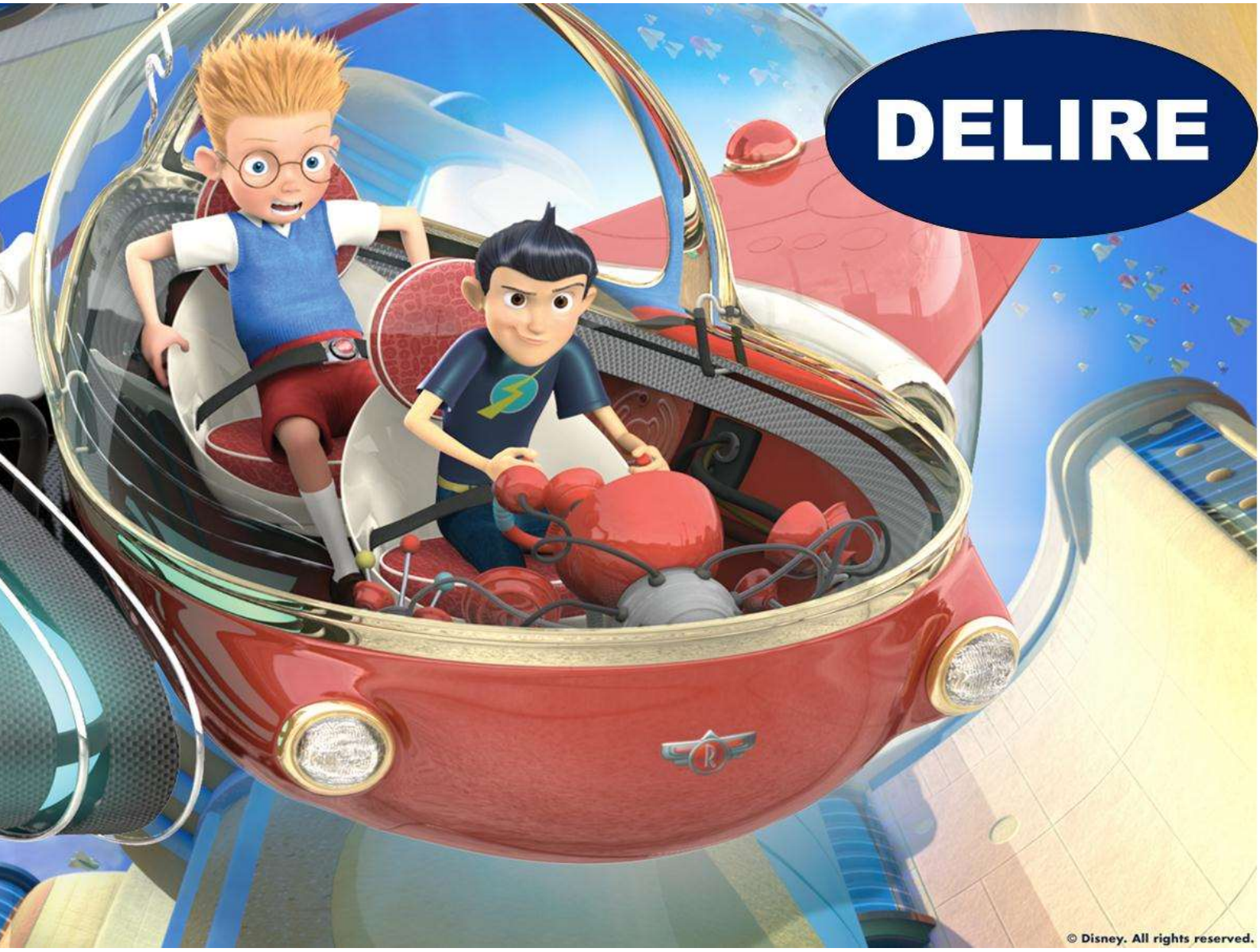


Le projet **DELIRE**

Développement par **E**quipe de **L**ivrables **I**nformatiques et **R**éalisation **E**ncadrée

PR3 – Unit tests et gestion des erreurs

Les tests, une priorité

Je suis persuadé que, lorsque vous recevez un cadeau à Noël, s'il ne fonctionne pas cela vous met en rogne. Appliquez-vous la même discipline lorsque vous allez livrer du code dans le cadre du projet ; ne faites pas la promotion de code dont vous savez pertinemment que le niveau de qualité est insuffisant voire déplorable :

1. Tout d'abord par respect pour vos partenaires dans le projet DELIRE puis plus tard dans votre carrière. .
2. Ensuite pour vous astreindre à une discipline qui sera votre force dans votre carrière : on saura que quand Myself livre du code, c'est généralement béton.
3. Enfin, parce que le jour où vous dirigerez une équipe, vous supporterez mal d'avoir de la daube livrée, mais qu'il sera difficile de demander à vos collaborateurs de livrer quelque chose de propre si vous-même préalablement ne le faisiez pas.

