Engineer

Refactorisation

🛠 Modifier la structure du code sans changer son comportement

Ce que c’est

La refactorisation modifie la structure interne du code sans changer ce que ce code fait pour ses utilisateurs. Les noms, les limites des fonctions et l’organisation des modules changent ; les entrées, les sorties et les effets de bord restent exactement les mêmes. Le verbe est refactoriser, et une telle modification s’appelle aussi une refactorisation.

Martin Fowler définit le terme dans son livre Refactoring et a repris cette définition sur son site en 2004. Comme nom, une refactorisation est une modification de la structure interne d’un logiciel qui rend le code plus facile à comprendre et moins coûteux à modifier, sans changer son comportement vu de l’extérieur. Comme verbe, refactoriser consiste à effectuer une série de telles modifications en conservant ce comportement extérieur tout au long du processus. Les deux aspects comptent. L’objectif est de rendre la prochaine modification moins coûteuse, avec pour contrainte que personne ne puisse constater de l’extérieur qu’il s’est passé quelque chose.

Fowler trace une limite nette : « La refactorisation n’est pas un autre mot pour nettoyer le code », écrit-il sur refactoring.com. Elle désigne une technique précise. Il emploie restructuration comme terme plus large pour la réorganisation du code par tout moyen, y compris ceux qui ne préservent pas le comportement à chaque étape. La taille des étapes fait de la refactorisation une technique : chacune est une petite transformation qui fait peu à elle seule et risque peu de mal tourner, et l’ensemble du système fonctionne encore après chacune. Une longue série peut néanmoins produire un changement structurel important.

Les étapes ont des noms. Le catalogue de refactoring.com, qui accompagne la seconde édition du livre, les répertorie avec les anciens noms qu'elles remplacent : Extract Function (« extraire une fonction »), aussi appelé Extract Method (« extraire une méthode »), et son inverse Inline Function (« remplacer un appel de fonction par son corps ») ; Inline Variable (« remplacer une variable par son expression ») ; Rename Variable (« renommer une variable ») ; Move Function (« déplacer une fonction ») ; Split Phase (« séparer les phases »), et des dizaines d'autres. Un nom partagé permet au développeur de dire en deux mots ce qu'a été la modification et à la personne chargée de la revue de la vérifier selon une recette connue.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce que la refactorisation ? » 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

Un logiciel qui réussit continue à évoluer. Les fonctionnalités et les corrections s'empilent, chacune tendant à rendre la suivante un peu plus difficile, jusqu'à ce que, comme le décrit Fowler, tout ajout prenne un temps péniblement long. La refactorisation fait contrepoids : elle remodèle le code pour que chaque ajout n'apporte pas de complexité inutile. Son rendement se mesure au coût de la prochaine modification, pas à l'apparence du code.

C'est pourquoi Fowler la considère comme une partie de la programmation ordinaire plutôt que comme une tâche inscrite au planning. Il cite trois moments. Avant d'ajouter une fonctionnalité, il vérifie si la forme du code la rend facile à ajouter et refactorise d'abord si ce n'est pas le cas, ce qu'il trouve généralement plus rapide que de forcer l'ajout. Une fois la fonctionnalité opérationnelle, il refactorise le résultat pour que quelqu'un qui revient quelques semaines plus tard puisse le comprendre sans difficulté. Et lorsqu'il doit lire du code inconnu pour le comprendre, il le refactorise afin que la prochaine personne ne bute pas au même endroit. Le premier cas a son propre nom, la refactorisation

préparatoire. Fowler attribue à Kent Beck la règle qui la sous-tend : pour toute modification, remodeler d'abord le code jusqu'à rendre la modification facile, ce qui peut être la partie difficile, puis seulement la réaliser.

