Série Théorique Pop_s0_th

Méthodes de travail pour le développement du projet

1. Introduction : les grands principes et les critères d'évaluations

Le sujet du projet du semestre de printemps est indépendant du précédent projet. Cependant la **méthode de travail** introduite en section 2 du projet d'automne reste valide. Nous allons reprendre ces éléments et les compléter en rappelant les bonnes pratiques : mettre en œuvre les grands principes (*abstraction, ré-utilisation*), vérifier par les tests avec le *scaffolding* et les *stubs*, et bien sûr, respecter les conventions de présentation du cours.

Le but du projet de printemps est surtout de se familiariser avec d'autres grands principes, comme la *séparation des fonctionalités* (*separation of concerns*) et *l'encapsulation* qui deviennent nécessaires pour structurer un projet important en ***modules*** indépendants et fiables.

Nous mettrons l'accent sur le lien entre *module* et *structure de données*, et sur la robustesse des modules aux erreurs (notions de type *opaque* et de *contrat* qui seront mis en œuvre à l'aide de ***classes*** en C++).

Par ailleurs, selon le sujet du projet, *l'ordre de complexité* des algorithmes pourra être testé avec des fichiers tests plus exigeants que les autres. L'évaluation du travail portera essentiellement sur la résolution du problème du point de vue informatique : structuration des données, modularité du programme, robustesse et ré-utilisabilité des modules, stratégie de test, ordre de complexité calcul/mémoire des algorithmes mis en oeuvre, compromis performance/occupation mémoire.

2.Charge de travail et organisation en groupe de deux personnes

Le travail sera réparti en trois rendus. Ils seront décrits dans des documents indépendants fournis au fur et à mesure.

Il est important de fournir un travail régulier dès la première semaine du semestre car l'effort est concentré sur 12 semaines au lieu de 14. De ce fait il est logique que la quantité de travail à effectuer par semaine soit plus importante que celle calculée à partir du nombre de crédits. Prenez ce facteur en compte pour organiser votre semestre.

De plus, donnez-vous un objectif raisonnable et une limite de temps à consacrer au projet pour atteindre des objectifs compatibles avec votre niveau car nous observons régulièrement des personnes qui pénalisent leurs autres matières parce qu'elles doivent y consacrer un temps plus important que la moyenne. Le barème est visible dans le document décrivant chaque rendu.

Le projet doit être réalisé en **groupe de deux personnes** du fait de l'importance du travail à fournir mais aussi pour apprendre à maîtriser l'organisation d'un tel travail en groupe. En particulier nous verrons qu'il est possible de travailler de manière indépendante sur une composante du projet grâce à la compilation séparée et d'autres outils qui sont présentés plus loin dans ce document. Si vous décidez d'adopter cette approche, il est important de bien analyser le problème pour bien se répartir le travail puis de faire le point à intervalles réguliers (minimum une fois par semaine).

Les responsabilités en matière de sauvegarde du code source doivent être clairement définies au sein du groupe. Pour le partage du code vous pouvez travailler avec un répertoire sur gdrive.epfl.ch. Une alternative est d'utiliser gitlab.epfl.ch qui est un outil de développement de logiciel (non obligatoire car cela demande du temps pour maîtriser cet outil) un bref tutoriel en anglais et quelques liens sont disponibles sur moodle. Dans tous les cas vous êtes responsables de restreindre l'accès de votre code aux seuls deux membres du groupe.

Enfin le projet comporte un oral final individuel pour lequel nous demandons à chaque membre du groupe de comprendre le fonctionnement de l'ensemble du projet. Une performance faible à cet oral peut conduire à un second oral approfondi et une possible baisse des notes des rendus pour la personne concernée.

3 Le développement de projet dans le monde réel

Le but de cette section est d'introduire les termes utilisés pour désigner les étapes du développement d'un projet logiciel pour situer vos contributions dans le cadre de ce cours. La Figure 1 donne la liste de ces étapes, une brève définition et leur coût relatif.

