

# Génie Logiciel

## Principes et Techniques

Pierre Gérard

Licence Pro. FC 2007/2008

### Table des matières

<b>I</b>	<b>Processus de développement logiciel</b>	<b>3</b>
<b>1</b>	<b>Motivations</b>	<b>3</b>
1.1	Qualités attendues d'un logiciel . . . . .	4
1.2	Principes du Génie Logiciel . . . . .	6
1.3	Maturité du processus de développement logiciel . . . . .	7
<b>2</b>	<b>Cycle de vie d'un logiciel</b>	<b>8</b>
2.1	Composantes du cycle de vie d'un logiciel . . . . .	8
2.2	Documents courants . . . . .	9
2.3	Modèles de cycle de vie d'un logiciel . . . . .	10
2.4	Modèles de processus logiciels . . . . .	12
<b>II</b>	<b>Conduite de projets</b>	<b>15</b>
<b>3</b>	<b>Gestion de projets</b>	<b>15</b>
3.1	Pratiques critiques de la gestion de projet . . . . .	15
3.2	Analyse de la valeur acquise . . . . .	16
3.3	Suivi des erreurs . . . . .	17
<b>4</b>	<b>Planification de projets</b>	<b>19</b>
4.1	Organigramme technique . . . . .	19
4.2	La méthode PERT . . . . .	20
4.3	Autres modèles . . . . .	23
4.4	Estimation des coûts . . . . .	23
<b>5</b>	<b>Assurance qualité</b>	<b>27</b>

<b>III</b>	<b>Techniques du Génie Logiciel</b>	<b>29</b>
<b>6</b>	<b>Métriques</b>	<b>29</b>
6.1	Métriques de Mac Cabe . . . . .	29
6.2	Métriques de Halstead . . . . .	30
6.3	Métriques de Henry-Kafura . . . . .	32
6.4	Métriques Objet de Chidamber et Kemerer . . . . .	32
6.5	Métriques MOOD . . . . .	34
<b>7</b>	<b>Analyse et gestion des risques</b>	<b>36</b>
<b>8</b>	<b>Tests logiciels</b>	<b>38</b>
8.1	Tests fonctionnels . . . . .	38
8.2	Tests structurels . . . . .	40
8.3	Test de flot de données . . . . .	42
8.4	Tests orientés objet . . . . .	43

## Première partie

# Processus de développement logiciel

## 1 Motivations

### Matériel et logiciel

- Systèmes informatiques
  - 80 % de logiciel
  - 20 % de matériel
- Depuis quelques années, la fabrication du matériel est assurée par quelques fabricants seulement
  - Le matériel est relativement fiable
  - Le marché est standardisé
- Les problèmes liés à l'informatique sont essentiellement des problèmes de **Logiciel**

### Spécificités du logiciel

- Un produit immatériel, dont l'existence est indépendante du support physique
  - Semblable à une œuvre d'art (roman, partition...)
- Un objet technique fortement contraint
  - Fonctionne ou ne fonctionne pas
  - Structure complexe
  - Relève des modes de travail du domaine technique
- Un cycle de production différent
  - La reproduction pose peu de problèmes, seule la première copie d'un logiciel a un coût
  - Production à l'unité
  - Semblable au Génie Civil (ponts, routes...)

### Un processus de fabrication original

- Le logiciel partage des propriétés contradictoires avec l'art, les technologies et le Génie Civil
- Les possibilités de réutiliser les savoir-faire des autres technologies sont (très) limitées
- Compte tenu du cycle de production, il faut bien faire tout de suite
  - « *La qualité du processus de fabrication est garante de la qualité du produit* »

### La « Crise du logiciel »

- Etude sur 8 380 projets (*Standish Group, 1995*)
  - Succès : 16 %
  - Problématique : 53 % (bujet ou délais non respectés, défaut de fonctionnalités)
  - Echec : 31 % (abandonné)
- Le taux de succès décroît avec la taille des projets et la taille des entreprises

### Le Génie Logiciel

- Conférence de l'OTAN à Garmish, Allemagne (1968)
  - L'informatique ne répond pas aux attentes qu'elle suscite

- L’informatique coûte très cher et désorganise les entreprises ou organisations
- Introduction de l’expression « Génie Logiciel » (*Software Engineering*)
  - Comment faire des logiciels de qualité ?
  - Qu’attend-on d’un logiciel ? Quels sont les critères de qualité pour un logiciel ?

## 1.1 Qualités attendues d’un logiciel

### Utilité

- Adéquation entre
  - Le besoin effectif de l’utilisateur
  - Les fonctions offertes par le logiciel
- Solutions :
  - Emphase sur l’analyse des besoins
  - Améliorer la communication (langage commun, démarche participative)
  - Travailler avec rigueur

### Utilisabilité

- « *Effectivité, efficacité et satisfaction avec laquelle des utilisateurs spécifiés accomplissent des objectifs spécifiés dans un environnement particulier* »
- Facilité d’apprentissage : comprendre ce que l’on peut faire avec le logiciel, et savoir comment le faire
- Facilité d’utilisation : importance de l’effort nécessaire pour utiliser le logiciel à des fins données
- Solutions :
  - Analyse du mode opératoire des utilisateurs
  - Adapter l’ergonomie des logiciels aux utilisateurs

### Fiabilité

- Correction, justesse, conformité : le logiciel est conforme à ses spécifications, les résultats sont ceux attendus
- Robustesse, sureté : le logiciel fonctionne raisonnablement en toutes circonstances, rien de catastrophique ne peut survenir, même en dehors des conditions d’utilisation prévues
- Mesures :
  - MTBF : Mean Time Between Failures
  - Disponibilité (pourcentage du temps pendant lequel le système est utilisable) et Taux d’erreur (nombre d’erreurs par KLOC)
- Solutions :
  - Utiliser des méthodes formelles, des langages et des méthodes de programmation de haut niveau
  - Vérifications, tests
  - Progiciels

### Interopérabilité, couplabilité

- Un logiciel doit pouvoir interagir en synergie avec d’autres logiciels
- Solutions :
  - Bases de données (découplage données/traitements)

- « Externaliser » certaines fonctions en utilisant des « Middleware » avec une API (Application Program Interface) bien définie
- Standardisation des formats de fichiers (XML...) et des protocoles de communication (CORBA...)
- Les ERP (Entreprise Resources Planning)

### **Performance**

- Les logiciels doivent satisfaire aux contraintes de temps d'exécution
- Solutions :
  - Logiciels plus simples
  - Veiller à la complexité des algorithmes
  - Machines plus performantes

### **Portabilité**

- Un même logiciel doit pouvoir fonctionner sur plusieurs machines
- Solutions :
  - Rendre le logiciel indépendant de son environnement d'exécution (voir interopérabilité)
  - Machines virtuelles

