Le projet **DELIRE**

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

PC1 – L'exécution des tests

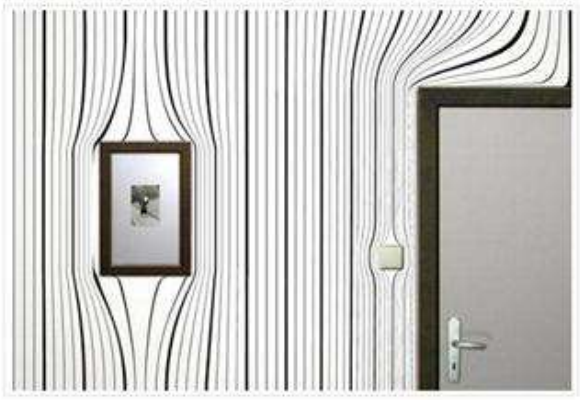
Optimiser l'organisation présente et future
La convergence d'un logiciel, basé sur l'exécution de tests, est un processus chronophage. Dans les grandes entreprises éditrices de logiciels, comme Microsoft par exemple, le nombre de personne impliquées dans les tests est le même que le nombre de développeurs de code. Et l'augmentation de la taille et de la complexité des programmes développés conduit à faire croitre encore plus le pourcentage de testeurs. Toute opération permettant de réduire le coût de la facture du processus sans réduire son efficience est la bienvenue.

L'amélioration du processus repose sur 3 piliers

1. Le travail d'architecture
2. L'ordre d'exécution des tests
3. L'automatisation, autant que faire se peut, de l'exécution des tests

Dans l'édification d'une maison, il ne viendrait à personne l'idée de:

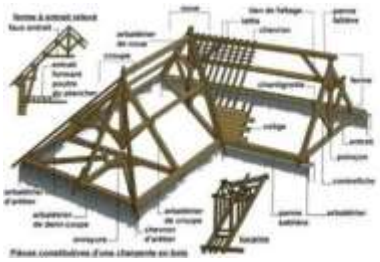
1. Poser le papier peint avant que le plâtre ne soit sec

2. Gâcher le plâtre avant d'avoir posé le toit et mis la maison hors d'eau

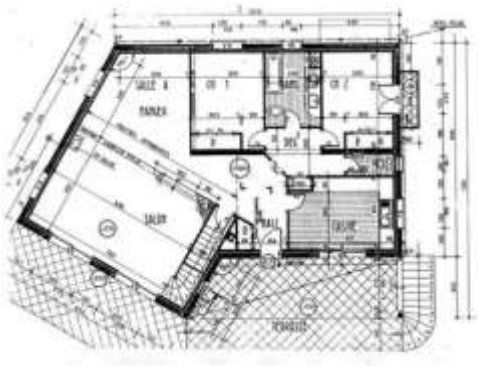


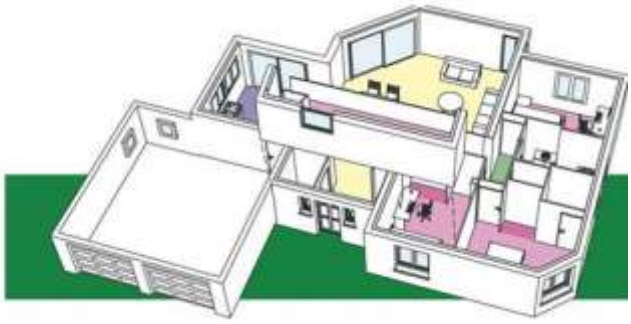
3. Installer la charpente avant d'avoir construit les murs porteurs



