Agilité, qualité et intégration continue

Vers l'application des bonnes pratiques de développement

Jérôme Buisine & Éric Ramat

28 novembre 2022

Principaux objectifs

L'objectif principal

Produire un code de la **meilleure qualité** possible

Principaux objectifs

L'objectif principal

Produire un code de la **meilleure qualité** possible

Remarques importantes

- Oublier la fameuse phrase « Tester c'est douter »
- Tester au maximum les fonctionnalités
- Couvrir au maximum le code
- Privilégier un code générique pour les évolutions futures
- Automatiser ce qui peut être automatisé

Principaux objectifs

On évitera donc... :

Déroulement du module

Organisation du module

- Cours à chaque séance sous un format court (15-30 min)
- **chaque TD/TP** :
  - une version (tag de projet) à fournir comme rendu
  - une note
- un projet final de 21h (intervenant extérieur) :
  - 3 jours de 7h
  - un projet à traiter selon une méthodologie agile (Scrum + TDD + ...)
  - une soutenance de retour d'expérience

Note finale

note du module $= \frac{1}{2}$ examen $+ \frac{1}{2}$ (moyenne des notes TD/TP + projet)

Plan

Plan

Plan

Quelques données historiques

Matériel vs Logiciel

Systèmes informatiques

- 80% de logiciel
- 20% de matériel

Remarques sur le matériel

- le matériel est fiable
- le marché est standardisé
- le coeur du matériel est assuré par quelques fabricants

Quelques données historiques

Matériel vs Logiciel

Systèmes informatiques

- 80% de logiciel
- 20% de matériel

Remarques sur le matériel

- le matériel est fiable
- le marché est standardisé
- le coeur du matériel est assuré par quelques fabricants

L'essentiel problème..

Les principaux problèmes sont maintenant principalement **logiciels**..

Des développements coûteux mis en échecs

1962 la sonde Mariner vers Vénus s'est perdue dans l'espace à cause d'une erreur dans un programme FORTRAN.

1981 le premier lancement de la navette spatiale a été retardé de deux jours à cause d'un problème logiciel.

1996 l'explosion d'Ariane 5 qui a coûté un demi milliard de dollars (non assuré..), est due à une faute logicielle d'une composante (dont le fonctionnement n'était pas indispensable durant le vol).

2000 le fameux bug de l'an 2000.

Les motivations du Génie Logiciel

Quand les chiffres parlent...

Étude sur 8380 projets (Standish Groupe, 1995)

- succès : 16%
- problématique : 53% (budgets/délais non respectés, défaut logiciel)
- échec : 31% (abandonnés)

Small vs. Large projects (The CHAOS Manifesto, 2013)

- succès : 76% vs. 10%
- problématique : 20 % vs. 52%
- échec : 4% vs. 38%

Les motivations du Génie Logiciel

Quand les chiffres parlent...

Étude sur 8380 projets (Standish Groupe, 1995)

- succès : 16%
- problématique : 53% (budgets/délais non respectés, défaut logiciel)
- échec : 31% (abandonnés)

Small vs. Large projects (The CHAOS Manifesto, 2013)

- succès : 76% vs. 10%
- problématique : 20 % vs. 52%
- échec : 4% vs. 38%

Un problème d'organisation plutôt que technique

- difficulté de formalisation/compréhension/découpage des besoins
- manque de répartition des rôles des différents acteurs
- nécessité d'utiliser les bons outils

Plan

Avant tout ! Qu'est ce qu'un logiciel ?

Un logiciel

- ensemble des programmes et des procédures nécessaires au fonctionnement d'un système informatique
- produit immatériel qui est indépendant du support physique
- peut être comparé à une oeuvre d'art

Ses spécificités et contraintes

- cherche à répondre à des besoins techniques
- traite généralement de l'information (mal traitée $\rightarrow$ engendre des erreurs)
- fonctionne ou ne fonctionne pas (la remarque « ça fonctionne sur ma machine » n'est pas la bienvenue)

Un cycle de production différent d'un produit « classique »

- aucun coût réel pour la reproduction
- utilisation d'un principe de license

Le Génie Logiciel

Définition

Un ensemble de bonnes pratiques

Le génie logiciel est une science de génie industriel qui étudie les **méthodes** de travail et les **bonnes pratiques**.

Le génie logiciel s'intéresse en particulier aux **procédures** qui permettent de produire des logiciels qui :

- correspondent aux **besoins** et attentes du client
- soient fonctionnels, **fiables** et performants
- aient une **maintenabilité peu coûteuse**
- respectent les **délais** et les **coûts** de conception

Le Génie Logiciel

Les principaux défis

Les questions à se poser :

- comme produire des logiciels de qualité ?
- quels sont les attentes d'un logiciel ?
- comment définir les critères de qualité d'un logiciel ?

Comment arriver à simultanément :

- assurer la maintenance du nombre croissant de logiciels (qui vieillissent).
- conserver un rythme de production logicielle qui puisse répondre à la demande.

Génie logiciel

Évaluer la qualité logicielle

Les différents gages de qualité

- **Validité :** réponse aux besoins des utilisateurs
- **Facilité d'utilisation :** prise en main et robustesse
- **Performance :** temps de réponse, débit, fluidité...
- **Fiabilité :** tolérance aux pannes
- **Sécurité :** intégrité des données et protection des accès
- **Maintenabilité :** facilité à corriger ou transformer le logiciel
- **Portabilité :** changement d'environnement matériel ou logiciel

Génie logiciel

Les principes

Généralisation

Regrouper l'ensemble des fonctionnalités semblables en une paramétrables (généricité, héritage)

Structuration

Manière de décomposer le logiciel (utilisation d'une méthode bottom-up ou top-down)

Abstraction

Mécanisme qui permet de présenter un contexte en exprimant les éléments pertinents et en omettant ceux qui ne le sont pas

Génie logiciel

Les principes

Modularité

Décomposer le logiciel en modules et composants

Documentation

Gestion des documents incluant identification, acquisition, production, stockage et distribution

Vérification

Déterminer l'atteinte des spécifications exprimées par les besoins

Génie logiciel

Les objectifs

Produire un logiciel

- comprendre et mettre en évidence les besoins
- organisation du projet
  - cycle de vie (en V, spirale, itératif...)
  - patrons d'organisation
- mise en place technique
  - qualité du logiciel (savoir l'évaluer)
  - test (unitaire, fonctionnel, d'interface...)
  - *design patterns*
  - automatisation des processus et suivi

Plan

Génie logiciel

Activités du développement

De l'analyse du besoin au produit

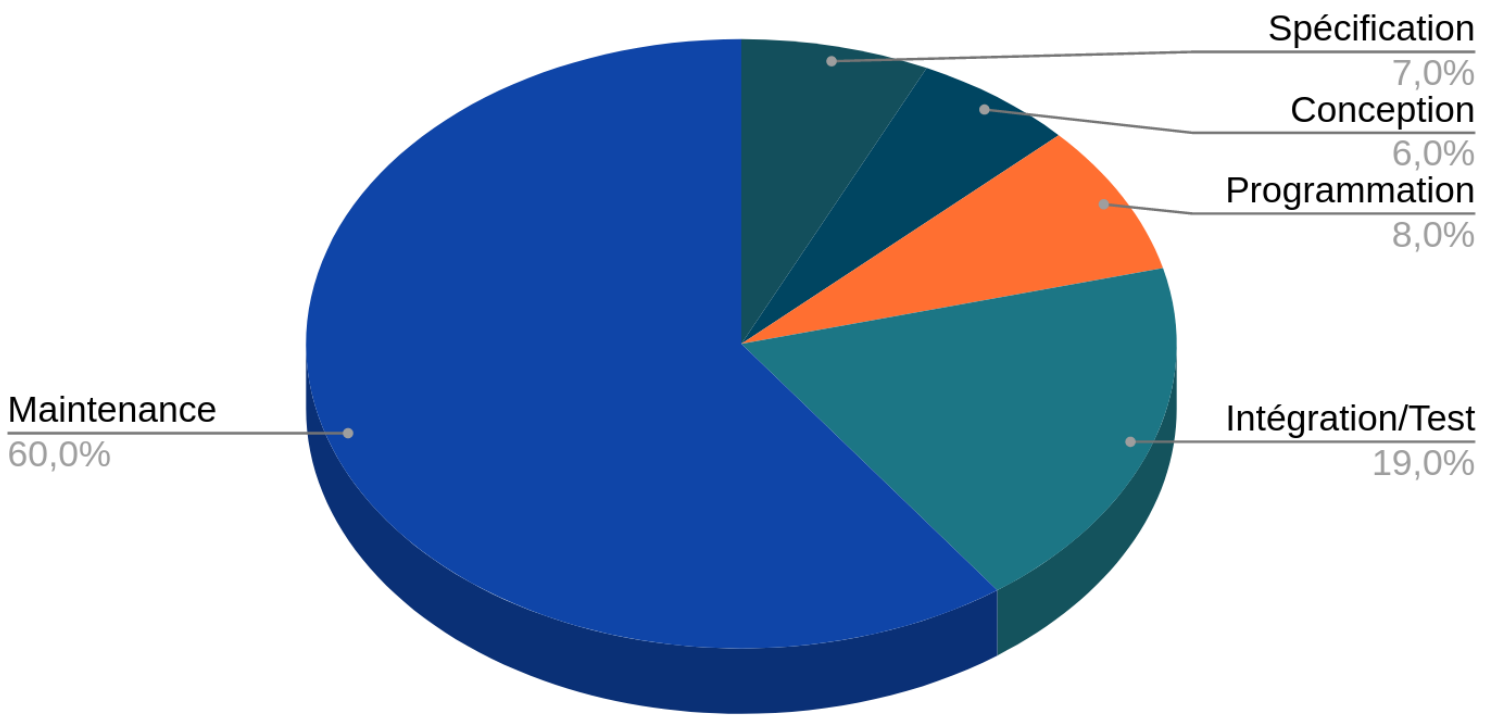
- **Analyse des besoins :** étude de faisabilité des besoins et fonctionnalités
- **Spécification :** établir une description claire de ce que doit faire le logiciel
- **Conception :** élaborer une solution concrète réalisant la spécification
- **Programmation :** Implantation de la solution conçue
- **Validation/Vérification :** assurer que les besoins du client sont satisfaits/le logiciel satisfait sa spécification
- **Maintenance :** assurer le suivi et l'évolution du produit
  - **Correction :** identifier et corriger des erreurs trouvées après la livraison
  - **Adaptation :** adapter le logiciel aux changements dans l'environnement (format des données, environnement d'exécution...)
  - **Perfection :** améliorer la performance, ajouter des fonctionnalités, améliorer la maintenabilité du logiciel

Génie logiciel

Répartition de l'effort

Une maintenance coûteuse en temps

L'effort demandé pour un logiciel de qualité n'est pas forcément à fournir là où l'on pensait



Génie logicielle

Sans oublier l'organisation !

