Le projet **DELIRE**

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

PR2 – Maintenabilité et évolutivité

Un prototype DOIT être jeté.
Il peut y avoir des points que techniquement vous ne savez pas comment traiter. Il peut donc être judicieux de faire un prototype.
Un prototype a pour unique objectif de valider une idée, un algorithme ou un protocole de communication par exemple. Il n'y a pas de critère de qualité pour le réaliser.

Mais ne tombez pas dans le piège hélas trop classique : ayant fait fonctionné le prototype, de prendre la décision de l'industrialiser. Un prototype DOIT être jeté.
Le coût d'une démarche essai/erreur pour tenter de rendre industriel un prototype se révèle toujours plus coûteux que la démarche qui consiste à le jeter à la poubelle et à redémarrer from scratch.

Mais, Antoine, mais je viens juste de trouver la DERNIERE erreur. Air connu. N'en croyez rien, c'est la dernière erreur avant la suivante. En phase de maintenance votre composant, fragile car construit avec une succession de cautères et de sparadraps se révèlera fragile et difficile à maintenir. Et la non anticipation de l'évolution de votre composant rendra difficile celle-ci.

Il faut donc conserver le principe mis en place dans le cadre du prototype et refaire une démarche standard de développement : architecture fonctionnelle, architecture logique, réalisation, convergence. Si la démarche essai/erreur s'était révélé la plus efficace, il y a longtemps qu'elle se serait imposée en génie logiciel.

Nous sommes exactement dans la même logique que recherche et innovation. La recherche a pour objectif de mettre a point une idée, avec d'ailleurs le risque que cela ne marche pas. L'innovation a pour objectif de rendre industrielle une idée mise au point dans le cadre de la recherche.

En règle générale, un prototype ne se fait pas en phase de réalisation. Il se fait parfois en parallèle durant la phase de spécification et de structuration, mais bien plus souvent avant le démarrage du projet.
En lancement d'un projet, on choisit sur étagère les techniques explorées par la recherche et on choisit celle qu'on va industrialiser dans le cadre de projet d'innovation.

On choisit en général de faire les innovations dans des domaines ou l'échec de l'innovation ne mettra pas en péril le projet. Et on a toujours sur le feu une solution de back up en cas d'échec de l'innovation. On peut se demander pourquoi on parle d'échec alors que l'idée a été mise au point dans le cadre d'un projet de recherche ? Certes, mais celui-ci n'a pas balayé tous les aspects industriels du problème, il a juste apporté la réponse à la question posée.

Ainsi, on sait depuis plus de 30 ans que les structures en composite (fibre de carbone par exemple) sont plus légères que l'aluminium pour les avions. Et pourtant ce n'est que depuis quelques années qu'on fabrique des avions commerciaux en composites. Avant cela, il a fallu :

1. Progresser dans la connaissance en RDM pour savoir comment le matériau se comportait dans le temps

2. Faire baisser le prix de fabrication : initialement le surcoût, prohibitif, n'était pas rentabilisé par l'économie de pétrole ; aujourd'hui la baisse d'obtention des composites et l'explosion du prix du kérosène rendent les composites attractifs.
3. Régler le problème de cage de faraday (naturelle en métal) pour permettre à l'avion d'être frappé par la foudre sans dégât.

Tout cela a pris de nombreuses années.

Mais que fait-on si une innovation est critique et nécessaire pour réussir un projet P. On va en général utiliser un projet peu exposé pour mettre au point l'innovation, avant le projet P. C'est ainsi, je l'ai déjà écrit, qu'Airbus a profité du projet A340-600 pour mettre au point la maquette numérique nécessaire pour le projet A380.

Oui, mais si il n'y a pas de projet pour ce faire ? To do or not to do, that's the question. Ainsi, Boeing a pris des paris insensés pour leB787, et le résultat est là pour valider ce que je dis régulièrement : pas plus de 10% d'innovation dans un projet.

Maintenabilité d'un composant

Vous passerez beaucoup plus de temps à maintenir votre composant que vous ne passerez à le développer. Il est donc judicieux de consacrer, lors de la phase de réalisation, un peu de temps à documenter et à justifier vos choix, ce temps qui vous semblera perdu alors se révèlera précieux plus tard.

Il m'est arrivé d'utiliser des astuces pour comprimer mon code. Et j'étais très fier de moi sur le moment ; malheureusement lorsque j'ai du reprendre mon code plus tard,

la logique de mes astuces était tout sauf évidente, et il en est même que je n'ai jamais retrouvé.