4. Et ne pas commencer par les fondations

5. Mais les fondations présupposent un plan d'architecture

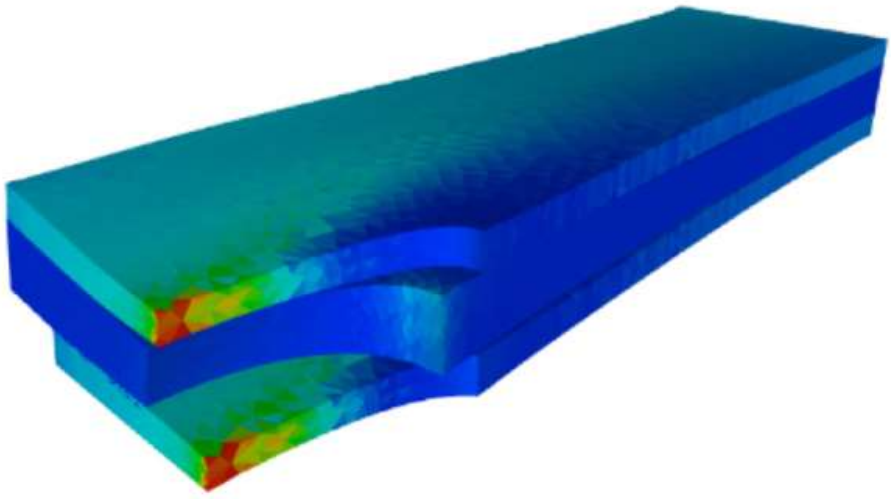




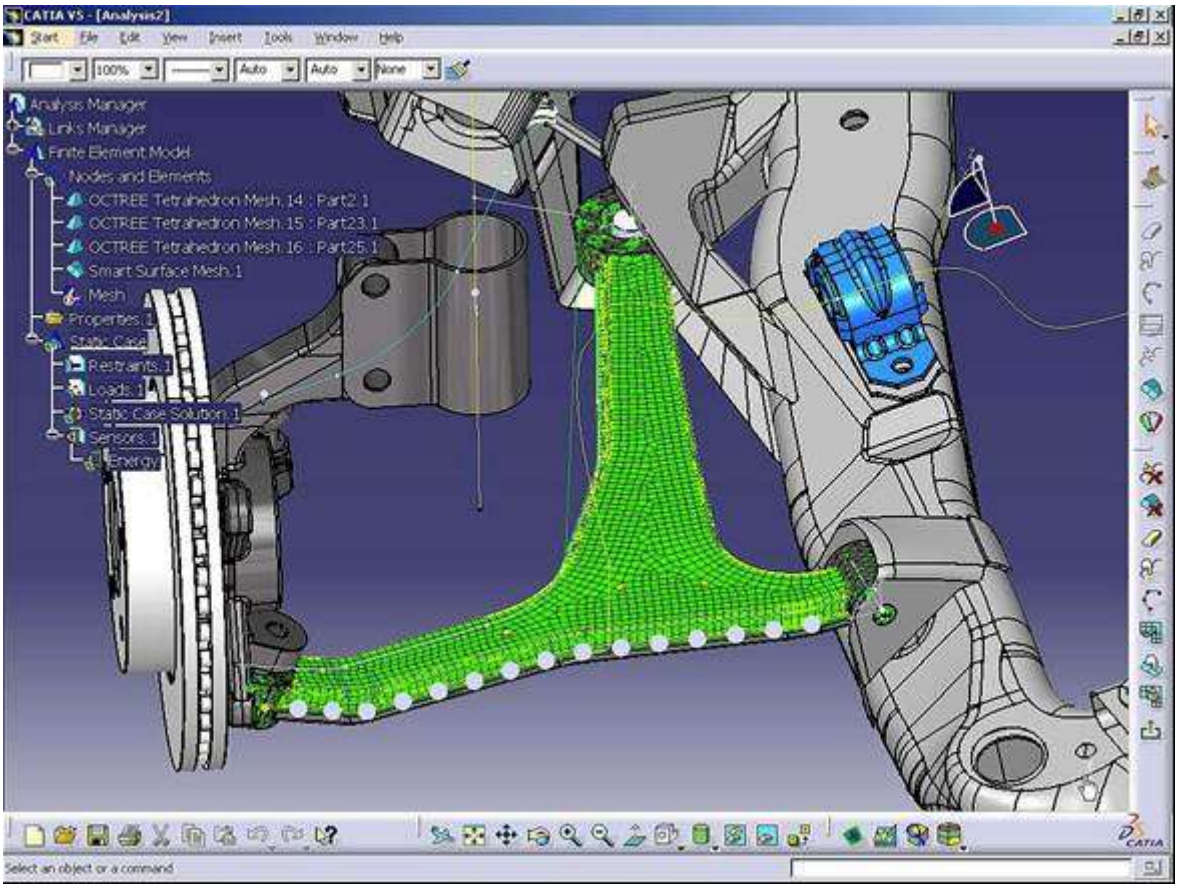
6. Lequel s'appuie sur un objectif, un budget et un planning

Il en est de même dans tout projet

1. On commence par les tests sur le pièces unitaires



2. On continue par les tests de petits assemblages



3. On continue par les tests fonctionnels du produit : le fonctionnement en mode nominal

1. On finit par les stress tests : que se passe-t-il aux limites du produit

Test d'intégration

Nous allons faire l'hypothèse que les unit tests sont terminés, et ont livré toutes les erreurs dont ils étaient capables. Il pourrait être tentant de démarrer les tests fonctionnels : en d'autres termes, de valider que le produit une fois assemblé est conforme aux spécifications. C'est un peu tôt. Durant le travail d'architecture l'accent a été mis sur les interfaces des différents modules : les services accessibles par les autres composants. Ensuite, le travail a pu être distribué parce que le fait d'avoir figé les interfaces permettait d'offrir un espace isolé à chacun. Au moment où on assemble les différents composants, il est important de vérifier que le contrat que définissaient les interfaces a été strictement respecté. Et ce pour deux raisons :

1. certains développeurs ont pu se tromper dans leur implémentation
2. il est possible que le travail d'implémentation a pu conduire certains développeurs à faire évoluer leur interface, et certains composants sont basés sur des versions obsolètes de cet interface.

Le rôle des tests d'intégration est de valider que l'ensemble des composants, qui ont été validés de façon individuelle par chacun des développeurs responsables de leur écriture, communiquent bien entre eux et qu'il n'y a donc pas de problème d'interface. Mais question, comment réaliser cette vérification

Il pourrait être tentant de se lancer directement dans des tests fonctionnels. Cela permettrait de faire d'une pierre deux coups :

1. D'une part de vérifier que les communications se passent bien entre les composants
2. D'autre part de valider que les algorithmes codés au sein de ces mêmes composants ont corrects.

Certes, c'est vrai si les tests se révèlent positifs. Mais partir avec cette hypothèse est oublier que le fait de faire des erreurs dans de la programmation, comme dans toute activité d'ingénierie, est normal. Et donc si, comme c'est probable, les résultats des tests sont foireux, il deviendra difficile de séparer le bon grain de l'ivresse, c'est à dire de savoir si c'est un problème de communication ou d'algorithme.

La bonne nouvelle c'est que pour valider leur composant les développeurs ont dû (ou aurait dû) écrire des tests unitaires. Ces tests (en dehors de ceux qui concernent les composants d'infrastructure basiques) sollicitent nécessairement des services d'autres composants. Pour les tests unitaires, les développeurs ont dû (ou auraient dû) simuler les composants sollicités.