Bien s'organiser une des clés du succès

- déterminer comment on va développer le logiciel
- analyse des coûts : établir une estimation du prix du projet
- planification : établir un calendrier de développement
- assurance qualité du logiciel : déterminer les actions qui permettront de s'assurer de la qualité du produit fini
- répartition des tâches : hiérarchiser les tâches et sous-tâches nécessaires au développement du logiciel

Génie logicielle

Autres notions clés

Les différents types de tests

- tests unitaires : faire tester les parties du logiciel par leurs développeurs
- tests d'intégration : tester pendant l'intégration (erreurs non détectables par les tests unitaires)
- tests de validation : pour acceptation par le client
- tests système : tester dans un environnement proche de l'environnement de production
- tests de recette : faire tester par le client sur le site de développement et sur le site de production
- tests de régression : enregistrer les résultats des tests et les comparer à ceux des anciennes versions pour vérifier si dégradation il y a

Génie logicielle

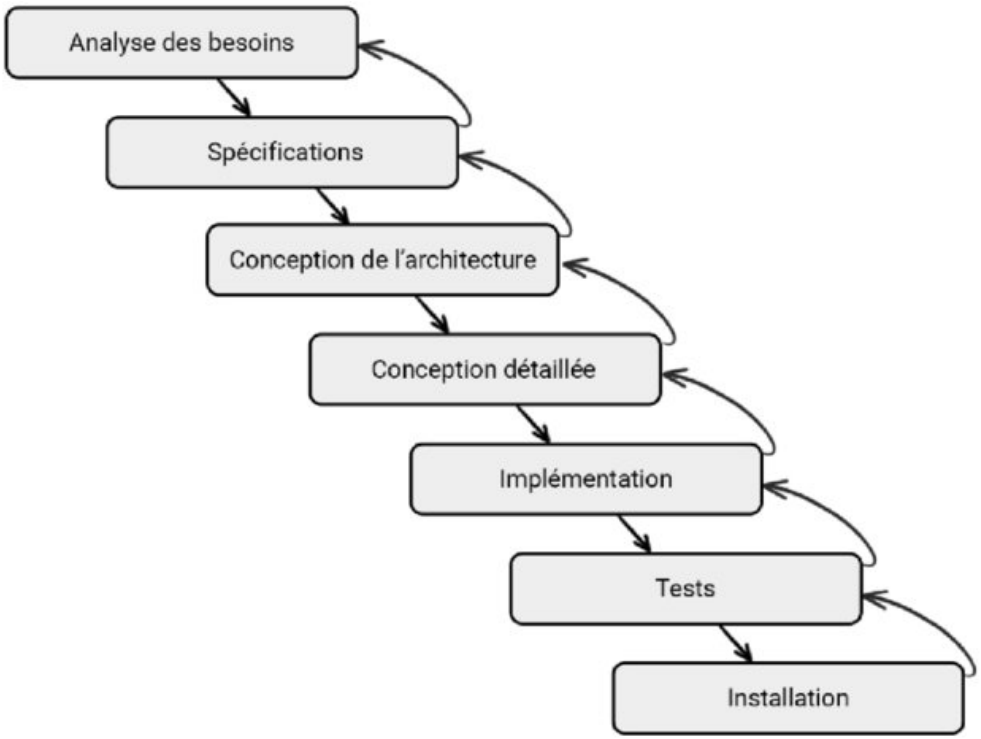
Autres notions clés

Livraison / livrable

- fournir au client une solution logicielle qui fonctionne correctement
- installation : rendre le logiciel opérationnel sur le site du client
- formation : enseigner aux utilisateurs à se servir du logiciel
- assistance : répondre aux questions des utilisateurs

Génie logicielle

Les différents modèles de cycle de vie

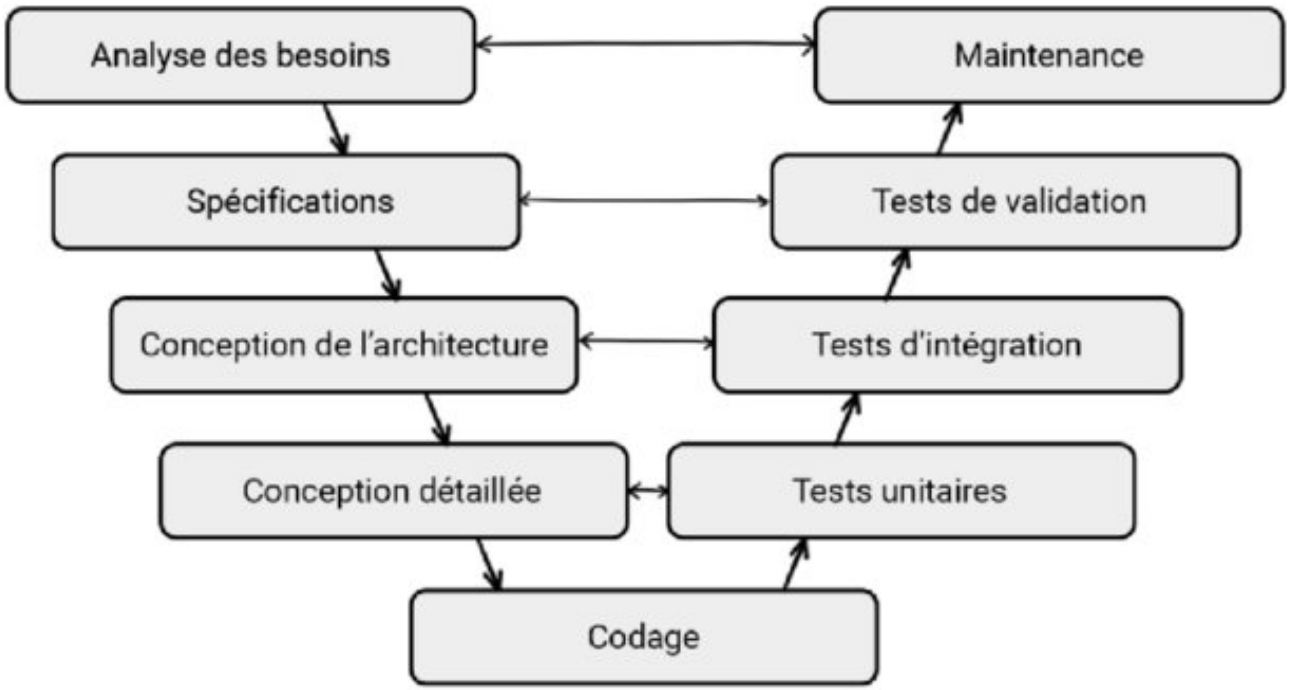


Modèle en cascade

Réalisation des étapes les unes après les autres avec un retour possible sur les précédentes

Génie logicielle

Les différents modèles de cycle de vie

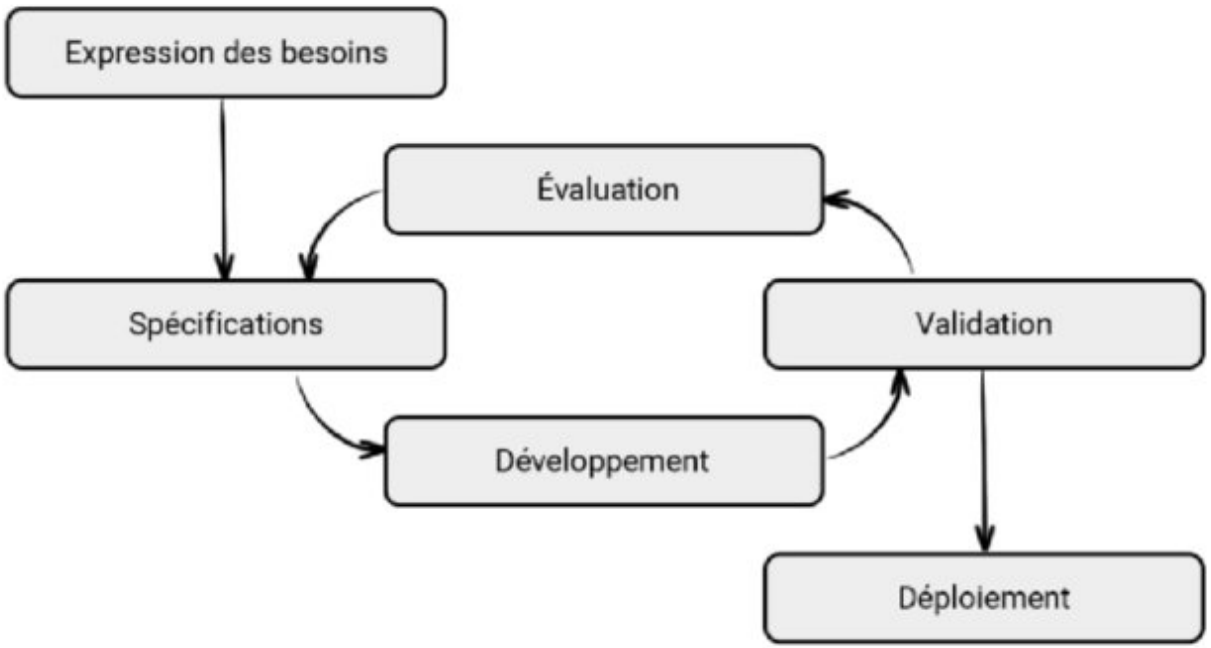


Modèle en V

Mise en correspondance d'une relation entre les étapes descendantes et ascendantes

Génie logicielle

Les différents modèles de cycle de vie

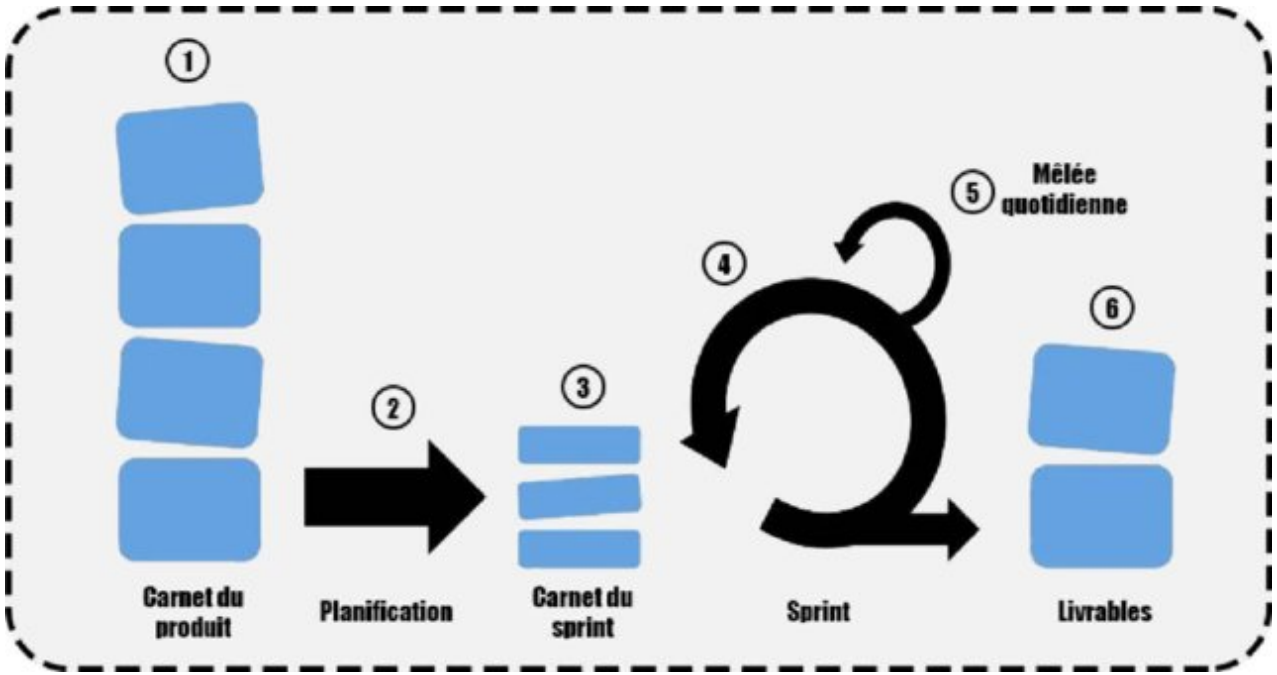


Modèle en itératif

Production du logiciel par itération : une itération = un produit (un artefact)
$\rightarrow$ vers les méthodologies agiles

Génie logicielle

Les différents modèles de cycle de vie



Modèle Scrum

Découpage d'un projet en « boîtes de temps », nommées sprints pour livrables intermédiaires

Plan

Plan

Un problème commun ?

