
BASTIEN
JAILLOT
LA
DETTE
TECHNIQUE
MÉTIER
PROJETS
MÉTHODOLOGIE

Publié par Les Contrôleurs du train de 13h37
http://letrainde13h37.fr

ISBN : 979-10-91997-05-8

Wagon 42 SAS
La Gare 42380 Périgneux France
http://wagon42.fr

Première édition : le 03 février 2015

**Si vous trouvez une erreur qui nous aurai échappée, n'hésitez pas à nous la reporter sur https://github.com/t13h37/feedback-jaillot-dettetech ou par mail à support@letrainde13h37.fr.**

# À propos du train de 13h37

*Le train de 13h37* est un éditeur indépendant qui publie des articles et des livres autour de la conception web.
En tant que travailleurs du web nous-mêmes, notre volonté à travers *le train de 13h37* est de participer à l'amélioration de la vie de nos pairs en leur proposant des ressources pratiques et utiles.

## Déjà parus

- *Design d'icônes : le manuel* (http://t13h37.fr/livre-icones), par Sébastien Desbenoit ;
- *Petit Précis de Créativité* (http://t13h37.fr/livre-creativite), par Virginie Caplet.

# Remerciements de l'auteur

Merci Loïc, Marie et Corinne de votre compréhension. Plein de #sharethelove.

Merci à mon épouse Claire qui m'a supporté pendant tout le temps de rédaction de ce livre… et pour son affection pendant toutes les phases difficiles d'apprentissage de la dette technique dans la pratique.

Merci aux copains de l'aventure JoliCode avec qui je partage tant de projets.

Merci à mes clients et à tous mes anciens collaborateurs qui m'ont fourni tellement d'anecdotes que j'en arrive à écrire ce livre.
J'espère avoir réussi à vous aider dans vos mésaventures.

Merci à mes relecteurs et autres passionnés du sujet, Élise Remazeilles, Simon Perdrisat et Jacques Bodin-Hullin.

# Pourquoi ce livre ?

Un surnom m'a un jour été donné : « Pompier du code ». J'aime cette notion. Un pompier c'est 50 % d'entraînement, 40 % de prévention et 10 % d'opération de secours. Ça correspond bien à ce livre, qui a pour but de sensibiliser à la dette technique et de donner des pistes pour la prévenir. Mon postulat est qu'une équipe sensibilisée et bien entraînée a bien moins de chances d'être un jour vraiment paralysée par la dette technique. Le dernier chapitre (qui est donc aussi le plus court) traite des solutions possibles pour la résorber lorsqu'il est déjà trop tard.

Ce livre est donc un recueil de conseils et recettes issus de nombreuses expériences, dans de multiples contextes.

Il s'adresse aux acteurs du Web qui participent au quotidien à des projets exposés à de nombreux aléas, sur des périmètres très mouvants et dans un contexte où les technologies et les usages avancent très vite.

Il s'adresse également aux décideurs et autres acteurs de la vie d'un projet qui souhaitent comprendre les enjeux et contraintes techniques de leurs interlocuteurs afin de fluidifier les échanges et améliorer le résultat final.

# Glossaire

## Agilité

L'agilité est la capacité d'une organisation à créer de la valeur et à ravir son client, tout en favorisant et en s'adaptant -à temps- aux changements de son environnement
— Jean Claude Grosjean, « Au fait c'est quoi l'agilité ? Qu'est ce qu'une organisation agile ? »

## Méthode agile

Une méthode agile est une approche itérative et incrémentale, qui est menée dans un esprit collaboratif avec juste ce qu'il faut de formalisme. Elle génère un produit de haute qualité tout en prenant en compte l'évolution des besoins des clients.
— Veronique Messager Rota, « Gestion de projet : Vers les méthodes agiles »

## Effet papillon

"L'effet papillon", en théorie du chaos, est une métaphore du phénomène fondamental de sensibilité aux conditions initiales. Elle est parfois exprimée à l'aide d'une question : « Un simple battement d'ailes d'un papillon peut-il déclencher une tornade à l'autre bout du monde ? »

## Post morterm

Un *post morterm* est l'action qui consiste, après un échec, à expliquer les causes et les impacts de cet échec et à énoncer les mesures mises en place pour que cet échec n'arrive plus.

## Projet pivot

> Dans le processus d'apprentissage par itérations, une startup peut découvrir par des retours terrain avec de vrais clients que son produit n'est pas adapté, qu'il ne répond pas à un besoin. Toutefois, pendant ce processus d'apprentissage, il se peut que la startup ait identifié un autre besoin (souvent connexe au premier produit). Lorsque la startup change de produit pour répondre à ce nouveau besoin identifié, on dit qu'elle a effectué un "pivot".
> — Wikipedia, article « Lean Startup »

## Revue de code

La revue de code est une pratique (absolument indispensable) qui consiste pour les développeurs à faire relire et valider par un tiers le code source écrit. Elle a été popularisée par les méthodes agiles comme l'*Extreme Programming*.

Les gains sont multiples, comme l'augmentation de la qualité et de la sécurité du code et une connaissance mieux partagée.

Sa pratique est très courante dans les projets *open source* et rendu plus accessible *via* les *pull request* des plateformes GitHub et Bitbucket.

Des conférences sur le sujet :

- « 100 % de revue de code » par Agnès Haasser à Paris Web 2014 ;
- « Revoir la revue de code » par Noëlie Andrieu à Sud Web 2014.

## Profiler / tracer

> En informatique, le profilage de code (ou "*code profiling*" en anglais) consiste à analyser l'exécution d'un logiciel afin de connaître son comportement à l'exécution.
> — Wikipedia, article « Profilage de code »

## Mesurer / métrique

Une métrique logicielle est une compilation de mesures issues des propriétés techniques ou fonctionnelles d'un logiciel.
Appliquée à la production logicielle, une métrique est un indicateur d'avancement ou de qualité des développements logiciels.
— Wikipedia, article « Métrique (logiciel) »

## Mêlée quotidienne / *standup meeting*

La mêlée quotidienne (*Daily Scrum*) est une réunion de planification « juste à temps » et permet aux développeurs de faire un point de coordination sur les tâches en cours et sur les difficultés rencontrées. Cette réunion dure 15 minutes au maximum. Le *Scrum Master* s'assure que la réunion ait lieu à heure fixe, mais sa présence est facultative. Le propriétaire du produit n'est pas présent.
À tour de rôle, chaque membre aborde 3 sujets :

- ce qu'il a réalisé la veille,
- ce qu'il compte réaliser aujourd'hui pour atteindre l'objet du *sprint*,
- les obstacles qui empêchent l'équipe d'atteindre le but du *sprint*.

— Wikipedia, article « Scrum (méthode) / Mêlée quotidienne »

## Réunion de lancement du projet / *kick-off meeting*

Le démarrage officiel du projet, sous forme d'une réunion où l'ensemble des participants au projet (le comité projet) sont conviés. La réunion est orchestré par le chef de projet (ou directeur) et se déroule généralement en trois phases (les présentations (société, personnes et projet), le déroulement du projet et le planning).
— Gwénaël Bonhommeau, « Les jalons d'un projet informatique »

## Utilisateur

La personne qui va utiliser le projet. Les enjeux varient du tout au tout entre le cas où l'utilisateur est obligé d'utiliser le projet et celui où le projet doit le conquérir.

## Porteur de projet

La personne qui commandite le projet.

## Client

La personne qui paye pour la réalisation du projet.

# Identifier

L'objectif de ce chapitre est de présenter ce qu'est la dette technique et en quoi elle est inévitable.

Je vous présenterai donc son histoire, ses différentes formes, et enfin comment appréhender ses impacts.

## Définition

> Le développement logiciel c'est […] combien de couches vous pouvez empiler avant qu'elles ne s'effondrent sous leur propre poids […] Il s'agit de gérer cette augmentation de la complexité.
> — Steve Jobs en 1997

*TL;DR : Tout choix a des conséquences.*

En construisant un projet Web, de nombreux choix sont effectués, et ceux-ci, combinés à leurs implémentations, ont un impact sur le cycle de vie de votre projet.

Cela s'appelle la **dette technique** : l'accumulation des risques pris lors des différentes phases techniques tout au long de la vie d'un projet.

Bien souvent, elle prend la forme d'une combinaison de fonctionnalités qui n'auraient jamais dû voir le jour (et qui sont la cause de tous les problèmes réels) avec la sédimentation naturelle du code, inéluctable (et qui peut seulement être atténuée).

Elle est inévitable. Appréhendez-la de manière à effectuer ces choix en connaissance de cause et avec honnêteté intellectuelle.

## Historique

Le terme de dette technique – selon Wikipédia – provient initialement de la logique d'intérêts que l'on retrouve dans le calcul d'une dette dite financière. Il s'agit donc de son application dans la vie d'un projet de développement logiciel.

Une dette (financière ou autre) est avant tout un investissement : un emprunt est contracté auprès d'une entité, et permet la réalisation de choses impossibles sans cet apport.
En contrepartie, le montant à rembourser est supérieur à celui qui a été emprunté, car s'y ajoutent les intérêts.
Si elle n'est pas remboursée rapidement, son coût augmente jusqu'au point où il est plus possible de la rembourser intégralement et où tout nouvel emprunt ne sert qu'à rembourser les intérêts précédents, créant par là-même de nouveaux intérêts et obligations.

Ainsi, dans l'élaboration d'un projet, des choix techniques ayant des impacts sur l'avenir du projet sont effectués. D'un côté, ils permettent au projet de débuter puis d'avancer ; d'un autre, ils grèvent les possibilités d'évolution du projet car l'accumulation de décisions et changements — qui forment ce que nous appellerons *l'historique projet* — augmente les risques d'impacts futurs, jusqu'au moment où il devient impossible de construire quoi que ce soit de solide. Les *bugs* et les effets de bord deviennent alors la normalité, empêchant toute évolution du projet.

Dans cette situation, le remboursement des intérêts consiste à mélanger des phases de réflexion et de correction sur l'architecture technique, afin de faire le point entre l'adéquation du socle technique et l'avenir fonctionnel envisagé. Il peut alors devenir pertinent de revenir sur des choix précédents et de prendre le temps de bien préparer le socle technique avant de réattaquer une nouvelle phase de conception.

La prise de conscience de la dette technique revient donc à peser l'impact des choix effectués : évaluer le coût de résolution et le comparer au retour sur investissement.
Tout l'enjeu est donc de réduire ce coût de résolution pour qu'il reste inférieur à ce que la dette créée rapporte **vraiment**.

## Identifier la dette technique

Force est de constater que la technicité des projets liés au Web va en grandissant de manière exponentielle, avec des problématiques qui se complexifient ou se croisent : référencement, accessibilité, réseaux sociaux, performance, multimédias et multi-écrans, API nécessaires pour des applications mobiles, diversité des plateformes mobiles, besoins d'interactivité,

*cloud*, élasticité, décentralisation / centralisation / *analytics*, etc.

Ces contraintes font que notre secteur d'activité est particulièrement propice à l'expansion de la dette technique dans les projets.

Or, la différence de culture (langage, besoins, compréhension des retombées) entre les concepteurs des projets et ceux qui les réalisent entraîne, selon le rôle occupé dans un projet, des perceptions différentes pouvant varier du tout au tout.

Pour schématiser :

- le client peut forcer pour que le projet sorte rapidement sans comprendre les enjeux techniques ;
- le développeur en sous-traitance et en retard souhaite par dessus tout en terminer avec le projet, et bâcle par conséquent les résolutions de tickets sans se soucier des effets de bord ;
- l'équipe technique reprenant le projet après un audit se demande ce qu'elle va pouvoir faire avec une telle horreur.

À l'inverse, la sur-qualité sur un projet peut également entraîner de la dette technique, en complexifiant le projet et en augmentant donc le niveau requis pour intervenir.

Pour vous aider à vous y retrouver, voici quatre types de dette technique que j'ai identifiés.


Commençons par la plus préjudiciable.

## La dette involontaire

Le manque de connaissances (technique ou métier) ainsi que la mauvaise communication au sein d'une équipe en sont les causes principales et il en résulte de mauvais choix techniques (ergonomie, algorithmie, stratégie, implémentation…).

J'ai ainsi vu des projets dont j'étais incapable de comprendre comment ils pouvaient ne serait-ce que fonctionner mais dont les développeurs et porteurs ne se plaignaient pas et où tout se passait bien. Le projet était à un niveau de qualité satisfaisant pour les personnes concernées.
La prise de conscience de la dette technique a commencé dès lors que plus personne ne se satisfaisait de l'existant (renouvellement des équipes ou nouvelle volonté interne).

Le plus souvent, les personnes qui se retrouvent dans cette situation sont noyées sous un flot quotidien de demandes – parfois contradictoires – et ne connaissent même pas le terme de "dette technique" (situation classique d'un « junior » qui est mal encadré ou trop livré à lui-même). Il leur est donc difficile de se projeter dans des problématiques qu'ils ne maîtrisent pas.

## La dette par négligence volontaire

Les intervenants ne cherchent pas à construire quelque chose de pérenne et ce en connaissance de cause, par négligence.
Deux scénarios différents peuvent causer cette dette :

- les choix techniques sont effectués par désintérêt du projet, situation courante quand le seul indicateur de qualité est le nombre de tickets résolus, et non le nombre de nouveaux *bugs* créés. Cette situation ne favorise pas la qualité globale mais le fonctionnement « à la rustine ». Elle est fréquente quand les intervenants sont complètement détachés des utilisateurs finaux, comme par exemple dans les grands groupes, le secteur public ou bien quand on pratique l'*offshore* ;
- les choix techniques sont effectués non pas pour le bien du projet mais avec la seule volonté de tester quelque chose en tant qu'intervenant. Le symptôme majeur est le test "à tout va" des dernières "tendances", qui paraissent souvent amusantes les premiers jours mais qui se révèlent être assez limitées pour la construction d'un projet complet, géré en équipe.

## La dette assumée

Elle résulte d'un choix technique pris en connaissance de cause à un moment où la volonté de bien faire est contrariée par des circonstances nécessitant de devoir bâcler la qualité : « bon, là on n'a plus le temps, on doit absolument

sortir le projet la semaine prochaine, donc on fait rapidement quelque chose qui marche, on nettoiera après ».

La différence avec la dette par négligence volontaire est l'intention qui la motive et le fait qu'elle est plus ou moins bien vécue.
Ce scénario offre ainsi une chance aux intervenants de connaître leurs points faibles et de les corriger plus tard.

Les risques s'accumulent en revanche si le projet navigue de priorités en priorités et ne prend jamais le temps de nettoyer la dette après coup.

Cette dette est la plus courante et est souvent due aux délais de réalisation très courts des projets, dans un périmètre fonctionnel rarement fixé au préalable.