Et pourtant, dans l'industrie du logiciel, le code qui est généralement promu n'est ni fait ni à faire. 4 raisons à cela :

1. La phase de réalisation comporte non seulement le développement du code, mais également le développement, l'exécution et la convergence des Unit test, et l'éradication des erreurs mise en évidence par ces Unit tests. On prend toujours du retard dans la phase de réalisation, et on finit par sacrifier les tests. Il faut dire que c'est difficile de renoncer à une fonctionnalité. Et puis, on est intrinsèquement persuadé que le code qu'on est en train de développer est naturellement d'excellente qualité.
2. On a du mal à se définir un planning. Comme l'esprit humain est optimiste, on pense qu'on a encore le temps de tout faire. Je constate cela chez mes enfants qui ont globalement votre âge : c'est dur de se mettre au travail tant qu'on n'est pas acculé à le faire. Résultat, on retarde les tests jusqu'au moment où il est trop tard.
3. Pisser du code c'est noble. Faire des tests c'est « chiant ». En écrivant et en exécutant des tests, on a le sentiment de faire des tâches de faible niveau, peu gratifiantes et finalement indignes de soi-même. Dès lors, le choix est rapidement fait Ce n'est pas évident de comprendre que dans un projet l'intérêt du groupe prime sur sa satisfaction personnelle.
4. Comme on a pas défini les tests à exécuter pour valider le code, au moment où il faudrait le faire et où on est déjà en retard, on se dit qu'on aura pas le temps de venir à bout de la tâche de tests et qu'il vaut donc mieux finir les « dernières » fonctionnalités. Au pire on fera les tests en parallèle des tests d'intégration (attitude que je qualifierai de « criminelle »)

C'est difficile de choisir un jouet. Surtout si, quand on était mioche, il suffisait de faire un caprice pour avoir les deux

Le plus simple, pour ne pas tomber dans ces pièges, est de s'imposer 2 disciplines :

1. On commence par écrire le code des tests avant de coder l'application : dès lors que les tests sont disponibles, on ne va pas ne pas les utiliser. Et qui plus est, on bénéficiera des idées d'architecture ou d'écriture de l'application que la mise au point des tests aura suggérées.
2. On s'impose un calendrier. Sur les 33 jours (16 janvier – 17 février) que dure la phase de réalisation on peut par exemple proposer :
    a. 7 jours pour écrire les tests, le banc de tests et totalement finaliser l'architecture interne de l'application
    b. 14 jours pour écrire le code de l'application
    c. 12 jours pour exécuter et faire converger les tests, et promouvoir le code.

Au bout d'une semaine de développement de l'application proprement dit, on est en général déjà fixé sur ce qu'on sera capable de produire.

Au-delà du test de fonctionnalité

Les Unit tests vous permettront de valider que lorsque vous appelez les services que vous avez développés (en vous appuyant sur votre banc de tests pour les services non disponibles) le code ne part pas en sucette.
C'est déjà un premier résultat. Mais ce n'est pas le seul attendu.

Vos Unit test vont permettre :

1. De valider que l'ensemble des branches du code est atteignables et se déroulent bien
2. De valider la performance unitaire de chaque couche de logiciel (le temps ayant été défini à partir des objectifs qualité définis dans le PQ
3. De valider que l'ensemble des cas d'erreurs remontées par les services externes appelés, ou identifiés par vos traitements, sont correctement traitées (rattrapage, ou message compréhensible) par votre composant
4. De valider que les objets alloués sont correctement désalloués en fin de traitement.

