) de commentaires
    - Si ce n'est pas possible, prendre le temps d'écrire un bon commentaire
  - Ne pas écrire de commentaire de fermeture de bloc

```
// Fin du if
```

  - Pour les évolutions / corrections, ne pas commenter le code, mais le supprimer
    - **Git** est là pour retrouver un ancien code
  - Commenter les intentions et les conséquences (si applicable)

BONNES PRATIQUES

- Pour la résolution des bugs
  - Toujours chercher la cause racine (pas de pansements !)

- Pour **Git**
  - Limiter les délais de séparation des branches
    - Plus ce laps de temps est long, plus les problèmes de conflits seront présents
  - Préférer des commits et des merges réguliers

BONNES PRATIQUES

- Tout ceci fait parti des principes SOLID
  - S Single responsibility principle
    - Chaque classe et méthode est responsable d'une seule chose
  - O Open/Close principle
    - Chaque classe et méthode doit être fermée à la modification mais ouverte à l'extension
      - Préférer créer une nouvelle classe plutôt que de modifier la classe directement
  - L Linskov substitution principle
    - Une instance peut être remplacée par une autre instance (dérivé ou implémentation) sans que cela ne modifie le code ou sa cohérence
  - I Interface segregation principle
    - Préparer plusieurs interfaces pour chaque client, plutôt qu'une seule
  - D Dependency inversion principales
    - Les dépendances doivent être des abstractions, pas des implémentations (couplage faible)

QUALIMÉTRIE

MESURER LA QUALITÉ DU CODE

QUALIMÉTRIE

- La qualimétrie est la mesure de la qualité du code
  - Permet de faciliter et de limiter le refactoring

QUALIMÉTRIE

- Nécessité d'auditer le code
  - Avec des *linters*, liés à l'environnement de développement
  - Avec des outils externes, comme **SonarQube**

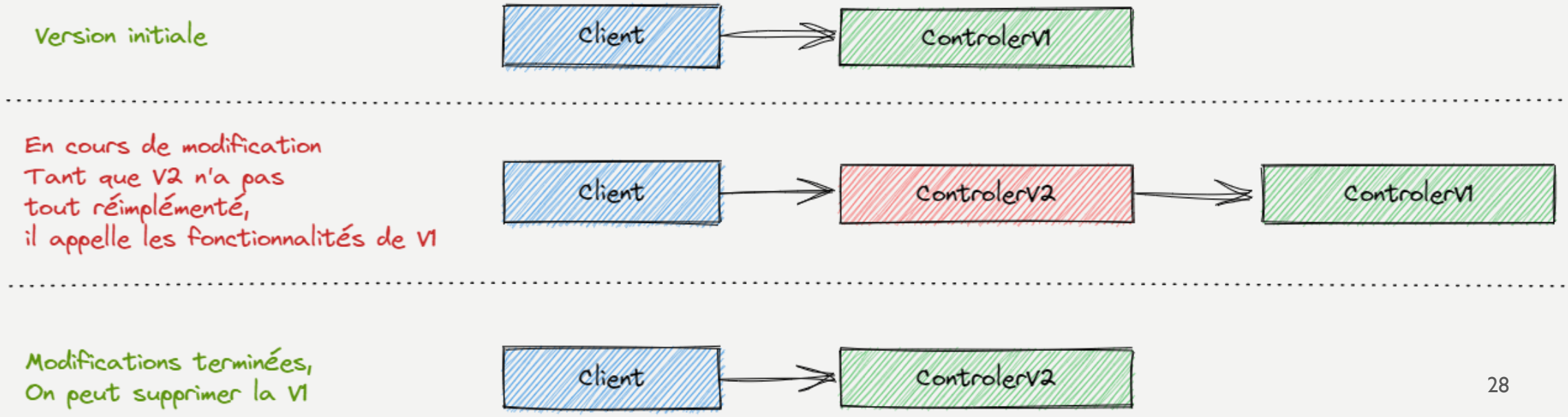
- Nécessité de couverture du code
  - Vérifier que la couverture est maximale (80% minimum)
  - Détecter les morceaux de code non utilisés et/ou non testés
    - Grâce aux tests unitaires
    - Grâce aux tests d'intégration
    - Grâce aux tests end-to-end

QUALIMÉTRIE

- Un code qui doit être refactoré est un code qui va être modifié
  - Identifier ce qui doit / peut être amélioré
  - Suivre la progression
  - S'assurer de la non-régression (utilité des tests unitaires)
  - Limiter les efforts à fournir

