Le projet **DELIRE**

Développement par **E**quipe

de **L**ivrables **I**nformatiques

et **R**éalisation **E**ncadrée

PC2 – Nombre d'erreurs, mode de tests, maintenance

Des erreurs, des erreurs, toujours des erreurs.

Depuis que le monde est monde (c'est-à-dire globalement depuis ma naissance) les hommes font des erreurs lorsqu'ils codent. C'est normal. Je l'ai déjà dit mais je le répète ici, l'objectif de l'ingénierie est de corriger ses erreurs.

Depuis que le monde est monde, le nombre d'erreurs faites pas les hommes lorsqu'ils codent est un invariant : c'est globalement 50 erreurs par kloc (kilo lines of code).

Et l'évolution des langages n'y a pas changé grand-chose : certes les langages sont plus évolués et plus productifs, le nombre de personnes susceptibles de les utiliser a cru, mais cela ne change pas le nombre d'erreurs.

Et le fait de devenir des experts ne va pas changer globalement le nombre d'erreurs que vous ferez : simplement vous allez devenir plus productifs (et la confiance que vous prendrez vous conduira à faire des erreurs que vous n'auriez pas faites initialement) et vous serez capables de les débusquer plus vite (quoique…)

50 erreurs par kloc, cela ne signifie pas qu'il reste 50 erreurs par kloc au moment où vous entrez en phase de convergence. 50 erreurs par kloc, cela représente l'ensemble des erreurs potentielles dans tout le cycle de vie du code : avant-projet, architecture, coding, unit tests, integration tests, functional tests, stress tests et maintenance après mise du produit sur le marché.

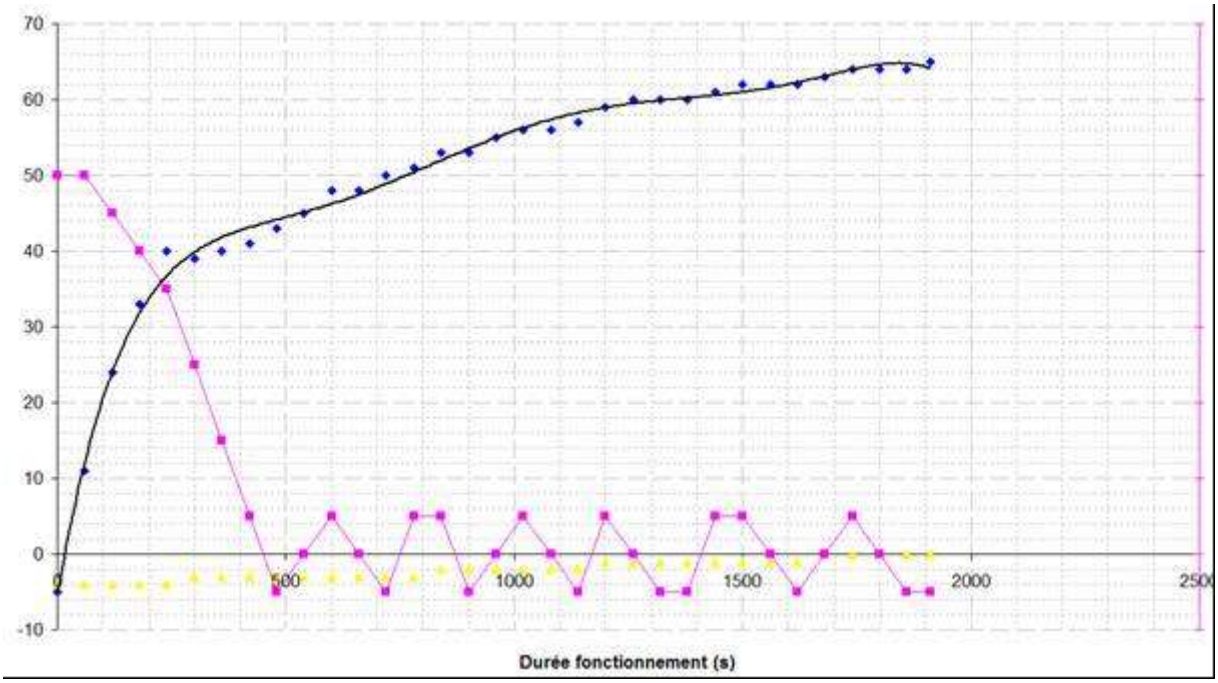
Ceci vous explique mieux pourquoi j'ai parfois insisté pour vous demander combien de lignes de code il y aurait en final dans votre projet.