Trouver un nommage pour versionner un document n'est pas évident :

```
Mémoire_v1.doc
Mémoire_v2.doc
Mémoire_v3.doc
Mémoire_Final.doc
Mémoire_Final_v2.doc
Mémoire_Final_Final.doc
...
Mémoire_Final_pour_de_vrai.doc
```

Parlons convention

Une **convention** dans un premier temps ?

```
Mémoire_v0.0.1.doc
Mémoire_v0.0.2.doc
Mémoire_v0.1.0.doc
Mémoire_v0.2.0.doc
Mémoire_v1.0.0.doc
Mémoire_v1.0.1.doc
...
Mémoire_v2.0.0.doc
```

Problème de collaboration

Comment gérer le maintien du document avec plusieurs personnes ?

Problème de collaboration

Comment gérer le maintien du document avec plusieurs personnes ?

- Qui travaille actuellement et sur quelle version ?
- Quelle est réellement la dernière version ?

Problème de collaboration

Comment gérer le maintien du document avec plusieurs personnes ?

- Qui travaille actuellement et sur quelle version ?
- Quelle est réellement la dernière version ?

Figure – Visage de la personne désignée volontaire pour la fusion du document

Problème de collaboration

Comment gérer le maintien du document avec plusieurs personnes ?

- Qui travaille actuellement et sur quelle version ?
- Quelle est réellement la dernière version ?

Figure – Visage de la personne désignée volontaire pour la fusion du document

Édition de document en ligne

Outils **collaboratifs** permettant d'éditer en ligne et en parallèle le même fichier aisément

Plan

Outils collaboratifs

Pour les développeurs :

(a) Fondation Apache en 2004

(b) Linus Thorvald en 2008

L'outil collaboratif `git`

Particularités :

- Système de contrôle de version décentralisé
- Chaque participant possède un clone de l'ensemble du référentiel en local
- Ajouter, contribuer et suivre les changements dans le code source

L'outil collaboratif `git`

Particularités :

- Système de contrôle de version décentralisé
- Chaque participant possède un clone de l'ensemble du référentiel en local
- Ajouter, contribuer et suivre les changements dans le code source

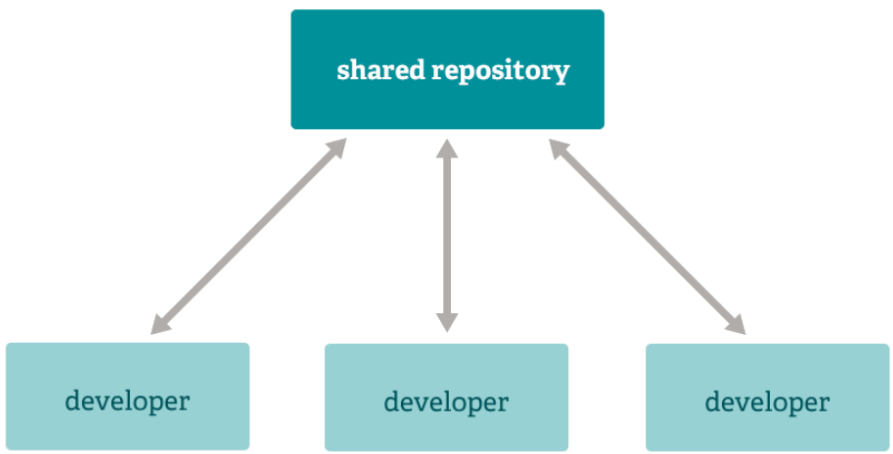


Figure – Échanges bilatéraux de développeurs (source : git-scm).

Plan

Interface de commandes

Commandes `git` de base :

```
# Initialisation du projet
git init

# Ajout de fichier dans l'index
git add file.txt

# Vérification de l'index
git status

# Suppression d'un fichier dans l'index
git reset file.txt
```

Ajout de modifications

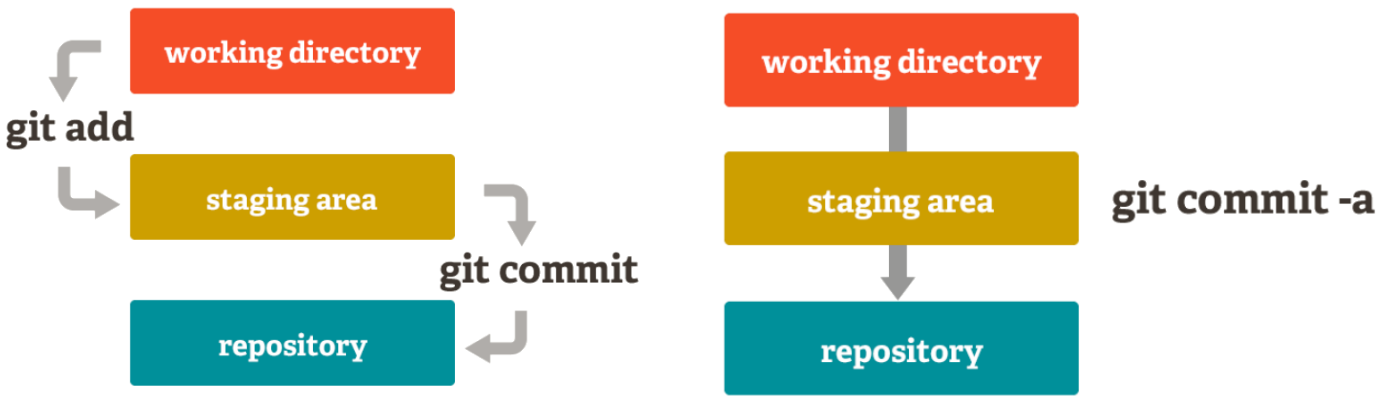


Figure – Ajout et validation de contenu de fichiers (source : git-scm)

Gestion des modifications

Ajout de contenu :

```
# Validation de l'index (fichiers ajoutés)
git commit -m "some updates"

# Ajout des modifications courantes et validation
git commit -am "some updates"

# Aperçu des commits
git log

# Affichage condensé des commits
git log --oneline
```

Suppression de modifications

Suppressions locales :

```
# Supprimer le dernier commit local
git reset --soft HEAD^

# Supprimer les 2 derniers commits locaux
git reset --soft HEAD~2
```

Plan

Version de projet

Numéros de version sous la forme `X.Y.Z`

- X - Majeur : suppression d'une fonctionnalité obsolète, modification d'interfaces, renommages...
- Y - Mineur : introduction de nouvelles fonctionnalités, fonctionnalité marquée comme obsolète...
- Z - Correctif : modification/correction d'un comportement interne, failles de sécurité...

Ajout de version

Réaliser une livraison avec version :

```
# Ajout d'un tag de version à l'état actuel du projet
git tag -a v1.0.0 -m "Releasing version v1.0.0"

# Liste l'ensemble des versions du projet
git tag -l

# Affiche les informations détaillées d'un tag
git show v1.0.0
```

Gestion des branches

Vers l'utilisation de branches ? ?

Gestion des branches

Vers l'utilisation de branches ? ?

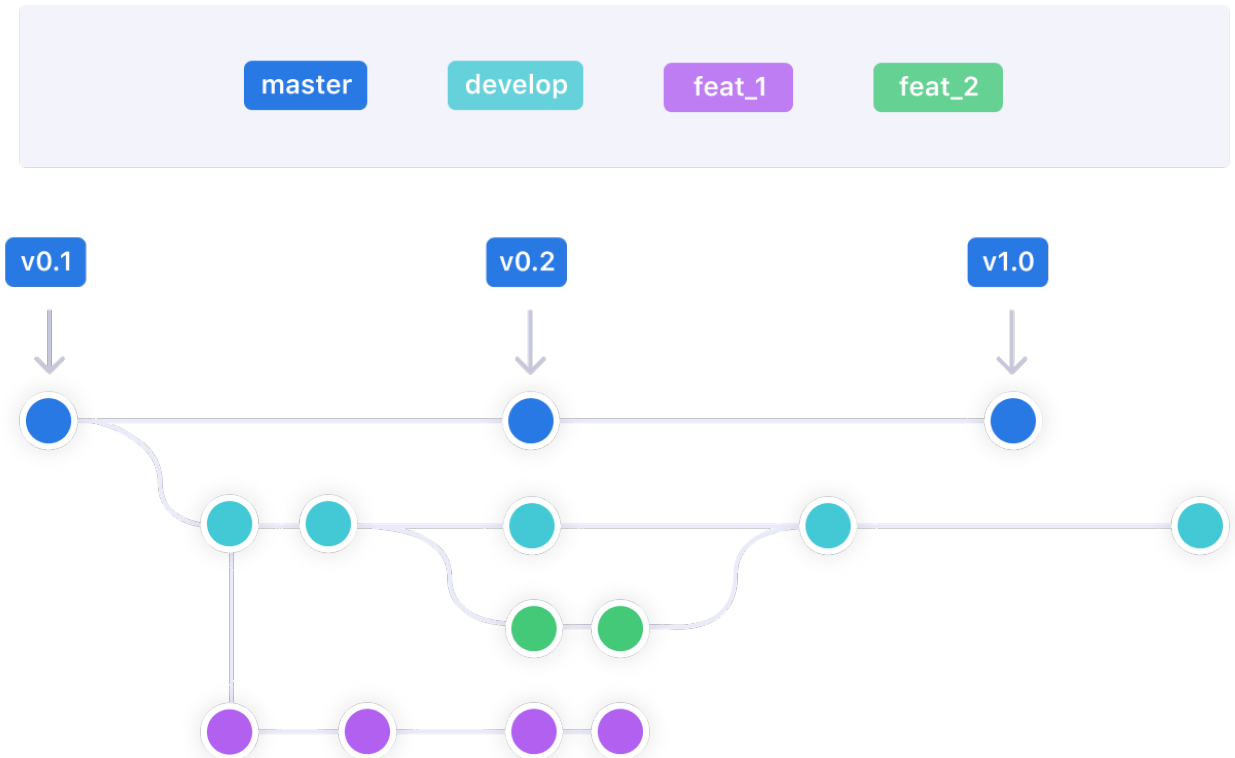


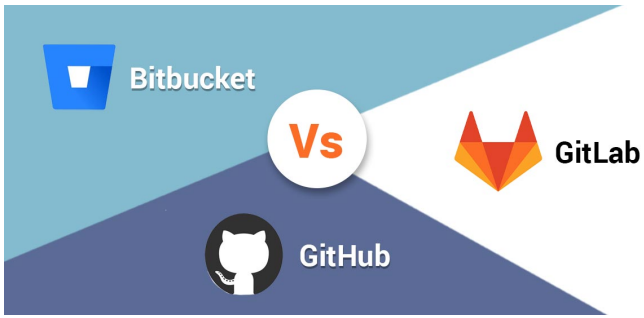
Figure – Exemple de gestion des branches sous `git` (source : qovery.com).

