Engineer

Développement guidé par les tests

écrire le test avant le code, un petit test à la fois

Ce que c’est

Le développement guidé par les tests est une manière de programmer dans laquelle le test d’un comportement est écrit avant le code qui réalise ce comportement, un petit test à la fois. Kent Beck, qui décrit son propre rôle comme une redécouverte, le qualifie de démarche de programmation. En 2023, constatant que les programmeurs ne s’accordaient pas sur le sens du terme, il a décrit cette démarche dans un court billet intitulé Canon TDD, où il utilise presque partout l’abréviation TDD.

La démarche comporte cinq étapes :

- Dresser la liste des cas que la modification doit traiter.
- Choisir exactement un cas de la liste et en faire un test concret qui s’exécute.
- Faire réussir ce test sans faire échouer les précédents et ajouter à la liste chaque nouveau cas révélé par le travail.
- Si la conception de l’implémentation peut être améliorée, la remanier maintenant ; cette étape est facultative.
- Revenir à l’étape 2 jusqu’à ce que la liste soit vide.

  La liste est une analyse du comportement, pas un document de conception : d’abord le cas de base, puis ses variantes, puis ce que la modification doit préserver du fonctionnement existant. Le test écrit à l’étape 2 est un véritable test automatisé, avec une préparation, un appel et des assertions ; il échoue à sa première exécution, puisque le comportement qu’il décrit reste à écrire. Beck appelle l’étape 3 le passage du rouge au vert, d’un test en échec à un test réussi.

Il est tout aussi utile de savoir ce que cette démarche ne signifie pas. On reproche souvent au TDD d’imposer l’écriture de tous les tests avant la moindre ligne de code, et Beck répond directement : ce n’est pas du TDD, et le critiquer revient à critiquer autre chose. Il est tout aussi souple dans l’autre sens. Une

équipe qui procède autrement avec de bons résultats peut continuer ainsi ; il précise qu'aucune récompense n'est décernée pour avoir suivi exactement ces étapes.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est le développement guidé par les tests ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Une seule équipe transforme une idée en un produit auquel vos clients peuvent se connecter, sans le faire passer entre trois prestataires.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. Application GIS : revenus multipliés par 10 — 10x croissance du chiffre d'affaires portée par le produit cartographique

Application GIS : revenus multipliés par 10 Contactez-nous →

Pourquoi c'est important

Beck précise le but de la démarche. Un programmeur qui modifie le comportement d'un système doit obtenir un système où la modification préserve le comportement existant tout en satisfaisant les nouvelles exigences, où le code est prêt pour la modification suivante et où le programmeur et ses collègues ont confiance dans ces trois points. Puisque chaque test est écrit alors que la modification qu'il vérifie est encore limitée et récente, le filet de sécurité grandit au même rythme que le code, au lieu d'être promis pour plus tard.

La deuxième raison concerne la conception. Beck distingue deux types de décisions : comment appeler un comportement et comment le système le réalise. Écrire le test d'abord force les décisions d'interface, parce que le test doit appeler quelque chose et indiquer ce qui revient. Les décisions d'implémentation attendent l'étape de refactoring, où des tests réussis les protègent. Un test difficile à écrire annonce tôt une interface qui sera difficile à utiliser.

La troisième raison est la double vérification. Une personne calcule la valeur attendue d'un test à partir de l'exigence, tandis que le code y parvient par une autre voie. Lorsqu'elles concordent, deux démarches indépendantes ont atteint la même réponse ; selon Beck, cette double vérification représente une grande partie de la valeur du TDD comme contrôle.

La démarche aide aussi à savoir quand s'arrêter. Beck conseille de continuer à tester et à coder jusqu'à ce que la crainte du comportement du code se transforme en ennui. Cette habitude précède largement le billet. Il situe sa

redécouverte entre 1994, lorsqu'il a écrit SUnit pour un client à l'occasion d'une mission de conseil, et octobre 1995, lorsqu'il a présenté le TDD à Ward Cunningham à la conférence OOPSLA d'Austin. Son article Simple Smalltalk Testing décrit un cadre de test simple pour Smalltalk, et la documentation unittest de Python le cite toujours comme l'article d'origine de Beck sur le modèle que partage unittest.

