PEPI IDL . Ingénierie du Développement Logiciel
partage d'expérience et de pratiques en informatique

Bonnes Pratiques de développement

Véronique Baudin

Veronique.Baudin@laas.fr

LAAS-CNRS

Violaine Louvet

louvet@math.univ-lyon1.fr

Institut Camille Jourdan
Université Lyon1 &
CNRS

La Londe Les Maures – Décembre 2011

PROJET INFORMATIQUE

- Un projet est par définition innovant et unique
- Projet = ensemble fini d'activités et d'actions dont l'objectif est de répondre à un besoin défini en respectant des contraintes de temps et de coûts.
- Objectif décliné en 5 aspects
  - Fonctionnel : répondre à un besoin
  - Technique : respecter des spécifications et des contraintes de mise en oeuvre
  - Organisationnel : respecter un mode de fonctionnement de la structure (contexte utilisateurs)
  - Respect des délais : respecter les échéances du planning
  - Respect des coûts : respecter le budget fixé

Quelques causes des problèmes de développement logiciel

- Mauvaise interprétation des demandes des utilisateurs finaux
- Surspécification
- Incapacité à tenir compte des changements du cahier des charges
- Construction d'un processus/algorithme non fiable
- Membre(s) de l'équipe isolé(s): incapable(s) de déterminer qui a changé quoi, quand, où et pourquoi
- Procédures de tests coûteuses ou inefficaces
- Découverte tardive de problèmes
- Faible qualité du logiciel
- .....

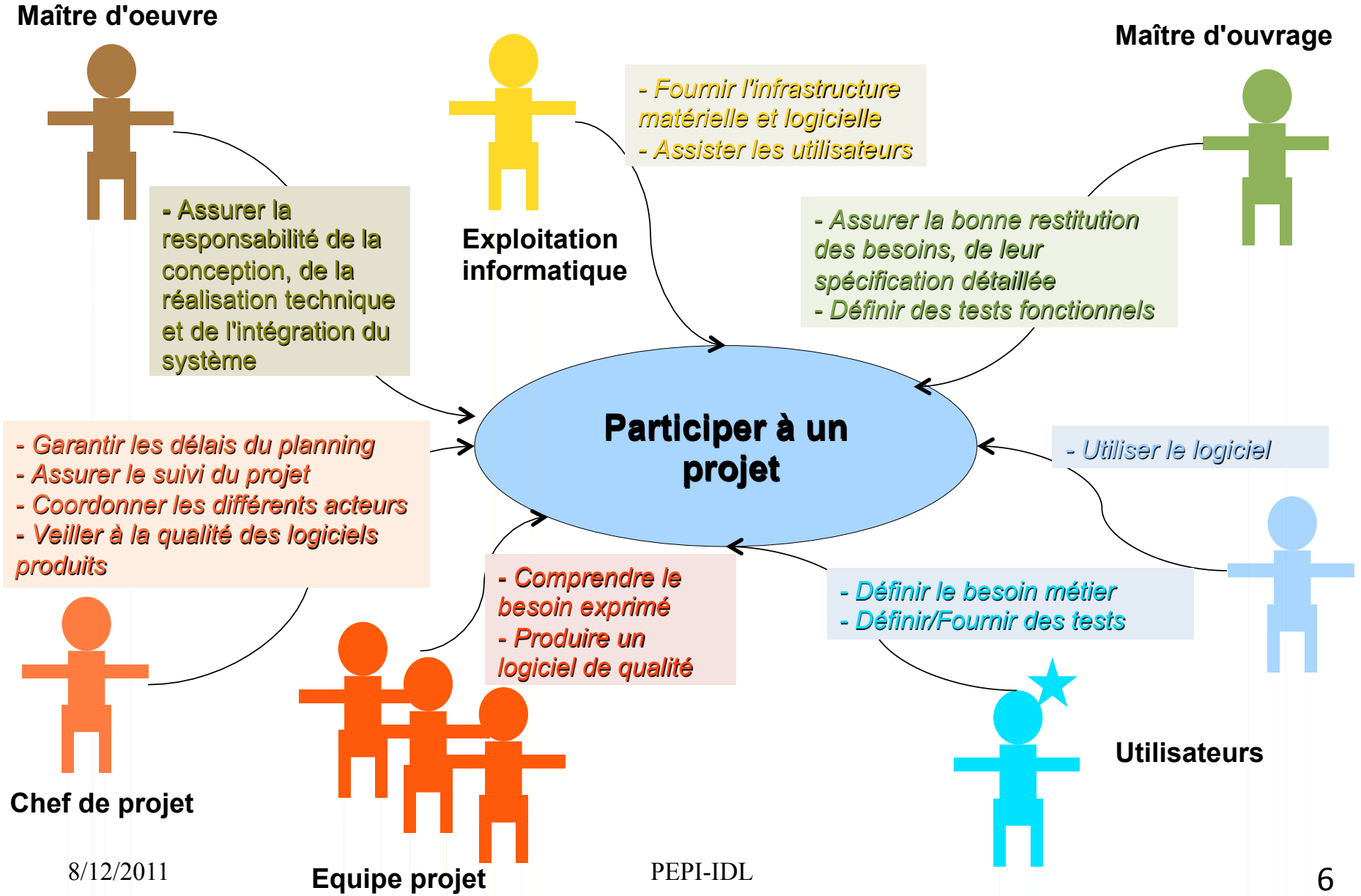
Quelques points clés pour le développement de logiciel

- Bien comprendre les demandes des utilisateurs finaux
- Tenir compte des modifications du cahier des charges
- Eviter de découvrir trop tard des défauts sérieux du projet
- Traiter au plus tôt tous les points critiques du projet
- Définir une architecture robuste et adaptée
- Bien maîtriser la complexité du système
- Bien communiquer avec l'utilisateur final
- Favoriser la réutilisation
- Faciliter le travail en équipe
- Concevoir en tenant compte des contraintes d'exploitation (ITIL)
- .....

PLAN

I. Gestion de projet
    Etude des besoins, phases d'analyse, de conception, tests
II. Qualité logicielle
    Référentiels existants
III. Sécurité
IV. Règles de programmation
    Règles de nommage, fonctions, commentaires, mise en forme, spécificités de certains langages
V. Standards et normes
    Normes des langages, problématiques des flottants
VI. Analyse de code
    Analyse statique, analyse dynamique
VII. Métriques
    Les différentes métriques, les outils, la couverture de code

I.I PROJET INFORMATIQUE



I.II Etapes de vie d'un projet

- Etude préliminaire ou de faisablité
- Etape de lancement
- Etudes générale et détaillée (ou spécifications)
- Étape de recherche et détermination de solutions pour le gestionnaire de projet
- Étape de réalisation et contrôle
- Étape d'analyse des recettes
- Étape de diffusion ou déploiement
- Étape de suivi des performances et de la qualité
- Étapes de maintenance puis retrait

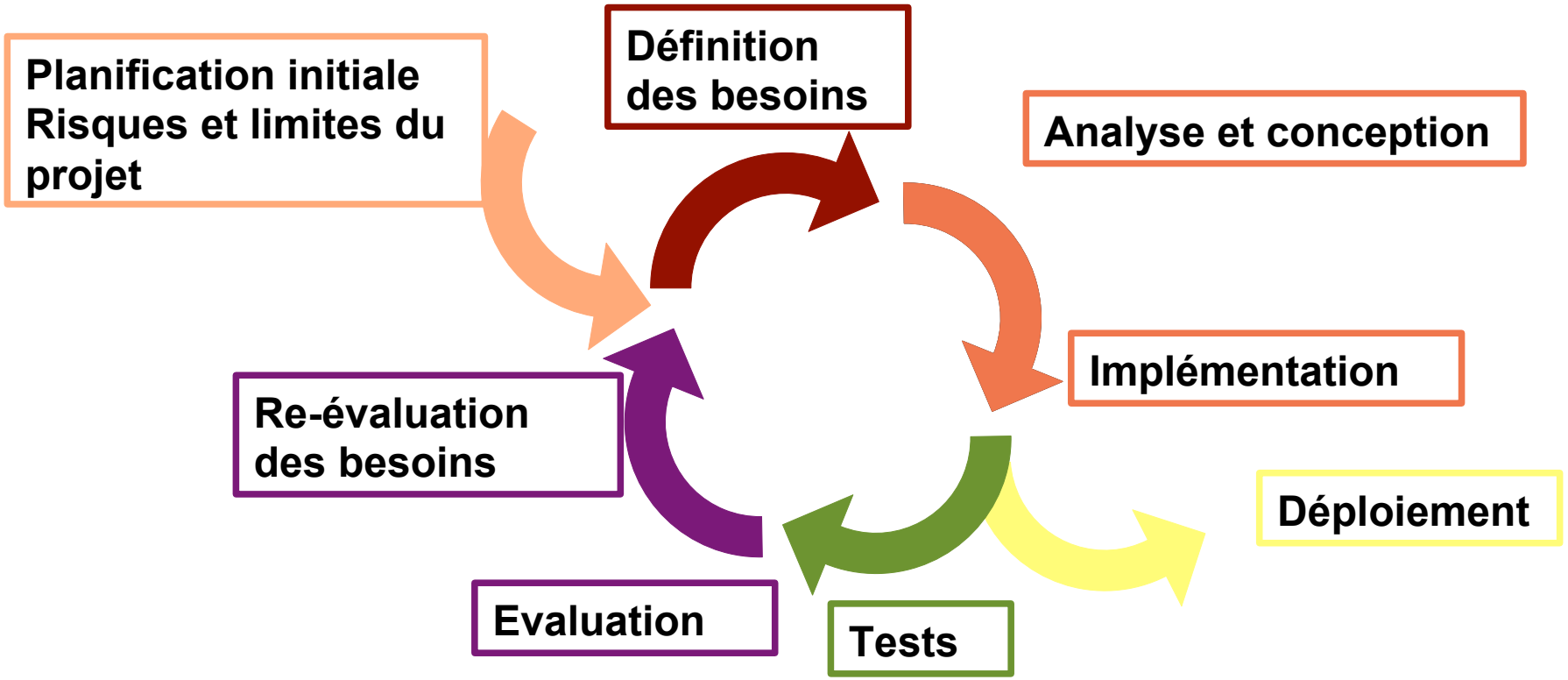
I.III Méthodes de développement logiciel (1)

Objectif = guider les développeurs de la phase d'analyse à celle de maintenance

- Point de vue théorique et descriptif
  - Existent en grand nombre
  - un aperçu ici :
    - http://www.projet-plume.org/ressource/presentations-envol-2008
- Point de vue pratique
  - Fortement lié à des approches utilisées couramment dans le monde du développement logiciel et largement décrites et enseignées.
  - 6 «bonnes pratiques »

I.III Méthodes de développement logiciel (2)

1. Développer un logiciel de façon itérative



I.III Méthodes de développement logiciel (3)

- Gains attendus
  - Mise en évidence tôt des incompréhensions/malentendus
  - L'utilisateur peut réagir tôt pour clarifier le cahier des charges /besoins
  - Les équipes de développement sont contraites de se concentrer sur les problèmes les plus critiques
  - Les tests effectués et répétés permettent une évaluation objective de l'état d'avancement du projet
  - Les incohérences entre cahier des charges/modèles de conception et implémentation peuvent être découvertes tôt
  - La charge de travail des personnes chargées des tests est répartie uniformément sur tout le cycle de vie du développement
  - Les équipes peuvent exploiter le savoir acquis au cours du projet et améliorer le processus de développement (pour ce projet et d'autres projets)
  - Les utilisateurs et le maitre d'ouvrage ont des preuves réelles de l'avancement du projet, tout au long du cycle de vie du logiciel.

I.III Méthodes de développement logiciel (4)

2. Gérer les exigences :

- *exigence = condition que doit remplir un système ou tâche qu'il doit accomplir*
- *Pour un système informatique: exigences dynamiques*
- *Gestion des exigences : 3 activités:*
  - *Mettre en évidence, organiser et décrire les fonctionnalités et les contraintes du système*
  - *Evaluer les changements à apporter au cahier des charges et estimer leur impact*
  - *Décrire les différents compromis et les décisions prises, et en faire le suivi.*

- Gains attendus :
  - *Le dialogue entre utilisateurs et développeurs concernés se réfère à des besoins bien définis*
  - *On peut affecter des priorités aux exigences, les filtrer et en faire le suivi*
  - *Une évaluation objective des fonctionnalités et des performances est possible*
  - *Les incohérences sont détectées plus facilement*
  - *On peut utiliser un outil pour gérer les exigences d'un système, leurs attributs, leurs dépendances et leurs liens vers des documents externes*
    - *Reqtify (outil propriétaire) http://www.geensoft.com/fr/article/reqtify*
    - *GenSpec via un googlegroup (http://groups.google.ca/group/genspec)*
    - *...*

I.III Méthodes de développement logiciel (5)

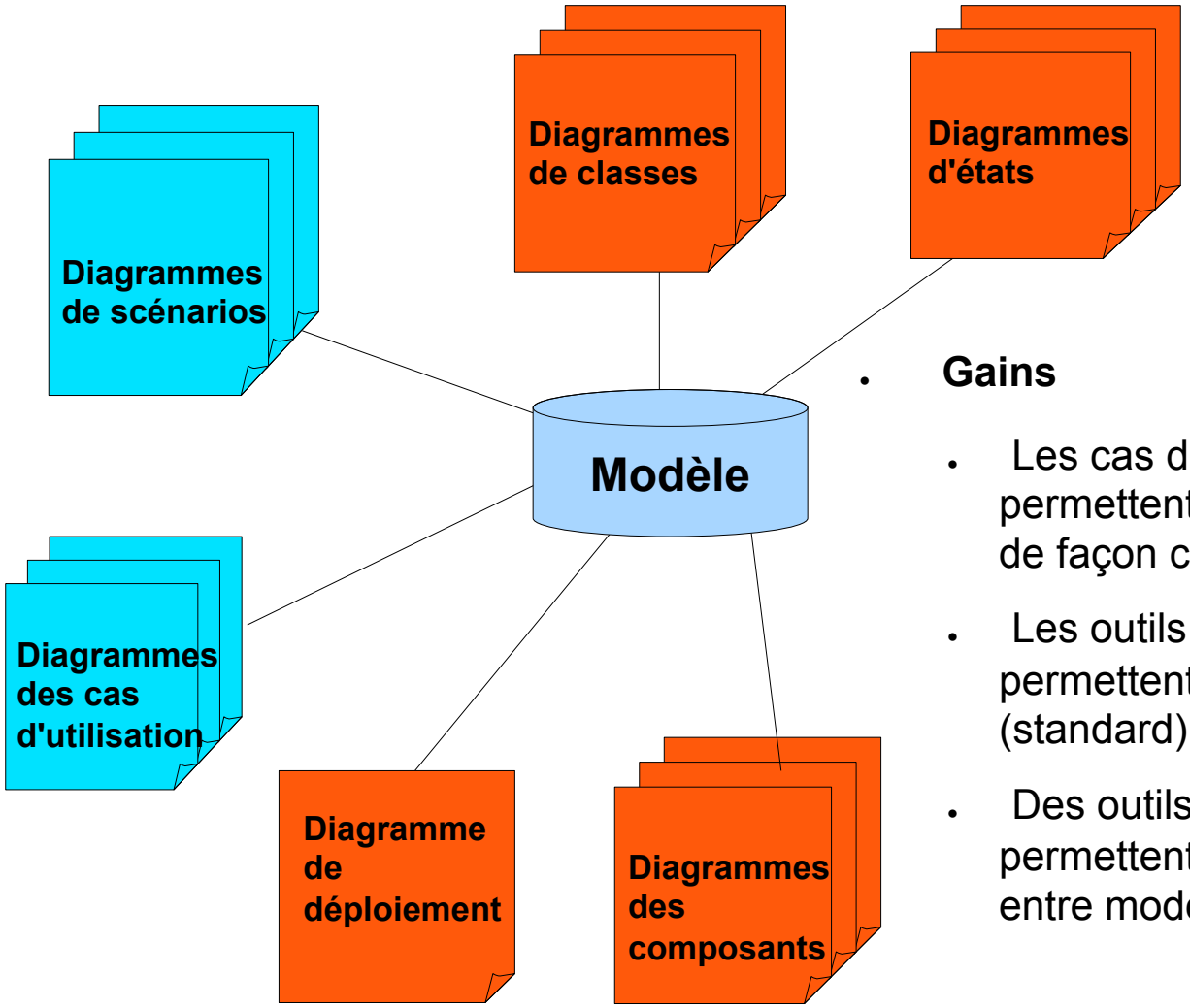
3. Utiliser des architectures à base de composants

- Importance de construire des architectures souples:
  - Niveau de réutilisation significatif
  - Division du travail claire entre les différents développeurs
  - Isolent les dépendances entre logiciel et matériel
  - Facilitent la maintenance
- Importance du développement à base de composants:
  - Permet la réutilisation et l'adaptation de composants existants
  - Facilité de mise en oeuvre par l'utilisation de plates-formes normalisées telles que EJB[1] de SUN , CORBA[2] de l'OMG[3] , SOA[4] promu par l'OASIS[5]

1. Enterprise Java Beans
2. Common Object Request Broker Architecture
3. Object Management Group
4. Architecture Orientée Services
5. consortium qui pilote le développement, la convergence et l'adoption de normes ouvertes pour la société de l'information mondiale.

I.III Méthodes de développement logiciel (6)

4. Modéliser graphiquement le logiciel



- **Gains**
  - Les cas d'utilisation et les scénarios permettent de spécifier un comportement de façon claire et non ambiguë
  - Les outils de modélisation graphiques permettent une modélisation UML (standard)
  - Des outils de génération de code permettent de conserver la cohérence entre modèle et code source

I.III Méthodes de développement logiciel (8)

5. Vérifier la qualité du logiciel

- Importance d'une évaluation en continu de la qualité du système
  - Fonctionnalités
  - Fiabilité
  - Performances
- Part importante des activités de test
- Gains
  - Évaluation des résultats de tests et non des documents => objectivité de l'évaluation du système
  - Permet d'identifier des incohérences entre cahier des charges et modèles de conception et d'implémentation
  - Anomalies de fonctionnement identifiées tôt

I.III Méthodes de développement logiciel (9)

6. Contrôler les changements apportés au logiciel

- Si le système utilise de nombreux composants logiciels et est développé par un grand nombre de développeurs ou des développeurs géographiquement répartis
- Alors, nécessité de livrer un ensemble de documents de référence testé à la fin de chaque itération pour assurer la traçabilité entre les éléments de chaque version

- Gains
  - Processus de gestion des changements bien défini
  - Demandes de changement contribuent à une communication claire entre les acteurs
  - Espaces de travail isolés pour des travaux effectués en parallèle (SVN, ClearCase, ...)
  - Évaluation et contrôle de l'impact des changements sur le logiciel
  - Métrique pour l'évaluation de l'état d'avancement du projet

PLAN

I. Gestion de projet
   Etude des besoins, phases d'analyse, de conception, tests
II. Qualité logicielle
   Référentiels existants
III. Sécurité
IV. Règles de programmation
   Règles de nommage, fonctions, commentaires, mise en forme, spécificités de certains langages
V. Standards et normes
   Normes des langages, problématiques des flottants
VI. Analyse de code
   Analyse statique, analyse dynamique
VII. Métriques
   Les différentes métriques, les outils, la couverture de code

II Qualité du logiciel (1)

- La qualité d'un logiciel n'a pas de mesure objective, ni de définition formelle

- Par contre:
  - En matière de process, une organisation peut se faire certifier ISO9001*
  - Qualité logiciel caractérisée par des facteurs de qualité.

Ces facteurs de qualité peuvent être considérés selon le point de vue de l'acteur (utilisateur ou concepteur du logiciel).

* "j'écris ce que je fais, je fais ce que j'écris"

II Qualité du logiciel (2)

utilisateur

Intégrité/sécurité

Aptitude du logiciel à se protéger contre des accès non autorisés

Correction/validité

Aptitude à répondre aux besoins et à remplir les fonctions définies dans le cahier des charges

Convivialité

Facilité
- d'apprentissage et d'utilisation,
- de préparation des données,
- de correction des erreurs d'utilisation,
- d'interprétation des retours

Robustesse/fiabilité

Aptitude à fonctionner dans des conditions non prévues dans le cahier des charges, ou anormales

Efficacité

Utilisation optimale des ressources matérielles (processeur, mémoire, réseau, ...)

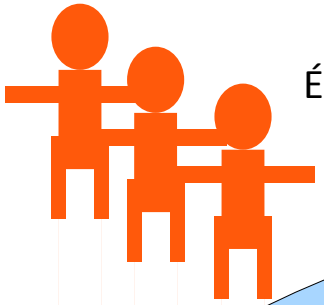
Extensibilité

Facilité d'ajout de nouvelles fonctionnalités

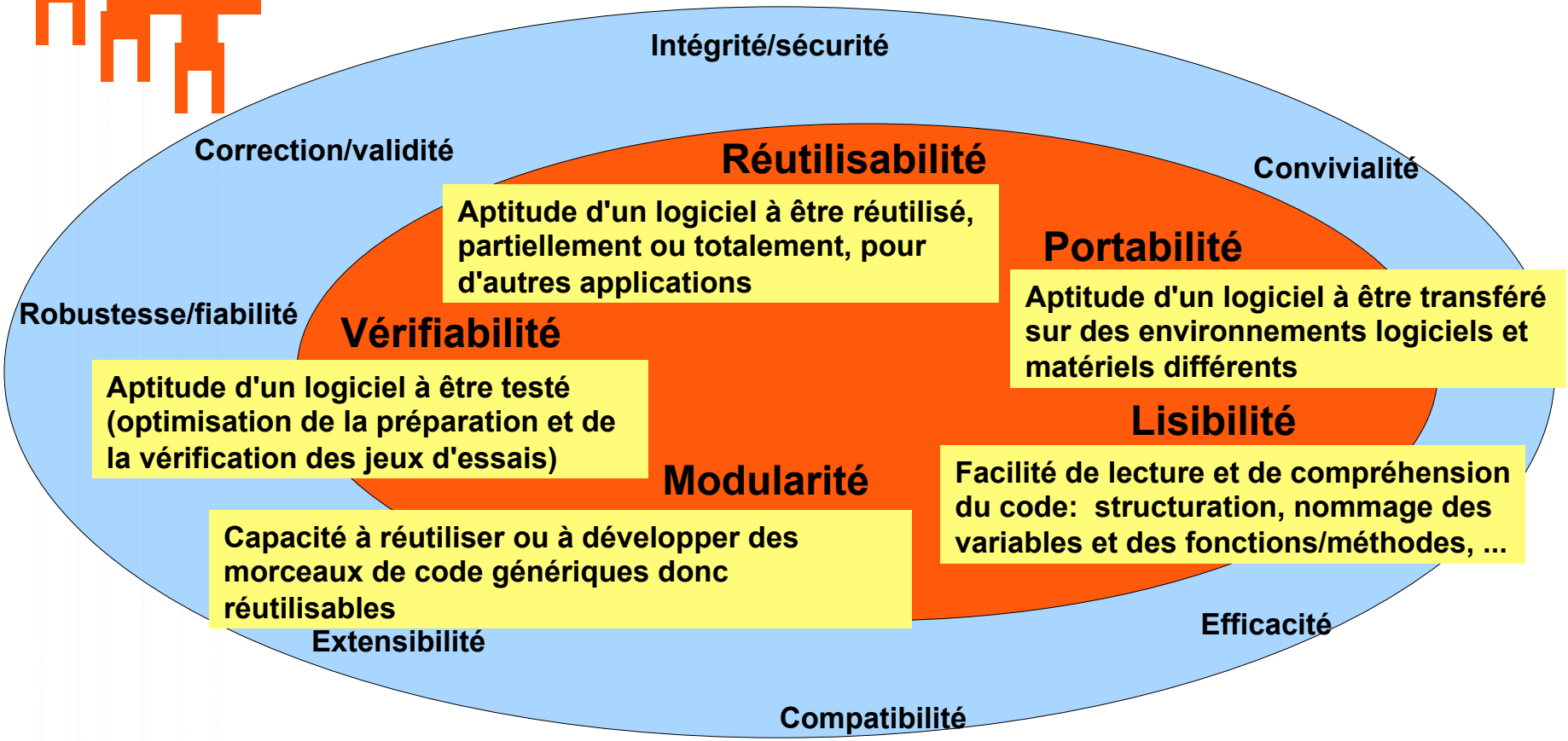
Compatibilité

Facilité de combinaison de ce logiciel avec d'autres

II Qualité du logiciel (3)



Équipe projet



II Qualité du logiciel (4)

- Normes:
  - ISO/CEI 9126 : définit un langage pour modéliser/décrire les qualités d'un logiciel
  - ISO25000/SQuaRE (Software Product Quality Requirement and Evaluation): a pour objectif de poser le cadre et la référence pour définir les exigences qualité d'un logiciel (ISO 9126) et la manière dont ces exigences seront évaluées (ISO 14598).
- Comment mesurer ?
  - Quelque chose de prévu pour fin 2011, à l'étude entre SEI et OMG
  - Création du CISQ (Consortium for IT Software Quality):
    - http://www.it-cisq.org
- Comment mettre en oeuvre à notre niveau ?
  - Par le biais de bonnes pratiques de développement / programmation

PLAN

I. Gestion de projet
   Etude des besoins, phases d'analyse, de conception, tests
II. Qualité logicielle
   Référentiels existants
III. Sécurité

IV. Règles de programmation
   Règles de nommage, fonctions, commentaires, mise en forme, spécificités de certains langages
V. Standards et normes
   Normes des langages, problématiques des flottants
VI. Analyse de code
   Analyse statique, analyse dynamique
VII. Métriques
   Les différentes métriques, les outils, la couverture de code

III Sécurité (1)

- Sécurité = 1 aspect de la sûreté de fonctionnement

- Sûreté de fonctionnement :
  - «aptitude d'une entité à satisfaire à une ou plusieurs fonctions requises dans des conditions données» - Alain Villemeur, Sûreté de fonctionnement des systèmes industriels, Eyrolles, 1988
  - «propriété qui permet aux utilisateurs du système de placer une confiance justifiée dans le service qu'il leur délivre» - LIS sous la direction de J.-C. Laprie, Guide de la sûreté de fonctionnement, Cépaduès, 1995
  - Englobe principalement : fiabilité, disponibilité, maintenabilité sécurité complété par durabilité, testabilité ou des compositions de ces aptitudes

Aspect à prendre en compte sur toute la durée du projet

III Sécurité (2)

- **Conception**
  - anticiper de mauvaises utilisations d'un logiciel permet de prévoir des contraintes qui empêchent son utilisation inadéquate et minimisent les impacts négatifs de ce type d'usage
    - *Logiciel calculette : prévoir une division par zéro*
- **Implémentation**
  - Principe de base = si une méthode ou un sous-programme reçoit de mauvaises données, elle/il ne doit pas être mis en difficulté, même si ces données sont dûes à une erreur d'une autre méthode/sous-programme

Protection contre des entrées invalides

- Vérifier les valeurs de toutes les entrées provenant de sources externes (fichier, utilisateur, réseau, interface externe, ...):
  - ✓ Valeurs numériques : limites spécifiées
  - ✓ Chaîne de caractères : longueur max + interprétation correcte
  - ✓ Tentative de saturation de buffers
  - ✓ Injection de commandes SQL, de code HTML ou XML
  - ✓ Dépassement de capacité pour les entiers,
  - ✓ Données passées aux appels système
- Vérifier les valeurs des paramètres d'entrée des méthodes/sous-programmes
- Mettre en place une méthode de prise en charge des mauvaises entrées

III Sécurité (3)

- **Utiliser des assertions à bon escient**
  - Assertion = code permettant au programme de se contrôler lui-même pendant son exécution *(voir bonus assert)*
  - À utiliser
    - pour dépister des erreurs dans le code en développement
    - pour documenter et vérifier des pré et post conditions
  - À désactiver pour la version de production

- **Prendre en charge les erreurs identifiées qui peuvent se produire**
  - Prises en charge possibles:
    - Retourner une valeur neutre si le contexte le permet
    - Attendre la donnée valide suivante
    - Retourner la réponse de l'exécution précédente
    - Enregistrer un message d'avertissement dans un fichier de log ou afficher un message d'erreur explicite
    - Appeler un sous-programme ou un objet de traitement d'erreur
    - Tout arrêter

III Sécurité (4)

- **Traiter les exceptions**
  - Exception = moyen de transmission d'erreurs ou d'évènements exceptionnels au code appelant
  - Mécanisme:
    - ✓ Throw pour alerter
    - ✓ Try-catch pour capturer
    - ✓ Try-finally pour corriger éventuellement et poursuivre l'exécution
  - Quand les utiliser ?
    - ✓ Pour avertir les autres parties du programme des erreurs qui ne doivent pas être ignorées
    - ✓ Lors d'occurence de conditions réellement exceptionnelles
  - Quelques conseils:
    - ✓ Éviter de déclencher une exception dans un constructeur ou un destructeur
    - ✓ Donner toutes les informations pertinentes qui ont conduit au déclenchement de l'exception dans le message de l'exception
- **Limiter les dégâts causés par les erreurs**
  - Principe = éviter qu'une erreur ne déclenche d'autres erreurs en chaîne
  - Approche proposée:
    - ✓ Hypothèse des méthodes publiques: données pas sûres, donc à vérifier
    - ✓ Hypothèse des méthodes privées: données vérifiées par méthodes publiques, donc sûres
    - ✓ Convertir les données d'entrée vers leur forme définitive le plus tôt possible: (string "oui" à convertir en true ou 1)

III Sécurité (5)

• Mettre en oeuvre une aide au déboggage

La version de développement d'un logiciel ne subit pas les mêmes contraintes que la version finale ou de production: performances, minimisation des ressources, présence d'opérations supplémentaires à destination du développeur, ...

- ✓ Vérifier que les " assert" arrêtent le programme
- ✓ Détecter les erreurs d'allocation mémoire en remplissant complètement la mémoire allouée
- ✓ Remplir complètement les fichiers ou flux alloués pour détecter les erreurs de format
- ✓ Vérifier que les clauses default ou else de tous les caseof provoquent une défaillance grave
- ✓ Remplir un objet avec des données factices avant de le détruire
- ✓ Si possible, modifier le programme pour qu'il envoie les journaux d'erreurs par mail
- ✓ Prévoir la suppression des aides au déboggage (#define, code de debbug "débrayable")

• Déterminer la bonne quantité de programmation défensive à laisser dans le code final

Durant le développement, les erreurs doivent être très visibles: provoquer l'arrêt du programme.En phase de production, les erreurs doivent être aussi peu bloquantes que possible

- ✓ Laisser le code qui détecte les erreurs importantes, et supprimer (debbug = off) le code qui détecte des erreurs non bloquantes
- ✓ Supprimer les codes qui provoquent des blocages brutaux: *prévoir une sauvegarde des données entrées ou calculées avant un arrêt*
- ✓ Enregistrer les erreurs pour effectuer un support technique: *utilisation de fichiers de log*
- ✓ S'assurer de la correction des messages d'erreur laissés dans la version de production
- ✓ Réfléchir à l'utilité de la défense et établir les priorités de ce type de programmation en conséquence: *vérification des données à bon escient, mais pas à toutes les étapes du programme*

PLAN

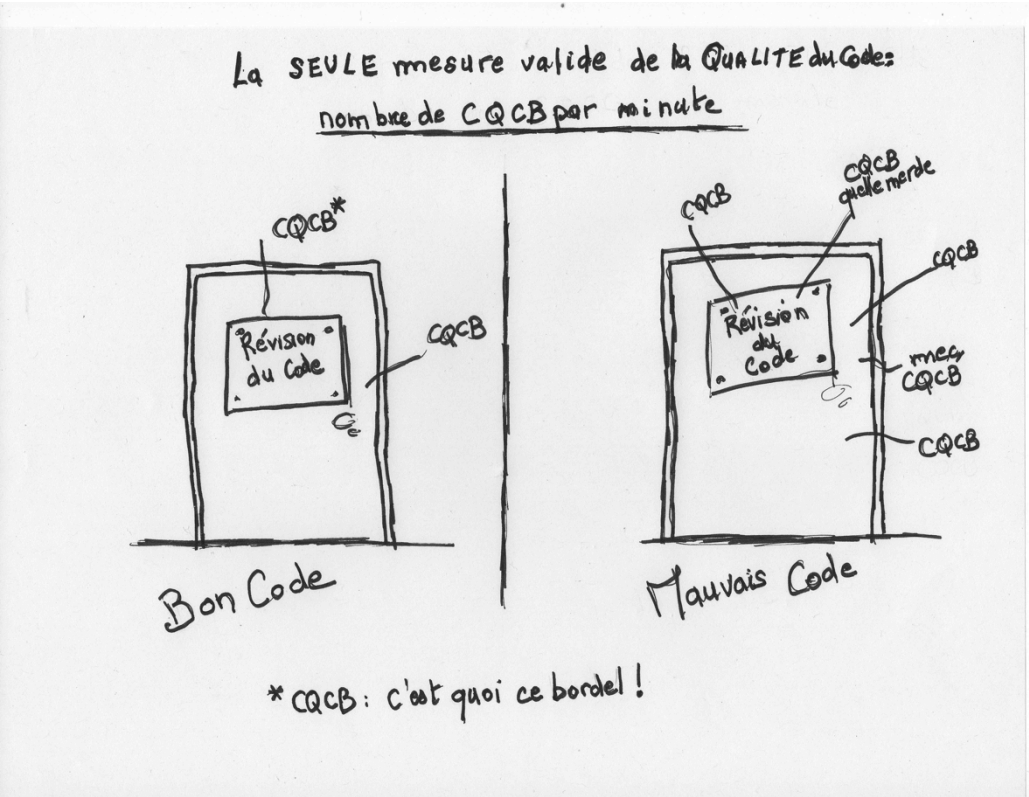
I. Gestion de projet
    Etude des besoins, phases d'analyse, de conception, tests
II. Qualité logicielle
    Référentiels existants
III. Sécurité

IV. Règles de programmation
    Règles de nommage, fonctions, commentaires, mise en forme, spécificités de certains langages
V. Standards et normes
    Normes des langages, problématiques des flottants
VI. Analyse de code
    Analyse statique, analyse dynamique
VII. Métriques
    Les différentes métriques, les outils, la couverture de code

Bon code et mauvais code

Mauvais code :

- Difficile à comprendre et à lire
- Difficile à maintenir
- Difficile à faire évoluer
- Difficile à optimiser
- Difficile à partager et à réutiliser



Code propre :

- Peut être lu et amélioré par un développeur autre que l'auteur d'origine
- Est facile à lire et à comprendre
- A des dépendances minimales
- Est ciblé : ne fait qu'une chose et la fait bien

IV Règles de programmation

IV-1 Règles de nommage

Définissent la façon de nommer les identificateurs utilisés de façon à faciliter la lecture du code et à aider à détecter des erreurs.

- Choisir des noms révélateurs des intentions : un nom qui nécessite un commentaire n'est pas auto suffisant.
- Eviter les possibilités d'amalgames et d'ambiguités
- Choisir des noms prononçables : cela facilite le travail en équipe
- Choisir des noms facilitant les recherches lexicales
- Préférer les identificateurs en anglais (plus universels)
- Choisir un mot par concept : par exemple, *get* ou *set*

Notation hongroise : Inventée par Charles Simony (d'origine hongroise) et utilisée chez Microsoft. Convention de nommage qui met en avant le type d'un objet.

- Par exemple, la variable booléenne danger est préfixée par un b pour indiquer un booléen : bDanger.
- Le préfixe est toujours écrit en minuscule puis la première lettre suivante en majuscule.
- La notation hongroise se plaît bien dans les langages peu typés (pleins de conversions implicites), mais elle tend à disparaître.

Choisir ses propres règles et rester cohérent :

- Utiliser une majuscule pour le début de chaque mot significatif dans le nom d'une variable, les autres restant en minuscule (ex : *StructuredMesh*)
- Commencer par une minuscule et appliquer ensuite la même règle (ex : s*tructuredMesh*)
- Utiliser les underscores pour séparer les mots (ex : S*tructured_Mesh*)

IV Règles de programmation

IV-2 Fonctions

➢Faire court: quelques dizaines de lignes à un écran
➢Une fonction ne doit faire qu'une seule chose, à un niveau d'abstraction donné.

➢Choisir des noms descriptifs, en général avec un verbe d'action.
➢Les blocs des instructions *if*, *else*, *while* … ne doivent occuper qu'une seule ligne : selon le niveau d'abstraction, il est probable que cette ligne soit un appel de fonction.

➢Eviter toute forme de redondance (fonctions faisant le même traitement ou fonction intégrant un traitement codé dans une autre fonction).

IV Règles de programmation

IV-3 Commentaires

Les commentaires sont un mal nécessaire : ils pallient notre incapacité à exprimer nos intentions par le code.

La lisibilité d'un code n'augmente pas avec le nombre de lignes de commentaires

La seule source d'information absolument juste est le code.

➢La cohérence entre commentaires et code est à l'entière responsabilité du programmeur
➢Ne pas compenser le mauvais code par des commentaires : il est préférable de le réécrire
➢S'expliquer directement dans le code plutôt que dans un commentaire :

```
// Verifier si l'employé peut bénéficier de tous
// les avantages
if ((employee.flags & HOURLY_FLAG) &&
(employee.age > 65))
```

IV Règles de programmation

IV-3 Commentaires

Les commentaires sont un mal nécessaire : ils pallient notre incapacité à exprimer nos intentions par le code.

La lisibilité d'un code n'augmente pas avec le nombre de lignes de commentaires

La seule source d'information absolument juste est le code.

➢La cohérence entre commentaires et code est à l'entière responsabilité du programmeur
➢Ne pas compenser le mauvais code par des commentaires : il est préférable de le réécrire
➢S'expliquer directement dans le code plutôt que dans un commentaire :

```
// Verifier si l'employé peut bénéficier de tous
// les avantages
if ((employee.flags & HOURLY_FLAG) &&
(employee.age > 65))
```

```
if (employee.isEligibleForFullBenefits())
```

Commentaires

➢Commentaires légaux : copyright, propriété, licence … à placer en début de chaque fichier source.
➢Commentaires TODO : permettent de laisser des notes de type «à faire».
➢Documentation d'entête : d'un fichier, d'une fonction, d'une classe … MAIS elle doit être à jour et réalisée que si elle apporte quelque chose par rapport au code.
➢Commentaires informatifs : algorithmes, référence à un article explicitant la méthode utilisée … de façon succinte. Le détail n'a pas sa place dans le code.
➢Expliquer les intentions, documenter les décisions.
➢Clarifier quand le code n'est pas suffisant.
➢Avertir des conséquences ou des précautions d'usage : fonction non thread safe, ...

➢Paraphrase du code, redondance :
i = 0 # set i to zero
➢Commentaire n'ayant de signification que pour l'auteur du code.
➢Commentaires non à jour.
➢Commentaires de journalisation : mieux vaut utiliser un système de gestion de version.
➢Commentaires parasites n'apportant aucune information nouvelle :

```
// Constructeur par défaut
Matrix();
```

➢Commentaires d'accolades fermantes : si ils sont nécessaires, c'est qu'il faut raccourcir la fonction considérée.
➢Lignes de code en commentaire : mieux vaut utiliser un système de gestion de version.
➢Informations loin du code concerné.

IV Règles de programmation

IV-4 Mise en forme

➢Les fichiers courts sont plus facile à comprendre que les fichiers longs.
➢Ajouter des espacements horizontaux (lignes vides) entre les éléments d'un fichier (fonctions, groupe d'instructions …).
➢Les concepts étroitement liés doivent être verticalement proches pour faciliter la lecture et la compréhension (par exemple fonction appelante et fonction appelée).
➢Une seule instruction par ligne.
➢Un nombre de colonnes par ligne de code limité pour ne pas avoir à utiliser un slider pour faire défiler le code horizontalement.
➢Utilisation de l'indentation dans les structures de contrôle (à la Python). Les commentaires associés doivent suivre aussi cette indentation.
➢Un espacement uniforme (nb espace) à mettre :

- avant ou après un opérateur comme “+" ou “="
- avant ou après une virgule dans une liste (d'arguments par exemple),
- avant ou après une parenthèse ouvrante ou fermante (d'une liste d'arguments par exemple).

➢Un alignement horizontal des déclarations : alignement des types, des noms de variables :

```
int     nrows;
int     nlines;
double  epsilon;
```

IV Règles de programmation

IV-5 Spécificités du C

➢ Le langage C permet de mélanger tests, appels de fonctions, opérations : attention à coder des lignes simples.

➢ Les noms des constantes symboliques et des macros instructions doivent être écrits en majuscule.

➢ Les arguments du programme principal sont standards :*int main(int argc, char *argv[])*

➢ Limiter l'utilisation des opérateurs ++ et -- : ils ne sont autorisés que pour les indices de boucles et les pointeurs, à condition qu'aucun autre opérateur n'apparaisse et que seule l'utilisation post-fixée soit utilisée.

➢ Limiter l'utilisation de l'opérateur de test ternaire : l'expression conditionnelle *?=* est interdite en dehors des macros.

➢ Utiliser les accolades et bien aligner les accolades ouvrantes et fermantes correspondantes.

➢ Éviter d'utiliser les opérateurs d'affectation spécifique au C : assignation composée (*+=, -=, *=, %=, /=*) et affectation multiple (*v1 = v2 = v3;*) déconseillées.

➢ Ne pas déclarer plusieurs variables dans la même instruction.

IV Règles de programmation

IV-6 Spécificités du C++

➢Mêmes conseils que pour le langage C.

➢Une règle qui semble se généraliser : les identificateurs de type commencent par une majuscule, et les instances par une minuscule.

➢Respecter le principe de l'encapsulation.

➢Les classes doivent être aussi simples que possible. Une classe trop complexe (+ de 20 méthodes) peut être le plus souvent découpée en plusieurs classes plus simples.

➢Définissez toujours les constructeurs, destructeurs et opérateur =.

➢Si votre classe contient des pointeurs, vous devez déclarer le contructeur de copie, l'opérateur = et le destructeur.

➢Eviter à tout prix l'utilisation de variables globales. Si vous avez absolument besoin de ressources globalement accessibles, faites une classe autour de ces ressources, et créez des méthodes contrôlant l'accès à ces ressources.

➢Lorsque vous définissez une fonction, prenez l'habitude de passer les arguments de préférence comme références constantes: l'appel généré a la simplicité d'un appel avec passage de paramètre par valeur, sans avoir pour autant l'inconvénient de nécessiter un appel à un constructeur et à un destructeur.

➢Utiliser la clause const autant que possible et de manière systématique.

➢Utiliser les bibliothèques standards C++ et pas les fonctionnalités propres à un système spécifique.

IV Règles de programmation

IV-7 Spécificités du FORTRAN

➢La casse n'est pas prise en compte dans le nom des identificateurs : utilisation de l “underscore“ pour définir des noms descriptifs.

➢Mots clés fortran, fonctions intrinsèques, types utilisateurs : en majuscule.

➢Utilisation du «*::*» dans les déclarations, séparant le type du nom de la variable.

➢Eviter les mots-clés *EQUIVALENCE, SAVE, GOTO, ENTRY, COMMON, PAUSE* (issus du FORTRAN 77).

➢Utiliser les fonctions intrinsèques du langage.

➢Respecter le nombre de dimensions des tableaux dans le passage des arguments des subroutines.

➢Utiliser *IMPLICIT NONE* pour assurer la déclaration de toutes les variables.

➢Spécifier la vocation des arguments d’une procédure avec l'attribut *INTENT*.

➢Utiliser les modules pour structurer le programme. D'une façon générale, utiliser les nouvelles fonctionnalités du FORTRAN90/95 pour réaliser des implémentations pseudo-orientées objet.

IV Règles de programmation

IV-8 Spécificités du Python

➢Ne pas mixer tabulations et espaces dans l'indentation, préférer les espaces.

➢Une seule instruction import par ligne, à placer en tête du fichier, dans l'ordre : bibliothèque standard, bibliothèques tierces, bibliothèques locales.

➢Eviter : *from module import **

➢Suivre les règles de nommage recommandées (voir http://www.python.org/dev/peps/pep-0008/).

➢Avant d'écrire une seule ligne de code, vérifier que cela n'existe pas déjà !

➢Les docstrings sont utiles pour un usage interactif (*help()*), ils expliquent comment utiliser le code et sont là pour les utilisateurs de votre code.

➢Pour tester des valeurs vraies ou des listes :

```
if x:

pass
```

```
if x == True:
    pass
```

```
if items:

pass
```

```
if len(items) != 0:
    pass
```

PLAN

I. Gestion de projet
    Etude des besoins, phases d'analyse, de conception, tests
II. Qualité logicielle
    Référentiels existants
III. Sécurité

IV. Règles de programmation
    Règles de nommage, fonctions, commentaires, mise en forme, spécificités de certains langages

V. Standards et normes
    Normes des langages, problématiques des flottants
VI. Analyse de code
    Analyse statique, analyse dynamique
VII. Métriques
    Les différentes métriques, les outils, la couverture de code

V Standards et normes

Quand il n'y a pas de norme, tout est permis :

• le langage BASIC en est un exemple flagrant, étroitement dépendant du compilateur pour lequel il est développé. L'apparition de standards ANSI 14 ans après la naissance du langage n'a pas suffi à modifier le comportement des utilisateurs.
• le langage C a aussi commencé avec une multiplication chaotique des compilateurs. Afin de remédier à ce chaos l'ANSI (American National Standards Institute) décida de fixer une norme, encore valide aujourd'hui : C ANSI.

La norme est une définition universellement valable d'un langage :
• définit comment le compilateur devra être fait ;
• définit les conditions d'application du compilateur au programme.

Quel que soit le système d'exploitation et le compilateur que l'on utilise, le programme écrit suivant une norme pourra fonctionner.

Essentielle à la portabilité des programmes.

Rétrocompatibilité

Il peut arriver qu'une norme devienne obsolète.

S'il est en général garanti que les versions ultérieures d'un interpréteur ou d'un compilateur supportent les versions antérieures de ses normes, une chose cependant est loin d'être assurée : ce que l'on appelle la rétrocompatibilité.

Exemple :

La norme est A ; j'écris un programme p(A) - un programme respectant la norme A - et le compile avec le compilateur c(A), qui supporte la norme A. Ça marche, et c'est normal.

La norme évolue, et passe à B ; j'écris un programme p(B) : un programme respectant la norme B, qui est une évolution de A ; je peux compiler p(B) avec le compilateur c(B), cela ne posera pas non plus de problème.

Comme les choses sont souvent bien faites, on pourra en général compiler p(A) avec c(B) : un interpréteur ou un compilateur gère presque toujours les versions antérieures de la norme (par exemple, le compilateur libre gcc comprend le C pré-ANSI) ;

mais peut-on compiler p(B) avec c(A) ?

Normes évolutives

Pour assurer une bonne rétrocompatibilité, deux principes doivent donc être observés :

- permettre à des formes évoluées de supporter des formes plus archaïques.
- laisser aux formes futures une marge suffisante pour supporter les versions actuelles.

La notion de norme est centrale dans les questions matérielles aussi bien que logicielles, car c'est elle qui permet la portabilité dans l'espace.

Mais tout aussi importante est la portabilité dans le temps : aussi est-il nécessaire que ces normes s'accompagnent de procédures ou de protocoles d'évolutivité, permettant aussi bien une compatibilité ultérieure qu'une rétrocompatibilité.

V Standards et normes

V-1 Normes de différents langages

Langage C

La version actuelle du langage est appelée C 99, deuxième édition de la norme internationale.
Elle est définie par :

- La norme initiale (ISO/IEC 9899:1999) ;
- Une première correction (ISO/IEC 9899:1999/Cor.1:2001) ;
- Une deuxième correction (ISO/IEC 9899:1999/Cor.2:2004) ;
- Une troisième correction (ISO/IEC 9899:1999/Cor.3:2007) ;

Lorsqu'on fait référence au langage C standard, c'est en tenant compte de ces trois correctifs.
Une nouvelle version est en cours d'élaboration : C 1x.

http://www.open-std.org/jtc1/sc22/wg14/

Langage C++

➢Première norme internationale de ce standard en 1998 (ISO/IEC 14882:1998), connue sous le nom de C ++98.
➢Version actuelle (ISO/IEC 14882:2003) sous l'acronyme C++ 03.
➢Dernière version (C++11). (2011-09-11)

http://open-std.org/JTC1/SC22/WG21/

Langage FORTRAN

Sa version actuelle, le FORTRAN 2008, correspond à la cinquième édition de la norme. C'est une révision mineure de la norme FORTRAN 2003. Elle a été publiée en juin 2010.

La définition du langage est divisée en trois parties qui évoluent indépendamment. Ainsi les parties 2 et 3 se fondent encore sur la norme précédente (FORTRAN 95).

Partie 1: Langage de base, avec corrections.

- La norme actuelle : ISO/IEC 1539-1:2010.

Partie 2: Chaînes de caractères de longueur variable :

- ISO/IEC 1539-2:2000.

Partie 3: Compilation conditionnelle :

- ISO/IEC 1539-3:1999.

V Standards et normes

V-2 Problème des nombres flottants

Précision limitée, qui se traduit par des arrondis (dus aux opérations, ainsi qu'aux changements de base implicites, si la base est différente de 10) qui peuvent s'accumuler de façon gênante. Par exemple, la soustraction de deux nombres très proches provoque une grande perte de précision relative.

Plage d'exposants limitée, pouvant donner lieux à des « overflows » (lorsque le résultat d'une opération est plus grand que la plus grande valeur représentable) et à des « underflows » (lorsqu'un résultat est plus petit, en valeur absolue, que le plus petit flottant normalisé positif), puis à des résultats n'ayant plus aucun sens.

Il est par exemple tentant de réorganiser des expressions en virgule flottante comme on le ferait d'expressions mathématiques. Cela n'est cependant pas anodin, car les calculs en virgule flottante ne sont pas associatifs.

Exemple : dans un calcul en flottants IEEE double précision, $(2^{60}+1)-2^{60}$ ne donne pas 1, mais 0. La raison est que $2^{60}+1$ n'est pas représentable exactement et est approché par 2^{60}.

Normes associées aux nombres flottants

La problématique des nombres flottants se situe à tous les niveaux :

- Type des langages
- Unité de calcul des processeurs
- Compilateurs
- Systèmes d'exploitation

Normes IEEE sur les nombres flottants.

La plus utilisée : norme IEEE 754, version actuelle : IEEE 754-2008, publiée en août 2008.

PLAN

I. Gestion de projet
    Etude des besoins, phases d'analyse, de conception, tests
II. Qualité logicielle
    Référentiels existants
III. Sécurité

IV. Règles de programmation
    Règles de nommage, fonctions, commentaires, mise en forme,spécificités de certains langages
V. Standards et normes
    Normes des langages, problématiques des flottants

VI. Analyse de code
    Analyse statique, analyse dynamique
VII. Métriques
    Les différentes métriques, les outils, la couverture de code

VI Analyse de code

Idée de base de l'analyse de code :utiliser l'ordinateur pour trouver les erreurs de programmation.

Analyse statique :

consiste à rechercher de manière automatique les erreurs dans le code source d'un programme sans l'exécuter.

Analyse dynamique :

réalisée sur un logiciel en l'exécutant sur un vrai processeur ou un processeur virtuel.

VI-1 Outils d'analyse statique

GCC/G++ / langages C/C++

Certaines options du compilateur permettent de faire des vérifications de type analyse statique :

-*Weffc++* : vérifie certaines des règles du livre "Effective and More Effective C++" de Scott Meyers.

-*Wextra* : effectue quelques analyses sur le code.

Splint / langage C

Analyse les vulnérabilités de sécurité (type buffer overflow …) et les erreurs de programmation.

Cppcheck / langages C/C++

Détecte les bugs et erreurs de programmation qui échappent au compilateur.

Ftncheck / langage FORTRAN

Supporte le FORTRAN 77 et quelques éléments du FORTRAN 90.

Forcheck / langage FORTRAN

Suite d'outils pour la vérification de programmes fortran du 66 au 2003. outil commercial.

RATS / langages C, C++, Perl, PHP et Python

"Rough Auditing Tool for Security“. Détecte les erreurs de sécurité classiques (buffer overflow ...).

FindBugs / langage JAVA

Analyse statique de code JAVA (bytecode ou code source), plugins disponibles pour Eclipse et Netbeans.

Pylint / langage Python

Analyse de code source Python pour détection de bugs et signes de mauvaise qualité du code.

VI-2 Outils d'analyse dynamique

Valgrind

Exécution des programmes sur un processeur virtuel, détections des problèmes mémoires, des situations de concurrences (multi-thread) ...

Electric Fence

Bibliothèque logicielle conçue pour faciliter le débogage de la gestion de la mémoire.

DUMA

Detect Unintended Memory Access. Bibliothèque logicielle permettant de détecter les dépassements mémoire dans les programmes C et C++. Fork d'Electric Fence.

Purify

Outil commercial de deboggage mémoire, particulièrement pour les programmes C et C++.

PLAN

I. Gestion de projet
Etude des besoins, phases d'analyse, de conception, tests
II. Qualité logicielle
Référentiels existants
III. Sécurité

IV. Règles de programmation
Règles de nommage, fonctions, commentaires, mise en forme, spécificités de certains langages
V. Standards et normes
Normes des langages, problématiques des flottants
VI. Analyse de code
Analyse statique, analyse dynamique
☞ **VII. Métriques**
Les différentes métriques, les outils, la couverture de code

VII Métriques

Permettent de quantifier la complexité d'un logiciel

VII-1 Différentes métriques

Métriques de taille

- nombre de lignes physiques (total des lignes des fichiers source);
- nombre de lignes de code (déclarations, définitions, directives et instructions) ;
- nombre de lignes de commentaires;
- nombre de lignes vides.

Métriques de structure

- **Nombre cyclomatique de Mc Cabe : v(G)**

Nombre de chemins linéaires indépendants dans un module de programme et donc complexité des flux de données.
Les constructions de langage suivants incrémentent le nombre cyclomatique :
if (...), for(...), while (...), case ...:, catch (...), &&, ||, ?, #if, #ifdef, #ifndef, #elif.

- **Métrique de Halstead**

Les métriques de complexité de Halstead procurent une mesure quantitative de complexité du code. Elles sont dérivées du nombre d'opérateurs et d'opérandes.

VII-2 Outils d'évaluation des métriques

Testwell CMT++

Outil commercial de mesure de complexité des codes pour C et C++. Il existe une version pour les codes JAVA : Testwell CMTJava.

Eclipse Metrics Plugin

Evaluation des métriques et analyse des dépendances de programme JAVA sous la forme d'un plugin Eclipse.

UCC, Unified CodeCount

Outil d'évaluation des métriques de type Lignes de Codes pour de nombreux langages (dont C/C++, JAVA et Python).

VII-3 Outils de couverture de code

La couverture d'un code (code coverage) est une mesure utilisée en génie logiciel pour décrire le taux de code source testé d'un programme.

Fcat

FORTRAN Coverage Analysis Tool. Permet l'analyse de la couverture des codes fortran, notamment les points chauds (parties des codes les plus souvent exécutées) et les points froids (parties des codes jamais exécutées).

Gcov

A utiliser avec les compilateurs GNU. Peut être utilisé avec l'outil de profiling gprof pour une analyse complète.

Cobertura

Outil d'évaluation de la couverture de code JAVA.

FIN !!

Merci pour votre attention

Des questions ?

Bibliographie

- Bonnes pratiques en informatique: les référentiels
  - www.guideinformatique.com/fiche-bonnes_pratiques_lesreferentiels-740.htm
- Tout sur le code : Pour concevoir du logiciel de qualité
  - Steve McConnell , adapté par Guy Le Doré 2e édition – Dunod Février 2005
  - Code complete (Steve McConnell : http://cc2e.com/Default.aspx
- Les fondements du développement agile : Laurent Desmons
  - http://www.dotnetguru.org/articles/dossiers/devagile/DeveloppperAgile.htm
- Software Program Manager's Network : http://www.spmn.com
- Introduction au RUP : P Kruchten – Eyrolles (2000)
- Cours méthodes agiles : http://liris.cnrs.fr/~yprie/ens/05-06/SIMA-M1-S1/CM-methodes-fin-6pp.pdf
- Rôle des normes : http://www.numeraladvance.com/Role_des_Normes/Normes_pour_SI/ISO_12207/L_ISO_12207.htm
- OASIS : http://www.oasis-open.org/home/index.php

Règles de programmation

• MISRA C est une norme de programmation en langage C créée en 1998 par le Motor Industry Software Reliability Association (MISRA). http://en.wikipedia.org/wiki/MISRA_C

• HIGH-INTEGRITY C++ CODING STANDARD MANUAL (HICCSM), liste de règles basées sur la norme ISO/IEC C++ 14882. http://www.codingstandard.com/HICPPCM/index.html

• NASA Software Safety Guidebook : http://www.hq.nasa.gov/office/codeq/software/docs.htm

• Conventions de codage java : http://java.sun.com/docs/codeconv

Bibliographie

Standards et normes

Standards ouverts : http://open-std.org/

Normes C++ : http://www.open-std.org/jtc1/sc22/wg21/, nouvelle version : http://fr.wikipedia.org/wiki/C%2B%2B0x

Normes C : http://www.open-std.org/jtc1/sc22/wg14/. Nouvelle version : http://en.wikipedia.org/wiki/C1X

Fortran : http://www.nag.co.uk/sc22wg5/. Nouvelle version : http://fortranwiki.org/fortran/show/Fortran+2008

Normes IEEE 754 : http://grouper.ieee.org/groups/754/
Papier très intéressant sur le problème des nombres flottants : http://www.validlab.com/goldberg/paper.pdf

Analyse statique de code

http://en.wikipedia.org/wiki/Lint_%28software%29
http://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html
http://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Dialect-Options.html
http://www.splint.org/
http://sourceforge.net/apps/mediawiki/cppcheck/index.php?title=Main_Page
http://www.dsm.fordham.edu/~ftnchek/
http://www.forcheck.nl/
https://www.fortify.com/ssa-elements/threat-intelligence/rats.html
http://findbugs.sourceforge.net/
http://www.logilab.org/857

Analyse dynamique de code

http://valgrind.org/
http://directory.fsf.org/project/ElectricFence/
http://duma.sourceforge.net/
http://www-01.ibm.com/software/awdtools/purify/

Evaluation de métriques

http://www.verifysoft.com/fr_cmtx.html
http://metrics.sourceforge.net/
http://sunset.usc.edu/research/CODECOUNT/
http://www.chris-lott.org/resources/cmetrics/

Couverture de code

http://www2.research.att.com/~yifanhu/SOFTWARE/FCAT/
http://gcc.gnu.org/onlinedocs/gcc/Gcov.html
http://cobertura.sourceforge.net/

Exemple d'utilisation d'une assertion en java

exempleAssert.java

```
/*
objectif = montrer une utilisation simple de assert
source: cours java SUN
*/
import java.io.*;

public class exempleAssert
{
    public static void main( String args[] )
    {
     assert (args.length >= 2 && args.length <= 4) : "Trop de paramètres (" + args.length + ")";
     System.out.println( "Vous donnez " + args.length  + " arguments \n");
     for (int i = 0; i < args.length; i++)
      System.out.print("Arg " + i + " = " + args[i] + "; ") ;
     System.out.println(" ") ;
     } // end main
} // end class exempleAssert
```

JDK1.4.2 sous Windows

```
javac   -source 1.4   exempleAssert.java


java -ea -cp . exempleAssert 1 2 3 4 5 6
Exception in thread "main" java.lang.AssertionError: Trop de paramÞtres (6)
        at exempleAssert.main(exempleAssert.java:12)


java  -cp . exempleAssert 1 2 3 4 5 6
Vous donnez 6 arguments

Arg 0 = 1; Arg 1 = 2; Arg 2 = 3; Arg 3 = 4; Arg 4 = 5; Arg 5 = 6;
```