Voici un exemple de scénario réaliste et les questions qu'il pose tout au long du projet. Cet exemple est issu de l'article « Dette technique 101 » de Maiz Lulkin et traduit par Frank Taillandier sur le blog d'Occitech

> Vous commencez à écrire une application. Au début il n'y a pas besoin de rôles utilisateurs. Tout le monde peut tout faire. À un moment donné vous avez deux permissions différentes pour une action spécifique, comme par exemple un type d'utilisateur qui peut voir des rapports et les autres qui ne peuvent pas. L'équipe technique considère la possibilité de créer un système complet de permissions à part entière. Mais à ce stade, ça ressemble vraiment à quelque chose d'inutilement complexe. Une méthode dans la logique métier et une autre dans la couche de présentation feront le boulot.
>
> Un peu plus tard, un autre cas de figure nécessite de différencier les utilisateurs, puis un autre et encore un autre. À ce stade, les développeurs réalisent que ça commence à être le chaos et que la solution est de refactoriser le code pour avoir un système décent de gestion des permissions. Cette refactorisation de code prendra plus de temps que de simplement ajouter une nouvelle méthode.

Cependant, elle va simplifier le code et permettra aux futures permissions d'être simplement ajoutées par une seule ligne de code ou par une nouvelle entrée dans la base de données.

Le problème est qu'il y a vraiment un besoin commercial d'avoir les permissions actuelles en production d'ici un à deux jours, car cela permettrait à cinq clients potentiels de signer un contrat cette semaine plutôt que la semaine prochaine ou peut-être jamais, s'ils n'apprécient pas que la société n'ait pas répondu favorablement à leur seule demande.

C'est le moment où il faut décider si on contracte de la dette. Toutes les informations nécessaires à cette prise de décision sont connues. Au départ, ajouter une permission demandait 3 points de story (nda : un *point de story* est une approximation du temps relatif requis pour développer une fonctionnalité donnée). Maintenant ça en demande 4. Bientôt cela en représentera 5, 6, qui sait ? La refactorisation complète demande maintenant un effort de 21. Donc la décision, aujourd'hui n'est pas entre 4 et 21 mais entre trois scénarios possibles :

- 4 maintenant (pour la permission), 22 plus tard (la refactorisation est désormais un peu plus compliquée) et quelque chose proche de 0 pour chaque nouvelle permission après ça, accompagné par un léger gain de la productivité générale. Dans ce scénario, l'entreprise a ajouté 5 clients à son portfolio et l'argent arrive tôt;
- 21 maintenant (pour la refactorisation), 0 plus tard (pour la permission); Dans ce scénario, l'entreprise n'as pas ajouté 5 clients à son portfolio de suite, et l'argent arrivera plus tard;
- 4 maintenant (pour la permission), aucune refactorisation du tout, et donc 5 pour les prochaines permissions, puis 6, puis 7… jusqu'à ce que la refactorisation soit suggérée, avec maintenant un coût avoisinant les 50. Dans ce scénario, l'argent est encaissé tôt, mais la prochaine fois cela demandera un travail spécifique pour ajouter des clients et prendra beaucoup plus de temps.

Vu le temps total, c'est toujours mieux de partir sur la meilleure conception possible. Tout comme c'est mieux pour une entreprise d'être en mesure de faire de nouveaux investissements sans avoir besoin d'aller à la banque. Et dans ce genre d'éventualités, partir sur le premier scénario est le plus sage. Une mise en garde cependant : même ce type de compromis ne peut pas être fait en permanence.

## La dette inévitable

La quatrième forme de dette technique résulte de la sédimentation naturelle du code : tout choix impliquant des conséquences, ce qui peut paraître la meilleure option à l'instant T ne le sera peut-être plus à T+1, voire T+10, et il faudra faire avec malgré tout.

Lorsque l'âge du projet augmente, que les intervenants défilent, que les technologies évoluent… même un projet bien mené de A à Z, avec la meilleure conscience des enjeux, subira les effets de l'âge.

J'aime y ajouter une notion supplémentaire : l'obsolescence naturelle.

Partons du postulat que pendant toute la durée du projet, vous allez apprendre des choses et vous améliorer. En regardant en arrière, vous réfléchissez à nouveau et vous vous dites : « si nous devions tout refaire maintenant, c'est comme ça que nous le ferions ».
Il n'y a pas donc pas d'impact pendant la vie du projet.
Mais un projet dont l'architecture n'est plus intéressante peut être une source de démotivation des équipes.

Sur ce projet, rembourser la dette revient donc à le rendre à nouveau intéressant pour les développeurs.

# Identifier les impacts

La dette technique a de nombreuses répercussions sur l'ensemble des acteurs concernés, qu'ils soient logiciels, humains ou organisationnels.

Un projet écrasé par la dette technique (ou en passe de l'être) se distingue par plusieurs indicateurs :

- des développeurs qui s'écroulent, abandonnent toute idée de réussite et travaillent à la rustine ;
- des chefs de projet qui croulent sous les tickets et vivent des réunions pénibles en permanence ;
- des utilisateurs finaux qui perdent toute confiance dans l'outil, c'est-à-dire les humains derrière la technique et donc la technique elle-même ;
- un porteur de projet qui est potentiellement ruiné car il ne peut pas continuer à investir à fonds perdus à ce rythme.

Analysons plus en détails ces différents impacts.

## Impacts techniques

Ils sont la définition même de la dette technique.
Un projet où elle est pharaonique est sclérosé : difficulté de maintenance, *bugs* en cascades, indéterminisme.

Quand le projet ne fonctionne pas, la dette est déjà là et on ne se pose pas de question : on doit passer en cellule de crise. Cette situation est le sujet du dernier chapitre de ce livre.

Quand le projet fonctionne, on commence à se demander « Mais comment cela peut-il marcher ? ».

J'appelle couramment cet effet "tomber en marche". Il s'agit de la pire des situations : si le porteur de projet a le sentiment que le projet est en bon état, il sera quasiment impossible de le convaincre qu'il faut prendre la dette technique en considération. On doit donc "marcher sur des œufs", en essayant d'améliorer les choses sans casser ce qui tient en place.

Dans un projet non essentiel ou de faible envergure (opération ponctuelle, site complètement indépendant), les impacts techniques sont souvent faibles car cloisonnés. Il y a de fortes chances que certains aspects ne soient jamais pris en compte (ne causant pas d'impacts réels, ils sont ignorés).

Sur un projet à petit budget, il y a généralement moins de personnes

impliquées et l'objectif est de produire.
L'effet papillon impacte le projet lui-même ainsi que les relations entre le développeur et le client, mais va rarement plus loin.

Le problème est beaucoup plus grave quand on parle d'un projet central ou pivot. Quand ce dernier donne l'illusion d'être viable et qu'il est fort probable que le site évolue un jour, on peut réellement parler de dette technique.

Une autre caractéristique de la dette technique, c'est qu'elle peut être exponentielle : si une application Web est globalement lente car la performance n'a jamais été prise en compte, un moyen *a priori* simple de l'accélérer est d'augmenter la puissance ou le nombre des serveurs… ce qui accroît la complexité de l'infrastructure d'hébergement, et donc la dette technique.

## Impacts humains

L'art de la fuite. "Trop, c'est trop". Il arrive un moment où, à force d'écoper, le manque d'intérêt et de motivation pour le projet se fait sentir, à tel point que l'abandon peut paraître la seule solution.

J'ai vu des projets où l'équipe client avait complètement démissionné de l'entreprise pour pouvoir s'affranchir d'une situation devenue invivable ; d'autres où des entreprises allaient au procès sur des projets commandités des années auparavant et où les intervenants étaient malmenés en permanence.

### Sur l'équipe technique

Il y a autant de types de développeurs que de développeurs (j'y inclus les intégrateurs, que l'on appelle de plus en plus souvent et à juste titre "développeurs *front*"). Chacun a sa manière d'appréhender les problèmes, d'envisager des solutions, de les faire admettre aux décideurs, et de les implémenter.

J'ai vu beaucoup trop d'entreprises où un développeur est une personne ignorée de tous, dont le rôle s'arrête à développer les nouvelles fonctionnalités et résoudre les tickets, et qui n'est donc pas considérée comme un membre du comité décisionnel.

Ainsi, les développeurs subissent souvent au quotidien le code, les *bugs*, les

quolibets issus de situations qu'ils ne maîtrisent pas ou plus. Survient alors un ras-le-bol démoralisant, quand ils commencent à sérieusement "payer" la dette technique (*bugs* sur *bugs*, tensions clients)… alors que celle-ci était prédite depuis longtemps. La communication et les efforts en vue d'une bonne compréhension par tous sont donc essentiels pour l'ensemble des intervenants du projet.

Un mot tout de même sur les impacts de la dette technique sur le marché de l'emploi. Celui-ci est actuellement beaucoup plus favorable aux développeurs qu'aux employeurs : je connais beaucoup plus d'entreprises qui recrutent que d'entreprises qui licencient. Il est ainsi plus aisé pour un développeur de quitter une entreprise (ou un projet en perdition) et d'aller travailler ailleurs ou sur un projet plus intéressant. Ce n'est pas le cas de tous les intervenants et utilisateurs d'un projet gravement impactés par la dette technique, qui doivent vivre avec et n'ont pas vraiment d'autre choix.

### Sur les équipes support

Ce sont les équipes support qui subissent les conséquences d'un projet infesté de *bugs* sans avoir la possibilité de résoudre les problèmes elles-mêmes.

Il s'agit souvent de personnes de très bonne volonté et qui veulent aider, mais qui sont prises entre des utilisateurs mécontents et les priorités des développeurs, eux-mêmes souvent affectés à des sujets plus "rentables" ou plus visibles que ce qui se passe en coulisse. C'est d'autant plus dommage qu'étant en contact avec les utilisateurs au jour le jour ils font partie de ceux qui connaissent le mieux leurs besoins.

### Sur les équipes clients

Les projets ont souvent un agenda serré, des investisseurs et/ou des utilisateurs qui comptent dessus…

Les porteurs de projet sont les personnes qui ont rédigé et discuté les besoins avec le prestataire retenu, et qui suivront sa réalisation. C'est cette relation qui est très fortement source de dette technique, comme nous le verrons dans le chapitre suivant.

En tant que coresponsable de la dette technique, les impacts peuvent être multiples, mais aussi être assez extrêmes, allant de la mise au placard au

licenciement.
Être responsable d'un projet difficile est souvent à double tranchant.

Quant aux équipes clients qui sont utilisatrices du projet, les impacts s'apparentent alors au point précédent « équipe support ».

## Impacts organisationnels

Le projet central est sclérosé ? Les équipes découvrent des *bugs* partout et n'en peuvent plus ?
Quelles seront les conséquences à l'échelle organisationnelle ?

Si la technique est un des éléments centraux et que rien ne marche :

- plus aucun planning ne peut être tenu ;
- on n'ose plus prendre de nouveaux marchés car les ressources sont déjà utilisées à réparer les soucis existants ;
- si de nouvelles ressources sont ajoutées, elles sont souvent mal formées et participent au projet dans le stress ;
- il n'y a aucune visibilité sur le retour à un mode fonctionnel ;
- le projet sort trop tard et « loupe le coche » ;
- les utilisateurs ne sont pas convaincus par le projet et ne l'utilisent pas ;
- les gestionnaires du projet côté client désespèrent et quittent l'entreprise.

Des sociétés engagées contractuellement à la réalisation d'un projet peuvent également finir par couler si elles ne maîtrisent pas le périmètre et la construction du projet.
Des sociétés porteuses de projet peuvent déposer le bilan à cause de leur dette technique.

C'est pour tâcher d'éviter ce genre de désastres que je rédige ce livre ; je vous invite donc à poursuivre vers les prochains chapitres : Prévenir et Résoudre.

# Prévenir

À l'issue du chapitre précédent, nous sommes donc en mesure d'identifier la dette technique et de prendre la mesure de ses impacts dans un projet.

N'oubliez pas qu'avoir une dette technique n'est pas nécessairement synonyme de problèmes incommensurables en devenir.
Mais sans totalement s'en prévenir, il convient *a minima* de se demander comment la gérer au mieux afin d'éviter un retour de flamme.

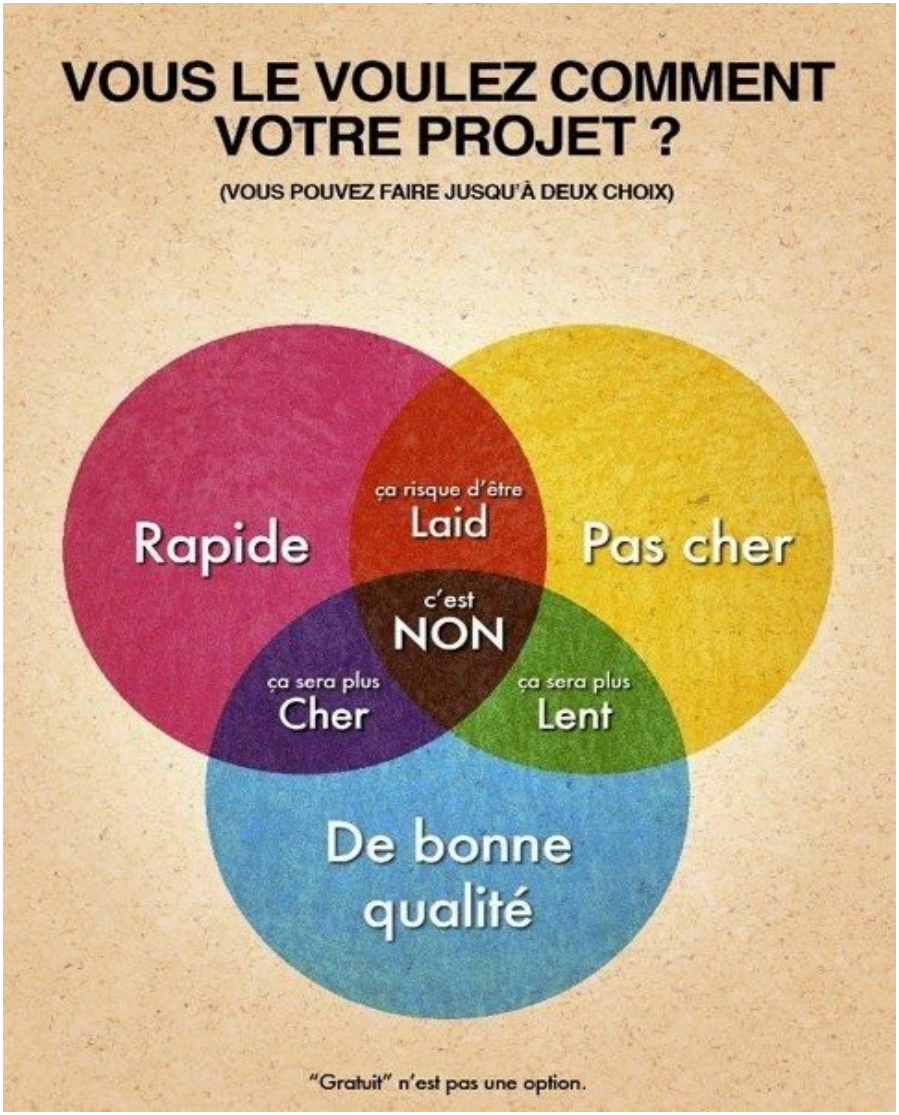
La situation que nous souhaitons tous éviter est le moment où le coût de développement d'une nouvelle fonctionnalité ou d'un correctif sera supérieur aux gains de cette fonctionnalité.