### **Réutilisabilité**

- On peut espérer des gains considérables car dans la plupart des logiciels :
  - 80 % du code est du « tout venant » qu'on retrouve à peu près partout
  - 20 % du code est spécifique
- Solutions :
  - Abstraction, généricité (ex : MCD générique de réservation)
  - Construire un logiciel à partir de composants prêts à l'emploi
  - « Design Patterns »

### **Facilité de maintenance**

- Un logiciel ne s'use pas
- Pourtant, la maintenance absorbe un très grosse partie des efforts de développement

	Répartition effort dév.	Origine des erreurs	Coût de la maintenance
Définition des besoins	6%	56%	82%
Conception	5%	27%	13%
Codage	7%	7%	1%
Intégration Tests	15%	10%	4%
Maintenance	67%		

*(Zeltovitz, De Marco)*

### **Maintenance corrective**

- Corriger les erreurs : défauts d'utilité, d'utilisabilité, de fiabilité...
- Identifier la défaillance, le fonctionnement

- Localiser la partie du code responsable
- Corriger et estimer l'impact d'une modification
- *Attention*
  - La plupart des corrections introduisent de nouvelles erreurs
  - Les coûts de correction augmentent exponentiellement avec le délai de détection
- La maintenance corrective donne lieu à de nouvelles livraisons (release)

### **Maintenance adaptative**

- Ajuster le logiciel pour qu'il continue à remplir son rôle compte tenu de l'évolution des
  - Environnements d'exécution
  - Fonctions à satisfaire
  - Conditions d'utilisation
- Ex : changement de SGBD, de machine, de taux de TVA, an 2000, euro...

### **Maintenance perfective, d'extension**

- Accroître/améliorer les possibilités du logiciel
- Ex : les services offerts, l'interface utilisateur, les performances...
- Donne lieu à de nouvelles versions

### **Facilité de maintenance**

- Objectifs
  - Réduire la quantité de maintenance corrective (zéro défaut)
  - Rendre moins coûteuses les autres maintenances
- Enjeux
  - Les coûts de maintenance se jouent très tôt dans le processus d'élaboration du logiciel
  - Au fur et à mesure de la dégradation de la structure, la maintenance devient de plus en plus difficile
- Solutions :
  - Réutilisabilité, modularité
  - Vérifier, tester
  - Structures de données complexes et algorithmes simples
  - Anticiper les changements à venir
  - Progiciels

## **1.2 Principes du Génie Logiciel**

### **Principes utilisés dans le Génie Logiciel**

- Généralisation : regroupement d'un ensemble de fonctionnalités semblables en une fonctionnalité paramétrable (généricité, héritage)
- Structuration : façon de décomposer un logiciel (utilisation d'une méthode bottom-up ou top-down)
- Abstraction : mécanisme qui permet de présenter un contexte en exprimant les éléments pertinents et en omettant ceux qui ne le sont pas
- Modularité : décomposition d'un logiciel en composants discrets
- Documentation : gestion des documents incluant leur identification, acquisition, production, stockage et distribution

- Vérification : détermination du respect des spécifications établies sur la base des besoins identifiés dans la phase précédente du cycle de vie

### **1.3 Maturité du processus de développement logiciel**

#### **Objectif CCM 6 !**

- Le Modèle de Maturité (CMM) du SEI
  1. Initial
  2. Reproductible
  3. Défini
  4. Maîtrisé
  5. Optimisé
- 75% des projets au niveau 1, 25% aux niveaux 2 et 3 selon Curtis
- Pour maîtriser le processus de développement logiciel et assurer la qualité du logiciel, il faut :
  - Séparer le développement en plusieurs étapes
  - Organiser ces étapes et modéliser le processus de développement
  - Contrôler le processus de développement

#### **Niveau de maturité 1 : Initial**

- Chaotique : plans et contrôles inefficaces
- Processus essentiellement non contrôlé, non défini
- Le succès dépend des individus

#### **Niveau de maturité 2 : Reproductible**

- Intuitif : dépend encore des individus
- Procédures de gestion utilisées, gestion des configurations et assurance qualité
- Pas de modèle formel de processus

#### **Niveau de maturité 3 : Défini**

- Qualitatif : institutionnalisé
- Procédures formelles pour vérifier que le processus est utilisé

#### **Niveaux de maturité 4 : Maîtrisé**

- Quantitatif : Processus de mesures
- Gestion quantitative de la qualité

#### **Niveaux de maturité 5 : Optimisé**

- Améliorations retournées dans le processus
- Stratégies d'amélioration du processus

## 2 Cycle de vie d'un logiciel

### Cycle de vie

*« La qualité du processus de fabrication est garante de la qualité du produit »*

- Pour obtenir un logiciel de qualité, il faut en maîtriser le processus d'élaboration
- La vie d'un logiciel est composée de différentes étapes
- La succession de ces étapes forme le cycle de vie du logiciel
- Il faut contrôler la succession de ces différentes étapes

### 2.1 Composantes du cycle de vie d'un logiciel

#### Etude de faisabilité

- Déterminer si le développement proposé vaut la peine d'être mis en œuvre, compte tenu de attentes et de la difficulté de développement
- Etude de marché : déterminer s'il existe un marché potentiel pour le produit.

#### Spécification

- Déterminer les fonctionnalités que doit posséder le logiciel
- Collecte des exigences : obtenir de l'utilisateur ses exigences pour le logiciel
- Analyse du domaine : déterminer les tâches et les structures qui se répètent dans le problème

#### Organisation du projet

- Déterminer comment on va développer le logiciel
- Analyse des coûts : établir une estimation du prix du projet
- Planification : établir un calendrier de développement
- Assurance qualité du logiciel : déterminer les actions qui permettront de s'assurer de la qualité du produit fini
- Répartition des tâches : hiérarchiser les tâches et sous-tâches nécessaires au développement du logiciel

#### Conception

- Déterminer la façon dont le logiciel fournit les différentes fonctionnalités recherchées
- Conception générale
  - Conception architecturale : déterminer la structure du système
  - Conception des interfaces : déterminer la façon dont les différentes parties du système agissent entre elles
- Conception détaillée : déterminer les algorithmes pour les différentes parties du système

#### Implémentation

- Ecrire le logiciel



### Tests

- Essayer le logiciel sur des données d'exemple pour s'assurer qu'il fonctionne correctement
- Tests unitaires : faire tester les parties du logiciel par leurs développeurs
- Tests d'intégration : tester pendant l'intégration
- Tests de validation : pour acceptation par l'acheteur
- *Tests système* : tester dans un environnement proche de l'environnement de production
- *Tests Alpha* : faire tester par le client sur le site de développement
- *Tests Bêta* : faire tester par le client sur le site de production
- *Tests de régression* : enregistrer les résultats des tests et les comparer à ceux des anciennes versions pour vérifier si la nouvelle n'en a pas dégradé d'autres