Gestion des branches

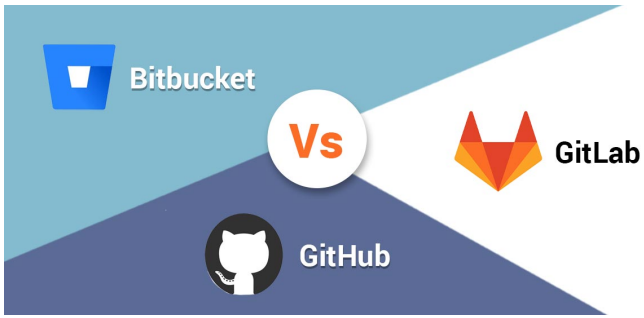
Commandes pour la gestion des branches :

```
# Liste toutes les branches disponibles
git branch

# Crée une nouvelle branche
git branch ma-branche

# Change de branche de développement
git checkout ma-branche

# Création et changement de branche courante
git checkout -b ma-branche

# Supprime une branche
git branch -d ma-branche

# Depuis la branche `develop`, récupération des modifications de `ma-branche`
# Plus évident à manipuler que rebase
git merge ma-branche
```

Serveur d'hébergement



Serveur d'hébergement



Interaction avec le serveur d'hébergement :

```
# Énumère les serveurs d'hébergements référencés
git remote -v

# Récupère les nouvelles branches existantes
git fetch <remote-name>

# Récupère et applique les modifications du serveur d'hébergement
git pull <remote-name> ma-branche

# Publie les modifications apportés par une branche sur un serveur
# !! toujours réaliser un pull avant de push !!
git push <remote-name> ma-branche
```

Autres commandes

Commandes à ne pas confondre :

- **git revert** : crée un nouveau commit qui annule les changements d'un commit précédent.
- **git checkout** : extrait le contenu du référentiel et le place dans votre arbre de travail (elle permet aussi le déplacement vers une autre branche).
- **git reset** : il modifie l'index (la « zone de transit »). Ou elle change le commit sur lequel une tête de branche pointe actuellement. Cette commande peut modifier l'historique existant.

Autres commandes

Cas d'utilisations possibles :

- **git revert** : si un commit a été fait quelque part dans l'historique du projet, et que vous décidez plus tard que ce commit est mauvais. L'utilisation de *git revert* annulera les changements introduits par le mauvais commit, en enregistrant le « undo » dans l'historique.
- **git checkout** : restaure une révision historique d'un fichier donné (`git checkout <file-path> <commit-hash>`).
- **git reset** : si vous avez fait un commit, mais que vous ne l'avez pas partagé avec quelqu'un d'autre et que vous décidez que vous n'en voulez pas, alors vous pouvez utiliser git reset pour réécrire l'historique de sorte qu'il semble que vous n'ayez jamais fait ce commit.

Outils de gestion de version

TP0 : Introduction à l'outil Git.

Plan

Plan

L'intégration continue

Définition

L'intégration continue est un **ensemble de pratiques** utilisées en génie logiciel consistant à **vérifier à chaque modification de code source** que le résultat des modifications ne produit **pas de régression** dans l'application développée.

L'intégration continue

Définition

L'intégration continue est un **ensemble de pratiques** utilisées en génie logiciel consistant à **vérifier à chaque modification de code source** que le résultat des modifications ne produit **pas de régression** dans l'application développée.

Objectif

Intégrer de façon continuelle les changements apportés à un projet

L'intégration continue

Les avantages

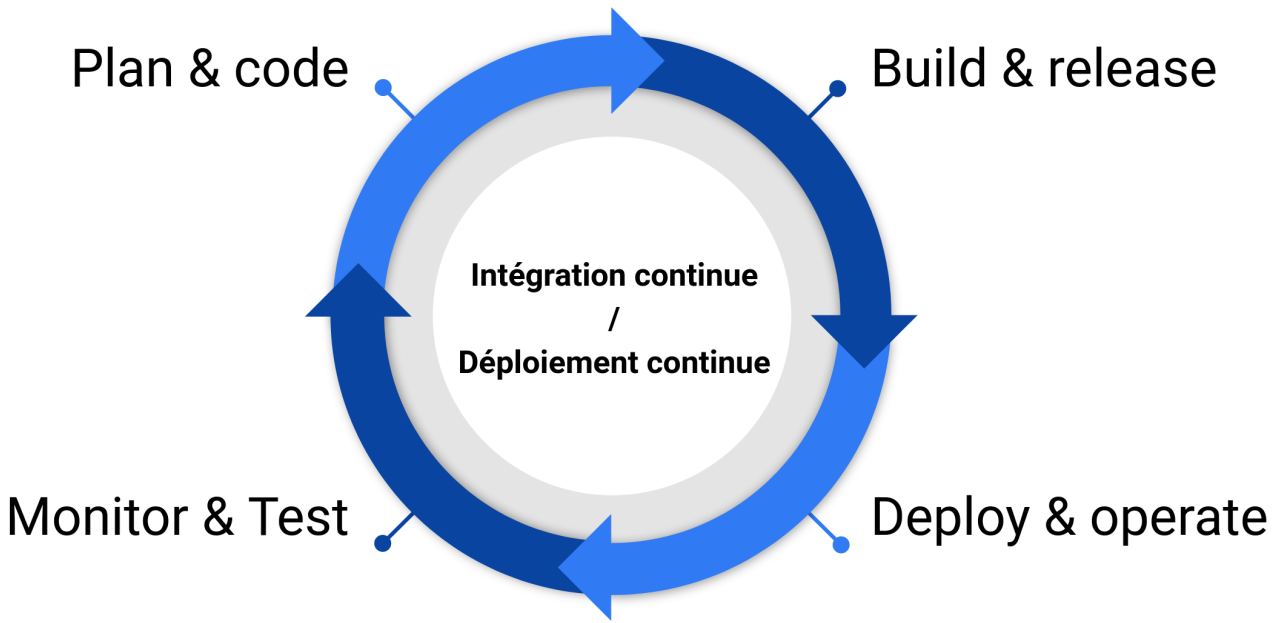
- réduire les risques liés à l'intégration (conflits notamment sur un gros projet).
- retour permanent du code développé.
- améliorer la qualité du code (métriques).
- pas de surcharge contrairement à une seule grande intégration finale.
- disponibilité continue d'une version actuelle opérationnelle.

L'intégration continue

Les inconvénients

- potentielles difficultés de conversions de processus habituels.
- nécessite des serveurs et des environnements supplémentaires (coûts supplémentaires).
- mettre au point des processus de test adaptés.
- des délais d'attente peuvent survenir lors de l'intégration de code approximativement au même moment par les développeurs.

L'intégration continue : vue comme un cycle



Plan

L'intégration continue : les outils les plus connus

Quelques outils existants d'intégration :

- **Jenkins :** qui utilise différents outils de build et des systèmes de gestion de versions (open source)
- **Travis CI :** fonctionne sans difficulté avec GitHub (disponible en version gratuite pour les projets open source)
- **Bamboo :** intégration, déploiement et gestion des versions (développé par Atlassian et disponible en version gratuite pour les projets open source)
- **GitLab CI :** pipeline qui peut être personnalisé et adapté à chaque projet (open source)
- **CircleCI :** possible d'opter pour une version directement sur le Cloud du fournisseur ou de créer un serveur local dédié pour l'utilisation (version gratuite mais limitée)
- **TeamCity :** vérifie automatiquement le code réécrit avant l'intégration dans la mainline et informe le développeur en cas d'erreurs (développé par Jetbrain)
- et bien d'autres : AWS CodeDeploy, Octopus Deploy, DeployBot...

Déploiement continue : les différents outils

Solutions de déploiement :

- **Infrastructure as a Service (IaaS) :** services basés sur le cloud tels que les serveurs, la mise en réseau et le stockage, généralement, propose un modèle de paiement à l'utilisation (ressources physiques et virtualisées).
  - **Solutions :** AWS, Digital Ocean, Google Cloud, Azure, Linode, Vultr, Alibaba Cloud...
- **Platform as a Service (PaaS) :** fournit une infrastructure comprenant le stockage, la mise en réseau et les serveurs ainsi que des « *middlewares* », des logiciels ou des outils de développement, des systèmes de gestion de base de données, des outils d'informatique décisionnelle pour déployer, gérer et faire évoluer les applications.
  - **Solutions :** Heroku, Engine Yard, Google App Engine, Dokku...
  - **Cas particulier :** Terraform (IaaS + PaaS)

Déploiement continue : les différents outils

Solutions de déploiement :

- **Backend as a Service (BaaS) :** orienté cloud, les développeurs externalisent généralement toutes les fonctionnalités *backend* d'une application mobile ou web. L'objectif est de se concentrer sur le *front-end* et améliorer l'expérience utilisateur des applications.
  - **Solutions :** Back4App, Firebase, Backendless, AWS Amplify, Kuzzle...
- **Container as a Service (CaaS) :** un peu inclassable : ni IaaS, ni PaaS, mais plutôt un moteur d'orchestration de conteneurs et de solutions cloud.
  - **Solution :** Kubernetes

Plan

L'intégration continue : les métiers

DevOps

Un ingénieur DevOps introduit des processus, des outils et des méthodologies pour **équilibrer les besoins** tout au long du cycle de vie du développement logiciel, du codage et du déploiement à la maintenance et aux mises à jour.

Principales tâches

- réduire la complexité de développement, en comblant le fossé entre les actions nécessaires pour modifier rapidement une application, et les tâches qui maintiennent sa fiabilité.
- fusionner les tâches quotidiennes liées au développement, au contrôle de la qualité, au déploiement et à l'intégration du développement logiciel en un ensemble unique et continu de processus.
- fait force de proposition de nouveau processus à mettre en place pour l'amélioration du produit.
- plus globalement, assure la qualité du développement pour le client.

L'intégration continue : les impacts

Impacts environnementaux « positifs »

La CI/CD dans un contexte de cloud computing :

- Le cloud computing est plus efficace et plus résilient que les capacités informatiques locales pour les particuliers et les entreprises.
- les applications logicielles couramment utilisées vers le cloud, diminuerait de 87% la consommation d'énergie (recherche de 2013, financé par Google...).
- facteur important pour le travail à distance (réduit le besoin de se déplacer et donc les émissions).

L'intégration continue : les impacts

Impacts environnementaux « négatifs » [1]

- Le refroidissement : 40% de la consommation totale d'énergie et jusqu'à 80% si le climat naturel du centre de données est plus chaud.
- Équipements souvent remplacés : 50 millions de tonnes métriques de déchets électroniques ont été générées dans le monde.
- produits chimiques de refroidissement utilisés / composants de ces batteries exploités de manière non durable.

À votre niveau :

- réduire au maximum la complexité du code.
- optimiser les processus CI/CD pour qu'ils soient lancés uniquement dans le cadre du besoin.
- utiliser les solutions de cloud computing répondant à votre besoin (éviter de sur-estimer son besoin et opter pour la scalabilité).

1. https ://earth.org/environmental-impact-of-cloud-computing/

L'intégration continue / déploiement continue

TP1 : Intégration continue, livraison, déploiement continue.

Plan

Tests unitaires vs fonctionnels

Test unitaire

test d'une unité individuelle, telle qu'une méthode (fonction) dans une classe, avec toutes les dépendances simulées.

Test fonctionnel

test d'une tranche de fonctionnalité dans un système. Ils s'assurent que le système fonctionne comme les utilisateurs s'y attendent.

La librairie Pytest

Un exemple simple :

```
import pytest

# content of test_sample.py
def func(x):
  return x + 1

def test_answer():
  assert func(3) == 5 # error !!
```

Un autre exemple :

```
import pytest

# content of test_sample.py
def pow(x, y):
   return x ** y

class TestFunc:

   def test_answer():
      assert func(3, 2) == 9
```

La librairie Pytest

Convention de tracking `pytest`

Pour que `pytest` puissent considérer vos tests unitaires, il faudra :

- Que le nom de votre classe commence par `TestXXXX`.
- Le nom de vos méthodes par `test_XXXX`.