L'idée majeure des tests d'intégration est de récupérer l'ensemble des tests unitaires et de les exécuter sur le produit assemblé. L'objectif des tests d'intégration n'est pas de vérifier la justesse du résultat mais de valider qu'un test s'exécute du début à la fin sans problème. A priori, lorsque le test unitaire se déroulait bien, il terminait par un RC0 (Return Code 0). Ce sont ces RC0 que nous allons exploiter pour valider que les tests d'intégration s'exécute du début à la fin sans problème.

Il y a bien entendu deux hypothèses dans ce qui vient d'être écrit :

1. d'une part le fait que la façon standard de valider qu'un test s'est bien déroulé a fait l'objet d'un consensus de la part de l'équipe (RC0). Ceci est en général défini dans les standards du PQ (Plan Qualité) partie process.
2. D'autre part, que le résultat du test unitaire n'était pas complètement hardcodé. Si dans votre test unitaire vous avez mis en dur dans la simulation d'un composant que le rayon était 1. et si le tests se révèle négatif dans le cas où la valeur calculée du diamètre n'est pas 2., alors il y a un petit souci pour utiliser votre test unitaire en tests d'intégration ; il faudra légèrement le remanier

Dans le cas où le test d'intégration se passe bien, tout se passe bien au sein de l'équipe.
Dans le cas où le test ne se passe pas bien, c'est là où un processus formel à du bon. Par principe, si un test foire, c'est toujours de la faute de l'autre. Très honnêtement ce n'est pas avec une démarche de ce type que l'on fait avancer le produit. L'idée est de d'avoir en tête de maximiser la productivité de l'équipe, et de ne surtout pas chercher à juger ou à accuser.

Si le test ne se déroule pas jusqu'au bout c'est qu'en général il est parti au carton avant. La règle que je vous propose est que c'est le propriétaire du composant qui s'est cartonné qui fait la première analyse du problème.

Deux cas peuvent se produire :

1. Le carton se passe au sein du composant
2. Le carton se passe à la frontière de deux composants

Le carton se passe au sein d'un des algorithmes du composant. C'est là où la mise au point d'un mode trace bien architecturé va se révéler profitable. A priori, ce cas aurait du être détecté par les tests unitaires, mais ceci n'est pas important. Une fois l'erreur identifié, le développeur va corriger son code, écrire un nouveau test unitaire

qui va valider la correction, et relivrer à la fois son code et son nouveau test unitaire qui va enrichir les tests d'intégration.
Qui dit modification de code dit nécessairement possibilité d'introduction de nouvelles erreurs. Et donc, avant de faire sa relivraison, le développeur devra valider son composant en exécutant dans son environnement perso l'ensemble de ses tests unitaires, en simulant les autres composants.

Le carton se passe à la frontière de deux composants, lorsque le composant CmpA sollicite un service du composant CmpB. Il faut valider si le carton a lieu à l'appel à l'appel du service de CmpB ou à son retour.
C'est là où en général avoir mis en place lors de l'écriture du code un second mode de trace va se révéler particulièrement judicieux.
Ce mode consiste à :

1. écrire avant chaque appel à un service externe le nom du service et la liste des arguments envoyés (CmpA)
2. écrire après chaque appel à un service externe le nom du service et la liste des arguments renvoyés (CmpA)
3. écrire après chaque entrée dans un service externe le nom du service et la liste des arguments envoyés (CmpB)
4. écrire avant chaque retour d'un service externe le nom du service et la liste des arguments renvoyés (CmpB)

En toute logique la lecture des dernière lignes du fichier trace devrait mettre en évidence la source du carton.
Là aussi, le fait de s'être mis d'accord au sein de l'équipe sur la façon standard d'activer ce mode trace inter composants sera un plus apprécié.

Faisons l'hypothèse ici qu'une erreur d'interface :

1. erreur sur le nom du service appelé
2. erreur sur le nombre d'arguments
3. erreur sur la typologie des arguments

explique le carton.
Là encore, il ne sert à rien de juger ou d'accuser. Ayez toujours en mémoire que le fait de faire des erreurs dans de la programmation est naturel. Le seul objectif est de les trouver et de les corriger avant la mise du produit en production.
A première vue, il semblerait que le développeur responsable du nom respect de l'interface telle qu'elle a été définie dans l'architecture logique du produit devrait avoir la charge de corriger son code.
Mais il peut se révéler beaucoup moins coûteux que ce soit l'autre composant qui s'adapte à la nouvelle interface. C'est juste un problème d'efficacité. Là aussi, inutile de râler et de faire sa mauvaise tête. C'est l'efficience et l'efficacité du groupe qui doit vous guider.

Même motif, même punition : le développeur va modifier son code, modifier éventuellement ses tests unitaires, et relivrer à la fois son code et ses tests unitaires pour les tests d'intégration.
Qui dit modification de code dit nécessairement possibilité d'introduction de nouvelles erreurs. Et donc, avant de faire sa relivraison, le développeur devra valider son composant en exécutant dans son environnement perso l'ensemble de ses tests unitaires, en simulant les autres composants : il aura bien entendu été modifier si besoin le code de simulation du composant appelé.

Les tests d'intégration sont terminés quand la boucle tests unitaire → tests d'intégration s'exécute end to end sans problème
Si le système est bien gaulé, réexécuter cette boucle est quasi instantané : les tests se faisant en automatique et étant capable de reconnaître eux-mêmes qu'ils se sont bien déroulés fait que l'opération est une simple opération presse-bouton.