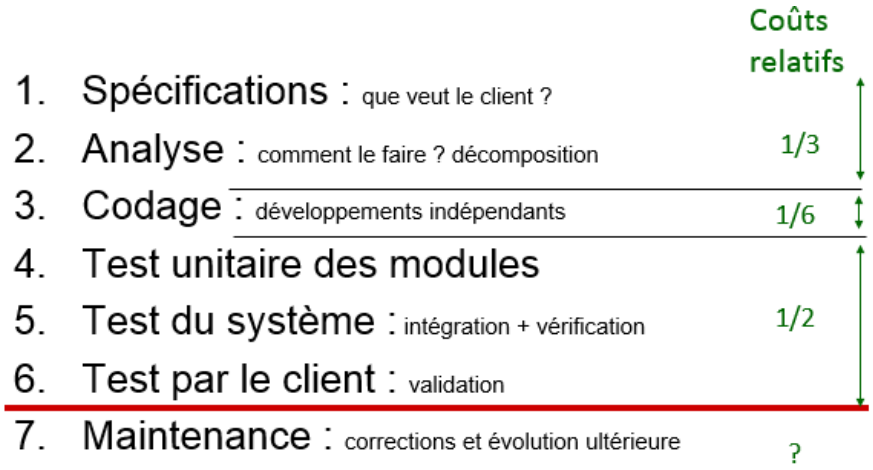


Figure 1: coûts relatifs des étapes du développement d'un projet (de 1 à 6)

Chaque étape produit un résultat qui est utilisé comme en entrée de l'étape suivante et ainsi de suite. En effet, les grandes compagnies de développement de logiciel ont du personnel spécialisé pour chaque étape. Le projet passe ainsi d'équipe en équipe au cours de son développement.

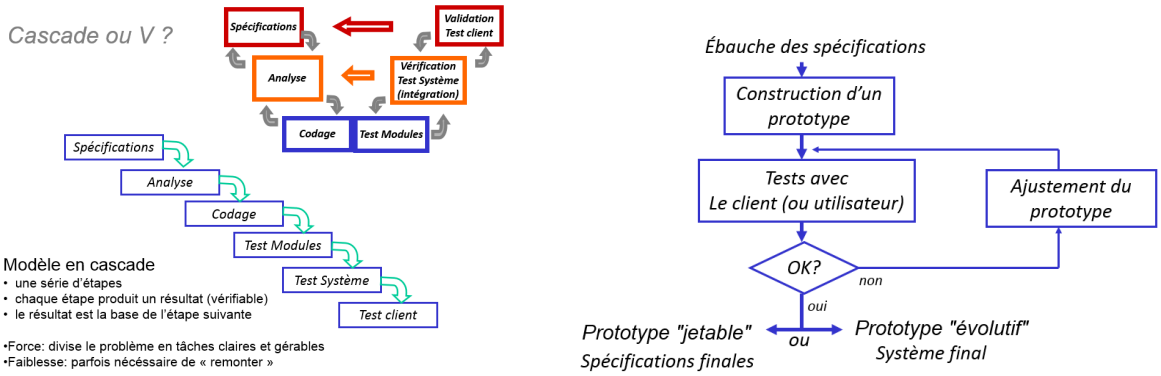


Figure 2: développement en cascade (gauche-bas) en V (gauche-haut) ou par prototypage (droite)

Pendant de nombreuses années le développement d'un projet s'est effectué selon l'approche en cascade (Figure 2 gauche-bas) mais celle-ci est peu efficace car on se rend compte d'éventuels problèmes seulement à la toute fin lors du test par le client. L'alternative avec le modèle de développement « en V » (Fig 2 gauche-haut) permet de remonter à certaines étapes antérieures si un problème de mise en œuvre est détecté après les spécifications.

Une approche récente très adaptée aux petites structures de développement est le *prototypage* (Figure 2 droite) qui consiste à impliquer le client plusieurs fois lors du développement. En effet, grâce à la possibilité d'essayer un prototype partiel du produit, le client peut mieux exprimer ses besoins et ré-ordonner ses priorités. Les six étapes du développement restent valides dans ce contexte mais elles opèrent sur une version réduite des spécifications.