### Livraison

- Fournir au client une solution logicielle qui fonctionne correctement
- Installation : rendre le logiciel opérationnel sur le site du client
- Formation : enseigner aux utilisateurs à se servir du logiciel
- Assistance : répondre aux questions des utilisateurs

### Maintenance

- Mettre à jour et améliorer le logiciel pour assurer sa pérenité
- Pour limiter le temps et les coûts de maintenance, il faut porter ses efforts sur les étapes antérieures

## 2.2 Documents courants

### Cahier des charges

- Description initiale des fonctionnalités désirées, généralement écrite par l'utilisateur

### Spécifications

- Décrit précisément les conditions que doit remplir le logiciel
- Modèle objet : indique les classes et les documents principaux
- Scénarios des cas d'utilisation : indique les différents enchaînements possibles du point de vue de l'utilisateur

### Calendrier du projet

- Ordre des différentes tâches
- Détails et ressources qu'elles demandent

### Plan de test du logiciel

- Décrit les procédures de tests appliquées au logiciel pour contrôler son bon fonctionnement
- Tests de validation : tests choisis par le client pour déterminer s'il peut accepter le logiciel

### Plan d'assurance qualité

- Décrit les activités mises en œuvre pour garantir la qualité du logiciel

**Manuel utilisateur**

- Mode d’emploi pour le logiciel dans sa version finale

**Code source**

- Code complet du produit fini

**Rapport des tests**

- Décrit les tests effectués et les réactions du système

**Rapport des défauts**

- Décrit les comportements du système qui n’ont pas satisfait le client
- Il s’agit le plus souvent de défaillances du logiciel ou d’erreurs

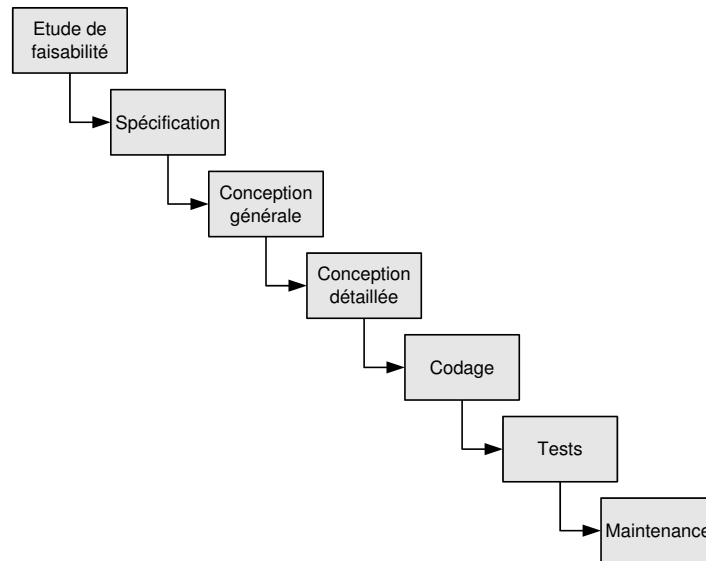
**Documents produits dans le cycle de vie**

<i>Document</i>	<i>Phase de production</i>
Manuel utilisateur final	Implémentation
Conception architecturale	Conception
Plan d’assurance qualité	Planification
Code source	Implémentation
Cahier des charges	Faisabilité
Plan de test	Spécification
Manuel utilisateur préliminaire	Spécification
Conception détaillée	Conception
Estimation des coûts	Planification
Calendrier du projet	Planification
Rapport des tests	Tests
Documentation	Implémentation

**2.3 Modèles de cycle de vie d’un logiciel****Modèles linéaires et incrémentaux**

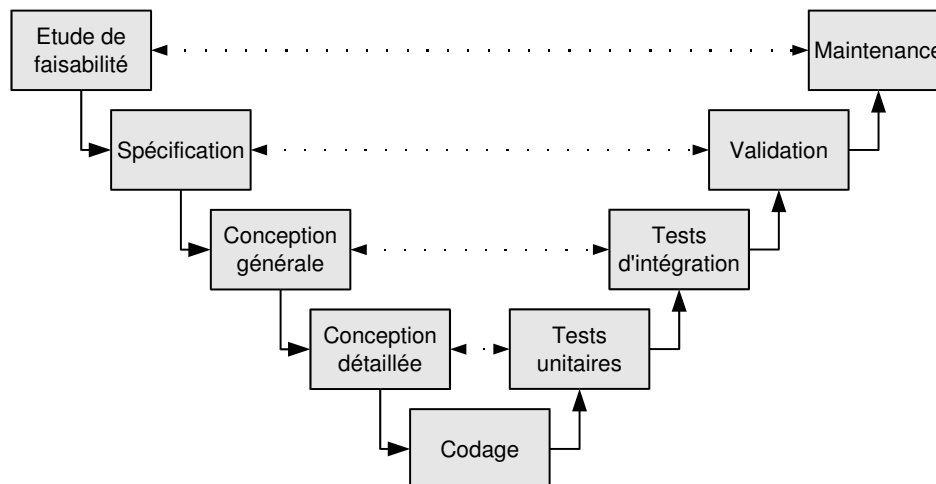
- Modèles linéaires
  - cascade
  - modèle en V
  - ...
- Modèles non linéaires
  - prototypage
  - modèles incrémentaux
  - modèle en spirale
  - ...

**Le cycle de vie en « Cascade »**



- Adapté pour des projets de petite taille, et dont le domaine est bien maîtrisé

### Le cycle de vie en « V »



- Adapté pour des projets dont le domaine est bien maîtrisé

### Le prototypage

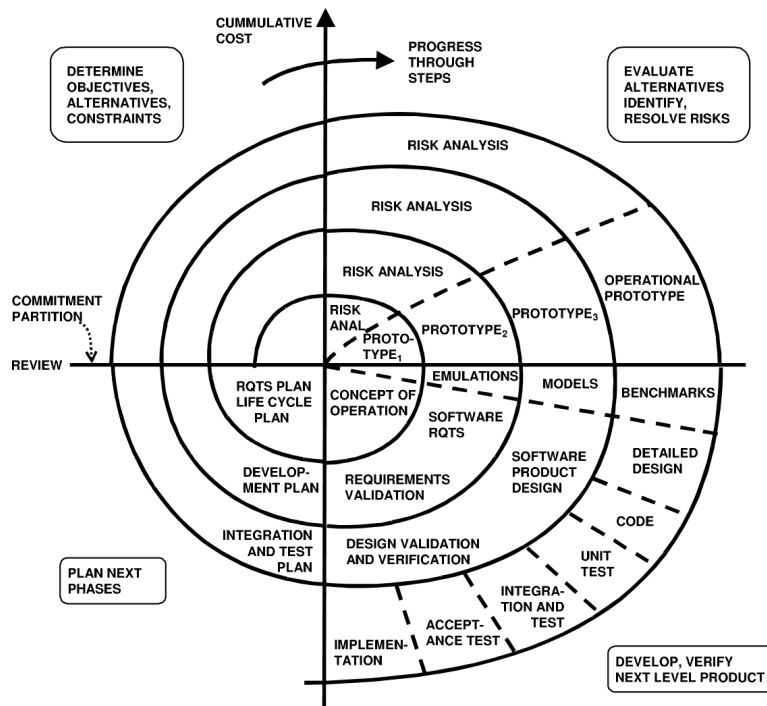
- Prototype : version d'essai du logiciel
  - Pour tester les différents concepts et exigences
  - Pour montrer aux clients les fonctions que l'on veut mettre en œuvre
- Lorsque le client a donné son accord, le développement suit souvent un cycle de vie linéaire
- Avantages : Les efforts consacrés au développement d'un prototype sont le plus souvent compensés par ceux gagnés à ne pas développer de fonctions inutiles