Nous allons donc étudier des moyens de prévention qui vous permettront d'allonger de façon conséquente la durée de vie de vos projets et ainsi de vous concentrer sur leur perfectionnement et leur plus-value.

## Est-ce possible ? Nécessaire ?

Il faut bien prendre conscience que la dette technique est inévitable : pour construire un projet, il est obligatoire de faire des choix architecturaux. Ces choix auront obligatoirement un impact sur la maintenance et l'évolutivité du projet.

Le simple fait d'être soumis à des contraintes (humaines, budgétaires, temporelles) pousse à faire des choix et des concessions à un moment donné sur l'un des axes prix / qualité / délai.




Le prix et le délai étant les facteurs les moins négociables, c'est le plus souvent sur la qualité que les projets se voient amputés, ce qui provoque la dette

technique.
La qualité étant difficilement quantifiable, sa gestion pose un véritable problème. Les décideurs ne comprennent que trop peu les enjeux techniques de la qualité (quand ils s'y intéressent). Sachant que la dette technique est inévitable, où placer le curseur de la qualité minimale acceptable ? Selon lse contextes de projet, j'accepte ou non des solutions techniques « sales ».
Trop de qualité pour un projet simple le complexifie peut-être inutilement et l'abus de qualité (surqualité) est d'ailleurs une dette à part entière.

# Comment prévenir ?

Tout l'enjeu repose donc sur les deux paradigmes suivants :

**Comment savoir *quoi* développer :** éviter les silos de prise de décisions, échanger avec toutes les personnes impactées sur les raisons qui motivent ces décisions et leurs priorités.

**Comment faire pour que ce soit *bien* développé :** travailler en équipe motivée mais pragmatique, entretenir une culture d'équipe et avancer ensemble, apprendre à gérer le temps et à atteindre un niveau homogène de qualité.

Cela passe avant tout par une compréhension minimale des contraintes de chaque intervenant du projet (budget, délai, technique…).

## Comprendre le projet

> The consultant's job is not to give the client what he wants. Rather, your job is to persuade the client to want the right things.
> — John Arundel (@bitfield https://twitter.com/bitfield/status/471986840644222977)

Un besoin exprimé n'est pas forcément à développer tel que l'indique la première version de l'expression du besoin. Il doit pouvoir s'adapter.
Il faut faire la différence entre les besoins du client et les besoins des utilisateurs finaux à qui s'adresse le projet.

Le nombre d'intervenants nécessaire pour construire un projet augmentant, il est impératif d'établir des ponts de compréhension. C'est en ça que la culture

d'entreprise – point traité plus loin dans ce chapitre – est fondamentale.

Voici quelques bonnes pratiques pour assurer une bonne compréhension du projet par toutes les personnes impliquées.

## Fixer l'objectif final du projet/site

Vendre des produits, mettre en avant une marque, générer des appels entrants… Fixer l'objectif final est le premier point à décider entre le porteur du projet et ceux qui vont le réaliser, avant même de lire quoi que ce soit. Il peut être nécessaire d'aider ce responsable à définir les objectifs.

« Quel est le but du projet ? », « Pourquoi ? », « En quoi va-t-il être meilleur que l'existant ? », « Quelle est la stratégie pour atteindre cet objectif ? ».

Traditionnellement, on utilise la notion de Produit Minimum Viable (MVP) qui caractérise un produit qui offre juste l'essentiel technique pour être montrable et communiquer.
J'aimerais y opposer la notion de « Produit Minimum Coup de Cœur » de Maiz Lulkin, dans laquelle le produit, même s'il ne répond pas à toutes les attentes techniques, permet de déclencher une réelle adhésion, un affect, chez les utilisateurs.

Ces questions doivent aboutir sur des actions **mesurables** qui permettront d'accomplir ces objectifs. Ces actions sont décomposées en étapes que doivent effectuer les visiteurs que l'on souhaite convertir en utilisateurs. La réflexion sur ces actions s'appelle le *design*, au sens anglophone du terme, c'est à dire la conception, et non pas uniquement le look final.

La mesure de la conversion est primordiale et doit se faire sur les objectifs définis pour le projet, car elle permet d'évaluer la performance de la plateforme et de pouvoir tester l'impact des itérations. On évite ainsi une grande part de préjugés pour s'appuyer sur du concret.

Le mieux étant souvent l'ennemi du bien, il faut faire attention à la dispersion en poussant les détails trop loin sur des fausses priorités qui n'ont aucun impact, voire un impact négatif.

Rien qu'avec ces points, on découvre souvent que le cahier des charges est *déjà* un boulet pour le projet et qu'il ne représente pas ce qui est nécessaire.

## Être transparent sur cet objectif

Si les intervenants (techniques, commerciaux…) du projet ne comprennent pas ce qu'ils font et pourquoi ils le font, vous pouvez être sûr que le résultat ne sera pas satisfaisant.
Autant mettre toutes les chances du côté du projet en écrivant ses objectifs au mur et en concentrant tous les efforts dessus.

De la même manière, les intervenants doivent appréhender les impacts. Cependant, les choix techniques et leurs solutions amènent parfois également tellement de contraintes qu'il est souvent de mise de les cacher.
Là encore, privilégiez la transparence sur les objectifs.

Un lexique / glossaire peut ainsi être mis en place afin que toutes les personnes concernées parlent le même langage et que les contraintes imposées à chacun soient bien validées en connaissance de cause.

Les humains sont plus performants sur les tâches qu'ils comprennent et qui les intéressent. Aidez l'équipe à s'impliquer dans le projet, notamment en donnant de la valeur aux retours que les intervenants peuvent faire, et vous constaterez une amélioration de la qualité réelle et ressentie du projet.

Ainsi, à chaque nouvelle requête, il faut s'interroger sur son lien avec les objectifs ultimes du projet, et itérer dessus pour construire la meilleure solution pour votre projet. Ou, au contraire, ne pas la construire et repousser cette réflexion à plus tard.

## Aller à l'essentiel dès le départ

Il faut ensuite continuer sur l'essentiel, et itérer encore et encore dessus. Tant que celui-ci n'est pas parfaitement mesuré, testé, éprouvé, itéré, ne pensez même pas à l'ajout de nouvelles fonctionnalités.

La méthode des **cinq pourquoi** (https://fr.wikipedia.org/wiki/Cinq_pourquoi) est une méthode de résolution de problèmes qui permet, en quelques questions « pourquoi ? » successives, de remonter à la source du problème ou de la demande qui est la cause principale de la défaillance. Cette méthode est souvent utilisée dans les systèmes de qualité.
Elle est particulièrement simple à mettre en place et à systématiser. Y compris envers soi-même.

Voici un exemple d'un projet réalisé il y a quelques années. Sa date de sortie était fixée par une conférence de presse ; son but était de récolter des contributions d'internautes afin d'améliorer la prise de décision. Bien que le délai ait été excessivement court, il fallait développer de nombreuses fonctionnalités, associées à de multiples points de détails amenés par des phrases du type « on ne peut pas se permettre de ». L'un des points les plus discutés concernait les règles de gestion de la modération. Les pistes étaient nombreuses, notamment pour faire participer la communauté de manière transparente. Ces discussions, le développement, le *debug* sur des données fictives…nous auraient fait réfléchir à des scénarios fictifs, par crainte de ce qu'il pouvait arriver. Ce sujet a rapidement été clos car je ne m'estimais pas capable de faire quelques chose de correct et que la problématique de modération n'arrive de toute façon qu'une fois qu'il y a du contenu… et qu'on ne peut pas avoir de contenu sans site en ligne. La priorité était donc de sortir ce site.
Deux ans après, ce sujet n'est toujours pas développé car le besoin ne s'en est jamais fait sentir : les rares fois où un contenu était litigieux, la situation était traitée individuellement et manuellement.
Résultat : le temps cumulé de modération sur deux ans est inférieur à la durée de la première discussion que nous avons eue sur le sujet.

Cet exemple nous montre qu'il ne faut pas toujours vouloir anticiper des situations complexes, le plus souvent par crainte, car elles sont finalement peu fréquentes. Vous pouvez également être certain que quand le besoin se fera sentir, ce ne sera pas la solution datant de deux ans qui sera sélectionnée car les pratiques et outils auront évolué entre temps.

De plus, tout ce code ajouté aurait amplifié la complexité du projet qu'il aurait ensuite fallu maintenir. Cette maintenance, et la multiplication des options qu'elle implique, complique et donc ralentit les itérations ultérieures.

Dire non, ou même être en position d'argumenter, est parfois difficile voire risqué pour un prestataire. Il faut donc être dans un contexte de prestation qui le permette.

## Mieux contractualiser, pour l'avenir

Vous l'avez compris, nous ne nous intéressons qu'aux projets qui ont une durée de vie au-delà du ponctuel et qui devront donc être maintenus.

Dans ce contexte, un projet réussi est donc un projet qui a de l'avenir, dont les fondations vont permettre de poser les briques pour le faire avancer, le tout avec une équipe soudée et motivée.

Le plus souvent, dans une relation prestataire/client classique, ce dernier souhaite à la fois avoir la maîtrise de son budget et "en avoir pour son argent". C'est pour cela que par principe ce qui est souhaité est traduit dans un langage compréhensible par des prestataires : le cahier des charges.

Or, évaluer un projet quand on ne comprend pas forcément le métier concerné est un exercice réellement difficile. Cela revient souvent à jouer au "Juste Prix" sur le budget prêt à être dépensé, en essayant de ne pas s'engager dans un projet qui amènerait à être déficitaire tout en prévoyant de se rattraper sur la maintenance et les avenants…

Une fois le projet évalué, le client sélectionne le prestataire selon la logique de concurrence du prix et la confiance accordée à la conception. Les développements peuvent alors commencer.

Pourtant, cette situation classique composée d'un cahier des charges et d'un contrat "au forfait" est peu souhaitable, pour la simple et bonne raison qu'elle empêche toute capacité d'adaptation.

Ainsi, le contrat "au forfait" pousse le plus souvent à en vouloir le plus possible **avant** la mise en ligne, empêchant toute phase d'expérimentation. Or, la logique voudrait en fait que la mise en ligne ne représente qu'une partie – certes conséquente – du développement, car c'est à ce moment-là que l'on apprend réellement comment les utilisateurs se comportent face au site.

De plus, le fondement même d'un contrat, c'est d'établir les engagements de chaque partie.
Le réflexe en cas d'adversité est de se référer au contrat signé pour forcer l'autre partie à respecter ses engagements.
Dans ce cas, le risque est que chaque partie en cherche toutes les failles pour forcer l'autre à faire le maximum pendant qu'elle fait elle-même le minimum vital pour en respecter les clauses.
S'ensuivra alors une phase de *debug* et au mieux, de Tierce Maintenance Applicative (TMA).

Enfin, comme les processus entre la rédaction du cahier des charges, la validation par les responsables, la soumission de l'appel d'offre, l'attente des

réponses, leurs évaluations, le début de la prestation, etc. sont très longs : le cahier des charges sera certainement déjà obsolète au démarrage du projet.

Tous les points que nous venons de voir montrent qu'il faut éviter certaines situations dans lesquelles les relations entre un porteur de projet et son/ses prestataires sont uniquement liées par la réalisation d'un contrat figé.

Alors comment faire pour que l'on puisse à la fois obtenir un engagement fort d'une équipe avec un budget maîtrisé et "en avoir pour son argent" ?

Je n'ai pas de recette magique, mais voici quelques conseils :

- ne pas penser qu'« en avoir pour son argent » correspond **exactement** à ce que l'on a en tête. Laisser de la place à l'adaptation peut souvent faire gagner beaucoup de temps de développement (et donc d'argent) ;
- sortir le plus rapidement possible une version présentable, et itérer dessus en fonction des réactions : on se rend souvent compte que ce qui semblait indispensable ne l'est plus ;
- ne pas exagérer les besoins "bloquants" ;
- les personnes en face sont *a priori* professionnelles, les laisser faire leur travail et ne pas les infantiliser ;
- mieux, s'appuyer sur leurs compétences métier car elles sont justement les plus à même d'aider et d'orienter dans la prise de décisions techniques : il faut savoir lâcher prise et capitaliser sur les compétences de chacun.

Un bon point de départ est donc d'établir un contrat d'engagement de moyens, et non de résultats. Des résultats peuvent être fixés, mais ils n'engagent pas l'intégralité de la durée du contrat, et sont des engagements *a minima*, laissant de la marge de manœuvre aux deux parties.

Des deux côtés des efforts doivent être faits, et il ne faut pas s'attendre à du "travail gratuit" (par exemple en le confiant à la personne la moins chère de l'entreprise, avec moins d'encadrement qu'un travail payant). Le travail gratuit ou sous-estimé est quasiment la garantie d'un travail bâclé.

Tous ces conseils montrent qu'il s'agit avant tout d'une suite d'échanges humains qui feront que le projet sera lancé sur de bonnes bases… ou non.

## L'Humain avant tout

Un projet, c'est donc avant tout une aventure humaine : ce sont des problèmes identifiés par des humains, étudiés par des humains, et dont la solution est conçue, réalisée et gérée par des humains.

Nous pouvons donc sereinement avancer que, quel que soit le projet, les risques sont avant tout humains. Les défaillances techniques sont bien plus souvent dues à des erreurs de communication et de compréhension qu'à du code mal développé.

De ce fait, de nombreux paramètres sont à prendre en compte dans les choix humains concernant votre projet.

L'erreur la plus classique porte sur le rôle de l'équipe technique : elle doit *co-élaborer le projet*, et non pas être le dernier maillon de la chaîne et n'avoir "plus qu'à implémenter" des décisions toutes déjà figées. Cela passe potentiellement par des contrats différents de l'obligation de résultats sur cahier des charges traditionnel, comme vu précédemment. Nous comprenons donc que la relation entre tous les intervenants du projet et le client doit plus s'apparenter à une logique de partenariat de réalisation qu'à une logique de hiérarchie client / prestataire.

En extrapolant, nous en arrivons à définir le rôle du responsable technique du projet. Selon la taille de l'équipe, il s'agira du seul développeur, du développeur principal ou du directeur technique.
Cette personne se doit d'être la personne "la plus communicante" et non pas "la plus capable".

Si vous voulez prévenir au maximum la dette technique, il faut également que les décisionnaires aient conscience des impacts sur la technique que peuvent avoir les orientations retenues. Beaucoup de décisions peuvent être adaptées si la communication est fluide entre les personnes orientées technique et les autres.