Attention à ne pas tout appeler erreur de code : lorsque vous mettez au point un interface utilisateur, il arrive souvent qu'on procède par tâtonnement : on visualise le résultat puis on corrige l'esthétique par exemple en réalignant certaines fenêtres. Je ne considère pas que ceci soit considéré comme autant d'erreurs que de tâtonnements. Mais si par la suite vous vous apercevez que votre fenêtre ne répond pas à ce pour quoi elle a été conçue, alors oui il s'agit d'une erreur.

Optimisation du coût.
On dit classiquement qu'une erreur trouvée en phase d'architecture coûte 1, en phase de coding coûte 10, en phase de Unit test coûte 100, en intégration tests 1000… Mais bien entendu, il s'agit d'une moyenne : à un moment, le fait de relire votre code en phase de coding ne vous fera plus découvrir d'erreur et il est largement temps de passer aux unit tests.
Bref, exprimé autrement, il est temps de changer de méthode de tests lorsque la productivité (nombre d'erreurs trouvées rapportés au nombre d'heures de tests passées) de la phase précédente a chuté d'un facteur 10.

Dans les grands projets, on considère que l'effort de tests est régulièrement réparti. On mesure la pente de découverte durant la phase (nombre d'erreurs trouvées rapportés au nombre d'heures de tests passées). En début de phase la découverte est significative, mais l'effort pour trouver un nouvel incident devient de plus en plus important, et on finit par tendre vers une asymptote. Et quand l'effort pour trouver un

nouveau problème devient disproportionné, il est temps de changer de méthode de tests



A titre d'information, chez DS le nombre de personnes employées en R&D est d'environ 4000 personnes, dont globalement 1500 développeurs et 1500 testeurs. A raison ne serait-ce que d'une erreur découverte pas jour et par testeur, c'est globalement 1500 erreurs qui tombent chaque jour en début de phase de tests. On est dans des ordres de grandeurs où les statistiques peuvent s'appliquer. Exprimé autrement, cela signifie qu'un testeur va peut-être avoir de la chance le premier jour mais un autre non, qu'un testeur va surestimer un nombre d'heures et un autre va les sous-estimer, globalement les conditions de tests seront identiques du début à la fin de la phase de tests.

Il n'en est pas de même lorsque vous allez faire vos tests pour le projet DELIRE. Le nombre d'heures que vous allez passer, le nombre d'erreurs à trouver, la productivité de chacun peut fortement modifier les résultats et donc fausser les statistiques.

Ma recommandation est que vous exécutiez le plan de tests tel que vous l'avez défini.

1. Je fais l'hypothèse (sans doute un tantinet optimiste) que vous avez rédigé, écrit, exécuté et fait converger vos tests unitaires.
2. Je fais également l'hypothèse que ces tests unitaires couvrent l'ensemble des fonctionnalités externes de votre composant (et que celles-ci sont conformes à l'architecture), valident l'ensemble du code (conformément aux STDs), permet de valider les performances, la convivialité, la finalité … dudit composant. Bref, tout ce qu'il était humainement possible de détecter au niveau du composant seul a été mis en place
3. Je fais l'hypothèse que les tests sont faits en réutilisant les tests unitaires.
4. Je fais l'hypothèse que les scénarios fonctionnels (qui ont été définis lors de l'élaboration de la maquette du produit, on peut rêver) vous permettent de

valider l'ensemble des troles et pour chacun de ces rôles l'ensemble des fonctionnalités nécessaires.

5. Je fais l'hypothèse que vos stress tests se limiteront à ré-exécuter sans problème plusieurs fois votre scénario de démonstration

Il n'y a plus qu'à dérouler le plan

Ceci étant posé, il est néanmoins intéressant de définir des indicateurs : donc faire des prévisions, faire des mesures, et comparer prévisions et mesures.