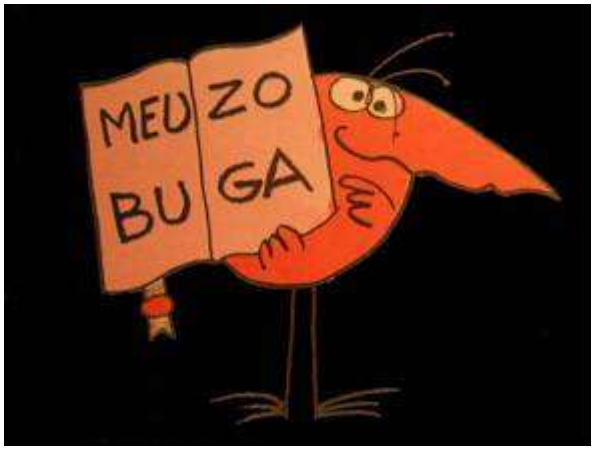
Vos Unit tests seront également utilisés comme test lors de la phase d'intégration (principe de paresse)
Vos Unit tests seront enfin réutilisés lorsque vous serez amenés à faire des corrections dans votre composants

Bref bétonnez l'écriture de vos Unit tests

Le modèle CUPRIMDSO d'IBM

Lorsque nous avons commencé, en 1981, à mettre CATIA sur le marché, nos critères de qualification était on ne peut plus simplets : du moment que cela se compilait et que cela se linkeditait, globalement c'était bon. En V1, nous n'avons jamais dépassé les 30 clients, et nous avions la capacité à traiter les problèmes de maintenance au fur et à mesure qu'ils nous étaient adressés.

Lorsque nous avons sorti la V2, en 1984, la bonne nouvelle est que le nombre de client a explosé. De ce fait, CATIA est devenu un produit majeur de l'offre IBM, lequel s'est aperçu que nous devenions sa principale source d'incidents clients. En 1985, IBM a détaché chez DS pour une durée de 3 ans Philippe P, un spécialiste de la qualité logicielle.



Sa première question a été : comment faites-vous vos plans de tests.

Notre réponse a été on ne peut plus claire : Ga, Bu, Zo, Meu ! C'est clair, nous n'en avions jamais fait.

Philippe nous a expliqué qu'IBM avait développé un modèle de qualité pour ses produits logiciels, le CUPRIMD, que je vais vous résumer ici.

C : Capability (Fonctionnalité)

L'objectif d'un progiciel n'est pas de faire plaisir à ceux qui le développent mais de rendre service aux clients qui l'utilisent en leur permettant d'exécuter un process end to end. Si le produit leur permet de réaliser 90% de ce dont ils ont besoin, l'objectif n'est pas atteint : il manque une fonctionnalité. En phase de spécification la priorité est de comprendre, formaliser et valider le scénario du processus du client. Ce sont les fameux scénarios fonctionnels qui vous permettront de valider à terminaison votre produit.

U : Usability (Convivialité)

Antoine, l'esthétisme n'est-il pas le point le plus important de la convivialité ?. Aussi surprenant que cela puisse paraître, la réponse est ni oui ni non, tout au contraire. L'esthétisme est un critère subjectif : personnellement je trouve que la tour Eigfelle st une horreur, un tas de ferraille. Mais je peux comprendre que d'autres ne soient pas d'accord avec moi.

La convivialité d'un produit va se traduire par 3 critères :

1. La facilité à pouvoir entrer dans le produit et à l'utiliser lors du premier contact. Il faut pour ce faire que le produit soit logique dans son approche, et conforme à un standard :
    a. Au minimum en interne (les mêmes actes produisent les mêmes effets d'une commande à l'autre, les même termes sont utilisés dans les diverses commandes

   b. Et si possible un standard externe : ainsi le fait d'avoir des commandes conformes au standard d'Office (Home, Insert, Review, View…) ne peut pas nuire
   c. Le fait de standardiser les tableaux de bord dans les voitures a permis de faciliter le marché de la location
2. La productivité du produit :
   a. Minimisation du nombre d'interaction
   b. Choix des interactions les plus simples : un click vaut mieux qu'une saisie clavier
   c. Minimisation des efforts de mémoire
   d. Facilité du chemin pour trouver la bonne commande à effectuer
   e. Votre produit doit être performant à la fois pour le débutant et pour l'expert : pas toujours facile à concilier
3. La lisibilité
   a. Maximiser la place réservée au travail de l'utilisateur
   b. Chercher toujours la représentation la plus simple à comprendre
   c. Présentez toujours les informations critiques en premier. Et évitez dans la mesure du possible les fenêtres déroulantes qu'il faut scroller pour obtenir l'information recherchée.