Ces points seront les fers de lance de votre stratégie de prévention : si un projet n'est pas cohérent dans sa conceptualisation, il ne pourra qu'être techniquement bancal. Il faut donc parvenir à travailler en bonne intelligence pour que la conception soit la plus efficiente possible.

Cela commence par la constitution d'une équipe de confiance.

## Culture d'équipe

> Une équipe, c'est souvent bien plus que la somme de ses individualités.
> – Eric Thomas, Président de l'Affa (http://cocorico-carioca.blogs.lequipe.fr/2014/01/eric-thomas-le-football-est-en-train-de-mourir-dans-lindifference-generale/)

Un projet est souvent tellement complexe qu'il devient difficile, voire impossible, d'être en mesure de le réaliser seul de bout en bout.
Il y aura donc au minimum le porteur du projet, et un ou plusieurs intervenants : développeur, infogérant, designer, intégrateur, ergonome, expert, consultant, et j'en passe.
Quand le nombre d'intervenants augmente, et pour que ces personnes puissent collaborer pour répondre à un besoin, il faut des "coordinateurs" : des chefs de projet, qui ne sont pas là pour produire mais pour gérer.

Sauf cas exceptionnel, ces personnes ont chacune leur bagage propre (éducation, âge, mode de vie, rapport au travail), les rendant naturellement peu aptes à se comprendre.
Il est donc indispensable de mettre en place le maximum de ponts entre elles pour faciliter leurs échanges.

Pour cela, il faut développer ce qui s'appelle une "Culture d'entreprise".

## Culture d'entreprise

Selon Wikipédia, la culture d'entreprise est :

> l'ensemble des éléments particuliers qui expliquent les bases du fonctionnement d'une organisation […], dans un certain sens, un ensemble de valeurs, de mythes, de rites, de tabous et de signes partagés par la majorité des intervenants
> — https://fr.wikipedia.org/wiki/Culture_d%27entreprise – tiré de *Culture d'entreprise*, par Christophe Durand, Jean-François Fili, Audrey Hénault, 2000
> http://culture.entreprise.free.fr/#_Toc476995192.

En cela, le développement de projet est assez proche d'un sport d'équipe. Or,

dans un sport d'équipe, un des éléments les plus privilégiés est l'*esprit* d'équipe, qui ne doit pas être sacrifié.

Recruter est une des tâches les plus ardues auxquelles une entreprise fait face. Recruter tout en gardant en tête une stratégie et une culture d'entreprise l'est encore plus.

Même les entreprises comme Google, Github, ou "la dernière *startup* à la mode" rencontrent des difficultés pour recruter. Il est donc d'autant plus difficile de bien recruter quand votre entreprise n'est pas aussi attirante.

Recruter une personne talentueuse dans son domaine est délicat car souvent :

- elle aime être mise au défi ;
- elle complexifie un besoin pour qu'il s'adapte à la technologie qu'elle a envie d'employer (et non le contraire) ;
- elle manque de pragmatisme.

À l'inverse, recruter un exécutant (un *soldat*) est plus facile… mais :

- il ne faut pas le laisser faire les choix d'architecture *seul* : il ne se sentira pas à l'aise, doutera et cela se répercutera rapidement dans ses choix ;
- il faut limiter le nombre de soldats afin qu'ils puissent tourner sur les projets tout en connaissant les autres interlocuteurs.
  Tout projet informatique sérieux doit commencer par un dénigrement systématique du travail effectué par les développeurs précédents !
  — Tous les développeurs, y compris envers eux-mêmes !

Reprendre un projet après une personne que l'on ne connaît pas, c'est la garantie de critiquer son travail, quasiment par principe, pour montrer que ce n'est pas une façon correcte de travailler et que notre méthode est meilleure. Évitez donc de créer cette situation en composant des équipes soudées, et favorisez de véritables transitions en cas de départ anticipé.

Il n'y a pas de profil parfait dans une équipe, qui est un ensemble de personnalités propres. Évitez les extrêmes, et embauchez des personnes qui ont les pieds sur terre et qui apprécient ce qu'ils font pendant la journée. Offrez-leur le temps et l'accompagnement nécessaires pour qu'ils puissent faire de la veille et entretenir leur curiosité. De nombreuses conférences existent et sont passionnantes, autant pour leurs contenus que pour la pluralité des rencontres possibles. Proches de chez vous existent aussi des

rassemblements, organisés sous la forme de Meetup ou autre.

Attention toutefois à ceux qui ne sortent jamais du même secteur d'activité nuit et jour : il faut garder le sens des priorités. Ne vous attendez pas à ce que les membres de votre équipe travaillent gratuitement en dehors du boulot "pour le boulot" : la priorité ne doit pas toujours être donnée au travail.

Autre chose, il ne faut *jamais* arriver à une situation sur un projet où le seul objectif est d'occuper un développeur.
Les équipes techniques sont là en support ou en recherche. Si elles ne font rien à un moment donné, ne leur inventez pas des besoins dans la précipitation, mais laissez-les expérimenter et s'auto-former pour qu'elles apprennent de nouvelles techniques qu'elles pourront réutiliser sur les futurs projets.

Créer une culture d'entreprise est difficile, la maintenir l'est tout autant.

Un premier moyen peut être de sensibiliser les équipes au partage de connaissances.

Vous pouvez par exemple organiser des mini conférences internes, ou faire venir des personnes externes le temps d'un repas pour qu'elles vous parlent d'un sujet qui leur tient à cœur (voir l'initiative brown bags lunch).
Trouver du temps au sein de l'entreprise est nécessaire pour faire émerger l'intelligence collective. Faire travailler moins permet, paradoxalement, de produire plus.

L'apport d'un regard nouveau étant souvent bénéfique, il vous faudra donc évaluer le juste ratio entre équipe habituée à travailler ensemble et apport de sang neuf.

## Choisir ses batailles

Nous l'avons vu, la dette technique n'est pas uniquement le fait des techniciens. En fonction du recrutement, elle aura un impact sur la sensibilité de tous les membres du projet : les graphistes, les chefs de projet, les développeurs, le client, les testeurs… et le client.

Oui, il faut savoir *recruter* son client.
Votre équipe a sa (ses) culture(s), ses spécialités, ses points faibles, sa propre vélocité, et tous les projets ne sont pas bons à prendre :

- quel prestataire ne s'est jamais vu demander de développer « le nouveau Facebook, révolutionnaire » ? Ces demandes sont à 99 % déraisonnables et le porteur de projet ne peut qu'être déçu. Non-adéquation avec le marché, demandes complètement « à côté de la plaque »… il faut savoir être lucide sur la crédibilité du projet. Quand les demandes techniques sont trop fantasques ou qu'un projet n'a vraiment aucun avenir, la qualité et l'implication de l'équipe seront rarement au rendez-vous. Il ne faut pas accepter toutes les missions et tous les clients.
- si vous êtes en train d'investir sur une technologie, ce n'est pas pour autant qu'il faut à tout prix la vendre au premier client venu ;
- si le client exige du *pixel perfect* en 2014, il y a de fortes chances qu'il soit ingérable quand il se rendra compte du milliard de scénarios possibles (périphériques * navigateurs * préférences utilisateurs) ;
- le client porte la connaissance du métier qu'il est censé représenter. De même, la conception web est un métier à part entière et l'adaptation d'un métier aux technologies du Web ne se fait pas en une simple traduction d'anciens procédés. Il faut donc estimer si le client est à même d'évoluer et d'écouter les conseils voire de repenser toute ou partie de ses processus. Dans le cas contraire, le prestataire et le contrat ne concerneront qu'une exécution et il sera difficile de pousser en avant les bonnes pratiques qui permettront la mise en place de procédés qualitatifs.

De même que les critères de choix d'un prestataire sont multiples et doivent s'adapter au projet, un prestataire ne doit pas nécessairement accepter tous les projets.
Il ne doit pas hésiter à communiquer avec le porteur du projet afin de cerner l'état d'esprit dans lequel le projet doit être mené et ne pas hésiter à *fuir* une situation qu'il pressent mal.

Il faut savoir éduquer et s'éduquer soi-même sur le projet et ses spécificités avant de se lancer.
Nous verrons dans la partie "Gestion de projet" que la sélection de ses batailles est un procédé critique : il faut savoir lâcher du lest si vous voulez que l'autre partie accepte une de vos demandes fortes. On ne peut pas toujours tout avoir.

## À bon ouvrier, bons outils

Un autre point qui ne concerne pas le recrutement, mais qui mérite d'être mentionné : les conditions matérielles de travail *comptent*. Le confort de travail pour un développeur a un prix ridicule en comparaison du gain de temps et de productivité qui en découleront.
Sans même parler de l'effet bénéfique de la valorisation pour la personne ainsi "choyée".

Exemples concrets :

- Sur un projet où j'intervenais en indépendant, j'exécutais le même script qu'une autre personne de l'équipe. Avec mon équipement et ma configuration, un script prenait vingt minutes alors qu'il prenait cinq heures sur la machine d'un autre intervenant… ;
- Sur un autre projet, un graphiste était obligé de fermer son navigateur pour pouvoir ouvrir son logiciel de design. Pratique… ;
- Dans un ancien emploi, je préférais travailler avec ma machine personnelle plutôt que mon matériel attribué : je n'aimais pas m'abîmer les yeux sur le mauvais écran ni être freiné par un disque qui "grattait" alors que mon ordinateur était équipé d'un des tout derniers disques flash ;
- Une bonne chaise est agréable. Si une personne souhaite travailler debout, c'est bien aussi, favorisez-le ;
- Si possible, ayez une douche dans vos locaux pour motiver votre équipe à venir en faisant du sport ou à en faire pendant une pause.

On pourrait penser que je m'égare sur le sujet de la dette technique. Mais pas tant que ça : si vous voulez éviter de rembourser trop de dettes plus tard, investir sur l'humain ne peut qu'améliorer les choses.

## Quelques points d'attention

Attention aux bourreaux de travail, et au zèle en général. Bien que des

exploits ponctuels puissent être très pratiques et appréciables, avoir en permanence quelqu'un qui en fait plus est dangereux. Ces efforts cumulés ont forcément un coût (physique et intellectuel pour celui qui les a fournis, humain dans ses rapports aux autres, etc.).

> L'homme absurde est celui qui ne change jamais.
> — Georges Clémenceau, *Discours au Sénat*, 22/07/1917

Se méfier du "phénomène de l'expert" : rester frais et naïf peut avoir du bon, cela permet de tester des choses sans forcément avoir toutes les clés pour l'avenir. Et si les tests fonctionnent, il sera toujours temps d'aller recruter de nouvelles têtes bien faites.

Les personnes qui vont réaliser l'aspect technique, au même titre que celles qui en réaliseront l'aspect visuel, sont des *artisans* : le code n'est pas un travail qui s'effectue à la chaîne, et les demandes effectuées doivent donc prendre en compte ce paramètre.
D'un côté comme de l'autre, il faut avoir cette passion qui fait que l'on pousse à la qualité tout en étant pragmatique et en acceptant les compromis.

Par ailleurs, la confiance n'exclut pas le contrôle. Ce n'est pas parce que l'on fait confiance à quelqu'un (ou à soi-même) que cette personne est infaillible. Une revue de code est toujours bénéfique à tous les niveaux. En effet, elle permet de contrôler que ce que l'on peut voir (« le site est joli et semble fonctionner ») correspond à un niveau de qualité technique cohérent, et non à une suite de *patchs* qui ne valident qu'un seul scénario.

Pour gérer au mieux tous ces aspects humains, il faut donc mettre en place les trois axes fondamentaux d'une bonne gouvernance : la transparence, la participation et la collaboration.

## Gestion d'équipe / gestion de projet

Tout en essayant d'éviter un énième plaidoyer de l'agilité, nous allons aborder quelques-uns de ses avantages ainsi que les inconvénients de certains états d'esprit.

Tout d'abord, il faut différencier la gestion de projet de la gestion d'une équipe technique. Là où l'une a trait à la communication entre un client et son projet et en assure l'aspect financier et "politique", l'autre concerne la

cohésion technique et la motivation de l'équipe et est menée par la personne technique de confiance sur laquelle l'équipe se repose.

## Hiérarchie

> Demandons-leur des estimations, que nous considérerons ensuite comme des délais.
> — Technical debt 101, Traduction sur le blog d'Occitech

Il existe, notamment en France, un rapport à la hiérarchie que l'on qualifiera de "particulier". Ainsi, une étude scandinave posant la question " Est-ce que la voix de votre manager a toujours raison ? " — analysant les impacts de la parole d'un manager sur une décision technique — montre des réponses très différentes en France de celles observées dans de nombreux autres pays.
Il en résulte que culturellement, en France, le manager est celui qui impose sa vision alors que dans les pays scandinaves, il n'est qu'un « facilitateur ».
D'ailleurs, selon la convention collective qui régit nos métiers en France, la Syntec, la gestion de projet a plus d'importance hiérarchique (et financière) que l'expertise technique…

Dans certains projets, on peut constater de plus en plus que les clients, *via* les porteurs du projet, s'entourent d'experts techniques indépendants afin de s'affranchir d'un grand nombre d'erreurs triviales. Ainsi, cette personne externe et non partisane pourra :

- favoriser la compréhension du fonctionnement du prestataire sélectionné ;
- aider à juger de la qualité technique quasiment en direct, ou en tout cas bien avant la publication finale ;
- éviter au porteur de projet de faire des demandes allant dans le sens contraire de celles déjà passées et qui impliquent de véritables challenges techniques ;
- être force de proposition sur des solutions qui améliorent réellement le projet, en faveur ou en défaveur des deux parties, en toute indépendance.

Il est vrai qu'il s'agit surtout de projets de grande envergure qui peuvent s'offrir les services d'experts indépendants, mais nous pouvons remarquer

qu'aujourd'hui plus de demandes d'audit de code sont faites en amont de la livraison finale qu'il y a quelques années.
Et ce sans que le projet soit pour autant dans une situation conflictuelle, juste pour valider que les bases sont saines et qu'elles permettront de construire sans contraintes fortes.

## Travailler en équipe

### Équipe projet

Un projet doit se mener en tant qu'équipe. Et pas seulement "équipe technique", mais équipe technique + équipe client + équipe créative, etc. Les décisions ne doivent pas être imposées, mais acceptées par tous les intervenants. Le danger, pour une équipe ou une personne, d'accepter une décision forcée, c'est de généraliser cette attitude dans le futur. Ce n'est alors plus un travail d'équipe mais un travail d'exécutant, beaucoup moins valorisant pour l'individu compétent et motivé. La motivation disparaîtra et au mieux vous aurez un développeur qui en fera le moins possible, au "pire" un développeur qui s'en ira.