La librairie Pytest

Définition d'une classe :

```
class Order:

def __init__(self):
    self._articles = []

def add_article(self, article):
    self._articles.append(article)

def remove(self, article):
    self._articles.remove(article)

@property
def articles(self):
    return self._articles
```

La librairie Pytest

Définition d'une classe :

```
class Order:

def __init__(self):
    self._articles = []

def add_article(self, article):
    self._articles.append(article)

def remove(self, article):
    self._articles.remove(article)

@property
def articles(self):
    return self._articles
```

Vers l'utilisation de fixture :

```
class TestOrder:

    @pytest.fixture
    def order(self):
        current_order = Order()
        current_order.add_article('Book1')
        current_order.add_article('Book2')
        return current_order

    def test_order(self, order):
        assert len(order.articles) >= 2

        order.remove('Book1')
        assert len(order.articles) < 2

    def test_order2(self, order):
        order.add_article('Book3')
        assert len(order.articles) >= 3
```

Le principe d'une fixture

Détermine des variables et un contexte de départ commun à tous les tests.

Les tests unitaires et fonctionnels

TP2 : Développement de la première version d'un simulateur de forêt.

Plan

Plan

Développement dirigé par les tests : définition

Test-driven development : TDD

Un processus où le développement de l'application sera fait en fonction d'un certain nombre de tests

Ce qui implique...

d'écrire tous les tests avant de commencer n'importe quel développement

Développement dirigé par les tests : les avantages

À première vue..

- Pas forcément très intuitif.
- Au final aucun test ne passer.

Développement dirigé par les tests : les avantages

À première vue..

- Pas forcément très intuitif.
- Au final aucun test ne passer.

Et pourtant..

- Demande davantage de réflexions en amont (notamment sur la conception).
- Rend le code plus simple.
- Code plus robuste.
- Éviter de développer des choses inutiles (on fixe les développements par rapport aux tests).

Si un test passe

$\rightarrow$ la fonctionnalité existe déjà, inutile de la développer de nouveau.

Développement dirigé par les tests : les avantages

Un développement plus long

- Un processus qui prend plus de temps.
- Besoin d'écrire les tests en amont $\rightarrow$ nécessite du temps.
- Vérifier que chaque test est ok après le développement.

Développement dirigé par les tests : les avantages

Un développement plus long

- Un processus qui prend plus de temps.
- Besoin d'écrire les tests en amont → nécessite du temps.
- Vérifier que chaque test est ok après le développement.

Mais au final...

- On est sûr de répondre aux tests demandés.
- Un processus itératif plus clair (valider les tests un par un).
- On évite les erreurs de copiers-collers d'autres tests....
- Si les tests sont bien écrits → moins de maintenance.
- Un code plus robuste et un gain de temps au final.

Développement dirigé par les tests : processus classique

Les différentes étapes

- Écrire les tests (unitaires et d'intégrations) ;
- Lancer tous les tests et s'assurer qu'ils sont en échec.
- Écrire le code nécessaire pour réussir un test.
- Relancer tous les tests pour vous assurer que seul celui-ci passe.
- Prendre un autre test et écrire le code nécessaire pour qu'il réussisse.
- Continuer ainsi jusqu'à ce que tous les tests réussissent.
- Factoriser le code.
- S'assurer de nouveau que tous les tests passent.

$\rightarrow$ la nouvelle fonctionnalité est implémentée et vérifiée.

Développement dirigé par les tests : un exemple (pytest)

Le besoin

Pouvoir ajouter un élément à une classe liste en Python. Un constructeur doit permettre d'initialiser la liste. On doit aussi pouvoir connaître le nombre d'éléments dans la liste à tout moment.

Développement dirigé par les tests : un exemple (pytest)

Le besoin

Pouvoir ajouter un élément à une classe liste en Python. Un constructeur doit permettre d'initialiser la liste. On doit aussi pouvoir connaître le nombre d'éléments dans la liste à tout moment.

Évaluation des besoins avant l'écriture du test :

- Constructeur avec paramètre.
- Méthode d'ajout d'élément en fin de liste.
- Méthode permettant de connaître la longueur

Développement dirigé par les tests : un exemple (pytest)

Le besoin

Pouvoir ajouter un élément à une classe liste en Python. Un constructeur doit permettre d'initialiser la liste. On doit aussi pouvoir connaître le nombre d'éléments dans la liste à tout moment.

Évaluation des besoins avant l'écriture du test :

- Constructeur avec paramètre.
- Méthode d'ajout d'élément en fin de liste.
- Méthode permettant de connaître la longueur

```python
def test_push_back():
  list_instance = MaListe([8, 9])

  assert list_instance.length() == 2, "length of 2"

  list_instance.push_back(10)

  assert list_instance.length() == 3, "length of 3"
  assert list_instance.last() == 10, "Should be 10"
```

Développement dirigé par les tests

TP3 : Application du TDD pour le simulateur de forêt : gestion des branches.
TP4 : Application du TDD pour le simulateur de forêt : gestion des graines.

Plan

Développement guidé par le comportement

Behavioral Driven Development

- Processus de développement agile logiciel
- Plus concrètement : technique combinée du Test Driven Development (Technique) et du Domain Driven Development (Business)
- Utilise des langages non technique (Gherkin langage)
- Encourage l'utilisation de la conversation et d'exemples concrets dans un langage simple

Développement guidé par le comportement

Behavioral Driven Development

- Processus de développement agile logiciel
- Plus concrètement : technique combinée du Test Driven Development (Technique) et du Domain Driven Development (Business)
- Utilise des langages non technique (Gherkin langage)
- Encourage l'utilisation de la conversation et d'exemples concrets dans un langage simple

Pourquoi cette nouvelle approche ?

- Permet un lien plus évident entre le technique et business
- Plus axée sur le comportement du système plutôt que sur l'implémentation du code

Développement guidé par le comportement

Les avantages

- Apporte une meilleure clarté de ce qui est à développer
- Implique les différents membres du projet : client, développeur, testeur...
- Conversation collaborative et illustration du comportement du système
- Permet d'éviter les fonctionnalités inutiles et d'inclure les fonctionnalités importantes
- Réduit l'effort pour les défauts post-modification et post-déploiement.
- Évite les erreurs d'interprétation pendant le processus de développement

Développement guidé par le comportement

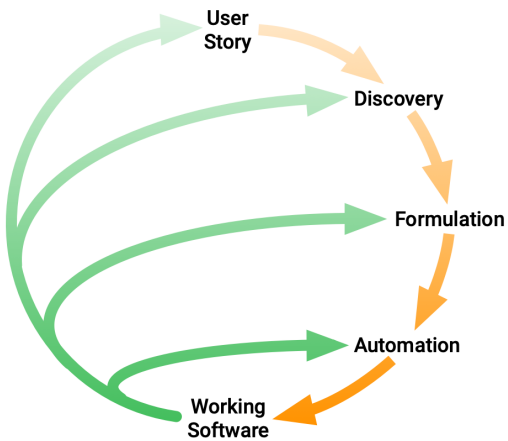
La démarche du BDD

- Le client et le prestataire de services ont une conversation sur le besoin
- Le client, le développeur et le testeur élaborent ensemble les exigences de ce besoin
- Les exigences sont définies dans des scénarios structurés
- Les scénarios aident au développement et servent de tests automatisés
- Ces scénarios sont utilisés par les testeurs comme base des tests

Développement guidé par le comportement

Processus itératif basé « User story »

1. **[Discovery]** Définir un changement à venir dans le système :
   → une « User Story » avec exemples concrets de la nouvelle fonctionnalité
2. **[Formulation]** Documenter ces exemples d'une manière qui peut être automatisée
   → Vérifiez l'accord
3. **[Automation]** Mettre en œuvre le comportement décrit par chaque exemple documenté
   → test automatisé pour guider le développement du code



https ://cucumber.io/docs/bdd/

Développement guidé par le comportement

Définition d'un scénario

Étant donné que la plateforme Pokemon Go est accessible
Quand je me connecte sur la plateforme
Et que j'attaque une arène
Et que je gagne mon combat
Alors j'occupe maintenant cette arène

Développement guidé par le comportement

Les outils existants

Tous principalement basés sur `cucumber.io` [1] :

- Un fichier `features/XXXX.feature` pour exprimer le scénario en langage Gherkin
- Un fichier `features/steps/XXXX_steps.py` (si Python par exemple), pour décrire les étapes

1. https://cucumber.io/docs/installation

Développement guidé par le comportement

Les outils existants

Tous principalement basés sur `cucumber.io` [1] :

- Un fichier `features/XXXX.feature` pour exprimer le scénario en langage Gherkin
- Un fichier `features/steps/XXXX_steps.py` (si Python par exemple), pour décrire les étapes

Exemple de définition du comportement (business part) :

```
# -- FILE: features/search.feature
Feature: Search
   Scenario: Search PyPI
      Given I navigate to the PyPi Home page
      When I search for "behave"
      Then I find a link to project "behave"
```

1. https://cucumber.io/docs/installation

Développement guidé par le comportement

Automatisation du scénario :

```python
from behave import given, when, then, step
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys

@given('I navigate to the PyPi Home page')
def step_impl(context):
    context.browser.get("https://pypi.org")
    assert "PyPI" in context.browser.title

@when('I search for "{search_term}"')
def step_impl(context, search_term):
    context.browser.find_element(By.ID, "search").send_keys(search_term)
    context.browser.find_element(By.ID, "search").send_keys(Keys.ENTER)

@then('I find a link to project "{search_result}"')
def step_impl(context, search_result):
    selector = f'a[href="/project/{search_result}/"]'
    assert context.browser.find_element(By.CSS_SELECTOR, selector)
```

Développement guidé par le comportement

Différents scénarios avec contexte commun :

```
# -- FILE: features/search.feature
Feature: Search
   Backgound:
      Given I navigate to the PyPi Home page
   Scenario: Search PyPI
      When I search for "behave"
      Then I find a link to project "behave"
   Scenario: Search PyPI
      When I search for "tensorflow"
      Then I find a link to project "tensorflow"
```

Behavioral Driven Development

TP7 : BDD avec behave (Python).

Plan

Métrique et mesure de qualité de code

Quelques définitions

- mesure indication quantitative de l'étendue, quantité, dimension, capacité ou taille d'un attribut de produit
- métrique (IEEE Std. 610.12) :
  - mesure quantitative du degré avec lequel un système, composant, ou processus possède un attribut donné
  - peut lier des mesures individuelles
- indicateur : métrique(s) donnant une indication sur produit (projet)

Métrique et mesure de qualité de code

Pourquoi mesurer ?

- Obtenir des indicateurs à jour de l'état du projet et de l'avancement du projet.
- Identifier les points bloquants de développements.
- Amélioration du processus de maintenance.
- Évaluation de la performance du produit et de l'équipe dans son développement.

Métrique et mesure de qualité de code

Pourquoi mesurer ?

- Obtenir des indicateurs à jour de l'état du projet et de l'avancement du projet.
- Identifier les points bloquants de développements.
- Amélioration du processus de maintenance.
- Évaluation de la performance du produit et de l'équipe dans son développement.