P : Performance
C'est la capacité du système à faire un traitement dans un délai raisonnable.
Et cette notion de raisonnable n'est pas une notion d'informaticien, amis de valeur perçue par l'utilisateur :

1. Il y a des traitements complexe mais qui semblent basiques à l'utilisateur, il attend une réponse rapide.
2. Il existe d'autres traitements, complexe ou non, qui lui apparaissent comme complexes et pour lesquels il est prêt à attendre un certain temps.

Par défaut, lorsque le temps de réponse d'un système interactif dépasse une seconde, on dit que le système colle.
A titre d'exemple, se connecter à la base de données est une opération que l'utilisateur accepte comme longue. Profitez en pour capter un maximum d'informations et d'exécuter des prétraitements qui boosteront l'application par la suite.

R : Reliability (Fiabilité)
C'est la capacité du système à ne pas se planter. La fiabilité se mesure en général en MTBF (Mean Time Between Failures), durée moyenne avant de se prendre un carton. Un bon MTBF est de 200 heures : en gros on peut utiliser le logiciel de façon nominale pendant un mois avant d'avoir un plantage.

La mesure du MTBF s'effectue lors de tests manuels. OIL n'est cependant pas interdit d'enregistrer des scénarios de tests fonctionnels : le fait qu'ils marchent sans problème permet de se lancer dans des tests manuels avec moins d'appréhension. Pour garantir 200 heures, il faudrait avoir réussi une fois 1000 heures de test sans s'être planté. Ou avoir exécuté au moins 6000 heures de tests. Dans le projet DELIRE, la fiabilité est à peu près impossible à mesurer. Si vous êtes capables de réaliser toute la démonstration sans le moindre problème, ce sera déjà un beau résultat.

I : Installabity (Installabilité)
Installer un logiciel est toujours une angoisse pour les administrateurs. Le rêve serait le logiciel s'installe de lui-même, dans le cadre de l'exécution d'un script : si l'installation requiert des paramètres ceux-ci devraient être saisis et contrôlés en début d'exécution
Pour un gros logiciel, l'administrateur peut admettre que la procédure initiale puisse être longue et pénible. Mais par contre, il attend de l'application que les futurs update (Release, Functional Delivery, Fix Package) s'installe sans problème.

M : Maintenability (Maintenabilité)
Pour pouvoir corriger un problème, le développeur doit disposer :

1. De la description claire des conditions du problème et des symptômes engendrés
2. Des données nécessaires pour reproduire l'incident. A commencer par la version sur laquelle a été constaté l'incident.

Il est donc nécessaire de livrer avec le produit un descriptif qui va spécifier la démarche à suivre en cas d'incident. En particulier, dites bien les langues que vous supportez dans les rapports d'incidents, vous seriez bien en peine de corriger un incidents documenté en arabe ou en chinois.
La maintenabilité, c'est aussi pouvoir identifier qu'un incident émis par un client est déjà connu voire même que sa correction est immédiatement disponible. La bonne solution étant que le client puisse télécharger la correction sans intervention de votre part.
Enfin la maintenabilité d'un logiciel c'est tout ce que vous avez mis en place en terme de mode trace et de debuging durant la phase de réalisation.

D : Documentation
Un progiciel c'est bien, encore faut-il que l'utilisateur puisse l'utiliser. Il est donc nécessaire que les commandes du produit soit documentées, et que la documentation soit conforme au comportement du produit.
Il faut également que l'ensemble des cas d'erreurs soit documentés, et que l'attitude à avoir face à chacun de ces cas d'erreurs soit clairement expliqué.
Enfin, au-delà de ce que le produit fait, il est nécessaire de documenter ce que le produit ne fait pas, les cas qui ne sont pas supportés.
Enfin, au-delà de la documentation commande par commande, les principes de base qui régissent le produit, et qui vont permettre à l'utilisateur de se retrouver rapidement en confiance, doivent être clairement explicités. La documentation est un composant à part entière du produit.

Depuis, IBM a rajouté SO, Service and Overall Satisfaction, mais globalement le modèle est resté le même depuis 40 ans.