C'est cette approche de prototypage évolutif que nous adoptons en vous faisant réaliser trois versions de plus en plus complètes du projet.

3.1 Spécifications

L'objectif premier des spécifications est de clarifier les besoins du client dans un document de spécifications (cahier des charges). Il s'agit d'identifier ce que veut le client (*Qu'est-ce que ça fait ?*) sans préciser *comment* cela doit être réalisé. Cela n'est pas toujours possible si certaines caractéristiques techniques sont imposées (Ex: temps de réponse de 0.1 s).

Les spécifications doivent aussi indiquer comment on peut vérifier/valider que les spécifications sont bien remplies lorsque le client obtient le produit final.

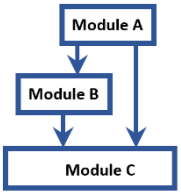
C'est la tâche la plus importante car la moindre modification a des conséquences très importantes: coût, échec du projet. Cette tâche est ardue car le client ne sait pas toujours ce qu'il veut. Il y aussi un risque d'écrire des spécifications ambigües qui peuvent être interprétées différemment de l'intention initiale du client, d'où l'importance d'adopter des notations claires et précises.

Dans le cadre du projet, l'enseignant joue le rôle du client et la donnée est la spécification du projet. Votre rôle est réduit concernant la définition de cette étape. Cependant, étant donné ce qui vient d'être mentionné plus haut, il est possible que certaines parties de spécifications ne soient pas assez claires ou soient mal comprises. Votre rôle est alors de poser des questions sur le **forum du projet** sur moodle pour améliorer ces spécifications.

3.2 Analyse

Il est nécessaire d'avoir une connaissance approfondie des spécifications pour pouvoir se lancer dans la phase d'analyse. Cette étape sert à définir *l'architecture* du projet, c'est-à-dire de :

1. sa décomposition en *modules*, chacun associé à une structure de donnée et un ensemble de tâches,
2. l'ensemble des *relations*, encore appelées *dépendances*, entre les modules.



Les relations sont précisées en identifiant *l'interface* de chaque module (fichier.h), c'est-à-dire l'ensemble des tâches qu'il se propose de réaliser => l'ensemble des fonctions qu'il offre à l'extérieur du module. Le module doit garantir la fiabilité de ces fonctions ; de ce fait on peut dire que l'interface est une sorte d'engagement du module à réaliser ces tâches correctement, c'est-à-dire un *contrat*.

Cette étape est essentiellement une étape de discussion « papier-crayon » pour préciser le rôle de chaque module et établir la *liste* des fonctions exportées, de leur *but* et de leurs éventuels *paramètres*.

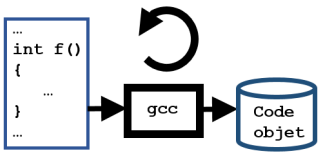
Important ! A ce stade on ne se soucie pas de *comment* on réalise chaque tâche avec chaque fonction (cela c'est le but de l'étape suivante du codage qui écrit *l'implémentation* des fonctions). Le détail des structures de données n'a pas besoin d'être précisé dès le début ; il est identifié par raffinements successifs. Ces choix peuvent être liés à des choix stratégiques de traitement algorithmique des données mais sans pour autant rentrer dans les détails de mise en œuvre des algorithmes.

Les principes guidant l'analyse sont les suivants:

- *Abstraction* à l'échelle du module : *but* du module sans entrer dans les détails *d'implémentation*
- *Ré-utilisation* : définition d'un ou de plusieurs modules de fonctions utilitaires de bas-niveau
- *Encapsulation* (information hiding) : afin de séparer *l'interface* de *l'implémentation*
- *Séparation des fonctionalités* (separation of concerns)
    - Pour nos projets l'approche Model-View-Controller sera vue en cours
- Et enfin Parcimonie/Simplicité : « la simplicité est la sophistication suprême » selon Léonard de Vinci

Dans le cadre de notre projet, il est fréquent qu'une architecture soit partiellement imposée, ce qui déborde du cadre des spécifications et constitue une partie du travail d'analyse.