Mais toute méthode a ses limites, et un jour ou l'autre une des méthodes de tests va commencer à ne plus être rentable. Il est temps de passer à la méthode suivante.
Fort bien Antoine, mais quel critère pourrons-nous plus tard utiliser pour savoir quand il faut switcher ?
En règle générale, deux méthodes sont appliquées

1. La première, qui est celle que je vous propose dans le projet DELIRE, consiste à définir les tests sous forme d'un plan à exécuter :
    a. Liste des tests
    b. Timing pour les exécuter

    Dans ce cadre, on fait l'hypothèse que les tests sont exhaustifs, couvrent tout le produit, et que celui-ci est réputé bon pour le service si tous les incidents trouvés en période de tests sont corrigés.
    Cela conduit à devoir reprendre des scénarios interrompus pour cause d'incidents pour les achever une fois la correction disponible
2. La seconde consiste à attendre que la limite asymptotique de la courbe apparaisse : cela montre que l'on atteint les limites de la méthode. On passe alors à la phase suivante (tests fonctionnels, stress tests…)
    En fait, Cela montre qu'il faut changer de méthodologie mais pas nécessairement de phase. Si la définition de vos tests fonctionnels est incomplète, vous atteindrez la limite asymptotique, alors que la réécriture des nouveaux tests fonctionnels vous aurait permis de continuer sur une bonne pente la découverte des incidents..

<u>Les tests d'intégration</u>

Vos unit tests ont permis de valider que chaque composant

3. A bien implémenté l'ensemble des méthodes définies dans les STDs
4. En respectant les algorithmes proposés
5. En prenant en compte tous les cas d'erreur remontés par les composants appelés
6. En traitant tous les cas d'entrée imaginables
7. En traitant autant que faire se peut tous les problèmes solubles à son niveau
8. En gérant de façon propre tous les retours d'erreur.

Ce n'est pas toujours le cas ? Du moins, ce devrait l'être, puisque telle est la mission des tests unitaires.
Dans la suite de ce paragraphe, nous faisons l'hypothèse que c'est le cas.

On peut donc imaginer que l'intégration des différents composants entre eux ne sera qu'un jeu d'enfant. Ce n'est pas souvent le cas.
Chacun, pour faire son implémentation, et en particulier pour appeler les interfaces des autres composants et en simuler le comportement s'est appuyé sur le document d'architecture : il en existe peut-être plusieurs versions dans la nature.

Le document d'architecture peut présenter un certain flou, pouvant prêter à interprétation, d'où risque de divergence
Chacun a implémenté, et donc les risques d'erreurs dans l'implémentation sont présents
Enfin chacun, dans le cadre des problèmes rencontrés en phase de maintenance a pu faire évoluer l'architecture de son composant :

1. Interface supprimée
2. Interface ajoutée
3. Interface modifiée (nouveaux arguments, changement de types des arguments, changement de l'ordre des arguments, nouvelle contrainte sur els entrée…
4. Ajout de nouveaux types d'erreurs renvoyées non spécifiée dans l'architecture du produit.

On ne va pas se poser le problème de la tenue de l'assemblage des pièces du moteur si on n'est même pas capable d'assembler lesdites pièces.
Les tests d'intégration ont pour objectif de vérifier que les appels inter-composants se passent bien : bref, pas de carton
Mais la méthodologie ne consiste pas à mettre tous les composants ensemble et à secouer le cocotier. On va procéder de façon méthodique.

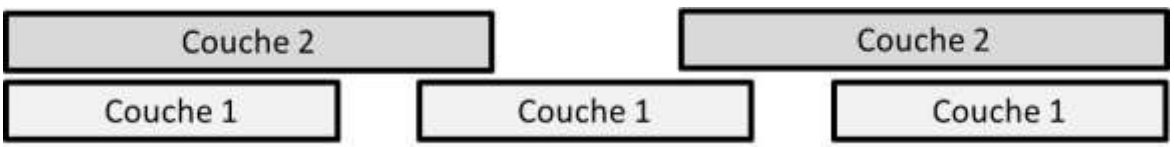
On peut imaginer que les différents composants, depuis la base de données jusqu'à l'interface utilisateur, définissent un certain nombre de couches.