Quelques règles :

1. La maintenabilité, c'est d'abord un travail d'architecture.
    a. N'oubliez pas que les problèmes les plus difficiles à résoudre ne sont pas ceux de programmation mais d'architecture
    b. Vos composants doivent appartenir à une typologie, telle celle proposée par Grady Booch. Sinon, vous allez mélanger dans votre composant des structures de données, des comportements, des dialogues et vous finirez par obtenir une bouillie infâme.
    c. Pour définir vos structures de code, n'hésitez pas à vous inspirer de vos grands anciens. Les design patterns de création, de structure et de comportement du GoF (Gang of Four) sont d'excellentes sources d'inspiration. Soyez paresseux.
2. Faites simple. La qualité première d'un code n'est pas d'être astucieuse mais d'être efficace : efficace en exécution mais également en maintenance.
3. Documentez, documentez, documentez.
    a. Plus c'est complexe, plus le code doit être commenté.
    b. Bien entendu, le commentaire doit apporter une valeur ajoutée au code : dans les parties simples, ou le choix des noms de variable et la structure du code sont suffisamment explicites, ne rajoutez pas de commentaires superfétatoires ou alourdissant.
    c. Si votre algorithme est décrit dans un document externe, ne vous contentez pas de mettre dans le code un URL qui risque d'être obsolète dans quelques temps : joignez dans votre composant un document .pdf qui vous permettra dans quelques temps de retrouver la logique du composant. Je me souviens d'avoir du faire de la maintenance dans le composant d'un ingénieur qui était parti en congés : son code était totalement imbitable, et truffé de verrues qui en rendaient la lecture totalement malaisée. Après réflexion, j'ai choisi de mettre le composant à la poubelle et de le réécrire entièrement. Ce qui n'est bon ni du point de vue de la productivité ni du point de vue du management
4. Faites des codes source de taille raisonnable.
    a. Au-delà de 100 lignes, commentaires compris, le code est illisible, et vous augmentez fortement sa complexité cyclomatique.
    b. Je me souviens d'avoir passé 5 heures pour trouver un bug dans un code source de 2600 lignes, bug qui générait une corruption de données dans les modèles de Boeing, en béta tests sur CATIA V4, et qui avait engendré une crise grave. A la fin de mon analyse et après avoir trouvé l'origine du bug, j'avais un mal de crâne épouvantable et une haine profonde pour le développeur du composant qui m'avait affirmé que le problème n'était pas dans son code mais dans la façon, de travailler chez Boeing.
    c. En faisant des composants de taille raisonnable, vous augmentez la chance qu'ils soient réutilisables.

Soyez paresseux.
Le code le plus facile à maintenir est celui qu'on ne développe pas.

Avant de développer un composant, posez-vous toujours la question de savoir si ce composant n'existe pas déjà :

1. D'abord au sein de votre entreprise.
    a. En 1985, nous avons fait une enquête dans le code de CATIA et nous avons découvert qu'il y avait plus de 60 routines qui faisaient un produit vectoriel, globalement un programme par développeur.
    b. Pour mémoire je rappelle qu'à cette époque, la mémoire était limitée à 16 mégas, et que le travail d'optimisation pour permettre de charger CATIA en mémoire était assez long et complexe.
2. Ensuite, sur le marché.
    a. Si vous prenez un composant standard, vous avez toutes les chances qu'il ait subi les affronts du temps et que l'immense majorité des problèmes aient d'ors et déjà été trouvées. Vous gagnez donc en coût de développement et en coût de maintenance ensuite.
    b. Attention, si vous utilisez un composant Cmp du marché, pensez à bien faire valider par votre service juridique le fait que l'entreprise est légalement autorisée à utiliser ce composant Cmp. Et ce point est particulièrement critique dans le cas où votre entreprise est éditrice de logiciel et va donc commercialiser ses produits. Gare à la facture si le propriétaire de Cmp s'aperçoit de la situation.
    c. Attention, le fait que le composant Cmp soit dit logiciel libre ne vous autorise pas par défaut à commercialiser une application qui l'utilise