## Le modèle incrémental de Parnas

1. Concevoir et livrer au client un sous-ensemble minimal et fonctionnel du système
2. Procéder par ajouts d'incrémentaux minimaux jusqu'à la fin du processus de développement
3. Avantages : meilleure intégration du client dans la boucle, produit conforme à ses attentes

## Le modèle en Spirale de Boehm

- Un modèle mixte
- A chaque cycle, recommencer :
  1. Consultation du client
  2. Analyse des risques
  3. Conception
  4. Implémentation
  5. Tests
  6. Planification du prochain cycle



- Avantages : meilleure maîtrise des risques, mais nécessite une (très) grande expérience

## 2.4 Modèles de processus logiciels

# Modélisation des processus logiciels

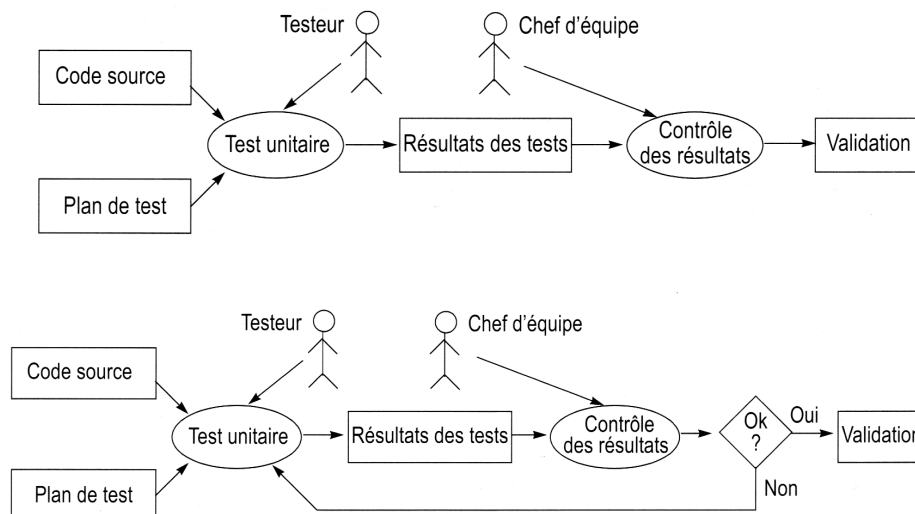
- Les modèles en V, spirale... sont des archétypes de modèles
- On peut les adapter à des projets particuliers

- On peut vouloir représenter le processus de développement (et ses parties) de manière plus fine
  - Une modélisation précise limite les risques d’ambiguïté
- Il est nécessaire de se doter d’un formalisme pour représenter le processus de développement
  - Pour ordonner les tâches
  - Pour déterminer les échanges d’information entre les différentes tâches
  - Pour permettre aux jeunes recrues de mieux travailler

### Modèle de processus logiciels

- Un modèle de processus logiciels décrit
  - Les tâches
  - Les artefacts (fichiers, documents, données...)
  - Les auteurs
  - Les décisions (facultatif)
- Règles à observer
  - Deux tâches doivent être séparées par un artefact
  - Une tâche ne peut être exécutée tant que ses artefacts d’entrée n’existent pas
  - Il doit y avoir au moins une tâche de début et une de fin
  - Il doit y avoir un trajet depuis chaque tâche jusqu’à la tâche de fin

### Exemples de processus logiciels

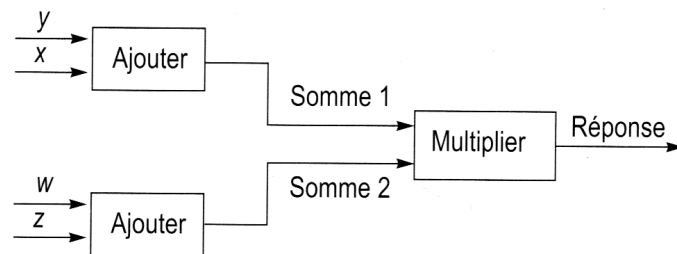
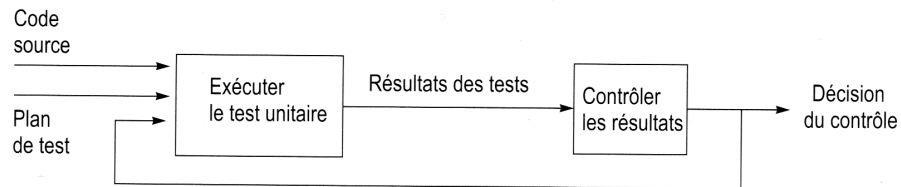


### Diagrammes de flots de données

- Indique la circulation des données à travers un ensemble de composants qui peuvent être
  - des tâches
  - des composants logiciels...
- Règles à observer
  - Les processus sont représentés par des cases qui contiennent des phrases verbales

- Les flèches représentent des données et doivent être accompagnées de phrases nominales
- Un processus peut être une activité ponctuelle ou continue
- Deux flèches sortant d'une case peuvent indiquer
  - Soit deux sorties simultanées
  - Soit deux sorties exclusives

### Exemples de diagramme de flots de données



pour représenter  $(x + y) \times (w + z)$

## Deuxième partie

# Conduite de projets

### 3 Gestion de projets

#### Gestion de projets

- Problèmes souvent humains
  - Planifier la progression
  - Motiver et coordonner un groupe de professionnels
- Techniques souvent communes à la gestion de projet en général
- Problème particulier de la visibilité
  - Un projet logiciel apparaîtra souvent à ses développeurs comme presque achevé alors qu'il ne l'est qu'à 90%

