Qualité logicielle

Millian Poquet

2021-01-06

Intervenants

Millian Poquet

- `millian.poquet@univ-grenoble-alpes.fr`
  `millian.poquet@inria.fr`
- Ingénieur de recherche à l'Inria
- Développement/maintenance logicielle (toute la chaîne)
- Domaine : systèmes distribués, simulation, gestion de ressources
- Études : DUT, licence, master, doctorat
- Enseignant depuis 2010 (algo, programmation, big data...)

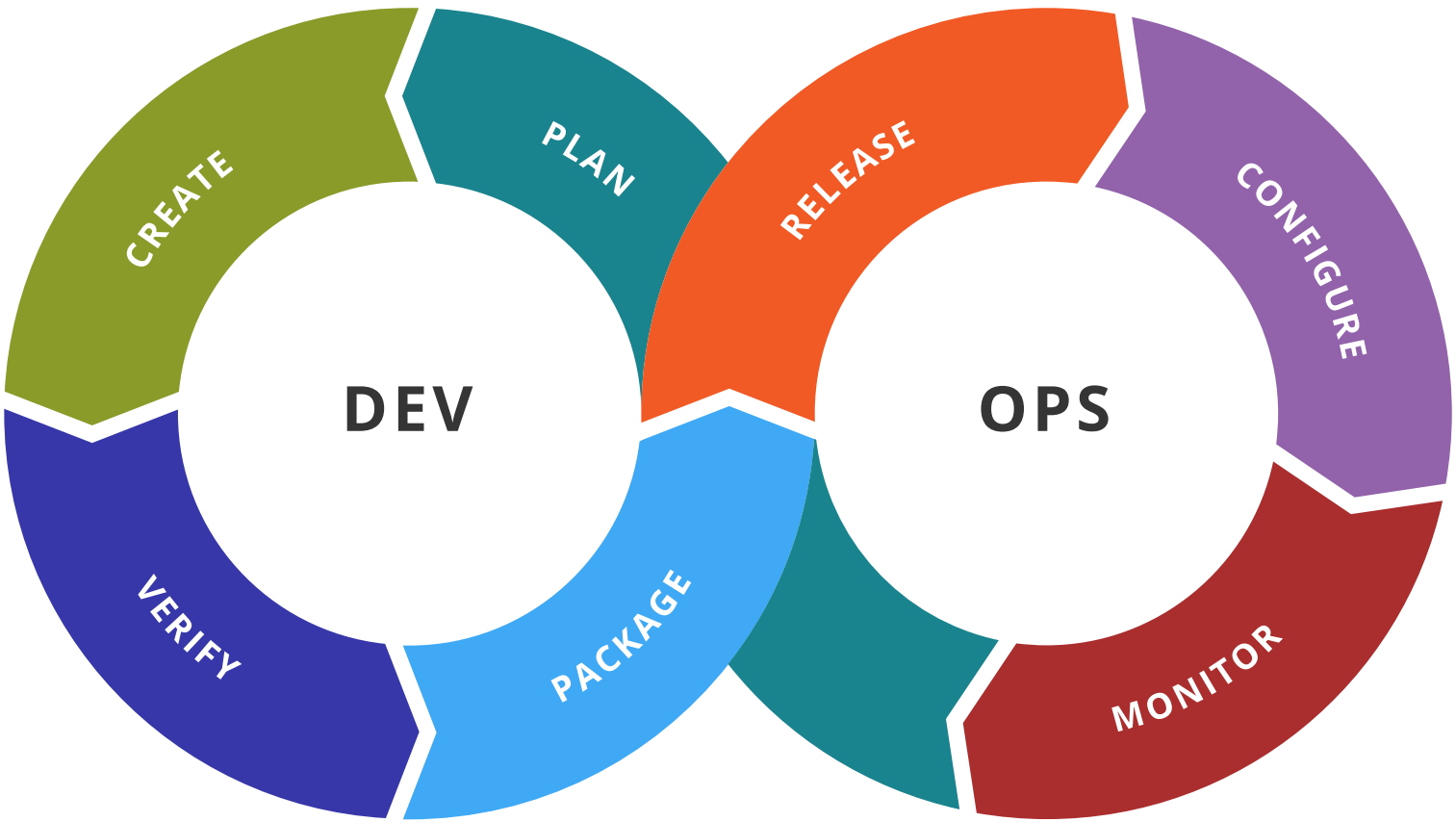
David Beniamine

- Il se présentera lui-même :)

