Le projet **DELIRE**

Développement par **E**quipe

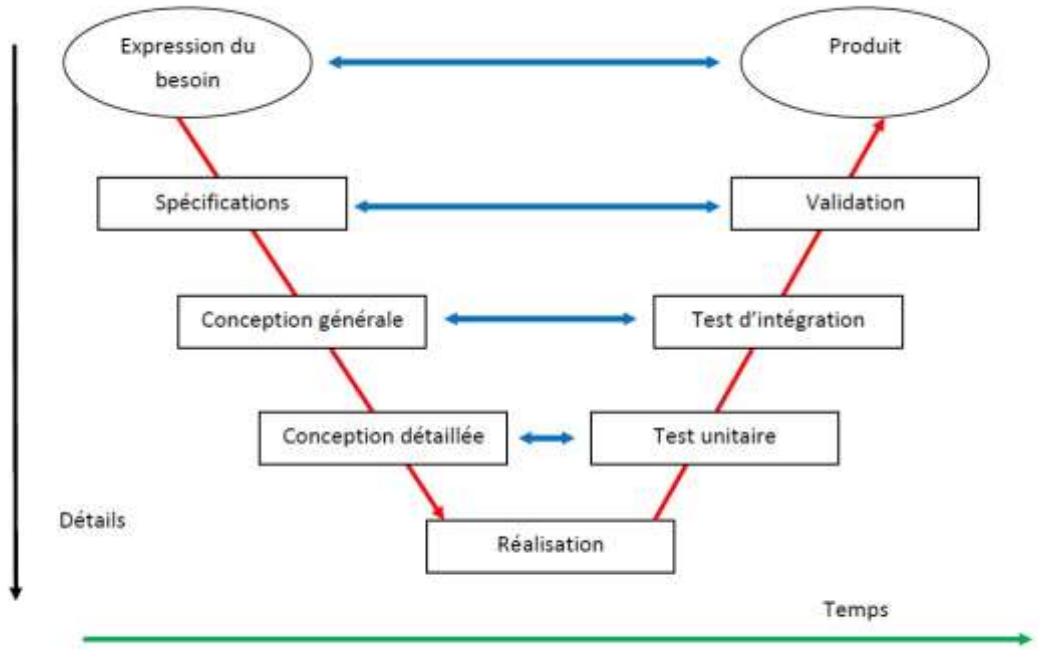
de **L**ivrables **I**nformatiques

et **R**éalisation **E**ncadrée

PStr4 – PT (Plan de Tests)

LES TESTS

Attendre la fin du coding et de l'assemblage pour commencer les tests est une approche suicidaire.
On va mélanger tous les problèmes sans savoir où commencer à déverminer.



L'exécution des tests suit une logique bottom up. C'est la base du cycle en V
On va commencer par valider le fonctionnement du coding au sein du composant, puis chaque composant en tant que structure d'encapsulation (Unit Tests), puis l'architecture des composants (Integration Tests), puis le respect des spécifications générales (Functional tests), et enfin on validera l'adéquation du produit au besoin du client au travers des tests de recette.

Par contre, la conception des tests suit une logique top-down. On commence par définir les tests de recette, puis les tests fonctionnels, puis les tests d'intégration et enfin les tests unitaires. Ma recommandation est de définir en début de phase les tests relatifs à cette phase. Ainsi, on rédigera les tests d'intégration avant d'attaquer l'architecture logique du produit : la rédaction des tests mettra en effet en évidence certains points qui eussent été oubliés sinon dans la définition de l'architecture.

Les tests unitaires de composants (écrits par les responsables de composants)

Validation du coding : les tests dits « boite blanche »
On va exécuter chaque branche de code, en faisant varier la valeur des paramètres.
On écrit donc un petit objet de test qui appellera cette sous partie du composant.
Il y a deux pièges classiques que l'on peut anticiper.