À quoi cela ressemble en pratique

Prenons un contrôle de contenu qui doit signaler les mots à la mode interdits dans les textes d'un site web. Le travail commence par la liste de tests, sous forme de simples notes :

```
- un mot interdit est signalé
- un mot interdit contenant une majuscule est signalé
```

L'étape 2 transforme exactement un élément en test. Le premier cas vient en premier pour une raison : il fournit au contrôle un mot interdit connu et vérifie que ce mot est signalé. Le test échoue tant que le contrôle ne peut pas le détecter. Il va dans un fichier nommé `test_wording.py`, que pytest découvre parce que son nom correspond à `test_*.py` :

```
from wording import find_banned


def test_reports_a_banned_word():
    assert find_banned("we leverage it", {"leverage"}) == ["leverage"]
```

L'exécution de `pytest -q` échoue immédiatement, car il n'existe aucun `find_banned` à importer. Cet échec est le but : on a vu le test échouer, donc une réussite ultérieure aura un sens. Le code le plus simple qui le fait réussir suffit pour l'instant, dans `wording.py` :

```
def find_banned(text, banned):
    return [word for word in text.split() if word in banned]
```

Le test réussit. L'élément suivant devient le test suivant :

```
def test_reports_a_capitalised_banned_word():
    assert find_banned("Leverage it", {"leverage"}) == ["Leverage"]
```

Il échoue et, comme pytest affiche les valeurs intermédiaires d'une assertion, la sortie montre une liste vide là où `["Leverage"]` était attendu. Comparer `word.lower()` au lieu de `word` le fait réussir. En écrivant cette modification, le programmeur remarque que `text.split()` laisse la ponctuation attachée au mot : « leverage, », avec une virgule, passerait donc inaperçu. Ce cas va sur la liste plutôt que dans le code, puis devient le troisième test :

```
def test_reports_a_banned_word_before_a_comma():
    assert find_banned("Use it, leverage, win", {"leverage"}) == ["leverage"]
```

Il échoue, puis une ligne de plus fait réussir les trois :

```
def find_banned(text, banned):
    words = [word.strip(".,;:!?") for word in text.split()]
    return [word for word in words if word.lower() in banned]
```

L'étape 4 est facultative et il y a peu à améliorer ici : deux lignes, chacune avec une tâche. Chaque élément de la liste a maintenant un test qui réussit ; le travail est donc terminé. Chaque valeur attendue a été écrite à partir de l'élément de la liste, jamais copiée depuis une exécution.

L'ordre de la liste était délibéré, et le propre historique du site en montre la raison. Un projet qui a écrit des tests pour ses propres contrôles, réalisé en août 2026, a constaté que ses contrôles personnalisés, notamment ceux d'orthographe, de voix rédactionnelle et de cohérence des chiffres, avaient réussi pendant des mois. Un contrôle qui n'a vu que des entrées correctes est indiscernable d'un contrôle qui ne lit rien. La correction consistait à introduire le défaut que chaque contrôle doit détecter : le contrôle orthographique reçoit un mot volontairement mal orthographié et le test échoue si l'exécution ne signale rien, le contrôle de voix reçoit un texte à la première personne qu'il doit rejeter et le contrôle des mots à la mode reçoit un mot interdit. Deux contrôles se sont révélés lire un ensemble de fichiers plus restreint que ne l'annonçait leur

documentation. La page ne décrit pas le TDD, mais sa leçon est celle que le TDD intègre dès la première étape : un test n'a de valeur qu'une fois qu'on l'a vu échouer.