Les tests fonctionnels
Les tests fonctionnels ont pour objectif de valider que le produit répond aux spécifications fonctionnelles élaborées en début de projet.
Les tests fonctionnels vont reposer sur deux types de tests fonctionnels

1. Des tests relativement simples que l'on pourra automatiser : par exemple, si le rayon vaut R, le diamètre doit valoir 2*R
2. Des scénarios qui devront être effectués de façon manuels

Les tests automatiques permettront d'éliminer des premières erreurs qui rendront ensuite les tests manuels plus rapides à exécuter.

Ces tests manuels sont gradués en difficulté et en temps d'exécution

1. Des scénarios de commandes unitaires : création d'un objet, effacement d'un objet, recherche et affichage de tous les objets d'un type données... Ces scénarios couvrent tous les rôles définis dans le produit (administrateur, chef de produit, engineer, reviewer...)
2. Des scénarios de processus élémentaires, faisant appel à plusieurs commandes enchainés
3. Des scénarios de processus métier complet.

Par défaut, quand une erreur est identifiée par un test fonctionnel, c'est toujours le responsable de la commande en cours qui doit démarrer l'analyse de la cause de l'erreur.

Même motif, même punition : le développeur va modifier son code, modifier éventuellement ses tests unitaires, réexécuter ses tests unitaires dans son environnement perso puis relivrer à la fois son code et ses tests unitaires pour les tests d'intégration.
Et avant de mettre à disposition la nouvelle version de code pour valider que la correction effectuée résout le problème identifié puis continuer les tests fonctionnels, on ré-exécute l'ensemble des tests d'intégration. Vous l'avez compris, le but est de maximiser tout ce qui peut être fait de façon automatique : un ordinateur ça travaille vite et ce n'est pas payé (vous allez me dire vous non plus ; mais je vous rappelle que l'unité de compte de notre budget ne s'exprime pas en $ mais en heure-ingénieur).
Bien, entendu, si vous avez mis au point des tests fonctionnels automatiques, vous les rejouer maintenant.

Question Antoine, une fois la nouvelle version de code mis à disposition avec l'incident corrigé, comment poursuit-on les tests fonctionnels :

1. Revérifions nous tous les tests précédemment exécutés et validées pour être sur de ne pas avoir de régression ?
2. Ou reprenons-nous le ou les tests à l'endroit où ils ont été interrompus par le problème identifié ?

Personnellement, et cela n'engage que moi, ma recommandation est de tenter de dérouler les tests fonctionnels jusqu'au bout et donc de les avoir réussis partie par partie. Une fois l'ensemble des tests déroulés par morceaux et tous les problèmes corrigés, on reprendra l'exécution de l'ensemble des tests fonctionnels pour les exécuter tous sans la moindre interruption ou problème identifié.

Les tests fonctionnels sont terminés quand la boucle [tests unitaire → tests d'intégration → tests fonctionnels automatiques → tests fonctionnels manuels] s'exécute end to end sans problème.
Bien entendu, je fais encore ici l'hypothèse que les scénarios des tests fonctionnels représentent une couverture fonctionnelle totale de l'utilisation du produit en mode nominal pour l'ensemble des rôles utilisateurs.

Le processus de correction des erreurs
Question : lorsqu'on trouve une erreur, doit-on s'arrêter immédiatement et attendre la disponibilité de la correction pour reprendre les tests ou continue-t-on les tests ? Ma recommandation est de continuer à dérouler les tests de façon à accumuler les erreurs : cela permettra à chaque responsable de composants de corriger simultanément plusieurs erreurs et de ne ré-exécuter les batteries de tests (tests unitaires, tests d'intégration) qu'une seule fois. Mais ceci a néanmoins une limite, on risque en effet de buter régulièrement sur la même erreur présente sous différentes formes à différents endroits du même scénario fonctionnel ou dans des scénarios fonctionnels différents. Là encore ce qui vous guide est la productivité de l'équipe. Je vous conseille de définir un niveau où de toute façon on bascule dans un processus de correction. Ce peut être :

1. Un certain nombre d'erreurs ouvertes et non corrigées : 25 par exemple, pour l'ensemble du groupe
2. Un nombre d'erreur maximum sur un composant donné : 5 par exemple.

Ce nombre pourra d'ailleurs évoluer (dans la réalité, diminuer) au fur et à mesure de la progression des tests fonctionnels.

Lorsqu'un développeur va devoir analyser une erreur, il est préférable qu'il ait une bonne description du scénario : c'est pourquoi le soin apporté à la rédaction de scénarios se révèle important. Une erreur est trouvée soit dans une commande interactive, soit dans un programme batch. Par défaut, c'est le responsable du composant de dialogue de la commande interactive ou du main du programme batch qui doit commencer l'analyse de l'erreur. S'il apparaît que cette erreur n'est pas localisée dans son composant mais qu'elle provient par exemple du retour d'un appel du service S du composant CmpB, alors il la transmet à la personne responsable du composant CmpB en lui fournissant toutes les informations récupérées durant sa phase d'analyse. C'est au responsable du composant CmpB de vérifier si l'erreur se situe dans son propre composant ou si elle provient du retour d'un appel du service S' du composant CmpC, auquel cas il la transmet à la personne responsable du composant CmpC en lui fournissant toutes les informations récupérées durant sa propre phase d'analyse.
Lorsque la personne responsable de l'erreur identifiée a terminé sa correction et après avoir revalidé ses tests unitaires, elle peut éventuellement vérifier la correction en se construisant un environnement constitué des composants officielle de la version en ayant surchargé son propre composant par cette nouvelle version. On ne met en place ce genre de vérification que si c'est facile, c'est à dire si on n'est pas

obligé de reconstruire tout un environnement spécifique chez soi. Sinon il vaut mieux relivrer son code en faisant l'hypothèse que sa correction est la bonne. C'est en particulier le cas lorsqu'une correction affecte deux (ou plus) composants : en ca cas on vérifiera que les deux livraisons sont synchronisées.