Lorsque vous trouverez des erreurs dans votre produit, commencez par en identifier la typologie : un problème de fonctionnalité, de convivialité, de performance ? Cela vous aidera à définir la priorité de l'incident.
Pour ce faire, ne restez pas dans une approche fournisseur (mon produit est conforme aux demandes exprimées) mais dans une approche utilisateur (mon produit me permet de faire mon travail)

Automatiser les tests

Vous aurez à ré-exécuter régulièrement vos tests. La bonne façon est de les automatiser : la paresse avant tout.

Il y a deux choses à automatiser

1. L'exécution des tests
    a. La bonne solution serait de développer un script qui relance régulièrement votre jeu de tests. par exemple tous les soirs à minuit. Ceci suppose que le jeu de tests s'exécute sur un desktop toujours allumé.
    b. La solution que vous retiendrez probablement est de lancer, en fin de soirée, un script qui enchainera toutes les opérations automatisables : compilations, linkedit, puis le jeu de tests.
2. L'analyse des résultats
    a. Si vous devez regarder l'ensemble des résultats de tests pour savoir si certains tests se sont mal passés, vous finirez par trouver cela pénible.
    b. La bonne solution est sans doute que, lorsqu'un test se passe mal, le test écrive un message dans un fichier dédié et que les résultats qui vont permettre d'analyser l'erreur soit stockés dans un fichier.
    c. Ce qui signifie que le test soit capable de valider par lui-même qu'il s'est bien déroulé, et donc qu'une partie du code du test soit là pour valider que le résultat produit est conforme à celui attendu :
        i. Temps d'exécution pour un test de performance
        ii. Mémoire consommé pour un test de capacity
        iii. Type d'erreur retournée en cas d'arguments volontaires mauvais…

La gestion des erreurs

Bien entendu, des erreurs vont être identifiées par le Unit tests. C'est leur rôle.

Mais l'exécution desdits Unit tests peut être l'occasion de trouver en bonus d'autres erreurs

1. Il existe sur le marché (et je fais l'hypothèse que votre marché se limite au logiciel libre sur le net) des outils d'analyse de code qui vous permettent de valider des règles pointeurs, des allocations de mémoire, des structurations de boucle
2. Il existe également sur le marché (et je fais l'hypothèse que votre marché est toujours le même) des outils de checks dynamiques pour identifier des problèmes de memory leaks
3. Il existe peut être sur le marché libre des outils qui identifient quel pourcentage de votre code est couvert par vos tests (sachant que l'optimum est bien entendu d'atteindre les 100%)

Il peut être également intéressant, une fois le code pissé, de faire une revue de code avec un de vos collègues : un œil extérieur ne peut nuire. Ce sera l'occasion de mettre en évidence les règles de programmation énoncées par le groupe et que vous n'avez pas respectées.

Bref, vous allez vous retrouver face à un nombre important d'erreur. Certes, l'optimum serait de toute les corriger. Mais le temps presse. Il est donc important que le groupe définisse les erreurs qu'il faut corriger impérativement et les erreurs ou les

warning qu'on peut laisser partir sur le marché, en ayant juré, c'est craché, de les corriger plus tard. Cette liste, initialement définie par le Responsable Qualité, sera ensuite enrichie durant les réunions de suivi du projet en fonction des erreurs identifiées par chacun des acteurs du projet

Ensuite, dans l'ensemble des tests qui ont émergés ou qui vont émerger (puisque une fois un ensemble d'erreurs corrigées, on ré-exécute les Unit tests) et qu'on va souhaiter traiter, il va falloir définir des priorités. Elles sont définies en fonction de 2 critères

1. L'impact pour l'utilisateur. Par exemple :
    a. Une fonctionnalité qui part en sucette ou qui ne se comporte pas correctement : priorité 1
    b. Une erreur non conforme : priorité 2
    c. Un problème de performances mineur : priorité 3
    d. Une faute d'orthographe dans un message : priorité 4
2. L'impact pour le fonctionnement du groupe
    a. On donne la priorité aux erreurs qui bloquent le plus de tests d'intégration par exemple
    b. Ou aux composants de base sur lesquels s'appuient un maximum de service

PCS

On regroupe souvent sous le terme performance 3 notions fondamentales :

1. La performance proprement dite
2. La capacity
3. La scalability

La performance proprement dite

On est purement dans l'optimisation d e la CPU. Pour un algorithme donné et une machine donnée, il existe une limite en deçà de laquelle on ne sait pas descendre. C'est cette performance dont on va chercher à s'approcher.