Métriques de produit

Indicateurs de produit : complexité, volume et caractéristiques de la conception

Métrique et mesure de qualité de code : métriques produit

Complexité d'Halstead

qui procurent une mesure quantitative de complexité pour des logiciels (Maurice Halstead)

- Basée sur le nombre d'opérateurs / opérandes et de la taille du vocabulaire

Mesures de la taille (volume) du logiciel

KLOC - mesure classique de la taille du logiciel en millier de lignes de code

Métrique et mesure de qualité de code : métriques produit

CBO : Coupling Between Object classes

Mesure le nombre de classes couplées

- méthodes déclarées dans l'une utilisent des méthodes de l'autre ou instancie des variables définies dans l'autre
- si la classe A est couplée à B alors B est couplée à A (symétrie).

WMC : Weighted Methods per Class

- Calculée sur un ensemble de n classes contenant chacune M_i méthodes

$$WMC = \frac{1}{n} \sum_{i=0}^{n} c_i M_i$$

- avec c_i est la complexité de la méthode
- Peut ne prendre en compte que les méthodes publiques

Métrique et mesure de qualité de code : métriques produit

DIT (Depth of Inheritance Tree)

- Distance maximale entre un nœud (une classe) et la racine de l'arbre d'héritage de la classe concernée
- Complexité dans l'arbre d'héritage
- Cette distance est calculée pour toutes les classes

NOC : Number of children

- Nombre d'enfants dans l'arbre d'héritage par rapport à une classe donnée
- $\rightarrow$ Mesure du niveau de factorisation : perte de généralisation

Métrique et mesure de qualité de code : métriques produit

Complexité cyclomatique : mesure de McCabe

Elle mesure le nombre de décisions d'un algorithme en comptabilisant le nombre de « chemins » linéairement indépendants (à partir d'un graphe)

- On produit un graphe de contrôle qui représente un code ou on calcule le nombre de points de décisions (if, for, while, case, ...)
- Elle est définie par :

$$M = E - N + 2P$$

avec :

- M = complexité cyclomatique.
- E = le nombre d'arêtes du graphe.
- N = le nombre de nœuds du graphe.
- P = le nombre de composantes connexes du graphe (un sous-graphe connexe du graphe).

Métrique et mesure de qualité de code : l'outil Pylint

La librairie pylint

- Analyse le code sans l'exécuter.
- Vérifie les erreurs, applique une norme de codage, recherche le code de moins bonne qualité.
- Fait des suggestions sur la façon dont le code pourrait être remanié.
- Intègre des mesures de complexité, dont celle de McCabe.

`Pylint` possède des codes spécifiques pour chaque message :

```
(C) convention, for programming standard violation
(R) refactor, for bad code smell
(W) warning, for python specific problems
(E) error, for probable bugs in the code
(F) fatal, if an error occurred which prevented pylint to stop.
```

Qualité de code et mesure

TP4 : Application du TDD et refactoring pour le simulateur de forêt.

Plan

Test d'interfaces web et web services

Pourquoi tester son interface web ?

- Globalement : vérifier que le contenu est tel qu'attendu.
- Vérifier qu'une procédure (histoire) se réalise intégralement.
  $\rightarrow$ par exemple : un tunnel de commande.
- Vérifier que les données d'API concordent avec l'interaction réalisée.
- Vérifier la robustesse de l'application (montée en charge).
- ...

Objectifs de tels outils

- Le processus de vérification d'une application peut-être lourd et redondant.
- Cherchent à répondre aux principales difficultés rencontrées par les développeurs et les ingénieurs d'assurance qualité lorsqu'ils testent des applications modernes.

Test d'interfaces web et web services : Cypress

La librairie Cypress

- Librairie Javascript.
- Conçue à l'origine pour exécuter des tests *End to end* (E2E) sur tout ce qui s'exécute dans un navigateur.
- Permet de tester sur plusieurs navigateurs.
- Intègre également des tests d'intégrations et unitaires.

Test d'interfaces web et web services : Cypress

La librairie Cypress

- Librairie Javascript.
- Conçue à l'origine pour exécuter des tests *End to end* (E2E) sur tout ce qui s'exécute dans un navigateur.
- Permet de tester sur plusieurs navigateurs.
- Intègre également des tests d'intégrations et unitaires.

Description d'un test E2E

Un test E2E typique visite l'application dans un navigateur et effectue des actions via l'interface utilisateur, comme le ferait un véritable utilisateur.

Test d'interfaces web et web services : Cypress

Les principaux concepts

- `visit` : accède à une page en question.
- `get` : se base sur les sélecteurs CSS pour accéder aux éléments.
- `contains` / `should` / `expect` : vérifie les propriétés des éléments du DOM.
- `invoke` : invoque une fonction javascript liée à un élément.
- `trigger` : déclenche un événement d'un élément.
- ...

Test d'interfaces web et web services : Cypress

Un premier exemple :

```
describe('Story 1', () => {

   it('Check home page', () => {
      cy.visit('/home')
      cy.wait(500) // add some delay

      cy.get('h1').contains('This is a home page') // first test
   })

   it('Check presentation page', () => {
      cy.visit('/home')
      cy.wait(500) // add some delay

      cy.get('button.btn-primary').click()
      cy.wait(500)

      cy.url().should('include', '/presentation')
   })
})
```

Test d'interfaces web et web services : Cypress

Un autre exemple :

```
describe('Story 2', () => {

   it('Check invoke', () => {
      cy.visit('/home')
      cy.wait(500) // add some delay

      cy.get('div.container')
         .should('be.hidden') // element is hidden
         .invoke('show')
         .should('be.visible') // element is visible now
   })

   it('Check trigger', () => {
      cy.visit('/home')
      cy.wait(500) // add some delay

      cy.get('input[type=range]').as('range')
      cy.get('@range').invoke('val', 5).trigger('change')

      cy.get('@range').should('have.value', 5)
   })
})
```

Test d'interfaces web et web services

TP5 : Test d'interfaces web et API du simulateur de forêt.

Plan

Conception par contrat (*design by contract*)

Définition

Paradigme de programmation dans lequel le déroulement des traitements est **régi par des règles**.

Objectifs

Ces règles, appelées des assertions, forment un **contrat** qui précise les responsabilités entre le client et le fournisseur d'un morceau de code logiciel.

- réduire le temps consacré au débogage du code, en détectant les erreurs au plus tôt dans le cycle de développement.
- faciliter la détection des erreurs.
- augmenter la réutilisation de code, en s'assurant qu'un code correct dans un environnement le sera aussi dans un nouvel environnement.

Conception par contrat (*design by contract*)

Bertrand Meyer

Eiffel

Langage Eiffel

Création du concept et motivation du langage (1985)

- Inspiré de la notation Z créée par Jean-Raymond Abrial.
- Objectif : organiser la communication entre les composants d'un logiciel.
- Spécifier les conditions des échanges directement dans le code.
- Vérifier à l'exécution le respect des règles et conditions.
- Contrairement aux assertions dans la plupart des langages, le contrat est séparé de la logique métier.

Conception par contrat (*design by contract*)

Paradigme basé sur 4 types d'assertions

- **Précondition :** doit être vérifiée par le client avant le lancement d'un traitement donné.

  → doit garantir une exécution du traitement sans erreur.
- **Postcondition :** doit être garantie par le fournisseur après le déroulement d'un traitement donné.

  → doit garantir que le traitement a bien réalisé son travail.
- **Invariant :** condition qui est toujours vraie.

  → peut caractériser l'état interne de tout le programme, ou seulement d'une.
- **Variant :** valeur numérique entière positive associée à une boucle.

  → doit nécessairement décroître à chaque évaluation, garantissant la terminaison.

Conception par contrat (*design by contract*)

Inclusion du paradigme

- **Langages de programmation :** Eiffel, D, Lisaac, Oxygene, SPARK, Vala ainsi qu'Ada (2012).
- **Mais aussi des modules :** OCL pour UML, JML pour Java, ACSL pour le C, Praspel pour PHP ou PSL pour VHDL.

Comment ?

- Basé sur une logique : utilise en général les expressions booléennes.
- Ajout d'un moyen pour que les postconditions puissent se référer à l'ancienne valeur des variables modifiées par le traitement.
- Assertions écrites directement dans le code source du programme.

  $\rightarrow$ permettent de fournir une documentation supplémentaire sur le sens que possède le code
- On peut y rajouter les quantificateurs de la logique du premier ordre

  $\rightarrow$ Exemple : $\forall x \exists y (x < y)$

Conception par contrat : un exemple

Fonction racine carrée de x

- Précondition : $x \geq 0$
- Postcondition : $résultat \geq 0$ et $résultat^2 = x$

Conception par contrat : un exemple

Fonction racine carrée de x

- Précondition : $x \geq 0$
- Postcondition : $résultat \geq 0$ et $résultat^2 = x$

L'exemple en Python :

```python
# L'exemple avec la librairie `deal`
import deal
import math

@deal.pre(lambda x: x > 0)
@deal.ensure(lambda x, result: x == result * result)
def racine_carree(x):
   return math.sqrt(x)

racine_carree(9)
```

Conception par contrat : librairie deal Python

@deal.pre

Condition qui doit être vraie avant l'exécution de la fonction.

@deal.post

Condition qui doit être vraie après l'exécution de la fonction.

@deal.ensure

Postcondition qui accepte non seulement le résultat, mais aussi les arguments de la fonction. Elle doit être vraie après l'exécution de la fonction.

@deal.inv

Condition dont on peut être sûr qu'elle est vraie pendant l'exécution d'un programme (peut-être affectée à une classe).

Conception par contrat : librairie deal Python

@deal.pre

```
import random

@deal.pre(lambda x, y: x < 0 and y < 0)
@deal.pre(lambda x, y: x < y)
def random_negative(x , y):
   return random.randint(x, y)
```

@deal.post

```
@deal.pre(lambda n: n > 0)
@deal.post(lambda result: len(result) > 0)
def ones(n):
   return [ 1 for _ in range(n)]
```

@deal.ensure

```
@deal.ensure(lambda x, result: x < result)
def double(x):
   return x * 2
```

Conception par contrat : librairie deal Python

@deal.inv

```
@deal.inv(lambda post: post.likes >= 0)
class Post:
   likes = 0

post = Post()
post.likes = 10
post.likes = -10 # InvContractError
```

Condition de vérification invariante dans les cas suivants :

- Avant l'exécution de la méthode de classe.
- Après l'exécution de la méthode de la classe.
- Après le paramétrage d'un attribut de classe.

Conception par contrat : librairie deal Python

Remarques importantes :

- Par défaut à l'exécution d'un programme les contrats sont vérifiés.
- Le programme en environnement de production se doit d'être performant. Les contrats devront être vérifiés sur un autre environnement.
- En production, il est possible de les désactiver avec l'option « -O » :
  $\rightarrow$ python -O main.py

Conception par contrat : librairie deal Python

Fonction à vérifier (`myrandom.py`) :

```
import random

@deal.pre(lambda x, y: x < 0 and y < 0)
@deal.pre(lambda x, y: x < y)
def random_negative(x: int , y: int) -> int:
   return random.randint(x, y)
```

Vérification avec Pytest (`test.py`) :

```
from myrandom import random_negative

# simule la fonction avec plusieurs paramètres
test_random_50 = deal.cases(random_negative)
# 20 de données aléatoires, par défaut 50
test_random_20 = deal.cases(random_negative, count=20)
```

Utilisation des indicateurs de typage Python (*Python hint*)

Il est nécessaire que la librairie « `deal` » sache le type de données d'entrée des fonctions pour simuler les jeux de données aléatoires. Python permet d'ajouter des indicateurs de typage de données et de sortie d'une fonction.