La refactorisation limite aussi les risques d'une manière que le développement de fonctionnalités ne permet pas. Fowler reprend une autre métaphore de Beck, celle des deux casquettes. Avec la casquette de refactorisation, chaque modification préserve le comportement et les tests restent verts du début à la fin. En ajoutant une fonctionnalité, on ajoute des tests et certains tests existants échouent, ce qui est plus stressant et plus risqué. Un développeur qui sait quelle casquette il porte sait ce que signifie un test en échec : pendant une refactorisation, la dernière étape était incorrecte.

La même idée permet de rembourser l'essentiel de la dette technique. L'article de Google sur la dette cachée dans les systèmes d'apprentissage automatique cite d'abord la refactorisation du code parmi les moyens de réduire cette dette. Les outils aident : de nombreux langages disposent désormais d'éditeurs automatisant les refactorisations courantes. Fowler les juge utiles mais non indispensables ; lorsqu'ils manquent, il s'appuie sur de petites étapes et des tests fréquents.

À quoi cela ressemble en pratique

Prenons une fonction écrite pour cet article, qui calcule le total d'une commande et imprime un reçu au même endroit :

```
function printReceipt(order) {
  let total = 0;
  for (const line of order.lines) {
    total += line.price * line.quantity;
  }

  // print the receipt lines
  console.log(`Order ${order.id}`);
  console.log(`Total: ${total}`);
}
```

Une nouvelle exigence arrive : le même total doit apparaître sur une facture. Plutôt que de copier la boucle, le développeur commence par refactoriser. Première étape : extraire une fonction, Extract Function. La boucle passe dans sa propre fonction, nommée d'après son calcul, puis les tests sont exécutés. La

deuxième étape extrait de la même manière l'impression : un nom de fonction remplace le commentaire, puis les tests sont relancés. Troisième étape : remplacer une variable par son expression, Inline Variable. La variable `total` n'est désormais utilisée qu'une fois, l'appel prend donc sa place. Après chaque étape, le programme fait exactement ce qu'il faisait auparavant et la modification pourrait être commitée séparément :

```
function printReceipt(order) {
  printLines(order.id, orderTotal(order));
}

function orderTotal(order) {
  let total = 0;
  for (const line of order.lines) {
    total += line.price * line.quantity;
  }
  return total;
}

function printLines(id, total) {
  console.log(`Order ${id}`);
  console.log(`Total: ${total}`);
}
```

La fonctionnalité n'est ajoutée que maintenant, par une ligne du code de facturation appelant `orderTotal`. Aucune des trois étapes de refactorisation n'a nécessité de nouveau test, puisque rien d'observable n'a changé : les tests qui couvraient déjà le reçu le prouvent.

Le code d'infrastructure se refactorise aussi, avec des enjeux plus élevés, car les noms renvoient à des ressources en cours d'exécution. La documentation Terraform avertit que, sans indication contraire, donner une nouvelle adresse à une ressource dans la configuration lui demande de détruire l'existante et d'en construire une nouvelle. Renommer `aws_instance.a` en `aws_instance.web` pour améliorer la lisibilité remplacerait donc un serveur. Depuis Terraform v1.1, un bloc `moved` enregistre au contraire l'ancienne adresse et le plan suivant met l'adresse à jour sans rien détruire :

```
resource "aws_instance" "web" {
  # (configuration unchanged)
}

moved {
```

```
  from = aws_instance.a
  to   = aws_instance.web
}
```

Pour les versions antérieures, la documentation renvoie à la commande `terraform state mv` pour le même déplacement. Dans les deux cas, la refactorisation n'est sûre qu'une fois le plan lu et le déplacement confirmé, sans destruction suivie d'une création.

Séparer ces modifications des fonctionnalités est plus facile lorsque l'historique indique leur nature. La convention de commits décrite dans l'étude sur une même norme d'historique imposée à quatre dépôts commence chaque message par un préfixe conventionnel désignant le type de modification, imposé par un hook de message de commit. La page, qui porte sur un travail effectué en juillet 2026, compte 26 refactorisations dans l'historique du plus grand dépôt, à côté de 156 commits de fonctionnalités et de 133 corrections. Elle précise aussi honnêtement que le hook vérifie le format et non le contenu. Les tests et la revue doivent encore montrer qu'un commit étiqueté comme refactorisation a laissé le comportement inchangé.