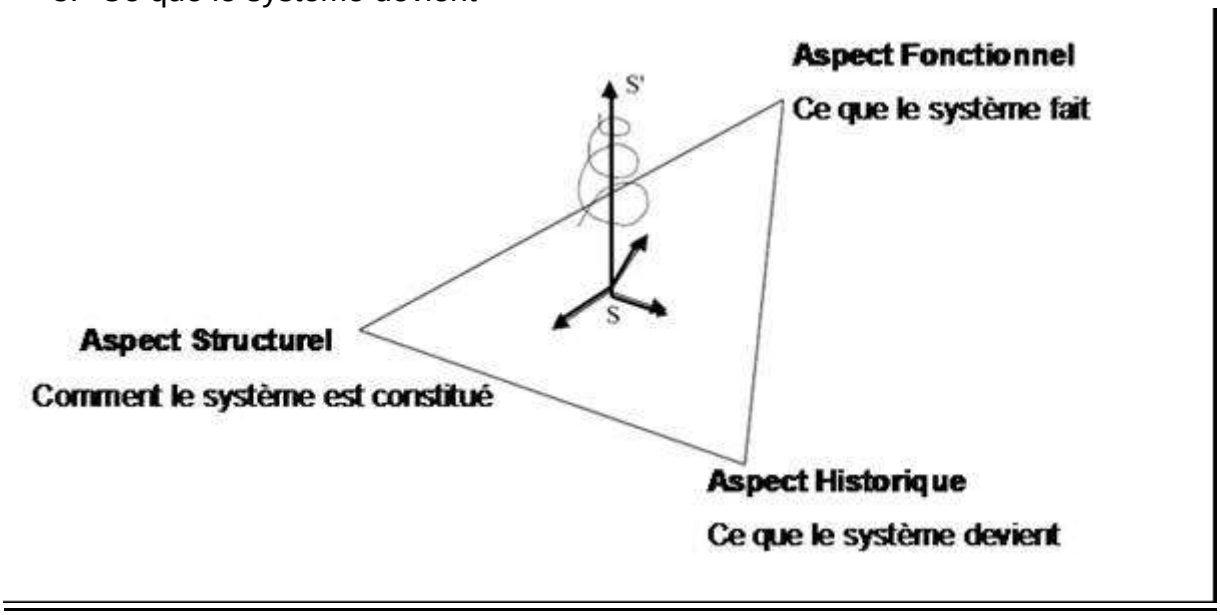
C'est d'ailleurs une tendance lourde de l'informatique que de développer le marché du composant. Ce qui est une excellente nouvelle : c'est la généralisation du concept de réutilisabilité mise en avant lors de la « crise du logiciel », dans les années 70. Et ce phénomène s'est accentué du fait d'Internet, qui permet d'offrir et de partager beaucoup plus simplement ses composants.

Mais Antoine, dans le projet DELIRE, on ne va pas aller chercher des composants existants ? Bien sur que si. A titre d'exemple, je serais surpris que, pour construire votre base de données, vous ne fassiez pas appel à Access ou un composant Linux équivalent. Au contraire, soyez paresseux.

<u>Pensez à demain</u>.
Il pourra arriver qu'un de vos collègues soit amené à maintenir un de vos composants durant vos congés : si pénétrer dans votre code est aussi pénible que rentrer dans une chambre d'adolescent, il est peu probable qu'il vous témoigne de la reconnaissance à votre retour des Bahamas.
Mais surtout, un jour vous voudrez évoluer dans votre carrière : si par malheur vos composants sont extrêmement difficiles à transférer parce que non documentés et complexes voire torturés, il y a bien des chances que votre chef de service vous bloque, peut-être pas définitivement, mais au moins suffisamment de temps pour que le poste que vous convoitiez ait été pourvu.
Et la meilleure façon de faciliter la transmission des composants est de respecter les standards de l'entreprise en matière de documentation, commentaires et coding. Le moyen le plus simple de communiquer est de partager le même langage et le même standard.

Evolutivité
Vos composants seront appelés à évoluer. Et ce d'autant plus facilement que ce point aura été anticipé.
Souvenez-vous que l'approche systémique privilégie 3 axes de travail

1. Ce que le système fait
2. Comment le système est constitué
3. Ce que le système devient



Il est donc pertinent de savoir dès le début quels sont les services qui seront rajoutés au composant dans les mois ou les années qui viennent. Et vous devez le documenter au sein du composant. Ainsi, dans le projet DELIRE, le fait de savoir qu'à la version Basic succèderont les versions Standard puis Advanced peut vous donner quelques excellentes idées non seulement d'architecture mais également de coding. Ceci étant posé, ne rêvez p
la visibilité ne dépasse pas deux ou

Think globally, act locally

La bonne approche est de faire des composants génériques.
Si vous faites un service par cas particulier, vous complexifiez inutilement votre composant, vous dupliquez vos efforts et votre code, et vous rendez votre composant difficilement maintenable.
N'oubliez pas qu'un service publié par votre composant est a priori définitif, bref difficile à supprimer par la suite.

Ne l'oubliez pas : plus vous réduisez la taille du code, plus vous factorisez celui-ci, moins cher sera la maintenance

T’es bien gentil, Antoine, mais développer un service générique prend plus de temps qu’un service spécifique, et cela peut être incompatible avec le temps dévolu pour le développer.
J’entends bien. Mais on peut peut-être couper la poire en deux : vous pouvez publier un service générique, et n’implémenter qu’un service spécifique. Avec le temps vous pourrez faire évoluer votre composant vers une totale généricité, sans impacter ceux qui l’appelaient déjà dans le cadre du scope initial. Ces évolutions se feront au fur et à mesure du temps et/ou des besoins.