Conception par contrat : librairie deal Python

Exemple avec des méthodes :

```python
import deal
from typing import List

class MyList:

    def __init__(self) -> None:
        self._elements = []

    @deal.pre(lambda *_, x: x > 0)
    @deal.ensure(lambda _: _['x'] in _['self']._elements)
    def add(self, x: int) -> None:
        self._elements.append(x)

    @deal.pre(lambda *_, mul: mul > -100 and mul < 100) # set default interval
    @deal.ensure(lambda _: sum(_['self']._elements * _['mul']) == sum(_['result']))
    def mult(self, mul: int) -> List[int]:
        return self._elements * mul

test = MyList()
test_mult = deal.cases(test.mult, count=30) # check mult method
```

Quelques spécificités

- Impossibilité d'accéder à `self` dans une précondition.
- Accès possible à `self` en postcondition via `@deal.ensure`.
- Besoin de borner des valeurs dans ce cas pour éviter un nombre en dehors des capacités.

Conception par contrat : librairie deal Python

L'avantage et l'inconvénient de `deal.cases`

- Augmente la robustesse de chaque test par un grand nombre de jeux de données.
- N'est pas une preuve réelle du contrat attendu...

Vers la notion de preuve

- permet de vérifier l'expression et d'en certifier la preuve (qu'importe les données d'entrée).
- `deal` propose la vérification par preuve mais celle-ci reste expérimentale.
  → `python -m deal prove`

Conception par contrat

TP6 : Conception par contrat pour l'ajout de fonctionnalités au simulateur de forêt.

Plan

Plan

Méthodes agiles

On parle aussi de : pratiques agiles

- Mettent en avant la collaboration entre des équipes auto-organisées et pluridisciplinaires et leurs clients
- Basées sur un cadre méthodologique léger mais suffisant centré sur l'humain et la communication

Principaux objectifs

- Une planification adaptative
- Un développement évolutif
- Une livraison précoce et une amélioration continue
- Apporter des réponses flexibles au changement

Méthodes agiles : l'historique

Vers des processus itératifs

1940 Définition du cycle de production itératif et incrémental avec les travaux de Walter Shewhart et William Edwards Deming

1950 Cycle de production itératif et incrémental : appliquées à la production informatique à la fin des années

1957 Projet de développement réalisé pour Motorola (technique similaire à ce qui sera plus tard l'**eXtreme programming**)

1981 Production en itérations rapides : Scott Shultz

1987 Barry Boehm publie le modèle en spirale (développement incrémental)

Apparition du terme agile

1991 Le terme agile apparait dans le monde économique anglophone

Méthodes agiles : le manifeste

Principe du manifeste

- Texte rédigé aux États-Unis en 2001 par dix-sept experts du développement logiciels.
- Composé de 4 valeurs et douze principes sous-jacents
- Basés sur les méthodes Scrum et eXtreme Programming

→ Tronc commun des méthodes agiles

Méthodes agiles : le manifeste

Principe du manifeste

- Texte rédigé aux États-Unis en 2001 par dix-sept experts du développement logiciels.
- Composé de 4 valeurs et douze principes sous-jacents
- Basés sur les méthodes Scrum et eXtreme Programming

→ Tronc commun des méthodes agiles

Manifeste Agile

- **Les individus et leurs interactions** plus que les processus et les outils
- **Des logiciels opérationnels** plus qu'une documentation exhaustive
- **La collaboration avec les clients** plus que la négociation contractuel
- **L'adaptation au changement** plus que le suivi d'un plan

Méthodes agiles : le manifeste

Le manifeste

- **Faire face aux idées reçues :** « Sur un projet agile, il n'y a pas de spécifications, de plan, de processus, d'outil et même pas de contrat. »
- **Au contraire :** mise en place de bonnes pratiques pour un développement évolutif.
- Pas uniquement réservé aux projets de développements logiciels.

Méthodes agiles : le manifeste

Principes sous-jacents

1. Notre plus haute priorité est de satisfaire le client en livrant rapidement et régulièrement des fonctionnalités à grande valeur ajoutée.
2. Accueillir positivement les changements de besoins, même tard dans le projet. Les processus Agiles exploitent le changement pour donner un avantage compétitif au client.
3. Livrer fréquemment un logiciel opérationnel avec des cycles de quelques semaines à quelques mois et une préférence pour les plus courts.
4. Les utilisateurs ou leurs représentants et les développeurs doivent travailler ensemble quotidiennement tout au long du projet.
5. Réaliser les projets avec des personnes motivées. Fournissez-leur l'environnement et le soutien dont ils ont besoin et faites-leur confiance pour atteindre les objectifs fixés.
6. La méthode la plus simple et la plus efficace pour transmettre de l'information à l'équipe de développement et à l'intérieur de celle-ci est le dialogue en face à face.

Méthodes agiles : le manifeste

Principes sous-jacents (suite)

7. Un logiciel opérationnel est la principale mesure d'avancement.
8. Les processus agiles encouragent un rythme de développement soutenable. Ensemble, les commanditaires, les développeurs et les utilisateurs devraient être capables de maintenir indéfiniment un rythme constant.
9. Une attention continue à l'excellence technique et à une bonne conception renforce l'agilité.
10. La simplicité – c'est-à-dire l'art de minimiser la quantité de travail inutile – est essentielle.
11. Les meilleures architectures, spécifications et conceptions émergent d'équipes auto-organisées.
12. À intervalles réguliers, l'équipe réfléchit aux moyens de devenir plus efficace, puis règle et modifie son comportement en conséquence.

Plan

Méthodes agiles : les plus connues

Liste non-exhaustive de méthodes agiles

1940 La méthode kanban issue de la lean production (Toyota).

1991 Développement rapide d'application, dite méthode RAD (*rapid application development*).

1995 Dynamic systems development method (DSDM).

1995 Le framework Scrum par Ken Schwaber, publié ensuite en 2010 par lui-même et Jeff Sutherland (*Guide Scrum*).

1999 eXtreme Programming (XP) par Kent Beck.

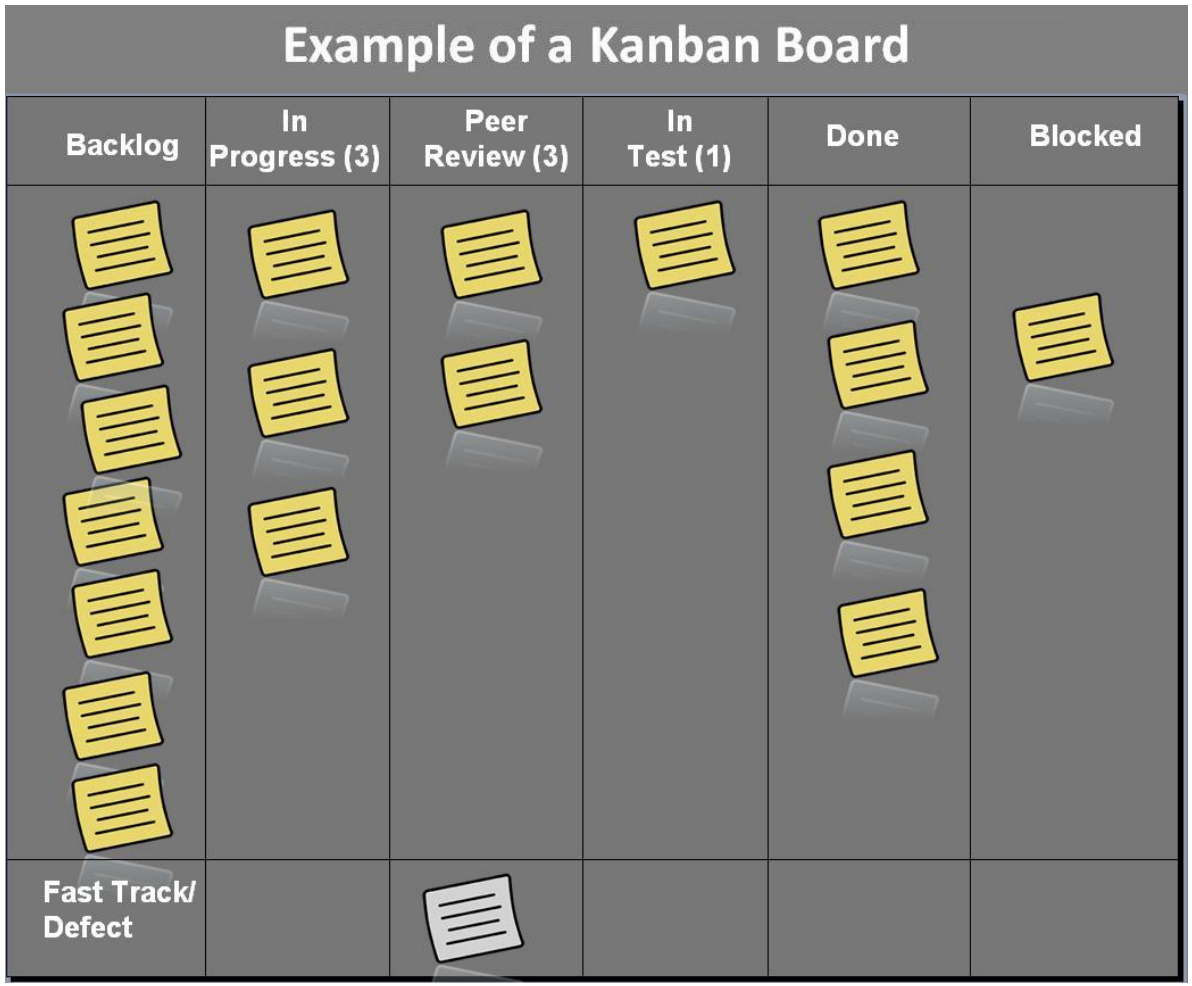
2000 Adaptive software development (ASD) avec une parenté avec RAD.

2003 Développement basé sur les fonctionnalités (FDD pour feature driven development).

2003 Behavior-driven development (BDD).

2004 Crystal clear publiée par Alistair Cockburn.

Méthodes agiles : Kanban



Source : https://en.wikipedia.org/wiki/Kanban_(development)

Méthodes agiles : eXtreme programming (XP)

Méthode définie par

- Une tentative de réconcilier l'humain avec la productivité
- Un mécanisme pour faciliter le changement social

L'objectif

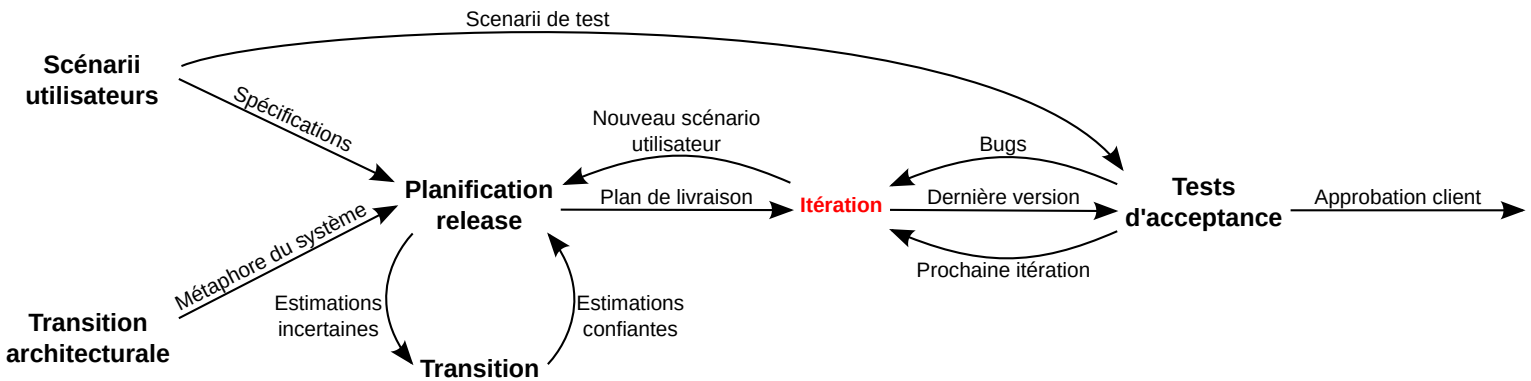
- But principal est de réduire les coûts du changement.
- Différent des méthodes traditionnelles : besoins sont définis et souvent fixés au départ du projet (ce qui accroît les coûts ultérieurs de modifications).
- Rendre le projet plus flexible et ouvert au changement en introduisant des valeurs de base, des principes et des pratiques.

