**Engineer**

# Dette technique

le surcoût futur d'un raccourci choisi aujourd'hui

## Ce que c’est

La dette technique est le coût futur d’un raccourci dans la conception ou le code d’un système : un choix qui fait gagner du temps aujourd’hui et rend les changements ultérieurs plus lents, plus risqués ou impossibles tant que le raccourci n’est pas supprimé. En anglais, les équipes abrègent souvent technical debt en tech debt. Le mot est une métaphore, choisie parce que le raccourci se comporte comme un emprunt. Il permet d’obtenir quelque chose maintenant et le fait payer ensuite, aussi longtemps qu’il reste en place.

L’idée vient du retour d’expérience de Ward Cunningham à OOPSLA en 1992, sur WyCash, un système de gestion de portefeuille écrit en Smalltalk. Une petite équipe conservait une connaissance générale d’environ quatre mégaoctets de sources, et les parties du programme arrivées à maturité avaient été révisées ou réécrites de nombreuses fois. Cette consolidation, affirme le rapport, rendait possible une croissance progressive régulière ; son absence constitue un piège grave : du code qui fonctionne et satisfait le client, mais n’a jamais été remis en ordre. Cunningham le résume en une phrase, traduite ici : « Livrer une première version du code, c’est comme s’endetter. » Le rapport lui-même ne parle que de dette et de son analogie avec la dette. Les travaux ultérieurs utilisent l’expression en deux mots, et un article de Google de 2015 attribue la métaphore à Ward Cunningham et la date de 1992.

Des chercheurs réunis à Dagstuhl en avril 2016 pour s’accorder sur ce que couvre le terme ont écrit une définition de travail. La dette y est un ensemble d’éléments de conception ou d’implémentation qui économisent des efforts à court terme, mais laissent le système dans un état qui renchérit les changements ultérieurs ou en rend certains inaccessibles. Ils limitent son impact aux qualités internes du système, surtout sa facilité de maintenance et d’évolution. Cela distingue la dette d’une fonctionnalité jamais construite : le coût de la dette retombe sur les personnes qui modifient le système, et n’atteint les utilisateurs

que par leur intermédiaire. Le même rapport conserve le vocabulaire financier. Le principal est ce que le raccourci a économisé, ou ce qu'une meilleure solution coûterait aujourd'hui. Les intérêts sont de deux sortes : les intérêts récurrents, payés sous forme de vélocité réduite, de défauts induits et de perte de qualité ; les intérêts qui s'accumulent, c'est-à-dire le coût supplémentaire de chaque nouvel élément logiciel construit sur le code imparfait. La dette se trouve surtout dans le code, mais aussi dans la documentation, les tests et les défauts connus.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce que la dette technique ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Définir d'abord le système sur le papier pour que ses éléments s'articulent et que son coût soit connu avant d'écrire du code.

**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. Un plan d'infrastructure pour 14 départements — 14 départements de recherche sur un même plan d'infrastructure

**Un plan d'infrastructure pour 14 départements** Contactez-nous →

## Pourquoi c'est important

Contracter une dette délibérément peut être une bonne décision. Le rapport de Cunningham dit qu'une petite quantité permet à une équipe d'avancer plus vite, à condition de la solder rapidement en réécrivant le code ; l'article de Google convient qu'emprunter est souvent un bon choix stratégique. Tous deux mettent en garde contre la dette laissée en place. Dans le rapport, les intérêts se paient en temps, à chaque minute perdue sur du code imparfait, et une implémentation que personne ne consolide peut s'alourdir au point de bloquer des organisations d'ingénierie entières. Avant d'en arriver là, le programme devient quelque chose qu'aucune personne ne peut maîtriser seule, puis chaque programmeur ne connaît qu'une partie, jusqu'à aboutir à un produit incapable de changer de direction.

Le problème est que la dette n'est souvent pas choisie. Le rapport de Dagstuhl observe qu'elle apparaît souvent sans intention et surprend les développeurs : elle est donc surtout découverte pendant la maintenance, bien après la décision qui l'a créée. Ses causes sont ordinaires : pression du calendrier, absence d'une personne clé ou manque d'informations sur une caractéristique technique au moment du choix. L'article de Google ajoute que la dette cachée est dangereuse, parce qu'elle produit des intérêts composés en silence.