#### 3.1 Pratiques critiques de la gestion de projet

##### Pratiques du chef de projet

- Opter pour une gestion des risques continue
- Prévoir des étapes réparties sur l'ensemble du cycle de vie consacrées à l'identification et à l'évaluation des risques, ainsi que des tâches pour y remédier
- Estimer les coûts et planifier le projet à partir de données empiriques
  - Prévoir une étape au début du cycle de vie pour évaluer le coût du projet et une série d'étapes ultérieures pour raffiner cette estimation. Au cours de chaque étape, les données devront être archivées pour les évaluations ultérieures
- Utiliser des métriques pour la gestion du projet
  - Choisir des métriques et prévoir des étapes pour enregistrer les valeurs de celles-ci, et d'autres pour analyser les progrès en fonction de ces résultats
- Suivre l'évolution de la valeur acquise
- Rechercher les défauts en fonction des objectifs de qualité
  - Déterminer des objectifs pour le nombre de rapports d'erreurs et prévoir des étapes pour communiquer ces résultats
- Considérer les employés comme la ressource la plus importante
  - Contrôler l'ensemble du processus logiciel pour estimer son impact sur le programmeur
- Utiliser un outil de gestion de configuration
  - Assurer que les modifications du logiciel sont effectuées de manière à minimiser les coûts globaux
  - Garder la trace des différences entre les versions pour contrôler les nouvelles versions
- Gérer et suivre l'évolution des besoins
  - Prévoir des étapes pour recueillir les besoins des utilisateurs
- Orienter la conception en fonction du système visé
- Définir et contrôler les interfaces
- Concevoir plusieurs fois pour ne coder qu'une seule
  - Prévoir des étapes pour contrôler la conception
- Identifier les éléments potentiellement réutilisables
- Contrôler les spécifications
- Organiser les tests comme un processus continu

- Prévoir des étapes de tests dans toutes les phases

### 3.2 Analyse de la valeur acquise

#### Mesures de base

- CBT : coût budgété du travail
- Quantité de travail estimée pour une tâche donnée
- CBTP : coût budgété du travail prévu
  - Somme des quantités de travail estimées pour l'ensemble des tâches devant être achevées à une date donnée
- CBA : coût budgété à l'achèvement
  - Total des CBTP et donc l'estimation de la quantité de travail pour le projet entier
- VP : valeur prévue
  - Proportion de la quantité de travail totale estimée attribuée à une tâche donnée
  - $VP = CBT/CBA$

#### Mesures de base

- CBTE : coût budgété du travail effectué
- Somme des quantités de travail estimées pour les tâches achevées à une date donnée
- CRTE : coût réel du travail effectué
  - Somme des quantités de travail réelles pour l'ensemble des tâches du projet

#### Indicateurs d'avancement

- VA : Valeur acquise
  - $VA = CBTE/CBA$
  - = somme des VP pour les tâches achevées
  - = PA (pourcentage achevé)
- IPT : indicateur de performance temporel
  - $IPT = CBTE/CBTP$
- VE : variance par rapport à l'échéancier
  - $VE = CBTE - CBTP$
- IC : indicateur d'écart sur les coûts
  - $IC = CBTE/CRTE$
- VC : variance par rapport aux coûts
  - $VC = CBTE - CRTE$

#### Exemple 1

Tâche	Trav. estimé (jh)	Trav. réel aujourd'hui	Date d'ach. estimée	Date d'ach. effective
1	5	10	25/01	01/02
2	25	20	15/02	15/02
3	120	80	15/05	
4	40	50	15/04	15/04
5	60	50	01/07	
6	80	70	01/09	



- Problème : calculer les indicateurs d'avancement au 01/04
- CBA : somme des estimations des quantités de travail
- $CBA = 330jh$
- Au 01/04, les tâches 1,2 et 4 sont achevées
- Le CBTE est la somme des CBT pour ces tâches
- $CBTE = 70jh$
- $VA = 70/330 = 21.2\%$
- Les tâches 1 et 2 devraient être achevées pour le 01/04, et pas 1,2 et 4
- $CBTP = 30$
- $IPT = 70/30 = 233\%$
- $SV = 70 - 30 = +40jh$
- La CRTE est la somme des quantités de travail réelles pour les tâches 1,2 et 4
- $CRTE = 80jh$
- $IC = 70/80 = 87.5\%$
- $VC = 70 - 80 = -10jh$

### Exemple 2

- Problème : que deviennent ces indicateurs, à supposer que la tâche 3 a également été achevée avec 140jh de travail et que nous sommes le 01/07 ?

Tâche	Trav. estimé (jh)	Trav. réel aujourd'hui	Date d'ach. estimée	Date d'ach. effective
1	5	10	25/01	01/02
2	25	20	15/02	15/02
3	120	140	15/05	01/07
4	40	50	15/04	15/04
5	60	50	01/07	
6	80	70	01/09	

- $CBTE = 190jh$
- $CBTP = 250jh$
- $CRTE = 220jh$
- $VA = 190/330 = 57.5\%$
- $IPT = 190/250 = 76\%$
- $VE = 190 - 250 = -60jh$
- Seules les tâches 1-4 sont réalisées, au lieu de 1-5
- $IC = 190/220 = 86.6\%$
- $VC = 190 - 220 = -30jh$

## 3.3 Suivi des erreurs

### Suivi des erreurs

- Conserver une trace des
  - Erreurs qui se sont produites
  - Durées entre deux erreurs successives
- Permet de
  - Mieux déterminer une date de livraison
  - Motiver les testeurs et les développeurs

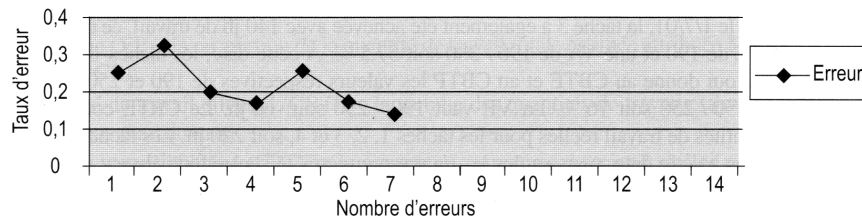
### Taux d'erreur

- TE : Inverse du temps qui sépare deux erreurs successives
  - Ex : si une erreur se produit tous les deux jours,  $TE = 0.5$  erreurs par jour
- Taux d'erreur instantané : estimation du taux d'erreur courant
- Taux d'erreur cumulé : bonne estimation des taux d'erreur à venir
  - Somme de toutes les erreurs divisé par le temps total
- Une représentation graphique permet d'estimer les tendances des taux d'erreur par régression linéaire
  - y : taux d'erreurs
  - x : deux possibilités :
    - nombre d'erreurs : donne une estimation du nombre d'erreurs restantes
    - temps écoulé : donne une estimation du temps avant livraison

### Exemple

- Les durées entre une série d'erreurs sont les suivantes : 4, 3, 5, 6, 4, 6, 7.