1. On considère qu'un composant qui n'appelle aucun autre composant est un composant de la couche 1
2. On considère qu'un composant qui appelle un autre composant de la couche ne peut pas faire partie d'une couche inférieure à I+1

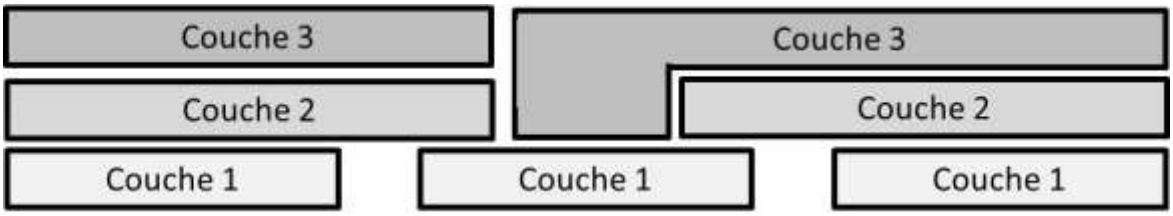
La bonne nouvelle, c'est que nous avons défini les tests unitaires de chacun des composants. On va donc les rejouer selon un certain ordre, en ne simulant pas les autres composants, mais en les appelant.

On va commencer par tester la couche 1. Mais Antoine, c'est juste re-exécuter les tests unitaires, et on sait qu'ils fonctionnent bien : oui certes, mais on va ainsi s'assurer qu'on n'a rien oublié dans les livraisons. Si on trouve des erreurs, on corrige avant de remonter d'un cran

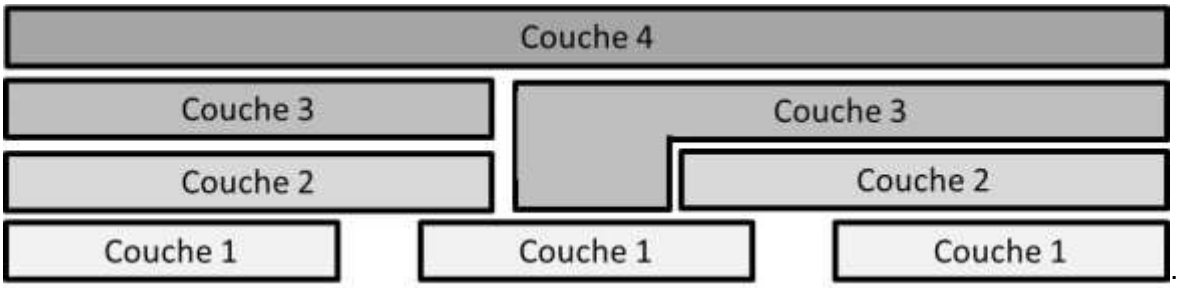
Puis ce premier test réussi, on teste l'ensemble couche 1 + couche 2. . Si on trouve des erreurs, on corrige avant de remonter d'un cran



Puis un autre niveau



Et on remonte ainsi, couche par couche, jusqu'à arriver à avoir validé que l'ensemble entier des unit tests s'exécute sans problème.



A la fin des tests d'intégration, on a juste vérifié que les communications entre les modules se passaient bien, pas plus, pas moins.

Les tests fonctionnels

Les tests fonctionnels ont pour but de démontrer que le produit est conforme aux spécifications, tant celles relatives aux fonctionnalités définies dans les SFG que celles relatives aux performances, à la convivialité ou à la fiabilité définies dans le PQ.

Le principe de base est d'exécuter des scénarios et de vérifier que la réponse du produit est conforme à ce qui est attendu.
Ces scénarios peuvent être automatiques ou manuels.
Il est de la responsabilité des tests, automatiques ou manuels, de restaurer en fin de séquence l'état initial de la base de données.