La métaphore donne aussi aux développeurs et aux responsables un langage partagé. Le rapport de Dagstuhl la qualifie de mécanisme de communication puissant, et la vision de ses participants comprend une gestion de la dette aussi attentive que celle des défauts et des nouvelles fonctionnalités, avec un moyen de faire le lien entre les préoccupations des développeurs et les décisions des responsables. Une étude présentée au séminaire montre ce que cela peut signifier dans une équipe. Graziela Tonin, Alfredo Goldman et Carolyn Seaman ont demandé à des équipes étudiantes comptant au total 60 personnes d'identifier et de suivre explicitement leur dette. Une liste visible leur a permis de négocier avec leur client du temps pour le refactoring, de garder une trace des parties du projet encore immatures et de voir si la qualité du code s'améliorait. La plupart des membres ont déclaré communiquer davantage, examiner plus attentivement si une dette était vraiment nécessaire et refactorer plus souvent. Les décisions d'endettement ayant été discutées, ajoutent les auteurs, certaines équipes ont fini par compter moins d'experts que personne d'autre ne pouvait remplacer.

## À quoi cela ressemble en pratique

La dette devient gérable lorsqu'elle est écrite à l'endroit où l'équipe planifie son travail. Une entrée de registre, rédigée pour cet article, peut être aussi courte que celle-ci :

```
TD-14       Totaux des factures calculés à trois endroits
Artefacts:  billing/web, billing/export, reports/monthly
Cause:      date de lancement fixe ; copier la règle prenait moins de temps
            que la déplacer dans un module partagé
Principal:  env. 2 jours pour déplacer la règle dans un seul module, avec tests
Intérêts:   tout changement de règle fiscale est appliqué et testé trois fois ;
            l'une des cinq dernières modifications a omis une copie
Suite:      régler au prochain changement de règle fiscale ; l'export e-facture
            prévu ce trimestre ne doit pas devenir une quatrième copie
Responsable: équipe de facturation
```

Les champs suivent le modèle du rapport de Dagstuhl : un élément lié à des artefacts concrets, avec une cause, un principal et des intérêts. Écrire les intérêts rend l'entrée utile, parce que cela transforme le sentiment que le code de facturation est désordonné en un coût que quelqu'un peut évaluer. Si les règles fiscales changent quatre fois par an et que chaque changement coûte une journée supplémentaire de travail et de tests, les deux jours de principal sont remboursés en six mois au plus. L'export prévu constitue des intérêts qui s'accumulent au sens de Dagstuhl : chaque nouvel élément construit sur la règle

dupliquée augmente le prix d'une correction ultérieure. Si, au contraire, le code était figé et que personne ne prévoyait d'y toucher de nouveau, la même entrée justifierait de laisser la dette en place.

La dette ne concerne pas seulement le code. L'étude de cas du site sur la refonte des processus internes, avec 8 000 heures économisées consigne le travail antérieur de la fondatrice chez Mappitall. Elle demande si la dette technique ralentit le lecteur et décrit celle qui s'accumule autour de l'organisation du travail : une architecture logicielle développée par accumulation plutôt que par conception, des systèmes fonctionnels mais inefficaces, et une planification qui laissait les personnes attendre ou les surchargeait. Il n'y avait pas le feu, raison pour laquelle on n'y avait pas touché. Revoir l'architecture, les systèmes et la planification a économisé environ 8 000 heures sur le travail mené de 2015 à 2022. La page attribue une partie de ce résultat à l'ancrage des changements dans le fonctionnement de l'équipe, puisque, si les améliorations ne deviennent pas permanentes, on finit par revenir aux anciennes pratiques.

## Où les problèmes surviennent

**Tout est appelé dette technique.** Une fois l'expression populaire, chaque désagrément reçoit cette étiquette, d'une vieille bibliothèque à une fonctionnalité manquante ou au style d'un collègue. Le rapport de Dagstuhl nomme directement ce risque et demande des définitions précises pour y répondre. Réserver le terme aux raccourcis dont le coût retombe sur les changements futurs, consigner les défauts comme des défauts et les souhaits comme des fonctionnalités, et refuser une entrée de registre qui ne peut nommer sa cause et ses intérêts.