Souvent une relecture critique permet de mettre en évidence les erreurs les plus criantes.
Les outils de checks permettent d'identifier les sources où on passe le plus de temps : c'est souvent ceux-là qu'on va tenter d'optimiser en premier.
Les machines sont de plus en plus multi cœurs, et cela peut permettre de booster singulièrement une application. Encore faut-il que le code développé le permette

Quelques règles à prendre en compte :

1. Par défaut, on accepte qu'une amélioration de performance ne se fasse au détriment de l'architecture (bref on regroupe dans un même code des traitements qui étaient initialement distribués) que si l'amélioration est forte, générale (pas pour une commande utilisée 2 fois dans l'existence du produit), localisée et bien documentée.
2. On n'optimise pas quelque chose qui ne serait dépendant que d'un seul type de machine. Les améliorations doivent être valides pour tous les clients.
3. Une amélioration de qualité ne se fait pas en supprimant des tests de fiabilité, par exemple le contrôle de validité des arguments reçus ; par contre, si ces

mêmes arguments doivent être contrôlés 6 fois de suite, un peu de factorisation ne peut nuire. .

La capacity

C'est la possibilité pour votre application de traiter de grandes quantités de données. C'et en général à ce moment que les algorithmes en N^2 au lieu de N*log(N) vont faire très mal.

En règle générale, les développeurs font leur test sur 3 objets : tout va bien. Et c'est en clientèle, sur des quantités de données industrielles, qu'on se retrouve avec des temps de réponse de plusieurs heures. Bien sur, vous ne ferez pas des tests sur des bases de données de production. Mais simuler des tests en faisant croitre le nombre d'objets manipulés et en regardant la forme de la courbe de réponse est une démarche qui doit devenir basique.

On reproche souvent à Google sa position un tantinet hégémonique. Mais je vous conseille de vous pencher un jour sur l'architecture qu'ils ont mis au point pour rechercher une information donnée dans une masse de données hallucinante.

La scalability

C'est la capacité pour un logiciel à accepter un nombre croissant d'utilisateurs sans s'effondrer

1. Architecture 3 tiers
2. Architecture stateless
3. Load and balancing…

D'autant plus que l'évolution vers du cloud computing va rendre ce point critique.

Je sais bien qu'on vérifiera ce point en system tests, en fin du cycle de validation du produit. Mais si on ne l'a pas anticipé durant la phase de spécification puis en Unit tests, on risque de se préparer un réveil douloureux.

Le logiciel embarqué

Votre application DELIRE a pour objet de permettre à des milliers d'ingénieurs de concevoir simultanément un document complexe. Au pire, si ça part en sucette, ils redémarreront l'application.

Mais vous êtes-vous demandé comment on développe des logiciels critiques, par exemple un logiciel embarqué qui permet de piloter un avion de ligne. Difficile d'imaginer que le pilote va faire un reboot sur son PC. C'est ce qu'on appelle des logiciels 0 défaut. Mais comment cela peut-il exister ?

Le développement repose sur X principes

1. Une architecture système béton. Et qui ne fera ensuite l'objet d'aucun compromis..
2. Des langages de spécifications de haut niveau, quasiment du langage humain, qui vont permettre de vérifier que toute l'application est spécifiée, qu'il n'existe aucune contradiction entre les spécifications, et qui va permettre de générés tous les cas de tests. Ces langages de haut niveau réduisent de facto la quantité de code produit par des humains : or on sait que le nombre d'erreurs est proportionnel à la quantité de code.
3. Des erreurs traitées localement. Chaque fois que le logiciel va détecter ou recevoir une erreur, il va tenter de la corriger par lui-même. S'il la remonte au

niveau supérieur, c'est après l'avoir clairement documentée, y compris les actions entreprises à son niveau. Au niveau le plus haut l'erreur doit systématiquement être traitée, ou au minimum le système doit se mettre dans un comportement dégradé mais sécurisé pour l'utilisateur.

4. Toutes les erreurs détectées sont systématiquement enregistrées à des fins de maintenance statistique corrective et/ou préventive.

Bref des principes qui sont à la base des logiciels complexes. Et pour lesquelles je vous ai bassinés dans le cadre du projet DELIRE. Sachez-le, la mise au point de logiciels 0 défaut coûte la peau des fesses.