OK, Antoine, mais maintenant comment on corrige les erreurs ?
S'il s'agit d'une erreur d'algorithme ou de respect d'interface, nous venons de le décrire
Mais le test peut être bloqué pour une raison simple : il manque dans le produit une fonctionnalité pour exécuter le scénario jusqu'au bout.
Je vais encore faire mon gros lourd, mais en toute logique les scénarios des tests fonctionnels ont dû être écrits avant la finalisation des spécifications fonctionnelles et validés à la main au cours de cette phase.
Merci, Antoine, pour ce rappel mais ce n'est pas ce qui va pouvoir nous faire avancer.

En gardant à l'esprit le fait que l'objectif est de sortir le produit en temps et en heures, trois cas peuvent se présenter :

1. Le manque identifié remet en cause une fonctionnalité mineure du produit : on va regarder si la correction est simple, sinon on n'hésitera pas à abandonner cette fonctionnalité pour la présente version, et on tentera de la finaliser pour la version suivante.
2. Le manque identifié met une fonctionnalité majeure en défaut, mais la correction est relativement simple. On corrige.
3. Le manque identifié met une fonctionnalité majeure en défaut, et la correction est tout sauf simple. On peut certes allumer un cierge à Notre Dame du Logiciel, mais cela a rarement donné des résultats tangibles. On va analyser avec calme et sérénité la situation. Là non plus, pas d'affolement, pas d'accusation ou pas de jugement : c'est l'ensemble du groupe qui est pénalisé, c'est en groupe qu'on va résoudre le problème.

C'est quoi une correction tout sauf simple ? C'est :

1. Soit un développement majeur au sein d'un composant
2. Soit un développement qui remet en cause l'architecture du produit.

Un développement majeur au sein du composant.
On repart dans un processus standard de développement, mais désormais la priorité va être le temps de développement : on n'hésitera pas si besoin à faire un développement de bourrin, quitte à reprendre ce nouveau développement dans la prochaine version du produit. J'appelle développement bourrin un développement qui sacrifie la qualité du résultat (performance, convivialité) à l'efficacité de programmation. Bien entendu, cela a ses limites, et il n'est pas envisageable de mettre sur le marché un produit avec des performances désastreuses.

1. Avoir un temps de réponse de l'ordre d'une minute quand une seconde serait nécessaire est tolérable ; passer à une heure ne l'est plus.
2. Demander à l'utilisateur de respécifier une valeur de temps en temps est acceptable ; lui faire rentrer la même valeur toute les 10 secondes ne l'est plus

Si ce développement a en plus le bon goût de remettre en cause l'architecture du produit, on se réfère au paragraphe immédiatement suivant.

Si le temps de développement de cette nouvelle fonctionnalité se révèle prohibitif, on se réfère au chapitre suivant le chapitre immédiatement suivant

Un développement remet en cause l'architecture du produit.
Là ce n'est plus un cierge effilé mais un cierge pascal qui va devoir être apporté à Notre Dame du Logiciel. Mais comme le résultat se fera attendre, on va donc commencer par réfléchir.
Modifier l'architecture signifie devoir solliciter pour la mise au point de ce nouveau développement dans le composant CmpA un service extérieur S qui ne faisait pas partie des services préalablement accessibles en externe. Et de plus ce service S peut être déjà disponible ou non (je dis que ce service qui n'est pas disponible est un service externe au sens où il n'a que faire dans le composant CmpA mais où il trouve logiquement sa place dans un autre composant CmpC : un composant est fortement cohérent et faiblement couplé).
Si le service est d'ores et déjà disponible, on se pose la question de la modification d'architecture

1. 1er cas : le service est présent dans un composant CmpB déjà référencé par le composant CmpA ou par un composant CmpC déjà référencé par un composant CmpB qui lui-même est référencé par le composant CmpA (et le raisonnement est récursif). En ce cas, on va modifier la carte d'identité du composant CmpA pour rajouter le service S dans les services accessibles du composant CmpB, ou pour rajouter le composant CmpC et son service S.
2. 2nd cas : le service est présent dans un composant CmpC qui ne fait pas partie des composants référencés par cascade par le composant CmpA. On analyse si il est préférable de modifier l'architecture (avec des risques de créer des boucles ingérables, ou s'il vaut mieux recopier au sein du composant CmpA le service du composant CmpC.
3. 3ème cas : le service S non encore existant trouve logiquement sa place dans un composant CmpB déjà référencé par le composant CmpA ou par un composant CmpC déjà référencé par un composant CmpB qui lui-même est référencé par le composant CmpA (et le raisonnement est récursif). En ce cas, on va développer le service S dans le bon composant puis on va modifier la carte d'identité du composant CmpA pour rajouter le service S dans les services accessibles du composant CmpB, ou pour rajouter le composant CmpC et son service S.
4. 4ème cas : le service est présent dans un composant CmpC qui ne fait pas partie des composants référencés par cascade par le composant CmpA. Ma recommandation est de développer ce service au sein du composant CmpA et de renvoyer à la version suivante le transfert du service S dans le bon composant CmpC, en ré-architecturant le produit. Je fais l'hypothèse que le besoin de ce service ne se révèlera pas ensuite dans le composant CmpI puis dans le composant CmpJ et ainsi en cascade.