QUALIMÉTRIE

- Pour les modifications importantes, respecter le principe `Open/Close`
  - Mettre progressivement en place les modifications en créant une extension
    - Qui jouera un rôle de proxy sur les méthodes non réimplémentées
    - Cela aura aussi l'avantage de permettre des commits plus réguliers



EXERCICE

- Mettre en place une classe de service `ProduitService`
  - Qui liste les produits depuis un fichier (fictif)
  - Qui ajoute un produit dans un fichier (fictif)

- Procéder à la mise à jour partielle
  - La liste des produits provient maintenant d'une base de données (fictive)
  - L'ajout se fait toujours dans un fichier (fictif)

QUALIMÉTRIE

- **JMeter**
  - Tests de performance d'applications Web et de serveurs
  - Tests de robustesse
  - Tests de rupture
- **SonarQube**
  - Règles de développements classiques
  - Ensemble de règles de design pattern
  - Qualité et lisibilité du code
  - Identification de failles
  - Couverture de code

QUALIMÉTRIE

Métrique	Seuil conseillé
Nombre d'erreurs bloquantes / critiques	0
Taux de couverture de code	80 %
Densité des commentaires	20 à 40%
Nombre de lignes de code / méthode	< 50 lignes
Complexité cyclomatique (McCabe)	20 – 30
Nombre de « Si » imbriqués	3
Nom de variable incompréhensible	0
Taux de répétitions	3%

TESTS

TESTS UNITAIRES

TESTS UNITAIRES

- Pour nous aider dans ces tâches
  - Tests unitaires
  - Tests d'intégration

- Un test unitaire doit être
  - Simple / Lisible
  - Isolé / Indépendant
  - Rapide à s'exécuter

TESTS UNITAIRES

- La qualité d'un test unitaire
  - C'est la capacité à déterminer le nombre de tests unitaires pour une méthode

- Exemple : méthode avec une condition « if – else »
  - Deux états
  - Au moins deux tests unitaires (un pour chaque état)

TESTS UNITAIRES

- Pour les tests unitaire en **JAVA**
  - Utilisation de **JUnit**
    - Dépendance **Maven** junit-jupiter-api

EXERCICE

- Rédaction d'un code et de son ou ses tests unitaires
  - En respectant des règles énoncées

- Ajouter une classe `Calculatrice`
  - Fonctionnalités `additionner` et `soustraire`

TDD

TEST DRIVEN DEVELOPMENT

TDD

- L'approche classique du test
  - Développer la fonctionnalité
  - Tester la fonctionnalité
    - Plus le bug est détecté tard, plus il remet en cause d'éléments, et sa correction peut être coûteuse
    - Le code étant déjà écrit, le risque est d'écrire un test qui valide le code (un test faux)
- L'approche **TDD**
  - Développer le test
  - Développer la fonctionnalité
    - Tant que le test n'est pas validé, la fonctionnalité n'est pas valide

TDD

- L'exécution d'un test **doit**, en premier lieu, donner un résultat négatif
  - La rédaction du code qui s'en suit doit valider progressivement le ou les tests
    - En respectant toujours des principes vu précédemment
  - Un refactor peut suivre
    - Principe « Red-Green-Refactor »

TDD

- Deux approches
  - Tout d'un bloc : un ou plusieurs tests pour l'ensemble des cas
    - Généralement pour les algorithmes facilement compréhensibles / dont on maîtrise déjà la finalité
    - Quelques cas simples, et les cas les plus complexes
    - Rédaction du code jusqu'à ce que tout passe au vert
  - Petits cycles itératifs
    - Rédiger le test d'un cas simple
    - Rédiger le minimum de code possible pour faire passer le test
    - Enrichir le test d'un deuxième cas simple
    - Compléter la fonctionnalité pour valider les deux tests
    - Refactoriser le code
    - Enrichir le test d'un troisième cas, plus complexe
    - Compléter la fonctionnalité pour valider les trois tests
    - Refactoriser le code
    - etc. jusqu'à couvrir tous les cas

TDD

- Peu importe l'approche, chaque test reste
  - Simple
  - Isolé
  - Indépendant
  - Rapdie
- Et on a toujours plusieurs tests pour valider une fonctionnalité

TDD

- Avantages techniques
  - Le temps de débogage est considérable réduit
    - La détection de bugs est plus immédiate
  - Les tests de non-régression sont déjà prêts
  - L'architecture est testable et de bonne qualité
  - On réduit le risque de valider un code faux par un test faux
  - Les tests peuvent ainsi être utilisés comme documentation