Où les problèmes surviennent

Des changements de comportement se glissent dans la refactorisation. Un développeur remet de l'ordre dans une fonction et y corrige un bug dans le même commit. Lorsqu'un test échoue, personne ne peut déterminer lequel des deux changements l'a causé et la personne chargée de la revue doit lire chaque ligne comme un changement de comportement possible. Portez une seule casquette à la fois : refactorisez dans un commit, changez le comportement dans le suivant et, si un test échoue pendant la refactorisation, annulez la dernière étape au lieu de poursuivre en corrigeant.

Une grande restructuration est appelée refactorisation. Une branche passe des semaines avec un build en échec pendant que le code est « en cours de refactorisation ». C'est une restructuration au sens de Fowler, avec précisément le risque que la refactorisation vise à éviter. Découpez le travail en étapes qui laissent chacune le système opérationnel et peuvent être fusionnées séparément. Si ce n'est pas possible, appelez cela une réécriture et planifiez-la comme telle.

Aucun test ne montre que rien n'a changé. La préservation du comportement est une affirmation ; sans tests, c'est une supposition. Avant de toucher à du code non testé, écrivez des tests qui figent ce qu'il fait aujourd'hui, même si cela semble incorrect, puis corrigez les bizarreries dans des modifications distinctes. La règle de Fowler dans son exemple de refactorisation préparatoire s'applique ici : chaque fois que vous avez envie d'exécuter le code et d'en examiner la sortie à l'œil, écrivez plutôt un test.

Un renommage dans le code d'infrastructure remplace la ressource. Un nouveau nom bien choisi dans Terraform ressemble à une refactorisation lors de la revue, tandis que le plan propose discrètement de détruire une base de données et d'en créer une vide. Enregistrez chaque renommage et chaque déplacement vers un module ou hors de celui-ci dans un bloc `moved` . Considérez toute destruction dans le plan d'une modification censée être une refactorisation comme une étape en échec.

Un ancien bloc `moved` est supprimé. Ces blocs s'accumulent et les retirer ressemble à du rangement. La documentation Terraform parle de rupture de compatibilité : toute configuration utilisant encore l'ancienne adresse recevrait alors un plan détruisant l'objet au lieu de le déplacer. Gardez les blocs historiques, chaînez-les lorsqu'un objet est renommé deux fois et ne les retirez d'un module privé qu'après que tous ses utilisateurs ont appliqué la version qui les contient.

La refactorisation attend une phase dédiée. Une équipe qui programme un « sprint de refactorisation » pour plus tard continue à ajouter des fonctionnalités à un code qui leur résiste. Un sprint prévu pour plus tard est facile à supprimer lorsqu'une échéance change. Fowler conseille l'inverse : refactoriser dans chaque modification, d'abord pour la rendre facile, puis pour laisser un résultat clair, afin que ce travail n'ait jamais besoin de sa propre case dans le planning.

Pour aller plus loin

- Définition de la refactorisation — définitions nominale et verbale de Martin Fowler, telles qu'elles figurent dans son livre.
- Refactoring — le site du livre de Fowler, avec les raisons d'intégrer la refactorisation à la programmation quotidienne et le catalogue complet.
- Extraire une fonction — l'entrée du catalogue consacrée à l'une des refactorisations les plus courantes, avec son inverse et son nom dans la première édition.

- Un exemple de refactorisation préparatoire — Fowler remodèle son propre code à petits pas avant d'ajouter une fonctionnalité et explique les deux casquettes.
- Refactoriser des modules — documentation Terraform sur les blocs `moved`, les renommages, la séparation des modules et la raison de conserver les anciens blocs.

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-ce que la refactorisation ? » 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/refactoring/