> Un manager c'est celui qui pense qu'avec neuf femmes on peut faire un bébé en un mois.
> — Adage populaire dans nos métiers

Il faut éviter l'« agilité par le *management* », où les processus « agiles » sont respectés mais où toutes les décisions sont prises par la direction (*top-down*).
Le principe de l'agilité c'est que les personnes qui produisent font partie du processus décisionnel, pour une **co-construction** (*bottom-up*).
Ils sont donc en droit de répondre « non » à des demandes, ou sa version plus diplomate : « pas maintenant ».

L'idéal, c'est donc quand le *product owner* n'est pas le chef de projet et qu'il y a un expert technique référent.

### Équipe technique

Il ne faut pas travailler seul, surtout quand on est indépendant. Je comprends qu'être indépendant est avant tout un choix de vie. Mais il est alors plus

difficile de se remettre en question (ou au contraire, de ne pas se remettre en question excessivement), d'apprendre de nouvelles choses, et il est à l'inverse plus facile de se démotiver. Travailler seul c'est aussi n'avoir aucun garde-fou ni aucune protection face à la difficulté de certains projets.

Ce n'est pas un hasard si les indépendants font souvent partie de groupements d'indépendants ou de collectifs comme (Paris|Sud) Web. C'est un petit monde où beaucoup de bonnes volontés sont échangées et qui est fort motivant. Mais le retour à la réalité des projets est souvent difficile.

C'est pour cela que je recommande d'éviter autant que possible de répondre seul à des projets. Il vaut mieux se répartir le travail — même à un faible pourcentage – afin que toute la compréhension et l'exécution d'un projet ne reposent pas sur les épaules d'une seule et unique personne. Cela permet une connaissance partagée du code (point traité en détail plus loin).

Exemple concret tiré de mon expérience : je participais en tant qu'indépendant à un projet (que je n'aurais jamais dû accepter, cf. plus haut) avec un client pénible, quand je suis tombé malade. M'inquiéter à propos de ce projet et savoir que, même après ma guérison, j'allais devoir m'y remettre ne m'ont pas aidé dans ma convalescence, d'autant plus que le projet n'a pas souffert de mon absence.

À l'opposé, j'ai eu il y a peu de temps l'occasion d'aider un ami indépendant en relisant ses *pull requests*, ce qui me permettait de discuter avec lui (même brièvement) de points d'architecture que je n'aurais pas conçus de la même manière et de partager quelques astuces que j'avais pu voir dans d'autres projets. Cet ami m'a même donné des permissions pour que je puisse prendre la main en cas de problème. C'est rassurant pour lui et ça lui permet de savoir qu'il est moins indispensable.

Être indispensable n'est jamais souhaitable.

## Se rendre facultatif

Autrement dit : ne pas se rendre indispensable. Quand je parle "d'être indispensable" j'entends "être l'unique détenteur d'une connaissance nécessaire à un projet."

Il y a deux causes principales à cette situation :

- un calcul sur le court terme pour asseoir sa position, garantie d'une situation conflictuelle à l'avenir. Si la direction veut se débarrasser de vous, elle y arrivera forcément. Maintenir une position dite « de prise d'otage » finit rarement bien pour le ravisseur ;
- l'accident, en étant le dernier d'une équipe à rester.

Dans l'économie actuelle et vu le marché de l'emploi immensément favorable aux développeurs, il est risible de s'astreindre à essayer d'être "indispensable". Au contraire, mieux vaut être le plus pragmatique et le plus transparent possible sur le travail effectué. En transmettant la connaissance au mieux afin de réduire le niveau nécessaire pour poursuivre votre travail, vous augmenterez ainsi les chances de pouvoir passer la main et aller vous intéresser à de nouveaux projets.

## Favoriser les échanges

Comme nous l'avons déjà vu, les échanges sont indispensables. Pour les favoriser, des outils doivent absolument être mis en place et tenus à jour :

- pour les questions courantes, plusieurs *chats* internes dont les historiques doivent être consultables et qui regroupent tous les membres des équipes, favorisant ainsi les rapports asynchrones sur quantité de sujets non vitaux :
  - un pour les messages d'annonce ;
  - un pour le suivi des tickets / *branches* / *pull requests* / *commits* ;
  - un pour le suivi des incidents ;
  - un pour les discussions générales ;
- un Wiki (ou assimilé) pour la documentation des enjeux ;
- un tableau de bord, qui permet de suivre en direct certaines métriques (passage des tests, taux de transformation, chiffre d'affaires, etc.).

Attention toutefois à la quantité d'informations échangées : trop d'informations forment du bruit et le bruit assourdit… ce qui finit toujours par nuire à la communication.

Pour responsabiliser et motiver tous les membres de vos équipes, pensez à partager l'accès au *monitoring* et aux tableaux de bord : il est très gratifiant

de voir l'impact de chaque nouvelle version sur les graphiques (moins d'erreurs, plus de conversion, etc.).

N'oubliez pas les échanges physiques, comme par exemple :

- *standup meeting* tous les matins pour parler de la veille et du jour même ;
- des *kick-offs* de lancement de *sprint*.

Un autre point important est de pouvoir être capable de comprendre après coup le suivi des échanges et de rapidement rattraper le retard si on doit s'absenter du projet.

Un exemple de prise en compte de ces problématiques, assez révélateur, est le mouvement « Devops ».

> Devops est un mouvement visant à réduire la friction organisationnelle entre les "devs" (chargés de faire évoluer le système d'information) et les "ops" (chargés d'exploiter les applications existantes).
> Ce que l'on pourrait résumer en travailler ensemble pour produire de la valeur pour l'entreprise. Dans la majorité des entreprises, la valeur sera économique mais pour d'autres elle sera sociale ou morale.

Cette définition montre bien ce que je propose dans cette section : réduire les points de blocage pour plus de cohésion et de compréhension dans les échanges du quotidien.

Tout ce qui favorise la cohésion et le caractère "facultatif" des personnes impliquées est bon à prendre.
De plus, avoir une certaine traçabilité permet d'éviter de favoriser les "coups en douce".

## Amélioration continue & Itération

> Arrêtons de croire que le design se fait uniquement avant le lancement ! C'est un travail continu !
> — Sébastien Desbenoit
> (https://twitter.com/desbenoit/status/382132926378676224)

La totalité des problèmes auxquels un projet sera confronté ne peut se deviner

sans tests en conditions réelles : le meilleur cahier des charges ne pourra jamais anticiper la réalité des utilisateurs. Un code parfait (s'il existait) ne le serait que pour le cas défini dans le scénario prévu, qui changera dès la prochaine itération. Mieux vaut donc éviter de faire de l'optimisation en première itération afin de se laisser une marge de progression et d'évolution.

La sortie d'un projet n'est que son début. C'est à ce moment-là que les choses sérieuses commencent, avec l'agrégation des données que vous n'avez pas manqué de mettre en place : Comment se comporte la plateforme ? Le site est-il suffisamment rapide ? Sur quelle(s) page(s) les utilisateurs bloquent-ils ? Quelles sont les exceptions remontées dans les *logs* ?

Autant de données qui permettront d'améliorer la qualité du projet au fur et à mesure, aussi bien d'un point de vue fonctionnel que technique (ces deux aspects allant de pair). De même, l'équipe, avec les différentes sorties itératives du projet, apprend de ses erreurs et fluidifie à chaque étape les processus décisionnels. Ainsi, le projet s'améliore sensiblement à chaque sortie.

## La technique

> Du code parfait qui ne remplit aucun objectif est encore du mauvais code
> — Anthony Ferrara dans « Beyond clean code »

Cette section permettra d'aborder quelques bonnes recettes du quotidien pour des développeurs "fainéants" ou pressés qui veulent être efficaces sans risquer de tout casser tout le temps.

Ces recettes viennent en complément des conseils de gestion de projets et du processus de décision proposés dans les parties précédentes. Coder moins revient à avoir plus de temps pour coder mieux et réfléchir à l'avenir.

Elles ne peuvent fonctionner que si la technique n'est plus considérée comme le dernier maillon de la chaîne qui applique une décision déjà prise.

L'idéal, c'est quand la technique a le temps d'avoir des idées, de les expérimenter et de pouvoir démontrer concrètement de nouvelles pistes. Attention à ne pas aller trop loin pour autant et à garder les pieds sur terre.

## Tout mesurer

> Optimiser est peut-être prématuré, mais mesurer ne l'est pas
> *Optimisation maybe premature but measurement isn't.*
> — Richard Warburto
> (https://twitter.com/RichardWarburto/status/527761292090933248)

Des mesures réelles permettent de ne pas se baser sur des préjugés pour prendre des décisions. Elles permettent de fonctionner par objectifs et d'itérer rapidement pour augmenter les transformations.

Ces mesures peuvent prendre plusieurs formes :

- compteur d'action : quand une action est réalisée, on incrémente un compteur ;
- temps effectué par une action : on place un *timer* avant l'action et un *timer* après, puis on mesure la différence et on l'enregistre ;
- *logs* textuels.

Ces métriques sont en général visualisables sous forme de graphiques à placer sur un tableau de bord visible de tous. Elles peuvent montrer les tendances (on fait deux fois plus de chiffre d'affaires le matin que le soir, mais moins que la semaine dernière), des totaux, etc. Elles sont souvent couplées à des alertes.

Ces mesures peuvent inclure (liste non exhaustive) :

- la transformation des objectifs du projet ;
- l'état des serveurs (espace disque, occupation mémoire, bande passante utilisée) ;
- les journaux (logs) de tous les services qui sont utilisés ;
- le temps minimum / moyen / maximum pour chaque action ;

Grâce à toutes ces mesures mises en place, vous serez en confiance au moment de déployer : si quelque chose ne marche plus, ce sera certainement visible dans les graphes.

Voici quelques exemples de mesures qui ont permi de sauver des situations qui auraient pu traîner bien plus longtemps :

- Symptôme : une forte augmentation de l'utilisation en bande passante de deux serveurs, associée à une métrique (nombre d'éléments maximum dans un panier). Un « pirate en herbe » cherchait en fait les failles du site en testant de nombreux cas à la marge. C'est la combinaison des différents facteurs qui nous a permi de le détecter. Sans cela, cette première tentative d'attaque aurait pu être invisible, jusqu'à ce qu'il réussisse et que son action soit dommageable ;
- Symptôme : explosion du taux d'erreurs de type *timeout*, les *logs* serveurs ne donnant aucun renseignement sur la page concernée. C'est en observant les métriques agrégées que nous avons pu constater que l'action « inscription utilisateur » avait un temps moyen dans la norme mais des temps maximums qui explosaient les plafonds. Nous savions donc où chercher dans le code et il s'est avéré qu'une fonction de cryptographie utilisée était sensible à la charge serveur. Si le serveur était très occupé, alors le temps nécessaire pour exécuter cette fonction était exponentiel, dépassant les 30 s du *timeout*.
- Symptôme : sur une mission où j'ai un jour effectué un audit, l'entreprise comptait jeter le *CMS* qu'ils avaient sélectionné car il était trop lent, et donc tout recoder. La cause se trouvait dans un mauvais algorithme (requête complexe à l'intérieur d'une boucle) appelé sur chaque page. La solution prenait deux minutes à implémenter, encore fallait-il savoir où chercher…

Dans ces exemples, avec pour seules indications « l'application est lente et il y a des erreurs », il est naturel de faire des erreurs de jugement et d'essayer d'optimiser l'existant pour régler les problèmes. Mais sans mesures réelles ce travail ne peut être fait qu'à l'aveuglette.

Prévoir et suivre des métriques réelles pour tous les points d'entrée de l'*API* développée permet d'éviter de réfléchir à des optimisations préventives : vu que tout est mesuré et profilé, on sera en mesure de comprendre ce qui

ralentit l'ensemble et de corriger point par point les aspects les plus critiques. Et, d'expérience, il s'agit rarement des endroits auxquels on s'attend le plus.

Les métriques permettent d'éviter de travailler dans le flou et aident donc à travailler en toute transparence. C'est le premier grand pas vers un meilleur partage de connaissances.

## Partage de connaissances et Paternité partagée

Nous avons vu que personne n'est indispensable et qu'il ne faut pas essayer de le devenir, c'est aussi mauvais pour vous que pour le projet.
Autant partir de ce principe dès le début et planifier en conséquence.

Dans un projet, il est souvent plus confortable de sectoriser les développements par développeurs afin de profiter au maximum de leur connaissance d'un sujet pour coder à un haut niveau de vélocité.
Seulement, un jour, pour une raison ou pour une autre, une nouvelle personne devra intervenir sur leur code. Si ce qu'il doit modifier est inconnu de sa part, vous pouvez être certain que son premier réflexe sera de critiquer et d'envisager à l'avance comment tout refactoriser pour que ce soit codé à sa manière (cf. la partie « se rendre facultatif » de ce même chapitre).

Le principe de partager la connaissance du code, par exemple en effectuant des **revues de code** ou du *pair programming*, c'est de ne pas être seul à écrire son code, et surtout de ne pas être seul à le connaître.
Travailler à deux sur du code lors d'un *pair programming* permet de mieux réfléchir : deux cerveaux valent mieux qu'un. En forçant chacun à exprimer à l'autre son idée d'implémentation, on se rend compte bien avant la première ligne de code que la vision n'est pas forcément la même et le consensus dégage quasi obligatoirement un code de bien meilleure qualité. Et ce, en avance de phase d'une revue de code. Un temps incroyable peut être ainsi gagné, ce qui tend à montrer que le *pair programming* est une excellente pratique. Sans pour autant devoir l'utiliser pour toutes les lignes de code, il est clair que certains points d'architecture, ou n'importe quelle partie complexe, gagnent à avoir deux cerveaux dédiés.

Cette paternité partagée du code permet également de donner une cohérence à l'ensemble du code en homogénéisant sa qualité sur tout le projet.
Autant donc faire tout votre possible pour partager la paternité de votre

travail et ainsi être serein quand vous vous absentez.

La revue de code systématique est – pour moi – une évidence. Les avantages sont tellement nombreux qu'il n'y a finalement que peu de raisons de s'en passer.
Jugez donc un peu :

- elle force à fonctionner par "branche de fonctionnalité" (*feature branch*), ce qui permet de séparer les développements par ticket / fonctionnalité / *bug* ;
- visualiser la somme de ces changements dans une interface adaptée (un outil de *diff*) permet une détection plus aisée des erreurs que dans son éditeur de code habituel ;
- dans un *workflow* où la revue de code est systématique, la responsabilité des erreurs est au moins autant sur les épaules des *reviewers* (les "critiques") que celles du du développeur. Cette situation force à produire et n'accepter que du bon code ou, à défaut, à s'assurer que les raccourcis effectués soient assumés par un maximum de personnes. Il n'est pas rare de voir des commentaires de revue de code du type : « tu as changé l'en-tête de cette fonction mais tu n'as pas documenté à tel endroit », « attention, tu as oublié tel cas à tel endroit ».