Un scénario automatique doit être capable de vérifier le résultat produit avec un résultat de référence défini ou calculé. Il doit doc surtout être capable d'identifier lorsqu'il ne se passe pas bien.
La portée des scénarios automatiques est nécessairement limitée : on teste des fonctionnalités élémentaires.
Exemple :

1. Demander le nombre de paragraphes dans la base de données : NbP
2. Effacer un paragraphe
3. Demander le nombre de paragraphes : il devrait être égal à NbP-1

(On tentera dans la mesure du possible de ne pas faire de commit après ce test)
Pour des raisons de productivité on commence par les scénarios automatiques.
Autant éliminer le maximum d'erreurs en minimisant l'effort à accomplir : la première qualité d'un ingénieur c'est la paresse.

Dès que l'objectif du test prend une certaine ampleur, il devient impératif de faire des scénarios manuels. Dans ces scénarios manuels on va certes valider les fonctionnalités du produit, mais on va valider en même temps un tas de petits accotés :

1. Validité et orthographe des messages
2. Performance du produit
3. Nombre d'interactions de la commande en cours
4. Positionnement correct des fenêtres, et des champs à l'intérieur des fenêtres….

Les scénarios manuels doivent également tester les cas d'erreurs : non pas tous les cas, mais les cas les plus classiques et qui font partie de l'utilisation quasi nominale du produit

Les scénarios sont organisés par rôles, et pour chaque rôle ils doivent être organisés en complexité croissante :

1. Une commande élémentaire ou un groupe de commandes élémentaire
2. Un enchainement de commandes
3. Un processus élémentaire
4. Un processus complet
5. Un scénario global qui couvre une utilisation complète du produit. Le scénario de démonstration devrait en faire partie.

On note toutes les erreurs. A la fin de l'exécution de chaque scénario, si le nombre d'erreurs de la backlog dépasse une certaine valeur, par exemple 15 erreurs soit 3 par personne, à corriger, alors on arrête d'exécuter les scénarios et on fait baisser la backlog. On reprend quand on est passé en dessous de 5 incidents encore ouverts par exemple.

A la fin des tests fonctionnels, on sait que le produit fonctionne pour tous les rôles, pour toutes les fonctionnalités, en utilisation nominale. Mais uniquement en utilisation nominale. On n'a aucune idée de ce qui se passe quand on sort des sentiers battus. C'est l'objet des stress tests.

Les stress tests

Les stress tests ont pour objectif de vérifier le comportement du produit en dehors du mode nominal, en d'autres termes comment le produit se comporte lorsque l'utilisateur fait une action non standard. Ce qui est important n'est pas que le produit corrige tout seul les erreurs de l'utilisateur, mais qu'il ait un comportement sain. J'appelle comportement sain, et par ordre décroissant de priorité :

1. Qu'il n'y ait pas de corruption de données
2. Que le produit ne parte pas en sucette
3. Que le message d'erreur soit compréhensible (pas de : « Thickness too small »)
4. Que le produit puisse proposer un mode de résolution

La première démarche va commencer par recenser tous les cas d'erreurs prévu par le produit, et trouver les conditions pour provoquer lesdites erreurs.
Ensuite, on va chercher comment mettre en défaut le produit. Et là il faut être créatif :

1. Utilisation à outrance de l'undo

2. Utilisation du produit sans avoir initialiser la base de données
3. Accès simultané avec plusieurs fois le même user
4. Non-respect des règles d'intégrité : tous les objets ont systématiquement le même nom
5. Sauvegarde d'un document de 200 mégas
6. Interruption du processus durant une opération sur la base de données…

On va ainsi des constituer une collection de tests tous plus aberrants les uns que les autres. Malheureusement, mon expérience me susurre à l'oreille que ces cas se produisent plus souvent que vous ne l'imaginez.
Certains de ces tests sont simples à effectuer, d'autres demande un investissement important.
Certains de ses tests peuvent provoquer de gros dégâts (défaillance catastrophique) s'ils se passent mal, d'autres ne seront qu'un message mal gaulé.
On va d'alors être capable de trier nos stress tests, un peu comme on avait trié les risques dans le PGR.

1. On exécutera les stress test à défaillance catastrophique
2. Et dans les autres, on définira le nombre qu'on souhaite exécuter, et on commencera par les plus simples.