3.3 Codage *(implémentation)*



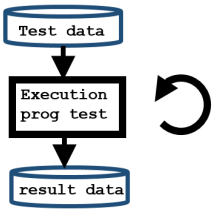
L'analyse permet de décomposer le problème en une liste de modules indépendants qui peuvent être *développés indépendamment les uns des autres* dans cette étape du *codage*. On doit disposer des éléments suivants :

- L'interface (fichier.h) des modules est disponible
  - donc utilisable dans d'autres modules.
  - Ce qui veut dire qu'on peut écrire des appels de fonctions des autres modules
- Pour chaque module: choix de structure de données + stratégie algorithmique
- (si approprié) l'apparence de l'interface graphique et le comportement de l'interaction sont aussi connus

Le résultat du codage est *l'implémentation* de chaque module dans le langage de programmation retenu (C++ dans notre cas). A ce stade on doit s'assurer au minimum que ce code compile.

Une bonne pratique est de ***recompiler très fréquemment***, chaque fois qu'on ajoute quelques lignes de codes source car on est bien plus efficace pour trouver un bug dans ces quelques lignes (depuis la dernière compilation avec succès) que dans l'ensemble des centaines de lignes de code du module entier.

3.4 Test unitaire des modules



Le but du test unitaire est de valider chaque fonction de chaque module.

Pour cela on retrouve le concept de *scaffolding* qui consiste à écrire des petits programmes qui les appellent avec des valeurs de test pour lesquelles on connait le résultat attendu.

Attention ! Si on obtient le résultat espéré on valide un test particulier mais pas forcément les autres appels possibles de la fonction. Il faut concevoir une batterie de tests qui couvre *l'ensemble des scénarios possibles* d'usage de la fonction car en général on ne peut pas tester tous les appels possibles d'une fonction (il y en a trop !). L'échec d'un test montre que votre fonction n'est certainement pas valide (et vous disposez d'une piste de recherche pour corriger cet échec). Par contre le fait de n'avoir aucun échec à votre batterie de tests ne garantit pas que votre fonction est valide car votre ensemble de tests pourrait être incomplet…

« testing can only show the presence of bugs, not their absence » E. Dijkstra

Le test unitaire d'un module est particulièrement adapté pour trouver les problèmes suivants :

- Bugs fréquents de bas niveau (cf bug check list)
- Erreurs de codage indétectable par le compilateur
- Oubli d'initialisation
- Nombre incorrect de passage dans une boucle
- Erreur de codage d'une condition
- Données en dehors de leur intervalle autorisé par le problème
- Certaines manipulations de pointeurs

Le résultat du test unitaire est un Module *vérifié*, c'est à dire dont les résultats sont conformes aux responsabilités identifiées dans la phase d'analyse à partir des spécifications.

Par contre il existe des problèmes qui seront détectés seulement beaucoup plus tard :

- Mauvaise compréhension d'une spécification correcte
- Spécification du module incomplète, ambigüe, erronée

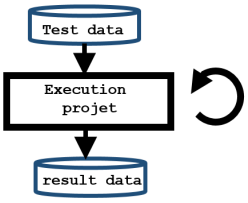
Dans ces cas, le module peut fonctionner correctement mais c'est le but du module qui a été mal défini…

3.5 Test du Système : intégration et vérification

Les modules ayant passé avec succès l'étape du test unitaire sont *intégrés* progressivement dans le système constituant le produit. Pour chaque ajout de module le système doit être testé avant d'intégrer un module supplémentaire.

Une fois tous les modules intégrés, le système doit être *vérifié* à l'aide d'un ensemble de tests pour s'assurer qu'il fait bien ce qui a été défini comme buts à l'étape d'analyse. Cette étape est de la responsabilité des personnes qui ont développé le système.