Où les problèmes surviennent

Tous les tests sont écrits en premier. Transformer chaque élément de la liste en test avant qu'un seul réussisse est le contresens que Beck désigne. Si le premier test oblige à revoir la démarche, chaque test spéculatif doit être repris ; un programmeur au sixième test sans aucune réussite est gagné, écrit-il, par la déprime ou l'ennui. Écrivez un test, faites-le réussir, puis choisissez le suivant. Si un nouveau test montre que le travail existant ne peut pas traiter un cas, choisissez entre poursuivre et recommencer ; Beck conseille de recommencer en prenant les cas dans un autre ordre.

Le test ne contient aucune assertion. Un test qui appelle le code sans rien vérifier augmente le taux de couverture sans rien prouver. Chaque test demande une préparation, un appel et une assertion ; Beck suggère d'essayer, de temps à autre, de l'écrire à rebours en partant de ses assertions.

L'assertion est supprimée. Quand un test ne réussit pas, supprimer son assertion donne l'illusion d'une réussite. Faites-le réellement réussir, ou remettez le cas sur la liste et revoyez l'ordre.

Les valeurs attendues sont copiées depuis la sortie. Exécuter le code, copier ce qu'il renvoie et le coller dans le test transforme une double vérification en vérification unique : le test se contente désormais de confirmer que le code fait ce qu'il a déjà fait. Calculez la valeur attendue à partir de l'exigence, à la main si nécessaire.

Le refactoring se fait pendant qu'on cherche à faire réussir le test. Modifier structure et comportement dans la même étape, c'est ce que Beck appelle porter deux casquettes. Faites d'abord fonctionner le code, puis améliorez sa structure : obtenez un résultat vert avec le changement le plus simple, puis ne nettoyez le code qu'une fois les tests réussis.

Le rangement va trop loin, ou l'abstraction arrive trop tôt. Ranger donne satisfaction et peut sembler plus sûr que d'affronter le prochain test. Arrêtez-vous lorsque le code est assez bon pour cette séance et laissez deux morceaux

semblables tels quels jusqu'à ce que le motif soit clair. Beck le résume ainsi : « La duplication est un indice, pas un ordre. »

Les tests connaissent l'implémentation. Les tests qui appellent des méthodes privées ou vérifient les appels effectués par le code, plutôt que ce qu'il a produit, cassent dès que les mécanismes internes changent. Dans son chapitre sur les tests unitaires de *Software Engineering at Google*, Erik Kuefler les qualifie de tests fragiles : ils échouent sur une modification sans rapport, qui n'introduit aucun véritable bogue. En TDD, c'est doublement coûteux, car le refactoring dépend de tests qui restent verts pendant les changements internes. Testez par l'interface publique, vérifiez l'état obtenu plutôt que les interactions et nommez chaque test d'après le comportement qu'il vérifie.

Pour aller plus loin

- Canon TDD — la propre présentation de Kent Beck en 2023 : les cinq étapes, la séparation entre conception de l'interface et de l'implémentation et les erreurs de chaque étape.
- Simple Smalltalk Testing: With Patterns — l'article de Beck sur un cadre de test simple pour Smalltalk, avec ses fixtures, cas et suites de tests, dans une copie archivée.
- Tests unitaires — le chapitre d'Erik Kuefler dans *Software Engineering at Google* sur les tests qui survivent au refactoring, passent par les API publiques et vérifient les comportements.
- Premiers pas — l'introduction de pytest : installation, premier test en échec et son compte rendu, et découverte des fichiers de test.
- unittest: cadre de tests unitaires — le cadre de la bibliothèque standard de Python, qui cite l'article de Beck comme la description d'origine du modèle qu'il partage.

Glossaire

Choisissez selon vos contraintes réelles

Présentez le processus de travail, les interfaces existantes et les utilisateurs qui ont besoin du changement. Nous pouvons définir une livraison testable et une transmission permettant à votre équipe d'assurer la maintenance des éléments livrés.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est le développement guidé par les tests ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/test-driven-development/