Par contre, mettez bien dans les commentaires du service et dans sa documentation publique les cas qui sont supportés à ce jour : en particulier pour anticiper le fait que vous pourriez transmettre votre composant à un autre développeur. Sinon vous ou votre successeur risquez que ce service soit appelé en dehors du scope pour lequel il est garanti et de vous trouver dans une crise de maintenance non prévue et non planifiée.

Le maintien de l’architecture
Votre architecture est votre bien le plus précieux

1. Faire une belle architecture c’est difficile
2. La massacrer c’est assez facile
3. La remettre en place est globalement impossible : un service public naît, vit et meurt au sein du même composant.

Une belle architecture est caractérisée par le fait que les composants sont fortement cohérents et faiblement couplés. Qu’ils puissent permettre de distribuer le travail n’est qu’un second objectif.

Dans votre travail de développeur du composant CmpA, vous serez amené à avoir besoin d’un service S qui devrait (forte cohérence) visiblement être offert par le composant CmpB. Problème, le responsable Rb du composant CmpB est overloadé sous le travail ou la maintenance et ne pourra pas vous faire ce développement avant longtemps. Il ne vous reste plus, après avoir pleuré, qu’à faire le développement vous-même.

Essayez de convaincre Rb de développer ce service en tant que société de service, c'est-à-dire qu’une fois développé, convergé et certifié, ce service sera intégré à son composant. En règle générale, Rb va accepter votre proposition mais y imposera parfois quelques contraintes de forme : nommage des variables, organisation des structures… Acceptez : n’oubliez pas que c’est lui qui passera le plus de temps dans ce service dans le cadre de la maintenance (oui je sais, vous votre code est toujours bon du premier coup). Ceci veut dire que vous allez recopier son composant dans votre WorkSpace de développement pour faire ce développement.
La bonne nouvelle est que vous disposez de tous les services internes de son composant pour appuyer votre développement.
Attention, si vous devez faire appel dans le cadre de ce développement à un service exporté par un autre composant CmpC et si CmpC n’était pas référencé dans la carte d’identité de CmpB, vous devez absolument valider ce point avec Rb.
A la fin de l’opération, transférez lui un package complet : coding, mais aussi documentation du service (puisque celui-ci est nécessairement un service public du composant CmpB), nouvelle carte d’identité du composant, Unit tests. Mais Antoine,

cela veut dire qu'on fait du boulot pour Rb : non, vous faites un développement dans le cadre d'un projet en équipe.

Il est possible également que Rb ne souhaite pas accueillir, du moins à court terme, le développement de ce service S. En ce cas, il est nécessaire de faire ce développement dans votre propre composant CmpA, avec le risque que cela vous pourrisse la qualité de votre architecture interne : faites en sorte d'isoler le malade. Soyez conscient du fait que vous aurez à dupliquer tous les services internes du composants CmpB qui seront nécessaires à votre développement. Ceux-là aussi, mettez-les en quarantaine. Vous devrez également référencer dans votre carte d'identité tout ou partie des composants que CmpB référençait lui-même : la bonne nouvelle est le fait que le composant CmpB faisant partie de vos composants référencés, cet ajout peu glorieux de composants tirés par le vôtre ne mettra pas en cause l'architecture globales du système
Dis-moi Antoine, ne serait-il pas plus judicieux de créer un nouveau composant pour exporter ce service ? Attention danger, ce service deviendrait public et pourrait donc être référencé par un autre composant, ce qui vous conduirait à devoir le maintenir dans le temps

Un jour quelqu'un vous demandera peut être d'exporter ce service dont il a besoin lui-même. Soyez très ferme pour refuser : votre composant CmpA n'est pas le réceptacle prévu pour ce service S, et vous exigez qu'il aille dans sa localisation naturelle, id est le composant CmpB. A deux, vous devriez être plus fort pour pouvoir faire plier Rb ou son chef.
Vous pourrez alors nettoyer votre composant CmpA de cette excroissance disgracieuse : éliminer le service, les services dupliqués, les composants référencés dans la carte d'identité.
Antoine, tu nous avais dit que remettre en place une architecture était globalement impossible. C'est bien parce que vous avez gardé ce développement du service S en interne que vous avez pu faire cette opération de nettoyage. En interne, vous pouvez faire ce que vous voulez, cela n'impacte que vous. Cela n'aurait pas été possible dès lors que le service aurait été remonté au niveau de l'architecture logique du système.
.