Dans le cadre de notre projet, c'est l'étape des derniers tests par le groupe d'étudiant.



3.6 Test par le client : validation

Le système vérifié est fourni au client qui effectue alors ses propres tests conformément à ce qui a été défini dans les spécifications. Cela constitue la *validation* du produit.

Il peut y avoir une différence avec l'étape de vérification si cette dernière est incomplète ou si les spécifications ont été mal comprises au moment de l'analyse du problème.

Dans le cadre de notre projet, c'est l'étape des tests par l'enseignant.

4. Méthode proactive de correction d'erreurs : les tests

"to test a program is to try to make it fail" B. Meyer

L'organisation systématique des tests, du simple au complexe, comme présenté dans les sections 3.4 à 3.6 permettent une meilleure maîtrise du développement. Nous complétons ici cette méthodologie.

Les anciens nous ont enseigné que "*l'erreur est humaine*" ; il est dans notre nature de nous tromper. Peu importe notre niveau de qualification, on fait des erreurs en codant et il vaut mieux s'armer des bonnes méthodes pour trouver les erreurs que de perdre du temps à s'en émouvoir.

La meilleure méthode est donc de faire des tests, le but étant de trouver des fautes... avant que le client ne les trouve par hasard (cf les lois de Murphy[1], effet démo, etc...).

4.1 Comment choisir les tests

Règle d'or : *la pertinence* des tests est plus importante que leur nombre. Votre choix doit être guidé par ces critères :

- Maximiser la couverture des comportements *distincts* du programme:
  - Identifier au minimum un cas pour chacune des règles à mettre en œuvre et leurs combinaisons
- identifier les sous-ensembles de données équivalentes, ne prendre qu'un représentant par sous-ensemble (tout dépend du contexte ; ex : nb positifs / négatifs)
- Identifier les valeurs limites, les cas particuliers et autour de ces valeurs. Ex: MAX, MIN, -1, 0, 1

Le résultat de la sélection est une batterie de tests (test suite) à exécuter systématiquement lorsqu'on modifie le programme.

[1] *Anything that can go wrong will go wrong.*

4.2 Refaire tous les tests encore et toujours

Constat : une même erreur a tendance à ressurgir plusieurs fois au cours de la mise au point

Le principe du *regression testing* consiste à vérifier que ce qui a été corrigé fonctionne toujours normalement en s'assurant que tous les tests continuent à fournir les résultats attendus.

Méthode: tout bug doit conduire à identifier un test qui sera ajouté à l'ensemble des scénarios de tests pour systématiquement vérifier qu'il ne se reproduit plus ultérieurement, par exemple à la suite d'une modification du code.

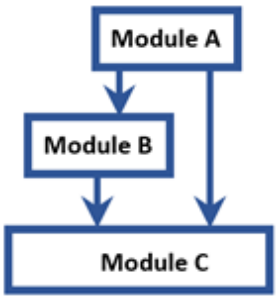
Principe de l'automatisation des tests: autant que possible, le code lui-même doit intégrer des instructions dédiées aux tests, par ex. vérifiant que certaines conditions sont effectivement remplies en début ou en fin d'algorithme.

De même du code produisant automatiquement des scenarios de test complète efficacement les tests "manuels". C'est un exemple de code supplémentaire de scaffolding.

4.3 Tester les relations entre les modules avec des stubs

La section 3.3 sur le codage a mentionné que l'étape d'analyse ayant fourni l'interface des modules, on pouvait écrire et compiler des appels de fonctions provenant d'autres modules. En effet on dispose du fichier en-tête (.h) qui est suffisant pour que le compilateur fasse son travail de production du code objet.

Cependant, qu'en est-il à l'étape du test unitaire du module (section 3.4) si on veut aussi exécuter des tests sur un module A qui dépend de fonctions fournies par un module B ?



Cela remet-il en question la possibilité du développement indépendant des modules ? En fait, pas totalement. La solution est de fournir le minimum permettant de simuler l'implémentation du module B tant qu'il n'est pas encore finalisé : on retrouve le concept de *stub* déjà rencontré pour le projet d'automne (section 2).

