Qualité logicielle

Adrien FAURE

2023-01-03

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 !)

Human in the loop

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 Fowler)

Citation

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

(Charlie Martin)

Ne pas se répéter

Principe

Éviter toute redondance de code.

Pourquoi ?

Tout code, qu'il comporte actuellement un bug ou non, peut causer des soucis plus tard.

Si un bout de code dupliqué est corrigé, il est probable qu'on oublie d'appliquer le correctif sur ses réplicats.

Keep It Simple, Stupid (KISS)

Principe

Toute complexité **non indispensable** devrait être évitée.

Pourquoi ?

Comprendre (et donc maintenir) un code simple est plus facile.

Conséquences

- Impacte le choix d'algorithmes.
- Impacte le choix de structures de données.
- Impacte le choix de paradigmes et de modularisation du code.
- Limite le *feature creep*.

Separation of concerns (SoC)

Principe

Identifier et séparer l'implémentation des parties d'un programme responsables de tâche(s) spécifique(s).

Pourquoi ?

- Identification de(s) sous-problème(s).
- Permet de créer des briques réutilisables.
- Modulariser participe à garder un code simple (KISS).

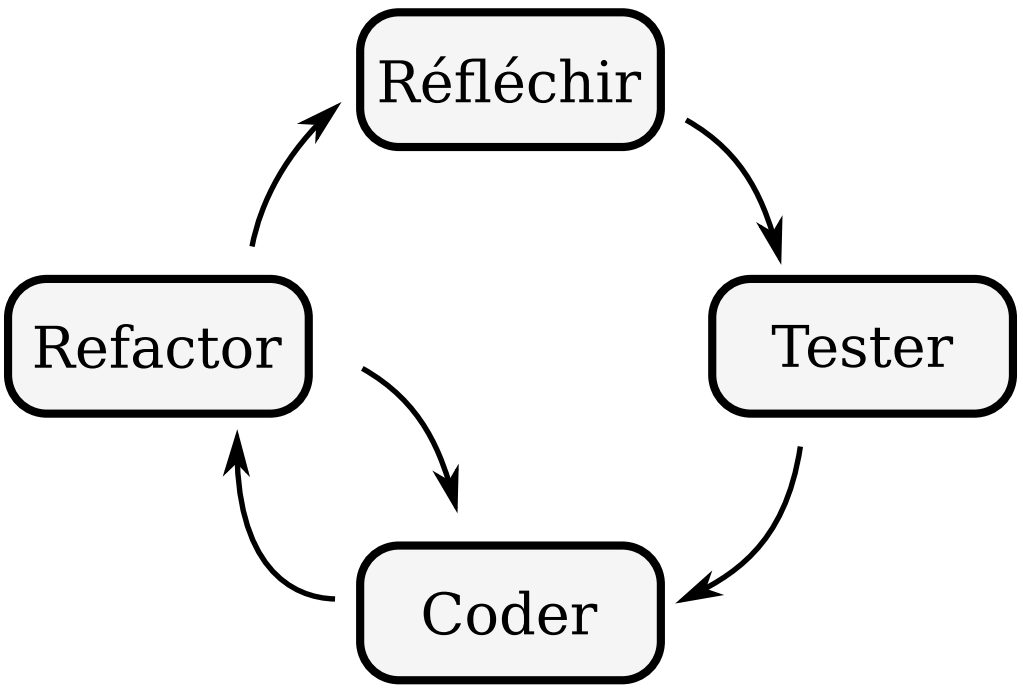
Conséquences

- Impacte la structuration du code.
- Choix du bon niveau de découpage (pas toujours évident): fonction, module, bibliothèque, classe etc.

Test Driven Development (TDD)

Principe

- Concevoir un logiciel pas à pas, par cycles itératifs.
- Écrire un test avant d'implémenter quelque chose.
- Alterner entre écriture des tests et du code en continu.



Test Driven Development (TDD)

Principe

- Concevoir un logiciel pas à pas, par cycles itératifs.
- Écrire un test avant d'implémenter quelque chose.
- Alterner entre écriture des tests et du code en continu.

Intérêts

- Permet de vérifier un besoin avant de coder.
- Permet de vérifier que l'API est satisfaisante avant de coder.
- Documente comment appeler chaque code.
- Fournit des tests de régression pour chaque code.

Pair Programming

Principe

Travailler en binôme sur une seule machine (ou partage d'écran).

- A développe l'application.
- B observe et critique chaque décision, détecte soucis.

Échanger régulièrement les rôles.

Intérêts

- Détection de soucis très efficace.
- Critique instantanée. Réduit cycles pour manque de qualité.
- Force communication/raisonnement/justification des choix.

Contrôle d'environnement logiciel

Principe

Définir et isoler les environnements logiciels utilisés.

- Pour développer l'application.
- Pour tester l'application.
- Pour déployer l'application.

Intérêts

- Coût d'entrée pour un nouveau développeur réduit.
- Documente les dépendances et leurs versions.
- Tester différentes combinaisons devient simple.