Historique de ce cours

2019-2020 : DevOps

- Difficile sans concepts/expérience de qualité logicielle
- $\rightarrow$ Besoin de plus d'heures et d'un cours à part



Objectifs

- Concepts
- Technologies
- Autonomie
- Critique

Organisation

Planning

- 2020-12-07 (8h) : Gestion de version
- 2021-01-04 (6h) : Test et intégration continue
- 2021-01-18 (4h) : Projet
- 2021-02-01 (6h) : Projet

Comment se passent les séances ?

- À distance, salle centrale sur BBB
- Cours en vidéo (ou en direct sur BBB)
- Centré sur la pratique (TP/projet)

Évaluation

- Projet
  - Développement nouveau logiciel ou amélioration d'un existant
  - En groupe (2 ou 3 personnes)
  - Soutenance
- (Rendu de certains TP)
- (Rendu d'exercices de cours)

Définition ?

Dur à définir ! Problème multiobjectif...

Tentatives de normalisation — *e.g.*, ISO/CEI 9126

- Capacité fonctionnelle : répond aux besoins fonctionnels ?
- Fiabilité : pannes ?
- Utilisabilité : requiert peu d'effort à l'utilisation ?
- Efficacité : performances ?
- Maintenabilité : requiert peu d'effort à son évolution ?
- Portabilité : transférable dans un autre environnement ?

Approche de ce cours

Pragmatique, pas théorique

- Quels outils sont essentiels pour une bonne qualité ?
- Quelles méthodes de travail améliorent la qualité ?

Aspects traités

- Contrôle de version
- Contrôle d'environnement
- Méthodes de travail
- Test
- Automatisation

Voir sous-cours dédié.

Définition

Logiciel qui regroupe outils et services pour le développement.

Outils courants.

- Serveur de gestion de version
- Bug tracker
- Espace de discussion (vue à long terme, feature requests. . .)
- Pages web / wiki
- Exécution automatique de tests
- Suivi de métriques du projet

GitLab

La forge que l'on va utiliser.

- Basé sur Git
- Interface web (clicodrome ou REST)
- Concurrent à GitHub depuis 2014
- Version communautaire libre (MIT)
- Architecture distribuée modulaire

Non centralisé. Beaucoup d'instances (la vôtre ?)

GitLab CI

Système d'exécution de code quand le dépôt est modifié.

- Déclencement configurable. Par exemple :
  - Sur chaque commit de la branche de développement
  - Chaque nuit sur la branche de développement
  - Chaque dimanche sur la branche de la dernière *release* stable
  - Quand on souhaite intégrer une nouvelle *feature*
- Environnement configurable dans chaque situation
  - Via l'utilisation de conteneurs (Docker)
  - Indirectement via la sélection de machines pré configurées
- On peut connecter ses propres *runners* au serveur GitLab

Peut-on prouver qu'un logicielle *marche* ?

Dans l'idéal, on voudrait garantir qu'un logiciel marche [1].

Malheureusement, c'est impossible automatiquement (cf. Théorème de Rice).

1. Définir ce qu'on entend par *marcher* se spécifie souvent projet par projet.

Peut-on prouver qu'un logicielle *marche* ?

Dans l'idéal, on voudrait garantir qu'un logiciel marche [1].

Malheureusement, c'est impossible automatiquement (cf. Théorème de Rice).

On peut par contre étudier un logiciel donné en **prouvant** certaines propriétés dessus (cf. Méthodes formelles). Ces méthodes sont robustes, elles ont été utilisées sur certains systèmes critiques (*e.g.*, contrôle de la ligne 14 du métro parisien ou de processeurs).

1. Définir ce qu'on entend par *marcher* se spécifie souvent projet par projet.

Peut-on prouver qu'un logicielle *marche* ?

Dans l'idéal, on voudrait garantir qu'un logiciel marche [1].

Malheureusement, c'est impossible automatiquement (cf. Théorème de Rice).