Méthodes agiles : eXtrême programming (XP)

Pousser à l'extrême :

- La revue de code faite en permanence (par un binôme).
- Les tests faits systématiquement avant chaque mise en production.
- Le code retravaillé tout au long du projet (refactoring).
- La solution la plus simple sera toujours celle qui sera retenue.
- Des métaphores seront définies et évolueront en concomitance.
- Les modifications faites plusieurs fois par jour.
- Des cycles de développement très rapides pour l'adaptation au changement.

Méthodes agiles : eXtrême programming (XP)



Source : https://fr.wikipedia.org/wiki/Extreme_programming

Méthodes agiles : Scrum

La méthode Scrum et ses objectifs

- Framework de développement
- Développement itératif
- Méthode considérée comme un groupe de pratiques répondant pour la plupart aux préconisations du manifeste agile.
- Découpage d'un projet en « boîtes de temps »

Les 3 pilliers de Scrum

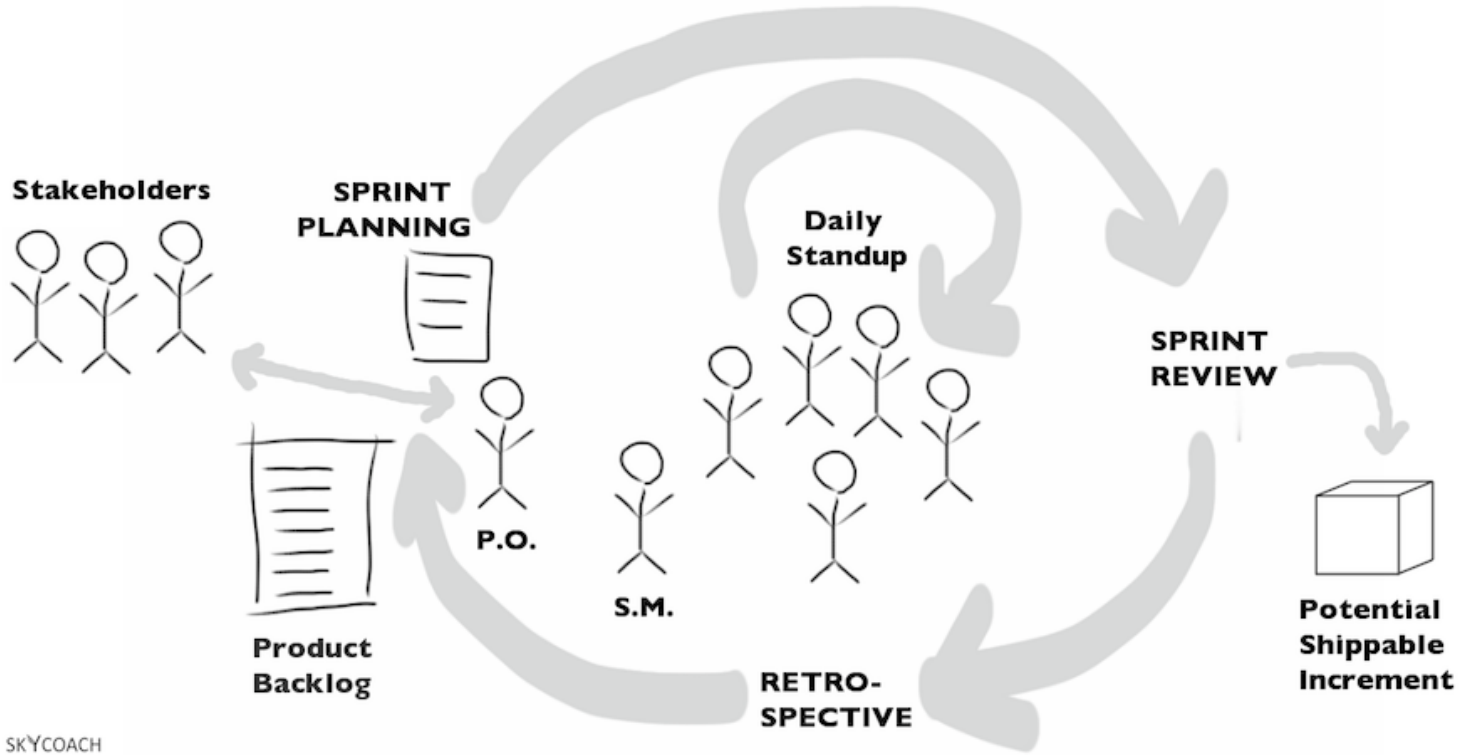
- La transparence
- L'inspection
- L'adaptation

Méthodes agiles : Scrum

Le glossaire

- **Product owner :** personne ayant la responsabilité de produire et de maintenir à jour le carnet de produit.
- **Scrum master :** Membre de l'équipe dont l'objectif principal est de la protéger des perturbations extérieures. Doit être transparent dans la communication entre le client et l'équipe.
- **Product backlog :** carnet de produit comprenant l'ensemble des fonctionnalités, améliorations, corrections attendues.
- **Definition of done :** ensemble de conditions nécessaires pour considérer qu'un élément de carnet de produit est livrable.
- **Sprint :** nom d'une itération dans Scrum (entre 2 et 4 semaines).
- **Sprint backlog :** liste des tâches à accomplir pendant un sprint.
- **Daily scrum :** réunion quotidienne de quinze minutes maximum pour faire le point sur l'avancée avec l'équipe.
- **Burndown chart :** graphique d'avancement qui représente l'évolution du reste à faire total de jour en jour ou de sprint en sprint.

Méthodes agiles : Scrum



Source : https://kmoscrum.wordpress.com/

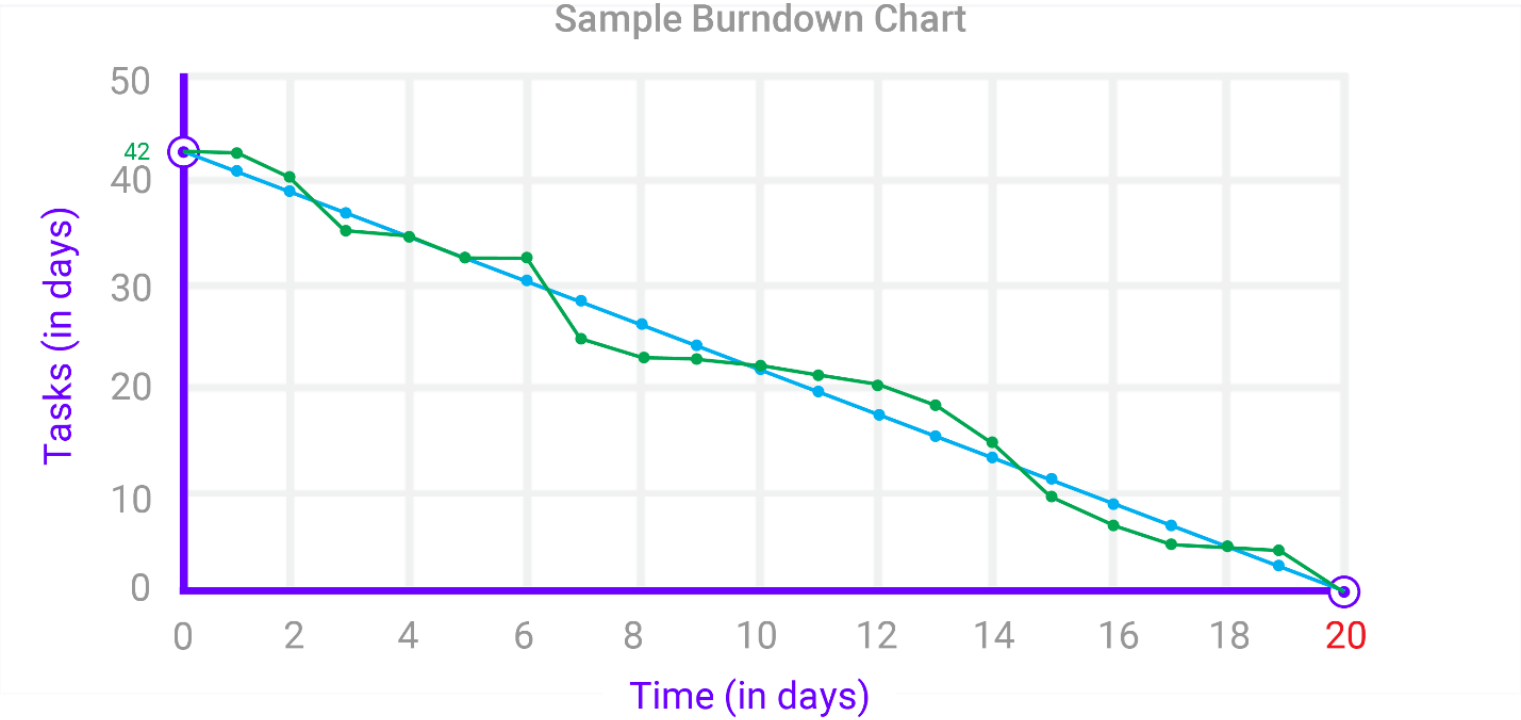
Plan

Méthodes agiles : pratiques agiles

Liste non-exhaustive de pratiques agiles

- Intégration continue
- Test driven development (TDD)
- Conception pilotée par le domaine (DDD pour domain-driven design)
- Burndown chart
- Programmation en binôme (pair-programming)
- Planning poker
- Réusinage de code
- Timeboxing : allouer à la réalisation d'une activité donnée une durée fixe.
- Récit utilisateur (User story)
- Modélisation C4 : élaboration collaborative d'une architecture évolutive ou émergente

Méthodes agiles : Burndown chart



Source : https://elearningindustry.com

Méthodes agiles : planning poker

L'importance de mesurer la complexité d'une tâche

- Estimer l'effort de développement de fonctionnalités
- Tout le monde peut s'exprimer librement
- Participants avec des niveaux d'expérience et d'expertise différents
- Favorise les échanges entre le responsable de produits et l'équipe de développement

Source : https://www.javierholguera.com

Méthodes agiles : planning poker

Déroulement

- Les participants s'installent autour d'une table.
- Le responsable de produit explique à l'équipe un scénario utilisateur.
- Les participants posent des questions au responsable de produit et évoquent les conditions de satisfaction qui permettront de le considérer comme "terminé".
- Chacun des participants évalue l'effort de développement de ce scénario, choisit la carte qui correspond à son estimation et la dépose face cachée.
- Au signal du facilitateur, les cartes sont retournées en même temps.
- S'il n'y a pas unanimité, la discussion reprend.
- On répète le processus d'estimation jusqu'à l'obtention de l'unanimité.