Durée	4	3	5	6	4	6	7
Taux	0.25	0.33	0.20	0.17	0.25	0.17	0.14



- Si on prolonge cette courbe, on voit qu'elle coupe l'axe des abscisses pour une valeur de 11 erreurs environ
- Le taux d'erreur devrait donc être nul après le 11ème erreur
- Comme 7 erreurs ont été trouvées, il ne devrait plus en rester que 4

## 4 Planification de projets

### Planification

- Élément indispensable de la conduite de projets
- Déterminer les tâches à accomplir
- Déterminer l'ordre des tâches
- Décider des ressources allouées à chaque tâche

### 4.1 Organigramme technique

#### Work Breakdown Structure (WBS)

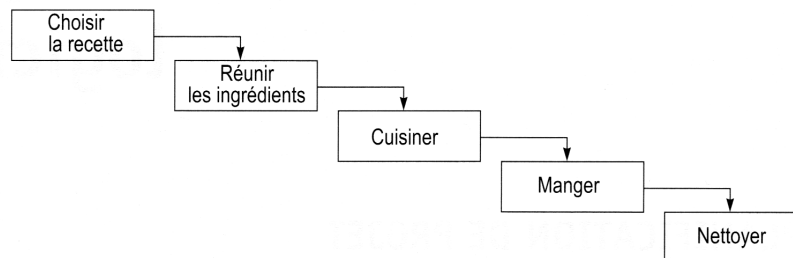
- Objectif : Diviser les tâches principales en tâches plus petites
- Nécessite de :
  - Pouvoir identifier leurs différentes parties
  - Trouver des livrables et des jalons qui permettront de mesurer l'avancement du projet
- WBS *Work Breakdown Tructure* : organigramme technique
  - Structure arborescente
  - Le premier niveau de décomposition correspond souvent au modèle de cycle de vie adopté

#### Règles d'obtention d'un organigramme technique (WBS)

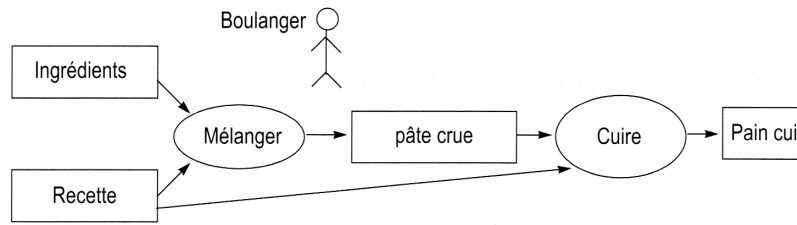
- Structure arborescente
- Pas de boucle
- Les actions itératives apparaissent dans le modèle de processus ou dans le modèle de cycle de vie
- Les descriptions des tâches et des livrables doivent être claires et sans ambiguïté
  - Chaque livrable doit être associé à une tâche, sans quoi il ne sera pas produit
- Chaque tâche doit avoir un critère d'achèvement
  - Le plus souvent un livrable
- L'achèvement de toutes les sous-tâches doit entraîner l'achèvement de la tâche

#### Exemple

- Modèle de cycle de vie pour la consommation de tartines



- Modèle de processus pour cuisiner du pain

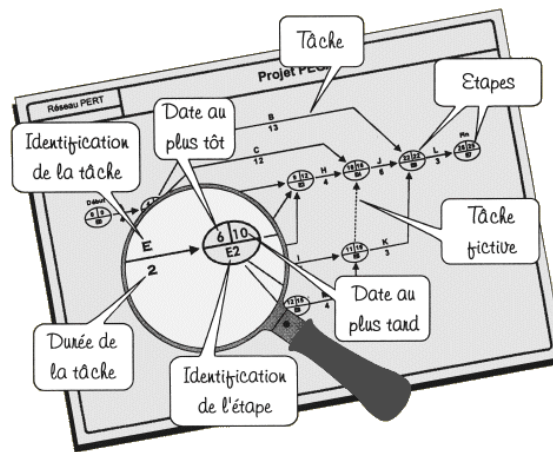


- Choisir la recette
  - Choisir les ingrédients → *Liste des ingrédients*
  - Vérifier leur disponibilité → *Liste de courses*
- Réunir les ingrédients → *Ingrédients réunis*
- Cuisiner
  - Mélanger
    - Ajouter l'eau → *Saladier d'eau*
    - Ajouter la levure et la farine → *Mélange*
    - Faire la pâte → *Pâte*
    - Laisser lever 1 → *Pâte levée*
    - Ajouter le reste de farine et pétrir → *Pâte pétrie*
    - Laisser lever 2 → *Pâte pétrie et levée*
    - Former des miches → *Miches*
    - Laisser lever 3 → *Miches levées*
  - Cuire → *Pain cuit*
- Manger
  - Découper en tranches → *Tranches de pain*
  - Beurrer → *Tartines beurrées*
  - Manger → *Goût satisfaisant*
- Nettoyer
  - Nettoyer ustensiles → *Ustensiles propres*
  - Nettoyer cuisine → *Cuisine propre*

## 4.2 La méthode PERT

### PERT

- Program Evaluation and Review Technique
  1. Identifier les tâches et estimer leur durée
  2. Ordonner les tâches
  3. Construire le réseau et l'exploiter



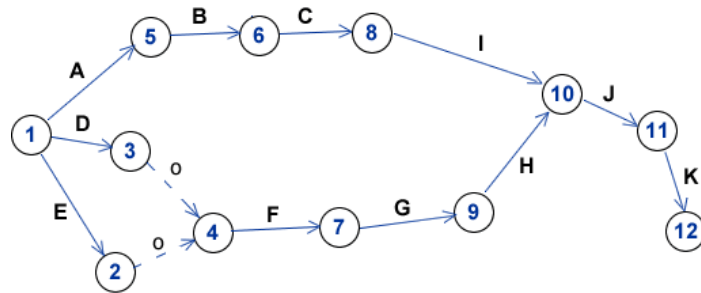
### Identification des tâches et de leur durée

Tâche	Durée	Référence
Mettre la farine dans un saladier	3 s	A
Mettre deux oeuf	30 s	B
Ajouter le lait doucement et mélanger	600 s	C
Dans une poêle mettre du rhum	3 s	D
Couper les bananes en fines lamelles	300 s	E
Les mélanger au rhum	30 s	F
Faire chauffer le mélange	120 s	G
Faire flamber	10 s	H
Faire cuire une crêpe	10 s	I
Verser du mélange bananes-rhum sur la crêpe	10 s	J
Manger	120	K