**La dette n'existe que dans les têtes.** Une dette que personne n'a écrite est payée par la prochaine personne qui touche au code, généralement sans savoir pourquoi le travail prend si longtemps. Tenir un registre à côté du backlog et le consulter avant de modifier une partie du système, comme les équipes étudiantes de l'étude de Tonin utilisaient leur liste. Un élément ne devient visible pour ceux qui décident du temps à y consacrer que lorsqu'il est écrit là où ils regardent.

**Le principal est remboursé là où aucun intérêt n'est dû.** Nettoyer un module que personne ne modifie donne le sentiment d'être productif sans rien rapporter, tandis que le code modifié chaque semaine continue de coûter. Les

intérêts ne s'accumulent que là où les personnes travaillent : classer les entrées selon la fréquence de modification de chaque partie et selon ce qui va être construit dessus, et rembourser d'abord les éléments coûteux.

**Le remboursement attend une grande réécriture.** Une dette laissée pendant des années suscite souvent un projet de remplacement du système entier, qui contracte sa propre dette tandis que l'ancien système continue de coûter. Cunningham conseillait un remboursement rapide, et l'article de Google énumère les moyens habituels de réduire la dette : refactoring, tests unitaires renforcés, suppression de code mort, réduction des dépendances, API au périmètre resserré et meilleure documentation. Aucun n'ajoute de fonctionnalité ; le but est de rendre le prochain changement moins coûteux. Les planifier dans le travail ordinaire, près du changement qui en a besoin.

**Des expérimentations restent en production.** Essayer une variante sous forme de branche conditionnelle dans le code de production coûte peu pour chaque expérimentation. L'article de Google décrit comment ces branches s'accumulent jusqu'à rendre le test de toutes les combinaisons difficile ou impossible. Il cite Knight Capital, dont le système a, selon l'article, perdu 465 millions de dollars en 45 minutes, apparemment parce que des chemins de code expérimentaux obsolètes se sont comportés de manière inattendue. Réexaminer chaque branche expérimentale et chaque indicateur de fonctionnalité à intervalles prévus, supprimer ce qui n'est plus utilisé et valoriser autant la suppression de code que son ajout.

**Le score d'un outil est pris pour la dette.** L'analyse statique peut compter les lignes dupliquées et les longues fonctions, et un tableau de bord donne à ce compte l'allure d'un solde. L'article de Google dit clairement que la métaphore ne fournit aucun indicateur strict à suivre dans le temps, et qu'une équipe avançant encore vite ne prouve pas une faible dette. À Dagstuhl, Forrest Shull a rapporté qu'aucun ensemble de règles de détection de dette n'était systématiquement corrélé à ses symptômes, comme la hausse du coût des changements et la baisse de la vélocité, d'un projet à l'autre. Confronter les résultats d'un outil à votre propre historique de modifications et de défauts, conserver les règles qui prédisent le coût et abandonner les autres. Les questions de l'article lui-même offrent un meilleur test, notamment la rapidité avec laquelle un nouveau membre de l'équipe peut devenir opérationnel.

# Pour aller plus loin

- The WyCash Portfolio Management System — le retour d'expérience de Ward Cunningham à OOPSLA '92, où apparaît pour la première fois l'analogie avec la dette.
- Managing Technical Debt in Software Engineering — le rapport du séminaire de Dagstuhl 16162, avec la définition de travail, le principal et les intérêts, ainsi que l'étude des listes explicites de dette.
- Hidden Technical Debt in Machine Learning Systems — l'article de 2015 de Sculley et de ses collègues de Google sur l'accumulation de la dette, les moyens de la réduire et sa résistance à la mesure.
- Refactoring — le site de Martin Fowler sur les petits changements qui préservent le comportement, cités en premier par l'article de Google parmi les moyens de réduire la dette.



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 dette technique ? » 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/technical-debt/