1. Le premier est de ne pas savoir comment valider le fonctionnement du code.
    a. Il est donc nécessaire d'avoir implémenter, dès la phase de coding, un mode trace qui va permettre de suivre ce déroulement.
    b. Attention, un mode trace ne signifie pas un dégueulis d'informations. Les sorties doivent être pertinentes et structurées.
    c. C'est donc un investissement lourd.
    d. Mais c'est un investissement qui sera rentable plus tard, dans les tests d'intégration ou de qualification, puis en phase de maintenance une fois le produit mis sur le marché.
2. Le second est de s'apercevoir qu'il y a des branches de code qu'on ne peut pas atteindre, ou des valeurs de paramètres qu'on ne sait pas simuler.
    a. Ceci peut généralement se détecter lorsqu'on conçoit les objets de test.
    b. D'où une règle classique : on écrit les objets de tests avant d'écrire le coding

Attention, cette démarche n'est pas naturelle. Bof, on verra bien si on a le temps pour écrire puis exécuter les objets de tests ; et on bouffe ses marges jusqu'à se retrouver coincé au final, à devoir tout tester simultanément dans une approche mortifère
Ecrire ses tests avant de réaliser le coding est une approche qui fait gagner du temps en anticipant des problèmes.
La démarche est récursive :

1. On va écrire les tests du composant avant d'en finaliser la définition publique
2. On va définir les tests d'intégration avant de figer l'architecture,
3. On va spécifier les tests fonctionnels avant de figer les spécifications fonctionnelles générales…

C'est toute la démarche du cycle en V

Validation du composant, ou tests dits « boite noire » : on vérifie que les entrées/sorties du composant fonctionnent conformément à ce qu'on en attend.
La difficulté majeure est de ne pas mélanger cette phase avec les tests d'architecture.
Quand on fait les tests de RDM (contrainte limite, fatigue, thermique…) d'une pièce mécanique, ce n'est pas dans le cadre d'un assemblage.
Il en est de même des tests du composant. Mais comme notre composant s'appuie sur d'autres composants, on est dans une situation un peu délicate.
La solution va donc s'appuyer sur un banc de test, où les autres composants seront simplement simulés : on impose les sorties. Attention, écrire un banc de test est une opération coûteuse.
A l'évidence, cette démarche n'est pas non plus naturelle. Bof, on verra bien une fois tout assemblé ; mais lorsque l'assemblage part en sucette, c'est clairement de la faute de l'autre. Parce que chacun est persuadé que son travail est excellent, et que par contre l'autre a travaillé d'une façon bizarre.

Une démarche qui va extraordinairement vous simplifier la vie consiste à tester les valeurs en entrée du composant. Et de renvoyer une erreur en cas de jeu de données incohérent. Ceci vous facilitera la tâche en phase de tests d'architecture.
Dans la réalité, ce renvoi d'erreur sera également utilisé si un comportement aberrant apparait durant l'exécution du composant.
Mais bien entendu, il est nécessaire que le composant appelant prenne en compte les erreurs renvoyées par le composant appelé.
Ceci signifie que la nomenclature des erreurs, qui font partie intrinsèque des interfaces du composant, doit être définie très tôt en phase d'architecture.

Les tests d'architecture, ou tests d'intégration, n'ont pas pour objectif de tester les fonctions, mais de s'assurer que les connexions se passent bien entre les composants : cohérence des interfaces attendues et obtenues, cohérences des valeurs d'appel, gestion des erreurs ….

Dans l'industrie, une des qualités d'un logiciel est de savoir fonctionner en mode dégradé : ce n'est pas parce qu'un des composants a retourné une erreur que le composant appelant doit systématiquement remonter une erreur : s'il est capable de retraiter cette erreur et de définir un comportement par défaut en mode non nominal, il est de son devoir de le faire.
Des logiciels ainsi créés sont dits « à tolérance de panne ». Ceci est particulièrement important dans des logiciels temps réel, tels des systèmes de pilotage d'avion.
C'est durant cette phase de tests d'intégration que l'on va tenter de passer dans tous les cas de dysfonctionnement. Sinon, ces cas apparaitront durant la phase d'exploitation, après mise sur le marché du produit. Et la résolution du problème sera onéreuse.
Ceci étant posé, il n'est pas toujours évident d'obtenir les conditions d'un cas problématique. Il est donc nécessaire de coupler ces tests avec des simulations de renvoi d'erreurs de la par de (pseudo) composants appelés dans le cadre des tests unitaires.