On procède donc en écrivant un **stub** pour chacune des fonctions exportées par B. Il s'agit d'écrire un squelette minimum de cette fonction qui :

- respecte son prototype
- contient dans le corps de la fonction, au choix :
    - Rien (si de type void) ou seulement un return compatible avec le type de la fonction
    - Fait un affichage dans le terminal indiquant la nature de l'action de la fonction
    - Fournit une version simplifiée de sa tâche

La plupart du temps cela suffit pour vérifier que les fonctions du module A font correctement ce qu'elles doivent faire à l'échelle du test unitaire. Il revient à l'étape d'intégration et de vérification (section 3.5) de finaliser les tests lorsque tous les modules ont passé leur étape de test unitaire.

Exemple: une interface **utilitaire.h** indique que son module doit fournir une fonction qui renvoie un angle entre deux vecteurs dont le type est appelé S2D pour cet exemple :

```
double util_angle(S2D a, S2D b);
```

En attendant que cette fonction soit validée, le module **utilitaire** doit mettre en place une *implémentation minimale* qui permette d'appeler les fonctions décrites dans utilitaire.h (sans faire planter le programme).

Voici un **stub** qui pourrait être écrit pour cette fonction:

```
double util_calcul_angle(S2D a, S2d b)
{
   cout << "util_calcul_angle not yet implemented" << endl;
   return 0. ;
}
```

A l'exécution il y aura affichage d'un message ce qui permettra de savoir que cette fonction a bien été appelée, voire de savoir combien de fois elle a été appelée. De plus la fonction renvoie une valeur compatible avec son type qui permet au niveau appelant de poursuivre son exécution pour le scénario correspondant à cette valeur. On peut changer la valeur renvoyée pour enrichir les tests du niveau appelant.

5. Méthode réactive de correction d'erreur imprévue : les bugs

Malgré tous nos soins dans les tests il peut arriver qu'une erreur survienne sans prévenir ; c'est le bug. Parfois l'indice qui semble indiqué par l'arrêt du programme n'aide pas à trouver l'origine du problème. C'est souvent le cas avec les problèmes de pointeurs.

Alors, comment trouver la cause du bug ?

Voici une méthode d'analyse et de verbalisation qui a fait ses preuves :

1. Décrire le comportement anormal observé et le noter pour ne pas l'oublier.
2. Formuler le comportement qui était attendu à la place
3. Verbaliser la différence entre 1) et 2) , l'interpréter par un mécanisme plausible avec des phrases: "*c'est comme si...*". Plusieurs hypothèses sont possibles
4. Faire une liste des hypothèses possibles, de la plus probable à la moins probable
5. Avoir le courage de se remettre en question. C'est probablement l'obstacle le plus grand dans cette recherche de bug car on renâcle à l'idée de devoir revenir sur du code qui nous a pris du temps à écrire. On est donc sans cesse tenté de fermer les yeux sur certaines faiblesses du code.
6. Analyser le code pour éliminer les hypothèses de cause
    a. Une seule hypothèse/modification du code à la fois
    b. Tester avec des jeux de données dont on connait le résultat

Le fait de verbaliser, parler à voix haute, est important pour le succès de la méthode car cette activité nous stimule pour nous représenter mentalement le problème et souvent on le résout par nous-même.

6. Exercices

Ces quelques questions visent à vérifier que vous savez distinguer les étapes du développement d'un projet logiciel ainsi que les éléments traités à chaque étape, et que vous comprenez les implications vis-à-vis du travail en groupe de deux personnes.