Attention, la revue de code n'est pas un remède à tous les maux, mais on évite ainsi de nombreuses situations délicates comme découvrir le code du collègue quand celui-ci est absent et qu'il ne peut donc pas nous aider dans sa compréhension. Mieux vaut parler pendant dix minutes d'un bout de code tendancieux une heure après l'avoir écrit que plusieurs semaines ou plusieurs mois après.

## La théorie de la fenêtre brisée

> […] est une explication statistique mise en avant pour établir un lien direct de cause à effet entre le taux de criminalité et le nombre croissant de fenêtres brisées à la suite d'une seule fenêtre brisée que l'on omet de réparer.
> Ce principe est fondé sur l'exemple d'un édifice dont une vitre brisée n'est pas immédiatement remplacée. Partant, toutes les autres seront cassées peu de temps après parce que la première laisse entendre que

> le bâtiment est abandonné, ce qui constitue l'amorce d'un **cercle vicieux**.
> — Wikipédia « Hypothèse de la vitre brisée » (https://fr.wikipedia.org/wiki/Hypoth%C3%A8se_de_la_vitre_bris%C3%A9e)

Cette théorie indique que si l'on permet à des intervenants de procéder à de mauvaises pratiques sur le projet une fois, les chances augmentent considérablement pour que cela devienne la nouvelle norme.

Ainsi, si on veut éviter que les intervenants n'aient le sentiment de travailler en toute impunité, je vous propose de mettre en place la situation inverse :

- s'affranchir de la peur de l'échec ;
- travailler dans la transparence ;
- mutualiser la responsabilité, notamment grâce les revues de code.

Le point essentiel est donc de savoir gérer l'échec dans un projet. Ne pas le préparer revient à le garantir, car – Loi de Murphy aidant – vous pouvez être certain que le pire scénario finira par se produire. Évitez de ne compter que sur votre bonne étoile.
Ainsi, essayez d'envisager le pire qui puisse arriver au projet : code de *debug* en production, base de données corrompue et inutilisable, serveur hacké, *data center* en fumée, etc. et planifiez une solution pour cela. Forcez-vous à trouver une solution à tout problème vraiment critique avant qu'il se produise, sans pour autant la développer. Cela vous permettra d'être beaucoup plus serein le jour où ledit problème se présentera : vous serez déjà entraîné pour l'occasion (vous vous souvenez de cette histoire de pompier dans l'introduction ?).

Cet affranchissement de l'échec est aussi valable à titre individuel. Si vous avez peur de l'échec, faites en sorte 1/ qu'il vous arrive le plus tôt possible et 2/ que vous soyez en mesure de réagir.

Cela commence par tester ce que l'on fait.

## Tester

Le moyen le plus connu pour mesurer la qualité du code est de pouvoir

s'assurer de son comportement. En ayant du code testé, on est en mesure de savoir si *a minima* le code fonctionne comme désiré.

Il existe de nombreux types de tests (liste non exhaustive) :

- les tests unitaires : vérifier qu'un composant est fonctionnel, individuellement ;
- les tests d'intégration : valider l'intégration des différent modules entre eux, dans un environnement précis, souvent celui de production ;
- les tests fonctionnels : valider la conformité des fonctionnalités en rapport avec ce qui est demandé ;
- les tests de non-régression : valider que l'application ne se dégrade pas. Consiste le plus souvent à exécuter tous les autres types de tests et valider que l'application continue de fonctionner ;
- les tests de vulnérabilité : vérifier la sécurité ;
- les tests de performance : valider que les performances annoncées dans la spécification sont bien atteintes ;
- la couverture de code, qui n'est pas un test en soi mais une métrique sur le pourcentage de code testé.

Un des arguments les plus souvent utilisés contre l'emploi des tests est le temps nécessaire pour les coder.
Cependant, vous pouvez être sûr que les incertitudes sur le bon fonctionnement d'un code non testé sont la garantie d'avoir des problèmes plus tard.

Comme il y a de toute manière des *bugs* dans un projet, ils seront bien plus facilement trouvés si le code est déjà testé sur de nombreux scénarios : il suffit d'en ajouter un nouveau qui prouve l'erreur (donc, qui ne passe pas), corriger, relancer tous les tests. Le nouveau test doit maintenant passer et on doit s'assurer que les anciens ne cassent pas (non-régression). En ajoutant ainsi des tests au fur et à mesure de la rencontre de problèmes, on s'assure que la qualité du code s'améliore, tout en évitant les régressions.

Une des meilleures pratiques mise en avant et pourtant rarement appliquée est le *TDD* (*Test Driven Development*, le "Développement Dirigé par les Tests"). Pour une fonctionnalité donnée, on écrit les tests avant d'écrire le code, ce qui permet de tester les scénarios validant le bon fonctionnement du code.

Écrire les tests en amont permet de réfléchir et de valider l'*API* publique de cette fonctionnalité et ainsi de se concentrer sur le design public du code avant son implémentation. Cette pratique a l'immense avantage de valider ou d'invalider certaines décisions d'implémentation car les tests sont les premiers utilisateurs du code à écrire.

> Prenons un exemple : j'ai récemment eu à tester un scénario qui impliquait le dépôt d'un *cookie* après une certaine requête. Le nombre de scénarios étant phénoménal (connecté / déconnecté, administrateur ou pas, *cookie* déjà présent, etc.), les reproduire à chaque itération de code était tellement pénible qu'on ne recettait plus tous les cas : on se contentait de tester le précédent qui avait échoué.
> Il aurai donc mieux valu avant tout écrire une suite de tests comprenant les scénarios que l'on allait rencontrer. Exécuter cette suite de tests pendant le développement aurai ainsi ensuite permis de vérifier la validité du code.

Cependant, la dette technique est liée au code qui n'est pas écrit, et les tests sont eux-mêmes du code, qui peut contenir des erreurs.
Ainsi je vous recommande d'être pragmatique sur l'écriture et l'ajout de tests.

> Exemple : en fonctionnant uniquement par tests unitaires pour tester une méthode dans 100% des cas, on en arrive à tester des situations irréalisables. Ainsi, pour faire passer les tests à la méthode en question, on ajoute du code contre ces situations, ce qui ajoute de la complexité au programme et augmente le nombre de scénarios à maintenir en cas de refactorisation.

Au moment où l'on décide de prendre des raccourcis dans l'implémentation pour pouvoir livrer plus vite un projet, les tests sont souvent les premiers à être supprimés. C'est dommage, mais c'est pragmatique. Ce que je recommande c'est d'ajouter au moins un test fonctionnel qui assurera la validation du bon déroulement du scénario le plus usuel et la transformation des objectifs.

*A minima*, même si le code n'est pas testé, il est primordial que les développeurs sachent écrire des tests et réfléchissent, au moment d'écrire du code, à comment le rendre testable. Il faut s'imaginer que la personne qui viendra modifier le code pour corriger un *bug* est très très violente et connaît l'adresse de tous les développeurs impliqués…
Même sans tests, le code doit donc être pensé pour être testable !

Fin 2014, les outils à disposition pour tester en amont votre travail sont légion :

- vous êtes en mesure de tester votre intégration avec tous les navigateurs possibles avec des outils comme Browserstack de Microsoft ;
- vous pouvez comprendre en détails le fonctionnement du rendu de votre site dans un navigateur *via* WebPageTest ;
- vous avez la possibilité de tester votre code sur une multitude de configurations, automatiquement, *via* des services comme JoliCi ou Travis-CI, dont le slogan est « Build Apps with Confidence » (« Construisez votre application en toute confiance »). Ça ne s'invente pas, nous parlions tout à l'heure de la peur de l'échec…

Ces outils vous permettent de vous prémunir d'avoir ne serait-ce qu'à tester manuellement ce que vous faites.
Mettez rapidement des outils en place, et ils vous diront si ce que vous codez est correct.

## Automatiser

Nous venons d'évoquer quelques outils qui permettent de faire des tests, mais il est possible d'aller beaucoup plus loin, tout en se préservant de nombreuses erreurs humaines.

Il n'y a rien de plus pénible que de faire trois fois la même chose.
Tout comme tester notre scénario de dépôt de *cookie* est pénible et source d'erreur à force de répétition, déployer un site en test / pré-production / production / redescente de la production vers le développement… est pénible et chronophage.
Tellement chronophage que c'est la garantie que cela ne sera pas fait et que toutes les bonnes pratiques ont peu de chance d'être employées car

finalement trop pénibles… si on ne les automatise pas.

Pour la santé mentale de l'équipe, et si on veut éviter le phénomène de la fenêtre brisée vu précédemment, il ne faut fournir aucune excuse aux intervenants pour avoir mal agi mais au contraire fournir tous les outils et les méthodologies pour qu'il soit plus simple et plaisant de bien faire.

Par exemple, le scénario phare d'une recette est de pouvoir tester une nouvelle fonctionnalité ou une correction de *bug* avec l'état de la production… sans courir le risque de la "casser" encore plus.
Pouvoir déployer, en un clic ou une ligne de commande, une nouvelle version du code sur une instance de la production dupliquée en direct permet de tester en situation quasi réelle sans risque d'effet(s) de bord.
Ne pas avoir cette possibilité reviendrait – finalement — à ne pas valider quoi que ce soit et à déployer n'importe quoi (et n'importe comment).

Cette pratique s'appelle l'intégration continue, et il est vraiment conseillé de la mettre en place.

## Architecture

Un client demande majoritairement un bon produit (soit généralement un retour sur investissement), et non un bon code. Il est difficile, pour un technicien, de s'adapter aux contraintes annoncées d'un projet et d'éviter le syndrome de l'ingénieur : face à un problème, il ne faut pas essayer de développer SA solution parfaite mais plutôt challenger la demande pour trouver la solution la plus simple possible.

Une demande de développement d'un site peut être résolue par un site en *SaaS* pour 3 € par mois, ou par un site construit à l'aide d'une brique logicielle spécifique pour plusieurs dizaines de milliers d'euros, ou encore par un développement à partir de zéro pour un montant encore plus élevé.

L'architecture d'un site est à adapter en fonction de la demande, mais il semble logique de connaître la spécialité d'un prestataire avant de s'adresser à lui, car chacun aura toujours sa préférence et ses propres intérêts.

De manière générale, mieux vaut viser la simplicité, la robustesse et l'agilité avec des composants réutilisables et interchangeables basés sur des contrats d'interface (de communication, pas visuelle). Ceci permet de se concentrer sur chaque partie de manière indépendante.

De plus, ce conseil s'applique aussi bien au niveau *macro* que *micro* :

- il sera parfois plus simple de séparer plusieurs sites confectionnés de manières complètement différentes, mais avec le même rendu pour donner l'illusion qu'il s'agit de la même plateforme, alors que dans d'autres circonstances il sera préférable de réaliser un site unique qui fait tout ;
- de même, pour un seul site il est souvent possible de découper certaines parties en composants, plus faciles à maintenir un par un, mais dont l'orchestration est plus ardue.

Cela ne devient possible et compréhensible que quand il existe une documentation et qu'elle est maintenue.

## Documenter

La documentation, c'est la hantise de 90 % des intervenants d'un projet. Pourtant, nous en avons tous besoin à un moment ou à un autre.

De la même manière que le *TDD* permet d'écrire le code utilisant la fonctionnalité avant que celle-ci ne soit développée — et donc de se prémunir d'un grand nombre de problèmes — la documentation préventive est excellente pour votre projet.

Ainsi, documenter comment installer votre projet vous permet de vous rendre compte si la procédure est devenue trop complexe et est donc un frein à l'entrée de nouvelles ressources sur le projet.
De même, documenter le fonctionnement d'un composant vous permet de réaliser qu'il est peut-être nécessaire de vous y intéresser de nouveau — et plus en profondeur — car sa mécanique n'est pas aussi intuitive que vous le croyez.

Attention, il existe plusieurs formes de documentation :

- le résumé très court détaillant pourquoi ce projet voit le jour, quels sont ses enjeux… ;
- la documentation fonctionnelle, qui regroupe l'ensemble des spécifications des fonctionnalités ;
- le *style guide*, qui permet de visualiser la charte graphique, prête à être utilisée ;

- la documentation des accès (comptes utilisateur/modérateur/administrateur de test, etc.), sans laquelle chacun recrée des accès à tout va, générant des dangers pour la sécurité ;
- la documentation d'installation ;
- la documentation, dans le code, du code ;
- etc.

Vous aurez besoin à un moment ou à un autre de ces informations, autant les rendre accessibles à tous les intervenants.

## Attention au zèle et à la surqualité !

L'objectif de tout développement est de produire quelque chose. Vouloir en faire trop sur la qualité revient exactement à la même chose que voulir développer trop de fonctionnalités.
Imaginez alors si on mélange un besoin de trop de fonctionnalités avec des développeurs qui mettent trop la qualité en avant : c'est la garantie absolue de figer votre projet à jamais et dès sa sortie… si tant est qu'il arrive à voir le jour.

Ce n'est pas parce qu'il y a une solution technique complexe *intéressante* à mettre en œuvre qu'il faut l'implémenter. Souvent, le meilleur choix consiste à trouver une solution alternative qui prendra moins de temps à développer.

> Avoir la foi, c'est monter la première marche même quand on ne voit pas la fin de l'escalier — Martin Luther King

C'est ce que m'a dit un jour un chef d'entreprise à propos d'un choix stratégique. En effet, lorsqu'on ne voit pas la fin d'un escalier, on peut avoir tendance à en surévaluer la difficulté et à se préparer au pire, tel un sportif s'entraînant pour les JO…
Mais bien souvent, l'escalier ne fait finalement que quelques marches, et soit vous êtes déjà presque épuisé avant de les gravir, soit vous n'avez jamais osé les monter par peur de vous retrouver face à une situation non maîtrisable.

Ainsi, beaucoup de projets ne voient pas le jour car « on ne peut pas se permettre de… ».

- Parce que l'outil de modération n'est pas suffisant… pour un site qui n'a pas encore de contenu ;
- Parce qu'il faudrait « migrer le CMS du site de la version x à la version y »… alors que le problème réel n'est en rien dû à un quelconque numéro de version mais plutôt au choix du CMS lui-même ;
- Parce que le code n'est pas d'une qualité exemplaire… et pourtant « du code parfait qui ne remplit aucun objectif est encore du mauvais code ».

Des millions d'exemples qui font perdre un temps infini aux porteurs de projets tout en sollicitant des ressources sur des sujets peu intéressants et souvent chronophages.

> Le mieux est l'ennemi du bien
> — Voltaire, La Bégueule, 1772

Il est donc important de comprendre les règles pour pouvoir choisir de les enfreindre si nécessaire : agilité, qualité, recommandations… ont des impacts réels dans vos projets.
Rappelez-vous qu'il n'est pas toujours nécessaire d'être extrémiste en aspirant à respecter les règles à 100 %.
Les connaître est souvent suffisant, et documenter le pourquoi de l'exception l'est aussi.