Dans le cas de DELIRE, on se contentera dans cette phase de valider que les appels entre composants se passent bien. On ne valide pas la pertinence des résultats.
Le système semble alors fonctionner correctement : mais les résultats qu'il renvoie peuvent être néanmoins totalement stupides.

Les tests de fonctionnalité, ou tests de vérification ou de validation, vont permettre de valider la pertinence des résultats retournés.
Si les tests de composants (boite banche ou boite noire) sont assimilables à un niveau lexicographique, et si les tests d'intégration sont comparables à un niveau syntaxique, les tests fonctionnels sont au niveau sémantique.

En règle générale, les tests fonctionnels s'exécutent en déroulant les scénarios fonctionnels. Pendant ces tests, on valide la pertinence des résultats, on valide également qu'il n'y a pas de problème dans le dialogue (texte qui sort de la boite, fautes d'orthographe, cas d'erreurs incompréhensibles, accès à la documentation, cohérence entre la documentation et le dialogue. On valide également les objectifs qualité (cohérence et standardisation de la présentation, accès simplifié aux commandes, nombre d'interactions….)

Les **stress tests** ont pour objectif de secouer le produit. Vous ne pouvez pas imaginer le nombre de comportements stupides d'un utilisateur face à un produit. Il existe des cas classiques de tests :

1. Base de données non initialisée
2. Utilisation de l'UNDO
3. Utilisation de données massives.

Dans DELIRE, on ne fera a priori pas de stress tests.

Nous allons faire l'hypothèse que les tests fonctionnels se sont bien passés, il resterait à exécuter les **tests** dits **de recette**.
On va tester non seulement les fonctions mais l'ensemble du produit, dans une approche client : temps de réponse, adéquation du produit et de sa documentation, performance globale, comportement aux cas aberrants…
L'ensemble des tests de recette ont été définis préalablement, et ont fait l'objet d'un accord entre le MOA et le MOE.
Les tests de recette sont de la responsabilité du MOA. Le MOE aurait trop beau jeu d'affirmer que ces tests se sont excellemment bien passés.
Bien entendu, le MOE aura eu l'intelligence d'exécuter préalablement les tests de recette pour ne pas risquer une catastrophe en phase de recette.

Les tests de recette peuvent révéler de mauvaises surprises : le produit ne plait pas à l'utilisateur, ne correspond pas à l'idée qu'il s'en faisait, voire ne répond pas à son attente.
Attention, on ne peut pas se mettre dans un mode émotionnel. Seul le fait que les tests se sont déroulés conformément ou non à ce qui était prévu dans la définition des tests de recette doit être pris en compte.
En cas de désaccord sur un résultat entre le MOE et le MOA, c'est le document de spécifications générales qui fait foi.
La réussite des tests de recette est importante car c'est elle qui va conditionner la mise en exploitation du produit. En cas de succès le client est dans l'obligation de payer le reliquat du, et on passe dès lors dans le mode de maintenance, incluant d'autres obligations.

Question : que ferait-on des réserves émises par le client durant les tests de recette ?
Il y a deux sortes de réserve :

1. L'identification de malfaçon : chaque point identifié fait l'objet d'un rapport d'incident qui devra être corrigé, par exemple15 jours avant la date prévue de déploiement du produit (elle a due être exprimée dans le CdC)
2. L'identification d'un nouveau manque non identifié préalablement ou sorti du scope des spécifications fonctionnelles générales. Chaque manque fera l'objet d'une demande d'évolution pour les futures versions du produit, évolution qui devra être payée par le client.

Dans le projet DELIRE, les tests fonctionnels seront réputés couvrir les tests de recette.