En dehors des tests à défaillance catastrophiques qui doivent impérativement être corrigés, je peux admettre que les erreurs trouvées durant le stress tests ne soient pas corrigées lors de la sortie du produit sur le marché, mais fassent l'objet d'une information dans le Program Directory, qui est le document des erreurs connues qui accompagne la sortie du produit.

Le suivi des tests

Faire des tests dans le projet DELIRE a pour objectif de trouver des erreurs pour améliorer la qualité de votre produit et tendre d'un démonstrateur vers une offre industrielle.
Mais comme nous sommes dans le cadre du projet DELIRE, l'objectif est également d'apprendre. Et comme vous l'avez compris, le principe de base de DELIRE c'est l'indicateur : une prévision, une mesure, une comparaison entre prévision et mesure.

Ce que je vous propose, au moment de vous lancer dans les différents plans de test est de commencer par prévoir ce à quoi vous vous attendez.
Le mécanisme de la prévision doit être quelque chose de simple. Il y a grosso modo 50 erreurs potentielles par kloc, nous allons viser d'en trouver 20. Nous allons répartir : 5 en unit tests, 2 en integration tests, 10 en fonctionnal tests, 3 en stress tests.

Ensuite, on comparera ce que l'on a trouvé en test par rapport à ce que l'on avait prévu. Notez également le temps passé dans chacune des phases, incluant la préparation et l'écriture de tests. Ceci vous permettra au final :

1. D'avoir une meilleure idée du pourcentage de ce qui est trouvée par chaque phase de tests

2. D’avoir une idée (nombre d’incidents trouvés / effort investi pour la phase de tests) de savoir quel mode de test est le plus productif. Je vous conseille d’ailleurs de tracer la courbe [nombre d’incidents trouvés / effort cumulé de tests]

Je vous recommande de noter pour chaque incident trouvé sa typologie : fonctionnalité, fiabilité, performance, convivialité.
N’oubliez pas de noter pour chaque incident le coût de sa résolution.

Je rappelle qu’après avoir corrigé un ou des incidents, on rejoue les test, du moins tous ceux qui sont automatisés :

1. Unit tests
2. Tests d’intégration
3. Test fonctionnels basiques automatisés.

Même si ces jeux de tests sont automatisés, leur exécution représente un certain coût. C’est pourquoi on essaye d’organiser des campagnes de correction, de façon à réduire le nombre de réexécution des tests. La première qualité d’un ingénieur, c’est la paresse. Optimisez, optimisez, il en restera toujours quelque chose.

Le mode maintenance

Tout développeur pense pouvoir livrer du code sans erreur.
L'histoire de l'informatique nous apprend pourtant que le code 0 défaut n'existe pas : Elle montre que grosso modo le nombre d'incident par kloc (millier de lignes de code) est d'environ 50 sur toute la vie du composant logiciel : spécification, architecture, coding, convergence, qualification et phase de maintenance
.
Entre la demande client et la livraison du produit résultant, de nombreuses occasions de faire des erreurs vont être rencontrées :

1. Non compréhension du besoin client
2. Erreur d'architecture du système
3. Erreur d'architecture interne du composant
4. Problème de fiabilité, de PCS (Performance Capacity Scalability), de convivialité, faute dans un message, corruption de données…
5. Introduction d'une régression lors d'une correction du composant…

Aujourd'hui les outils de Dassault Systèmes équipent des centaines de milliers d'entreprise. Tous les jours, des millions d'engineers utilisent nos solutions pour créer les produits de demain. Faire de la maintenance la priorité des équipes de développement de DS, et s'assurer que le processus de maintenance permet de livrer en temps et en heure des correctifs de qualité, sont des nécessités industrielles.

Se retrouver bloqué dans l'exécution de son travail peut non seulement être pénible, mais également couter cher à l'entreprise cliente (contrat perdu, non-respect des engagements vis-à-vis du donneur d'ordre, retard de programme, nécessité d'élaborer une autre solution…
De plus, étant donné que les entreprises travaillent de plus en plus en entreprise étendue, et que les programmes mobilisant de plus en plus d'engineers, ce coût peut rapidement devenir critique du fait du nombre d'acteurs en collaboration impactés.