Si vous êtes amenés à développer un nouveau service dans un composant CmpA, il est évident (mais cela va encore mieux en le disant) que ce service doit faire l'objet d'un ou de tests unitaires du composant CmpA, et que ces tests une fois convergés doivent être promus pour enrichir le spectre des tests d'intégration
Si vous êtes amenés à développer temporairement un nouveau service dans un composant CmpA alors que sa place devrait être logiquement dans le composant CmpC, il est évident (mais cela va encore mieux en le disant) que ce service ne devra pas faire partie des services publics (c'est à dire accessibles depuis les autres

composants) du composant CmpA. Bien entendu ce service doit faire l'objet d'un ou de tests unitaires du Composant CmpA, et que ces tests une fois convergés doivent être promus pour enrichir le spectre des tests d'intégration. Lors du transfert de ce service vers le composant CmpC, il sera judicieux de transférer également le test unitaire, qui devra potentiellement être adapté pour son nouveau composant propriétaire.
Dans les deux cas, l'approche « développement bourrin » est à privilégier.
Pour votre information, dans l'industrie et en particulier les gros programmes qui se font en multi entreprises, lorsqu'une entreprise est amenée à demander la modification d'une interface une fois que celle-ci a été figée (c'est à dire en fin de phase de preliminary design qui correspond à la fin des spécifications fonctionnelles et de l'architecture logique du produit) elle a pour obligation de payer tous les impacts induits dans les autres entreprises du programme. Croyez-moi, cela limite les velléités, et poussent les équipes à bétonner l'architecture. Air connu...

Mais si malgré tout le développement du service ne tient plus dans le temps imparti ? Dans un projet logiciel standard, où il est important de livrer un produit complet fonctionnellement, la seule solution consiste à négocier une augmentation de budget et un décalage de la remise du produit.
Dans le cas du produit DELIRE, vous n'en avez pas la possibilité. Il ne vous reste plus qu'à mettre vos regret dans votre poche, votre mouchoir par-dessus après avoir essuyer vos larmes de dépit et à passer à la phase suivante : l'analyse du pourquoi vous vous êtes retrouvés dans cette situation. Le problème n'est pas d'avoir fait une erreur (je rappelle que cela fait partie intégrante du développement logiciel, mais d'en tirer les leçons qui vous permettront de ne pas refaire la même erreur plus tard. Ne l'oubliez pas, dans DELIRE, l'important n'est ni le but ni le chemin mais le cheminement. Et pour notre démonstration : vous essayerez de démontrer ce qui est possible ; et ceci peut se faire par tous les moyens comme par exemple de mettre en dur des valeurs dans la base de données. Ensuite, lors de votre démonstration vous expliquerez là où vous avez un trou, et reviendrez dessus lors de la phase d'analyse du projet.

Quelles erreurs rechercher dans les tests fonctionnels ?
L'objectif ici est de livrer un produit pour un client donné. Votre mission (et vous l'avez acceptée), est de réaliser un produit conforme aux spécifications.

Il y a deux sortes de spécifications :

1. Les spécifications fonctionnelles : elles ont été décrites dans le document des SFG
2. Les spécifications non fonctionnelles : principalement performances et convivialité. Elles sont décrites dans le PQ (Plan Qualité)

Vous l'avez constaté, je ne parle pas ici de fiabilité : la taille et le délai du projet DELIRE ne permettent pas de mettre en place une véritable approche de fiabilité basée sur un MTBF. De mon point de vue, si vous avez exécuté les tests unitaires puis les tests d'intégration puis les tests fonctionnels sans erreur, et si vous avez été capable de rejouer deux ou trois fois la démonstration sans problème, alors vous pouvez considérer que votre produit a un niveau de fiabilité honorable.

Vérifier les spécifications fonctionnelles est basique : on déroule les tests fonctionnels, en faisant l'hypothèse que ces tests définissent une couverture fonctionnelle complète de l'utilisation du produit en mode nominal.

Vérifier les performances est lui aussi basique : cela se fait à la montre et au feeling (le produit ne colle pas). Et on s'assure que ces tests sont faits sur une machine où tout le code se trouve en local pour ne pas avoir à supporter la latence du réseau.

Les tests de convivialité se décomposent en :

1. Vérifier les spécifications telles qu'annoncées dans le PQ : il s'agit par exemple du respect de certains standard ou du nombre d'interactions nécessaires pour exécuter une commande.
2. Vérifier le respect des standards de présentation tels qu'annoncés dans l'architecture fonctionnelle (la maquette du produit) : couleur ou police de caractère, définition des menus et des fenêtres, enchaînement des commandes...
3. Éliminer toutes les scories qui rendent le produit peu agréable : fautes d'orthographe dans les textes affichés, mauvais alignement des fenêtres, texte qui dégueule du champ graphique, ascenseur qui ne fonctionne pas...

Bref, il faut que votre produit soit agréable à prendre en main.

Mais avec ce que nous venons d'énoncer, vous ne validez que l'utilisation nominal d'un seul utilisateur du produit. Je rappelle que votre produit doit permettre de faire travailler une pléthore d'engineers en architecture distribuée et avec un bon niveau de service en termes de sécurité. En d'autres termes, valider que la base de données est accessible de façon simultanée par plusieurs machines sur le réseau, et vérifier que deux (ou plus) personnes ne peuvent pas simultanément modifier la même donnée fait partie des tests qui doivent nécessairement être exécutés (et démontrés lors de la soutenance finale).

Vous l'avez constaté, j'ai plusieurs fois insisté sur le fait que les tests fonctionnels ne couvraient que l'utilisation nominale du produit. L'utilisation du produit en dehors de ce cadre fait partie des stress tests

Les stress tests
Bien entendu, si tout le monde utilisait votre produit tel qu'il doit être utilisé, ce serait merveilleux. Mais nous ne vivons pas dans un milieu de Bisounours. Régulièrement les utilisateurs se trompent dans l'ordre des commandes, entrent des paramètres erronés ou travaillent sur une base de données non encore initialisée.