Cas particulier des **tests de performance**
En règle générale, on considère qu'un produit qui répond en plus d'une seconde est un produit qui « colle ». Problème, on risque de ne s'apercevoir de ceci qu'au moment des tests de recette.

On va donc tenter d'anticiper ce problème par deux approches :

1. On va mettre en place, dès la phase de spécifications générales, des objectifs de performance
2. On va, une fois l'architecture du produit définie, pouvoir tartiner cette performance sur chacun des composants. Dès lors, en phase de test de composant, on pourra vérifier grâce à l'horloge de l'ordinateur qu'on est bien en deçà du temps spécifié.

Bien entendu, cette démarche est récursive, et on va pouvoir retartiner cet objectif de performance sur chaque code source, et le vérifier en phase en tests boite blanche

Ceci signifie que dans l'écriture de vos tests, depuis les tests de recette jusqu'aux tests boite blanche, vous devriez avoir des objectifs de performance à spécifier. Qui devront être vérifiés en phase d'exécution.

Question numéro 1 : que se passe-t-il si on ne tient pas l'objectif ?
On va commencer par faire une première revue de code ou de design pour voir si un « loupé » ne serait pas à l'origine de cette dérive.
Si ce n'est pas le cas, on va tenter de récupérer des performances sur d'autres couches de logiciel.
On est dans une démarche similaire au respect du poids total dans la conception d'un avion. Ce qui est nécessairement perdu d'un coté doit être récupéré de l'autre.

Question numéro 2 : comment fait-on ce tartinage ?
En règle générale, on s'appuie sur son expérience
Dans votre cas, vous allez faire une première proposition au feeling, puis vous affinerez votre prévision au fur et à mesure du projet. ? C'est ainsi que vous vous construirez votre expérience.

Maintenant soyons honnête.
Vous allez en général vous fixer des objectifs globaux de performance.
Puis dans la panique de la réalisation, vous allez totalement oublier de suivre cet aspect. Et la veille de la soutenance finale, vous allez découvrir avec ravissement que votre logiciel est plutôt plus performant que ce que vous aviez prévu. Le jour de ladite soutenance vous nous annoncerez benoitement et très fiers que vous avez bien géré vos performances. Que nenni : vous aurez juste eu la chance que les performances vous rattrapent.
La seule chose que vous avez à faire comme tests de performances dans DELIRE est de vérifier que les temps de réponse annoncés (dans les conditions définies, tout en local et en ne prenant pas en compte la latence du réseau) sont tenus.

Une démarche similaire devrait être fait pour la consommation mémoire, mais il est acceptable de s'appuyer sur les capacités de l'Operating System à traiter ce problème dans une première approche. Sachez néanmoins que les problèmes de fuite mémoire (memory leaks) sont une plaie en software

A priori, dans votre document qualité vous avez du définir des objectifs qualité, un des objets du plan de tests est de mesurer de façon formelle l'atteinte de ces objectifs.
Ainsi, les **tests de convivialité** :

1. Au minimum, vérifiez que l'accessibilité des commandes, le nombre d'interactions ou le respect d'un standard sont au rendez-vous.
2. Si vous en avez l'opportunité, faites prendre en main le logiciel par un néophyte.

, Il y a encore de nombreux **autres tests** dans le logiciel :

1. Tests d'installation du produit (Question, avez-vous prévu une doc d'installation ?)
2. Tests de documentation (Question, avez-vous prévu une doc d'installation ?)
3. Test de NLS, National Language Support (Question, avez-vous prévu qu'ion puisse choisir sa langue d'utilisation ? C'est un choix qui se fait en phase d'architecture logique)
4. Tests de régression (sans objet dans le cas du projet DELIRE)

Nous ne nous y intéresserons pas dans la suite de ce projet. Mais nous aurons l'occasion d'y revenir lors de la phase de convergence.

Méthodes de sûreté de fonctionnement
Dans la gestion des systèmes complexes, en particulier en aéronautique, on met également en place des méthodes de sûreté de fonctionnement, qui reposent sur des critères généraux.