On peut par contre étudier un logiciel donné en **prouvant** certaines propriétés dessus (cf. Méthodes formelles). Ces méthodes sont robustes, elles ont été utilisées sur certains systèmes critiques (*e.g.*, contrôle de la ligne 14 du métro parisien ou de processeurs).

On peut aussi modéliser un système puis **vérifier** que certaines propriétés sont garanties dans ce modèle (cf. Model checking). En pratique, on explore un graphe (gigantesque) de scénarios de l'automate fini du système en vérifiant que des formules écrites en logique temporelle restent vraies.

1. Définir ce qu'on entend par *marcher* se spécifie souvent projet par projet.

Que sont les tests dans tout ça ?

Une approche **simple** pour de **faibles** garanties.

Principe

- Exécuter le code réel sur des scénarios bien définis.
- Vérifier que le comportement observé est celui attendu.

Exemples

- Les valeurs retournées d'une fonction sont les bonnes ?
- Le programme a **bien** terminé ? (code de retour normal)
- Le programme a terminé en moins de 100 millisecondes ?
- L'état du service est celui attendu après cette requête ?
- L'erreur renvoyée est celle attendue ?

Critique des tests

Limites

Testing shows the presence, not the absence of bugs.
(Edsger Dijkstra)

Avantages

- Scénarios d'utilisation = exemples pour utilisateurs.
- Détecter une cassure (*break*) des scénarios testés devient trivial.
- Évaluation des performances du programme.

Classification des tests (1)

De nombreuses classifications...
Voyons les types de test très importants.

Test unitaire

Test d'une (petite) portion d'un logiciel **en isolation**.

Exemple : entrées/sorties/erreurs d'une fonction/module.

Test de (non) régression

Re-exécuter des tests lors d'une modification pour éviter de *break*.

Au cœur des méthodologies *récentes* de suivi de qualité.

Classification des tests (selon l'angle d'approche)

Isolation des composants ?

Test unitaire, d'intégration, *système*...

Connaissance du code testé ?

Tests dits en boîte blanche/grise/noire.

Fonctionnel ou non ?

Non fonctionnel s'intéresse au logiciel dans son environnement.
Exemples : test de charge, de performance, de sécurité...

Comment décider qu'un test passe ou non ?

- Quels résultats accepter quand plusieurs sont équivalents ?
- Faut-il que l'algo produise la solution optimale à un problème ou est-ce qu'une solution de qualité *raisonnable* suffit ?

Que faut-il tester ?

Impossible de répondre de manière générique. Nombreux facteurs.

- Criticité du logiciel.
- Méthodologie de travail utilisée par les développeurs.
- Intuitions et rigueur des développeurs.

Quelques exemples de bonnes pratiques.

- *Bien* découper en composants, et *bien* les tester.
- Tout algo qui semble non trivial.
- Si système à états inévitable, tester les états du système.
- Ne pas oublier l'interface externe du logiciel.
  - API pour le cas d'une bibliothèque ou d'un service.
  - CLI pour un programme.
  - Si utilisateur=humain, l'ergonomie se teste aussi.

Testception

Les tests étant eux-mêmes du code, les tester aurait du sens... Faut-il le faire ? Comment ?

Souvent, on se sert juste de certaines métriques pour vérifier qu'un test utilise bien la bonne sous-partie du logiciel qui nous intéresse.

La plus courante est la couverture (*coverage*), qui permet de savoir quelles instructions/fonctions/branches[2] sont utilisées par les différents tests (et combien de fois elles sont appelées).

Au niveau de tout un projet, le *coverage* permet de voir quelles parties du code sont faiblement testées. Suivre son évolution aide à détecter de nouveaux codes morts ou peu testés.

2. Plutôt que de compter les instructions appelées, on peut les compter au sein de l'arbre syntaxique de la fonction — *e.g.*, savoir si un code qui suit un branchement `if`/`else` n'est appelé qu'après la partie `if` peut montrer un problème (ou non !)

Idées pour maintenir une qualité

Citation

Programs should be written for people to read,
and only incidentally for machines to execute.

(Harold Abselson, Gerald Jay Sussman)

Citation

Any fool can write code that a computer can understand.
Good programmers write code that humans can understand.

(Martin Flower)

Citation

If the computer doesn't run it, it's broken.
If people can't read it, it will be broken. Soon.

(Charlie Martin)

TODO