L'objectif des stress tests est de garantir que le produit se conduit de façon convenable lorsque les utilisateurs font une erreur de manip. J'entends par convenable :

1. pas de départ en sucette
2. pas de corruption de données
3. un message clair pour expliquer pourquoi la séquence est invalide.

Identifier les stress tests nécessite de partir des scénarios et d'imaginer à chaque interaction ce qui pourrait être exécuté par erreur.
Je ne peux que vous donner des pistes de réflexion

1. Tester systématiquement le undo quand il est proposé même s'il n'a aucune raison d'être
2. Entrer des chaînes blanches dans les champs de saisie
3. Modifier de façon incohérente les données dans la base de données.

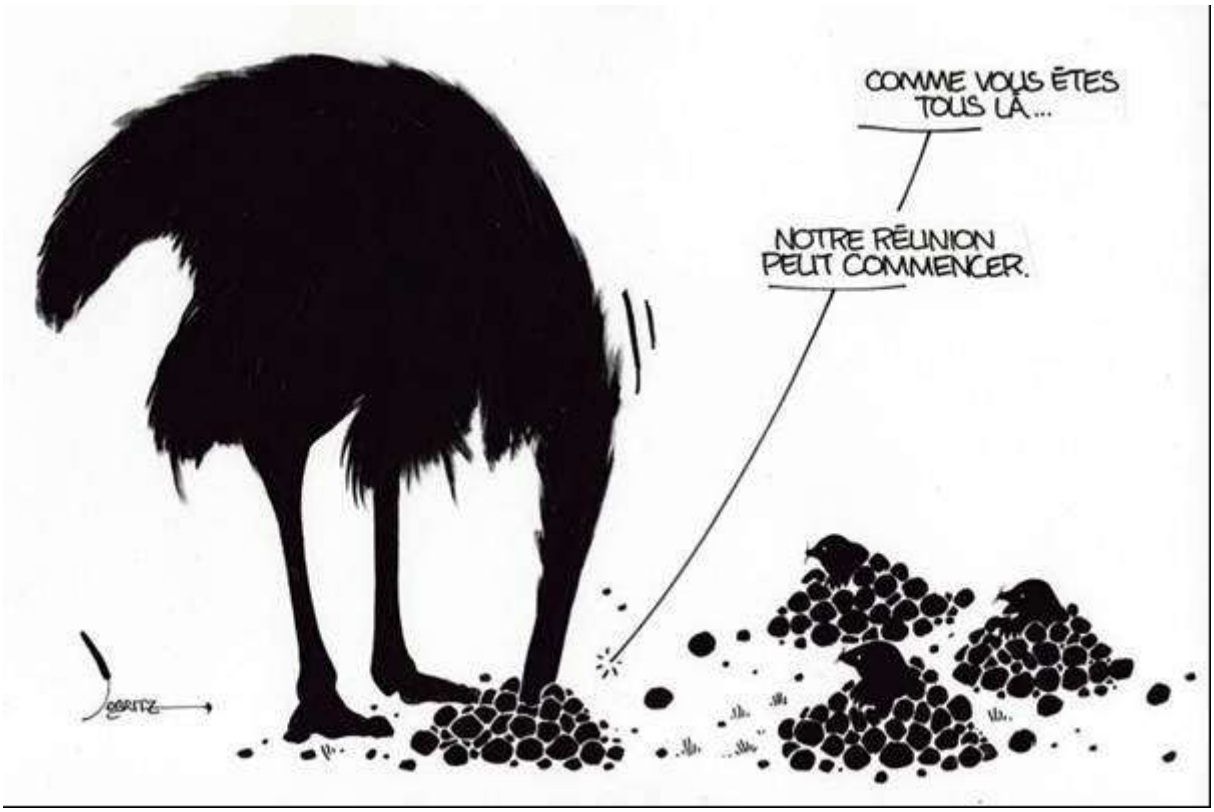
Plus un conseil : si vous savez rendre automatiques les stress tests c'est uen bonne approche.
Valider que le produit ne part pas en sucette, ou que les messages d'erreurs sont clairs est relativement simple. Par contre, valider qu'il n'y a pas de corruption de données nécessite soit une commande qui teste les données en mémoire, ou un programme batch qui va scanner la base de données : question naïve, la structure des données en base vous permet-elle d'identifier une incohérence ?

Quelles erreurs corriger ?
La question a l'air a priori stupide : toutes, bien sur ! Pas si simple.
En fait la réponse va dépendre de là où on en est dans les tests de convergence
Au début des tests l'objectif est de corriger toute les erreurs
Au fur et à mesure qu'on va avancer vers la date de remise du produit au client (ou, dans le cas du projet DELIRE la date de soutenance finale) on va se concentrer sur les erreurs les plus graves.
En fin de phase de convergence, on se focalisera sur les erreurs complètement bloquantes pour la sortie du produit. Toutes les autres seront a priori corrigés dans la version suivante.

Mais alors Antoine, cela veut dire que le produit sortira avec des erreurs. Certes ! Et la bonne solution est de ne pas le cacher : au moment où le produit sort sur le marché, il est accompagné d'un program directory qui liste les problèmes connus et éventuellement des by-pass qui permettent de les contourner.

Dernier commentaire, mais pétri de bon sens, n'oublier de tout noter :

1. nombre d'erreurs trouvées
2. nombre d'erreurs corrigées
3. typologie : gravité, domaine de qualité (fonctionnalité, performance, fiabilité, convivialité...)