### Ordonnancement des tâches

Pour faire	Il faut faire
A	-
B	A
C	B
D	-
E	-
F	D-E
G	F
H	G
I	C
J	I-H
K	J

### Construction du réseau



- Les tâches A, D et E peuvent se faire en parallèle
- Il faut qu’elles soient toutes deux finies pour pouvoir débiter F, d’où les tâches fictives de durée nulle de 3 à 4 et de 2 à 4.
- La crêpe *doit* être cuite (I) et le mélange *doit* être flambé (H) pour pouvoir commencer à verser du mélange sur la crêpe (J)

### Exploitation d’un réseau PERT

1. Calcul des dates au plus tôt de chaque étape
  - Quand se terminera le projet ? Quel est le délai nécessaire pour atteindre une étape déterminée ?
  - *Elles se calculent de gauche à droite en partant de 0 et rajoutant la durée de la tâche à la date précédente. En cas de convergence, on prend la valeur la plus élevée*
2. Calcul des dates au plus tard de chaque étape
  - Quand doit démarrer le projet pour être achevé à temps ? A quelle date chaque étape doit-elle être atteinte pour que le projet ne prenne pas de retard ?
  - *Elles se calculent de droite à gauche en partant de la date de fin au plus tard et en retranchant la durée de la tâche à la date précédente. En cas de convergence comme au 7 niveau on prend la valeur la plus faible*
3. Calcul des marges et du chemin critique

### Calcul des dates et des marges

Etape	Date au plus tôt	Date au plus tard	Marge
1	0	0	0
2	300	483	183
3	3	483	480
4	300	483	183
5	3	3	0
6	33	33	0
7	330	513	183
8	633	633	0
9	450	633	183
10	643	643	0
11	653	653	0
12	773	773	0

### Chemin critique

- On peut des maintenant répondre à certaines questions
- La crêpe sera mangée a la date 773 (soit a peu près 13 mn après le début de la fabrication)
- Il faut avoir fini la sauce au plus tard a la date 643 (étape 10)
- On pourra mélanger le rhum et les bananes au plus tôt à la date 300 (étape 4)
- On remarque aussi que certaines étapes ont une marge nulle et que pour d'autres on est plus libre
- Le chemin critique relie les étapes qui n'ont aucune marge
- Un retard dans une tâche sur le chemin critique entraînera un retard dans le projet entier
- L'analyse des temps de batements peut motiver une réaffectation des ressources associées à chaque tâche

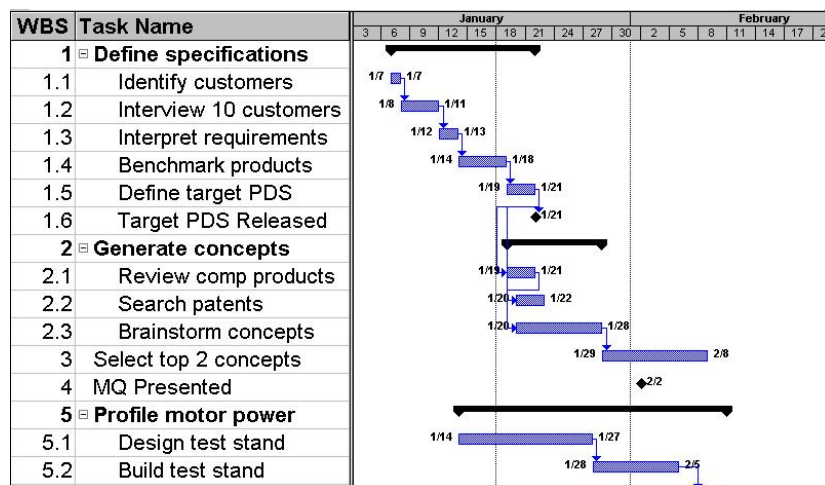
### 4.3 Autres modèles

#### Extensions des réseaux PERT

- PERT Charge pour prendre en compte les ressources affectées au projet
- Ressource : moyen nécessaire au déroulement et à l'aboutissement d'une tâche
- Les tâches sont caractérisées par des durées et des intensités de ressources
- PERT Cost pour gérer les coûts
- Permet d'optimiser l'échéancier des paiements en
  - Jouant sur les surcoûts affectant les tâches critiques
  - Jouant sur les économies possibles sur les tâches non critiques

#### Diagrammes de Gantt

- Utilise les dates au plus tôt et des dates au plus tard
- Permet d'établir un planning en fonction des ressources



### 4.4 Estimation des coûts

#### Le modèle COCOMO de base

- CONstructive COSt Model

- Développé à la firme TRW (organisme du DoD, USA) par B.W. Boehm et son équipe
- Fondé sur une base de données de plus de 60 projets différents
- Modèle d'estimation
  - du coût de développement d'un logiciel en nombre de mois-hommes ( $E$  : effort)
  - du temps de développement en mois ( $TDEV$ )
  - en fonction du nombre de lignes de codes en milliers ( $KLOC$ )

### Trois types de projets

- Mode organique
  - Petites équipes
  - Applications maîtrisées et problèmes bien compris
  - Pas de besoins non fonctionnels difficiles
- Mode semi-détaché
  - Expérience variée des membres de l'équipe de projet
  - Possibilité l'avoir des contraintes non fonctionnelles importantes
  - Type l'application non maîtrisée par l'organisation
- Mode embarqué
  - Contraintes serrées
  - L'équipe de projet a, en général, peu l'expérience de l'application
  - Problèmes complexes

### Calcul de l'effort

- Formule générale
  - $E = a \times KLOC^b$
  - $a$  et  $b$  estimés en fonctions de données empiriques
- Organique
  - $E = 2.4 \times KLOC^{1.05}$
- Semi-détaché
  - $E = 3.0 \times KLOC^{1.12}$
- Embarqué
  - $E = 3.6 \times KLOC^{1.20}$

### Calcul du temps de développement

- Formule générale
  - $TDEV = a \times KLOC^b$
  - $a$  et  $b$  estimés en fonctions de données empiriques
- Organique
  - $TDEV = 2.5 \times KLOC^{0.38}$
- Semi-détaché
  - $TDEV = 2.5 \times KLOC^{0.35}$
- Embarqué
  - $TDEV = 2.5 \times KLOC^{0.32}$

### Modèle COCOMO intermédiaire

- Estimation modifiant l'estimation brute fournie par le modèle COCOMO de base en se servant des attributs



- Logiciel
- Matériel
- Projet
- Personnel

### **Les attributs du logiciel**

- Besoin en fiabilité
- Taille de la Base de Données
- Complexité du produit

### **Les attributs du matériel**

- Contraintes sur le temps l'exécution
- Contraintes sur la mémoire
- Contraintes sur le stockage
- Contraintes du temps de passage entre deux processus (synchronisation)
- ...

### **Les attributs du projet**

- Techniques de programmation moderne
  - Programmation Orientée Objet
  - Programmation Événementielle
- Utilisation l'Ateliers de Génie Logiciel (CASE)
- Contraintes de développement
  - Délais
  - Budget
- ...