1. La disponibilité : période de pleine capabilité pour le démarrage d'une mission
2. La fiabilité et la maintenabilité en mission : stabilité de la capabilité durant toute la mission
3. La sécurité de mission : maintenir un niveau acceptable de risque d'accident pendant la durée de la mission
4. La survivabilité, (ou invulnérabilité) : maintenir une capabilité acceptable pendant un temps donné même dans un état dégradé.
5. La durée de vie : intervalle de temps au bout duquel la probabilité de défaillance devient inacceptable, ou lorsque le produit est considéré comme irréparable après une panne.

La sureté de fonctionnement ne sera pas prise en compte dans le projet DELIRE

ORGANISATION DES TESTS DANS DELIRE

Les Unit tests

L'écriture (puis l'exécution en phase de réalisation) des tests unitaires de composant est de la responsabilité du responsable du composant. Il s'agit de tests :

1. De la base de données
2. De l'interface utilisateur
3. Des composants de services
4. Du composant d'édition de contenu

L'objectif est de tester un composant et non l'intégration du système. C'est en général aisé pour le test de la base de données. C'est plus compliqué pour les tests de présentation, car il s'agit de simuler les autres composants, en particulier la base de données. Vous devrez écrire un pseudo composant, ayant les mêmes interfaces, et qui renvoie systématiquement les mêmes réponses.
C'est durant ces tests unitaires que vous devez valider les temps de réponse du composant vis-à-vis des objectifs de performance.

Les tests d'intégration

Ma recommandation est de balayer très rapidement les possibilités du logiciel pour mettre en évidence d'éventuels problèmes de crash, ou de blocage. Je vous conseille pour ce faire de rejouer très rapidement les scénarios fonctionnels, en ne vous souciant pas de la pertinence des entrées et des sorties.

Les tests fonctionnels

Ils s'exécutent en déroulant les scénarios fonctionnels qui ont permis de valider les SFG et l'architecture logique en phase de spécification.
On valide l'adéquation aux besoins, la pertinence des résultats, les objectifs qualité (essentiellement convivialité et performances). On valide également la documentation. Accessoirement on profite de ces tests pour identifier les éventuelles fautes d'orthographe dans les messages et les panels

Dans le projet DELIRE, on ne fera ni stress tests (en priant le ciel que les professeurs ne les exécutent pas pour vous après la soutenance) ni de tests de recette (on considérera que les tests fonctionnels les englobent).

L'objectif de la phase de spécification et structuration est que les scénarios de tests soient totalement décrits (unitaires et fonctionnels) : conditions initiales, scénario à suivre, résultats attendus, points à valider…

Les tests de performance seront basiques : on se contentera de valider que le produit ne colle pas. Pour faire ces tests, on se met dans les conditions optimales : tout le code en local, pas de latence du réseau.

En termes de **livrables**, les scénarios seront livrés en tant qu'annexes : associés à des fiches techniques pour les tests unitaires, en tant que tels pour les tests fonctionnels. Le livrable PT se contentera de présenter la philosophie qui a présidé à l'écriture de tests, et donnera la liste exhaustive des tests qui devront être réalisés.

Un ordre de grandeur sur le **temps à passer** ? Je rappelle que l'exécution des tests, l'analyse des erreurs et leur correction représente un temps équivalent au coding (de l'ordre de 20% du projet).
La tentation classique, lorsqu'on dérape (planning et budget) en phase de réalisation, est de sacrifier le temps consacré aux tests. C'est une stratégie à court terme : le coût de maintenance qui sera induit vous pénalisera pour d'autres développements. Si on dérape, on doit sacrifier des fonctionnalités. D'où l'intérêt des 3 phases (Basic, Standard et Advanced) en phase de spécification.

On peut comparer ceci à la construction d'une maison : si le budget et le planning deviennent problématiques, il ne faut surtout pas basculer sur des matériaux et des solutions bon marché, on s'en mord toujours les doigts ensuite. La solution est de retarder des fonctionnalités (une salle de bain, le garage, la peinture).
La qualité est toujours payante à terme. C'est ce qu'a compris une entreprise comme BMW.