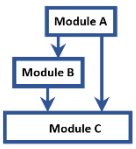
Les questions de a) à f) portent sur le cas général (*dans le monde réel*) tandis que les questions g) portent sur la mise en œuvre de notre projet avec ses adaptations à un cours de niveau introductif.

a) Spécifications

- les spécifications contiennent-elles une description des éléments suivants ?
  - les actions que doit réaliser le programme
  - la manière de tester si ces actions sont correctement effectuées
  - les structures de données à utiliser
  - les algorithmes à employer
  - la décomposition du projet en modules
- Dans l'approche de développement par *prototypage* est-il encore nécessaire de distinguer les six étapes du développement ?

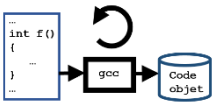
b) Analyse

- Quelle est la différence entre *l'interface* et *l'implémentation* d'un module ?
- Lequel des deux éléments précédents doit être précisé dans cette phase ?
- Qu'est-ce que *l'architecture* du projet ?
- A-t-on besoin du compilateur pour cette étape ?
- Pourquoi dit-on que l'interface d'un module est un *contrat* ?



c) Codage

- Dans un module A, peut-on écrire des appels à une fonction définie dans un module B et compiler pour obtenir le code objet du module A ?
- Quand est-il recommandé de compiler :
  - Quand le module est totalement écrit ?
  - Chaque fois qu'une fonction est totalement écrite ?
  - Pour chaque groupe de quelques lignes de codes ?
  - A chaque nouvelle ligne de code ?
- L'étape du codage détecte-t-elle les erreurs syntaxiques ou sémantiques ?

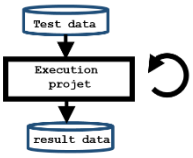


d) Tests Unitaires

- Puis-je tester l'exécution du Module A s'il dépend de fonctions du module B et que ce module B n'est pas encore finalisé et testé ?
- Comment définiriez-vous un test utile ?
- Y a-t-il un ordre particulier à respecter pour effectuer les test unitaires ?
- Ce travail peut-il être effectué indépendamment et en parallèle par différentes personnes pour tous les modules ?

e) Intégration et Vérification

- Y a-t-il un ordre particulier à respecter pour intégrer les modules ?
- Pourquoi dois-je refaire tous les tests à chaque étape de l'intégration ?



f) Validation

- Comment se pourrait-il que la validation donne un résultat différent de la vérification ?

g) Travail en groupe de deux personnes

- Quelles sont les étapes du développement où il faut travailler ensemble et celles où on peut travailler indépendamment de l'autre membre du groupe ?

- Identifier les faiblesses possibles de l'étape d'analyse pouvant induire des pertes de temps à l'étape du codage.

- En supposant que les deux membres d'un groupe se sont répartis le codage de deux modules A et B ; dans quels cas une personne est-elle cliente de l'autre personne ? Quel contrat les lie ?

Dans quel contexte est-il recommandé d'effectuer l'étape du codage ensemble ? Pourquoi ?

- **<u>Questions avancées (semaine 2)</u>**: le présent document a surtout souligné que l'interface exportait des fonctions. Cependant il est aussi possible d'exporter des structures (le cours précisera qu'on parle alors d'un type *concret*) ou des classes.

    - Supposons qu'une personne est responsable d'un module C dont l'interface exporte des fonctions et les modèles des structures manipulées par ces fonctions (comme paramètre ou résultat).

        - Le module C est-il fiable ? En d'autres termes, la personne responsable de ce module peut-elle garantir que les fonctions exportées dans l'interface fonctionnent toujours correctement ?

        - Supposons que la personne responsable du module C décide de changer le type un champ d'un modèle de structure. Cela peut-il avoir des conséquences pour les autres modules qui exploitent l'interface du module C ?

    - Supposons qu'une personne est responsable d'un module D dont l'interface exporte SEULEMENT des fonctions (pas de classe/structure).

        - Le module D peut-il être fiable ? En d'autres termes, la personne responsable de ce module peut-elle garantir que les fonctions exportées dans l'interface fonctionnent toujours correctement ?

        - Supposons que la personne responsable du module D décide de changer le type un champ d'un modèle de structure dans *l'implémentation* du module D. Cela peut-il avoir des conséquences pour les autres modules qui exploitent l'interface du module D ?