### **Les attributs du personnel**

- Compétence de l'analyste
- Compétence du programmeur
- Expérience dans l'utilisation du langage de programmation
- Expérience dans le domaine de l'application
- Expérience dans l'utilisation du matériel

### **Calcul de l'effort**

- Organique
  - $E = 3.2 \times KLOC^{1.05}$
- Semi-détaché
  - $E = 3.0 \times KLOC^{1.12}$
- Embarqué
  - $E = 2.8 \times KLOC^{1.20}$
- Les estimations obtenues par la formule ci-dessus sont multipliées par les 15 facteurs de coût liées aux attributs du logiciel, du matériel, du projet et du personnel

### Modèle COCOMO expert

- Inclue toutes les caractéristiques du modèle intermédiaire
- Ajouts :
  - L'impact de la conduite des coûts sur chaque étape du cycle de développement
  - Le projet est analysé comme une hiérarchie : module, sous-système et système
- COCOMO expert permet une véritable gestion de projet
  - Utile pour de grands projets
  - Ajustement des paramètres possibles en fonction de données locales protant sur les habitudes de développement
  - Problème : nécessite une estimation de la taille du projet en KLOC

### Analyse en points de fonction

- Plutôt que d'estimer le nombre de lignes de code, il peut être plus judicieux d'estimer des points de fonction
- Les éléments les plus courants à prendre en compte sont les :
  - Interrogations : paires requête-réponse
  - Entrées : les champs individuels ne sont généralement pas comptés séparément (nom, prénom... comptent pour 1)
  - Sorties (comme les entrées)
  - Fichiers internes : fichiers tels que le client les comprend
  - Interfaces externes : données partagées avec d'autres programmes

### Comptage des points de fonction

- Des coefficients sont attribués aux éléments, selon leur complexité

Eléments	Simple	Moyens	Complexes
Sorties	4	5	7
Interrogations	3	4	6
Entrées	3	4	6
Fichiers	7	10	15
Interfaces	5	7	10

- Les coefficients pondèrent une somme du nombre d'éléments recensés pour obtenir les points de fonction du logiciel
  - Manque de standard pour compter les PF
  - Estimation des coefficients à faire en interne
  - Relation entre points de fonction et coût à estimer en interne

## 5 Assurance qualité

### Qualité

- Difficile à définir
- ISO 8402 : « *l'ensemble des caractéristiques d'une entité qui lui confèrent l'aptitude à satisfaire des besoins exprimés et implicites* »
- Un logiciel est de qualité lorsqu'il fonctionne comme il est supposé le faire
- Il est plus facile de mesurer les défauts de qualité
  - Mécontentement du client
  - Nombre de rapports d'erreurs

### Inspections formelles

- Activité formelle et planifiée
- Un concepteur présente des documents sur un projet à un groupe d'autres concepteurs qui en évaluent les aspects techniques avec pour objectif de trouver les erreurs
- Contrôle effectué par des personnes techniquement compétentes
- Participation active de l'auteur
- Porte sur un produit fini
- Inspection périodique au cours du processus de développement

### Rôles pour une inspection

- Le modérateur :
  - Il choisit l'équipe
  - Il dirige l'inspection
- Le lecteur :
  - Il n'est généralement pas l'auteur du produit
  - Il guide l'équipe dans la structure du produit
- Le secrétaire :
  - Il consigne le déroulement de l'inspection
  - Il note toutes les erreurs trouvées
- L'auteur :
  - Il est à l'origine du produit examiné
  - Il répond aux questions
  - Il corrige les erreurs et fait un rapport au modérateur

### Etapas de l'inspection

- Présentation générale par l'auteur au reste de l'équipe
- Préparation
  - Les membres de l'équipe étudient le produit dans la limite d'un temps calculé en fonction du nombre de LOC
  - Ils peuvent s'aider d'une liste de contrôles
- Réunion pour l'inspection
  - Organisée par le modérateur
  - Le lecteur conduit l'inspection
  - Le secrétaire consigne les problèmes dans un rapport
  - En cas de désaccord, il est possible de produire des rapport individuels

- Intégration des remarques : l’auteur corrige les erreurs
- Suivi
  - Le modérateur contrôle le rapport et les corrections
  - Si les critères sont satisfaits, l’inspection prend fin

## Troisième partie

# Techniques du Génie Logiciel

## 6 Métriques

### 6.1 Métriques de Mac Cabe

#### Complexité structurelle selon Mc Cabe

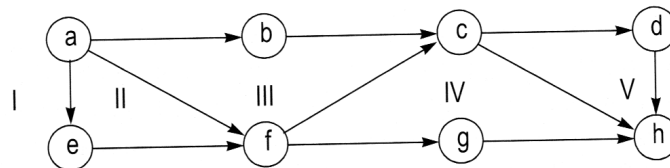
- Métrique la plus utilisée après les lignes de code
- Met en évidence la complexité structurelle du code
  - On produit un graphe de contrôle qui représente un code
  - Le nombre de faces du graphe donne la complexité structurelle du code

#### Nombre cyclomatique de Mc Cabe

$$C = a - n + 2p$$

avec

- $a$  = nombre d'arcs du graphe de contrôle
- $n$  = nombre de nœuds du graphe de contrôle
- $p$  = nombre de composantes connexes (1 le plus souvent)



- Ici,  $n = 8$ ,  $a = 11$  et  $p = 1$  donc
- $C = 11 - 8 + 2 = 5$
- $C$  correspond au nombre de faces (en comptant la face extérieure)

#### Calcul direct du nombre de Mc Cabe

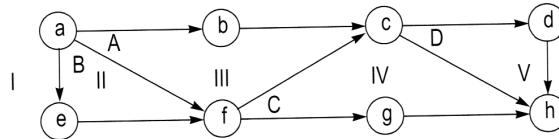
- Produire un graphe de contrôle et l'analyser peut s'avérer long dans le cas de programmes complexes
- Mc Cabe a introduit une nouvelle manière de calculer la complexité structurelle

$$C = \pi + 1$$

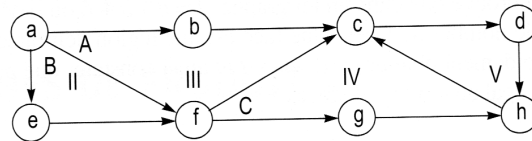
avec  $\pi$  le nombre de décisions du code

- 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

## Nombre de faces et formule de Mc Cabe



- 4 décisions donc  $C = 5 + 1 = 5$



- 3 décisions donc  $C = 3 + 1 = 4$
- Pourtant, même nombre de faces : la formule de Mc Cabe serait incorrecte ? *Non*
- Le second graphe est invalide parce qu'il ne possède pas de noeud d'arrivée

## 6.2 Métriques de Halstead

### