Qualité des logiciels

Mme Soumia ZITI_LABRIM
s.ziti@fsr.ac.ma
Université Mohamed V
Faculté des Sciences
Département Informatique

Plan

- Introduction et définitions
- Critères de qualité logiciel
- Assurance qualité logiciel
- Cycle de vie de développement logiciel
- Test du logiciel
- Gestion de configuration du logiciel
- Normes de développement

Introduction et définitions

Génie logiciel :

Décrit l'ensemble de méthodes (spécification, conception, test…) , techniques et outils (Atelier de Génie logiciel) nécessaires à la production de logiciels de qualité industrielle.

Objectifs de génie logiciel :

Satisfaire les utilisateurs, en produisant des logiciels adaptés aux besoins ; les gestionnaires, en réduisant les coûts de maintenance et les chefs de projet, en rendant les logiciels productifs dans un délai raisonnable.

Risque de logiciel :

Diffusion massive en pilotage, système sécuritaire, industrie nucléaire (réservation de ticket, pilotage aéronautique, diagnostique médicale, opération financière …)

Introduction et définitions

Qualité du logiciel

Déf. intuitive : La qualité est la conformité avec les besoins, l'adéquation avec l'usage attendu, le degré d'excellence ou , tout simplement, la valeur de quelque chose pour quelqu'un

Déf. ISO : Ensemble des traits et des caractéristiques d'un produit logiciel portant sur son aptitude à satisfaire des besoins exprimés ou implicites

Déf. IEEE : La qualité du logiciel correspond au degré selon lequel un logiciel possède une combinaison d'attributs désirés.

Déf. de Crosby : La qualité du logiciel correspond au degré selon lequel un client perçoit qu'un logiciel réponde aux multiples attentes.

Déf. de Pressman : Conformité aux exigences explicites à la fois fonctionnelles et de performances, aux standards de développements explicitement documentés et aux caractéristiques implicites qui sont attendues de tous logiciels professionnellement développés

Introduction et définitions

Différents points de vue

	Utilisateur	Vendeur	Gestionnaire de projet	Programmeur	Professeur
Facilité d'utilisation	X				
Compatibilité	X				
Multiple fonctions	X	X		X	
Haute performance	X	X			
0 défauts	X		X		
Développement rapide	X	X	X		
Faibles coûts de développement			X		
Code élégant				X	X

Introduction et définitions

Logiciel :

- Un ensemble des documents de gestion de projet,
- Une spécification décrivant la liste des fonctions à remplir par le logiciel, les facteurs qualité du logiciel (portabilité, évolutivité, robustesse, . . .), les contraintes (performances temporelles et spatiales, . . .) et les interfaces avec son environnement
- Une conception décrivant la découpe de la spécification en modules (ou objets), la description les interfaces entre ces modules (ou objets) et la description des algorithmes mis en place
- Un code source et un exécutable

Produits :

Programmes sources et machines, des procédures et des ensembles de données enregistrées.

Client et Fournisseur :

Le client commande un logiciel, le fournisseur le réalise.

Introduction et définitions

Caractéristiques de logiciel

Produit unique

Conçu et fabriqué une seule fois, reproduit

Inusable

Défauts pas dus à l'usure mais proviennent de sa conception

Complexe

Le logiciel est fabriqué pour soulager l'humain d'un problème complexe ; il est donc par nature complexe

Invisible

Fabrication du logiciel = activité purement intellectuelle

Difficulté de perception de la notion de qualité du logiciel

Technique non mature

Encore artisanal malgré les progrès

Introduction et définitions

Plan de développement

- La description du logiciel à réaliser en différents niveaux de produits (programmes et documents)
- Les moyens matériels et/ou logiciel à mettre à disposition ou à réaliser (Méthodes, Techniques, Outils)
- Le découpage du cycle de vie en phases, la définition des tâches à effectuer dans chaque phase et l'identification des responsables associés
- Les supports de suivi de l'avancement (Planning et calendriers)
- Les moyens utilisés pour gérer le projet
- Les points clés avec ou sans intervention du client

Introduction et définitions

Constat sur le développement de logiciel

- Le coût du développement du logiciel dépasse souvent celui du matériel,
- Les coûts dans le développement du logiciel :
  - 20% pour le codage et la mise au point individuelle
  - 30% pour la conception
  - 50% pour les tests d'intégration
- Durée de vie d'un logiciel de 7 à 15 ans
- Le coût de la maintenance évolutive et corrective constitue la part prépondérante du coût total
- Plus de la moitié des erreurs découvertes en phases de tests proviennent d'erreurs introduites dans les premières étapes
- La récupération d'une erreur est d'autant plus coûteuse que sa découverte est tardive

Introduction et définitions

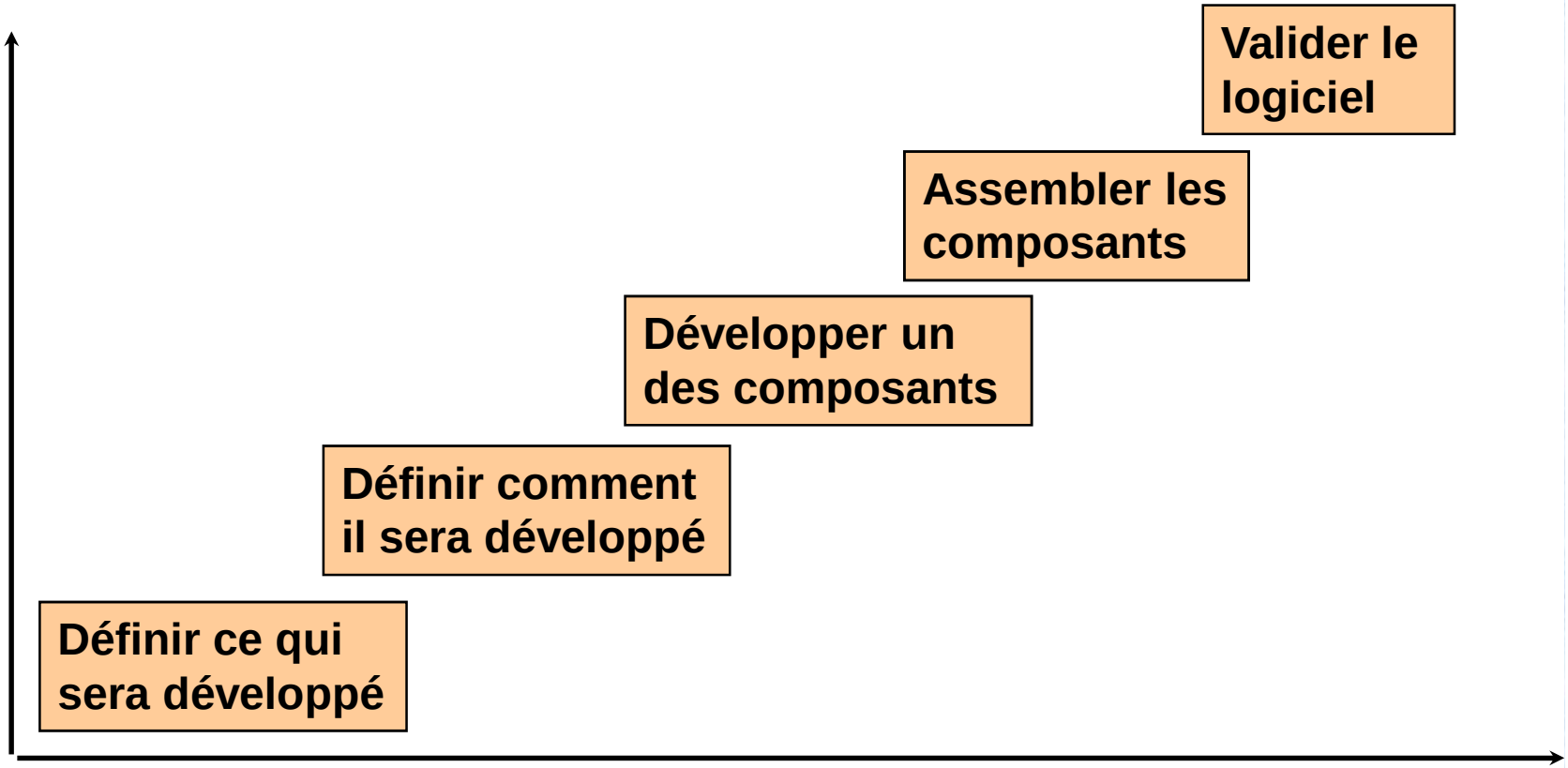
Etapes de développement de logiciel

- **Analyse (du problème)**
  - comprendre et recenser les besoins
  - Spécification (par exemple cahier des charges)
- **Conception (du logiciel)**
  - **Préliminaire**
    - éclater le logiciel en sous-parties
    - définir les interfaces entre ces sous-parties
    - Définir l'architecture du logiciel
  - **Détaillée**
    - préciser l'architecture des sous-parties
- **Implantation**

  - codage, intégration, tests

Introduction et définitions

Activités de développement de logiciel



- L'organisation de ces activités et leur enchaînement définit le *cycle de développement* du logiciel

Introduction et définitions

Importance de qualité de logiciel

- Le logiciel est une composante majeure des systèmes informatiques (environ 80% du coût) utilisés pour :
  - communication (ex. syst. téléphone, syst. email)
  - santé (monitoring)
  - transport (ex. automobile, aéronautique)
  - échanges économiques (ex. commerce)
  - Entertainment
  - ...
- Les défauts du logiciel sont extrêmement coûteux en terme
  - d'argent
  - de réputation
  - de perte de vie

Introduction et définitions

Facteurs de non qualité de logiciel

- **Mauvaise spécification :**
  - Vague, incomplète, instables…
- **Mauvaise estimation :**
  - Fausse, oublis, précisions insuffisantes…
- **Mauvaise répartition des tâches :**
  - Organisation inadaptée, contraintes omises
  - Ecart non détecté à temps
- **Mauvaise réalisation technique**
  - Codage, tests, documentation
- **Problèmes humains**
  - Mauvaise distribution des travaux
  - Conflits, rétention d''information
- **Manque d'expérience du métier de chef de projet**

Introduction et définitions

Exemples de désastres historiques

- 1988 abattage d'un Airbus 320 par l'USS Vincennes : affichage cryptique et confusion du logiciel de détection
- 1991 échec de missile patriot : calcul imprécis de temps dû à des erreurs arithmétiques
- London Ambulance Service Computer Aided Despatch System : plusieurs décès
- Le 3 Juin 1980, North American Aerospace Defense Command (NORAD) rapporta que les U.S. étaient sous attaque de missiles
- Echec du premier lancement opérationnel de la navette spatiale dont le logiciel d'exploitation temps réel consiste en environ 500,000 LOC : problème de synchronisation entre les ordinateurs de contrôle de vol
- Réseau téléphonique longue distance d'AT&T : Panne de 9 heures provoqué par un patch de code non testé

Introduction et définitions

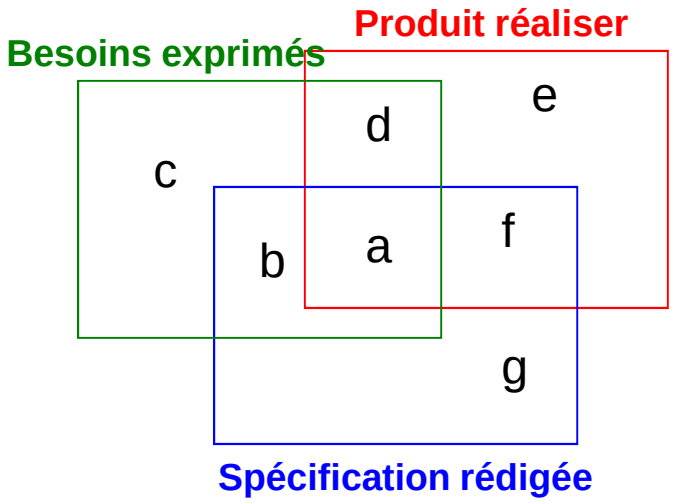
Exemple type : Ariane 5

- C'est le lanceur successeur de Ariane 4 pour le lancement de satellites en orbite terrestre.
- 37 seconde après décollage réussi, elle explose en vol
- Des information incorrectes avaient été transmises mettant le système de contrôle en défaut
- L'expertise a montré que le défaut provenait de la tentative de conversion d'un nombre flottant 64-bit en entier signé 16-bit et qu'il n' y avait pas de gestion d'exceptions implantées
- Cette partie provenait d'Ariane 4 et du fait de l'accélération et de la vitesse moins importante, le problème n'était jamais apparu ni en vol réel ni en test

Introduction et définitions

Problème de conformité aux besoins

a. Satisfaction
b. Non-conformité
c. Insatisfaction client
d. Satisfaction hasardeuse
e. Prestations inutiles hors spécification
f. Spécifications et prestations inutiles
g. Spécification inutiles



Critères de qualité logiciel

ISO/IEC 9126 propose 6 caractéristiques de qualité du produit logiciel

- **Capacité fonctionnelle** (*Functionality*)
- **Fiabilité** (*Reliability*)
- **Facilité d'utilisation** (*Usability*)
- **Rendement** (*Efficiency*)
- **Maintenabilité** (*Maintainability*)
- **Portabilité** (*Portability*)

Critères de qualité logiciel

Capacité fonctionnelle

Définition

Ensemble d'attributs portant sur l'existence d'un ensemble de fonctions et leurs propriétés. Les fonctions sont celles qui satisfont aux besoins exprimés ou implicites

Sous-caractéristiques

- **Aptitude :** présence et adéquation d'une série de fonctions pour des tâches données
- **Exactitude :** fourniture de résultats ou d'effets justes ou convenus
- **Interopérabilité :** capacité à interagir avec des systèmes donnés
- **Sécurité :** aptitude à empêcher tout accès non autorisé (accidentel ou délibéré) aux programmes et données

Critères de qualité logiciel

Fiabilité

Définition

Ensemble d'attributs portant sur l'aptitude du logiciel à maintenir son niveau de service dans des conditions précises et pendant une période déterminée

Sous-caractéristiques

- **Maturité :** fréquence des défaillances dues à des défauts du logiciel
- **Tolérance aux fautes :** aptitude à maintenir un niveau de service donné en cas de défaut du logiciel ou de violation de son interface
- **Possibilité de récupération :** capacité à rétablir son niveau de service et de restaurer les informations directement affectées en cas de défaillance ; temps et effort nécessaire pour le faire

Critères de qualité logiciel

Facilité d'utilisation

Définition

Ensemble d'attributs portant sur l'effort nécessaire pour l'utilisation et l'évaluation individuelle de cette utilisation par un ensemble défini ou implicite d'utilisateurs

Sous-caractéristiques

- **Facilité de compréhension :** effort que doit faire l'utilisateur pour reconnaître la logique et sa mise en œuvre
- **Facilité d'apprentissage :** effort que doit faire l'utilisateur pour apprendre son application
- **Facilité d'exploitation :** effort que doit faire l'utilisateur pour exploiter et contrôler l'exploitation de son application

Critères de qualité logiciel

Rendement

Définition

Ensemble d'attributs portant sur le rapport existant entre le niveau de service d'un logiciel et la quantité de ressources utilisées, dans des conditions déterminées

Sous-caractéristiques

- **Comportement vis-à-vis du temps :** temps de réponses et de traitement ; débits lors de l'exécution de sa fonction
- **Comportement vis-à-vis des ressources :** quantité de ressources utilisées ; durée de leur utilisation lorsqu'il exécute sa fonction

Critères de qualité logiciel

Maintenabilité

Définition

Ensemble d'attributs portant sur l'effort nécessaire pour faire des modifications données

Sous-caractéristiques

- **Facilité d'analyse :** effort nécessaire pour diagnostiquer les déficiences et causes de défaillance ou pour identifier les parties à modifier
- **Facilité de modification :** effort nécessaire pour modifier, remédier aux défauts ou changer d'environnement
- **Stabilité :** risque des effets inattendus des modifications
- **Facilité de test :** effort nécessaire pour valider le logiciel modifié

Critères de qualité logiciel

Portabilité

Définition

Ensemble d'attributs portant sur l'aptitude du logiciel à être transféré d'un environnement à l'autre

Sous-caractéristiques

- **Facilité d'adaptation :** possibilité d'adaptation à différents environnements donnés sans que l'on ait recours à d'autres actions ou moyens que ceux prévus à cet effet pour le logiciel considéré
- **Facilité d'installation :** effort nécessaire pour installer le logiciel dans un environnement donné
- **Conformité aux règles de portabilité :** conformité aux normes et aux conventions ayant trait à la portabilité
- **Interchangeabilité :** possibilité et effort d'utilisation du logiciel à la place d'un autre logiciel donné dans le même environnement

Critères de qualité logiciel

Compromis de qualité

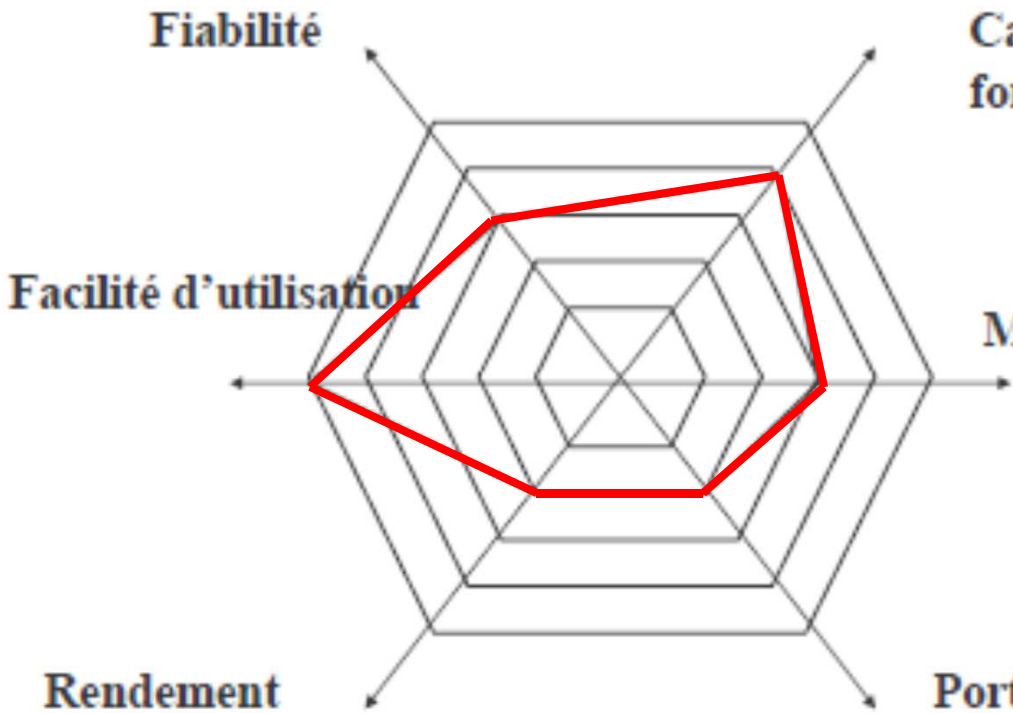
- Les critère de qualité peuvent être contradictoires, pour chaque, le choix des compromis doit s'effectuer en fonction du contexte.
  - Facilité d'emploi **vs** maintenabilité
  - Fiabilité **vs** portabilité
  - Capacité **vs** rendement

- La qualité s'obtient par la mise en place de procédure et des méthodes de phase de spécification

- La démarche qualité ne consiste pas à corriger les défauts mais à les prévenir

Critères de qualité logiciel

- **Schéma de compromis de qualité**



Assurance qualité logiciel

Définition de l'AQL

- Un modèle planifié et systématique de toutes les actions nécessaires pour fournir une confiance adéquate qu'un article ou un produit est conforme à ses exigences techniques établies afin d'obtenir de la qualité requise.
- Un ensemble d'activités conçu pour évaluer le processus par lequel les produits sont développés ou fabriqués.

➔Mise en œuvre d'une approche préventive de la qualité : L'AQ consiste en un ensemble d'actions de prévention des défauts qui accompagnent le processus de développement des artefacts logiciels.

- Passe par l'élaboration d'un MANUEL QUALITE

Assurance qualité logiciel

Objectifs de l'AQL :

- Assurer un niveau de confiance acceptable que le logiciel sera conforme aux exigences fonctionnelles techniques.
- Assurer un niveau de confiance acceptable que le logiciel sera conforme aux exigences de gestion concernant l'échéancier et le budget.
- Initier et gérer des activités visant à l'amélioration et la plus grande efficience des activités de développement et d'assurance de qualité logicielle.
- Initier et gérer des activités visant à l'amélioration et à l'augmentation de l'efficience des activités de maintenance et d'assurance de qualité logicielle

Assurance qualité logiciel

Contrôle de la qualité

- Ensemble des processus et méthodes pour contrôler le travail et observer si les exigences sont satisfaites.
- La sélection des caractéristiques d'évaluation ainsi que la spécification des techniques d'évaluation utilisée
- Les revues et la suppression des défectuosités avant la livraison du produit.

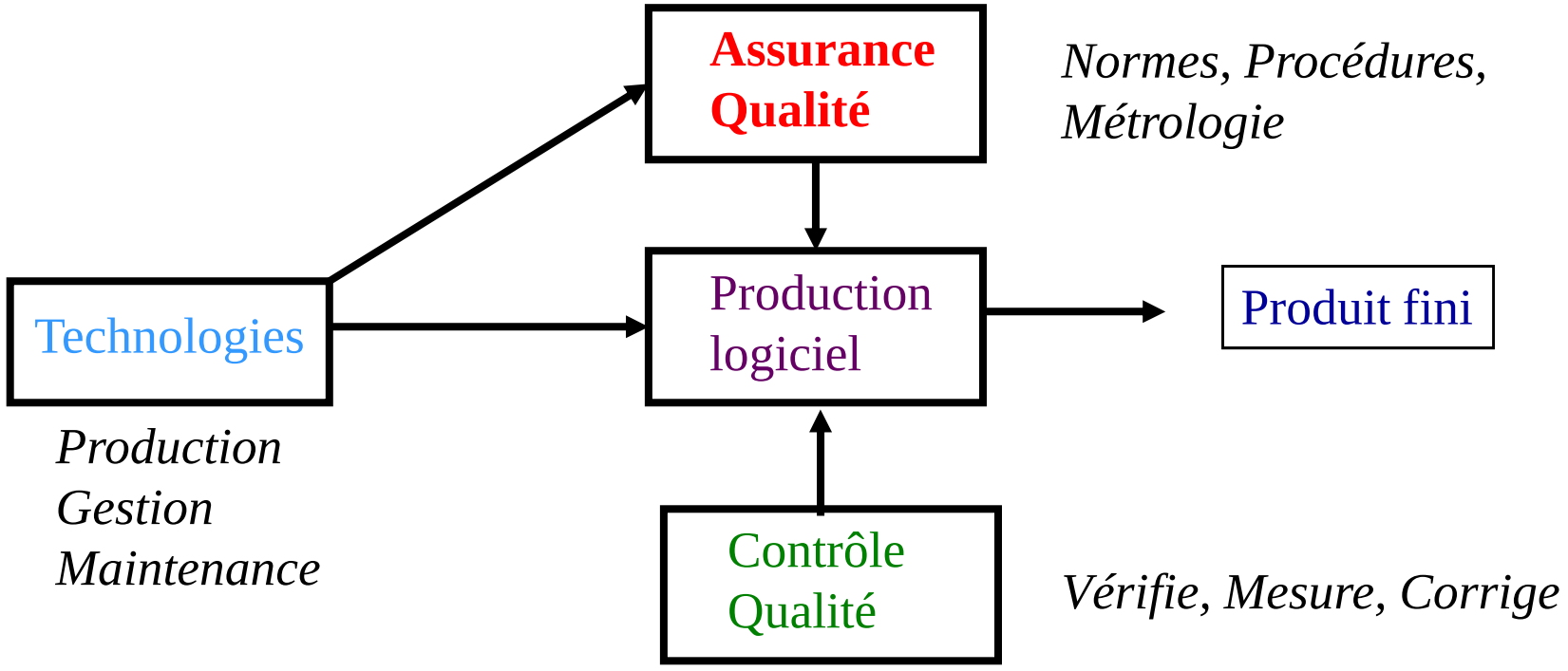
- La **validation** **:** effectuée à la fin du processus de développement pour assurer la conformité aux exigences du produit

➔ **répondre à la question "sommes nous en train de faire le bon produit?,**

- La **vérification** : effectuée à la fin d'une phase pour s'assurer que des besoins établis pendant la phase précédente ont été répondus

➔ **répond à la question "est ce que nous faisons le produit correctement"**

Assurance qualité logiciel



Assurance qualité logiciel

Directives complémentaires

- Les directives d'identification, d'implantation et d'analyse des métriques nécessaires au processus d'évaluation du produit final,
- Les directives de définition des indicateurs qui permettent des évaluations partielles pendant le cycle de développement
- **But :**
  - Les informations générales sur les indicateurs de qualité des logiciels ;
  - Les critères de sélection de ces indicateurs
  - Les directions pour l'évaluation des données de mesurage
  - Les directions pour l'amélioration du processus de mesurage
  - Les exemples de types de graphes d'indicateurs

Assurance qualité logiciel

Manuel qualité :

- Document décrivant les dispositions générales prises par l'entreprise pour obtenir la qualité de ses produits ou de ses services
- **Rôle** : Usage interne et externe
- **Organisé en 6 parties:**
  - Organisation de l'entreprise
  - Activités de production et de contrôle technique
  - Activité de gestion
  - Activité de contrôle de la qualité
  - Plan type du PLAN QUALITE
  - Lignes directrices permettant d'établir le plan qualité

Assurance qualité logiciel

Utilisation du manuel qualité :

- **Communiquer** la politique, les procédures et les exigences
- **Mettre en œuvre** un système effectif;
- Fournir une **maîtrise améliorée** des pratiques et faciliter les activités relatives à l'assurance;
- Fournir les **bases documentaires** pour auditer les systèmes qualité;
- Assurer la **continuité** du système qualité et de ses exigences en cas de modification des circonstances;
- **Former le personnel** aux exigences de système qualité et aux méthodes relatives à la conformité
- Présenter leur système qualité pour des usages externes, tel que la **démonstration** de leur conformité aux normes
- Démontrer la conformité de leurs systèmes qualité avec les normes relatives à la qualité dans des **situations contractuelles**.

Assurance qualité logiciel

Plan qualité logiciel :

- Document décrivant les dispositions spécifiques prises par une entreprise pour obtenir la qualité du produit ou du service considéré

- Il comprend les techniques et les activités à caractère opérationnel qui ont pour but à la fois de piloter un processus et d'éliminer les causes de fonctionnement **non satisfaisant à toutes les phases de la boucle de la qualité** en vue d'atteindre la meilleure efficacité économique

- Il incluse : **le produit** auquel il est prévu d'être appliqué, les **objectifs** relatifs à la qualité du produit (exprimer ces objectifs en termes mesurables chaque fois que cela est possible), les **exclusions** spécifiques ainsi que la **période de validité**

Assurance qualité logiciel

Plan type de qualité logiciel :

1)But, Domaine d'application et responsabilité:

2) Documents applicables et documents de références:

3) Terminologie

4) Organisation:

5)Démarche de développement

6)Documentation

7) Gestion des configurations

8) Gestion des modifications

9) Méthodes, outils et règles

10) Contrôle des fournisseurs

11) Reproduction, protection, livraison

12) Suivi de l'application du plan qualité (plan de contrôle)

Assurance qualité logiciel

Principes généraux de l'AQL

- **La définition des exigences de qualité** : interpréter le contexte
- **La préparation des processus d'évaluation** : Planifier les activités de contrôle qualité
- **L'exécution de la procédure d'évaluation** : réaliser l'évaluation du logiciel avant et après l'exécution

Assurance qualité logiciel

La définition des exigences de qualité

- Avoir des exigences et spécifications explicites
- Avoir un processus de développement logiciel avec
  - Analyse des exigences,
  - Tests d'acceptabilité,
  - Feedback fréquent des usagers

Assurance qualité logiciel

La préparation de processus d'évaluation.

À ce niveau, l'objectif est d'initier l'évaluation et de mettre au point ses bases. Ceci est fait en trois sous-étapes :

- **La sélection des métriques de qualité**. Ces dernières doivent correspondre aux caractéristiques énumérées plus haut.
- **La définition des taux de satisfaction**. Les échelles de valeurs doivent être divisées en portions correspondant aux niveaux de satisfaction des exigences.
- **La définition des critères d'appréciation**. Ceci inclut la préparation de la procédure de compilation des résultats par caractéristique. Il est possible aussi de prendre en compte dans cette procédure des aspects de gestion tels que le temps ou les coûts.

Assurance qualité logiciel

L'exécution de la procédure d'évaluation

- Il s'agit de donner un cadre rigoureux pour :
  - Guider le développement du logiciel, de sa conception à sa livraison.
  - Contrôler les coûts, évaluer les risques et respecter les délais.
- **Quatre méthodes complémentaires:**
  - **Méthodes formelles** : Vérifier mathématiquement des propriétés spécifiées
  - **Métriques** : Mesurer un ensemble connu de propriétés liées à la qualité
  - **Revues et Inspections** : Relecture et examen par humain des exigences, design, code, ... basés sur des checklists
  - **Tests** : Données explicites pour exécuter le logiciel et vérifier si les résultats correspondent aux attentes

Assurance qualité logiciel

Méthodes formelles

- **Preuves de**

  * Correction : L'expression formelle des spécifications

  * Terminaison de l'exécution : La production de la preuve d'une conception et d'une implémentation sans erreurs

  * Propriétés spécifiques : L'explicitation précise des propriétés d'architecture de développement

- **Assistants de preuve (theorem prover)**

  * systèmes : Coq, PVS, Isabelle / HOL, ...

  * Encore difficiles d'usage pour des non spécialistes

Assurance qualité logiciel

Métriques

- Mesures des processus (les séries d'activités reliées au développement du logiciel), des produits (les objets produits, livrables ou documents qui résultent d'une activité d'un processus), des ressources (les entités exigées par une activité d'un processus)
- **Mesure des caractéristiques « internes »**

mesures objectives : taille, complexité du flot de contrôle, cohésion modulaire / couplage entre modules, ...

- **Mesure des caractéristiques « externes »**

évaluations stochastiques (statistiques) : délai moyen de réponse à une requête, nombre de requêtes simultanées sans « écrouler » un serveur, ...

Ce sont des mesures **postérieurs** : arrivent parfois « trop tard »

Assurance qualité logiciel

Métriques de MacCabe

- Analyse du graphe de contrôle
- Mesure de la complexité structurelle ;
- **Nombre cyclomatique :**

Représente le nombre de chemins indépendants

$$C = a - n + i + s$$

n : nb de nœuds (blocs d'instructions séquentielles)

a : nb d'arcs (branches de programme)

i : nb de nœud d'entrée

s : nb de nœud de sortie

Plus le nombre cyclomatique est grand, plus il y aura de chemins d'exécution dans la fonction, et plus elle sera difficile a comprendre.

Assurance qualité logicielse

Métriques de MacCabe

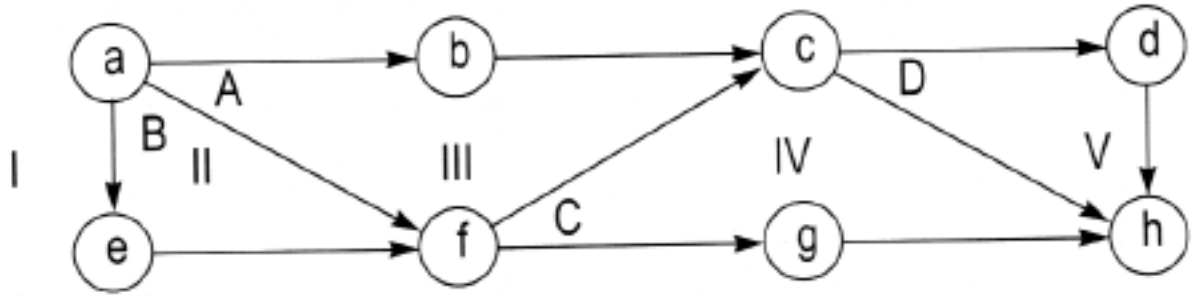
Calcul direct : MacCabe a introduit une nouvelle manière de calculer la complexité structurelle

C = Pi+ 1 avec Pi le nombre de décisions du code

- Une instruction **IF** compte pour 1 décision
- Une boucle **FOR** ou **WHILE** compte pour 1 décision
- Une instruction **CASE** ou tout autre embranchement multiple compte pour une décision de moins que le nombre d'alternatives

Le nombre de faces du graphe donne la complexité structurelle du code

Exemple **C=4+1= 5**



Assurance qualité logiciel

Métriques de MacCabe

- Le nombre cyclomatique donne une évaluation du **nombre des chemins indépendants** dans le graphe et donc une indication sur le **nombre de tests nécessaires**
- Cette métrique indique la borne supérieure du nombre de tests à effectuer pour que tous les arcs soient couverts au moins une fois.
- Dans la pratique il semble que la limite supérieure du nombre cyclomatique soit de 30 environ.
- La **valeur maximum du nombre cyclomatique** peut être définie comme un **critère de qualité** dans le plan qualité.

Assurance qualité logiciel

Métriques de MacCabe

- La complexité cyclomatique d'une méthode **augmente** proportionnellement au **nombre de points de décision**. Une méthode avec une haute complexité cyclomatique est plus difficile à comprendre et à maintenir.
- Une complexité cyclomatique **trop élevée** (supérieure à 30) indique qu'il faut **refactoriser** la méthode.
- La complexité cyclomatique est liée à la notion de **"code coverage",** c'est à dire la couverture du code par les tests. Dans l'idéal, une méthode devrait avoir un nombre de tests unitaires égal à sa complexité cyclomatique pour avoir un "code coverage" de 100%. Cela signifie que **chaque chemin de la méthode a été testé**.

Assurance qualité logiciel

Métriques de MacCabe – Exercice 1

Calculer le nombre cyclomatique du programme suivant

```
1  void sort(int *px, int n){
2          int i, j, temp;
3          for(i = 2; i < n; i++) {
4                  for( j = 1; j < i; j++) {
5                          if( px[ i ] < px[ j ] ) {
6                                  temp = px[ i ];
7                                  px[ i ] = px[ j ];
8                                  px[ j ] = temp;
9                          }
10                 }
11         }
12}
```

Assurance qualité logiciel

Métriques de MacCabe – Exercice 2

soit le programme « recherche dichotomique » en langage C:

```
void recherche_dico (elem cle, elem t[], int taille, boolean &trouv, int &A)
{ int d, g, m;
        g=0; d=taille -1;
        A=(d+g) /2;
        if (t[A]= =cle) trouv=true;
        else trouv=false;
        while (g <=d && !trouv)
                { m= (d+g) /2;
                        if (t[m]= =cle)
                        {       trouv=true;
                                A=m;
                        }
                        else if (t[m]> cle) g=m+1;
                                else d=m-1;
                }
}
```

Calculer le nombre cyclomatique de cette procédure.

Assurance qualité logiciel

Métriques de Halstead

- Complexité liée à la **distribution** des variables et instructions.
- Fondées **empiriquement** et toujours considérées comme valides, contrairement a ses formules complexes de prédiction
- **Métrique textuelle** pour évaluer la taille d'un programme.
- **Alternative** au calcul du nombre de lignes de code source.
- **Entités de base** : **Opérandes** (jetons qui contiennent une valeur) et

Operateurs (tout le reste :virgules, parenthèses, operateurs arithmétiques...)

Assurance qualité logiciel

Métriques de Halstead

- La base des mesures est fournie par le vocabulaire utilisé. On évalue le **nombre d'opérateurs et d'opérandes.**
  - **n1** = nombre **d'opérateurs** uniques
  - **n2** = nombre **d'opérandes** uniques (termes, constantes , variables)
  - **N1** = nombre total d'apparition de ces opérateurs
  - **N2** = nombre total d'apparition de ces opérandes.
- **Exemple**

  a := a + 1;
  - 3 opérateurs ➔ + := ;
  - 2 opérandes ➔ a 1

Assurance qualité logiciel

Métriques de Halstead

- **Mise en œuvre** : « Une fois que le code a été écrit, cette mesure peut être appliquée pour prédire la difficulté d'un programme et d'autres quantités, en employant les équations de Halstead :
- **Vocabulaire du programme** $I = n1 + n2$
- **Taille observée du programme** $L = N1 + N2$
- **Taille estimée du programme** $Le = n1(Log2\ n1) + n2\ (Log2\ n2)$
- **Volume du programme :**Estimation du nombre de bits nécessaires pour coder le programme mesuré

$$V = L\ Log2\ (n1 + n2)$$

Assurance qualité logiciel

Métriques de Halstead

- **Difficulté du programme :** propension d'erreurs du programme

$$D = (n1/2)\ (N2/n2)$$

- **Niveau du programme** $L1 = 1/D$
- **Effort :** Mesure de l'effort intellectuel nécessaire a l'implémentation d'un algorithme

$$E = V/L1$$

- **Nombre d'erreurs :** une estimation du nombre d'erreurs dans le programme

$$B = E^{2/3}/3000$$

- **Temps :** Temps nécessaires pour implémenter un algorithme, où S est le nombre de Stroud (=18 pour Halstead)

$$T = E/S$$

Assurance qualité logiciel

Métriques de Halstead – Exercice 1

- **Calculer les mesures de Halstead pour le programme suivant :**

```
z = 0;
while x > 0
    z = z + y;
    x = x - 1;
end-while
print (z);
```

Opérandes		Opérateurs	
=	3	z	4
;	5	0	2
w/ew	1	x	3
>	1	y	2
+	1	1	1
-	1		
print	1		
()	1		

Assurance qualité logiciel

Métriques de Halstead - Exercice 2

- **Calculer les mesures de Halstead pour le programme suivant :**

```
1  void sort(int *px, int n){
2          int i, j, temp;
3          for(i = 2; i < n; i++) {
4                  for( j = 1; j < i; j++) {
5                          if( px[ i ] < px[ j ] ) {
6                                  temp = px[ i ];
7                                  px[ i ] = px[ j ];
8                                  px[ j ] = temp;
9                          }
10                 }
11         }
12}
```

Assurance qualité logiciel

Métriques de Henry-Kafura

- Mesurer la **complexité des modules** d'un programme en fonction des **liens** qu'ils entretiennent
- On utilise pour chaque **module i** :
  - Le nombre de **flux** d'information **entrant** noté **in_i**
  - Le nombre de **flux** d'information **sortant** noté **out_i**
  - Le **poids du module** note **$poids_i$** calcule en fonction de son nombre de LOC et de sa complexité

$$HK_i = poids_i * (out_i * in_i)^2$$

- **La mesure totale HK correspond a la somme des HK_i**

Autre métrique : métrique MOOD, métriques Objet de Chidamber et Kemerer

Assurance qualité logiciel

Métriques de Henry-Kafura- Exercice

- A partir des **in_i** et **out_i** ci-dessous, calcul des métriques HK en supposant que le poids de chaque module vaut 1

Module	a	b	c	d	e	f	g	h
in_i	4	3	1	5	2	5	6	1
out_i	3	3	4	3	4	4	2	6
HK_i	144	81	16	225	64	400	144	36

$$HK = 1110$$

Assurance qualité logiciel

Revues et Inspections

- **Examen détaillé** d'une **spécification**, d'une **conception** ou d'une **implémentation** par une personne ou un groupe de personnes, afin de déceler des **fautes**, des **violations** de normes de développement ou d'autres **problèmes.**

- **Examen systématique** de certains points pour la **recherche** de potentiels défauts et la **vérification** de l'application de certaines règles menée par un spécialiste du domaine inspecté

Assurance qualité logiciel

Type de Revues et Inspections

- **Auto-correction (desk-checking)**

  – relecture **personnelle**

  – **bilan** : efficacité quasi nulle pour les documents amont, faible pour le code

- **Lecture croisée (author-reader cycle)**

– un **collègue** recherche des ambiguïtés, oublis, imprécisions

– **bilan** : efficacité faible pour les documents amont, plus adapté pour la relecture du code

- **Revue (walkthrough)**

– **discussion** informelle au sein d'un groupe

– un lecteur **résume** paragraphe par paragraphe

– **bilan** : contribution moyenne à l'assurance qualité (évaluation très liée à la prestation du lecteur)

Assurance qualité logiciel

Type de Revues et Inspections

- **Revue structurée**

– constitution pendant le **débat** d'une liste de défauts, utilisation d'une liste de défauts typiques (**checklist**)

– direction des débats par un secrétaire

– **bilan** : bonne contribution à l'assurance qualité

- **Inspection**

– cadre de relecture plus **formel**

– recherche des défauts **avant** les débats

– **suivi** des décisions et corrections

– **bilan** : excellente contribution à l'assurance qualité

Assurance qualité logiciel

Organisation des Revues et Inspections

1. Préparation

- Recherche des défauts
- Rédaction d'un rapport de défauts basé sur des fiches type

2. Cycle de réunions

- Examen des défauts
- Prise de décision

3. Suivi

- Vérification des corrections ou nouvelle inspection

Assurance qualité logiciel

Planification des Revues et Inspections

Pour chaque type de document :

- Dates de début et de fin par rapport au plan projet
- Critères de sélection des inspecteurs
- Plan d'inspection (parties à inspecter)
- Types de défauts les plus communs (checklist)
- Formulaires d'inspection (description de défauts)
- Critères de succès de l'inspection

Assurance qualité logiciel

Responsabilité des Revues et Inspections

– Inspecteurs :

- responsables de la qualité du produit final
- responsables du respect des principes de qualité

– Auteur :

- mise à disposition des documents à la date prévue
- fournit une opinion sur chaque défaut signalé

– Secrétaire :

- enregistre les défauts considérés et les décisions prises
- assiste le modérateur

– Modérateur :

- responsable du bon déroulement des réunions, des convocations, des tenues, des suspensions…
- veille au maintien des objectifs et aux facteurs humains
- préside la prise de décision

Assurance qualité logiciel

Bilan des Revues et Inspections

- La formalisation oblige à planifier et à observer les principes de qualité
- **Excellente contribution** à l'assurance qualité
- Amélioration du cycle de vie (contrôle au plus tôt)
- Influence **positive** sur la communication et la formation dans le projet

☛ **la meilleure des méthodes de relecture**

Assurance qualité logiciel

Défauts typiques

Référence aux données :

- Variables non initialisées
- Indices de tableaux hors bornes
- Accès à des structures/records à champs variables ou à des unions
- Confusion entre donnée et pointeur vers la donnée
- Déréférence de pointeurs nuls
- Pointeurs sur des données désallouées ou pas encore allouées
- Pointeurs sur des données devenues inutiles mais non libérées

Calculs :

- **Conversions** de type (implicites et explicites)
- **Underflow/overflow** (dépassement de capacité du type)
- Division par **zéro**
- **Précédence** des opérateurs

Assurance qualité logiciel

Défauts typiques

Comparaison :

- incohérence des types : **mélanges d'entiers et de booléens**
- inclusion ou non des bornes incorrecte :

  < au lieu de <=, >= au lieu de >, ...
- inversion du test : **== au lieu de !=, > au lieu de <, ...**
- confusion en égalité (==) et affectation (=) :

  if (x = y) {...} au lieu de if (x == y) {...}:
- confusion entre opérateurs binaires (bits) et logiques :

  et (&, &&, and), ou (|, ||, or), négation (~, !, not)
- négation incorrecte d'une condition logique

  !(x ==1 && (y < 2 || !z)) équiv. x !=1 || (y >= 2 && z)
- précédence des opérateurs booléens

  x || y && z équiv. x || (y && z)

Assurance qualité logiciel

Défauts typiques

Contrôle :

- **Switch** : ensemble de case incomplet, cas default manquant, break oublié
- **Rattachement du else au if**
- **Terminaison du programme** : boucles et récusions sans fin
- **Boucles** : conditions initiales (indices, ...) incorrectes, itérations en plus ou en moins, incohérences après une sortie de boucle anticipée, incohérences après une sortie de boucles emboîtées
- **Procédures et fonctions** : incohérences après une sortie anticipée
- **Exceptions non rattrapées**

Assurance qualité logiciel

chek-liste

- **Commentaires**
- **Structure du code**
- **Références aux données**
- **Calculs**
- **Forme des décisions**
- **Définition de constantes**
- **Comparaisons**
- **Contrôle**
- **Taille des modules**
- **Le plan qualité prévoit en général l'utilisation de règles de programmation qui doivent également être vérifiées.**

Assurance qualité logiciel

Tests

- Le test est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus
- **Vérification** (conformité aux spécifications)
  - tests unitaires
  - tests d'intégration
- **Qualification**
  - validation par rapport aux contraintes non fonctionnelles
    - tests de performance
    - tests de capacité de charge
  - validation par rapport aux besoins
    - bêta-test (chez l'utilisateur final)

Assurance qualité logiciel

- **Défauts typiques – Exercice 1**

  Trouver 4 erreurs

```
int factorielle(int n)
{
   int f;
   while (n >= 0) {
      n = n-1;
      f = f*n;
   }
   return n;
}
```

Assurance qualité logiciel

Défauts typiques – Exercice 1

Trouver 4 erreurs

```
int factorielle(int n)
{
  int f;
  while (n >= 0) {
    n = n-1;
    f = f*n;
  }
  return n;
}
```

- variable f non initialisée (à 1)
- tour de boucle en trop (n = 0)
- opération sur un mauvais indice (n-1 et non n)
- retour d'une mauvaise variable (ici n)

Assurance qualité logiciel

Défauts typiques – Exercice 1

Trouver 4 erreurs

```
int factorielle(int n)
{
   int f = 1;
   while (n > 0) {
      f = f*n;
      n = n-1;
   }
   return f;
}
```

- variable f initialisée (à 1)
- pas de tour de boucle en trop
- opération sur un bon indice (n)
- retour de la variable correcte (f)

Assurance qualité logiciel

Défauts typiques – Exercice 2

Trouver 5 erreurs

```
...
/* Entrée pt cardinal */
printf("Direction : ");
scanf("%c", direction);
switch (direction)
{
  case 'N' : y++;
  case 'O' : x--;
  case 'S' : x++;
}
deplacer(x,y);
...
```

Assurance qualité logiciel

Défauts typiques – Exercice 2

Trouver 5 erreurs

```
...
/* Entrée pt cardinal */
printf("Direction : ");
scanf("%c", direction);
switch (direction)
{
  case 'N' : y++;
  case 'O' : x--;
  case 'S' : x++;
}
deplacer(x,y);
...
```

- variable au lieu de pointeur sur variable
- case manquant
- break oubliés
- copy/paste non ou mal adapté
- default manquant

Assurance qualité logiciel

Défauts typiques – Exercice 2

Trouver 5 erreurs

```
...
/* Entrée pt cardinal */
printf("Direction : ");
scanf("%c",&direction);
switch (direction) {
   case 'E' : x++; break;
   case 'N' : y++; break;
   case 'O' : x--; break;
   case 'S' : y--; break;
   default : error();
}
deplacer(x,y);
```

- pointeur sur variable
- tous les case présents et corrects
- break présents
- pas d'erreur de copy/paste
- default présent

Assurance qualité logiciel

Défauts typiques – Exercice 3

Trouver 5 erreurs

```
int tab[]; /*de taille size*/
int indiceTqTabNul(int size){
   int i;
   if (size > 0)
      for(i=1; i <= size; i++)
         if (tab[i] = 0)
            return i;
   else
      printf("tab est vide");
      error();
   return -1;
}
```

Assurance qualité logiciel

Défauts typiques – Exercice 3

Trouver 4 erreurs

```
int tab[]; /*de taille size*/
int indiceTqTabNul(int size){
    int i;
    if (size > 0)
        for(i=1; i <= size; i++)
            if (tab[i] = 0)
                return i;
    else
        printf("tab est vide");
        error();
    return -1;
}
```

- oubli d'un cas d'indice
- cas d'indice hors borne
- test par = au lieu de ==
- accolades manquantes

Assurance qualité logiciel

Défauts typiques – Exercice 3

Trouver 5 erreurs

```
int tab[]; /*de taille size*/
int indiceTqTabNul(int size){
   int i;
   if (size > 0) {
      for(i=0; i < size; i++)
         if (tab[i] == 0)
            return i; }
   else {
      printf("tab est vide");
      error(); }
   return -1;
}
```

- pas d'indice oublié
- pas d'indice hors borne
- test par '==' et non '='
- accolades appropriées

Assurance qualité logiciel

- **Défauts typiques – Exercice 4**

  Trouver 1 erreur

```
int compte = ...;
int decouvert_max = -1000;
void transact(int montant)
/* débit: montant < 0
 * crédit: montant > 0 */
{
  if (compte + montant <
        decouvert_max)
    printf("Refusé");
  else
    compte += montant;
}
```

Assurance qualité logiciel

Défauts typiques – Exercice 4

Trouver 1 erreur

```
int compte = ...;
int decouvert_max = -1000;
void transact(int montant)
/* débit: montant < 0
 * crédit: montant > 0 */
{
  if (compte + montant <
      decouvert_max)
    printf("Refusé");
  else
    compte += montant;
}
```

- overflow non maîtrisé : compte + montant > 2^{31} → compte + montant < 0
- underflow non maîtrisé : si compte=-500 et montant=-2^{31} alors compte + montant > 0 ! (= 2^{31}-500)

Assurance qualité logiciel

- **Défauts typiques – Exercice 4**

  Trouver 1 erreur

```
void transact(int montant)
{
  if (montant > 0 && compte > 0 && compte+montant < 0)
    printf("Overflow");
  else if (montant<0 && compte<0 && compte+montant>0)
    printf("Underflow");
  else if (compte+montant < decouvert_max)
      printf("Refusé");
  else
    compte += montant;
}
```

Assurance qualité logiciel

Défauts typiques – Exercice 5

Trouver 3 erreurs

```
/* Si p non nul pointe sur
 * un entier positif */
if (p != NULL & *p >= 0) ...

/* Si x est pair et
 * de valeur absolue >= 5 */
if (x%2 == 0 &&
      x <= -5 || 5 <= x) ...

/* Si x et y sont positifs
 * ou nuls */
if (x && y >= 0) ...
```

Assurance qualité logiciel

Défauts typiques – Exercice 5

Trouver 3 erreurs

```
/* Si p non nul pointe sur
 * un entier positif */
if (p != NULL & *p >= 0) ...

/* Si x est pair et
 * de valeur absolue >= 5 */
if (x%2 == 0 &&
      x <= -5 || 5 <= x) ...

/* Si x et y sont positifs
 * ou nuls */
if (x && y >= 0) ...
```

- expression évaluée ne devant pas l'être
- précédence des opérateurs logiques
- comparaison factorisée

Assurance qualité logiciel

Défauts typiques – Exercice 5

Trouver 3 erreurs

```
/* Si p non nul pointe sur
 * un entier positif */
if ((p != NULL) && (*p >= 0))

/* Si x est pair et
 * de valeur absolue >= 5 */
if (x%2 == 0 &&
    (x <= -5 || 5 <= x)) ...

/* Si x et y sont positifs
 * ou nuls */
if (x >= 0 && y >= 0) ...
```

- expression évaluée que si nécessaire
- parenthésage des opérateurs logiques
- comparaison non factorisée

Assurance qualité logiciel

- **Défauts typiques – Exercice 6**

  Trouver 3 erreurs

```
typedef struct lst {
  int elem;
  struct lst *reste;
} *list;

int dernierElem(list l)
{
  do {
    l = l->reste;
  } while (l->reste != NULL);
  return l->reste->elem;
}
```

Assurance qualité logiciel

Défauts typiques – Exercice 6

Trouver 3 erreurs

```
typedef struct lst {
  int elem;
  struct lst *reste;
} *list;

int dernierElem(list l)
{
  do {
    l = l->reste;
  } while (l->reste != NULL);
  return l->reste->elem;
}
```

- cas d'un argument pointeur nul (liste vide) non traité
- premier élément non traité (liste à un élément)
- déréférence d'un pointeur nul au terme de l'itération

Assurance qualité logiciel

Défauts typiques – Exercice 6

Trouver 3 erreurs

```
typedef struct lst {
  int elem;
  struct lst *reste;
} *list;

int dernierElem(list l)
{
  if (l == NULL) error();
  while (l->reste != NULL)
    l = l->reste;
  return l->elem;
}
```

- cas d'un argument pointeur nul (liste vide) traité
- premier élément traité correctement
- déréférence du bon pointeur au terme de l'itération

Assurance qualité logiciel

Méthode SCOPE (*Software CertificatiOn Programmein Europe*)

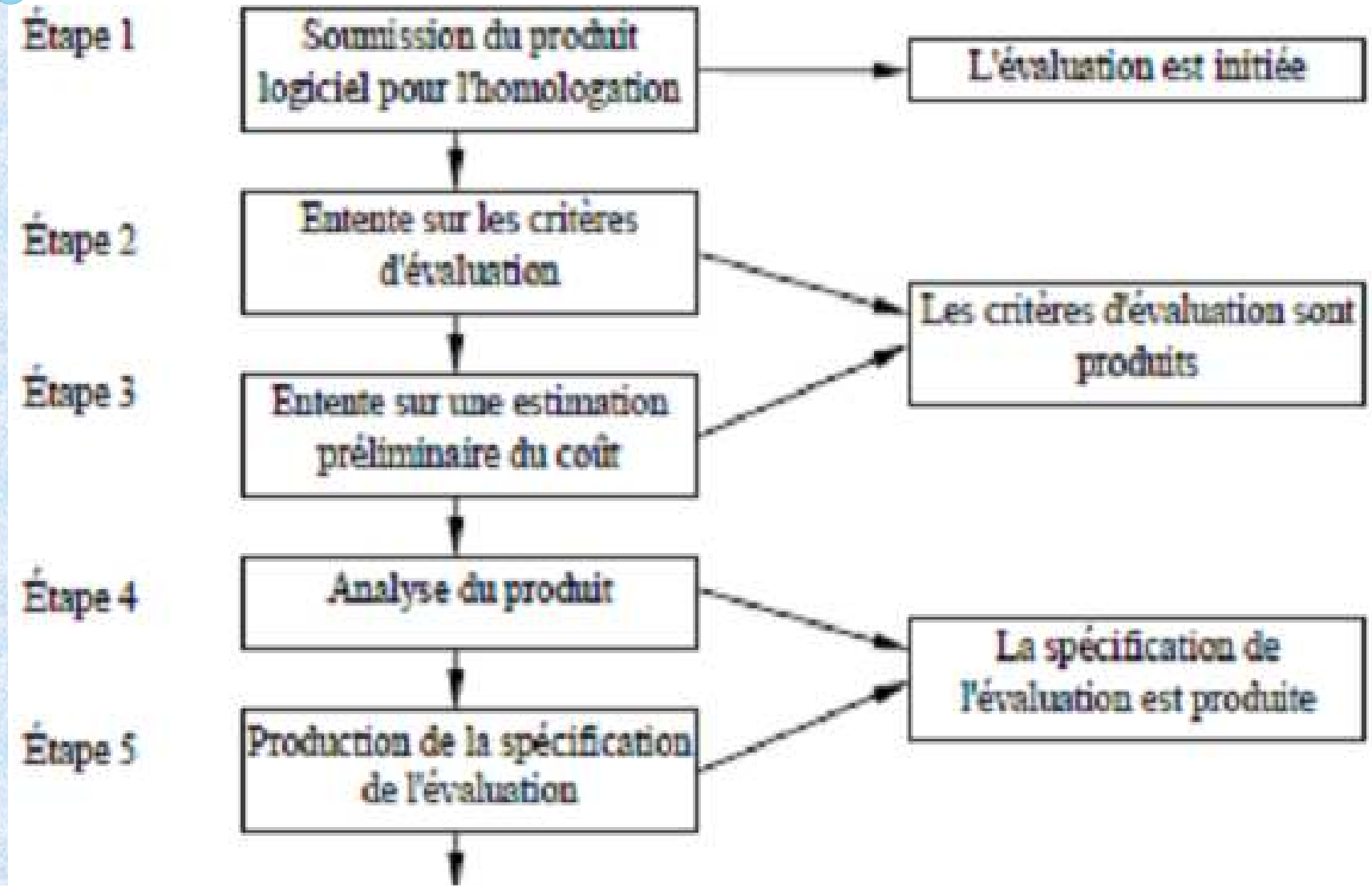
est un projet européen ESPRIT

Objectifs

- **Définir les procédures** d'attribution d'un label de qualité à un logiciel quand celui-ci satisfait un certain ensemble d'attributs de qualité
- **Développer des technologies** nouvelles et efficaces d'évaluation, à des coûts raisonnables, permettant l'attribution de ce label
- **Promouvoir l'utilisation** des technologies modernes de l'ingénierie des logiciels. Celles-ci, étant utilisées durant le développement des logiciels, contribuent à l'attribution de ce label

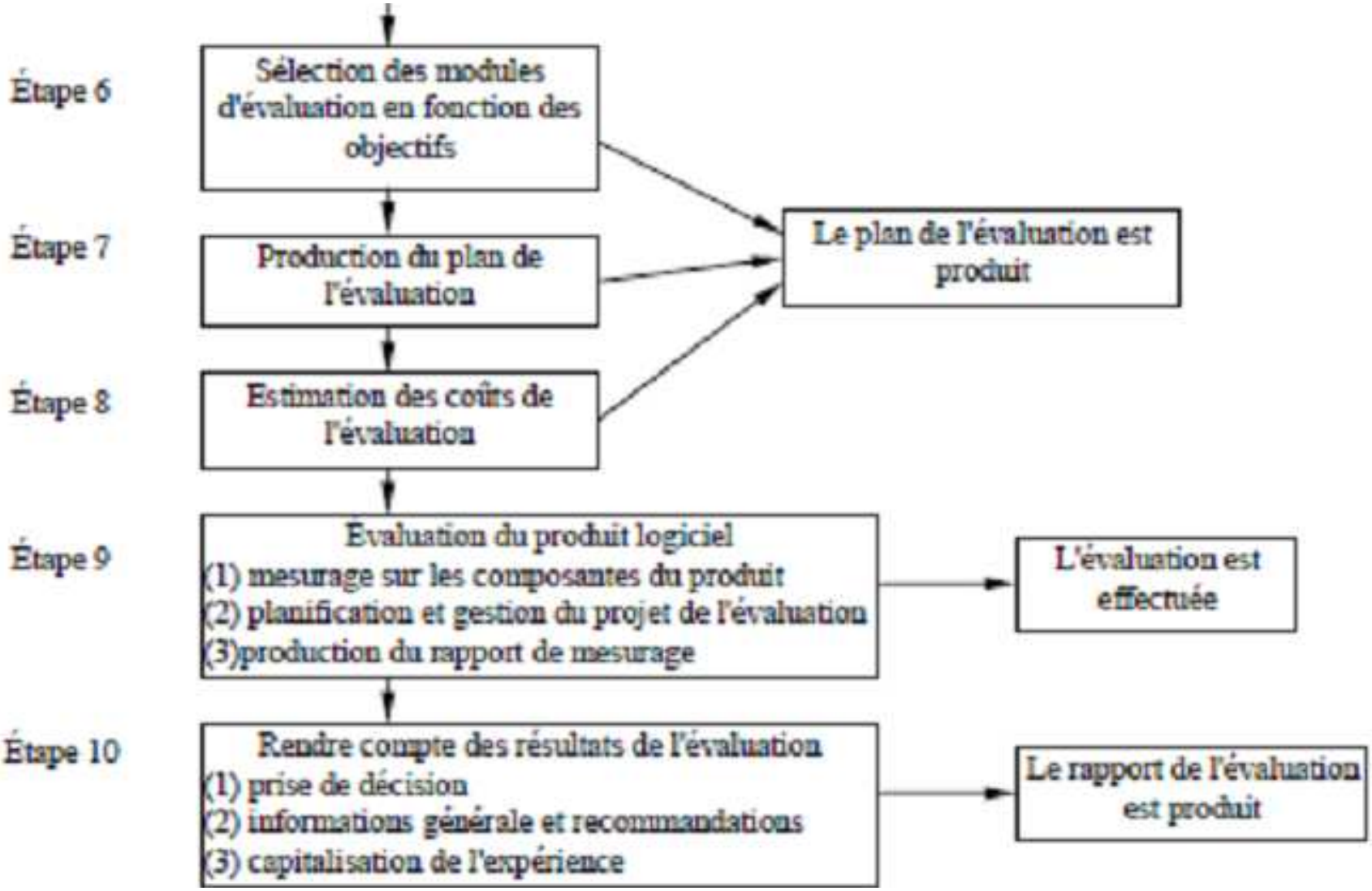
Assurance qualité logiciel

- **Modèle SCOPE**



Assurance qualité logiciel

- **Modèle SCOPE**



Assurance qualité logiciel

Méthode SCOPE

La méthode d'évaluation s'appuie sur trois types d'analyse techniques

- **L'analyse statique** qui consiste à examiner le code pour **évaluer** les caractéristiques de qualité.
- **L'analyse dynamique** qui consiste entre autres à **simuler** le déroulement de l'application pour effectuer des mesures.
- **L'inspection** qui concerne particulièrement les **interfaces "utilisateur".**

Assurance qualité logiciel

Méthode SCOPE

L'évaluation peut se faire selon le niveau de détail souhaité

Niv.	Environnement	Personnes	Économie	Application
D	petit dommage à la propriété	pas de risques pour les personnes	perte économique négligeable	loisirs, domestiques
C	dommage à la propriété	peu de personnes touchées	perte économique significative	alarmes de feu, contrôle de processus
B	dommage environnemental réparable	menace pour des vies humaines	grande perte économique	systèmes médicaux, systèmes financiers
A	dommage environnemental irréparable	des personnes mortes	désastre financier	systèmes de transport, systèmes du nucléaire

Assurance qualité logiciel

Méthode SCOPE

Choix des techniques pour chaque niveau

	Niveau D	Niveau C	Niveau B	Niveau A
Capacité fonctionnelle	test fonctionnel (boîte noire)	+ inspection des documents (listes de contrôle)	+ test des composantes	+ preuve formelle
Fiabilité	facilités des langages de programmation	+ analyse de la tolérance aux fautes	+ modèle de croissance de la fiabilité	+ preuve formelle
Facilité d'utilisation	inspection des interfaces utilisateur	+ conformité aux normes sur les interfaces	+ test en laboratoire	+ modèle mental de l'utilisateur
Rendement	mesurage du temps d'exécution	+ test avec bancs d'essais (*benchmarks*)	+ complexité algorithmique	+ analyse des performances
Maintenabilité	inspection des documents (listes de contrôle)	+ analyse statique	+ analyse du processus de développement	+ évaluation de la traçabilité
Portabilité	analyse de l'installation	+ conformité avec les règles de programmation	+ évaluation des contraintes de l'environnement	+ évaluation de la conception des programmes

Cycle de vie de développement logiciel

Définition

Le « **cycle de vie d'un logiciel** » (*software lifecycle*), désigne toutes les étapes du développement d'un logiciel, de sa conception à sa disparition.

Ce découpage permet de définir des jalons intermédiaires permettant la **validation** du développement logiciel (conformité du logiciel avec les besoins exprimés), et la **vérification** du processus de développement (adéquation des méthodes mises en œuvre).

L'origine de ce découpage provient du constat que les erreurs ont un coût d'autant plus élevé qu'elles sont détectées tardivement dans le processus de réalisation.

Le cycle de vie permet de détecter les erreurs au plus tôt et ainsi de maîtriser la **qualité** du logiciel, les délais de sa réalisation et les coûts associés.

Cycle de vie de développement logiciel

Activités du cycle de vie

- **Définition des objectifs** : consistant à définir la finalité du projet et son inscription dans une stratégie globale.
- **Analyse des besoins et faisabilité** : c'est-à-dire l'expression, le recueil et la formalisation des besoins du demandeur (le client) et de l'ensemble des contraintes.
- **Conception générale** : Il s'agit de l'élaboration des spécifications de l'architecture générale du logiciel.
- **Conception détaillée** : consistant à définir précisément chaque sous-ensemble du logiciel.
- **Codage** (Implémentation ou programmation) : soit la traduction dans un langage de programmation des fonctionnalités définies lors de phases de conception.

Cycle de vie de développement logiciel

Activités du cycle de vie *(suite)*

- **Tests unitaires** : permettant de vérifier individuellement que chaque sous-ensemble du logiciel est implémentée conformément aux spécifications.
- **Intégration** : dont l'objectif est de s'assurer de l'interfaçage des différents éléments (modules) du logiciel. Elle fait l'objet de *tests d'intégration* consignés dans un document.
- **Qualification** (ou *recette*) : c'est-à-dire la vérification de la conformité du logiciel aux spécifications initiales.
- **Documentation** : visant à produire les informations nécessaires pour l'utilisation du logiciel et pour des développements ultérieurs.
- **Mise en production** : déploiement sur site du logiciel.
- **Maintenance** : comprenant toutes les actions correctives (maintenance corrective) et évolutives (maintenance évolutive) sur le logiciel.

Cycle de vie de développement logiciel

Phase : définition des besoins

- **Activités (du client, externe ou interne) :**
  - Consultation d'experts et d'utilisateurs potentiels
  - Questionnaire d'observation de l'usager dans ses tâches
- **Productions :**
  - Cahier des charges (les « exigences »)
    - Fonctionnalités attendues
    - Contraintes non fonctionnelles
      - Qualité, précision, optimalité, ...
      - temps de réponse, contraintes de taille, de volume de traitement, ...

Cycle de vie de développement logiciel

Phase : Analyse des besoins / spécification

- **Activités :**
  - Spécifications générales : Objectifs, contraintes (utilisation de matériels et outils existants) et généralités à respecter
  - Spécifications fonctionnelles : description détaillé des fonctionnalités du logiciel
  - Spécifications d'interface : description des interfaces du logiciel avec le monde extérieure (hommes, autres logiciels, matériels...)
- **Productions :**
  - Dossier d'analyse
  - Ebauche du manuel utilisateur
  - Première version du glossaire

Cycle de vie de développement logiciel

Phase : Planification / Gestion de projet

- **Activités :**
  - Tâches : découpage, enchaînement, durée, effort
  - Choix : normes qualité, méthodes de conception
- **Productions :**
  - Plan qualité
  - Plan projet destiné aux développeurs
  - Estimation des coûts réels, réalisation de devis
  - Liste de dépendances extérieures (risques)

Cycle de vie de développement logiciel

Phase : Conception

- **Activités :**
  - Définition de l'architecture du logiciel
  - Choix de solutions et leurs faisabilités
  - Description externe des procédures et des structures de données.
  - Définition des interface entre les différents modules
- **Productions :**
  - Dossier de conception
  - plan d'intégration
  - plans de tests
  - planning mis à jour

Cycle de vie de développement logiciel

Phase : Codage et tests unitaires

- **Activités :**
  - Chaque module codé et testé indépendamment

- **Productions :**
  - Modules codés et testés
  - Documentation de chaque module
  - Résultats des tests unitaires
  - Planning mis à jour

Cycle de vie de développement logiciel

Phase : Intégration

- **Activités :**
  - Modules testés et intégrés suivant le plan d'intégration
  - Test de l'ensemble conformément au plan de tests

- **Productions :**
  - Logiciel testé
  - Jeu de tests de non-régression
  - Manuel d'installation : guide dans les choix d'installation
  - Version finale du manuel utilisateur
  - Manuel de référence : liste exhaustive de chaque fonctionnalité

Cycle de vie de développement logiciel

Phase : Qualification

- **Activités :**
  - Test en vraie grandeur dans les conditions normales d'utilisation
  - Intégration dans l'environnement extérieur (autres logiciels, utilisateurs).
  - Vérification que le logiciel développé répond aux besoins exprimés dans la phase de spécifications des besoins.

Productions :

- diagnostic OK / KO

Cycle de vie de développement logiciel

Phase : Maintenance

- **Activités :**
  - Formation de l'utilisateur et assistance technique
  - Maintenance corrective : non conformité aux spécifications, d'où détection et correction des erreurs résiduelles.
  - Maintenance adaptative : modification de l'environnement (matériel, fournitures logicielles, outil, ...)
  - Maintenance évolutive : changement des spécifications fonctionnelles du logiciel

  Productions :
  - Logiciel corrigé
  - Mises à jour, patch
  - Documents corrigés.

Cycle de vie de développement logiciel

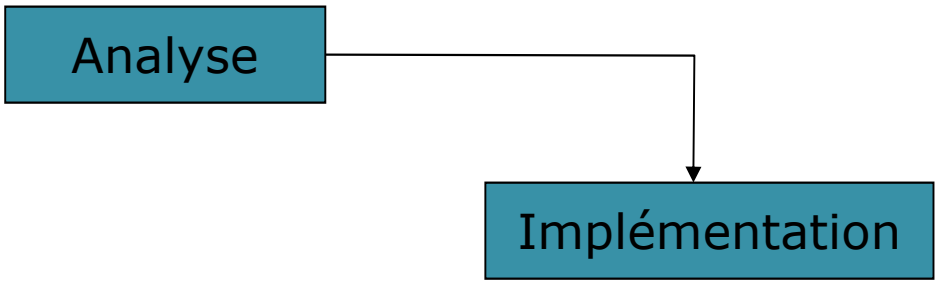
Modèles de cycles de vie

- Modèle linéaire
  - Modèle en cascade
  - Modèle en V
- Modèle itératif
  - Modèle par prototype
  - Modèle en spirale
  - Modèle incrémental
- Modèle avec méthode formelle
- Modèle avec réutilisation de composantes

Cycle de vie de développement logiciel

Le modèle linéaire

Historiquement, la première tentative pour mettre de la rigueur dans le ‘développement sauvage’ (coder et corriger ou ‘code and fix’) a consisté à distinguer une phase d’*analyse* avant la phase d’*implémentation* (séparation des questions).



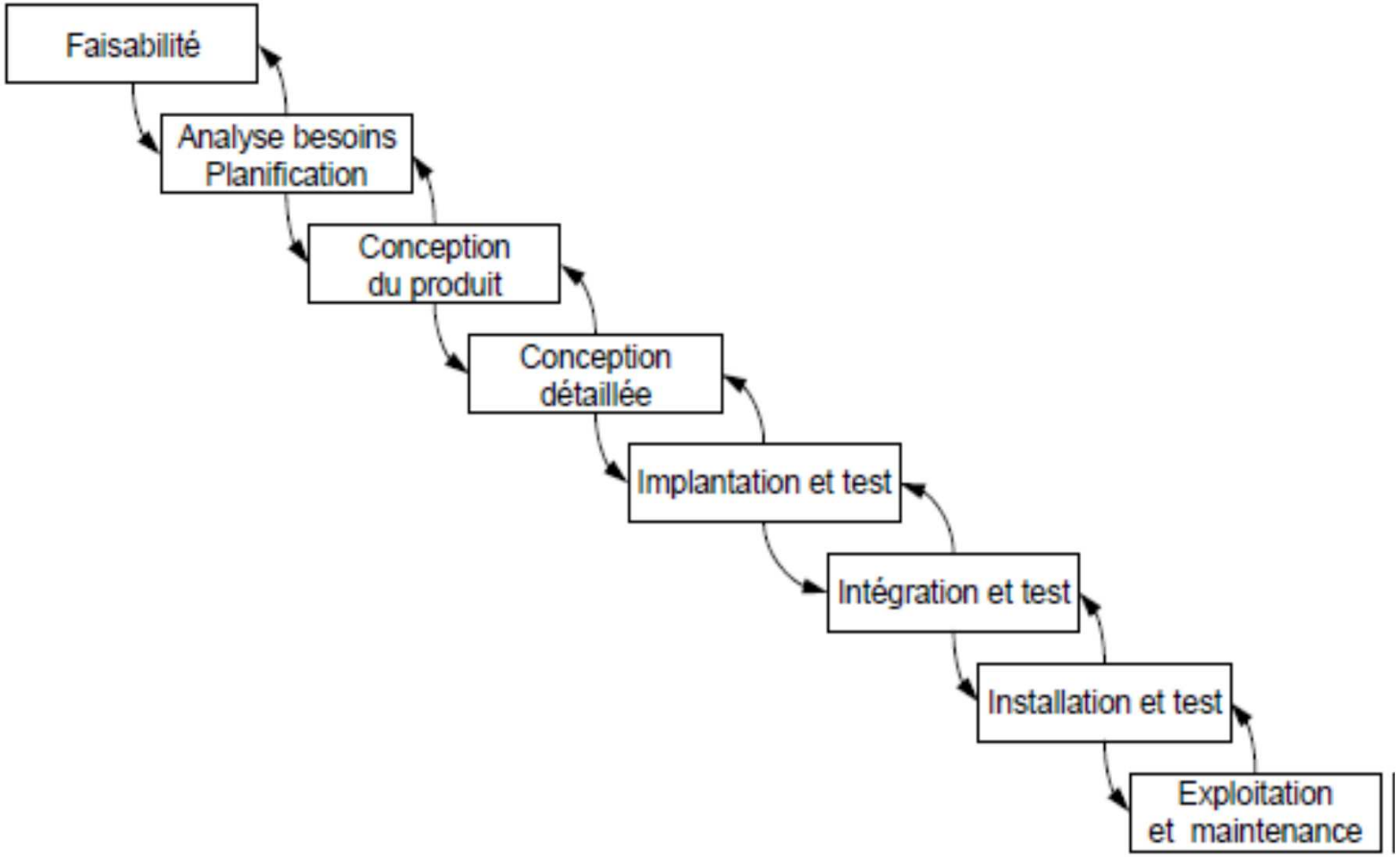
Cycle de vie de développement logiciel

Le modèle en cascade

- Il a été mis au point dès 1966, puis formalisé aux alentours de 1970.
- Le modèle de la cascade défini des étapes (ou phases) durant lesquelles les activités de développement se déroulent
- Une étape doit se terminer à une date donnée par la production de certains documents ou logiciels Les résultats de l'étape sont soumis à une étude approfondie
- L'étape suivante n'est abordée que si les résultats sont jugés satisfaisants

Cycle de vie de développement logiciel

Le modèle en cascade



Cycle de vie de développement logiciel

Le modèle en cascade

Faisabilité (Pourquoi?)

- Pourquoi faut-il réaliser un logiciel?
- Y a-t-il de meilleur alternatives?
- Le logiciel sera-t-il satisfaisant pour les utilisateur?
- Y a-t-il un marché pour le logiciel?
- A-t-on le budget, le personnel, le matériel nécessaire?

Cycle de vie de développement logiciel

Le modèle en cascade

Analyse des besoins (quoi?)

- Définir précisément les fonctions que le logiciel doit réaliser ou fournir
- Le résultat de cette phase est la réalisation du cahier de charge du logiciel

Cycle de vie de développement logiciel

Le modèle en cascade

Conception (comment?)

- Définir la structure du logiciel
- Les résultats comprennent l'architecture du logiciel (décomposition en modules) et la spécification des interfaces des modules
- La définition des algorithmes de chacune des procédures des modules est appelé la conception détaillée du logiciel

Cycle de vie de développement logiciel

Le modèle en cascade

Implantation et test (comment?)

- Implanter les procédures des modules
- Tests unitaire

Intégration et test

- Intégrer les différents modules
- Valider / vérifier l'adéquation de l'implantation, de la conception et de l'architecture avec le cahier de charge (acceptation)

Cycle de vie de développement logiciel

Le modèle en cascade

Installation et test

- Déploiement du logiciel chez le client et tests avec des sous ensemble d'usager choisi

Exploitation et maintenance

- Utilisation en situation réelle, retour d'information des usagers, des administrateurs, des gestionnaires…
- Maintenance corrective, perfective et adaptative

Cycle de vie de développement logiciel

Le modèle en cascade

Inconvénients

- Entraîne des relations conflictuelles avec les parties prenantes en raison :
  - Du manque de clarté de la définition des exigences
  - D'engagements importants dans un contexte de profonde incertitude
  - D'un désir inévitable de procéder à des changements
- Entraîne une identification tardive de la conception, et un démarrage tardif du codage
- Retarde la résolution des facteurs de risque (intégration tardive dans le cycle de vie)
- Exige d'accorder une attention très importante aux documents

Cycle de vie de développement logiciel

Le modèle en cascade

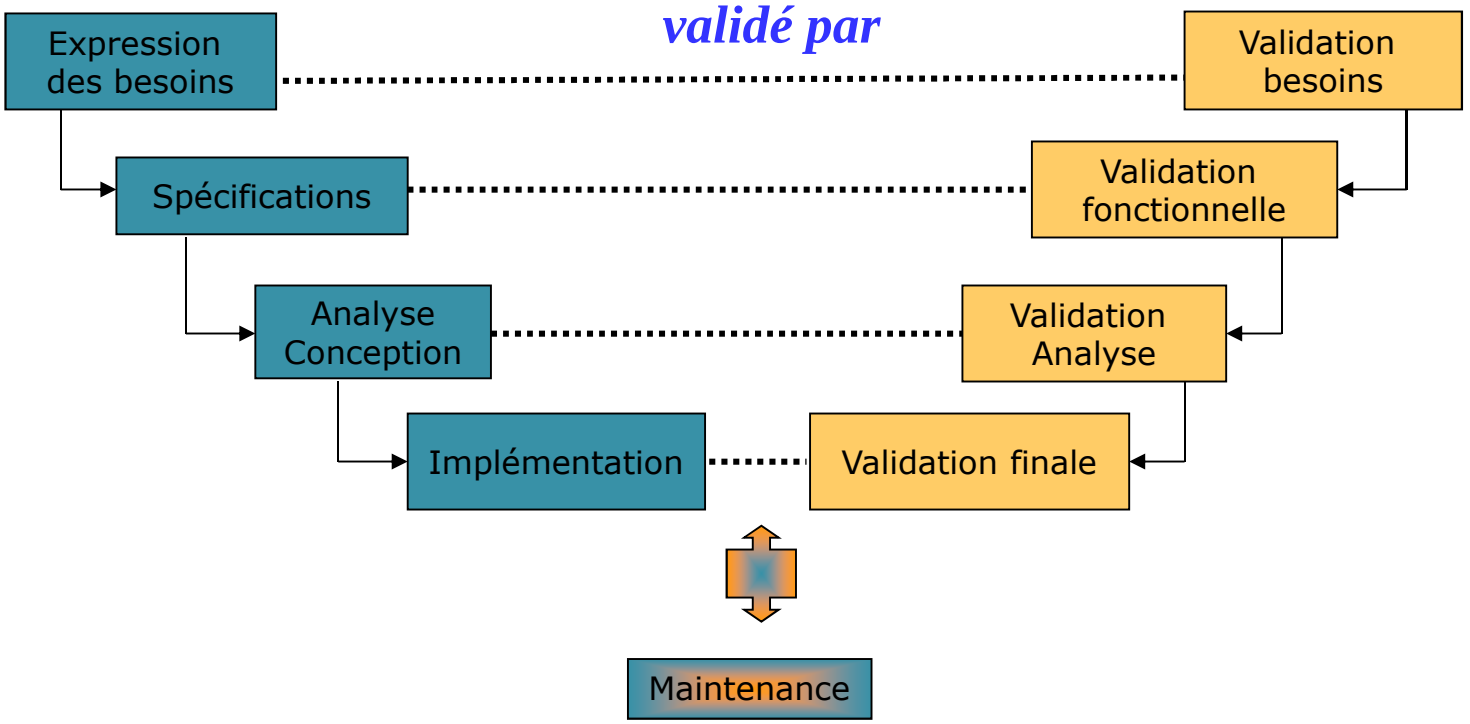
Validation et vérification

Les versions actuelles du modèle en cascade contiennent de la validation et de la vérification à chaque étape

- Faisabilité et analyses des besoins + validation
- Conception, conception détaillée + vérification
- Implantation + tests unitaires
- Intégration + tests d'intégration et d'acceptation
- Installation + tests de déploiement

Cycle de vie de développement logiciel

Le modèle en V



Validations des étapes intermédiaires sous forme de documents

Cycle de vie de développement logiciel

Le modèle en V

- Modèle de cascade amélioré dans lequel le développement des tests et du logiciels sont effectués de manière synchrone
- Le principe de ce modèle est qu'avec toute décomposition doit être décrite la recomposition et que toute description d'un composant est accompagnée de tests qui permettront de s'assurer qu'il correspond à sa description.
- Problème de vérification trop tardive du bon fonctionnement du système.

Cycle de vie de développement logiciel

Le modèle en V

- **Objectifs** :Validations intermédiaires pour prévenir les erreurs tardives : meilleure planification et gestion

- **Principes du cycle de vie en V**

  - Validation à chaque étape
  - Préparation des protocoles de validation finaux à chaque étape descendante
  - Validation finale montante et confirmation de la validation descendante

Cycle de vie de développement logiciel

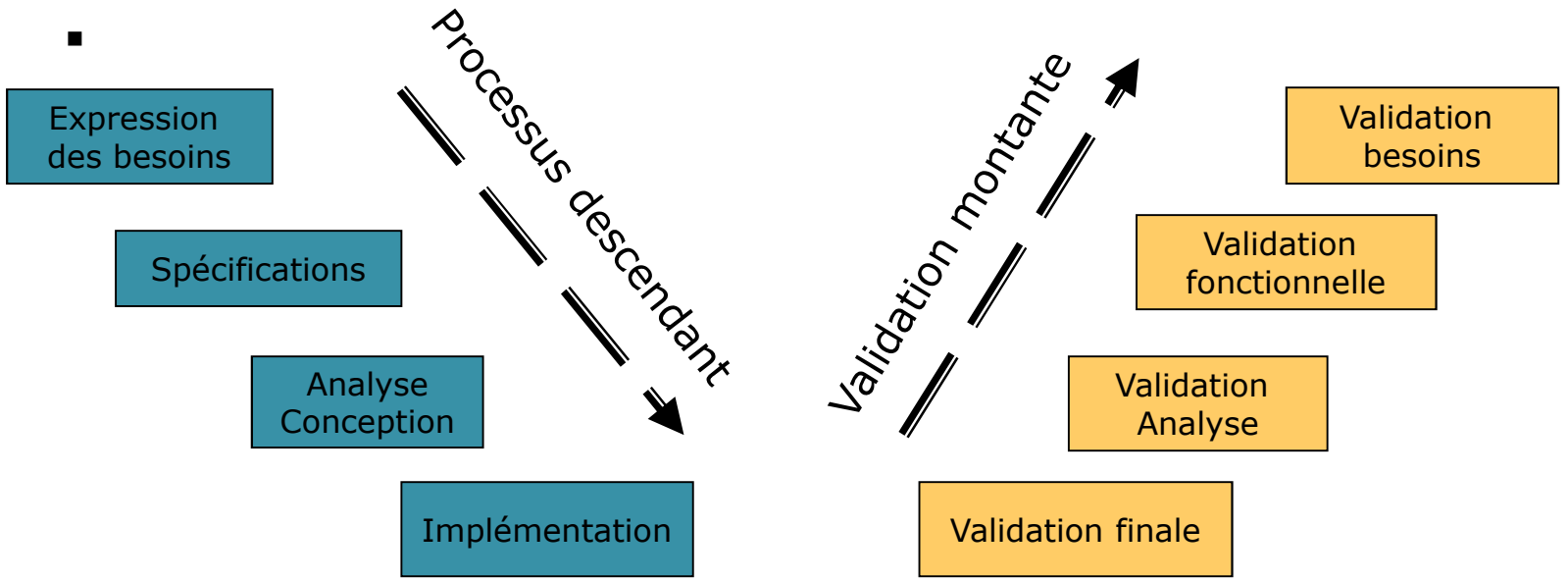
Le modèle en V

- **Conséquences :**
  - Obligation de concevoir les jeux de test et leurs résultats
  - Réflexion et retour sur la description en cours
  - Meilleure préparation de la branche droite du V

- **Les activités** de chaque phase peuvent être réparties en 5 catégories :
  - Assurance qualité
  - Production
  - Contrôle technique
  - Gestion
  - Contrôle de qualité

Cycle de vie de développement logiciel

Le modèle en V



Protocoles de validation définis par l'analyse descendante

Cycle de vie de développement logiciel

Intérêt du modèle en V

- Validations intermédiaires
  - Bon suivi du projet : avancement éclairé et limitation des risques en cascade d'erreurs
  - Favorise la décomposition fonctionnelle de l'activité
  - Génération de documents et outils supports
- Modèle très utilisé et éprouvé

Cycle de vie de développement logiciel

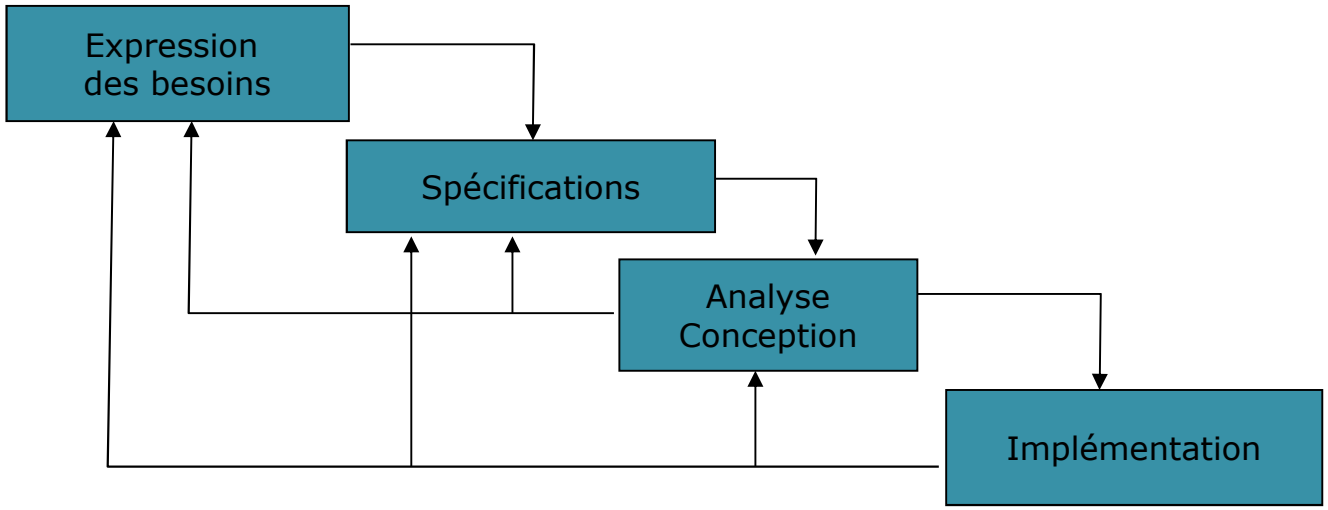
Limitation modèle en V

- Un modèle séquentiel (linéaire)
  - Manque d'adaptabilité
  - Maintenance non intégrée : logiciels à vocation temporaire
  - Validations intermédiaires non formelles : aucune garantie sur la non transmission d'erreurs
  - Problème de vérification trop tardive du bon fonctionnement du système.

Cycle de vie de développement logiciel

Amélioration du modèle en V

- Retours de correction des phases précédentes
  - Fonctionne si corrections limitées pour casser la linéarité : cycle de vie itératif



Cycle de vie de développement logiciel

Le modèle itératif

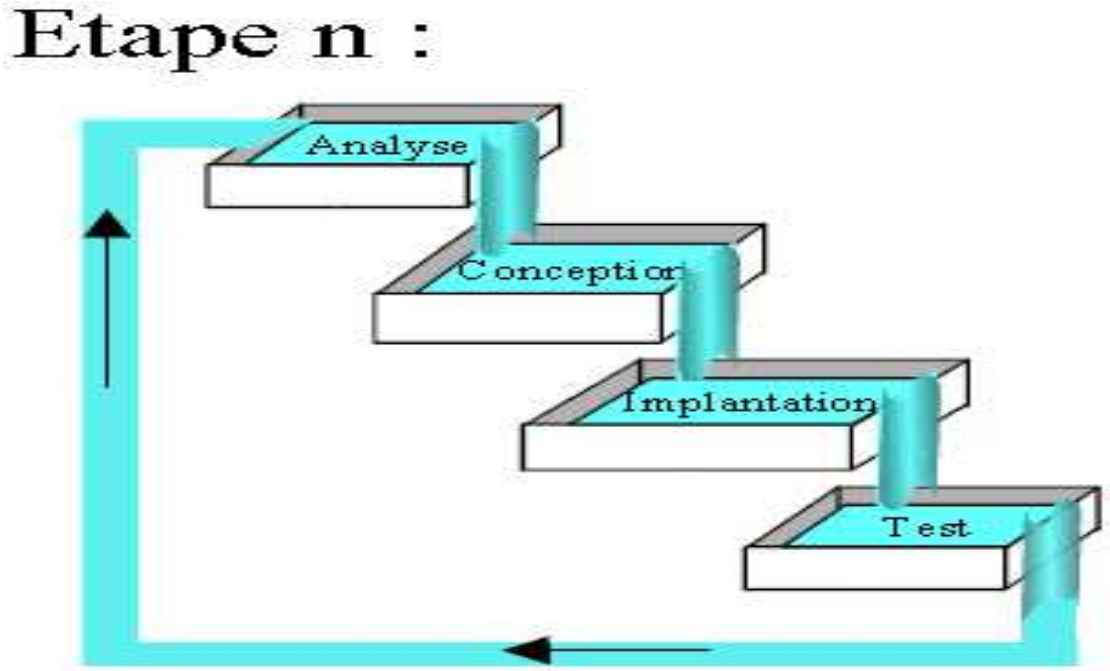
Principe

A chaque étape, on rajoute de nouvelles fonctionnalités

Caractéristiques

Chaque étape est relativement simple

On peut tester et essayer au fur et à mesure le logiciel que l'on développe



Cycle de vie de développement logiciel

Le modèle par prototype

- Ce modèle est intéressant pour les cas où les besoins ne sont pas clairement définis ou ils changent au cours de temps

- Le prototypage permet le développement rapide d'une ébauche du futur logiciel afin de fournir rapidement un prototype exécutable pour permettre une validation concrète (ici expérimentale) et non sur papier (document)

- Progressions par incréments successifs de versions successives du prototype : itérations

Cycle de vie de développement logiciel

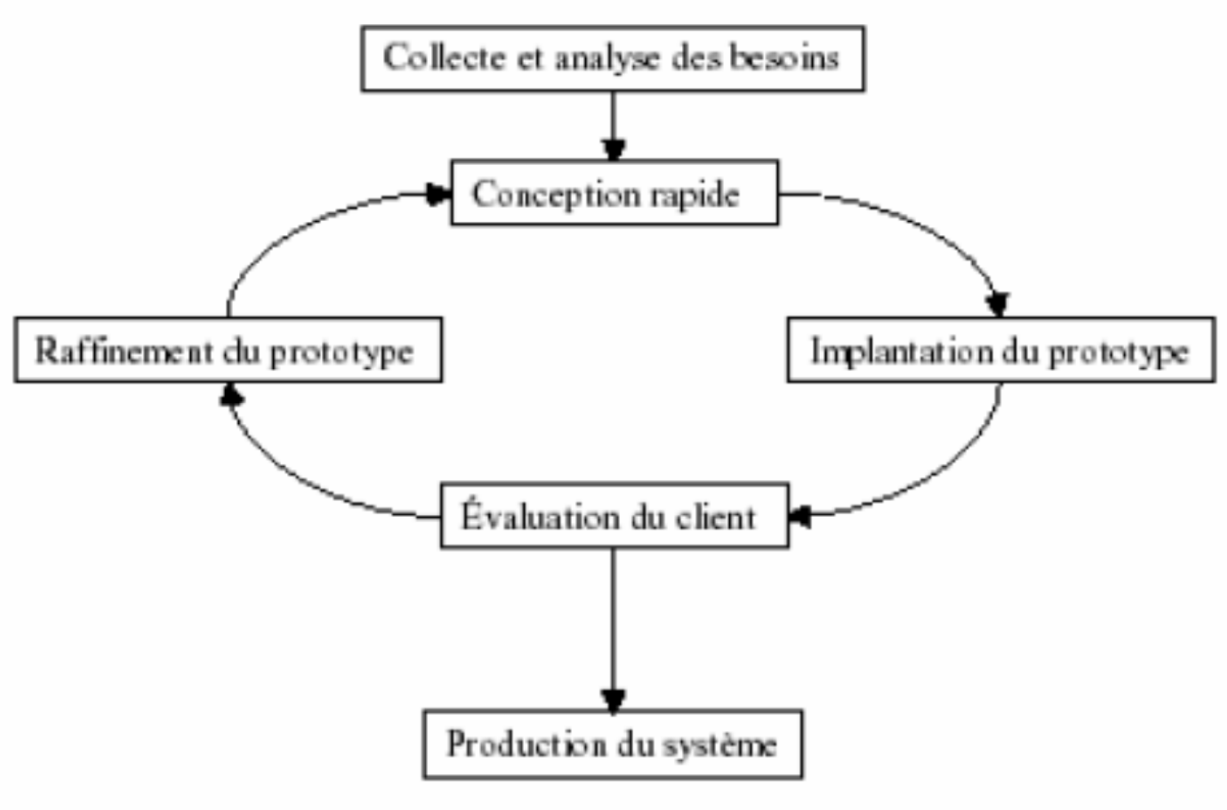
Le modèle par prototype

- **Prototype jetable :**
  - Modèle gérable d'un point de vue changement des spécifications qui sont générées par prototypage
  - Construit et utilisé lors de l' analyse des besoins et la spécifications fonctionnelles
  - Validation des spécifications par l'expérimentation

- **Prototype évolutif :**
  - Première version du proto = embryon du produit final
  - Itération jusqu'au produit final
  - Difficulté à mettre en œuvre des procédures de validation et de vérification
  - Modèle en spirale, développement incrémental

Cycle de vie de développement logiciel

Le modèle par prototype



Cycle de vie de développement logiciel

Le modèle en spirale

- Proposé par B. Boëhm en 1988, ce modèle général met l'accent sur l'évaluation des risques

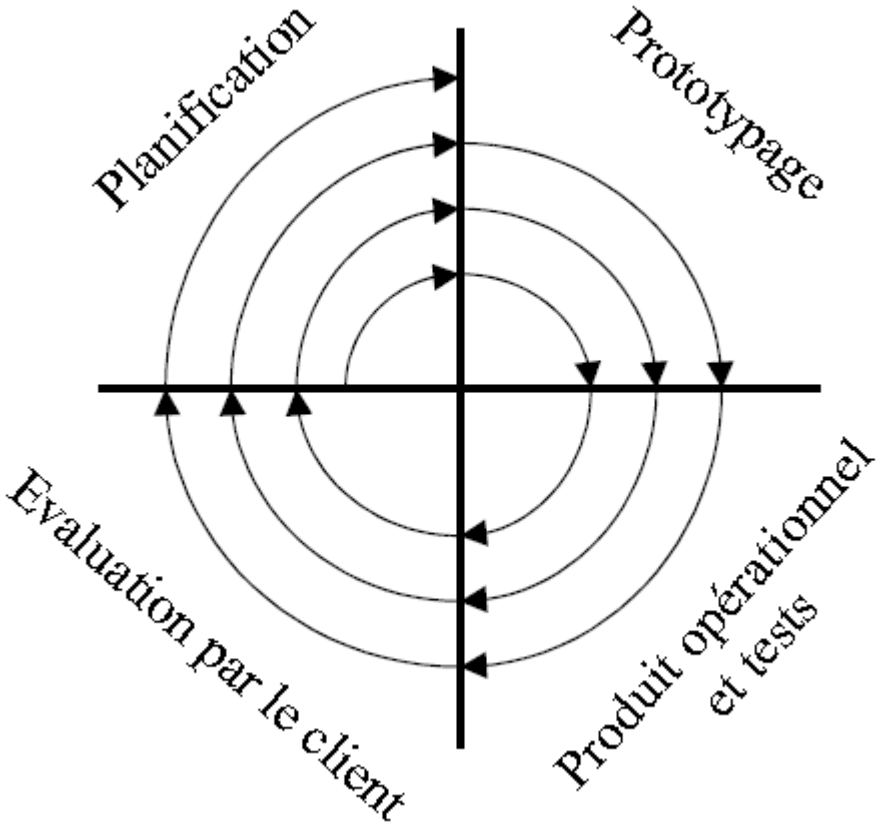
- A chaque étape, après avoir défini les objectifs et les alternatives, celles-ci sont évaluées par différentes techniques (prototypage, simulation, ...)

- L'étape est réalisée et la suite est planifiée

- Le nombre de cycles est variable selon que le développement est classique ou incrémental

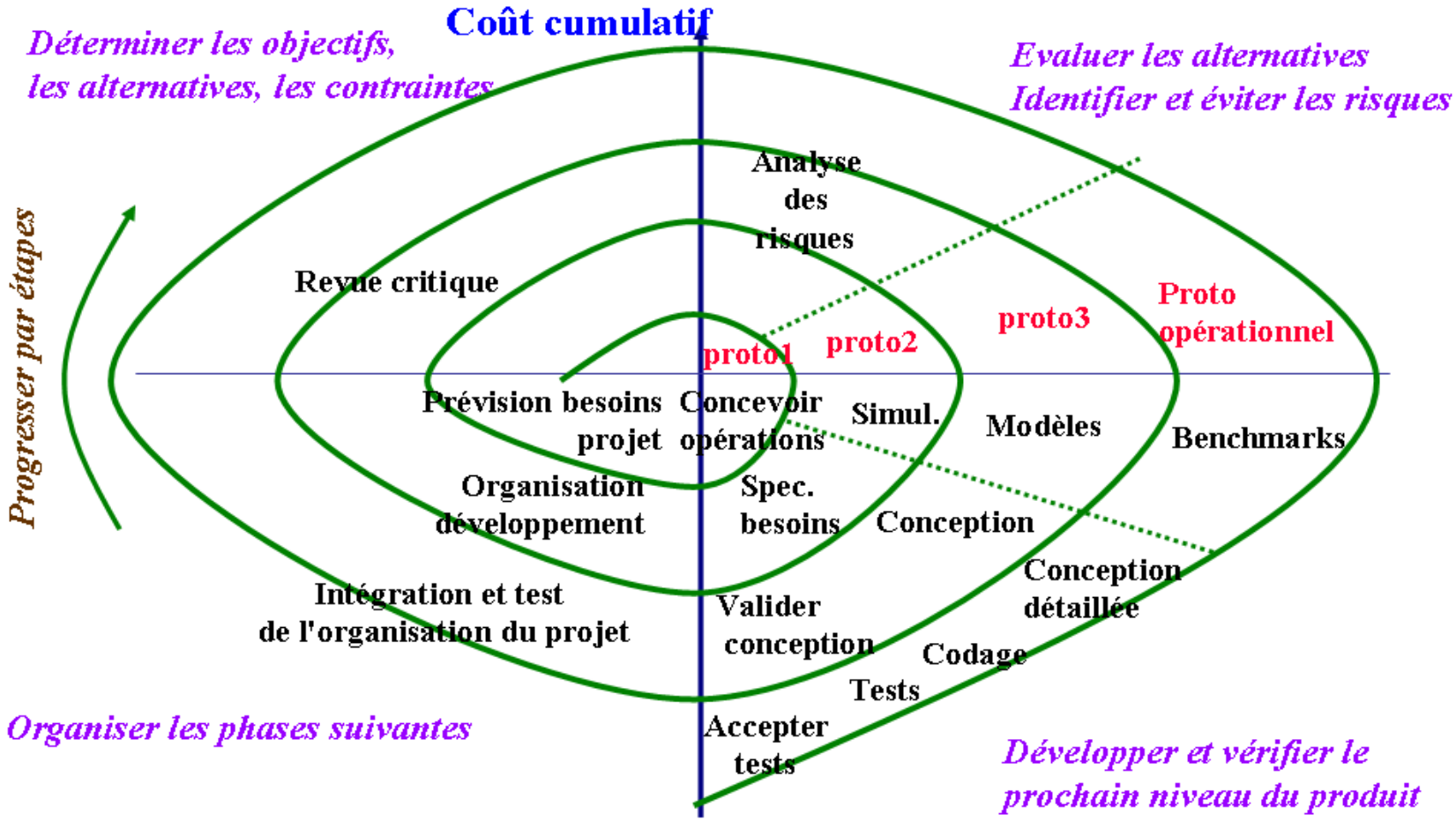
Cycle de vie de développement logiciel

Principe du modèle en spirale



Cycle de vie de développement logiciel

Le modèle en spirale



Cycle de vie de développement logiciel

Le modèle en spirale

Risques majeurs :

- Défaillance du personnel
- Calendrier et budget irréalistes
- Développement de fonction inapproprié
- Développement d'interfaces utilisateurs inappropriées
- Produit « plaqué or »
- Validité des besoins
- Composants externes manquants
- Tâches externes défaillantes
- Problème de performance
- Exigences démesurées par rapport à la technologie

Cycle de vie de développement logiciel

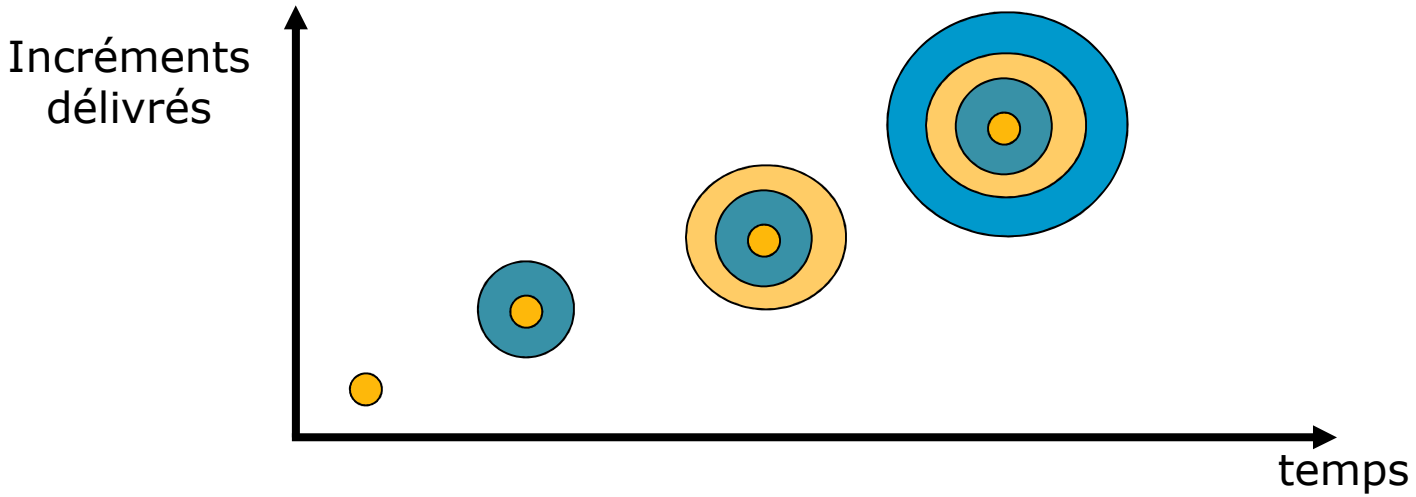
Le modèle en spirale

Risque	Remède
Défaillance de personnel	Embauches de haut niveau, formation mutuelle, leaders, adéquation profil/fonction, …
Calendrier et budgets irréalistes	Estimation détaillée, développement incrémental, réutilisation, élagage des besoins, …
Développement de fonctions inappropriées	Revues d'utilisateurs, manuel d'utilisation précoce, ...
Développement d'interfaces utilisateurs inappropriées	Maquettage, analyse des tâches, …
Volatilité des besoins	Développement incrémental de la partie la plus stable d'abord, masquage d'information, ...
Problèmes de performances	Simulations, modélisations, essais et mesures, maquettage, …
Exigences démesurées par rapport à la technologie	Analyses techniques de faisabilité, maquettage, ...
Tâches ou composants externes défaillants	Audit des sous-traitants, contrats, revues, analyse de compatibilité, essais et mesures, ...

Cycle de vie de développement logiciel

Le modèle incrémental

Face aux dérives bureaucratiques de certains gros développements, et à l'impossibilité de procéder de manière aussi linéaire, le *modèle incrémental* a été proposé dans les années 80.



Cycle de vie de développement logiciel

Le modèle incrémental

- Le développement est réaliser par suite d'incréments ou composantes, qui correspondent à des parties des spécification, et qui viennent intégrer le noyau de logiciel

- L'objectif est d'assurer la meilleur adéquation aux besoins possibles

- Le choix d'incrément est un compromis entre la durée de développement et le niveau de prise en compte des spécification

Cycle de vie de développement logiciel

Avantages du modèle incrémental

- Le système peut être mis en production plus rapidement, avec la diffusion ensuite sous forme de version
- Les premiers incréments permettent de renforcer l'expression de besoin et autorisent la définition des incrément suivant
- Diminution du risque global d'échec du projet
- Les fonctions les plus utilisées seront les plus testées et donc les plus robustes

Cycle de vie de développement logiciel

Risques du modèle incrémental

- Remettre en cause les incréments précédents ou pire le noyau ;
- Ne pas pouvoir intégrer de nouveaux incréments.
- Les noyaux, les incréments ainsi que leurs interactions doivent être spécifiés globalement, au début du projet. Les incréments doivent être aussi indépendants que possibles, fonctionnellement mais aussi sur le plan du calendrier du développement.

Cycle de vie de développement logiciel

Extreme Programming (XP)

- Nouvelle approche de développement incrémental basée sur la réalisation rapide, en équipe, d'incréments fonctionnels

- **Caractéristiques :**
  - Emphase sur la satisfaction du client
    - le produit dont il a besoin
    - des livraisons aux moments où il en a besoin
    - un processus robuste si les exigences sont modifiées
  - Emphase sur le travail d'équipe
    - managers, clients, développeurs : ensemble

Cycle de vie de développement logiciel

Valeurs de l'Extreme Programming (XP)

- **Communication:**
  - Au sein de l'équipe de développeurs et avec le client
- **Feedback**
  - Itérations rapides permettant la réactivité, test du code dès le début
- **Simplicité**
  - Code simple livrable ne contenant que les exigences du clientavec un design propre et simple
- **Courage**
  - Dans le cas des choix difficiles, peut être facilité par les valeurs précédentes

Cycle de vie de développement logiciel

Pratique de l'Extreme Programming (Equipe)

- **Responsabilité collective du code :** polyvalence e des développeurs, contribution collective aux nouvelles fonctionnalités, à la correction de bogues et à la restructurations (pas de droits exclusifs)
- **Programmation en binôme :** partage des compétence, prévention des erreurs, motivation mutuelle
- **Langage commun \ Métaphore** : Les mots utilisés pour désigner les entités techniques doivent être choisis parmi les métaphores du domaine de projet
- **Rythme durable :** maintenir un niveau optimal de travail entre énergie nécessaire pour le développement et le repos réparateur

Cycle de vie de développement logiciel

Pratique de l'Extreme Programming (Code)

- **Refactoring** : Investissement pour le futur, réduction de la dette technique
- **Conception simple** : facilité des reprises du code, facilité de l'ajout de fonctionnalité
- **Tests unitaires**: Test en premier pour une programmation ultérieure plus facile et plus rapide, fixe la spécification (ambiguïtés) avant le codage, feedback immédiat lors du développement, itération du processus pour tout tester
- **Intégration continue** : Ajout de valeurs chaque jour, évite la divergences, efforts fragmentés et permet la diffusion rapide pour la réutilisation
- **Règles de codage** : cohérence globale, code lisible / modifiable par toute l'équipe

Cycle de vie de développement logiciel

Pratique de l'Extreme Programming (Projet)

- **Client sur site** : orienter le développement en fonction des réactions des usagers et des points critiques du domaine d'utilisation
- **Tests d'acceptation** : conformité par rapport aux attentes du client
- **Planification itérative et livraison fréquente**
  - Les fiches sont rédigés tout au long du projet
  - Une fiche correspond à une fonctionnalité (scénario)
  - Sur la fiche figure la description du scénario, l'ordre de priorité, la durée de réalisation, les noms des développeurs…

Cycle de vie de développement logiciel

Le cycle standard XP

1. Le client écrit ses besoins sous forme de scénario
2. Les développeurs évaluent le coût de chaque scénario, si le coût est trop élevé pour un scénario ou difficile à estimer, le client le découpe en plusieurs scénario et les soumet aux développeurs
3. Le client choisi, en accord avec les développeur, les scénario à intégrer à la prochaine livraison
4. Chaque binôme des développeur prend la responsabilité d'une tâche

Cycle de vie de développement logiciel

Le cycle standard XP (suite)

5. Le binôme écrit les tests unitaires correspondant au scénario à implémenter
6. Le binôme procède à l'implémentation du programme
7. Le binôme réorganise le code (refactoring)
8. Le binôme intègre ses développement à la version d'intégration, en s'assurant que tous les tests de non régression passent

Cycle de vie de développement logiciel

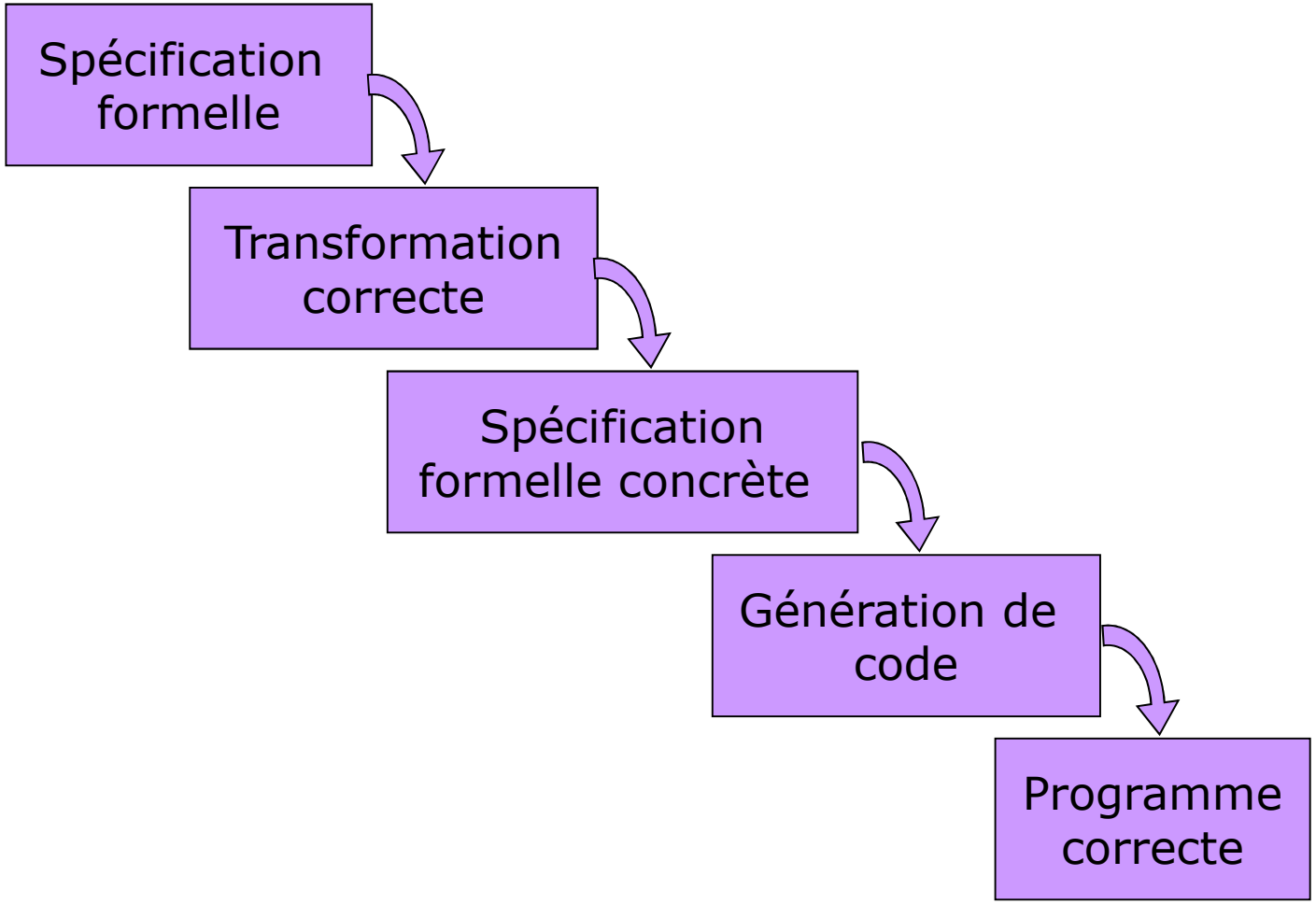
Le modèle avec méthode formelle

S'appuie sur la transformation d'une **spécification formelle**, c'est-à-dire définie à partir de structure logique et mathématique, en un programme exécutable via une suite de **représentation formelles intermédiaires**

Chaque information est **prouvée correcte** par rapport à la précédente, permettant d'assurer que l'implantation finale est correcte par aux spécifications formelles précédente

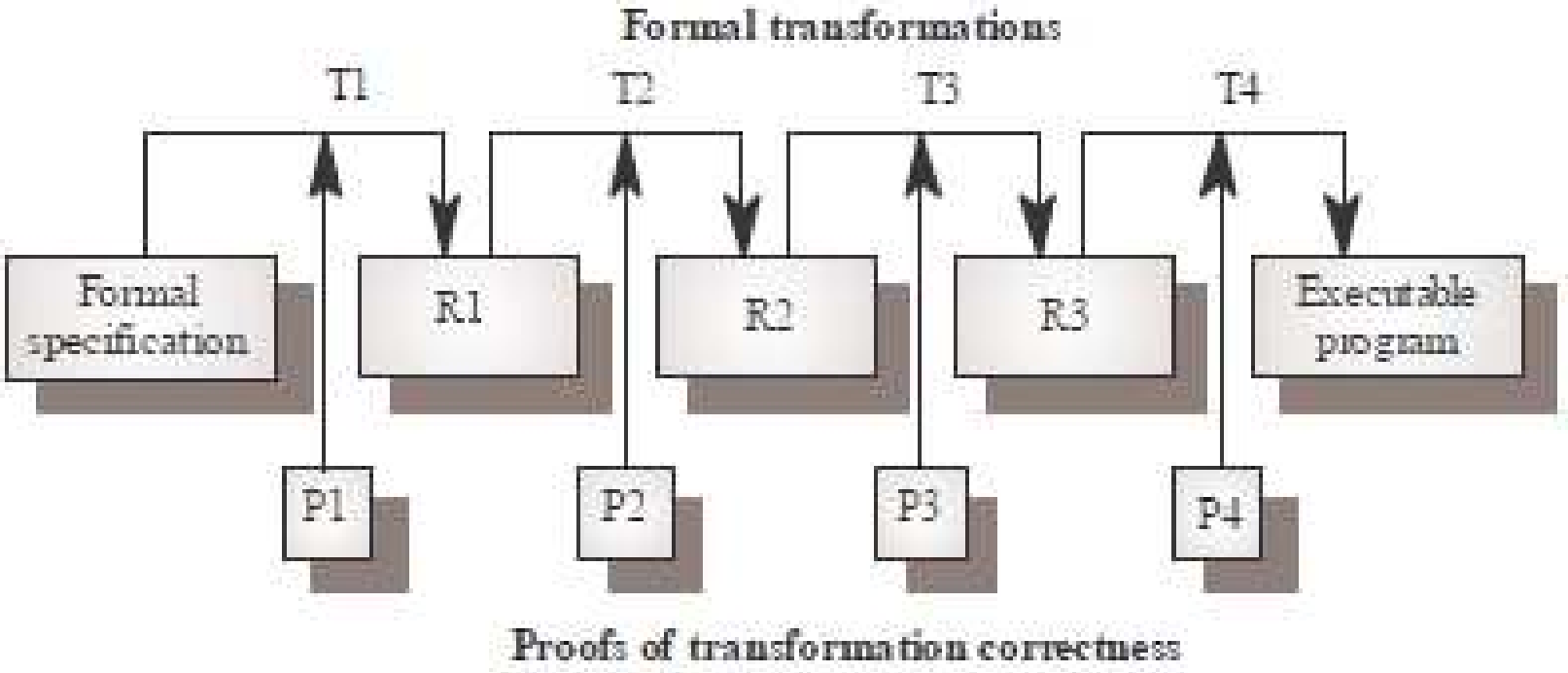
Cycle de vie de développement logiciel

Développement formel d'un système



Cycle de vie de développement logiciel

Raffinement de la transformation formelle



Cycle de vie de développement logiciel

Développement formel d'un système

- **Problèmes**
  - Nécessite de l'équipe de développement des connaissances en formalisation et preuves du fait de l'automatisation partielle des preuves
  - Certains aspects du système sont difficiles à formaliser (ex interface graphique)
- **Applications**
  - Logiciels critiques, mettant en jeu la sécurité des personnes (système de transport, spatial, aéronautique…) ou pour lesquels la présence d'une faute peut avoir des conséquences très importantes pour l'entreprise (logiciel Cartes à Puces)

Cycle de vie de développement logiciel

Le modèle avec réutilisation de composants

Cycle de vie de développement logiciel

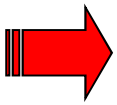
Le modèle avec réutilisation de composants

- Développer un logiciel à l'aide d'une base de composants génériques pré-existants.
- L'élaboration de la spécification est dirigée par cette base : une fonctionnalité est proposée à l'utilisateur en fonction des composants existants
- Ce modèle permet d'obtenir rapidement des produits de bonne qualité (utilisation des composants prouvés)
- Le travail d'intégration peut s'appuyer sur des outils dirigés par des descriptions de haut niveau du système pour générer le code
- Situation typique chez les sociétés de services (hébergement de serveurs, déploiement automatique de site Web, . . .).

Cycle de vie de développement logiciel

Etapes des processus

- Analyse des composants
- Besoins en modification
- Conception avec réutilisation
- Développement et intégration



Cette approche est très **répondu** car elle peut permettre d'important réduction de **coût**

Cycle de vie de développement logiciel

Conclusion

- Il n'y a *pas de modèle idéal car tout dépend des circonstances*
- Le modèle en cascade ou en V est risqué pour les développements innovants car les spécifications et la conception risquent d'être inadéquats et souvent remis en cause.
- Le modèle incrémental est risqué car il ne donne pas beaucoup de visibilité sur le processus complet.
- Le modèle en spirale est un canevas plus général qui inclut l'évaluation des risques.
- Souvent, un même projet peut mêler *différentes approches*, comme le prototypage pour les sous-systèmes à haut risque et la cascade pour les sous systèmes bien connus et à faible risque.

Test logiciel

Définition

Le test est une activité importante pour la **recherche d'anomalie** dans le comportement du logiciel dont le but est d'arriver à un produit **« zéro défaut ».** Examen d'une, de plusieurs unités ou d'un ensemble intégré de composants logiciels par exécution

Le test est un processus **manuel ou automatique**, qui vise à établir qu'un système vérifie les propriétés exigées par sa spécification, ou à détecter des différences entre les résultats engendrés par le système et ceux qui sont attendus par la spécification

Le test est un processus d'inférence de certaines propriétés de **comportement d'un produit,** basé, en partie, sur les résultats de l'exécution du produit dans un environnement connu et avec des données d'entrées choisies

Test logiciel

Terminologie

Faute :La cause d'une erreur, résultat d'*erreurs humaines*

Erreur : Un écart entre une valeur ou condition calculée, observée ou mesurée et la valeur ou condition qui est vraie, spécifiée ou théoriquement correcte.

Défaut, anomalie : La manifestation d'une erreur dans un logiciel. Un défaut peut causer une panne.

Panne : La fin de la capacité d'un système ou d'un de ses composants d'effectuer la fonction requise, ou de l'effectuer à l'intérieur de limites spécifiées.

Jeux de test : ensemble de données de test

Oracle de test : prédiction du résultat

Scénario de test : séquence d 'actions

Test logiciel

Objectifs

- Mettre en évidence les erreurs d'un logiciel : Détecter des défauts avant qu'ils ne causent une panne du système en production.
- Compiler un ensemble de situation d'erreurs devant servir à la prévention des erreurs (par des actions correctives et préventives).
- Amener le logiciel testé à un niveau acceptable de qualité (après la correction des défauts identifiés et retestage).
- Effectuer les tests requis de façon efficiente et effective dans les limite de temps et budget définis.
- Vérifier que le logiciel permet une implémentation correcte et cohérente des spécifications fonctionnelles et de performance

Test logiciel

Difficultés

- **Le test exhaustif est en général impossible à réaliser** : le test est une méthode de vérification partielle de logiciels et la qualité du test dépend de la pertinence du choix des données de test
- **Processus d'introduction des défaut** : manipulation de données abstraites et succession de transformations, perte d'informations, introduction d'erreurs, …
- **Difficultés d'ordre psychologique ou «culturel» :** Les erreurs peuvent être dues à des incompréhensions de spécifications ou de mauvais choix d'implantation
- **Difficultés formelles :** aucun algorithme général (donc aucun outil), capable de démontrer l'exactitude totale d'un programme

Test logiciel

Stratégies de test

- Une stratégie de test devrait être idéalement une grande organisation à l'image d'une organisation de développement.

- Enoncé de l'approche générale du test, en identifiant
  - Types de tests qui seront effectués pour les différentes étapes du projet et comment ils seront effectués
  - Définition des couvertures de tests.
  - Choix des outils
  - Moyens et délais à investir dans l'activité de tests (effort)

- Développer une stratégie de test, qui répond efficacement aux besoins d'une organisation, est critique pour assurer le succès du développement des applications. L'application d'une stratégie de test à un projet de développement devrait être détaillée dans le **plan qualité** du projet.

Test logiciel

Objectifs de stratégies de test

- Concevoir les tests avant que le logiciel ne soit réalisé
- Rendre l'effort de test efficace pour la détection des erreurs
- Planifier le test pour satisfaire les besoins ou la conception du logiciel
- Définir les critères d'arrêt des tests
- Définir des dossiers de test (scenario)
- Etablir des procédures de tests
- Archiver des résultats de tests

Test logiciel

Techniques de tests

- Techniques utilisées durant les phases de tests (types de tests)
- Techniques fonctionnelles, structurelles , dynamiques, statiques, flots de données, graphes cause-effet, …

Types de tests

- Catégories de tests effectués tout au long de la vie du logiciel.
- Tests unitaires, tests d'intégration, tests système, tests de non régression,…

Test logiciel

Test fonctionnel (boite noire)

- Tests sans la connaissance du code sousjacent
- Tests basés sur la *spécification*

Test logiciel

Test fonctionnel (boite noire)

Avantages

- pas besoin du code source
- s'applique aussi bien aux tests unitaires que systèmes

Désavantages

- ne test pas les fonctions cachées

Approches:

- Divination d'erreurs (Erreur guessing)
- Partitionnement en Classe d'équivalences
- Analyse de Valeurs aux Bornes
- Tables de Décision
- Graphes Causes-Effets
- Tests dérivés des exigences (ex: cas d'utilisations)

Test logiciel

Divination d'erreurs

- **Approche adhoc basée sur l'expérience**
- **Approche:**
  1. Faire une liste d'erreurs possibles ou situations conduisant à des erreurs (modèle d'erreurs)
  2. Développer des cas de tests pour couvrir le modèle d'erreurs
- **Développez et maintenez vos modèles d'erreurs**
- **Exemple :** fonction de tri de tableau
  - **Modèle d'erreur:**
    - tableau vide
    - tableau trié
    - tableau trié à l'envers
    - grand tableau non trié
  - **Générer des cas de tests pour ces situations**

Test logiciel

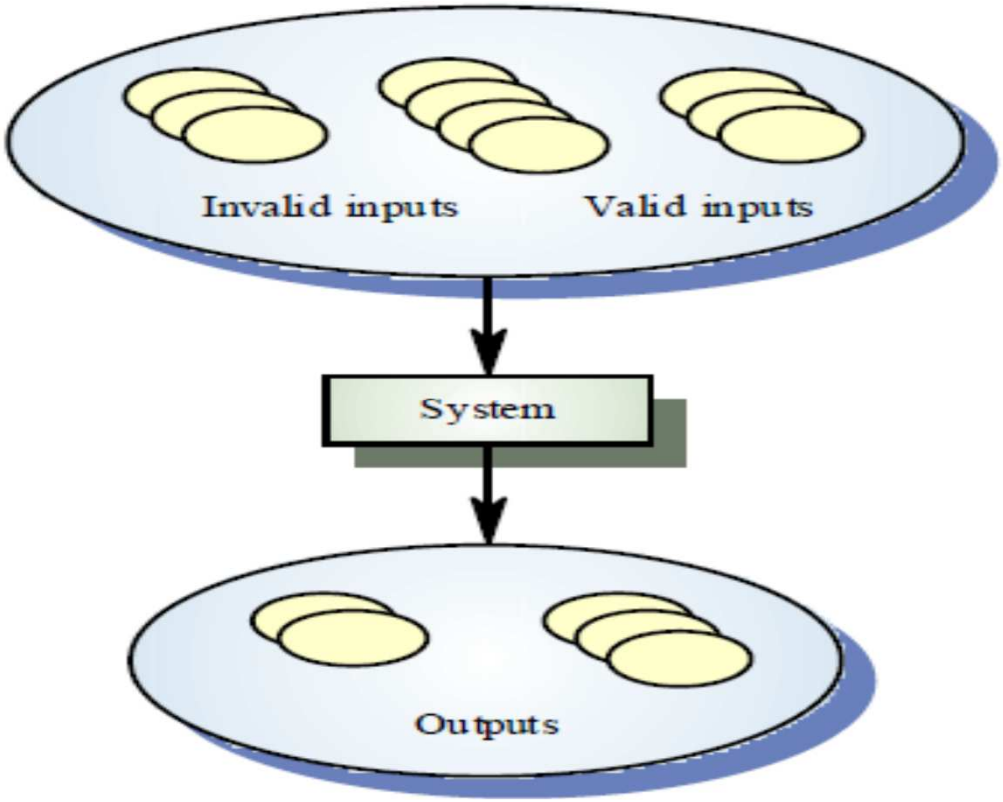
Partitionnement en Classes d'équivalences

- Partition du domaine de données en classes d'équivalences selon la spécification
- Idéalement les CEs devraient être
  - telles que chaque donnée est dans une classe
  - Disjointes
  - les éléments d'une même classe sont mis en correspondance avec leur résultats de façon similaire
- En pratique il est difficile de déterminer les CE : usage d'heuristiques
- Considérez la création de classes d'équivalences pour les conditions suivantes: défaut, vide, blanc, nulle, zéro, ou aucune.

Test logiciel

Classes d'équivalences

Une classe d'équivalence correspond à un ensemble de données de tests supposées tester le même comportement, c'est-à-dire activer le même défaut.



Test logiciel

Heuristiques d'identification de CE

Pour chaque donnée:

1. **Si spécifie un intervalle de valeurs valides, définir**
   - 1 CE valide (dans l'intervalle)
   - 2 CE invalides (une à chaque bout de l'intervalle)
2. **Si spécifie un nombre (N) de valeurs valides, définir**
   - 1 CE valide
   - 2 CE invalides (aucune et plus de N)
3. **Si spécifie un ensemble de valeurs valides, définir**
   - 1 CE valide (dans l'ensemble)
   - 1 CE invalide (en dehors de l'ensemble)

Test logiciel

Heuristiques d'identification de CE

Pour chaque donnée (suite):

4. **Si il y a une raison de croire que le programme traite les données valides différemment, défini**
   - 1 CE valide par donnée valide
5. **Si spécifie une situation *doit être*, définir**
   - 1 CE valide (satisfaisant le *doit être*)
   - 1 CE invalide (ne satisfaisant pas le *doit être*)
6. **Si il y a une raison de croire que les éléments d'une CE ne sont pas traités de façon identiques par le programme**
   - subdiviser la CE en plus petites CE.

Test logiciel

Exemple de Classes d'équivalences

- **Spécification**
  - données trois entiers (côtés du triangle: *a*, *b*, *c*) : chaque côté doit être un nombre positif inférieur ou égal à 20.
- **Résultat type du triangle:**
  - Équilatéral si *a* = *b* = *c*
  - Isocèles si 2 paires de côté sont égaux
  - Scalène si aucun côté n'est égal à l'autre ou
  - PasUnTriangle si *a* >= *b* + *c*, *b* >= *a* + *c*, *ou c* >= *a* + *b*
- **Selon l'heuristique #5 puis #6**

Condition de donnée	CE Valide	CE Invalide
Côtés (a, b, c)	ous > 0 et <= 20 (1	a>20 (2) b>20 (3) c>20 (4) a <= 0 (5) b <= 0 (6) c <= 0 (7)

Test logiciel

Exemple de Classes d'équivalences

CE #1 trop large peut être subdivisée (heuristique #6)

- **Basé sur le traitement de données**

  1.1. a, b, c tel que le triangle est équilatéral

  1.2. a, b, c tel que le triangle est isocèle

  1.3. a, b, c tel que le triangle est scalène

  1.4. a, b, c tel que ce n'est pas un triangle

- **Basé sur les données**

  1.5. a = b = c

  1.6. a = b, a <> c

  1.7. a = c, a <> b

  1.8. b = c, a <> b

  1.9. a <> b, a <> c, b <> c

- **Basé sur la propriété de triangle**

  1.10. a, b, c tel que a >= b + c

  1.11. a, b, c tel que b >= a + c

  1.12. a, b, c tel que c >= a + b

Test logiciel

Identification des cas de test de CE

1. Jusqu'à ce que toutes les **CE valides soient couvertes** par des cas de tests,
   – écrire un nouveau cas de test couvrant **le plus** de CE valides non couvertes possible
2. Jusqu'à ce que toutes les **CE invalides soient couvertes** par des cas de tests,
   – écrire un nouveau cas de test couvrant ***une et seulement une*** des CE invalides non couvertes

Test logiciel

Exemple d'identification des cas de test de CE

Cas de Tests pour triangle

CE	a	b	c	Résultat attendu
1.1, 1.5	12	12	12	Equilatéral
1.2, 1.6	6	6	7	Isocèles
1.3, 1.9	9	3	7	Scalène
1.4, 1.10	8	2	3	Pas un triangle
1.13	-5	-5	-5	
2	30	15	6	
3	10	100	99	
4	7	14	21	
5	-5	10	11	
6	12	-10	10	
7	8	5	-1	

Test logiciel

Problèmes de partitionnement en CE

- Spécification ne définie pas toujours les résultats attendus pour les cas de **tests *invalides***
- Langages **fortement typés** éliminent le besoin de la considération de certaines données invalides
- ***L´approche forcebrute*** de définition de cas de test pour chaque combinaison de CEs
  - donne une bonne couverture, mais
  - non pratique lorsque le nombre d'entrées et classes associé est grand

Test logiciel

Exercice : nextDate

- **Données: *mois*, *jour, an* représentant une *date***
  - *1 <= mois <= 12*
  - *1 <= jour <= 31*
  - *1812 <= an <= 2012*
- **Résultat: date du jour suivant la date donnée**
  - Doit considérer les années bissextiles
  - Année bissextile si divisible par 4 et pas siècle
  - Année siècle bissextile si multiple de 400

Test logiciel

Exercice : nextDate - corrigé

Classes d'équivalences

Condition de donnée	CEs Valides	CEs Invalides
mois	1 <= mois <= 12	mois < 1 mois > 12
jour	1 <= jour <= 31	jour < 1 jour > 31
an	1812 <= an <= 2012	an < 1812 an > 2012

Raffinement possible selon la façon dont les données sont traitées

- Jour incrémenté de 1 à moins que dernier jour d'un mois
- Dépend du mois (30, 31, Février)
- A la fin du mois, prochain jour est 1 et mois est incrémenté
- A la fin d'une année, jour et mois remis à 1, et an incrémenté
- Problème d'années bissextiles

Test logiciel

Exercice : nextDate - corrigé

Décomposition des CEs valides

Condition de données	CEs Valides
mois	30 jours (1)
	31 jours (2)
	Février (3)
jour	1 <= jour <= 28 (4)
	jour = 29 (5)
	jour = 30 (6)
	jour = 31 (7)
an	an = 1900 (8)
	1812 <= an <= 2012 et an <> 1900 et an mod 4 = 0 (9)
	1812 <= an <= 2012 et an mod 4 <> 0 (10)

Meilleure couverture obtenue en sélectionnant les cas de tests selon le produit Cartésien des CEs

➔ 36 cas de tests.

Test logiciel

Exercice : nextDate - corrigé

cas de tests

ID	CE	mois	jour	an	résultat
1	1, 4, 8	6	14	1900	15/06/1900
2	1, 4, 9	6	14	1912	15/06/1912
3	1, 4, 10	6	14	1913	15/06/1913
4	1, 5, 8	6	29	1900	30/06/1900
5	1, 5, 9	6	29	1912	30/06/1912
6	1, 5, 10	6	29	1913	30/06/1913
7	1, 6, 8	6	30	1900	01/07/1900
8	1, 6, 9	6	30	1912	01/07/1912
9	1, 6, 10	6	30	1913	01/07/1913
10	1, 7, 8	6	31	1900	Erreur
11	1, 7, 9	6	31	1912	Erreur
12	1, 7, 10	6	31	1913	Erreur

Test logiciel

Analyse des Valeurs aux Bornes (AVB)

- Erreurs ont tendance à survenir vers les valeurs extrêmes (*bornes*)
- AVB **améliore** le Partitionnement en Classes d'équivalences en
  - Sélectionnant les éléments juste à et autour des **bornes** de chacune des CEs
  - Dérivant des cas de tests en considérant des CEs des **résultats** également
- **Conditions de bornes**
  - Situation à la bordure des limites opérationnelles envisagées
  - Types de données ayant des conditions de bornes : Numérique, Caractère, Position, Quantité, Vitesse, Location, Taille
  - Caractéristiques de conditions de bornes

Test logiciel

Directives de l'AVB

- **Pour chaque condition de borne**
  - Ajouter la valeur de borne correspondante dans au moins un cas de test valide
  - Ajouter la valeur juste au delà (ou en deçà) de la valeur de borne correspondante dans au moins un cas de test invalide.
- **Appliquer les mêmes directives pour les résultats**
  - inclure des cas de tests avec des données, tels que des résultats aux bornes sont produits
- **Conditions de bornes secondaires**
  - internes au logiciel (non définis dans la spécification)
  - exemples: Puissances de deux (bit, kilo, mega…), table ASCII

Test logiciel

Tables de Décisions

- **Idéales pour situations où:**
  - une combinaison d'actions est sélectionnée sous un ensemble de conditions variables
  - les conditions dépendent des variables de données
  - la réponse produite ne dépend pas de l'ordre d'assignation ou évaluation des données
  - la réponse produite ne dépend pas d'entrées/sorties précédentes
- **Format**

Conditions	Combinaison de conditions (variantes)
Actions	Actions sélectionnées

Test logiciel

Exercice

- **Considérons le cas précédent :** données trois entiers (côtés du triangle: *a*, *b*, *c*) : chaque côté doit être un nombre inférieur ou égal à 20.
- **Données** : a, b et c
- **Conditions** : a < b+c ; b < a+c ; c < a+b ; a=b ; a=c ; b=c ; a > 20 ; b > 20 ; c > 20
- **Actions** : déterminer que abc est un triangle **équilatéral**, **isocèle** ou **scalène** ; un **non triangle** ; une **erreur** ou une **variante impossible**

Test logiciel

Exercice

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
c1: a<b+c	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-
c2: b<a+c	-	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-
c3: c<a+b	-	-	N	Y	Y	Y	Y	Y	Y	Y	Y	-	-	-
c4: a=b	-	-	-	Y	Y	Y	Y	N	N	N	N	-	-	-
c5: a=c	-	-	-	Y	Y	N	N	Y	Y	N	N	-	-	-
c6: b=c	-	-	-	Y	N	Y	N	Y	N	Y	N	-	-	-
c7: a>20	N	N	N	N	N	N	N	N	N	N	N	Y	-	-
c8: b>20	N	N	N	N	N	N	N	N	N	N	N	-	Y	-
c9: c>20	N	N	N	N	N	N	N	N	N	N	N	-	-	Y
a1: not a triangle	X	X	X											
a2: Scalene											X			
a3: Isosceles							X		X	X				
a4: Equilateral				X										
a5: impossible					X	X		X						
a6: error												X	X	X

Test logiciel

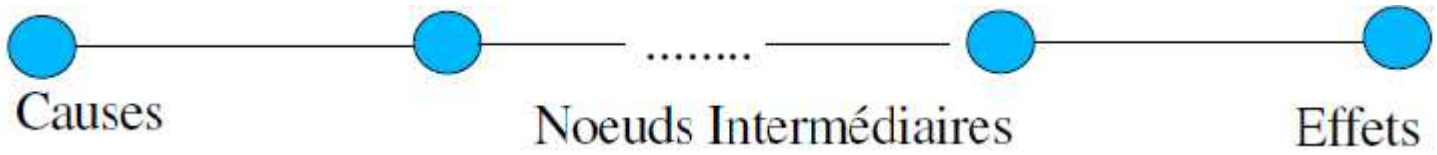
Graphe causes-effets

- **Approche systématique** d'aide à la sélection d'ensemble de cas de tests à haute efficacité
  - identification/analyse de relations d'une table de décision
  - génération de formule booléenne
- **Nœuds:**
  - Cause – instance distincte de condition donnée ou CE
  - Effet – résultat observable d'un changement de l'état du système
  - Nœud Intermédiaire combinaison de causes représentant une expression sur des conditions de données

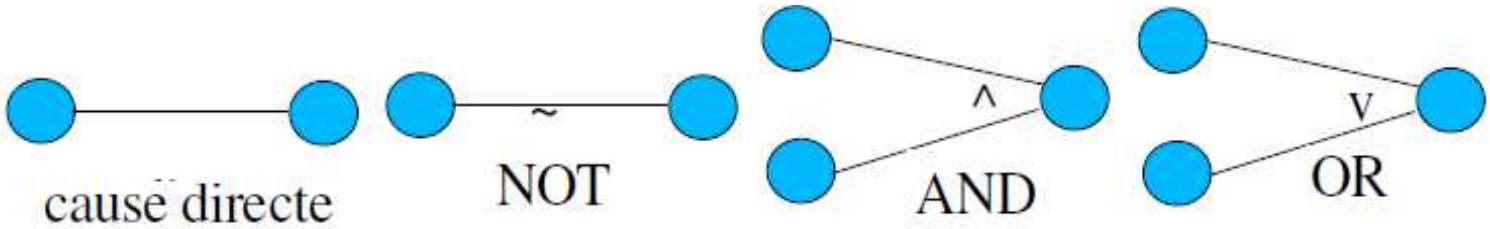
Test logiciel

Graphe causes-effets

- **Liens valides de** : cause à nœud intermédiaire, cause à effets, nœuds intermédiaire à nœuds intermédiaire et nœuds intermédiaire à effet

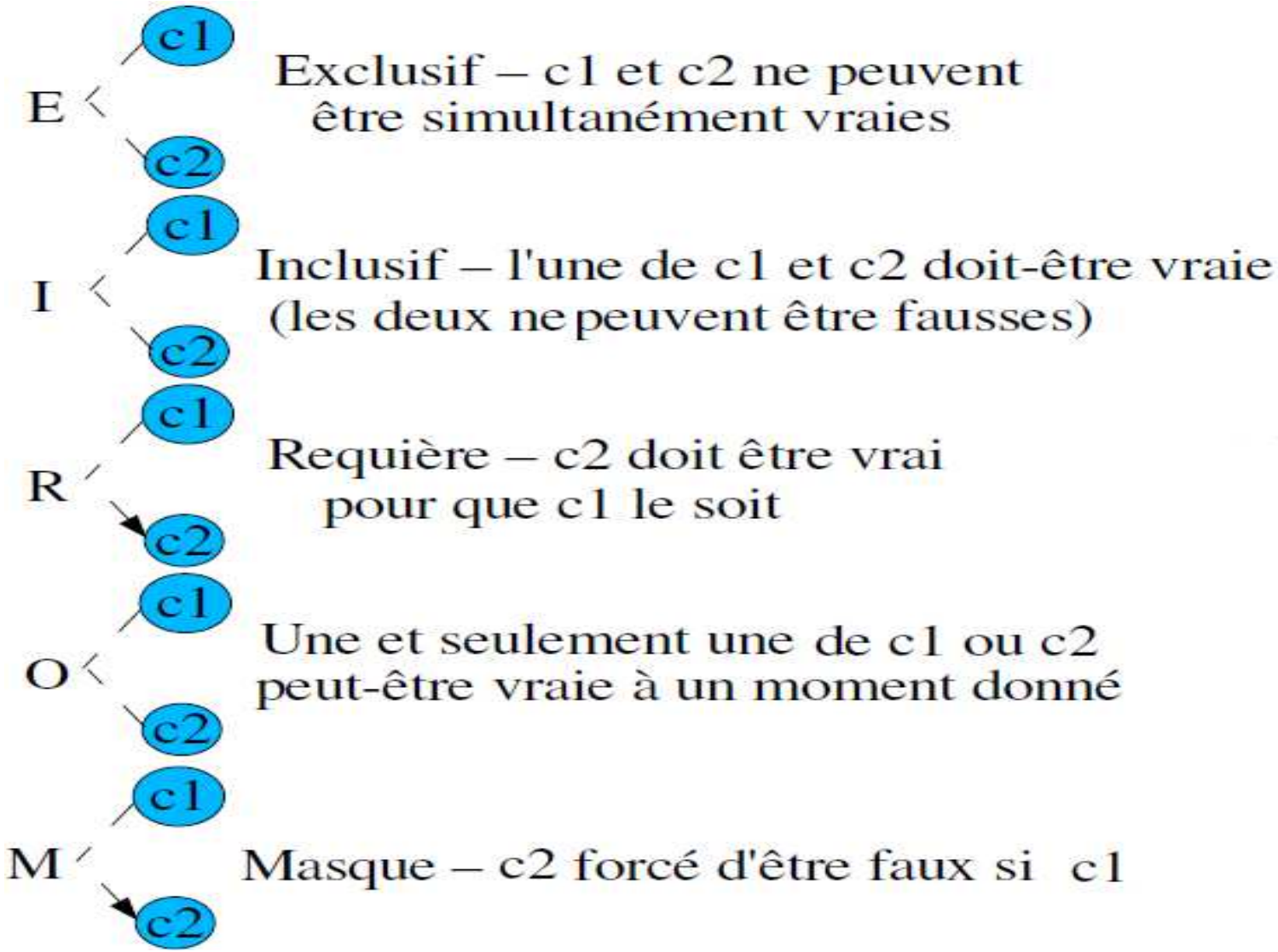


- **Types de liens**



Test logiciel

Contrainte entre causes



Test logiciel

Exemple

Une mise à jour de fichier dépend de la valeur dans deux champs. La valeur dans le champs 1 doit être ``A'' ou ``B ''. La valeur dans le champs 2 doit être un chiffre. Dans cette situation la mise à jour du fichier est faite. Si la valeur dans le champs 1 est incorrecte, le message d'erreur X12 est affiché. Si la valeur dans le champs 2 est incorrecte, le message d'erreur X13 est affiché.

Causes

– c1: caractère dans champs 1 est "A"

– c2: caractère dans champs 1 est "B"

– c3: caractère dans champs 2 est un digit

Effets

– e1: mise à jour effectuée

– e2: message X12

– e3: message X13

Test logiciel

Génération de cas de tests à partir de GCE

- **Décomposer la spécification en parties de taille raisonnable**
- **Définissez des graphes séparés par partie**
  1. Identifier les ***Causes et Effets***
  2. Déduisez les ***relations Logiques et Contraintes*** – représentez comme un graphe
  3. Développez une ***table de décision***
  4. Traduisez chacune des **variantes** de la table de décision en cas de test
- **Déterminer la combinaison de conditions uniques**
  1. En travaillant avec un seul effet à la fois
  2. Mettez l'effet dans l'état ***vrai*** (1)
  3. Recherchez en allant à reculons dans le graphe toutes les **combinaisons d'entrées** forçant l'effet à l'état vrai
  4. Créer une ***colonne*** dans la table de décision par combinaison
  5. Déterminez **l'état** de tous les autres effets pour chaque combinaison

Test logiciel

Exemple

Table de vérité

– Doit contenir l'ensemble des combinaisons des causes possibles. Donc ici 8 possibilités

– Le remplissage s'effectue en trouvant les effets à 1

DT	Causes			Effets		
	c1	c2	c3	e1	e2	e3
1	0	0	0	0	1	1
2	0	0	1	0	1	0
3	0	1	0	0	0	1
4	0	1	1	1	0	0
5	1	0	0	0	0	1
6	1	0	1	1	0	0
7	1	1	0			
8	1	1	1			

Cas impossibles vu les spécifications

Test logiciel

Exemple

Constats

– 2 DT pour tester le fichier à jour

– 2 DT pour tester le message M1

– 3 DT pour tester le message M2

DT	Causes			Effets		
	c1	c2	c3	e1	e2	e3
1	0	0	0		1	1
2	0	0	1		1	
3	0	1	0			1
4	0	1	1	1		
5	1	0	0			1
6	1	0	1	1		
7	1	1	0			
8	1	1	1			

Test logiciel

Génération de Formules Booléenne à partir de GCE

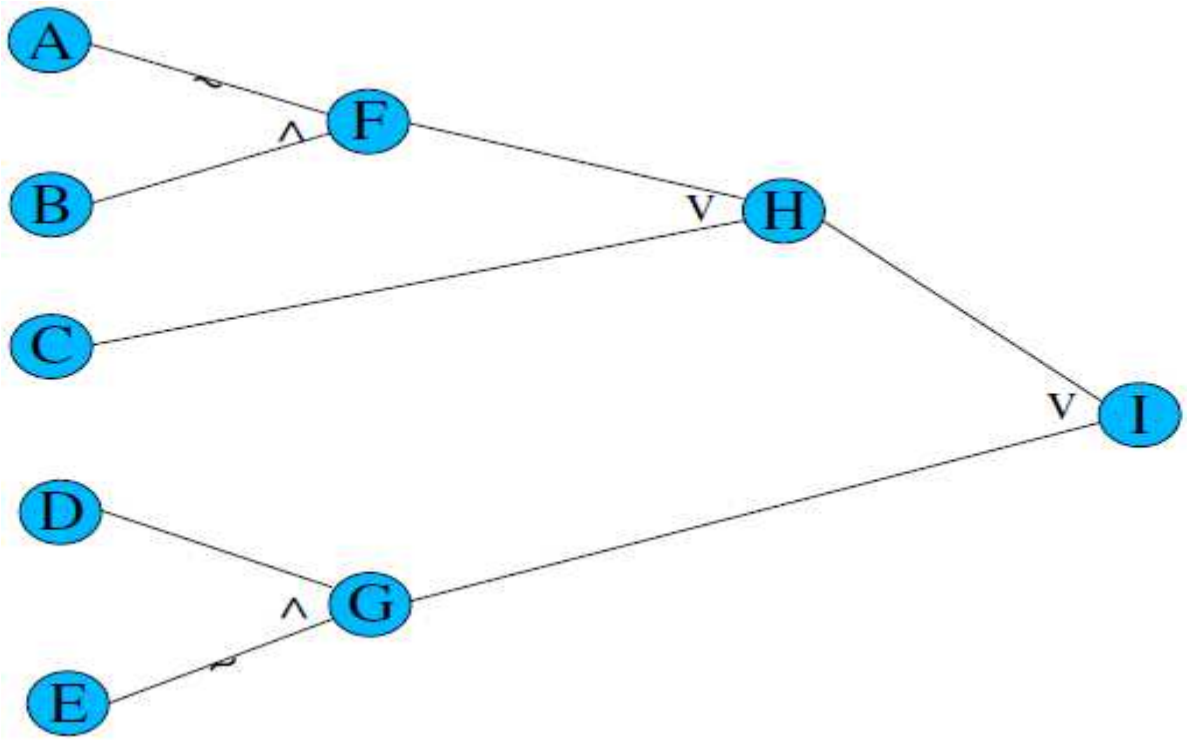
- **En allant des effets vers les causes**

1. **Transcrivez les formules nœud à nœud à partir du graphe**
   - écrire la formule correspondante à chaque effet ainsi que ses prédécesseurs
   - Pour chaque nœud intermédiaire : écrire la formule pour le nœud intermédiaire ainsi que ses prédécesseurs
2. **Dérivez la formule booléenne complète**
   - remplacer les variables intermédiaires par substitution jusqu'à ce que la formule résultante ne comprenne que des causes
   - factoriser et réécrire la formule sous forme de somme de produits

Test logiciel

Exemple

I = H ∨ G
H=F ∨ C
F=~A ∧ B
G=D ∧ ~E



I = (~A ∧ B) ∨ C ∨ (D ∧ ~E)

Test logiciel

Dérivation de CT d'exigences fonctionnelles

Implique la clarification et reformulation des exigences tel qu'elles soient testables

- **Obtenir une formulation testable (sous forme point)**
  - Enumérer les exigences ***uniques***
  - Grouper les exigences ***liées***
- **Pour chaque exigence développer**
  - Un cas de test montrant le **fonctionnement** de l'exigence
  - Un cas de test essayant de la **falsifier**
  - Tester les **bornes** et contraintes lorsque possible

Test logiciel

Exemple : Exigences d'un système de location de films

Le système permettra la location et retour de films

1. Si un film est **disponible** pour location, il peut être **prêté** à un client.
   1.1. Un film est disponible pour la location jusqu'à ce que toutes les copies aient été simultanément empruntées.
2. Si un film était **non disponible** pour la location, alors le **retour** du film le rend disponible.
3. La **date de retour** est établie quand un film est **prêté** et doit être montrée quand le film est retourné.
4. Il doit être possible de **déterminer l'emprunteur courant** d'un film loué par une requête sur celui ci.
5. Une requête sur un **membre** indiquera tous les films que celui-ci à en location

Test logiciel

Exemple : Exigences d'un système de location de films

- **Situations de tests pour l'exigence 1.**
  - Essayer d'emprunter un film disponible.
  - Essayer d'emprunter un film non disponible.
- **Situations de tests pour l'exigence 1.1**
  - Essayer d'emprunter un film pour lequel il y a plusieurs copies, toutes empruntées.
  - Essayer d'emprunter un film pour lequel toutes les copies sauf une ont été empruntées.
- **Situations de tests pour l'exigence 2**
  - Emprunter un film non disponible
  - Retourner un film et l'emprunter

Test logiciel

Exemple : Exigences d'un système de location de films

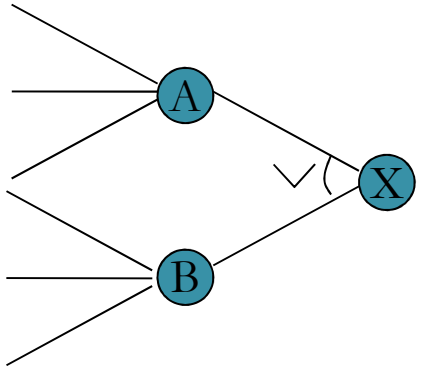
- **Situations de tests pour l'exigence 3**
  - Emprunter un film, le retourner et vérifier les dates.
  - Vérifier la date d'un film non retourné.
- **Situations de tests pour exigence 4**
  - Requête sur un film emprunté.
  - Requête sur un film retourné.
  - Requête sur un film venant d'être retourné.
- **Situations de tests pour exigence 5**
  - Requête concernant un membre sans films.
  - Requête concernant un membre avec 1 film.
  - Requête concernant un membre avec plusieurs films

Test logiciel

Règles de simplification

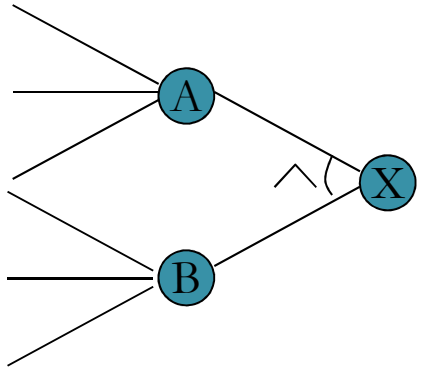
Forme Disjonctive (OU)

- **R1** : **si X doit être à 1**, ne pas prendre la situation ou **A=B=1**
- **R2** : **si X doit être à 0**, énumérer toutes les situations ou **A=B=0**



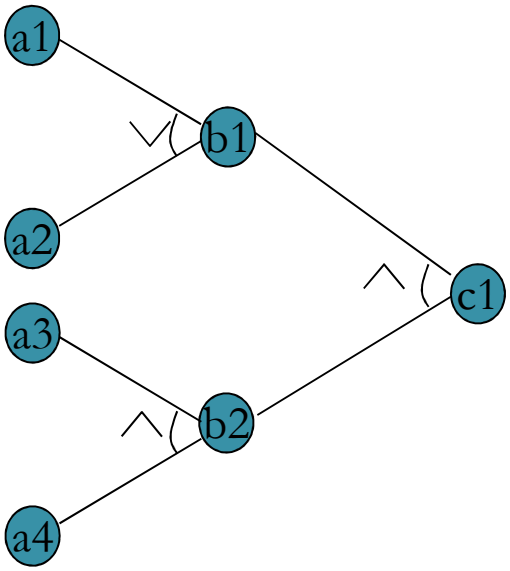
Forme Conjonctive (ET)

- **R3** : **si X doit être à 1**, énumérer toutes les situations ou **A=B=1**
- **R4** : **si X doit être à 0**,
  - **R4.1** Inclure une seule situation ou **A=B=0**
  - **R4.2** Pour les autres combinaisons, choisir une **seule** échantillon de causes générant cette valeur à **1**.



Test logiciel

Exemple



A1	A2	B1
0	0	0
0	1	1
1	0	1
1	1	1

A3	A4	B2
0	0	0
0	1	0
1	0	0
1	1	1

Pour chaque combinaison,
il faut prévoir toutes les valeurs
de A1, A2, A3, A4 les produisant.

B1	B2	C1	Nb DT
0	0	0	1*3=3
0	1	0	1*1=1
1	0	0	3*3=9
1	1	1	3*1=3

Avant simplification :

13 DTs pour C1=0

3 DTs pour C1=1

Test logiciel

Exemple

A1	A2	B1
0	0	0
0	1	1
1	0	1
1	1	1

A3	A4	B2
0	0	0
0	1	0
1	0	0
1	1	1

B1	B2	C1
0	0	0
0	1	0
1	0	0
1	1	1

Test logiciel

Exemple

A1	A2	B1
0	0	0
0	1	1
1	0	1
1	1	1

R1

A3	A4	B2
0	0	0
0	1	0
1	0	0
1	1	1

B1	B2	C1
0	0	0
0	1	0
1	0	0
1	1	1

Cas 1 : C1=1 ➔ R3

→B1=1

→{A1,A2}={0,1} ,{1,0}.

→R1 exlus {1,1}

→B2=1

→{A3,A4}={1,1}

➔DT={0,1,1,1},{1,0,1,1}

Test logiciel

Exemple

R1

A1	A2	B1
0	0	0
0	1	1
1	0	1
1	1	1

A3	A4	B2
0	0	0
0	1	0
1	0	0
1	1	1

B1	B2	C1
0	0	0
0	1	0
1	0	0
1	1	1

Cas 2 : C1=0

➔ R4.1

Une seule situation ou **B1=B2=0**

➔DT={0,0,0,0}

➔ R4.2

B1=0,B2=1

➔DT={0,0,1,1}

B1=1,B2=0

R4.2 : un seul échantillon générant le **B1=1**

➔DT={1,0}

B2=0

➔DT={0,0},{0,1}{1,0}

➔DTs={1,0,0,0}{1,0,0,1}{1,0,1,0}

Test logiciel

Exemple

Avant simplification

a1	A2	a3	a4	c1
0	1	1	1	**1**
1	0	1	1	**1**
1	1	1	1	**1**
0	0	0	0	**0**
0	0	0	1	**0**
0	0	1	0	**0**
0	0	1	1	**0**
0	1	0	0	**0**
0	1	0	1	**0**
0	1	1	0	**0**
1	0	0	0	**0**
1	0	0	1	**0**
1	0	1	0	**0**
1	1	0	0	**0**
1	1	0	1	**0**
1	1	1	0	**0**

Après simplification

a1	A2	a3	a4	c1
0	1	1	1	**1**
1	0	1	1	**1**
0	0	0	0	**0**
0	0	1	1	**0**
1	0	1	0	**0**
1	0	0	0	**0**
1	0	0	1	**0**

Test logiciel

Dérivation de CT à partir de Cas d'utilisations

Pour tous les cas d'utilisations

1. Développer un **Graphe de Scénarios**
2. Déterminer tous les scénarios possibles
3. Analyser et ordonner les scénarios
4. Générer des **cas de tests** à partir des scénarios pour atteindre l'objectif de couverture
5. Exécuter les cas de tests

Test logiciel

Graphe de Scénarios

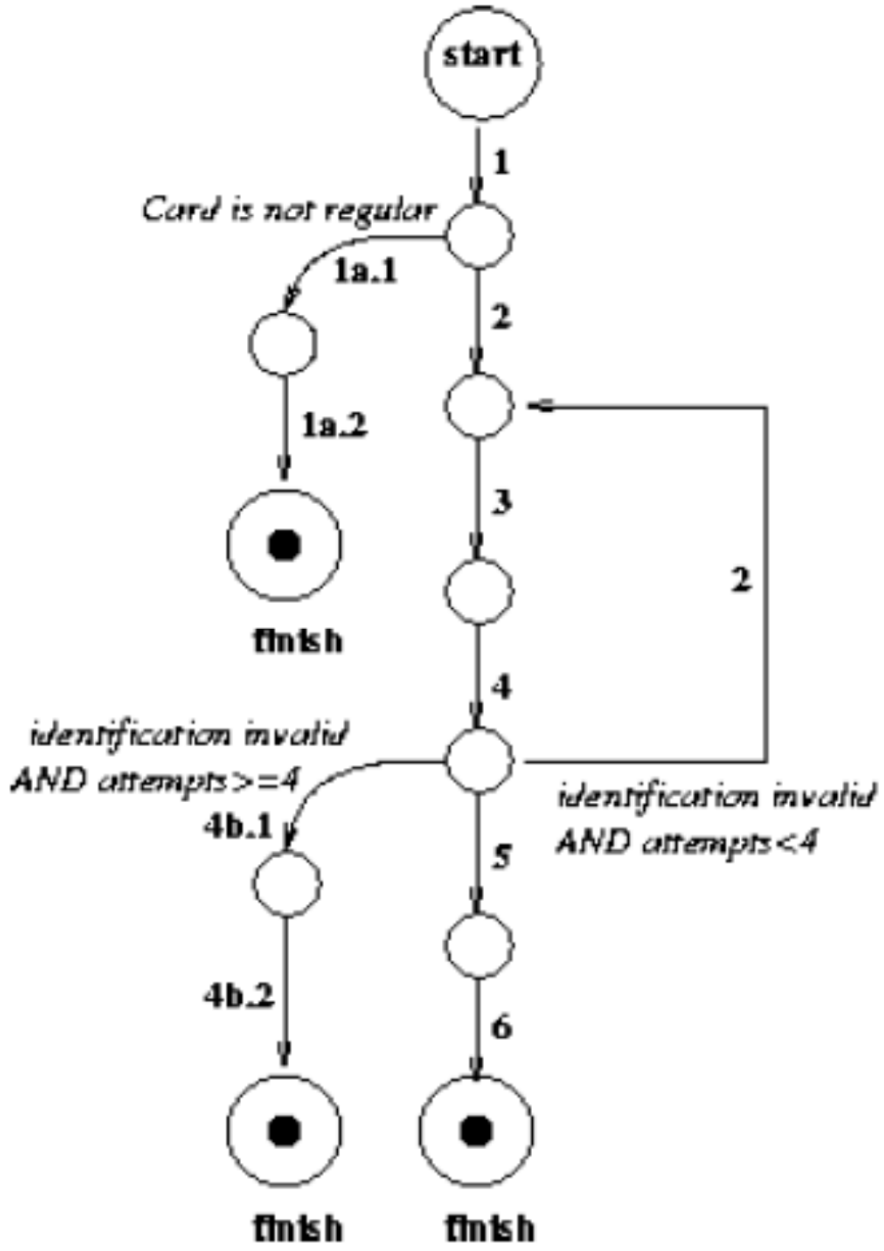
Généré d'un cas d'utilisation

- Nœuds correspondent à des **points d'attente d'événements** de l'environnement ou de réaction système
- Il y a un ***nœud de départ* unique**
- Fin du cas d'utilisation est un ***nœud de terminaison***
- **Arcs** correspondent à l'occurrence des événements : Peut inclure des *conditions* ou arc de boucle spécial
- **Scénario**: *chemin* de nœud de départ à nœud de terminaison

Test logiciel

Exemple

Title: User login
Actors: User
Precondition: System is ON
1: User inserts a Card
2: System asks for (PIN)
3: User types PIN
4: System validates USER identification
5: System displays a welcome to USER
6: System ejects Card
Alternatives:
1a: Card is not regular
1a1: System emits alarm
1a2: System ejects Card
4a: User identification is invalid
AND number of attempts < 4
4a.1 Go back to Step 2
4b: User identification is invalid
AND number of attempts >= 4
4b.1: System emits alarm
4b.2: System ejects Card
Postcondition: User is logged in



Test logiciel

Exemple-scénario

- Chemin du début à la fin
- Boucles doivent être restreintes pour obtenir un nombre fini de scénarios

Id	Events	Description
1	1 - 2 - 3 - 4 - 5 – 6	User login with regular card, right PIN on first trial. Normal course of events
2	1 - 1a.1 – 1a.2	User login with non regular Card
3	1 - 2 - 3 - 4 - 2 – 3 - 4 - 5 – 6	User login with regular card, wrong PIN on first trial, the right PIN on second trial
4	1 - 2 - 3 - 4 - 2* - 3 - 4 - 4b.1 – 4b.2	User login with regular card, wrong PIN on first, second, third and fourth trials

Test logiciel

Ordonnancement des Scénarios

- Si il l y a trop de scénarios pour tout tester
- Ordonnancement peut être basé sur criticalité et fréquence
- Inclure toujours le *scénario primaire :* devrait être testé en premier

Génération de Cas de Tests

- Selon l'objectif de couverture:
  - Toutes les branches du graphe de scénarios (objectif minimal)
  - Tous les Scénarios
  - *n* scénarios les plus critiques

Test logiciel

Exercice1

Supposons que nous élaborions un compilateur pour le langage BASIC. Un extrait des spécifications précise :

«L'instruction **FOR** n'accepte qu'un seul paramètre en tant que variable auxiliaire. Son nom ne doit pas dépasser deux caractères non blancs; Après le signe = est précisée aussi une borne supérieure et une borne inférieure. Les bornes sont des entiers positifs et on place entre eux le mot-cléTO. »

Déterminer par analyse partitionnelle des domaines des données d'entrée les cas de test à produire pour l'instruction FOR.

Test logiciel

Exercice1

DT obtenues par analyse partitionnelle pour l'instruction FOR:

–FOR A=1 TO 10	cas nominal
–FOR A=10 TO 10	égalité des bornes
–FOR AA=2 TO 7	deux caractères pour la variable
–FOR A, B=1 TO 8	Erreur -deux variables
–FOR ABC=1 TO 10	Erreur -3 caractères pour la variable
–FOR I=10 TO 5	Erreur -Borne sup < Borne inf
–FOR =1 TO 5	Erreur -variable manquante
–FOR I=0.5 TO 2	Erreur -Borne inf décimale
–FOR I=1 TO 10.5	Erreur -Borne sup décimale
–FOR I=7 10	Erreur -TO manquant

Test logiciel

Exercice2

Considérons les spécifications suivantes :

«Ecrire un programme statistique analysant un fichier comprenant les noms et les notes des étudiants d'une année universitaire. Ce fichier se compose au maximum de 100 champs. Chaque champ comprend le nom de chaque étudiant (20 caractères), son sexe (1 caractère) et ses notes dans 5 matières (entiers compris entre 0 et 20). Le but du programme est de :

–calculer la moyenne pour chaque étudiant,

–calculer la moyenne générale (par sexe et par matière),

–calculer le nombre d'étudiants qui ont réussi (moyenne supérieure à10)»

Déterminer par une approche aux limites les cas de test à produire pour cette spécification

Test logiciel

Exercice2-corrigé

DT obtenues par test aux limites pour l'exemple Etudiants :

– passage d'un fichier vide, puis comprenant 1 champ, 99, 100 et 101 (5 tests différents, la valeur -1 n'ayant pas de sens dans ce cas);

– inclure un nom d'étudiant vide, un nom avec des caractères de contrôle, un nom avec 19, puis 20, puis 21 caractères;

– inclure un code sexe vide, puis avec un caractère faux (C par exemple);

– avoir un étudiant sans aucune note et un avec plus de 5 notes;

– pour certains champs, les notes ne doivent pas être des nombres entiers, mais des caractères, des réels avec plusieurs décimales, des nombres négatifs ou des nombres entiers supérieurs à20.

 Les DT aux limites doivent être passées indépendamment : les erreurs peuvent se compenser.

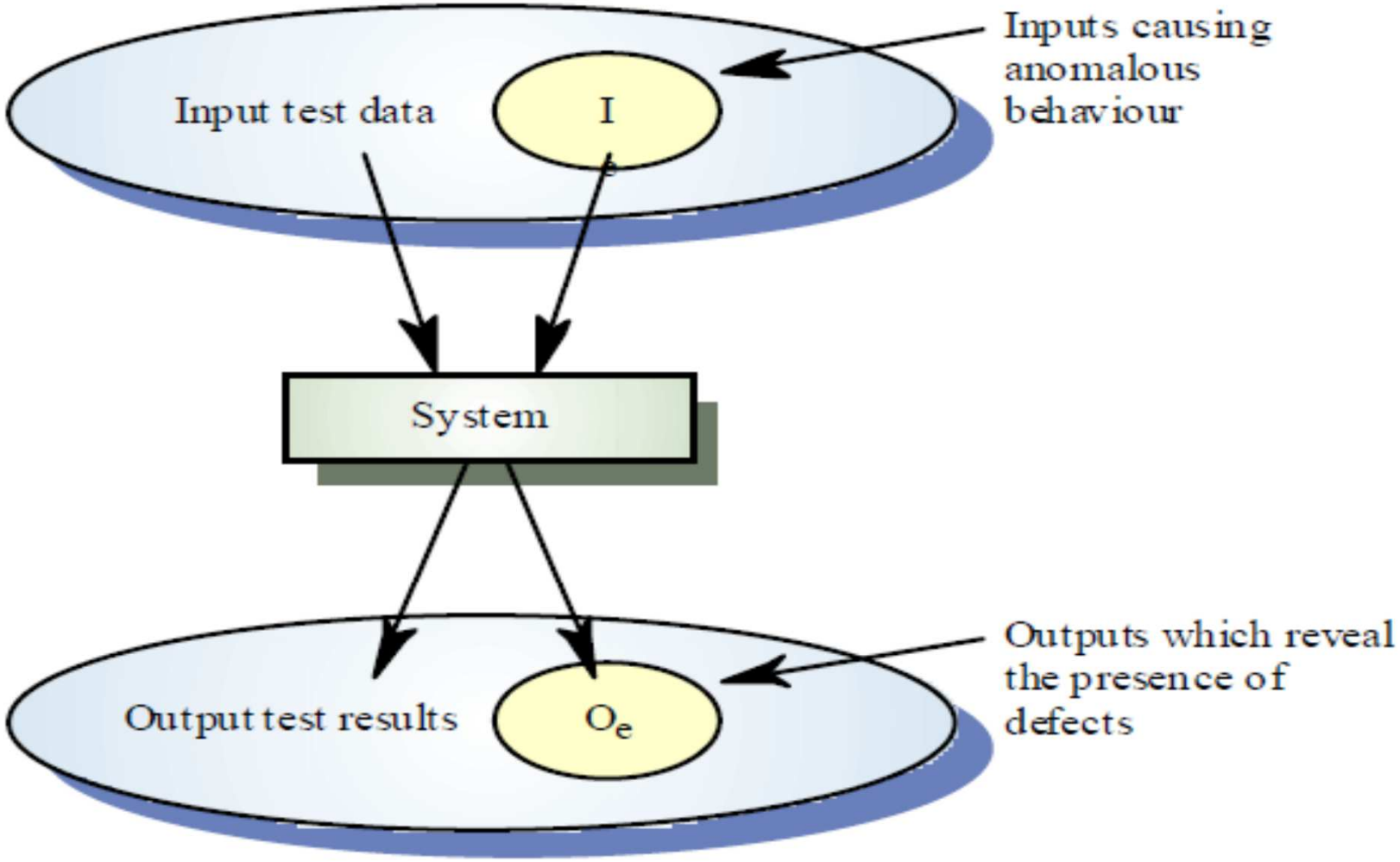
Test logiciel

Test structurel (boite blanche)

- Réalisés et menés à bien par le développeur
- Tester en fonction de la structure interne :classes, méthodes et fonction
- Etablir des jeux de test en fonction de la conception détaillée du programme
- Permettre de détecter des éventuelles erreurs commises… mais pas les omissions . Les erreurs ont tendance à se concentrer dans des chemins, instructions, conditions qui sont hors de l'« exécution normale »
- Analyser la structure interne du programme en déterminant les chemins minimaux. Afin d'assurer que:
  - Toutes les conditions d'arrêt de boucle ont été vérifiées.
  - Toutes les branches d'une instruction conditionnelle ont été testés.
  - Les structures de donnée interne ont été testées (pour assurer la validité).

Test logiciel

Test structurel (boite blanche)



Test logiciel

Approches de test structurel

- Approches orientées flot de contrôle
  - Basées sur l'analyse du flot de contrôle à travers le Programme
  - Exemples de critères de couverture de tests
    - couverture d'instructions (tous les nœuds)
    - couverture de branches (tous les liens)
    - couverture de chemins (tous les chemins)
    - couverture de conditions
- Approches orientées flot de données
  - Basées sur l'analyse du flot de données à travers le programme
  - Exemples de critères de couverture de tests : toutes-définitions, tous-usages, tous-paths…

Test logiciel

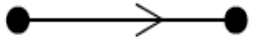
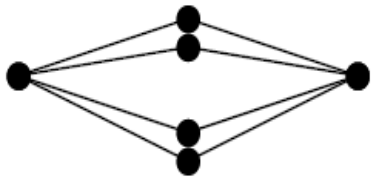
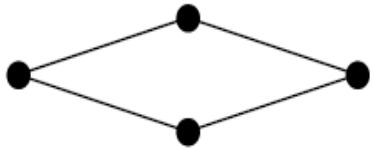
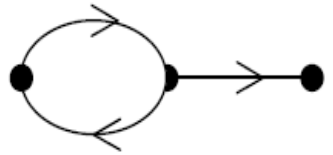
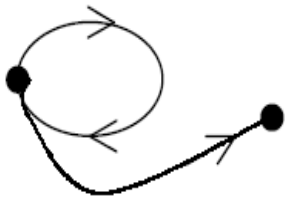
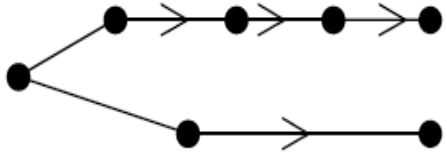
Graphe de flot de contrôle (GFC)

- Un programme structuré peut être représenté par un graphe de flot de contrôle
  - Les sommets ou nœuds représentent les instructions
  - Les arcs représentent le flot de contrôle entre les instruction
- Utilisation d'un GFC simplifié surtout pour les grandes unités
  - Obtenu en groupant des sous chemins de décision à décision
  - Nœuds Décisions: avec plus d'un lien partant fini avec une décision
  - Nœuds Jonctions: avec plus d'un lien entrant concentre plusieurs chemins

Test logiciel

Graphe de flot de contrôle (GFC)

La structures de contrôle se présente sous la forme d'un graphe dit graphe de flot. On représente les instructions comme cela :

➢ Suite linéaire : 	➢ Case : 
➢ If : 	➢ Until : 
➢ While (for) : 	➢ Else : 

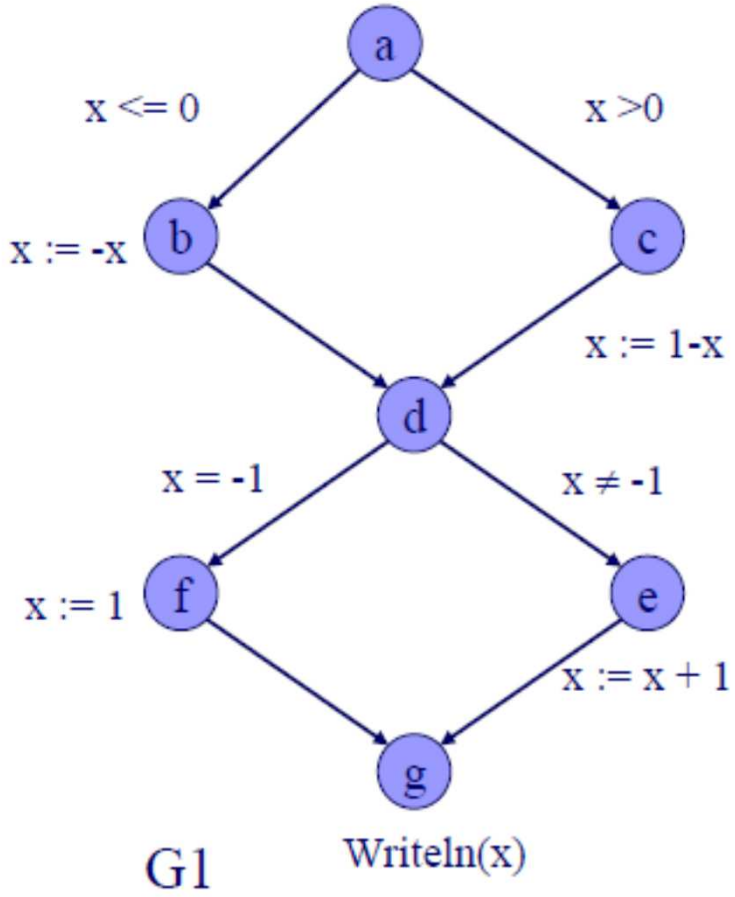
Test logiciel

Exemple

Soit le programme P1 suivant :

```
if x <= 0 then x := -x
else x := 1 - x;
if x = -1 then x=1
else x := x+1;
writeln(x)
```

Ce programme admet le graphe de contrôle suivant :



Test logiciel

Chemin de contrôle

- Le graphe G1 est un graphe de contrôle qui admet une entrée (le nœud a), une sortie (le nœud g).
- Chemin de contrôle : Séquence de nœuds et d'arcs dans le graphe de flot de contrôle, initiée depuis le nœud de départ jusqu'à un nœud terminal
  - le chemin [a, c, d, e, g] est un chemin de contrôle,
  - le chemin [b, d, f, g] n'est pas un chemin de contrôle.
- Le graphe G1 comprend 4 chemins de contrôle :
  - β1 = [a, b, d, f, g] ; β2 = [a, b, d, e, g] ; β3 = [a, c, d, f, g] ; β4 = [a, c, d, e, g]

Test logiciel

Expression des chemins de contrôle

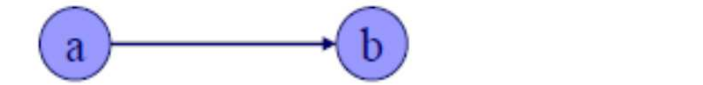
- Le graphe G1 peut-être exprimé sous forme algébrique sous la forme suivante :

  G1 = abdfg+ abdeg+ acdfg+ acdeg (+ désigne le «ou» logique)

- Simplification de l'expression de chemins

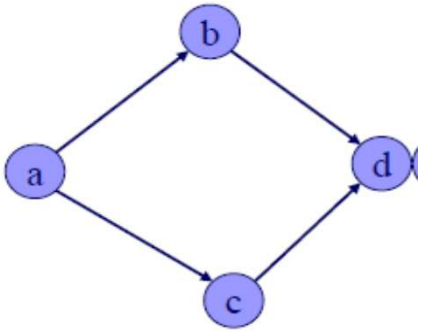
  G1 = a (bdf+ bde+ cdf+ cde) g= a (b + c) d (e + f) g

- On associe une opération d'addition ou de multiplication à toutes les structures primitives apparaissant dans le graphe de flot de contrôle



Forme séquentielle : ab

Forme itérative : ab (cb)* d



Forme alternative :
a (b + c) d

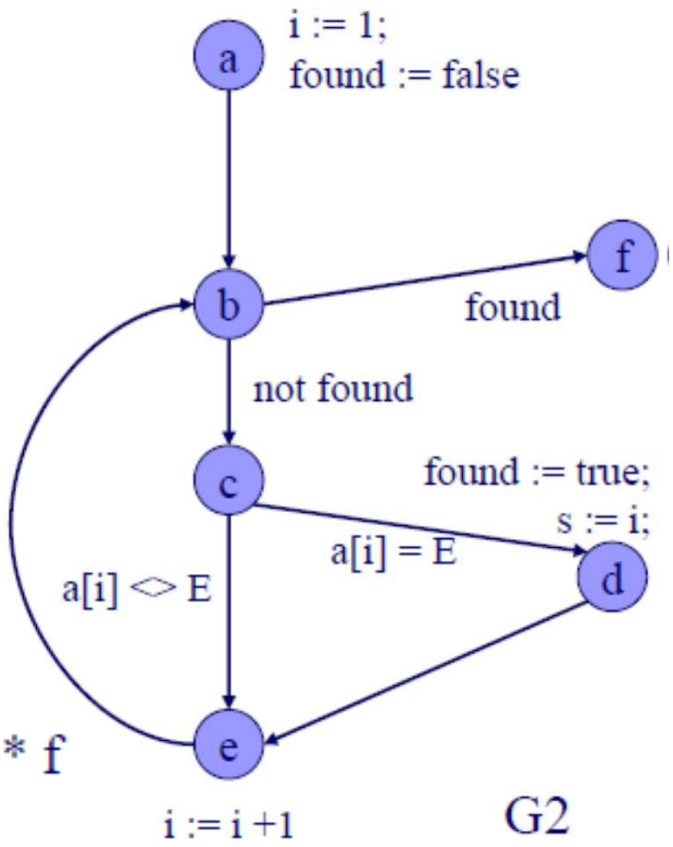
Test logiciel

Exercice 1

- Etablir le graphe de flot de contrôle du programme suivant et donner l'expression simplifiée des chemins de contrôle :

```
i := 1;
found := false;
while (not found) do
      begin
      if (a[i] = E) then
         begin
         found := true;
         s := i;
         end;
      i := i + 1;
      end;
```

G2 = ab [c (1 + d) eb]* f



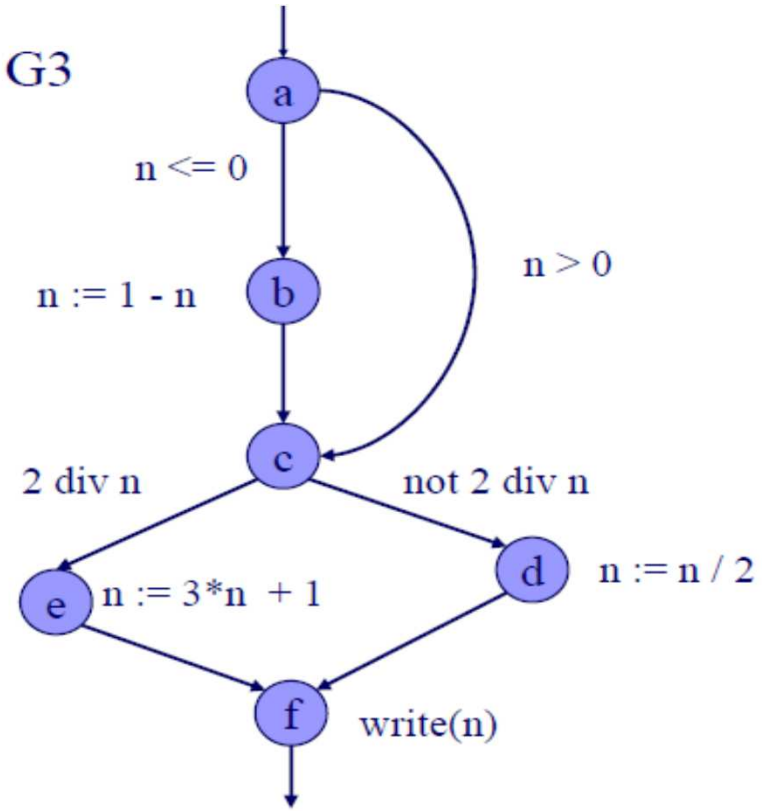
Test logiciel

Exercice 2

▪ Etablir le graphe de flot de contrôle du programme suivant et donner l'expression simplifiée des chemins de contrôle :

```
if n <= 0 then n := 1-n
    end;
if 2 div n
      then n := n / 2
      else n := 3*n + 1
      end ;
write(n);
```

G3 = a (1 + b) c (e + d) f



Test logiciel

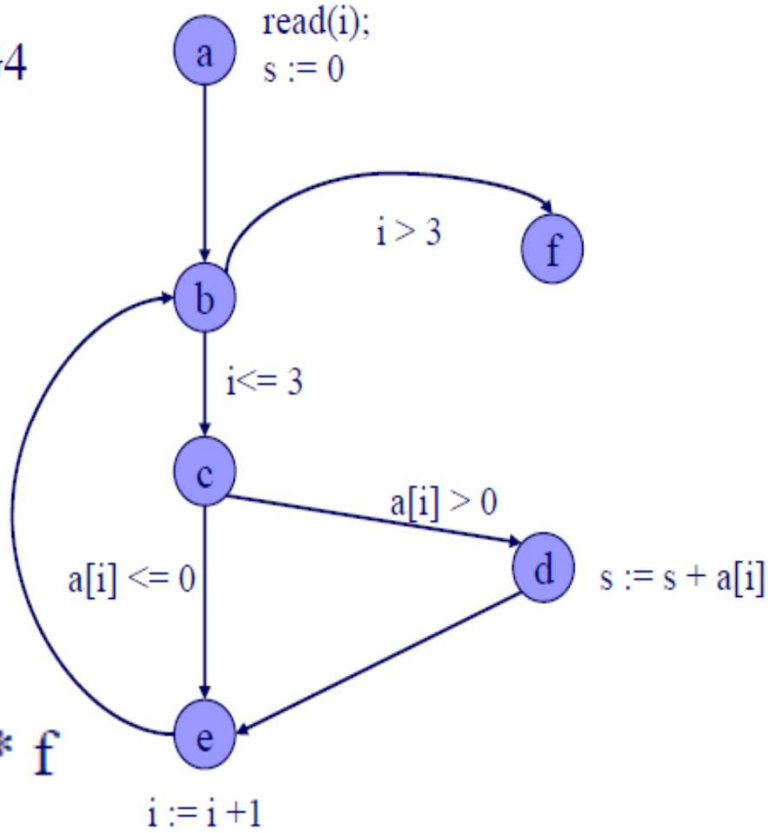
Exercice 3

- Etablir le graphe de flot de contrôle du programme suivant et donner l'expression simplifiée des chemins de contrôle :

```
read(i);
s := 0;
while (i <= 3) do
      begin
      if a[i] > 0 then s := s + a[i];
      i := i + 1;
      end
end;
```

G4 = ab [c (1 + d) eb]* f

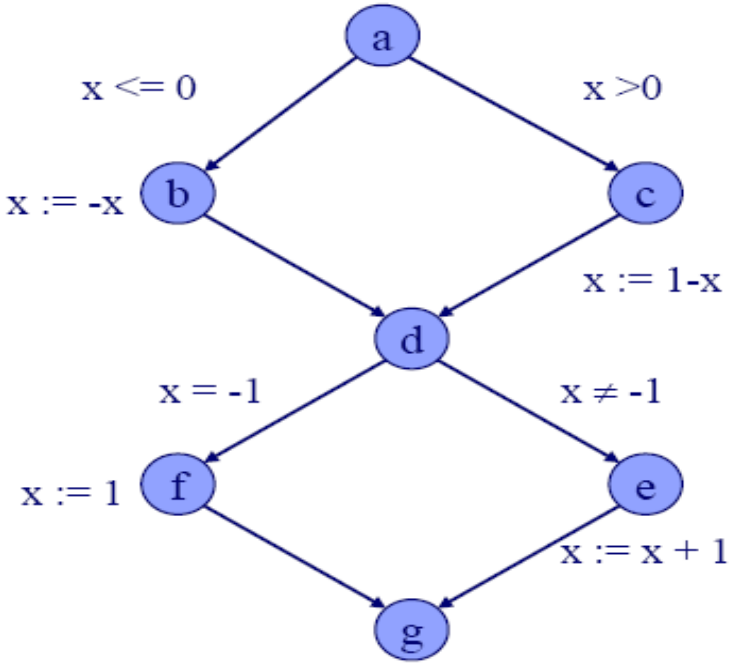
G4



Test logiciel

Problème de chemin non exécutable

- Le chemin ***abdfg*** est un chemin non exécutable
- Le problème de la détection de chemin non exécutable est un problème difficile, indécidable dans le cas général

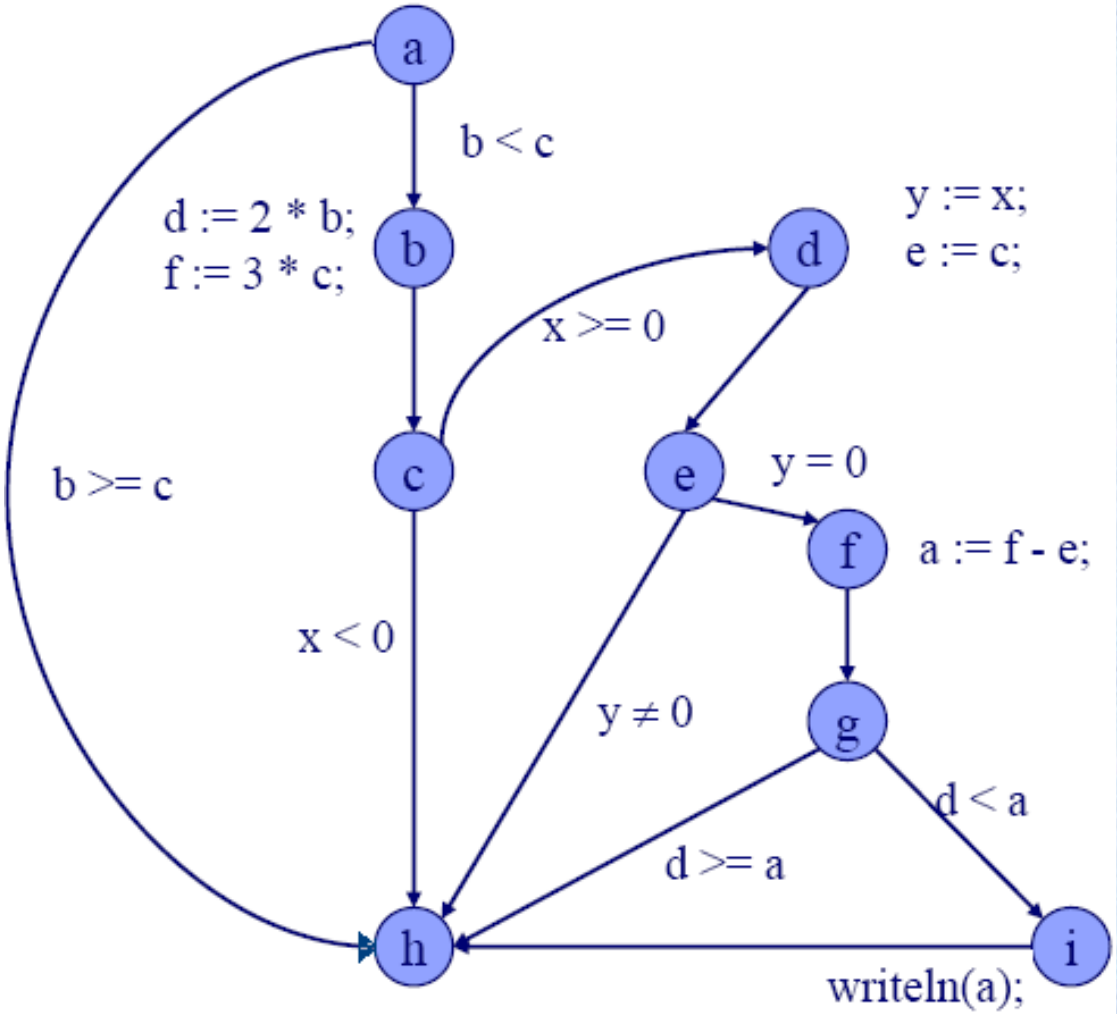


G1

Test logiciel

Exercice

- Trouver une DT permettant d'exécuter le chemin [a,c,d,e,f,g,h]
- Donner l'expression de chemin
- Trouver un chemin non exécutable



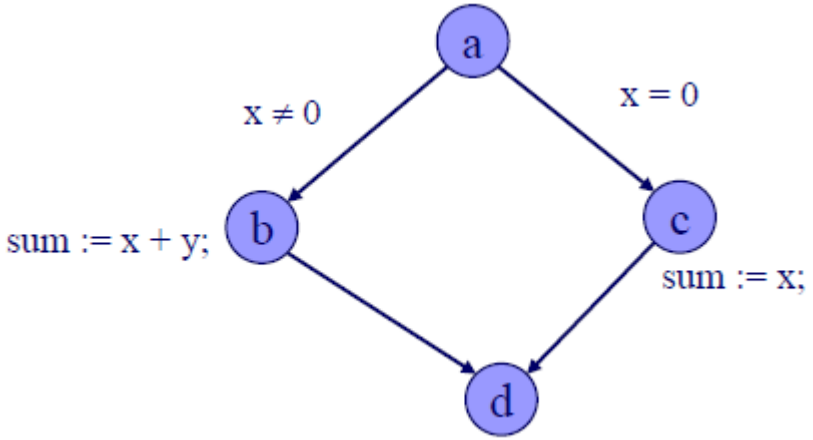
Test logiciel

Critère de couverture des instructions

- Sélectionner un jeu de test T tel que, lorsqu'on exécute la procédure P sur chacun des DT, **chaque instruction** de P est exécutée **au moins une fois**
- Détecter des instructions fautives :Une seule exécution d'une instruction fautive causera une défaillance
- **Taux de couverture :** $\frac{\text{nb de nœuds couverts}}{\text{nb total de nœuds}}$
- **Exemple :**

TC=0,75

Erreur détectée par le chemin ***acd***



Test logiciel

Exemple : Algorithme d'Euclide

```
Begin
        Read(x); Read(y);
        While x <> y Do
                If x > y Then x := x - y;
                Else y := y - x;
                End if;
        End while
        pgcd := x;
End
```

– Trouver un jeu de test qui permet de couvrir toutes les instructions du programme.

Test logiciel

Exemple : Algorithme d'Euclide

```
Begin
        Read(x); Read(y);
        While x <> y Do
                If x > y Then x := x - y;
                Else y := y - x;
                End if;
        End while
        pgcd := x;
End
```

- Pour constituer les jeux de test, on va tenter de grouper dans des classes Ci les éléments du domaine d'entrée C qui activent les mêmes instructions dans P
- C1 = {(x,y) | x > y}; C2 = {(x,y) | x < y}; C3 = {(x,y) | x = y}
- Jeu de test: {(4,2), (2,4), (3,3)}

Test logiciel

Limites du critère de couverture des instructions

```
Begin
        If x < 0 Then x := -x;
        End if;
        z := x;
End;
```

Le jeu de test {-4} permet de couvrir toutes les instructions

– Mais manque de complétude

– Pas de test sur les entiers positifs !!!

Test logiciel

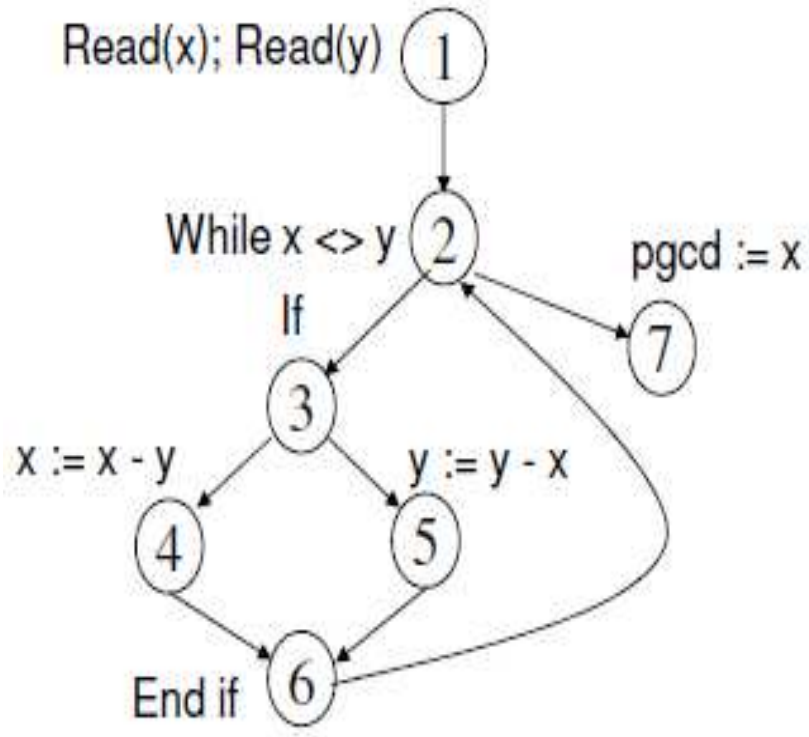
Critère de couverture des arcs du GFC

- Au lieu de s'intéresser aux instructions, on s'intéresse ici aux **branchements** de contrôle conditionnels dans un programme
- **Sélectionner un jeu de test T tel que, lorsqu'on exécute P sur chacun des DT, chaque arc du graphe de flot de contrôle de P est traversé au moins une fois**
- Pour chaque instruction conditionnelle (If, While), tester le cas où la condition est **vraie** et celui où elle est **fausse**
- Critère de sélection plus fort que la couverture des instructions : La couverture de tous les arcs équivaut à la couverture de toutes les **valeurs de vérité** pour chaque nœud de décision.
- Mettre en évidence des **défauts** dans les instructions conditionnelles ou itératives

Test logiciel

Exemple

```
Begin
        Read(x); Read(y);
        While x <> y Do
                If x > y Then
                        x:= x - y;
                Else
                        y := y - x;
                End if;
        End while;
        pgcd := x;
End
```



Test logiciel

Critère de couverture des conditions (et des arcs)

- **Sélectionner un jeu de test T tel que, lorsqu'on exécute P sur chacun des DT, chaque arc du graphe de flot de contrôle de P est traversé au moins une fois et toutes les valeurs possibles des constituantes des conditions complexes sont calculées au moins une fois**
- Dans la couverture des conditions logiques, au lieu de passer une fois dans le cas **true** et une fois dans le cas **false**, on cherche les différentes façons de rendre **la condition logique vraie ou fausse** ; on augmente ainsi la confiance obtenue dans le logiciel (couverture).

Test logiciel

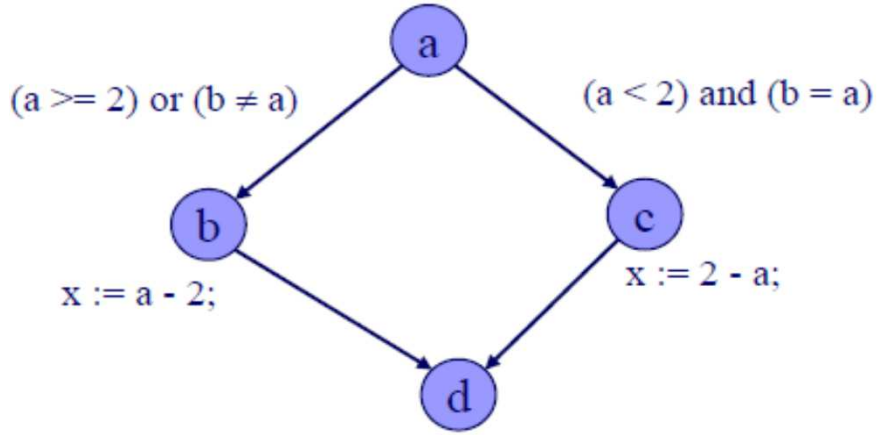
Conditions composées

- Un critère de test de conditions multiples impose que :
  - Chaque condition primitive doit être évaluée à la fois comme vraie et comme fausse
  - Toutes les combinaisons V/F entre primitives d'une condition multiple doivent être testées
- Pour éventuellement repérer les erreurs plus subtiles dans l'évaluation des conditions composées et tenir compte de toutes les valeurs possibles de leurs parties composantes
  - **Décomposer les conditions composées**
  - Condition composée = **Conjonction** ou **disjonction** de deux ou plusieurs clauses élémentaires (i.e., formules atomiques ou leur négation)
  - Ex. x > 4 and y > 8 and z <> 0 (trois clauses élémentaires)

Test logiciel

Exemple

```
if ((a < 2) and (b = a))
then x := 2 - a
else x := a - 2
```



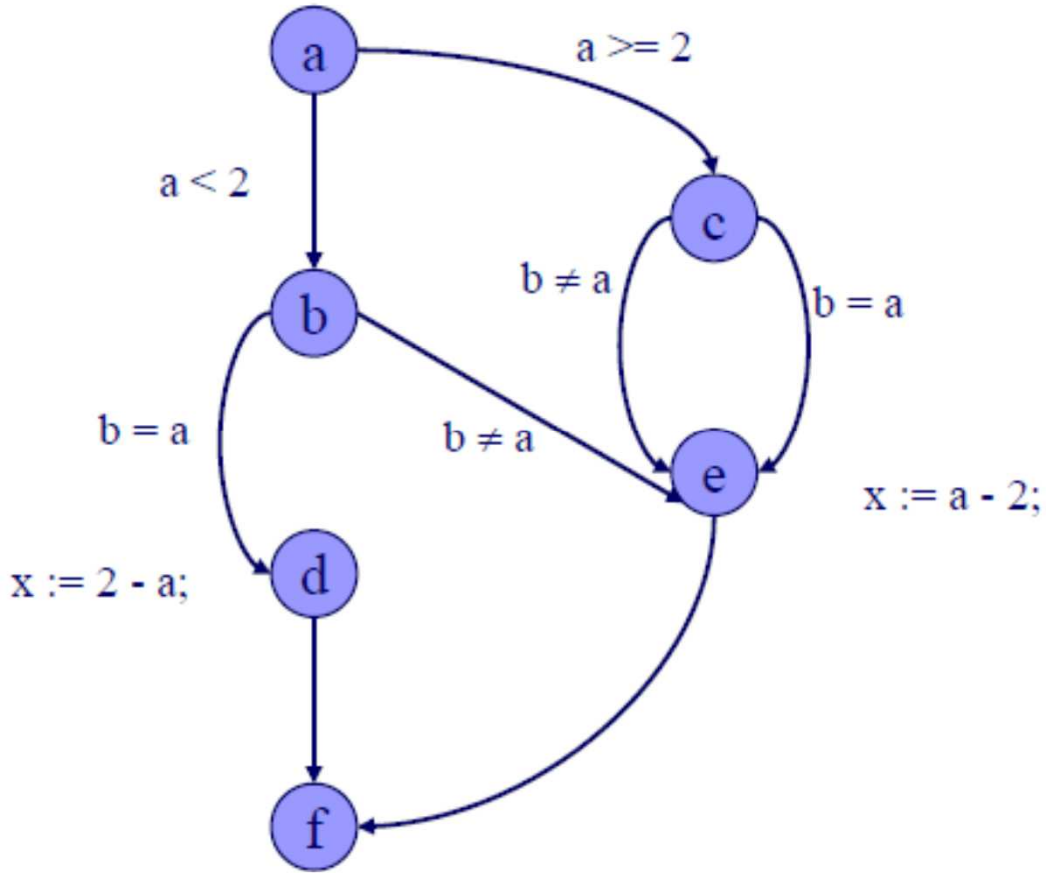
- Le jeu de test **DT1 = {a=b=1}, DT2 = {a=b=3}**

satisfait le critère tous-les-arcs sans couvrir toutes les décisions possibles -ex. **DT3 = {a=3, b=2}**

Test logiciel

Exemple

- Le graphe de flot de contrôle doit décomposer les conditionnelles :



Données de test :

- DT1 = {a=b=1}
- DT2 = {a=1, b=0}
- DT3 = {a=3, b=2}
- DT4 = {a=b=3}

Test logiciel

Exemple

- Pour couvrir les arcs de ce graphe, on devra choisir un jeu de test où toutes les valeurs possibles de c1 et c2 sont expérimentées au moins une fois
- **Attention** : toutes les combinaisons de valeurs de c1 et c2 ne sont pas expérimentées

```
If c1 and c2 Then
    Strue;
Else
    Sfalse;
End if;
```

```
If c1 Then
    If c2 Then
        Strue;
    Else
        Sfalse;
Else
    Sfalse;
End if;
```

Test logiciel

Limites

- Malgré tout, il demeure des erreurs non détectées par les jeux de test satisfaisant le critères des conditions et celui des chemins indépendants

```
If x <> 0 Then y := 5;
Else z := z - x;
Endif;
If z > 1 Then z := z / x; /* division par 0 */
Else z := 0;
Endif;
```

Jeu de test qui couvre tous les arcs et conditions :

{<x=0, z=1>, <x=1, z=3>}

Mais ne détecte pas l'erreur

Test logiciel

Critère de couverture des chemins indépendants

- **Sélectionner un jeu de test T tel que, lorsqu'on exécute P sur chacun des DT, tous les 1-chemins du graphe de flot de P sont parcourus au moins une fois**
- **1-chemin** : chemin parcourant les boucles 0 ou 1 fois
- **Chemin indépendant** : (1-)chemin qui parcourt au moins un nouvel arc par rapport aux autres chemins définis dans une base B (i.e., introduit au moins une nouvelle instruction non parcourue)
- La notion d'indépendance est introduite car le code peut contenir des **boucles**, le nombre de chemins possible est parfois infini

Test logiciel

Trouver une base de chemins linéairement indépendants

- La représentation vectorielle d'un chemin est un vecteur qui compte le nombre d'occurrence de chaque arc

- **Exemple**. Chemin1 = (1, 0, 0, 0, 0, 0, 0, 1)

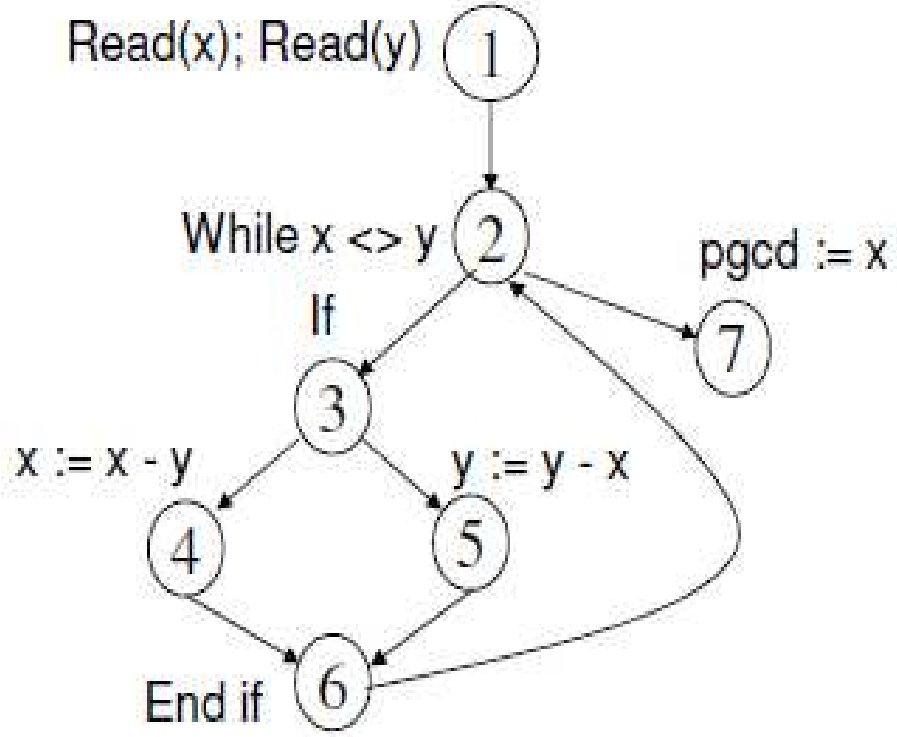
- Un ensemble de chemins est linéairement indépendants si aucun ne peut être représenté par une **combinaison linéaire** des autres (en représentation vectorielle)

- Une base comportant un **nombre de chemins égale au nombre cyclomatique** nous assure de couvrir tous chemins indépendants du graphe de flot. Mais on ne couvre pas nécessairement tous les 1-chemins du graphe

Test logiciel

Exemple : Chemins linéairement indépendants de l'algorithme d' Euclide:

- 1-2-7
- 1-2-3-4-6-2-7 (nouveaux arcs: 2-3, 3-4, 4-6, 6-2)
- 1-2-3-5-6-2-7 (nouveaux arcs: 3-5, 5-6)



Test logiciel

Méthode de sélection des jeux de test

- Construire le **graphe de flot** de contrôle de P
- Déterminer **l'indice de complexité cyclomatique** du graphe de flot
- Définir un ensemble **de base B de chemins indépendants** dans le graphe
- Construire un **jeu de test** qui permettra l'exécution de chaque chemin de l'ensemble B
- Sélection **difficile** dans le cas de gros programmes car
  - Résoudre un système de **contraintes** composé des nœuds prédicats qui se trouvent sur le chemin à parcourir
  - **Attention** : tous les chemins ne sont pas nécessairement satisfiables ! (Problème indécidable…)

Test logiciel

Exemple :Sélection des jeux de test dans l'algorithme d' Euclide (partitionnement)

– Chemin 1-2-7

- D1 = {(x,y) | x = y}
- Cas de test : <x=3, y=3>

– Chemin 1-2-3-4-6-2-7

- D2 = {(x,y)| x > y, x – y = y}
- Cas de test : <x=8, y=4>

– Chemin 1-2-3-5-6-2-7

- D3 = {(x,y)| x < y, x = y - x}
- Cas de test : <x=3, y=6>

Jeu de test : {<x=3, y=3>, <x=8, y=4>, <x=3, y=6>}

Test logiciel

Limites… Exemple

```
found := false; counter := 1;
While (not found) and counter < numberItems Do // erreur <=
        If table(counter) = desiredElem Then
                found := true;
        End if
        counter := counter + 1;
End while;
If found Then Write(« Elément existe »);
Else Write(« Élément n'existe pas »);
Endif;
.
```

Test logiciel

Limites… Exemple

- Soit le jeu de test suivant qui parcourt tous les chemins indépendants du graphe
  - Table vide
  - Table avec 1 élément ne contenant pas celui désiré
  - Table avec 3 éléments dont le premier est désiré

- **Malheureusement, on n'a pas découvert l'erreur !**

Test logiciel

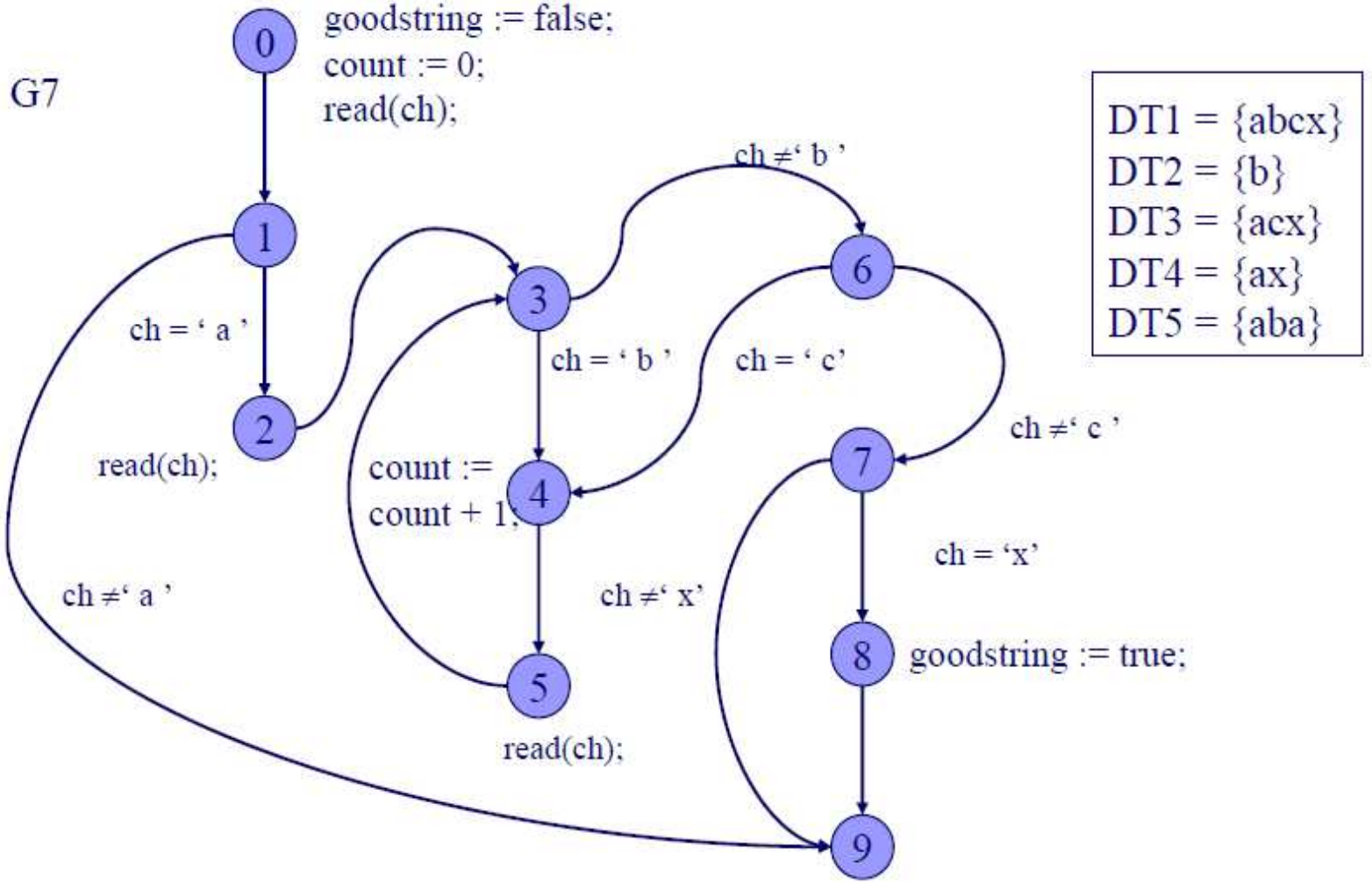
Exercice

- Construire le graphe de flot de contrôle du programme suivant et donner des DT qui couvrent tous les chemins indépendant

```
function goodstring (var count : integer) :
   boolean;
var ch : char;
begin
   goodstring := false;
   count := 0;
   read(ch);
   if ch = ' a ' then
    begin
    read(ch)
   while (ch=' b ') or (ch=' c ') do
     begin
     count := count + 1;
     read(ch);
     end;
   if ch = ' x ' then
           goodstring = true;
   end;
end;
```

Test logiciel

Exercice



Test logiciel

Couverture des i-chemins

Pour sélectionner le bon jeu de test, il faudrait employer un critère de couverture de tous les i-chemins d'exécution possibles (pas seulement les 1-chemins indépendants)

Mais ce critère est difficilement applicable en pratique à cause de la complexité de la sélection des jeux de test correspondants

Test logiciel

Cas des boucles simples (bornées par n itérations autorisées)

- Trouver un jeu de test permettant de couvrir chacun des cas suivants
  - On saute la boucle
  - Une itération de la boucle
  - Deux itérations
  - m itérations (m < n)
  - n - 1, n et n + 1 itérations

Cas des boucles imbriquées

- Commencer par la boucle la plus intérieure, les indices des autres boucles étant à leur valeur minimale
- Appliquer les tests de boucle simple à la boucle la plus interne
- Passer à la suivante, les boucles externes maintenues à leur valeur minimale, les internes à valeur « typique »
- Continuer vers la boucle la plus externe

Test logiciel

Boucles concaténées

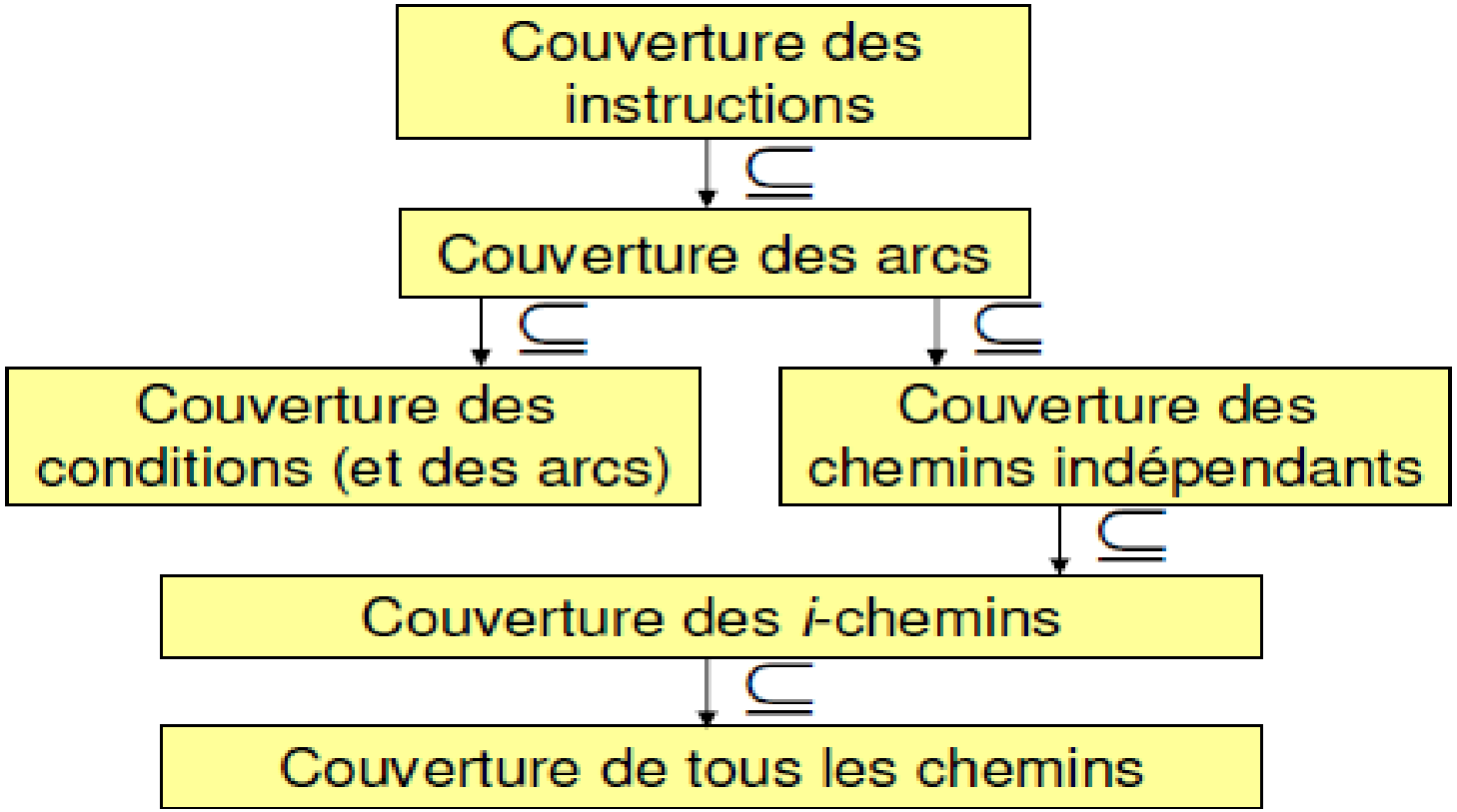
- Si réellement indépendantes : traiter chacune comme une boucle simple
- Si dépendantes (index de la 1ère boucle = valeur initiale de l'index de la 2e boucle) : approche des boucles imbriquées

Boucles non structurées

- Si possible, recoder ou refaire la conception

Test logiciel

- **Degré de couverture des jeux de test générés par les différents critères**



Test logiciel

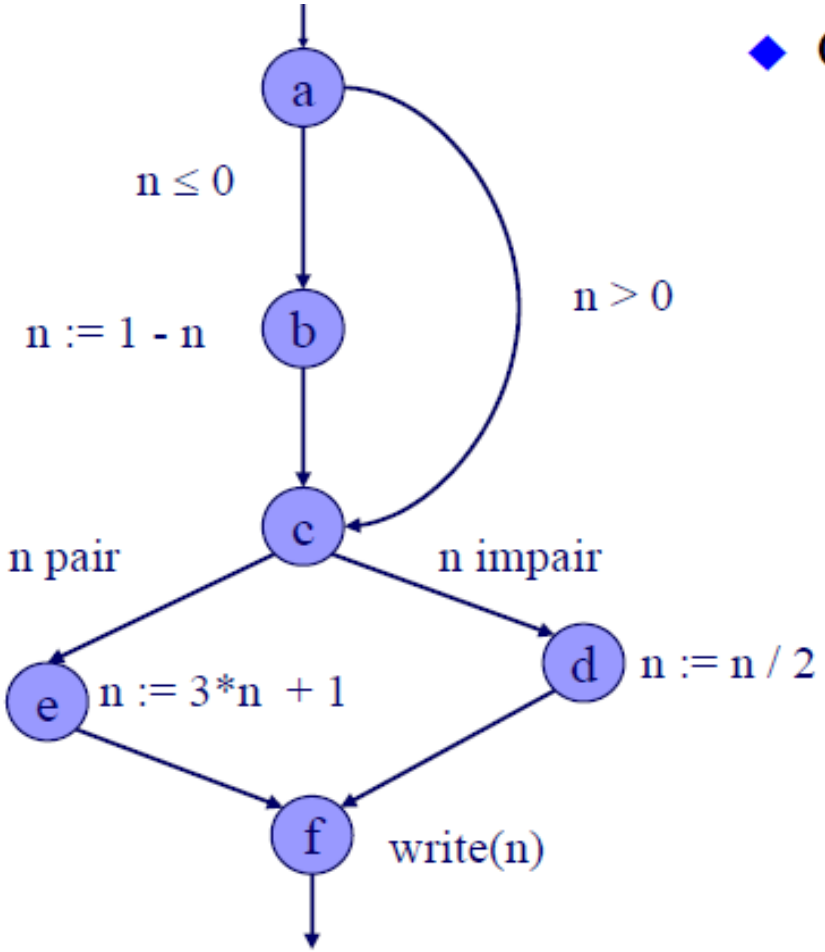
Exercice1

- Construire le GFC du programme suivant
- Calculer des DT suivant les critères de recouverement de tous-les-nœud, tous-les-arcs et tous-les-chemins-indépendant

```
If n ≤ 0 then n := 1-n
     end;
If n pair
              then n := n / 2
              else n := 3*n + 1
              end ;
Write(n);
```

Test logiciel

Exercice1



- Critères de test :
  - *tous-les-nœuds*
    - » *n = 0, n = -1*
  - *tous-les-arcs*
    - » *n = 2, n = -2*
  - *tous-les-chemins-indépendants*
    - » n = -1, n = -2, n = 1, n = 2

Test logiciel

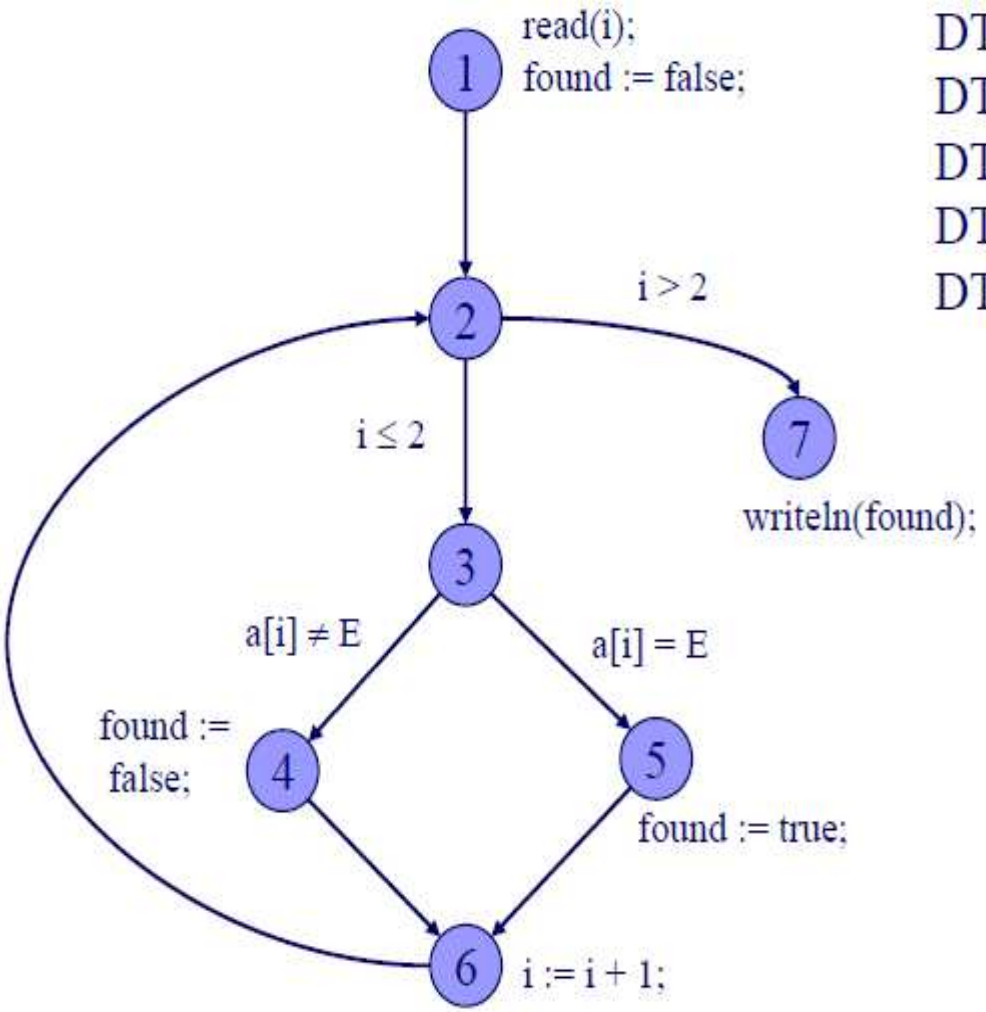
Exercice 2

- Construire le GFC du programme suivant
- Calculer des DT suivant les critères de recouvrement de tous-les-nœud, tous-les-arcs et tous-les-chemins-indépendant

```
program p (input, output) ;
var a : array[1..2] of integer;
E, i : integer;
begin
    read(i, E, a[1], a[2]);
    found := false;
while i <= 2 do
    begin
    if (a[i] = E) then found := true;
    else found := false;
    i := i + 1;
    end;
writeln(found);
end;
```

Test logiciel

Exercice 2



DT1 = {i=3}
DT2 = {i=1, E=10, a[1]=20, a[2]=10}
DT3 = {i=1, E=10, a[1]=10, a[2]=20}
DT4 = {i=1, E=10, a[1]=10, a[2]=10}
DT5 = {i=1, E=10, a[1]=20, a[2]=30}

Test logiciel

Analyse des flots de données

- Les critères basés sur le flot de données sélectionnent les données de test en fonction des définitions et des utilisations des variables du programme
- Tests basés sur les flots de donnés
  - Où les données sont **définies**
  - Où les données sont **modifiées**
  - **Séquence d'événements** liés aux données

Test logiciel

Usage de variable :

- Une variable est définie lors d'une instruction si la valeur de la variable est modifiée (affectations).
  - ***d(v,n) :*** l'assignation d'une valeur à v au noeud n
- Une variable est dite référencée si la valeur de la variable est utilisée.
  - ***c-use(v,n)*** : La variable v est utilisée dans un traitement au noeud n (ex: gauche d'une assignation, argument d'appel de procédure, instruction de sortie)
  - ***p-use(v,n,m) :*** La variable v utilisée dans un prédicat entre les nœuds m et n (ex: dans une expression utilisée pour une décision)
- Dés-allocation de variable
  - **kill k(v,n) :** La variable v est détruite ou libérée au nœud n

Test logiciel

Usage de variable :

- Chaque ***champs d'un record r*** est traité comme une variable individuelle
  - **Définition de r** est interprétée comme la définition de chacun des champs de r,
  - **Référence à r** est interprétée comme l'usage de chacun des champs de r
- Les ***tableaux*** sont considérés globalement en général
  - **La définition de a[e]** est interprétée comme un c-use des variables de l'expression e suivi d'une définition de a
  - **La référence à a[e]** est interprétée comme un c-use des variables de e suivi de l'usage de a

Test logiciel

Notion d'instruction utilisatrice

- Une instruction J2 est utilisatrice d'une variable x par rapport à une instruction J1, si la variable x qui a été définie en J1 est peut être directement référencée dans J2, c'est-à-dire sans redéfinition de x entre J1 et J2
- **Exemple**

```
(1)     x := 7;
(2)     a := x+2;
(3)     b := a*a;
(4)     a := a+1;
(5)     y := x + a;
```

Considérons l'instruction (5) :

(5) est c-use de (1) pour la variable x

(5) est c-use de (4) pour la variable a

Test logiciel

Exercice

- Trouver les définitions et les utilisations des variables du programme :

```
   int main (void)
   {
1    char *line;
2    int x = 0, y;
3    line = malloc(256 * sizeof (*line));
4    fgets (line, 256, stdin);
5    scanf ("%d", &y);
6    if (y > x)
7       y = y - x;
8    else {
9       x = getvalue();
10      y = y - x;
11   }
12 printf ("%s%d", line,y);
13 free(line);
   }
```

Test logiciel

Exercice

```
 int main (void)
 {
1   char *line;
2   int x = 0, y;
3   line = malloc(256 * sizeof (*line));
4   fgets (line, 256, stdin);
5   scanf ("%d", &y);
6   if (y > x)
7      y = y - x;
8   else {
9      x = getvalue();
10     y = y - x;
11  }
12  printf ("%s%d", line,y);
13  free(line);
}
```

d(x,2)
d(line,3)
d(line,4)
d(y,5)
u(y,6) p-use *u(x,6) p-use*
d(y,7) *u(x,7) c-use* *u(y,7) c-use*
d(x,9)
d(y,10) *u(x,10) c-use* *u(y,10) c-use*
u(y,12) c-use
k(line,13)

Test logiciel

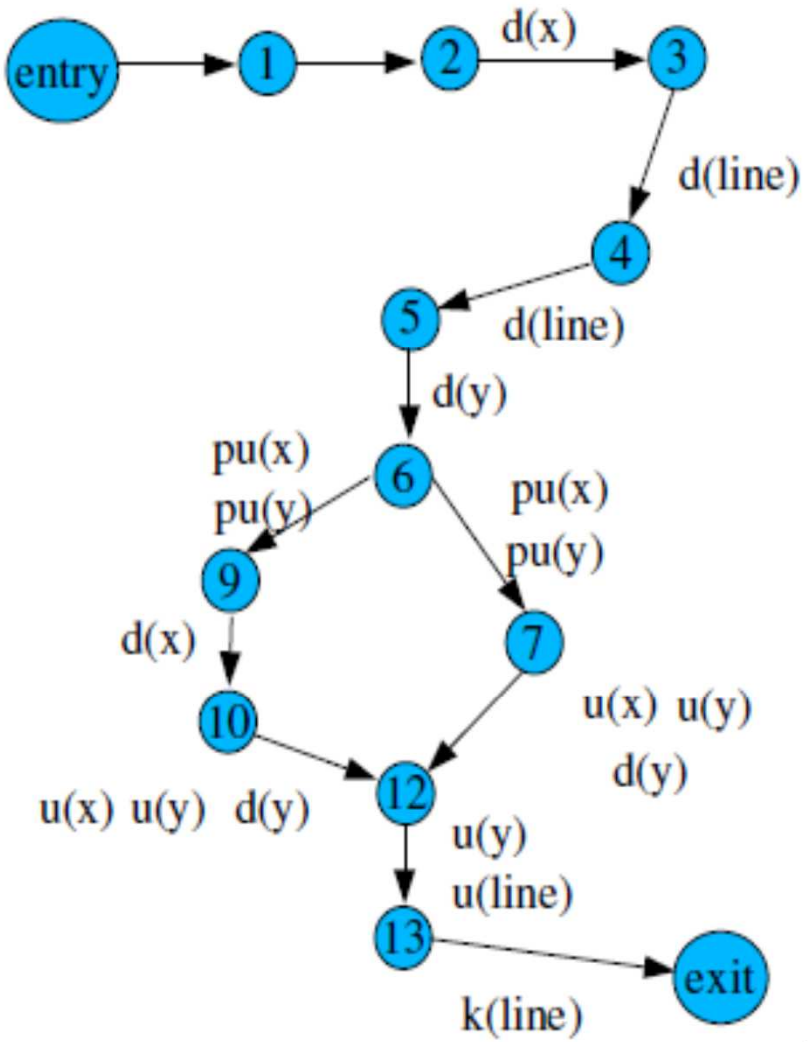
Chemins de flots de données

- **Chemin dr-strict de v :** c'est un chemin d'utilisation reliant l'instruction de définition de la variable à une instruction utilisatrice (sans redéfinition)
- **Chemin complet** : noeud initial est le noeud de début, noeud final est le noeud de sortie
- **Chemin simple :** tous les nœuds sauf possiblement 1er et dernier distincts
- **Chemin sans boucle :** tous les nœuds distincts
- **Pairedu pour variable v (d,u) :**
  - d noeud ou v est **définie**
  - u noeud ou v est **utilisée**
  - Le chemin entre v et u est un chemin **dr-strict**
- **Chemin définition-usage (dupath) pour v** noté <n1,...,nj,nk> est soit
  - **C-use** de v au noeud nk, et un **chemin dr-strict simple** ou
  - **P-use** de v sur le lien nj> nk, et <n1,n2,...nj> chemin **dr-strict sans boucle**

Test logiciel

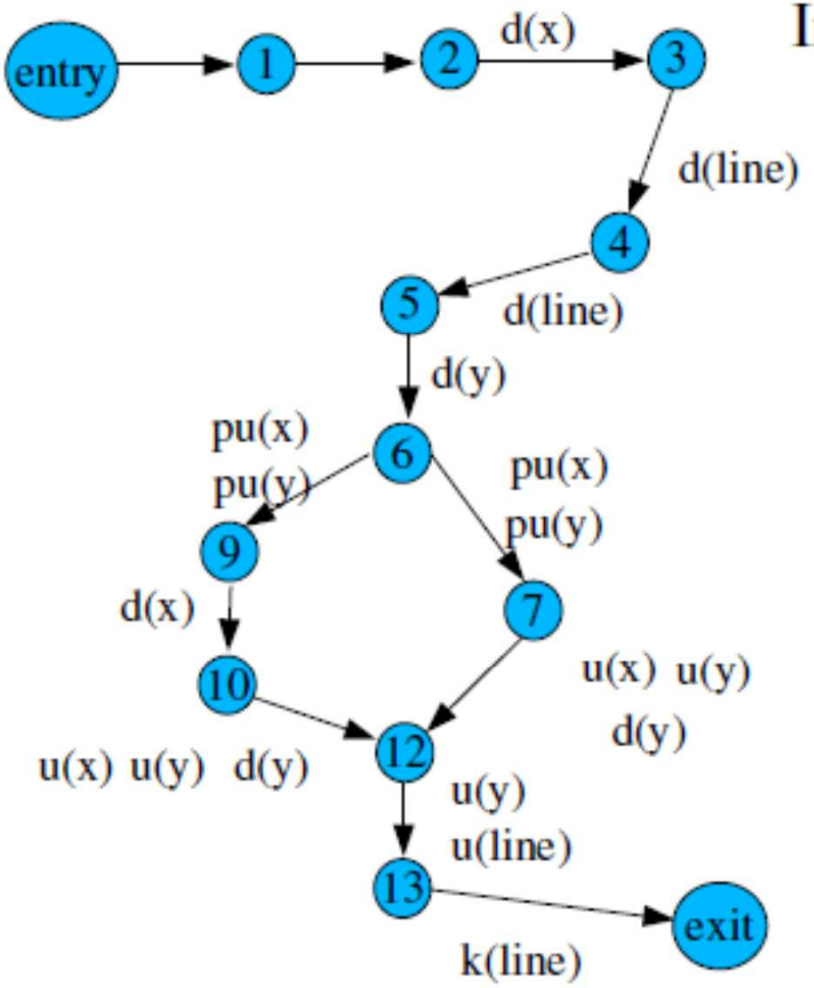
- **Exemple (organigramme annoté)**

```
 int main (void)
 {
1   char *line;
2   int x = 0, y;
3   line = malloc(256 * sizeof (*line));
4   fgets (line, 256, stdin);
5   scanf ("%d", &y);
6   if (y > x)
7      y = y - x;
8   else {
9      x = getvalue();
10     y = y - x;
11  }
12 printf ("%s%d", line,y);
13 free(line);
}
```



Test logiciel

- **Exemple : Information définition \ utilisation**



Information de Définition/Usage

Variable	Define	Use	Comments
x	2		
		6-7, 6-9	p-use
		7	c-use
	9		READ
		10	c-use
line	3		mem-alloc
	4		READ
		12	c-use
y	5		
		6 -7, 6-9	p-use
		7	c-use
	7		
		10	c-use
	10		
		12	c-use

Test logiciel

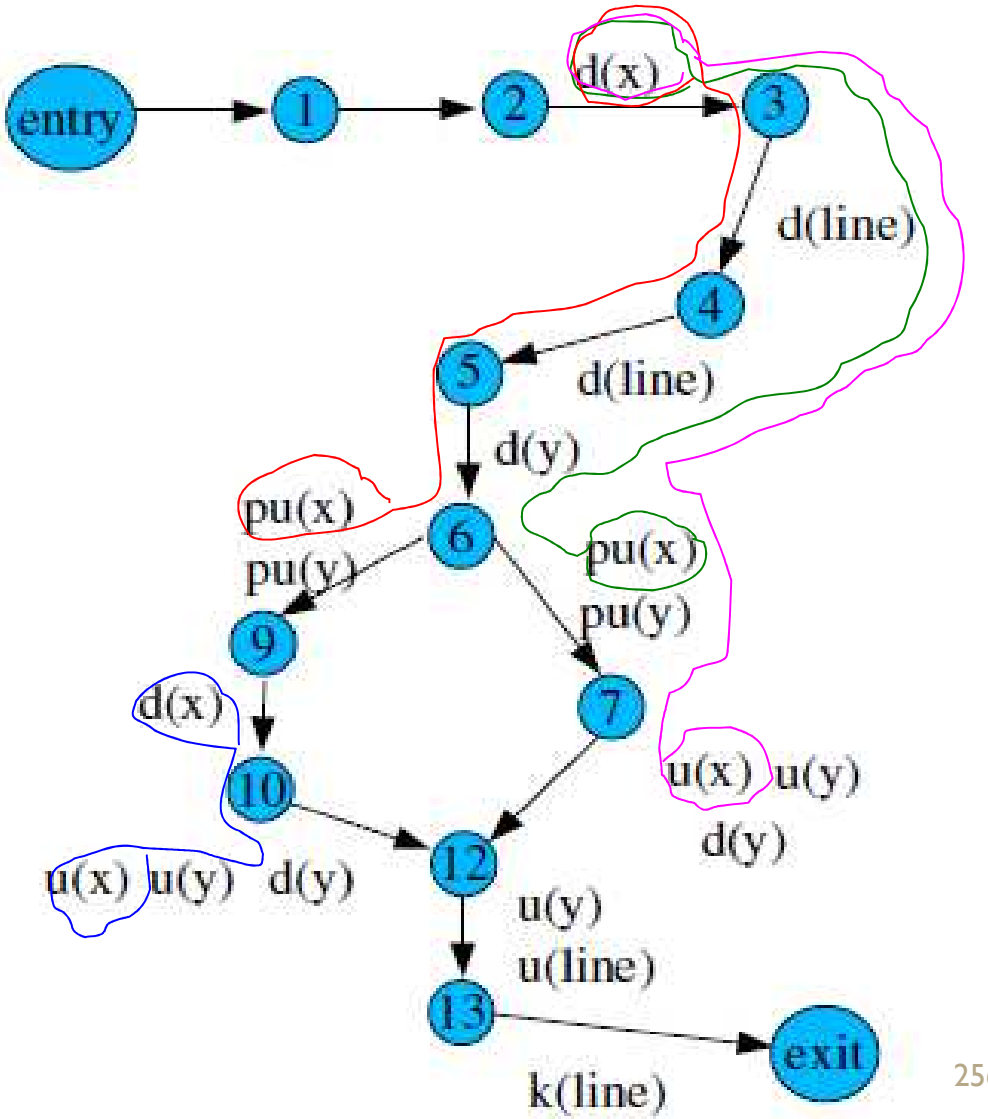
- **Exemple : chemin du-path de *x***

2 – 3 – 4 – 5 – 6 - 7

2 – 3 – 4 – 5 – 6 - 9

2 – 3 – 4 – 5 – 6 - 7

9 - 10

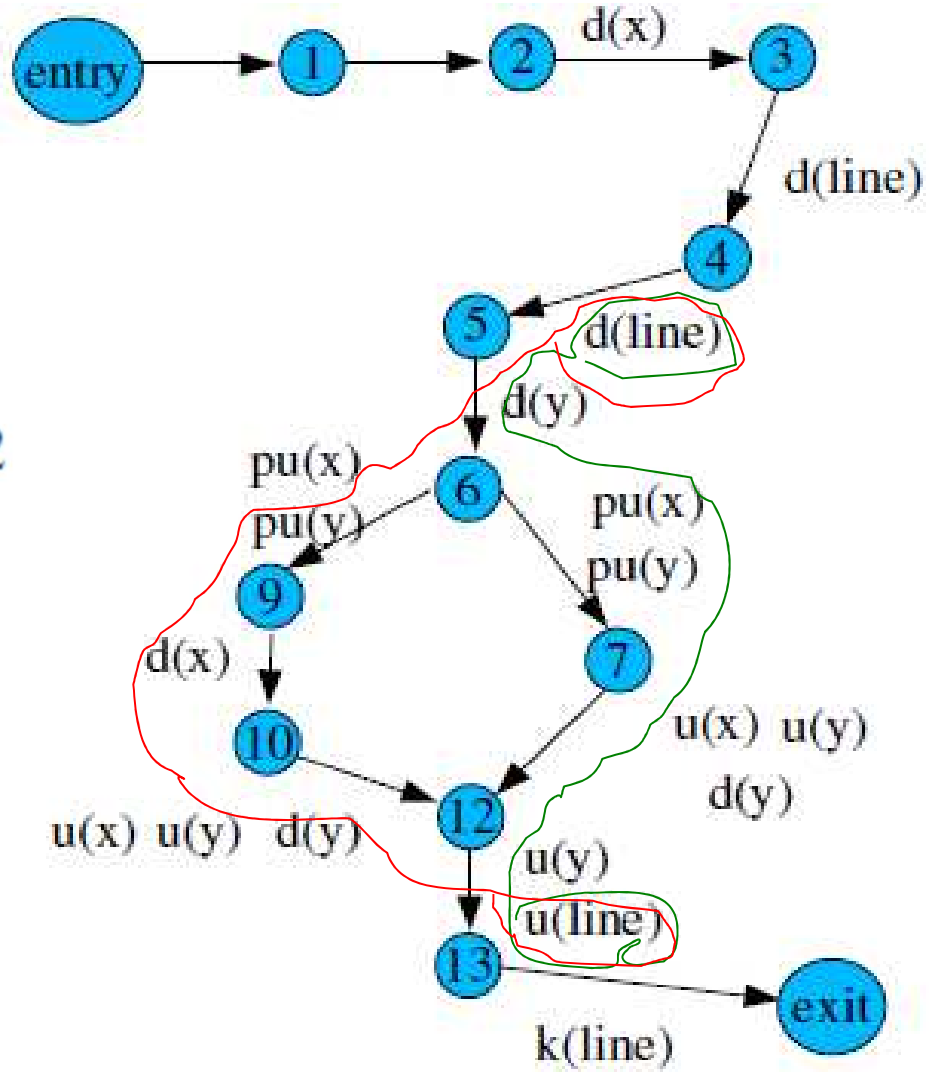


Test logiciel

- **Exemple : chemin du-path de *line***

4 – 5 – 6 – 7 - 12

4 – 5 – 6 – 9 - 10 - 12



Test logiciel

- **Exemple : chemin du-path de *y***

5 – 6 - 7

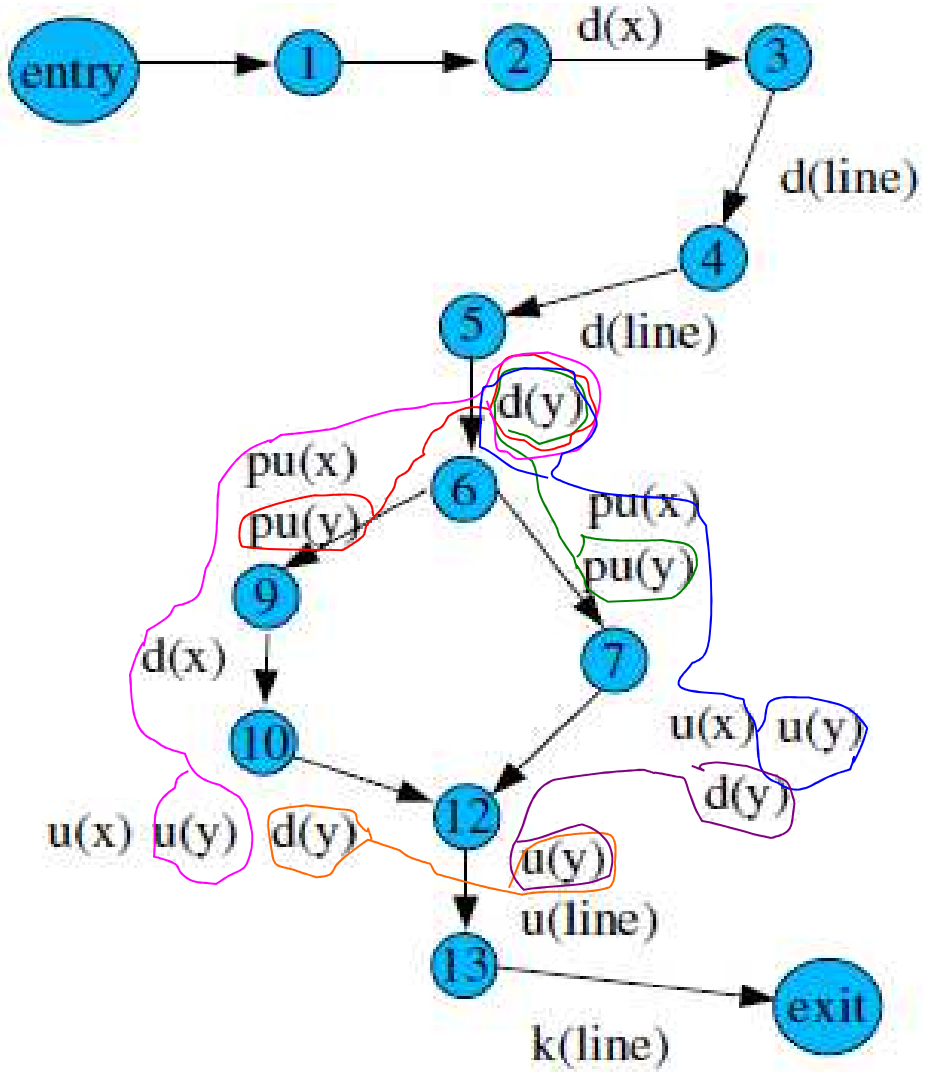
5 – 6 - 9

5 – 6 – 9 - 10

5 – 6 – 7

10 – 12

7 – 12



Test logiciel

Stratégie de test de flot de données

Il existe trois type de couverture du graphe de flots de donnée

- Couverture de toutes-les-définitions (All-defs)
- Couverture de toutes-les-utilisations (All-Uses)
- Couverture de quelques-utilisation(mélange de c-use et p-use)
- Couverture de tous-du-paths (All-paths)

Test logiciel

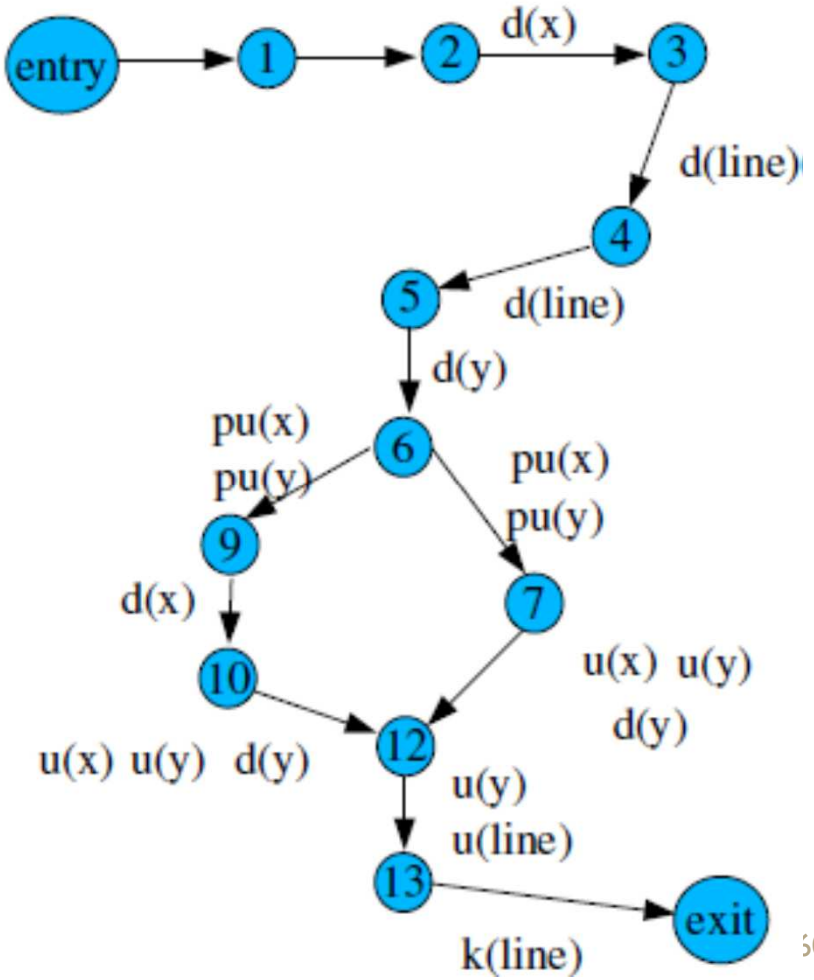
Couverture All-defs

Au moins un chemin dr-strict de chaque définition variable à au moins 1 usage de celle ci

Exemple:

Tests couvrant

- 2 – 3 – 4 – 5 – 6 – 7 (x)
- 9 – 10 (x)
- 4 – 5 – 6 – 7 – 12 (line)
- 5 – 6 (y)
- 7 – 12 (y)
- 10 – 12 (y)



Test logiciel

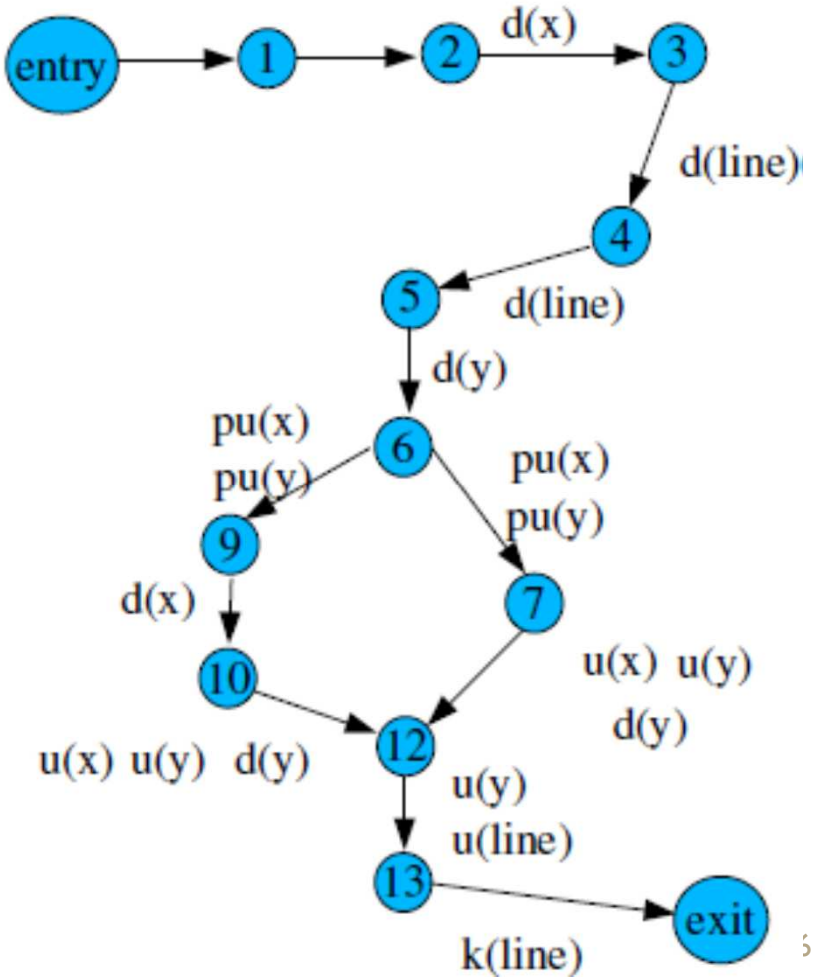
Couverture All-Uses

au moins un chemin dr-strict de chaque définition de variable à tous les usages de celle ci

Exemple:

test couvrant les chemins

- 2 – 3 – 4 – 5 – 6 – 7 (x)
- 2 – 3 – 4 – 5 – 6 – 9 (x)
- 9 – 10 (x)
- 4 – 5 – 6 – 7 – 12 (line)
- 4 – 5 – 6 – 9 – 10 12 (line)
- 5 – 6 7(y)
- 5 – 6 – 9 (y)
- 5 – 6 – 9 – 10 (y)
- 7 – 12 (y)
- 10 – 12 (y)



Test logiciel

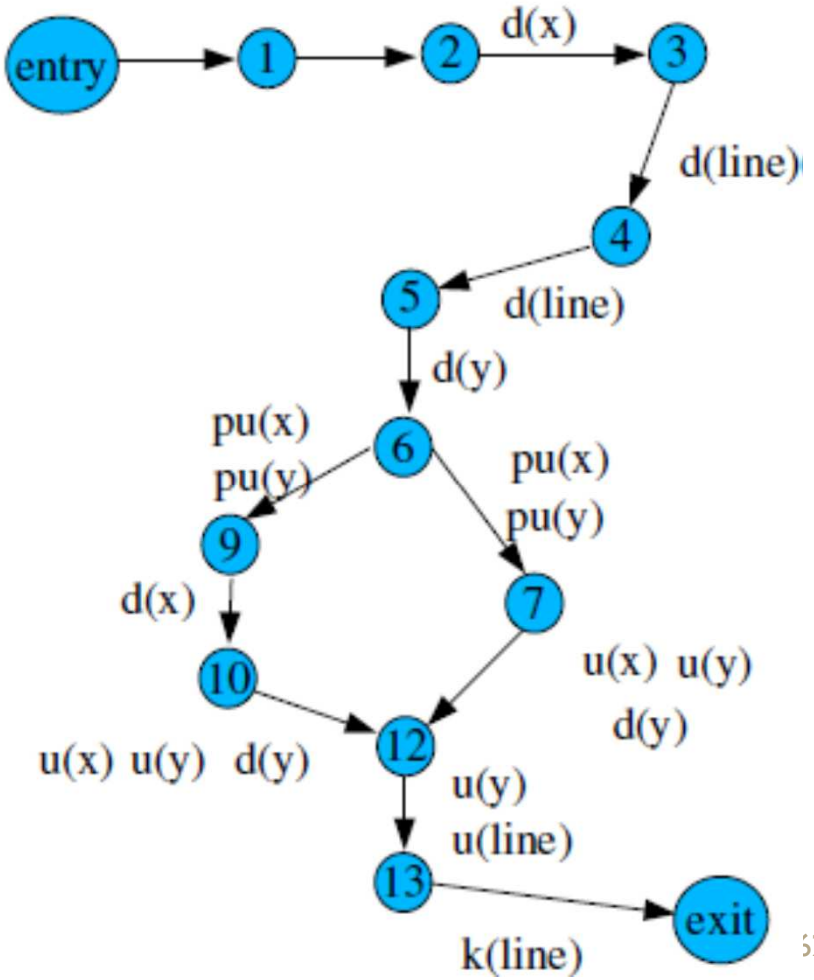
Couverture Tous-P-Uses/Quelques-C-Uses:

Au moins 1 chemin dr-strict de chaque définition de v à tous p-uses de v, si aucune à 1 c-use

Exemple:

Test couvrant les chemins

- 2 – 3 – 4 – 5 – 6– 7 (x)
- 2 – 3 – 4 – 5 – 6– 9 (x)
- 9 – 10 (x)
- 4 – 5 – 6 – 7 – 12 (line)
- 5 – 6 – 7 (y)
- 5 – 6 – 9 (y)
- 7 – 12 (y)
- 10 – 12 (y)



Test logiciel

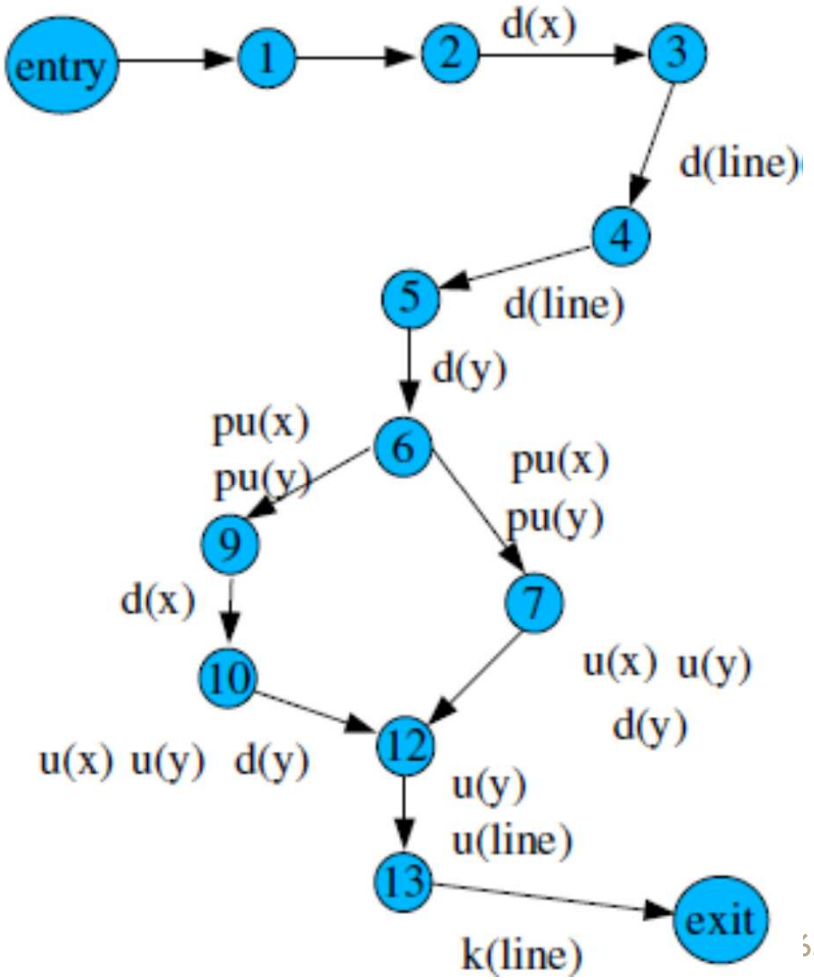
Couverture Tous-C-Uses/Quelques-P-Uses:

Au moins 1 chemin dr-strict de chaque définition de v à tous c-uses de v, si aucune à 1 p-use

Exemple:

Test couvrant les chemins

- 2 – 3 – 4 – 5 – 6– 7 (x)
- 9 – 10 (x)
- 4 – 5 – 6 – 7 – 12 (line)
- 4 – 5 – 6 – 9 – 10 – 12 (line)
- 5 – 6 – 7 (y)
- 5 – 6 – 9 – 10 (y)
- 7 – 12 (y)
- 10 – 12 (y)



Test logiciel

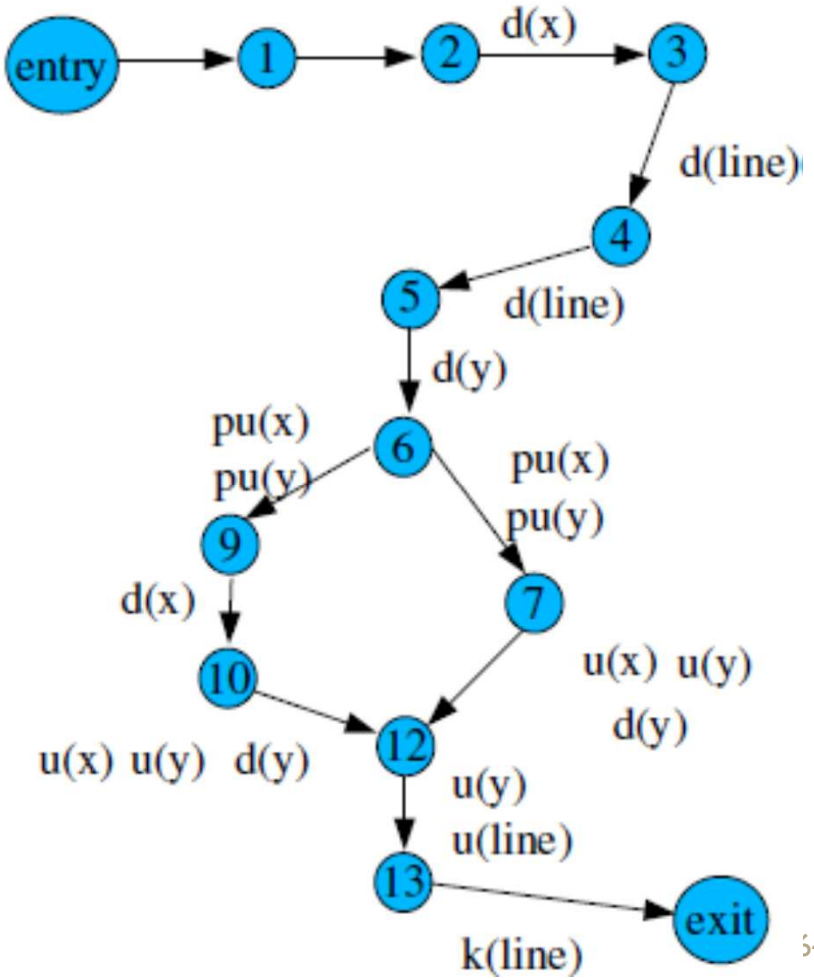
Couverture All-Paths:

Couverture de tous les chemins du-paths

Exemple:

Test couvrant les chemins

- 2 – 3 – 4 – 5 – 6– 7 (x)
- 2 – 3 – 4 – 5 – 6– 9 (x)
- 9 – 10 (x)
- 4 – 5 – 6 – 7 – 12 (line)
- 4 – 5 – 6 – 9 – 10 – 12 (line)
- 5 – 6 – 7 (y)
- 5 – 6 – 9 (y)
- 5 – 6 – 9 – 10 (y)
- 7 – 12 (y)
- 10 – 12 (y)



Test logiciel

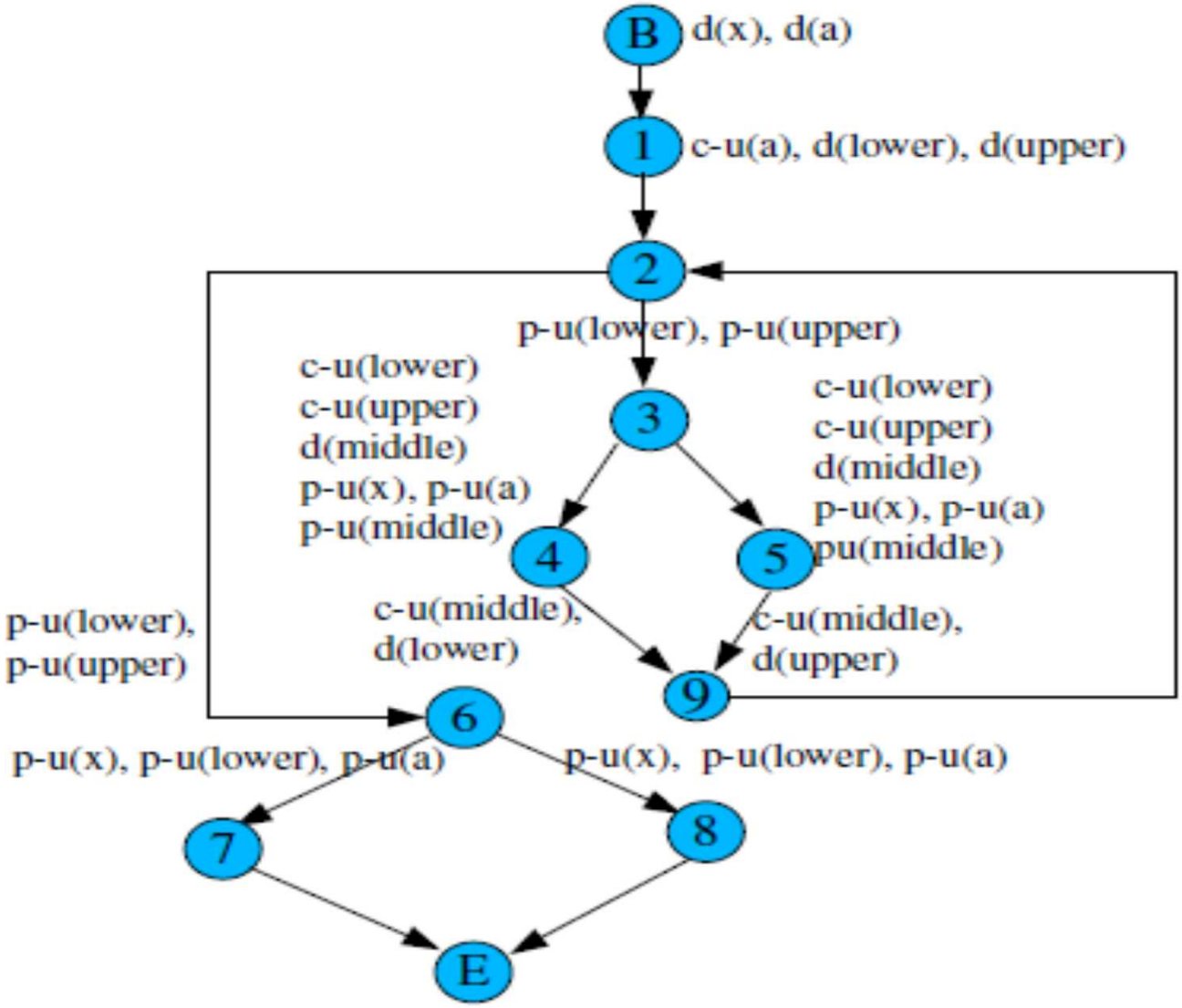
Exercice:

- Donner l'organigramme annoté du programme suivant avec les informations de définition / utilisation
- Donner les chemins du-path de chaque variable
- Trouver une couverture all-paths

```
boolean search(int x, int a[]) {
    int lower, upper, middle;
1   lower = 1; upper = a.length;
2   while (lower < upper) {
3       middle = (lower + upper)/2;
        if (x > a[middle])
4           lower = middle + 1;
5       else upper = middle;
    }
6 if (a[lower] == x)
7       return true;
8   else return false;
}
```

Test logiciel

- **Exercice:** ***Organigramme annoté***



Test logiciel

Exercice:

Informations de définition / utilisation

Variable	Define	Use	Comments
x	B	3-4, 3-5 6 -7, 6-8	p-use p-use
a	B	1 3-4, 3-5 6-7, 6-8	c-use p-use p-use
lower	1 4	2-3, 2-6 3 6 -7, 6-8	p-use c-use p-use
upper	1 5	2-3, 2-6 3	p-use p-use
middle	3	3-4, 3-5 4 5	p-use c-use c-use

Test logiciel

Exercice:

Chemins du-paths

- x
  - B-1-2-3-4
  - B-1-2-3-5
  - B-1-2-6-7
  - B-1-2-6-8
- a
  - B-1
  - B-1-2-6-7
  - B-1-2-6-8
  - B-1-2-3-4
  - B-1-2-3-5
- upper
  - 1-2-3
  - 1-2-6
  - 5-9-2-3
  - 5-9-2-6
- middle
  - 3-4
  - 3-5
- lower
  - 1-2-3
  - 1-2-6
  - 1-2-6-7
  - 1-2-6-8
  - 4-9-2-3
  - 4-9-2-6

Couverture All-paths

- B-1-2-3-4-9-2-3-5-9-2-3-5-9-2-6-7-E
- B-1-2-3-5-9-2-3-4-9-2-6-8-E
- B-1-2-6-7-E
- B-1-2-6-8-E

Test logiciel

Test unitaire

But :

- Validation de la conformité de chaque composant logiciel pris unitairement par rapport à sa spécification détaillée.
- Vérification du respect des spécifications fonctionnelles et techniques

Quand ?: Dès qu'une pièce de code a été codée et compilée correctement

Comment ?: sur machine hôte, généralement sans banc de tests en utilisant le test structurel

Qui ?: Pour les logiciels de faible criticité, elle peut être réalisée par l'équipe de développement (mais pas par le développeur ayant codé la pièce de code).

Test logiciel

Données de test

- Données fictives
- Possibilité d'importer des données de production, après traitement de désensibilisation.
- L'accès à des données par rapport systèmes externes peut nécessiter de simuler l'accès ou la création de ces données
- Réutiliser des anciens jeux d'essais

Ressources

- Documents de spécification
- Spécification de test :Scénario, jeux d'essais
- Feuilles de résultats
- Précédents tests

Test logiciel

Démarche du test unitaire

Analyse statique

- Examen statique du code : relecture, inspection, qualimétrie, mesures de complexité, ..)
- Nombre cyclomatique, mesure de complexité de Mc Cabe
- Mesure de Halstead
- Gestion des variables : non initialisation, non utilisation
- Taux de commentaires
- Auto documentation

Test logiciel

Démarche du test unitaire

Analyse dynamique (test structurel)

- Couverture si la stratégie est axée sur le flot de contrôle
  - Passage par tous les nœuds
  - Parcours de tous les arcs ou toutes les condition
  - Parcours de tous les chemins ou de i-chemins
- Couverture si la stratégie est axée sur le flot de donnée
  - Passage par toutes les affectations de variables
  - Passage par toutes les utilisations de variables dans les condition
  - Passage par toutes les autres utilisations (calculs, …)
- Entité testée au travers de ses interfaces
- Vérification du service rendu et non de la façon dont il est rendu
- Analyse partitionnelle suivant les classes d'équivalences
- Trouver les DT en fonction des résultats attendus

Test logiciel

Test d'intégration

But : Validation des sous-systèmes logiciels entre eux

- Tests d'Intégration Logiciel/Logiciel (interface entre composants logiciels)
- Tests d'Intégration Logiciel/Matériel (interface entre le logiciel et le matériel)

Quand ? Dès qu'un sous-système fonctionnel (module, objet) est entièrement testé unitairement

Comment ?

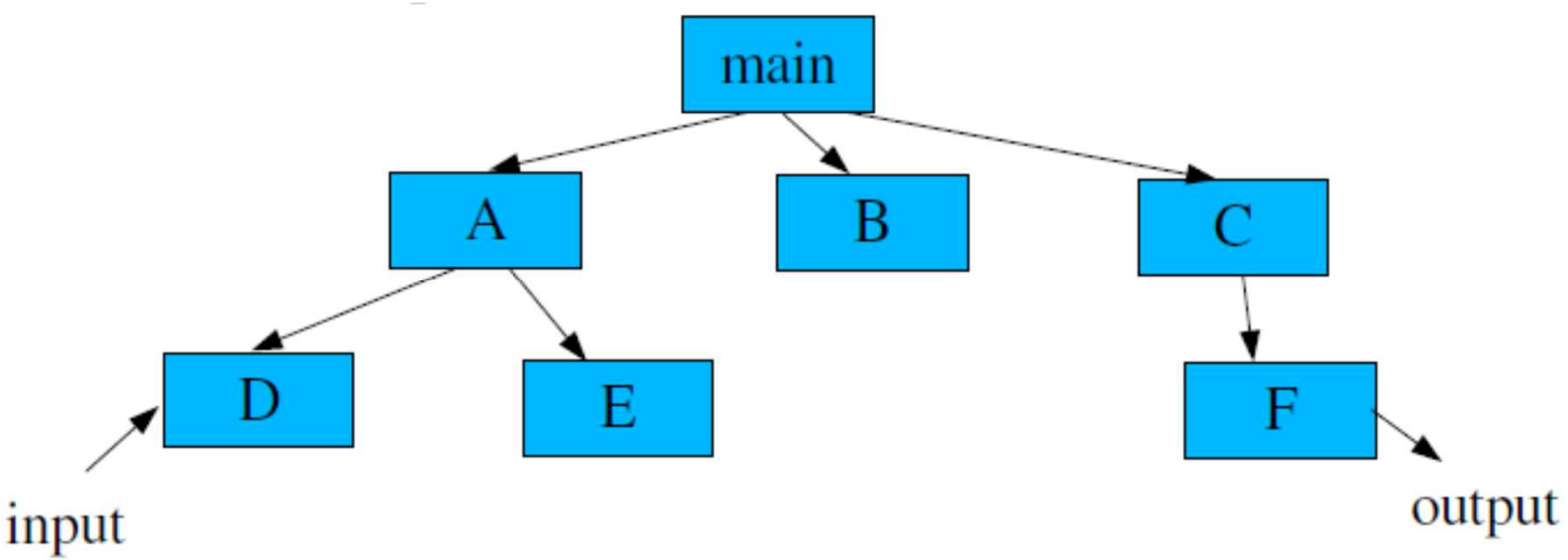
- Logiciel/logiciel généralement sur machine hôte
- Logiciel/matériel sur machine cible avec un banc de test minimal (simulation des entrées, acquisition des sorties).

Qui ? toujours par une équipe de tests indépendante de l'équipe de développement.

Test logiciel

Objectifs du test d'intégration

- Déterminer comment des modules de bas niveau, et fonctionnant correctement, sont assemblés pour former des modules de plus haut niveau tout en gardant le bon fonctionnement
  - Valider les interfaces des composants et les interactions matérielles
  - Nécessiter une représentation de *structure d'invocation de modules*



Test logiciel

Périmètre couvert par le test d'intégration

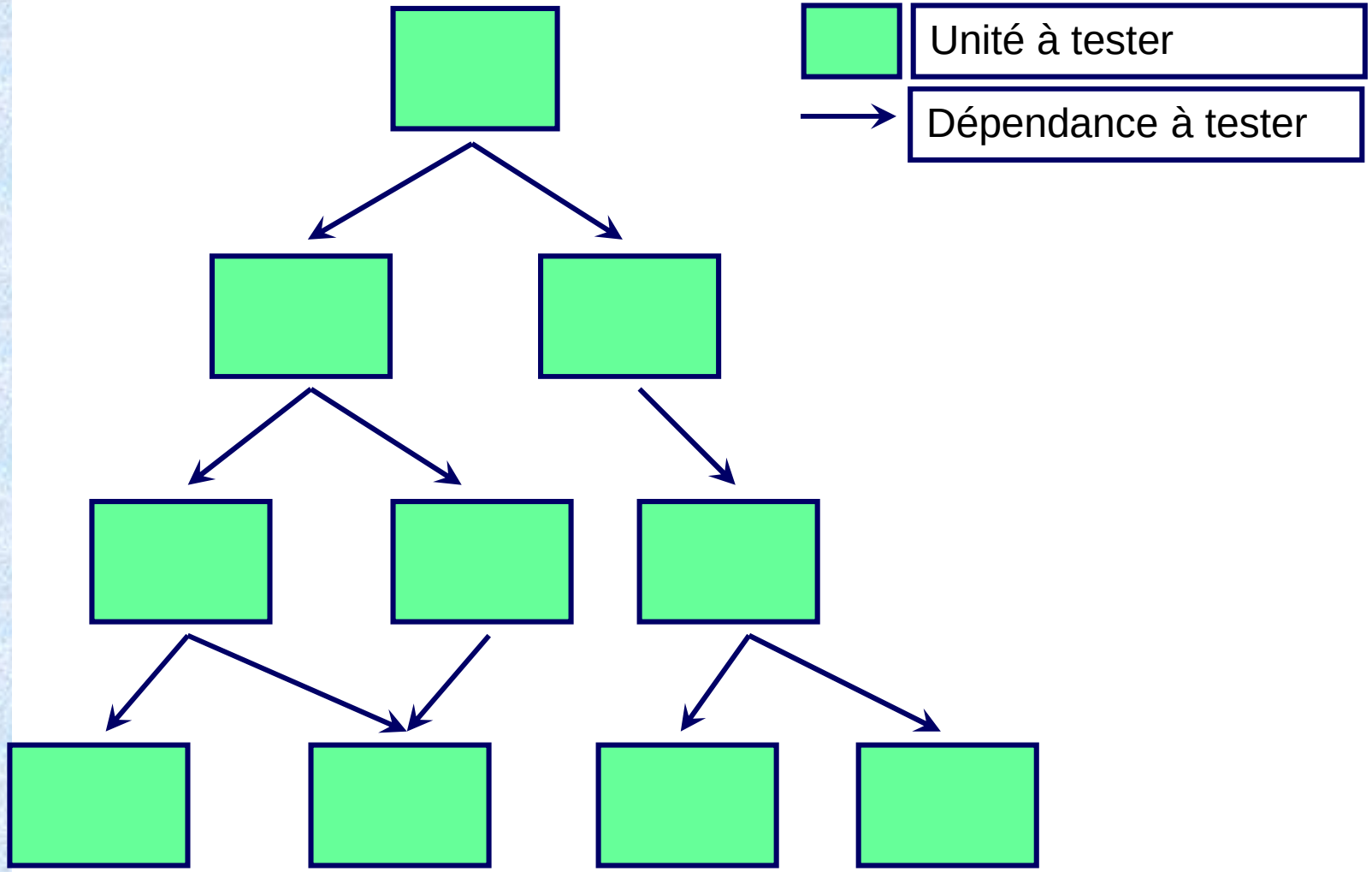
- Bon dialogue entre les modules,
  - Appel, transmission de données, compatibilité
- Livraison par module métier et le fonctionnement des composants de chaque module fonctionnel dans l'environnement d'intégration,
- Fonctionnement des menus de module métier,
- Fonctionnement des types d'habilitation et types de profils utilisateurs : les droits d'accès, le droit d'utiliser les différents modes pour chaque dialogue technique, etc.
- Synchronisations, points de reprises

Périmètre exclus par le test d'intégration

- Vérifications liées à une action fonctionnelle
- Création des différentes entités fonctionnelles

Test logiciel

Dépendance de Test d'intégration



Test logiciel

Principe de test d'intégration

Lorsqu'un module est testé en isolation

- Les modules invoqués doivent être remplacés
- Les modules testés testés doivent être invoqués

Stub

- Remplace un module invoqué
- Stub pour module de *lecture* passe les données de tests
- Stub pour module de *sortie* retourne les résultats de tests
- Peut remplacer des composants : base de donnée, réseau
- Doit être déclaré/invoqué comme le module *réel :* Même nom que module remplacé, même liste de paramètres, même type de retour, même modificateurs (static, public ou private)

Driver module

utilisé pour invoquer les modules sous test : passage de paramètres, traitement de valeurs retournées

Test logiciel

Approche de test d'intégration

- **Approche Big-Bang**
- **Approche de haut en bas (top-down)**
- **Approche de bas en haut (bottom-up)**
- **Approche mixte**

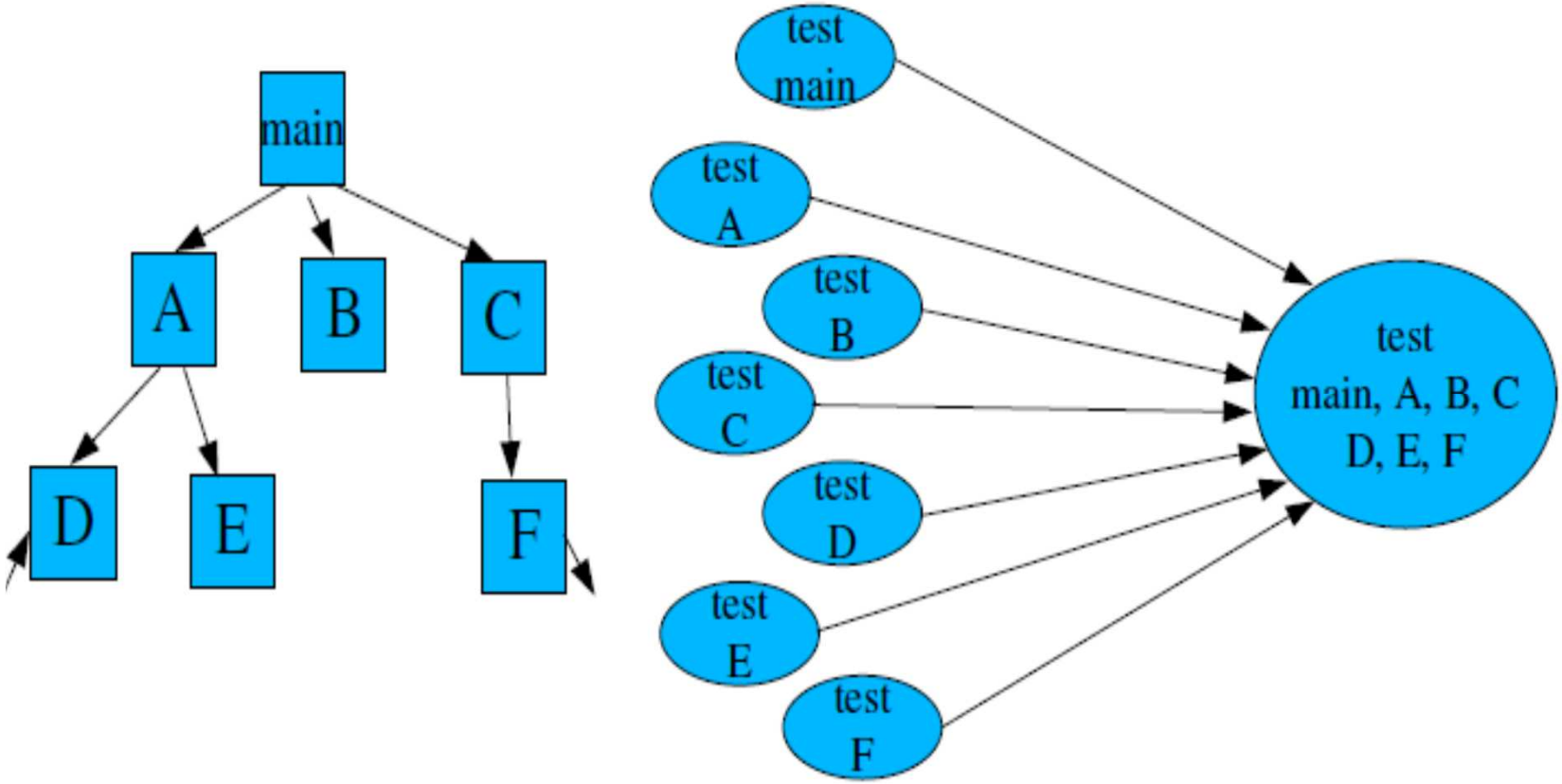
Test logiciel

Test d'intégration- Big bong

- Stratégie **non incrémentielle**
  - Test unitaire de chaque module en isolation
  - Intégration de tous les composants à tester en une seule étape. (intégration massive)
- Stratégie d'intégration rapide, utilisation privilégiée dans les petits projets.
- Stratégie inadapté pour les projets importants : risque de se perdre suite à un nombre important de modules à intégrer et perte d'efficacité.

Test logiciel

Test d'intégration- Big bong



Test logiciel

Avantages du Big-bong

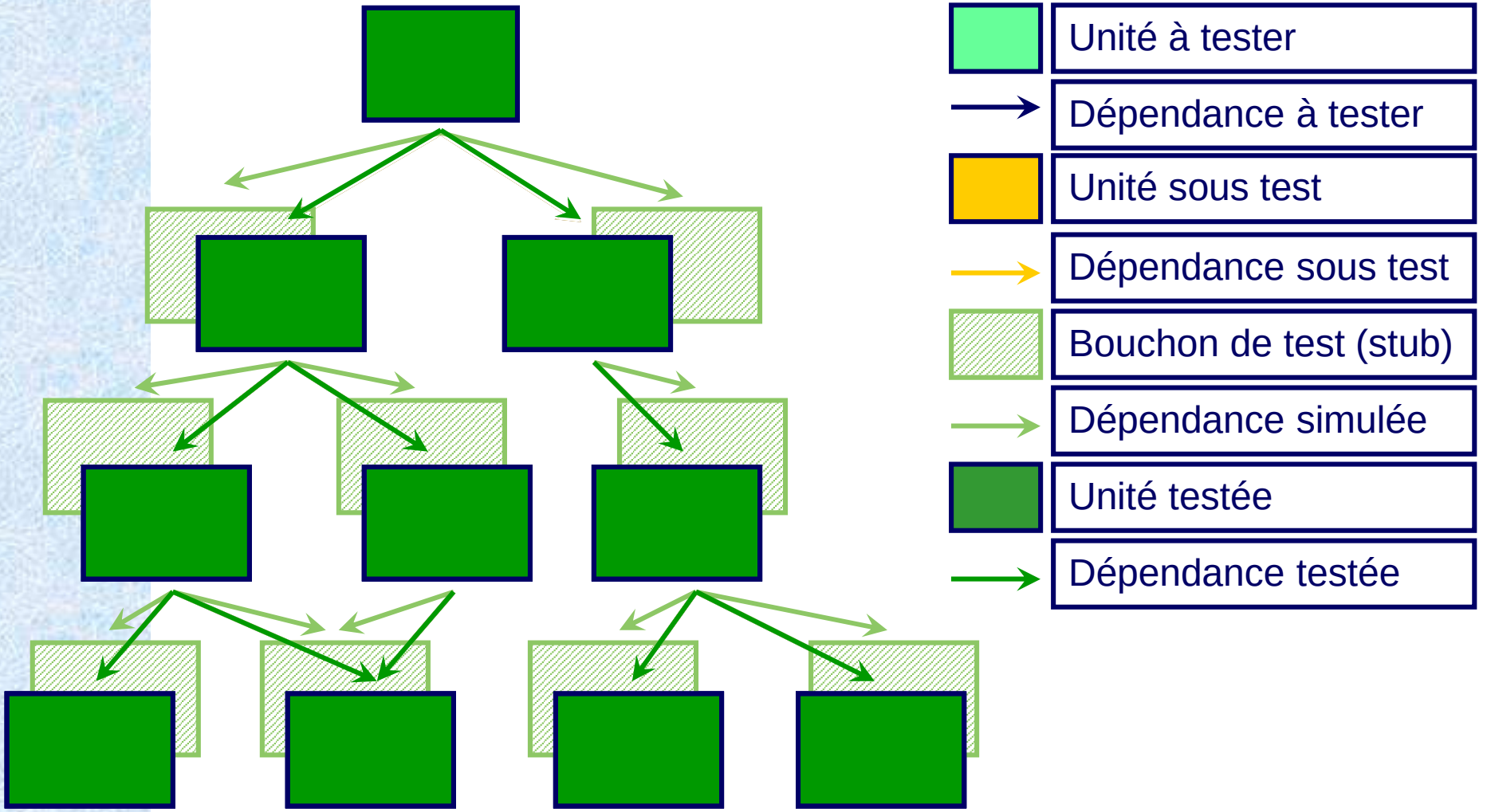
- Convient aux petits systèmes

Inconvénients du Big-bong

- Besoin de driver et stubs pour chaque module
- Ne permet pas le développement en parallèle
- Difficile de localiser les fautes
- Facile de rater des fautes d'interface

Test logiciel

Test d'intégration : Top-Dawn

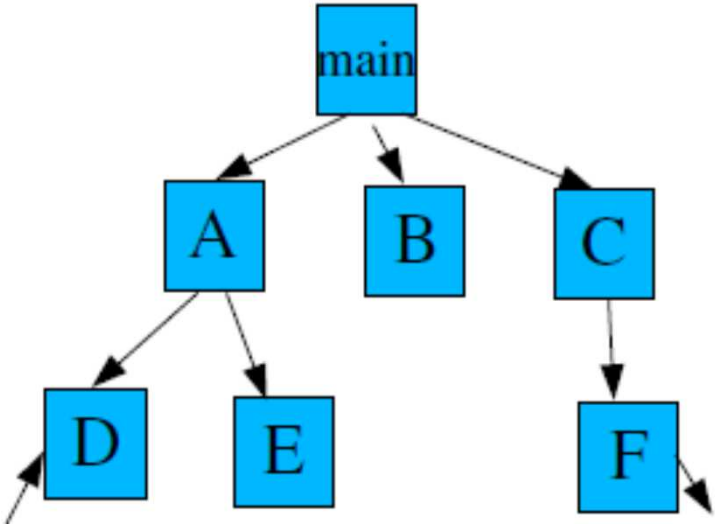


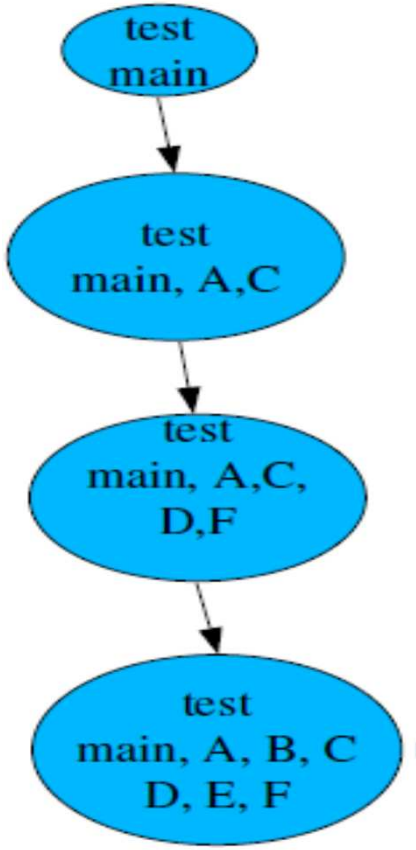
Test logiciel

Test d'intégration : Top-Dawn

Possible d'altérer l'ordre tel sont testés le plus tôt possible

- Unités critiques
- Unités d entrées/sorties



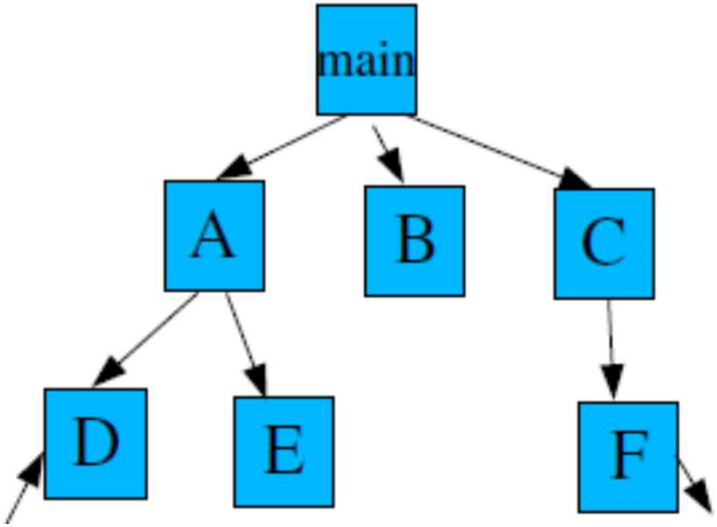


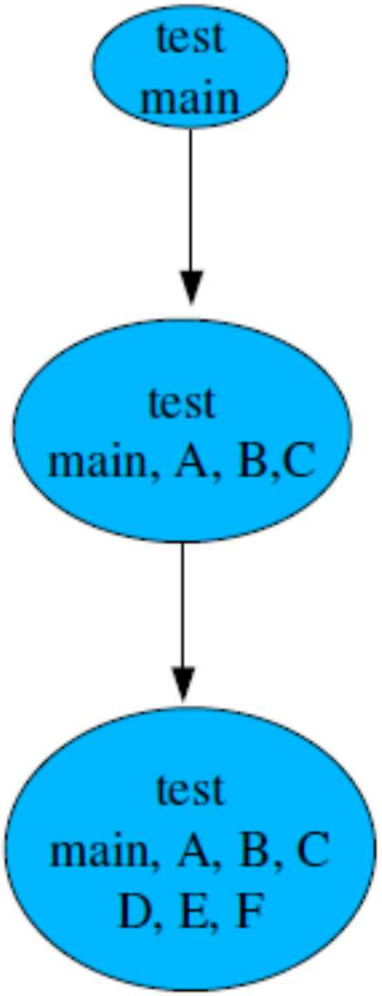
Test logiciel

Test d'intégration : Top-Dawn

Stratégie **incrémentielle**

1. Tester les modules de haut niveau, **puis**
2. Module appelés jusqu'aux modules de plus bas niveau





Test logiciel

Avantages du Top-Dawn

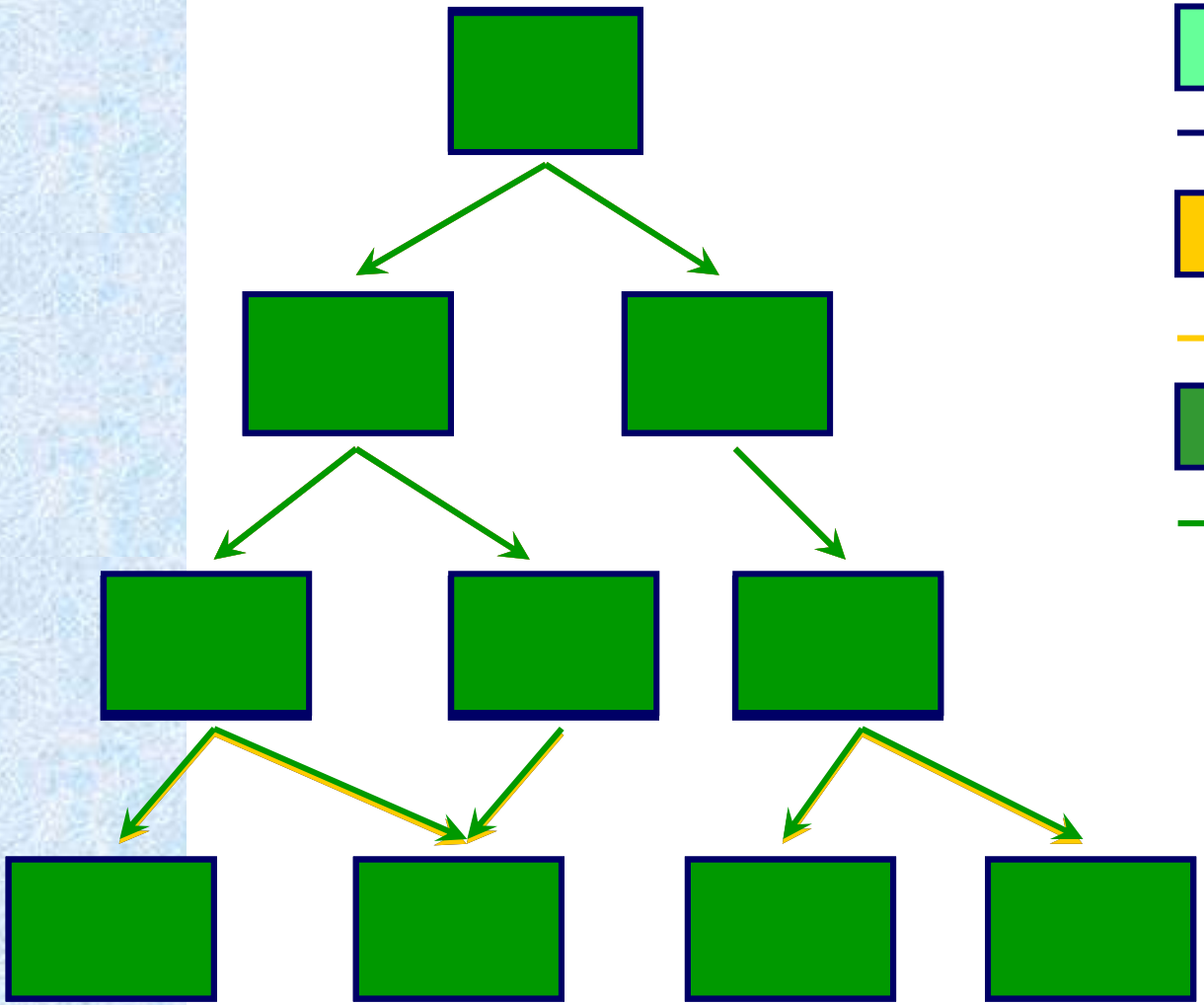
- Facilité de localisation de fautes
- Peu ou pas de *drivers*
- Possibilité d'obtenir un *prototype*
- Différents ordres de test/implémentation possibles
- Problèmes majeures de design trouvés en premier ; dans modules de la logique au dessus de la hiérarchie

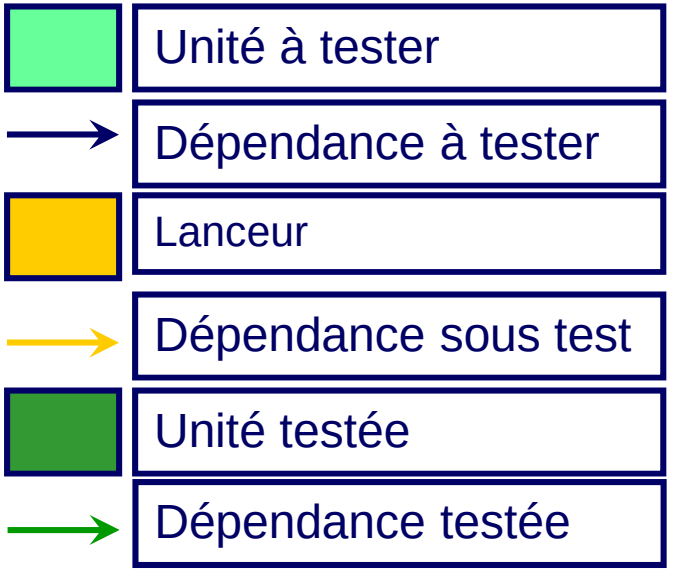
Inconvénients du Top-Dawn

- Nécessite beaucoup de stubs
- Modules potentiellement **réutilisables** (en dessous de la hiérarchie) peuvent être testés insuffisamment
- Effort important de simulation des composants absents et multiplie le risque d'erreurs lors du remplacement des bouchons.

Test logiciel

Test d'intégration- Bottom-Up



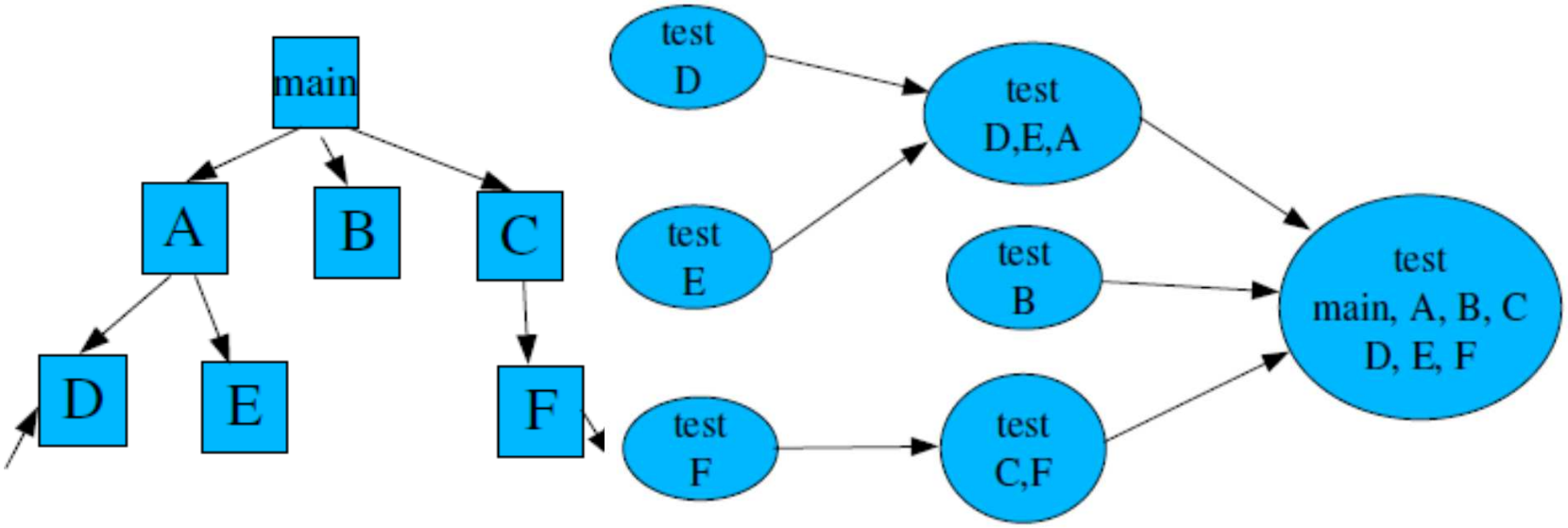


Test logiciel

Test d'intégration : Bottom-Up

Stratégie **incrémentielle**

1. Tester les modules de bas niveau, **puis**
2. Modules les appelant jusqu'au module de plus haut niveau



Test logiciel

Avantages du Bottom-Up

- Démarche est naturelle avec faible effort de simulation
- Construction progressive de l'application s'appuie sur les modules réels. Pas de version provisoire du logiciel (pas de stubs)
- Les composants de bas niveau sont les plus testés,
- Définition des jeux d'essais plus aisée

Inconvénients du Bottom-Up

- Détection tardive des erreurs majeures
- Planification dépendante de la disponibilité des composants
- La simulation par « couches » n'est pas obligatoire
- Besoin de *drivers*
- Pas de concept de squelette : *prototype*

Test logiciel

Test d'intégration- Mixte

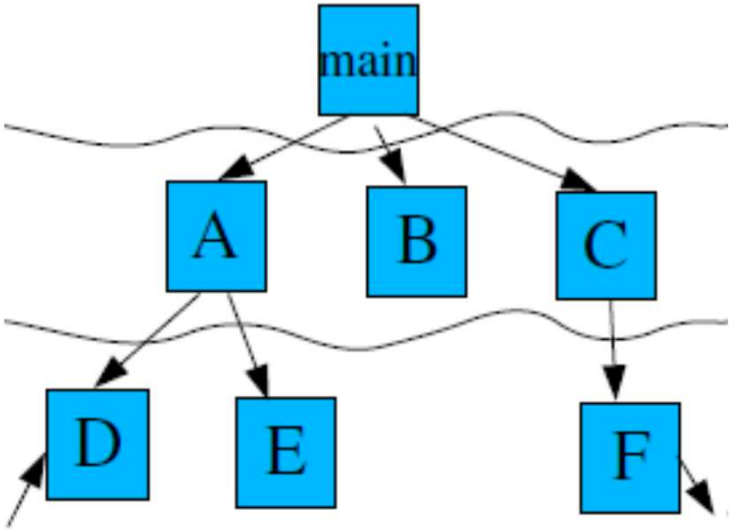
Combinaison des approches descendante et ascendante.

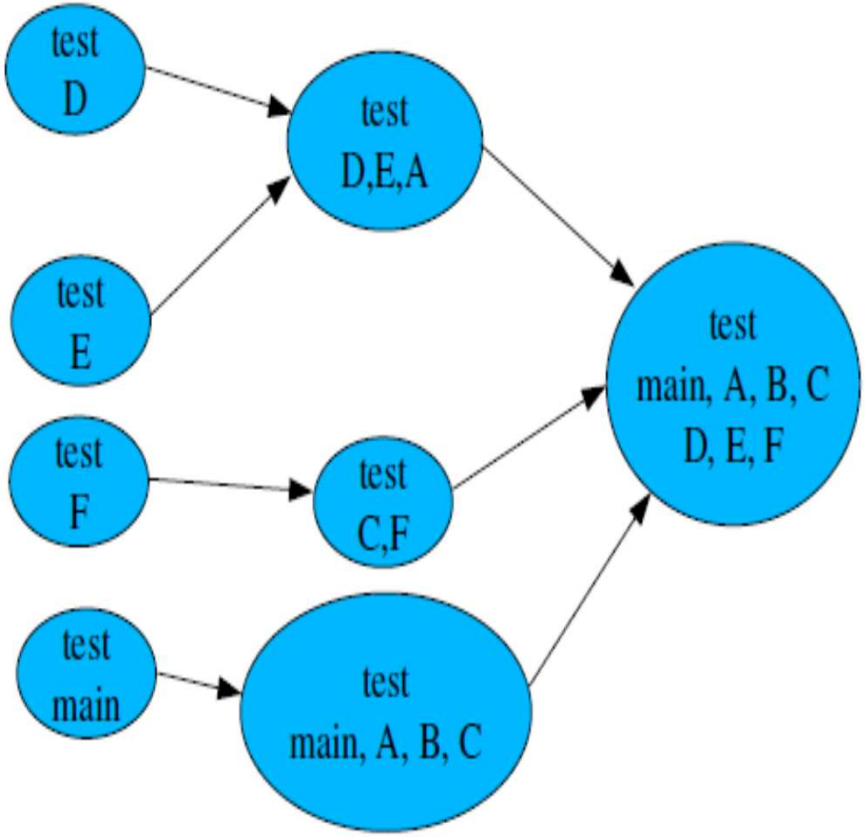
Trois couches

– logique (au dessus): topdown

– intermédiaire

– opérationnelle (dessous): bottomup



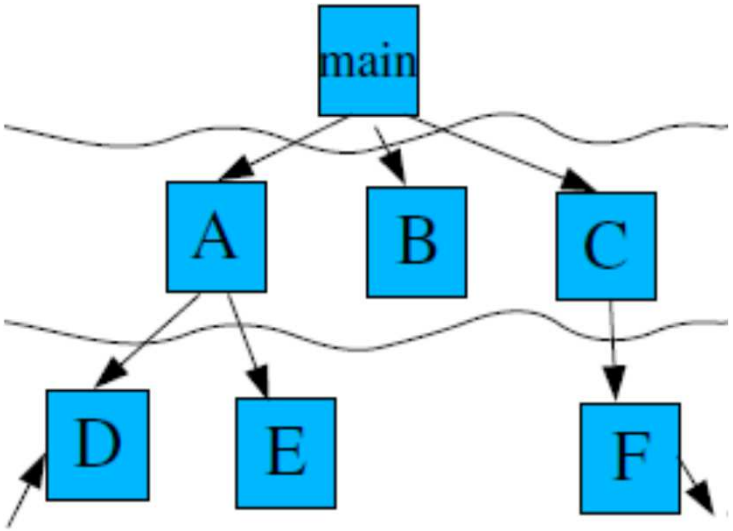


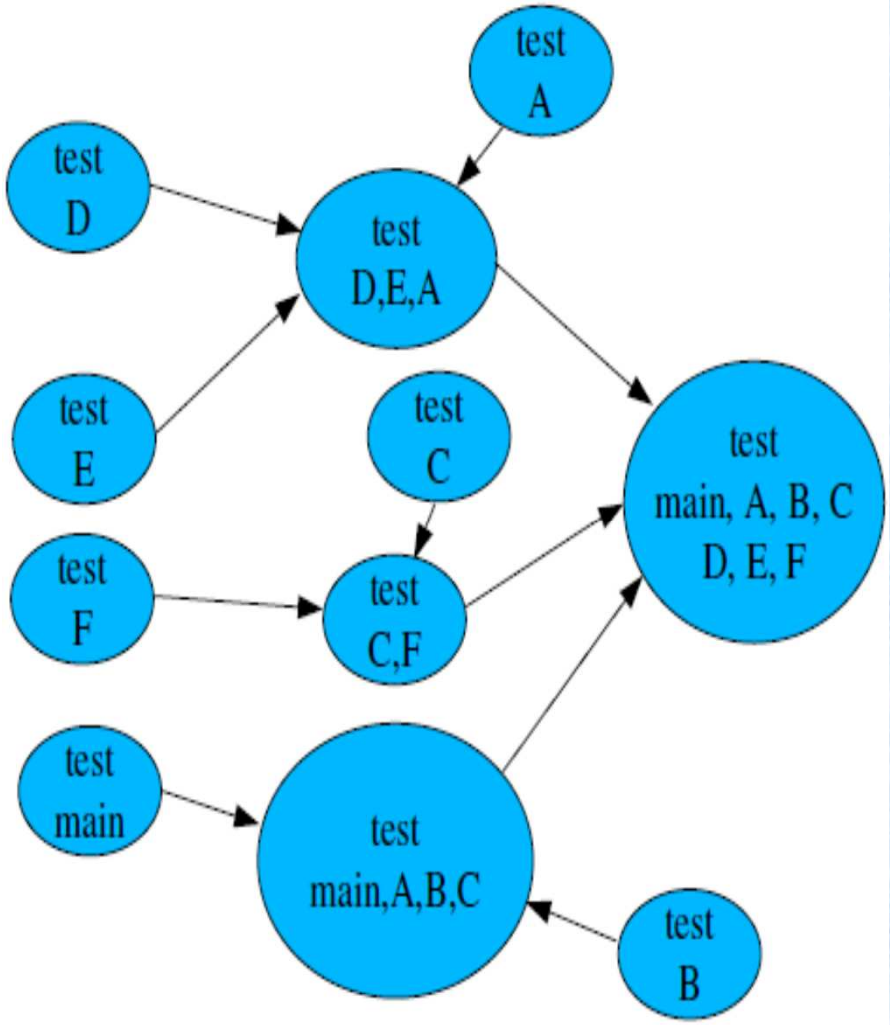
Test logiciel

Test d'intégration- Mixte modifié

Modules testés d'abord en isolation

avant intégration





Test logiciel

Avantages du test mixte :

- Suivre le **planning** de développement de sorte que les premiers composants terminés soient intégrés en premier ,
- Prise en compte du **risque** lié à un composant de sorte que les composants les plus critiques puissent être intégrés en premier.
- La principale difficulté d'une intégration mixte réside dans sa **complexité** car il faut alors gérer intelligemment sa stratégie de test afin de concilier les deux modes d'intégration : **ascendante** et **descendante**.

Test logiciel

Test système

But : Vérifier la conformité du logiciel à la spécification du logiciel

Quand ? Dès que l'ensemble des sous-systèmes fonctionnels ont été testé et intégré, il est donc effectué après l'assemblage du logiciel

Comment ? toujours réalisés sur machine cible et nécessitent généralement la fabrication d'un banc de tests élaboré et en utilisant des tests fonctionnels et de robustesse

Qui?: Ces tests sont toujours réalisés par une équipe de tests indépendante de l' équipe de développement.

Test logiciel

Objectifs du test système

Vérifie la satisfaction des exigences

- Fonctionnalité
- Fiabilité
- Recouvrement
- Multitasking
- Configuration
- Sécurité
- Compatibilité
- Stress
- Performance
- Serviabilité
- Installabilité

Test logiciel

Tests d'acceptation (User Acceptance Tests)

- Tests systèmes effectués par les clients ou sous leur responsabilité
- Vérifie si le système fonctionne selon les attentes des clients

Types commun de tests d'acceptation

- **Tests alpha:** tests usagers effectués sur un système pouvant avoir des fonctionnalités incomplètes, dans le cadre de l'environnement de développement (Effectués par un panel "maison" incluant des usagers finaux)
- **Tests Beta:** tests usagers effectués dans le cadre de l'environnement de l'usager final

Test logiciel

Tests de Fonctionnalité

- Assurer que le système supporte les exigences fonctionnelles requises
- Cas de tests dérivés par des exigences
  - Sous forme traditionnelle
  - Cas d'utilisations

Test logiciel

Forced Error Test (FET)

- **Forçage du système** dans toutes les situations **d'erreurs** à l'aide d'une liste de toutes les erreurs possibles
- **Vérification de**
  - La conception du traitement d'erreurs, consistance des méthodes de communication
  - La détection/traitement de conditions d'erreurs communes
  - Le recouvrement du programme de chaque condition d'erreur
  - La correction d'états instables provoquées par les erreurs
  - Vérifie des messages d'erreurs pour assurer: la correspondance entre message et type d'erreur détecté, une description d'erreur est claire et concise, contourner et recouvrer l'erreur…

Test logiciel

Tests d'Utilisabilité

Facteurs influençant la facilité d'utilisation du système

- **Accessibilité:** facilité d'entrer, de navigation, sortie du système
- **Réponse du système:** possibilité des usagers à faire ce qu'ils veulent quand ils le veulent de façon intuitive/convenable
- **Efficacité:** possibilité des usagers à effectuer leur tâche de façon optimale selon le temps, nombre d'étapes, …
- **Compréhensibilité:** possibilité d'apprentissage de l'utilisation du système, documentation, aide

Activités typiques

- Expériences contrôlées dans des environnements simulées avec des usagers novices à experts
- Analyse post-expérimentale par experts humains, psychologues…
- Collection de données pour améliorer l'utilisablité du logiciel

Test logiciel

Tests d'Installabilité

Met l'accent sur exigences liées à l'installation

- documentation
- processus d'installation
- fonctions de support système

Cas de Tests devraient comprendre

- état de début/entrée
- exigence à tester (but du test)
- scénario d'installation/désinstallation (actions et entrées)
- résultat escompté (état final du système).

Test logiciel

Tests de Serviceabilité

Met l'accent sur exigences de maintenance

- Procédures de changements (pour scénarios de services adaptatifs, perfectifs et correctifs)
- Documentation de support
- Outils de diagnostics

Test logiciel

Tests de Performance/Stress/Charge

- **Tests de Performance** : évaluent la satisfaction d'exigences de performance spécifiées : temps de réponse, utilisation de mémoire, ratio entrée/sorties…

- **Tests de Stress** : accent sur le comportement du système à, au alentours de au delà de conditions de surcharge en conjonction avec tests de performance pour pousser le système à la panne

- **Tests de Charge** : vérifie traitement d'une charge particulière tout en maintenant des temps de réponse acceptables en conjonction avec tests de performance

Test logiciel

Phases des Tests de Performance

1. Planification

- Définir les objectifs, délivrances, attentes
- Regrouper les exigences de système et de tests
- Sélectionner métriques de performance à collecter
- Écrire le plan de test, concevoir scénarios-usagers, créer scripts

2. Phase de Test

- Générer les données de test, exécuter les tests et collecter les résultats

3. Phase d'analyse

- Analyse des résultats pour localiser la source de problèmes logiciels et matériels
- Changement du système pour optimiser la performance logicielle et matérielle
- Conception de tests additionnels (si objectif de test non atteint)

Test logiciel

Tests de Configuration

- Vérifie la **compatibilité** avec toutes les configurations du matériel supporté
- Définie le **degré** avec lequel une application ou composant peut être configuré en variantes multiples selon les exigences de configurabilité
- Consiste en **l'exécution** d'un ensemble de tests sous différentes configurations : tests exhibant les principales fonctionnalités du système de tests
- Sélectionne les **configurations à tester** : type et caractéristiques de matériel à tester afin de ramener les configurations identifiées à un nombre gérable

Test logiciel

Tests de Compatibilité

- Test de compatibilité avec les autres **ressources** système de l'environnement d'opération : logiciels, bases de données, standards, etc.
- Test de compatibilité **code source ou objet** avec différentes versions de l'environnement d'opération
- Tests de **compatibilité/conversion** lorsqu'il y a des procédures ou processus de conversion
- **Compatibilité en arrière :** Compatibilité avec les versions précédentes du logiciel
- **Compatibilité en avant** : Compatibilité avec les versions futures du logiciel
- Sélection des **combinaisons du logiciel** selon la popularité, l'âge, le type ou le fabricant

Test logiciel

Tests de Sécurité

- Met l'accent sur les **vulnérabilités** liées à l'accès non-autorisé ou à la manipulation du système
- Identifie les **vulnérabilités** et protéger le système
  - **Données** : Intégrité, confidentialité et disponibilité
  - **Ressources réseau**
- Pour évaluer un logiciel par rapport à la sécurité
  - Identifier les **zones** du logiciel susceptibles d'être exploités pour les attaques de sécurité
  - Définir les **tests de Pénétration** : essayer de pénétrer un système en exploitant les méthodes de pirates
  - Faire le **test de mot-de-passe** : en utilisant des outils de dépassement buffer ou d'injection SQL

Test logiciel

Tests de Multitasking

- Investigation de l'exécution **simultanée** de multiple tâches
- **Source** potentielle de problèmes
  - interférences entre **sous-tâches** exécutantes
  - interférences lorsque **copies multiples** exécutent
  - interférences avec **d'autres produits** exécutants
- Tests conçus pour **révéler** des erreurs de timing possibles, forcer la contention pour les ressources, etc.
- **Difficultés**
  - **La reproductibilité** n'est pas garantie : ordonnancement varié de l'ordre d'exécution des tâches, répétition des tests plusieurs fois…
  - Comportement peut dépendre de **la plate forme** (matériel, système d'exploitation, ...)

Test logiciel

Tests de Recouvrement

Habilité à récupérer de pannes avec des conditions exceptionnelles associés avec matériel, logiciel ou personnes

- Détection de **pannes** et conditions exceptionnelles
- Passage au systèmes en standby
- Récupération de **l'état d'exécution et configuration** (incluant statut de sécurité)
- Récupération **de données et messages**
- Remplacement de **composants en panne**
- Sortie de **transactions incomplètes**
- Maintenance de **pistes d'audits**
- **Procédures externes** : bandes backup ou scénarios de désastres variés

Test logiciel

Tests de Fiabilité

- Popularisés par l'approche de développement **Clean-room (d'IBM)**
- Application de **techniques statistiques** à des données collectés durant le développement et l'opération (profil opérationnel) pour spécifier, prédire, estimer, et évaluer la fiabilité d'un système
- **Exigences de fiabilité** exprimés en terme de
  - **Probabilité de non panne** dans un intervalle de temps a spécifié
  - **Temps moyen** entre pannes escompté (MTTF)

Test logiciel

Test de non-régression

- Les logiciels subissent continuellement des **modifications** (par exemple : maintenance corrective, maintenance évolutive). Il est par conséquent nécessaire d'effectuer une revalidation du logiciel.
- **Revalidation** d'un système logiciel suite à des **modifications**
  - Identification des **tests à ré-exécuter**
  - Identification des **tests à ajouter**
- Pour des raisons de coûts et de délais, il n'est pas raisonnable de ré-exécuter l'ensemble des tests effectués sur la **version initiale**.

Test logiciel

But du test de non-régression

- **Optimiser le processus de validation** d'un logiciel suite à ces modifications. C'est un processus de démonstration de la couverture de l'impact des modifications.
- **Quand fait-on des tests de non-régression ?:** Ils se déroulent durant chaque phase de développement des modifications (ie de la spécification des modifications jusqu'au codage). Ils font partie du processus d'analyse d'impact des modifications du logiciel.
- **Re-testage sélectif d'un système ou d'un composant** pour vérifier que les modifications n'ont pas causés d'effets fortuits et que le système ou le composant est toujours conforme à ses exigences spécifiées
  - Pour confirmer que les bugs fixés l'ont été effectivement
  - De nouveau bugs n'ont pas été introduits
  - Les fonctionnalités qui étaient corrects sont intactes

Test logiciel

Portées du test de non-régression

- **Locale:** tests direct du code changé/ajouté
- **Alentours:** test des fonctions supportes ou directement touchées par le changement
- **Confidence:** suite de test prédéfinie exécutée de façon routinière après de chaque changement au produit/système
- **La suite de tests** de régression inclus les cas de tests les plus effectifs : tests de bornes , tests ayant montrés des bugs et tests pour bugs rapportés par les usagers
- **La taille de la suite** de tests de régression doit demeurer raisonnable : réduire le nombre de cas de tests redondants non-effectifs

Gestion de configuration logicielle

Définition

- « ***La GCL est une discipline de management de projet qui permet de définir, d'identifier, de gérer et de contrôler les articles de configuration tout au long du cycle de développement d'un logiciel.*** »

- GCL est une activité de soutien de projet qui peut être définie comme un outil de communication sophistiqué entre des acteurs indépendants, contribuant à l'édification de systèmes ouverts

- GCL permet la gestion des documents, Les logiciels d'application, des logiciels de base, des différentes fiches produites, le matériel…

- GCL est utilisée pour la gestion de systèmes complexes : aéronautique, spatiale, automobile…

Gestion de configuration logicielle

Objectifs

- Acquérir et de présenter de manière claire et complète la configuration instantanée du produit et **l'état d'accomplissement des spécifications** physiques et fonctionnelles
- Gérer et régénérer les différentes versions d'un produit logiciel
- Récupérer facilement les composants d'un logiciel existant
- Établir la trace des interventions faites sur un logiciel
- Eviter les modifications intempestives,
- Assurer des livraisons d'ensembles cohérents
- Eviter les pertes d'informations
- Utiliser une documentation correcte et exacte à **n'importe quel instant du cycle** de vie du produit et par tout intervenant

Gestion de configuration logicielle

Activités

- Identification de la configuration,
- Maîtrise de la configuration,
- Enregistrement de l'état de la configuration,
- Audit de la configuration

Gestion de configuration logicielle

Identification de la configuration

- Déterminer les constituants du produit,
- Choisir les articles de configuration,
- Fixer dans des documents les caractéristiques physiques et fonctionnelles d'un article de configuration (interfaces et évolutions ultérieures)
- Allouer des caractères ou des numéros d'identification aux articles de configuration et à leurs documents ,

Gestion de configuration logicielle

Maîtrise de la configuration

Activités comprenant la maîtrise des évolutions des articles de configuration après établissement formel de leur document de configuration.

Objectif

- Maintenir la cohérence
- Permettre des évaluations des impacts techniques, des coût et délais et de l'écarts par rapport au référentiel
- Maîtriser les modifications

Les documents

La fiche de fait technique , la fiche de modification et la fiche de version logiciel

Gestion de configuration logicielle

Le circuit de la maîtrise des configurations

- **Description des processus**
  - Processus privé, interne ou officiel
  - Chaque processus s'appuie sur un domaine de visibilité
- **Gestion des espaces**
  - Un espace est un compte utilisateur placé sous la responsabilité d'un individu ou d'un groupe dont l'accès est contrôlé et dans lequel sont stockés les produits de développement logiciel à leurs différents états de développement
  - Description des principaux espaces : Développement, Intégration, Validation, Référence, Archivage et Livraison
  - Les transferts entre les espaces

Gestion de configuration logicielle

Enregistrement de l’état de la configuration

Action d'enregistrer et de présenter sous une forme définie les documents établis pour la configuration, ainsi que l'état des demandes d'évolution et de la mise en œuvre des évolutions approuvées

Archivage des " états stables " successifs des produits et de vérifier la complétude et la cohérence de ces " états ".

Suivi des versions de chacun des éléments identifiables constituant les différents produits constituant l’application

Gestion de configuration logicielle

Audit de la configuration

Examen destiné à s'assurer de la conformité d'un article de configuration avec ses documents de configuration.

- Contrôle des modifications apportées à chaque composant d'un logiciel
- Les jeux d'essais, les données et les procédures de test,
- Revue de l'assurance qualité

Gestion de configuration logicielle

GCL de développement

Cette GCL est centrée sur l'activité des **développeurs**. Chaque modification de chaque composant source est alors enregistrée.

Le contrôle de modifications intempestives est mis en place par des mécanismes de réservation. La fabrication des exécutables est partie intégrante des fonctionnalités

Gestion de configuration logicielle

GCL d'intégration

- La GCL doit permettre d'ordonnancer avec une sécurité maximum et une déperdition de temps minimum les opérations de réassemblage qui surviennent à chaque intégration
- Assemblage des ensemble : développés par la même équipe ou ayant les même caractéristique technique
- Le problème de la réservation d'un source par un développeur s'estompe au profit de l'alimentation d'un environnement de maintenance ou de qualification.
- La GCL fonctionne comme un serveur de configurations dont chacune implémente un ensemble cohérent de fonctionnalités

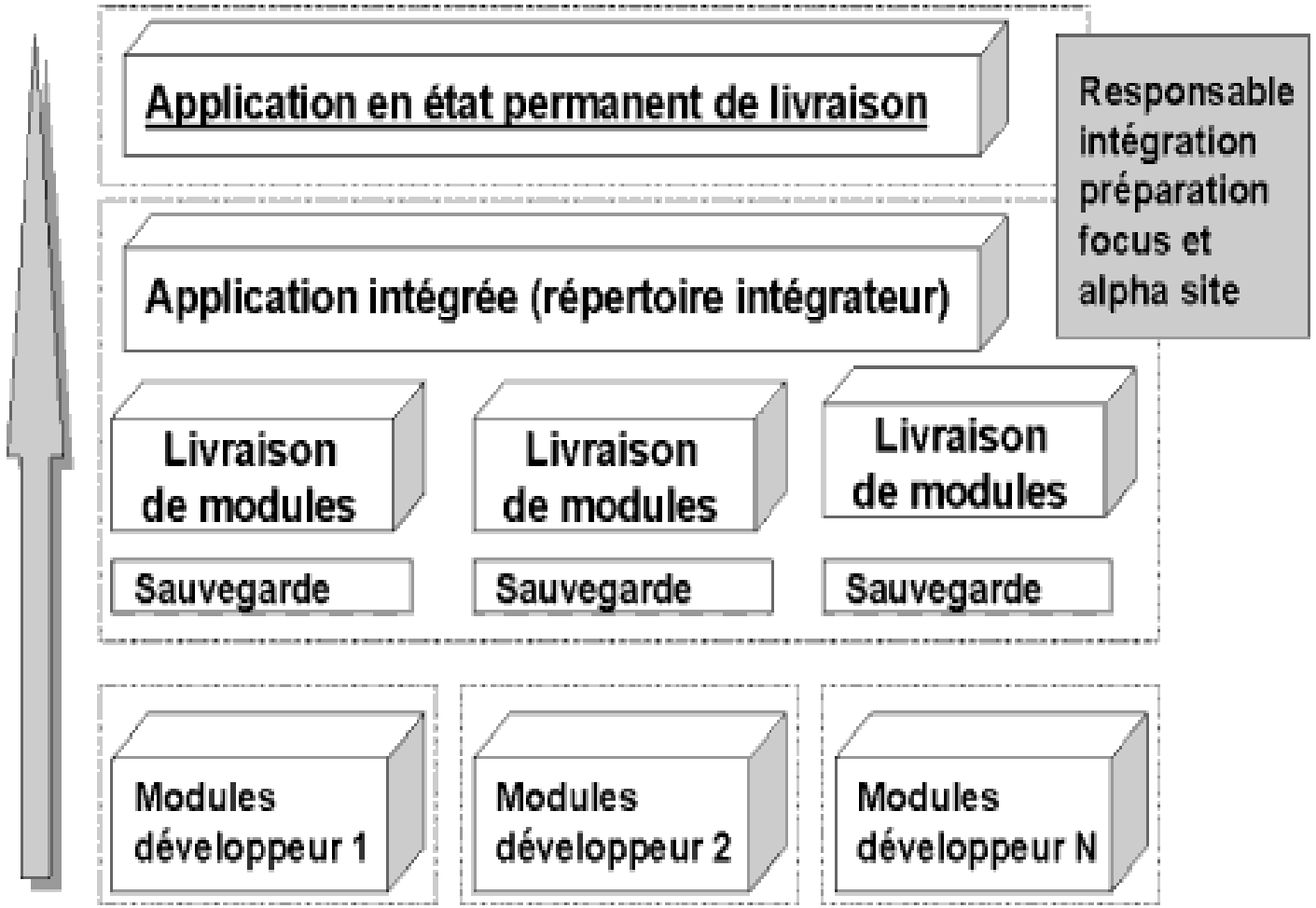
Gestion de configuration logicielle

Solution GCL

- **L'espace de travail** : gérer des tâches multiples, développer et de fabriquer des logiciels en parallèle mais en parfaite cohésion, de partager des fichiers devant être communs et d'isoler des fichiers spécifiques pertinents pour certaines tâches
- **Les processus de développement** : Suivis des fichiers source du logiciel, des modifications, et du travail réaliser par chaque équipe
- **Les composants du produit** : Liste complète et règles d'évolutions
- **Les versions** : en préalable à chaque livraison ou focus, le membre de l'équipe projet responsable de l'intégration procède à une opération de gestion structurée des documents en respectant un mode opératoire de fiabilisation et de sécurisation

Gestion de configuration logicielle

Solution GCL



Normes et standards de qualité logiciel

- **Règle** fixant les **conditions** de la **réalisation** d'une opération, de l'exécution d'un objet ou de **l'élaboration** d'un produit dont on veut unifier l'emploi ou assurer l'interchangeabilité.
- **Document** établi par **consensus** et approuvé par un **organisme reconnu**, qui fournit, pour des usages communs et répétés, des règles, des lignes directrices ou des caractéristiques, pour des activités ou leurs résultats, garantissant un niveau d'ordre **optimal** dans un **contexte donné**

Modèles de Certification : détecter et corriger les produits non-conformes (**Norme ISO 9000)**

Modèles de Maturité et d'Amélioration : mesurer l'aptitude d'un fournisseur à réaliser et à fournir du logiciel (**CMM**)

Normes et standards de qualité logiciel

ISO 9000 (International Standard Organization 9000)

- **Ensemble de recommandations et standards** pour la garantie de la qualité dans les relations clients-fournisseurs (pas spécialement logiciel).

- **Philosophie ISO 9000 :** Toute opération influençant la qualité doit être sous contrôle (qui doit être visible)

- La **certification** permet d'obtenir une large reconnaissance de ce contrôle. Elle est payante et valable trois ans.

- Pour les **applications civiles** adoptée par **60 pays**. C'est un passage obligé afin d'avoir la certification pour pouvoir répondre à un appel d'offres.

Normes et standards de qualité logiciel

Versions ISO 9000 (1)

- **ISO 9000-1 en 1994** ***Norme pour le management de la qualité*** *:* Ligne directrice pour leur sélection et utilisation de ces standard.

- **ISO 9000-2 en 1997** ***Normes pour le management de la qualité et l'assurance de la qualité :*** Lignes directrices pour l'application de l'ISO 9001, l'ISO 9002 et l'ISO 9003

- **ISO 9000-3 en 1997** ***Normes pour le management de la qualité et l'assurance de la qualité :*** Lignes directrices pour l'application de l'ISO 9001 au développement, à la mise à disposition, à l'installation et à la maintenance de logiciel.

- **ISO 9000-4 en 1993** ***Normes pour la gestion de la qualité et l'assurance de la qualité :*** Guide de gestion du programme de sûreté de fonctionnement

Normes et standards de qualité logiciel

Versions ISO 9000 (2)

- **ISO 9001 en 1994 :** *Systèmes qualité - Modèle pour l'assurance de la qualité en conception, développement, production, installation et prestations associées*

- **ISO 9002 en 1994 :** *Systèmes qualité - Modèle pour l'assurance de la qualité en production, installation et prestations associées*

- **ISO 9003 en 1994 :** *Systèmes qualité - Modèle pour l'assurance de la qualité en contrôle et essais finals*

Normes et standards de qualité logiciel

Versions ISO 9000 (3)

- **ISO 9004-1 en 1994 :** ***Management de la qualité et éléments de système qualité*** -- Lignes directrices pour répondre aux besoins des clients et de l'organisation.

- **ISO 9004-2 en 1991 :** ***Gestion de la qualité et éléments de système qualité*** -- Lignes directrices pour les services

- **ISO 9004-3 en 1993 :** ***Management de la qualité et éléments de système qualité*** – Lignes directrices pour les produits issus de processus à caractère continu

- **ISO 9004-4 en 1993 :** ***Gestion de la qualité et éléments de système qualité*** -- Lignes directrices pour l'amélioration de la qualité

Normes et standards de qualité logiciel

Versions ISO 9000 (3)

- **La norme ISO 9000 (2000**) est en cours d'élaboration et remplacera la norme ISO 9000 de 1994. L'ISO 9000 (2000) ne comportera plus que deux normes:

- la norme 9001 qui devient la norme d'exigence unique.
- la norme 9004 qui devient la norme guide de management.

La nouvelle norme impliquera davantage le management dans les disfonctionnements rencontrés dans la démarche qualité.

Normes et standards de qualité logiciel

- La norme définit les 20 éléments de qualité à respecter:

1. Responsabilités du management

- Définition d'une politique qualité
- Mise en place d'une organisation
- Validation périodique du système qualité

2. Définition d'un système de qualité

La politique qualité doit être documentée dans un manuel qualité conforme aux normes en vigueur et aux habitudes de l'entreprise Il est important que les procédures qualité mises en place trouvent l'agrément des développeurs et ne soient pas perçues comme un frein à leur créativité mais plutôt comme un cadre rassurant dans lequel ils pourront évoluer et produire un travail de qualité

Normes et standards de qualité logiciel

3. **Analyse du contrat entre client et fournisseur**

Il est tout à fait indispensable de ne pas s'engager sur un contrat irréaliste.

4. **Contrôle de la conception**

Dans tous les cas la conception nécessite un contrôle rigoureux par des inspections qui peuvent être réalisées au moyen de listes de défauts typiques ou d'autre de testes

5. **Spécification des achats et fournitures**

6. **Contrôle des produits fournis par le client**

Contrôle des Achats, de la sous-traitance, et des fournitures procurées par le client

Normes et standards de qualité logiciel

7. Identification des produits et traçabilité

Suivi tout au long du cycle de développement des liens entre cahier des charges, spécifications, conception et codage. La norme impose de savoir répondre aux questions concernant la documentation les corrections les exécutions , les rapport incident…

8. Contrôle du processus

Il s'agit ici du contrôle de la production des disquettes, CD ou tout autre media sur lequel le produit final sera livré. A vérifier également que la bonne version soit livrée en ce qui concerne les bibliothèques, jeux de tests, documentation…

Normes et standards de qualité logiciel

9. Gestion de configurations: spécifique aux projets logiciels

Les produits logiciels sont constitués de différents éléments qui évoluent au cours du temps et qui peuvent différer d'une installation à l'autre

Contrôler l'ensemble des données constituant le système:

– Documents (sources, jeux de tests, plans d'intégration, . . .)

– Assurer la cohérence des divers composants

– Construire/reconstruire un système

10. Gestion de la documentation et des données

- Quiconque en a besoin doit pouvoir y accéder
- Tout document doit être approuvé
- Versions des documents cohérentes entre elles et avec le code qu'elles documentent.

Normes et standards de qualité logiciel

11. Test et inspection

Nécessité de mettre en œuvre des tests et inspections et des mesures des outils de tests

12. Statut des tests et inspections

On doit pouvoir à tout moment savoir si un document ou un code source a été inspecté, validé ou est en attente de validation. En aucun cas un document non validé ne doit servir de base à de nouveaux développements Les documents inspectés et validés doivent être conservés dans un espace différent de ceux qui ne le sont pas

13. Contrôle des produits non-conformes

Les produits non conformes doivent être identifiés. On doit pouvoir fournir une procédure qui permet de corriger rapidement les erreurs et de valider les modules ainsi corrigés.

Normes et standards de qualité logiciel

14. Actions correctives et préventives

La norme est assez floue à ce propos car elle ne prévoyait pas l'amélioration des produits ; alors qu'en informatique les produits évoluent dans le temps . Il faut s'assurer qu'il existe une procédure de report des erreurs, s'assurer que toute erreur reportée est un jour corrigée, pouvoir répondre sans hésiter à la question « telle erreur a-t-elle été corrigée?.

15. Emballage, stockage, livraison,

Une fois les produits développés, il faut les stocker, les emballer et les acheminer vers le client,

Normes et standards de qualité logiciel

16. Contrôle des enregistrements concernant la qualité

On veillera à garder trace de tous les enregistrements qualité effectués en interne (inspection, tests..) et chez les clients (mesures statistiques en particulier, erreurs reportées et le sort qui leur a été réservé…)

17. Audit qualité internes

Il s'agit de vérifier que les plans qualité sont respectés dans l'entreprise de manière générale. On pourra le prouver si on a fait une procédure sérieuse de collecte de données. Il faut pouvoir produire une procédure d'audit lors de la certification ou de son renouvellement.

18. Organiser la formation

La norme encourage la formation programmée à tous les niveaux:

- Nouvelles techniques, nouveaux langages
- Formation aux méthodes qualité
- Formation aux méthodes de gestion de projet

Normes et standards de qualité logiciel

19. Service après vente

Prévoir un contrat de maintenance

20. Techniques statistiques

Dans une industrie de production il est important de pouvoir prélever des échantillons au hasard et de les tester. Les tests statistiques consistent à faire des mesures a posteriori en phase d'exploitation:

- Nombre de pannes
- Nombre d'erreurs découvertes (par le client, par le fournisseur)
- Temps moyen entre deux pannes
- Durée moyenne d'indisponibilité .

Normes et standards de qualité logiciel

Capability Maturity Model (CMM)

- Développé par **Software Engeneering Institute**, financé par le **Department of Defense (**DoD), associé à **l'Université Carnegie Mellon**

- Sa **mission** est de **promouvoir le transfert de technologie** en matière de logiciel, particulièrement pour les entreprises travaillant pour le DoD le modèle de maturité proposé fin des années 1980, raffiné en 1993 a eu une grande influence dans l'amélioration des processus

- **Avantages :** dédiée spécifiquement à l'industrie du logiciel : plus détaillé et spécifique que ISO 9000. Cependant, moins répandu et réputé en Europe que la norme ISO 9000.

Normes et standards de qualité logiciel

Niveau 1 : Initial

- Le processus de développement est « ad hoc », et parfois même chaotique.
- Le logiciel est développé sans méthode prédéfinie : Peu de procédures sont définies
- Le développement repose sur la compétence de quelques personnes : le succès repose sur des efforts individuels.
- Il est impossible de récupérer l'expérience acquise dans un projet lors du développement d'un autre projet
- On ne peut prédire en terme de gestion de projet des éléments aussi importants que la taille ou le coût d'un projet
- Les réactions se font essentiellement en fonction des crises et non pas de façon systématique et planifiée
- Beaucoup d'organisations sont encore assez proches de ce niveau.

Normes et standards de qualité logiciel

Niveau 2 : Répétable (Méthodes élémentaires de gestion)

- Système commun de management de projet et des techniques de contrôle
- La gestion de projet et la planification sont faites sur des bases reproductibles
- Des activités de mesures commencent à être mises en place, en particulier au niveau des fonction, des coûts et des délais
- On réagit de façon planifiée et non plus en fonction des crises
- Les problèmes sont traités au fur et à mesure qu'ils arrivent et ne sont pas accumulés jusqu'à provoquer une crise majeure
- Les mesures utilisées pendant un projet permettent de prévoir ce qui sera susceptible de se passer pour les projets futurs.

Normes et standards de qualité logiciel

Niveau 3 : Défini (Définition du processus de développement)

- Les processus de gestion et technique sont documentés, standardisés à un processus standard de l'organisation.
- Tous les projets utilisent une version approuvée et adaptée des processus standards pour développer et maintenir le logiciel.
- Gestion de configuration rigoureuse, respects des normes et standards, inspections et tests formels, existence d'un service de GL ou Qualité logiciel.
- Des mesures sont faites régulièrement pour améliorer le processus
- Des revues sont mises en place afin de garantir la qualité du logiciel.

Normes et standards de qualité logiciel

- **Niveau 4 : Maîtrisé (Gestion du processus de développement)**

  - Les processus et le produit sont quantitativement compris et contrôlés.
  - Le procédé de développement logiciel est stable et permet de garantir un niveau constant de qualité
  - Des objectifs précis de qualité et de productivité sont affectés à chaque projet
  - Des mesures détaillées du développement et de qualité sont collectées. Des actions correctrices sont prises dès qu'une divergence par rapport aux objectifs est constatée
  - Des procédés de mesure statistiques sont mis en place afin de déterminer s'il s'agit d'un manquement passager aux objectifs ou bien s'il s'agit d'une divergence grave par rapport aux standards de productivité et de qualité.

Normes et standards de qualité logiciel

Niveau 5 : Optimisé (Contrôle et optimisation)

- Les processus sont continûment améliorés par les analyses des mesures.
- Le processus de développement porte en lui les moyens de réaliser sa propre optimisation
- Des contrôles statistiques sur le processus sont utilisés pour guider l'organisation
- Le processus intègre un feed back provenant de ces mesures

Remarques :

- De nombreuses entreprises ont atteint les niveaux 2 et 3
- Pratiquement aucune n'est encore arrivée aux niveaux 4 et 5.

Normes et standards de qualité logiciel

Les apports du CMM

Des résultats publiés montrent que se positionner dans la grille CMM permet d'accroître la rentabilité

Exemple:

Le département Logiciel de Hughes Air Craft, Fullerton, Californie a dépensé environ 500'000$ entre 87 et 90 pour améliorer son processus de production de logiciels

Pendant cette période, il est passés du niveau 2 au niveau 3, avec de bons espoirs d'atteindre les niveaux 4 et 5

– Augmentation de l'économie engendrée: 2 millions de $ par an

– Diminution du nombre de crises

– Diminution du nombre d'heures supplémentaires

– Amélioration de la compétence des employés

Normes et standards de qualité logiciel

Comparaison Iso 9000 – CMM

- ISO 9000 est une norme, CMM est une méthode
- CMM est dédié à l'industrie du logiciel, ISO 9000 définit un cadre pour les rapports clients fournisseurs
- CMM est plus détaillé et spécifique
- ISO 9000 établit un niveau acceptable de management de projet auquel le fournisseur doit souscrire pour que les relations client-fournisseur puissent s'établir avec certaines garanties de qualité pour le client
- CMM permet au fournisseur de s'auto-évaluer et de progresser sur une grille allant de 1 à 5.