L'activité de maintenance est donc une priorité de DS R&D
Dans de nombreuses équipes, l'activité de maintenance est l'activité principale. 60% de travail consacré à la maintenance n'est pas un cas aberrant ou isolé.
De plus le nombre croissant de niveaux sur lequel un incident peut se produire pousse naturellement à un accroissement de la charge de maintenance.
En parallèle, l'évolution des architectures supportées (Cloud Computing, tablette et Smartphone…) est également source de nouveaux problèmes

Mais, et vous n'en serez guère surpris, la maintenance n'enchante guère les ingénieurs de développement. C'est toujours un choc pour les jeunes embauchés, de découvrir que cette entreprise qui leur a été présentée comme développant les produits du futur (ce qui est vrai) a deux caractéristiques :

1. Il faut plus d'une heure de galère en voiture pour arriver le matin, du fait des travaux du futur tramway. .
2. L'essentiel du travail consiste à déverminer le code massacré par les anciens.

Bref, en matière de maintenance, mieux vaut prévenir que guérir

Des méthodes existent pour réduire le nombre d'incidents résiduels lorsque le code est livré en clientèle :

1. Investissement sur les spécifications
2. Soin particulier apporté à l'architecture logique
3. Ingénierie intelligente des tests

Il me semble que je vous en ai déjà parlé

Une autre bonne méthode pour réduire la charge de maintenance est de limiter ne nombre de lignes de code. Non pas en interdisant aux développeurs de travailler (même si pour certains ce ne serait aps stupide), mais en favorisant au maximum la réutilisation de code existant.
Cette réutilisation permet de :

1. Réduire le cout de développement
2. Accélérer la sortie d'un produit
3. Diminuer le coût de maintenance
4. Réduire les risques d'incohérence du produit.

Une autre méthode pour réduire la charge de maintenance est de faire des opérations ciblées. Certes, mais ceci suppose de connaitre la localisation des erreurs. Ce qui semble délicat
La méthode consiste à identifier par mesure statistique les points d'accumulation de problèmes dans le code de DS. C'est le rôle de l'outil que nous avons développé, DSDC (DS Defect Classification) : à partir des informations renseignées lors de l'ouverture et de la clôture des incidents, il permet de mettre en exergue des zones de non qualité où des opérations préventives seraient particulièrement judicieuses et fécondes.

Mais hélas, trois fois hélas, ce travail en amont n'éliminera pas l'ensemble des erreurs du produit, tant s'en faut. Des incidents seront détectés par les clients, et il sera nécessaire de les corriger.
La qualité de la maintenance repose sur 4 principes

1. Etre sûr qu'on a bien compris le problème
2. Corriger rapidement : le client attend. Qui plus est, plus la correction est longue plus les risques de duplicate et donc plus le nombre de clients en attente augmentent.
3. Avoir une correction efficace : corriger le problème identifié, et tout le problème identifié.
4. Ne pas dégrader l'architecture du produit : une correction bricolée est acceptable pour dépanner le client, mais une solution plus élégante tant à cours qu'à long terme doit être trouvée et mise en place par la suite.

Vitesse ne veut pas dire précipitation. Rien n'est plus désastreux en termes de maintenance que d'introduire, dans le cadre de la correction d'un incident, une ou des régressions dans le comportement du composant

1. Avoir une fonctionnalité nouvelle en défaut est acceptable
2. Avoir une correction annoncée qui ne fonctionne pas est tolérable
3. Avoir une régression de comportement d'une fonctionnalité qui marchait auparavant est inacceptable : elle peut mettre en péril la bonne réalisation d'un projet de l'entreprise.

Avant de livrer une correction, il est important de rejouer les Unit Tests du composant, voire de les compléter par le test qui couvre le cas corrigé dans ladite correction.

De façon à optimiser le travail de maintenance, en particulier la qualité des données qui sont échangées entre le client et le développeur, des méthodologies spécifiques doivent être décrites et exécutées. Elles font l'objet de la documentation de maintenance. Voire PC4 – la documentation associée au produit.