Les réunions de suivi, où tout le monde est présent, sont l'occasion de rappeler la nécessité de capitaliser cette information.

Le produit en mode maintenance
Il y a une vie après la sortie sur le marché

Après que le produit soit sorti sur le marché, il va y avoir des améliorations qui vont être rendues disponibles au travers de nouvelles livraisons. Globalement, ces nouvelles livraisons se décomposent en :

1. Fix Package
2. Fonctionnal Delivery

Un Fix Package (FP) a pour objectif de rassembler un certain nombre de corrections de tout genre (performance, fiabilité, convivialité) à l'exception d'évolutions fonctionnelles. A priori, un FP n'est pas accompagnées d'une évolution de la documentation utilisateur, sauf dans le cas où l'erreur portait sur une erreur de documentation, non conforme à ce qui avait été implémenté. Un FP est cumulatif, c'est à dire que le FP i+1 contient toutes les corrections du FP i, qui lui même englobait toutes celles du FP i-1. Les FP livrent non seulement toutes les corrections des incidents trouvés en clientèle, mais également des incidents trouvés lors des tests de la phase de convergence et dont on n'avait matériellement pas eu le temps de rendre disponibles les corrections avant la mise sur le marché.

Un Fonctionnal Delivery (FD), encore appelé Release, est un ensemble de corrections d'erreur et de nouveauté fonctionnelles sur le produit.
Ces évolutions fonctionnelles doivent avoir deux caractéristiques :

1. Elles ne doivent pas modifier l'interface utilisateur. En fait, pour être plus exact, si une nouvelle commande est appelée à devoir remplacer une commande préexistante, elle est livrée en parallèle de l'ancienne, et on laisse aux utilisateurs le temps nécessaires pour migrer de l'ancienne vers la nouvelle. En règle général, les éditeurs de logiciels ont d'excellentes raisons pour sortir cette nouvelle commande : plus performante, plus sécurisée, plus efficace. Et les responsables de projets on d'excellentes raisons de vouloir continuer à utiliser l'ancienne commande : pas de formation des utilisateurs à programmer, un temps d'exécution connu (ce qui a permis de dimensionner le projet, une performance et une fiabilité connues, des by-pass définis et communiqués au sein de l'équipe ; on utilisera la nouvelle commande dans le cadre du prochain projet de l'entreprise.
2. Les données sont compatibles ascendant. C'est à dire qu'on a le droit de rajouter de nouveaux objets dans la base, on a le droit de rajouter de nouvelles descriptions de données sur des objets préexistants, mais ces objets doivent pouvoir être échangés et relus avec un site qui travaille encore sur une release antérieure.

A la différence d'une nouvelle version, le FD ne permet pas de modifier le prix des produits.
Un FD est également cumulatif de l'ensemble des FD précédents et de l'ensemble des corrections disponibles (en d'autres termes, il englobe le dernier FP) ; et on redémarre les FP à vide à partir du nouveau FD. Lorsqu'un client souhaite soumettre un incident, on commence par lui demander de basculer sur la dernière release disponible et de tenter de reproduire l'incident sur le nouvel environnement.
Par contre, le FD n'est pas cumulatif de l'ensemble du produit pour des raisons de coût de téléchargement. Lorsqu'on installe un nouveau client, on commence par

installer le produit de base initial puis on l'upgrade avec le dernier FD puis éventuellement le dernier FP disponible.
Bien entendu, l'installation d'un nouvel FD, qui peut entrainer un update de la structure de la base de données, doit conserver intact le produit en fonctionnement : objets dans la base, settings d'exécution...

Même motif, même punition : le développeur en charge de la correction d'un incident à clore pour le prochain FP ou le prochain FD va modifier son code, modifier éventuellement ses tests unitaires, ré exécuter ses tests unitaires dans son environnement perso puis relivrer à la fois son code et ses tests unitaires pour les tests d'intégration.
Puis on va rejouer l'ensemble des tests d'intégration. Et les tests fonctionnels automatiques. Enfin on va rejouer les tests manuels qui concernent la ou les fonctionnalités impactées par la correction.
Et avant de clore le FP ou le FD, on va revalider de façon manuelle que tous les scénarios fonctionnels continuent à s'exécuter. La priorité des priorités dans un FP ou un FD est la non régression : on peut admettre qu'une nouvelle fonctionnalité promise ne marche pas, on peut tolérer qu'une correction annoncée ne corrige rien, mais par contre la régression (de fonctionnalité qui part au carton, ou de performance) peut mettre en péril le fonctionnement d'un projet d'entreprise. Et sachez-le, une fois un FD ou un FP installé, il n'est en général pas possible de faire machine arrière.

La correction d'une erreur est une priorité, avant même le développement de nouvelles fonctionnalités. Ceci étant dit, il est également vrai que la maintenance et le débogage ne sont pas la tasse de thé de la plupart des développeurs. Autant donc que le mécanisme soit le plus efficient et le plus efficace possible. En d'autres termes, une fois identifiée une erreur doit être corrigée le plus rapidement possible. Et il est nécessaire que la correction soit valide, que vous ne livriez pas une modification qui corrige probablement une erreur mais pas celle identifiée par le client. La qualité des échanges entre le client et le développeur est fondamentale pour être sûr que le développeur aura bien compris le problème et sera capable de le reproduire (ou de vérifier qu'il est d'ores et déjà corrigé sur le dernier FD). Cette qualité d'échange et de données transférées doit être formalisée dans une documentation de maintenance qui fait partie de la documentation du produit (et qui est décrite dans le document PC4 – La documentation associée au produit)