## Dans la durée

> Pour augmenter la vitesse et la capacité de développement, investissez davantage dans la qualité, et non pas moins
> *To increase development velocity and capacity, invest more in quality, not less*
> — Matt McClure
> (https://twitter.com/matthewlmcclure/status/381183618041004032)

Nous l'avons vu, les problématiques de dette technique sont majoritairement liées à des problèmes humains et donc de confiance.
Or la confiance est difficilement extensible et c'est pour cette raison que les projets sont encadrés par une pléthore de documents, papiers, contrats, qui — tout en pensant nous protéger du pire — en sont quasi systématiquement

la cause.
L'enjeu majeur d'une relation client / prestataire est donc la mise en place d'une confiance mutuelle qui permet de faire comprendre à l'autre que le but n'est pas de l'arnaquer, mais bien de construire un projet tout en gagnant sa vie.

Dans une logique de confiance, il faut que chacun comprenne les enjeux et les difficultés de l'autre.
Parfois il est nécessaire de prendre des raccourcis pour que le projet sorte plus vite. D'autres fois, il faut revenir sur ceux-ci pour améliorer la qualité du projet, quitte à en reprendre de nouveaux une fois les précédents raccourcis remboursés.

Il faut donc racheter cette dette technique de temps en temps et cela passe par une connaissance de celle-ci. Car plus la dette technique est cachée, plus elle sera vicieuse.

Il faut donc communiquer dessus et qu'il y ait une véritable volonté d'écoute des deux côtés afin de trouver un consensus qui permette de faire évoluer le produit aussi bien techniquement que fonctionnellement.

## L'avenir ?

Toutes les bonnes pratiques de gestion de projet que nous venons d'étudier s'appliquent sur des problématiques d'hier, d'aujourd'hui et de demain. Mais que peut-on dire des projets à beaucoup plus longue échéance ?

Bien entendu, dans un futur à moyen terme, le rapport aux technologies aura évolué, et toutes ces problématiques seront vraisemblablement de l'histoire ancienne.
Toute personne en activité depuis au moins cinq ans aura reçu un cursus technique pendant ses études. Et quand je parle de cursus **technique**, je pense à adopter une attitude de « résolution de problème » (https://fr.wikipedia.org/wiki/R%C3%A9solution_de_probl%C3%A8me) si chère à tout quotidien d'un technicien.
Le web décentralisé sera généralisé, avec une prise de conscience de la propriété intellectuelle, du contrôle du partage et de l'interopérabilité.
Les *API*s seront normalisées autour de protocoles ouverts et décentralisés et de composants réutilisables.

Il sera aussi possible de développer et de "bidouiller" à l'aide d'outils de type NoFlo. Nous utiliserons tous des ordinateurs quantiques, capables de prouesses impossibles à imaginer à ce jour.

… Cependant, personne ne peut prédire précisément l'avenir et nous sommes donc bien obligés de faire des choix — réfléchis mais dans la limite des connaissances accessibles au moment donné.
Sans connaître les évolutions techniques à venir, la seule chose qu'il est possible de faire c'est donc d'être en capacité de se former, d'évoluer et de s'adapter rapidement tout en maîtrisant sa dette technique.
Ceci marche aussi bien pour le code que pour le projet lui-même : évitez de tout figer dans le marbre (e.g., dans un contrat) et laissez-vous une marge de sécurité pour la réaction et l'expérimentation.

Et pour finir, inévitablement mais le plus tard possible, vous découvrirez que — malgré toutes les précautions prises — *quelque chose cloche* : il sera alors temps de passer en mode **résolution**, ce qui est l'objet du chapitre suivant.

# Résoudre

Vous êtes maintenant en mesure d'identifier et d'essayer de prévenir la dette technique dans vos projets. Mais que faire quand celle-ci est déjà en place ? Quel « chemin de migration » est possible pour vous aider à la rembourser et ainsi repartir sur de bonnes bases ?

> Il vaut mieux prévenir que guérir.
> — Dicton populaire

Ce chapitre est sûrement celui que les victimes de la dette technique au quotidien auront envie de consulter directement.
Si c'est votre cas et que vous arrivez sur ce chapitre sans avoir lu les deux précédents (« Identifier » et « Prévenir »), je vous souhaite la bienvenue et vous encourage fortement à les lire dès maintenant.
En effet, ce livre démontre qu'encourager la prise de conscience des contraintes de chacun tend à réduire la dette technique.
Ainsi, si vous êtes dans une situation critique de dette technique, commencez par établir un *post-mortem* et formez votre équipe aux principes abordés dans ce livre. Que vous sachiez *a minima* ce qu'il faudrait changer si ce projet était à refaire.

Ce chapitre est celui qui correspond le plus à mon quotidien professionnel, les entreprises étant malheureusement bien plus disposées à investir quand elles sont au pied du mur qu'en prévention.
Le renversement de cette situation est d'ailleurs un objectif fort de l'écriture de ce livre.

Ce chapitre sera donc le plus court, beaucoup de choses ayant déjà été dites plus haut en termes de prévention et d'éducation.

## Évaluer

Nous voici de retour au chapitre « Identifier » : vous avez pris conscience que vous êtes en situation de dette technique : bravo (et… désolé). Il est maintenant temps d'aller encore plus en profondeur et de séparer le bon grain de l'ivraie.

## Le regard extérieur

Je me rends compte qu'il y a toujours un élément déclencheur pour que, face à une situation de dette technique certaine, un porteur de projet ou un prestataire en vienne à faire appel à une aide extérieure.
Au-delà de la question des compétences, il y a avant tout un problème de confiance : le porteur de projet souhaite se sortir d'une situation de prise d'otage où il a l'impression qu'il n'est plus en capacité de faire avancer les choses et qu'il a besoin qu'un électrochoc soit administré aux forces en présence.

Faire appel à un regard extérieur a plusieurs avantages.
Tout d'abord, et le plus souvent, il se moque de l'historique : "Institutions will try to preserve the problem to which they are the solution" (Clay Shirky).
Contrairement aux personnes habituées à un système, le regard extérieur n'en connaît ni les avantages ni les inconvénients et n'a pas appris à éviter instinctivement les bugs connus.
Il est aussi en mesure de détecter les failles de conception d'un système parce qu'il a cette capacité à critiquer sans vouloir excuser.
De plus, le regard extérieur n'a pas de position dans la hiérarchie réelle, il est donc plus libre pour énoncer des vérités pas forcément agréables à entendre.
Il lui est également plus facile de détecter les erreurs de conceptions triviales.
Enfin, il bénéficie d'autres expériences qui sont toutes aussi enrichissantes, qu'elles soient similaires ou *a contrario* très différentes.

Si vous n'êtes pas en mesure de faire appel à quelqu'un de véritablement extérieur, essayez au moins d'en adopter l'attitude. Oubliez que vous êtes intervenu sur le projet et offrez-vous un regard neuf sur le travail réalisé.
Attention, ce n'est pas chose aisée : rien ne remplacera un regard extérieur et impartial.

## Numéroter ses abattis

Tout commence par un audit de l'existant.
Comme il s'agit du chapitre « Résoudre la dette technique », le résultat risque de ne pas être joli à voir et il a fort à parier que ce ne sera donc pas très plaisant à vivre. Mais cela contribuera à atteindre un objectif essentiel : la prise de conscience des intervenants du projet.

Si celle-ci n'est pas encore partagée, alors **communiquez mieux**.

Dans le cas contraire, quand le projet est proche du gouffre, c'est le moment de se poser et de réfléchir.

Classez toutes les erreurs de conception par ordre de priorité, autrement dit "qu'est-ce qui dérange le plus" : logique mélangée avec la vue ? code procédural ? trop de copié-collé ? HTML et CSS sans homogénéité ni structure ? etc.
De même, séparez ce qui fonctionne de ce qui ne fonctionne pas.

## Collecter

Le système est opérationnel et on ne sait pas pourquoi ?
Parfait, il est temps de tout documenter. Voici une liste non-exhaustive des questions à se poser :

- Qui a les accès au nom de domaine et au certificat SSL ? Qui est chargé de les renouveler ?
- Quelle est la configuration DNS ?
- Quels serveurs sont utilisés ? Où sont-ils hébergés ? Qui paye ? Quel OS ? Quelle version du système et des composants ? Sont-ils à jour (on parie que non ?) ?
- Qu'y a-t-il d'installé dessus ? Sont-ce les configurations par défaut ?
- Qui a les accès à ces fameux serveurs ? Et aux autres ?
- Quelle est la stratégie de déploiement ? S'il n'y en a pas, est-il possible d'en créer une (accès SSH, *shell* restreint, …) ?
- Que disent les outils techniques d'audit de qualité de code (complexité du code, *code smells*, couplage des éléments, cohérence du style…) ?
- etc.

## Mesurer

Ce qu'il vous faut, c'est comprendre l'activité. Le fonctionnement normal ne doit plus avoir le moindre secret pour vous, ne serait-ce que pour vérifier qu'aucune de vos modifications n'aura d'impact négatif sur les métriques vitales.

Mesurer est vraiment excellent pour le moral et la cohésion d'une équipe,

c'est LE moyen de faire en sorte que des objectifs soient partagés, tout en s'assurant que l'on ne détruit rien.

## Moyens d'action

Cet audit permettra d'évaluer les solutions envisageables pour assurer l'avenir du projet, solutions qui peuvent varier en fonction de son état d'avancement :

- le projet est en ligne et "fonctionnel" mais il est nécessaire de le redynamiser et de le faire évoluer ;
- le projet est livré mais impossible à mettre en ligne car non satisfaisant d'un point de vue technique et/ou fonctionnel ;
- le projet est en cours de construction mais il est clair qu'il court à la catastrophe…

D'autre part, la "moins mauvaise des solutions" dépendra également de la motivation des équipes et du budget restant.
Retenez qu'il y a rarement des solutions parfaites car elles ont toutes un coût.

Voici plusieurs possibilités :

- tout jeter et recommencer à zéro ;
- améliorer l'existant ;
- construire une nouvelle architecture à côté de l'existant et mettre en place des ponts entre les deux pour pouvoir les faire évoluer en parallèle. Et ainsi pouvoir simultanément développer du code propre tout en améliorant l'existant.

Mais avant de voir en détail l'implémentation de ces différentes solutions, sortons "la ceinture et les bretelles" et protégeons-nous des risques de la refonte.

### Processus de décision

Première étape : comprenez les forces en présence. Comment sont prises les décisions ? Par qui ?

Deuxième étape, intervenez :

- Limitez les couches hiérarchiques ;
- Réduisez tous les temps de prise de décisions, pour une itération très rapide. C'est le moment de prendre toutes les décisions « choc » : le projet étant en perdition tout le monde sera content qu'il s'achève rapidement, quitte à ce que certains points ne soient pas traités. Vive la relativité !

## Tailler dans le vif

N'hésitez pas à supprimer les fonctionnalités non abouties. C'est le moment !

Il faut assurer le minimum fonctionnel vital.
Si nécessaire, coupez toutes les branches mortes : vous êtes passé en service minimal, éliminez tout ce qui est inutile (quitte à le réactiver plus tard).

Ce n'est pas parce qu'une demande est sympa ou intéressante qu'elle doit être développée : **maîtriser un périmètre fonctionnel est primordial**.

## La ceinture et les bretelles

Si le site fonctionne avec des données réelles, il va falloir éviter de détruire le peu qui fonctionne. Mieux vaut donc s'assurer qu'aucune de vos modifications sur l'architecture ne casse plus qu'elle ne répare.

### Ne pas réinventer la roue

Vous êtes "en guerre". Il ne vous coûtera pas bien cher de prendre de multiples services *SaaS* (monitoring, sauvegarde, déploiement) pour une durée d'un mois (souvent, le premier est même gratuit) plutôt que d'installer votre propre infrastructure (que vous devrez ensuite gérer).
Libre à vous de les adopter définitivement ou non par la suite, mais en attendant ils vous apporteront plus de sérénité que vous ne l'imaginez. Vous n'êtes pas là pour passer 4 jours à mettre en place l'infrastructure de toutes ces briques.

### Créer des sauvegardes de tout ce qui tourne

Sauvegardez tout : la configuration, les données, le code, **tout**.

Mettez en place des sauvegardes automatiques, toutes les 10 minutes s'il le faut. L'objectif est d'assurer l'intégrité des données.
Prévoyez également la procédure permettant de restaurer une de ces sauvegardes en production. En effet, les sauvegardes sont indispensables mais sont encore plus utiles quand elles sont elles-mêmes testées et que l'on est sûr de pouvoir les utiliser à 100 %.
Rien de pire, au moment critique de réparer un crash à l'aide d'une sauvegarde, que de se rendre compte que celle-ci n'est pas complète.

Des scénarios catastrophe arrivent tous les jours :

- le traditionnel exemple de l'action effectuée en production alors qu'on pensait être en développement ;
- un script de mise à jour qui se passe mal et s'interrompt en plein milieu ;
- la coupure de courant impromptue ;
- etc.

## Cloisonner, cloisonner, cloisonner

Ce qui existe ne doit pas interférer avec ce que vous allez construire et (si possible) réciproquement.
Essayez, dans la mesure du possible, de dupliquer l'infrastructure ailleurs afin de ne pas toucher à l'existant. Sur un projet en danger, on n'est jamais trop prudent, et un expert judiciaire en informatique vous dira toujours de ne pas travailler sur les disques durs d'origine, mais plutôt sur des copies parfaites, afin de pouvoir tester et retester tout ce que vous aurez envie de faire.

Par exemple : si vous devez mettre à jour un composant, mieux vaut le tester dans un premier temps sur un serveur de configuration similaire avant de le tester sur le serveur réel où vous n'aurez pas le droit à l'erreur.

## Versionner

Placez l'intégralité des fichiers du projet dans un gestionnaire de versions, quitte à ce qu'il soit distinct dans un premier temps, avec des synchronisations manuelles.

### Ajouter des tests

Au même titre que les mesures sont l'assurance du développeur une fois le projet en production, les tests sont son assurance avant la mise en production.

Pour pallier l'urgence, commencez au moins par mettre en place des tests fonctionnels afin de vérifier les scénarios les plus importants !

Consultez le chapitre « Prévenir » pour plus d'informations sur la mise en place de tests de manière pragmatique.

## Repartir de zéro ?

La solution la plus risquée que vous pourrez choisir sera de vouloir repartir sur une réécriture de zéro d'un existant au complet. Cette non-solution, bien que souvent tentante, est le meilleur moyen de replonger.

Tout réécrire de zéro est souvent plaisant… au départ : une nouvelle architecture, c'est "sympa". La comparer à la précédente est motivant, drôle, excitant.
Puis on se rend compte que tous les développeurs qui ont travaillé sur ce projet depuis tant d'années ont peut-être écrit beaucoup de mauvais code, mais ont aussi développé de nombreuses fonctionnalités en répondant à des scénarios pas forcément documentés au moment de la refonte.
Réécrire l'ensemble des fonctionnalités de la précédente version équivaut donc à la traversée du désert : le fun a disparu, et on se retrouve dans la même situation que les précédents développeurs : vite, il faut en finir, on bâcle et on n'en parle plus.
Vous revoilà en flagrant délit de dette technique, "*insert code and try again*" (NdR: "même joueur, code encore").