  - Evite également les étourderies liées au copier-coller de code (réguliers …)
  - Evite d'oublier de déclarer la méthode de test comme étant une méthode de test (@**Test**)

TDD

- Le **TDD** est bien du test, avec une philosophie différente
  - « Tester c'est construire » plutôt que « Tester c'est douter »

TDD

- Démonstration **JAVA** (**JUNIT**)

EXERCICE

- **JAVA (JUNIT)**
  - Soit une méthode qui attend 2 entiers : a et b
  - Additionner si a est positif
  - Soustraire si a est négatif
    - a est transformé en entier positif

EXERCICE

- **JAVA (JUNIT)**
  - Fizz Buzz FizzBuzz
    - Pour des entiers de 1 à 100
    - Les multiples de 3 sont remplacés par « Fizz »
    - Les multiples de 5 sont remplacés par « Buzz »
    - Les multiples de 15 sont remplacés par « FizzBuzz »

    - 12Fizz4BuzzFizz78FizzBuzz.....

BDD

BEHAVIOR DRIVEN DEVELOPMENT

BDD

- Pour Behavior Driver Development
- Peut être associé au **TDD**
- Permet de la rédaction d'un test fonctionnel dans le langage naturel (français par exemple)
  - Pour les cas d'usages
  - Facilement compréhensible et vérifiable par un non-technicien
  - Sert de base documentaire

- Le langage utilisé pour **BDD** est **Gherkin**
- Un outil souvent utilisé est **Cucumber**

BDD

- On place des fonctionnalités à décrire dans des fichiers *.feature*

```
Feature: API Démonstration
  Scenario: Le client appelle /api/demo en GET
    Given le client appelle /api/demo
    When le client donne un id 10
    Then le client reçoit un statut 200
    And le client reçoit le message "Hello"

    When le client donne un id 15
    Then le client reçoit un statut 404
```

BDD

- Qu'on peut aussi écrire en français (ou autre langue)

```
#language: fr
Fonctionnalité: API Démonstration
  Scénario: Le client appelle /api/demo en GET
    Sachant que le client appelle /api/demo
    Quand le client donne un id 10
    Alors le client reçoit un statut 200
    Et le client reçoit le message "Hello"

    Quand le client donne un id 15
    Alors le client reçoit un statut 404
```

BDD

- Puis coder en **JAVA** (ou autre)

```java
@Sachantque("^le client appelle /api/demo$")
public void given() throws Throwable { }

@Quand("le client donne un id {int}")
public void whenClientGivesId(int id) throws Throwable {
    this.result = this.mockMvc.perform(MockMvcRequestBuilders.get("/api/demo/" + id));
}

@Alors("^le client reçoit un statut (\\d+)$")
public void thenClientReceivesStatus(int status) throws Throwable {
    result.andExpect(MockMvcResultMatchers.status().is(status));
}

@Et("^le client reçoit le message \"(.+)\"$")
public void andClientReceivesText(String value) throws Throwable {
    result.andExpect(MockMvcResultMatchers.content().string(value));
}
```

BDD

- Pour utiliser **Cucumber** en **JAVA**
  - Dépendances Maven
    - cucumber-java
    - cucumber-junit-platform-engine
    - cucumber-spring (si utilisateur de **SPRING** ou **SPRING BOOT**)

    - junit et junit-platform-suite

BDD

- Pour configurer les tests **Cucumber**
  - Créer la classe de test `CucumberIntegrationTest`
  - Les fichiers *.feature* seront dans le répertoire /src/test/resources/sous-repertoire
  - Les classes de test de **Cucumber** seront dans un sous-package « cucumber »
    - fr.formation.cucumber par exemple
    - Seront suffixées de préférence par `Steps` ou `StepDefinitions`
      - `OperationSteps` ou `OperationStepDefinition` par exemple

```
@Suite
@IncludeEngines("cucumber")
@SelectClasspathResource("<sous-repertoire_resources>")
@ConfigurationParameter(key = GLUE_PROPERTY_NAME, value = "<package.test.cucumber>")
public class CucumberIntegrationTest {

}
```

EXERCICE

- Implémenter les tests **BDD** des fonctionnalités précédemment implémentées

EXERCICE

PETIT PROJET FIL-ROUGE

EXERCICE FIL ROUGE

- Mise en place d'une **API REST** « Liste des produits »
  - Lister des produits
  - Voir un produit
  - Ajouter un produit
  - Modifier un produit
  - Supprimer un produit

- `Produit` a quelques attributs : `id`, `nom`, `prix`

- Utiliser **TDD** et **BDD**
- Utiliser une base de données embarquée (**H2** par exemple)