## Itérer !

Comme vous éviterez, dans la mesure du possible, de réécrire une nouvelle version complète, tâchez de **construire un pont pour pouvoir bâtir à côté**.

Le processus sera donc itératif : plutôt que de tout refaire d'un coup, nous allons procéder par itération, brique par brique.

1. Commencez par identifier une (minuscule) brique à nettoyer ;

2. Trouvez par où commencer, soit son point d'entrée ;
3. Identifiez le « contrat » du point d'entrée de cette brique, et assurez-vous qu'il est bien compris :
    a. Identifiez toutes les dépendances externes, créez des ponts ;
    b. Identifiez toutes les dépendances internes, créez des ponts ;
    c. Codez des tests du service fourni par cette brique ;
    d. Ajoutez des métriques sur le fonctionnel afin d'améliorer la sécurité ;
4. Une fois que vous êtes bien serein sur le contrat de cette brique, recodez-la en vous assurant *via* les tests que vous ne détruisez rien ;
5. Refactorisez l'ensemble pour obtenir du code propre ;
6. Recommencez à l'étape 1 avec une autre brique. Eh oui, c'est ça l'itération ;)

Dans un projet, on se complique souvent la vie car les composants dont on a besoin à l'instant présent ne sont pas encore disponibles, ou pas dans une version stable. On utilise alors de vieilles versions, des composants moins intéressants, etc.
C'est le moment de questionner ces choix et de se rendre compte qu'ils ne sont plus d'actualité. C'est d'ailleurs pour cette raison qu'il est intéressant de documenter ces choix d'architecture dans le projet, afin de pouvoir revenir dessus plus facilement.

Exemple : il y a quelques années, j'utilisais sur un projet un merveilleux module nommé *Search API* qui me permettait de me baser sur des *indexes* de recherche pour pas mal de choses. Une *API* vraiment super. Ce module permet d'avoir plusieurs moteurs, comme *Elasticsearch*, *Solr* ou base de données. À l'époque, *Elasticsearch* était trop jeune et le module base de données ne gérait pas les recherches avec une condition « OU », qui était primordiale pour ce projet. Nous étions donc partis sur *Solr*.

Quelques mois plus tard, l'hébergeur revint vers nous pour râler à propos de l'installation de *Solr* qui ne convenait plus, qu'il fallait changer de serveur, etc. Une manipulation à ne pas prendre à la légère.

Pragmatique dans l'âme, j'ai alors cherché alors un moyen de contourner le problème. Apparemment, l'intégration base de données de *Search API* permettait dorénavant de faire des requêtes « OU », ce qu'il n'était pas capable de faire précédemment.
Mise à jour du module, suppression de *Solr*, tout fut réglé en 15 minutes.

Là encore, la méthode des « cinq pourquoi » a fonctionné (il n'est pas toujours obligatoire d'en arriver à cinq) :

- **Pourquoi faut-il déplacer le site ?** : Solr dérange sur le serveur ;
- **Pourquoi utilise-t-on *Solr* ?** : il est le seul à nous permettre de faire une requête « OU » ;
- **Pourquoi a-t-on besoin de cette requête « OU » ?** : elle est indispensable à telle fonctionnalité ;
- **Pourquoi ne pas utiliser le module de bases de données maintenant qu'il gère la requête « OU » ?** : parce qu'on ne savait pas qu'une nouvelle version la gérait ;
- **Pourquoi ne pas le mettre à jour ?** : Faisons-le !

Problème résolu : on utilise une nouvelle version du module base de données, et on supprime *Solr*.

Si j'avais essayé de répondre à la demande initiale, cela aurait coûté beaucoup

plus cher et fait perdre du temps à tout le monde : personne n'aurait été satisfait.

Ainsi, préférez commencer petit, puis itérez. De l'extérieur, vous verrez un grand nombre de choses sur lesquelles vous vous poserez des questions et que vous serez tenté de changer.
Avant de démonter quoi que ce soit, fixez votre objectif et mettez-vous au travail. En d'autres termes : restez posé, écoutez ce qui se passe et fondez vos perspectives sur de petites victoires réalisables.

Pour y parvenir, il est nécessaire de mettre en place une véritable stratégie de déploiement continu : vous ne pouvez pas passer à la suite si vous n'êtes pas en mesure de déployer sereinement vos nouvelles modifications.
Les meilleures pratiques sont en accord avec tous les conseils déjà énoncés dans ce livre :

- déployez vite vos nouvelles modifications : n'attendez pas d'en avoir un certain nombre avant de déployer, sinon vous prenez le risque de multiplier les sources d'erreur ;
- agissez en toute transparence, en communiquant les nouveautés, correction de bugs, etc. ;
- complétez la documentation, par incrément ;
- validez, *via* les métriques que vous aurez mis en place, que vous ne cassez rien et que les modifications sont réellement déployées.
- si quelque chose ne fonctionne pas ou plus, apprenez de vos erreurs : réparez-les et faites en sorte qu'il y en ait moins plus tard et qu'elles soient encore moins dommageables.

Tous ces mécanismes en place, la dette technique diminuera d'autant que la qualité globale augmentera.
Il deviendra ainsi plus agréable de travailler sur l'existant et vous prendrez plaisir à fournir un service qui s'améliorera et à en parler.

# En conclusion

La dette technique est une notion impérative à connaître pour quiconque participe, de près ou de loin, à un projet Web.

Sans pour autant la craindre, il est nécessaire de comprendre que tout choix a des conséquences et qu'un projet où la qualité est peu prise en compte n'aura pour durée de vie que celle d'une preuve de concept : suffisant pour tester quelque chose mais dangereux à mettre en production.

Le cycle de vie d'un projet doit donc prendre en compte cette dette technique et il est vital de se poser les bonnes questions sur la volonté de le rendre pérenne. Des raccourcis sont parfois bons à prendre pour limiter les coûts et améliorer les délais : on contracte alors une dette. Mais ils doivent être effectués pour créer de la valeur, en comprenant que pour créer encore plus de valeur il faudra impérativement, à un moment ou à un autre, initier un nettoyage (une refactorisation) de ces raccourcis et ainsi rembourser sa dette.

Ce procédé doit être géré par une entente entre porteur de projet et équipe technique, sans se voiler la face, en confiance et en toute honnêteté.

Pour garantir la prise en compte de ces problématiques, il faut les énoncer dès la contractualisation du projet : gardez toujours en tête qu'un projet web ne se conçoit pas comme un produit à la chaîne.
Si un porteur de projet pousse à rogner sur la qualité, alors il ne faut pas s'attendre à des miracles… Un prêté pour un rendu, en somme.

S'il y a un concept à retenir de ce livre, c'est que la dette technique et ses risques sont en définitive peu liés au code mais plutôt à tout ce qui gravite autour. Pour s'en prémunir ou la réparer, il n'y a en fait pas vraiment d'ingrédients secrets. À vous d'établir la recette magique avec la bonne volonté des intervenants, pas trop d'ego, une bonne dose de motivation et **beaucoup** de pragmatisme.

Et enfin, surtout, communiquez !
Favorisez la communication, ne blâmez pas ceux qui ne savent pas… bref, gardez un esprit d'équipe sain. Celui-ci ne doit pas se limiter à l'équipe technique mais doit englober toutes les personnes liées au projet : on construit ce projet ensemble, et non les uns contre les autres.

Ces composantes arrivent rarement par magie. À vous de vous inspirer des bonnes pratiques de ce livre et de les appliquer à votre contexte et à votre niveau. Plus nous serons nombreux à être acquis à ces principes, mieux la dette technique sera maîtrisée.

# Bibliographie

## Sources


- *Working Effectively with Legacy Code*, Michael Feathers (http://www.amazon.fr/Working-Effectively-Legacy-Michael-Feathers-ebook/dp/B005OYHF0A)
- *The Clean Coder: A Code of Conduct for Professional Programmers*, Robert C. Martin (http://www.amazon.fr/Clean-Coder-Conduct-Professional-Programmers-ebook/dp/B0050JLC9Y)
- *L'entreprise du bonheur*, Tony Hsieh, (http://www.amazon.fr/Lentreprise-du-bonheur-Tony-Hsieh/dp/2848994878)


## Ressources

### Dette technique


- https://medium.com/@joaomilho/festina-lente-e29070811b84 — traduction : http://www.occitech.fr/blog/2014/11/intro-dette-technique/
- *Technical Debt*, David Shirey (http://www.sitepoint.com/technical-debt/)
- Maîtriser sa dette technique, Julien Kirch (http://blog.octo.com/maitriser-sa-dette-technique/)
- *Technical Debt Quadrant*, Martin Fowler (http://martinfowler.com/bliki/TechnicalDebtQuadrant.html)
- *How to Measure Technical Debt*, Assaf Stone (http://www.softwareandi.com/2011/12/how-to-measure-technical-debt.html)
- *On technical debt, writer's block, cats, and canaries*, Shark Punk (http://blog.sharkpunch.com/post/82245897943/on-technical-debt-writers-block-cats-and-canaries)
- La dette technique, l'épée de Damoclès de l'éditeur logiciel, Benoît

Sautel (http://www.fierdecoder.fr/2014/06/la-dette-technique-l-epee-de-damocles-de-l-editeur-logiciel/)

## Quoi développer


- *Be lazy, don't build that feature (yet)*, Sébatien Saunier(http://sebastien.saunier.me/blog/2013/07/24/be-lazy-dont-build-that-feature-yet.html)
- *Product Strategy Means Saying No*, Des Traynor (http://insideintercom.io/product-strategy-means-saying-no/)
- Retrouver la souplesse en management grâce à Donald Reinertsen, Alexis Nicolas (http://www.mom21.org/retrouver-la-souplesse-en-management-grace-a-donald-reinersten/)
- De l'importance du cadrage de projet, Alain Buzzacaro (http://lebuzzdubuzz.wordpress.com/2014/03/16/de-limportance-du-cadrage-de-projet/)
- *The Power Of Simplicity*, Jeremy Keith (http://adactio.com/articles/6574/)
- Loi de Conway, Antoine Vernois (https://blog.crafting-labs.fr/?post/2009/03/20/loi-de-Conway)
- Les todo-list sont-elles contre-productives ?, Lionel Dricot (https://ploum.net/les-todo-list-sont-elles-contre-productives/)


## Comment développer


- Code défensif et sur-optimisation de code, Amaury Bouchard (http://www.geek-directeur-technique.com/2013/07/31/code-defensif-et-sur-optimisation-de-code)
- Entre complexité et simplicité, vous misez sur quoi ?, Frédéric Hardy (http://blog.mageekbox.net/?post/2013/07/29/Entre-complexite-et-simplicite-vous-misez-sur-quoi)
- *From STUPID to SOLID Code!*, William Durand http://williamdurand.fr/2013/07/30/from-stupid-to-solid-code/
- Le développeur, créatif sous-estimé et expert bafoué, Sylvie Clément (http://www.oelita.fr/le-developpeur-creatif-sous-estime-et-expert-bafoue/)
- Les tests sont les panneaux routiers du développeur, Frédéric Hardy (http://blog.mageekbox.net/?post/2014/03/10/Les-tests-sont-les-

panneaux-routiers-du-d%C3%A9veloppeur)
- *Beyond Clean Code*, Anthony Ferrara (http://blog.ircmaxell.com/2013/11/beyond-clean-code.html)
- Les phases du programmeur, Jean-Baptiste Dusseaut (http://www.arpinum.fr/2014/04/08/les-phases-du-programmeur/)
- *Five Weird Tricks to Become a Better Developer*, Jordi Boggiano (http://slides.seld.be/?file=2014-11-27+Five+Weird+Tricks+to+Become+a+Better+Developer.html)
- *Do Things that Don't Scale*, Paul Graham (http://paulgraham.com/ds.html)
- *Why Bad Software Succeeds*, David Hayes (http://pressupinc.com/blog/2014/12/bad-software-succeeds/)

## Développer ensemble

- *Choose Collective Code Ownership*, François Zaninotto (http://redotheweb.com/2014/02/27/collective-code-ownership.html)
- L'Agilité des Agités, Nicolas Martignole (http://www.touilleur-express.fr/2014/03/21/lagilite-des-agites/)
- Des process dans une startup ?, Guillem Bertholet (http://www.guilhembertholet.com/blog/2014/03/21/des-process-dans-une-startup/)
- Le Meilleur Process, C'est Pas De Process, Lukasz Szyrmer (http://www.infoq.com/fr/articles/no-process-best)
- *You are what you document*, Yevgeniy Brikman (http://brikis98.blogspot.fr/2014/05/you-are-what-you-document.html)
- *Manifesfo for Software Craftsmanship* (http://manifesto.softwarecraftsmanship.org/#/fr-fr)
- *Mean People Fail*, Paul Graham (http://www.paulgraham.com/mean.html)
- *Measuring Success in Web Projects*, Kevin Herrington (http://stauffer.com/blog/2014/11/10/measuring-success-web-projects)
- *We Invite Everyone at Etsy to Do an Engineering Rotation: Here's why*, Dan Miller (https://codeascraft.com/2014/12/22/engineering-rotation/)

## À propos de la refonte de zéro


- Ploum.net en J2EE, Lionel Dricot (https://ploum.net/ploum-en-j2ee/)

# À propos de l'auteur

Expert technique en technologies Web, Bastien est un passionné qui s'est investi très tôt dans la promotion des logiciels libres et des formats ouverts.

Au quotidien, il met ses compétences au service de grands groupes, pour lesquels il dirige techniquement des refontes de sites à fort trafic et dans des contextes internationaux. Il intervient également auprès de *startups* ou de petites structures qui ont besoin de compétences fortes et d'une *task-force* efficace pour la réussite de leurs projets.

D'une manière plus générale, son approche du développement Web met en avant les méthodes et techniques qui favorisent l'évolution à long terme des architectures Web, et qui facilitent la mise au point de systèmes robustes et innovants.

En 2012, il co-fonde JoliCode, avec pour objectif d'apporter son soutien à des projets innovants. À la même période, il co-fonde l'association Démocratie Ouverte

Il a contribué aux projets Giroll (un Groupe d'Utilisateurs de Logiciels Libres, à Bordeaux), Jelix, Démocratie Ouverte. Il est également conférencier, notamment sur le thème de la dette technique au PHP Tour, Forum PHP, Sud Web et Paris Web. De manière générale, il participe fréquemment à des conférences sur le Web.

# Colophon

La police utilisée pour le corps du livre est la Frutiger, créée par Adrian Frutiger sur demande l'aéroport de Paris Charle de Gaulle. Celle destinée pour les titres est la Exo, dessinée par Natanael Gama.

Les Contrôleurs Marie Alhomme, Corinne Schillinger et Loïc Mathaud espèrent que vous avez effectué un agréable voyage.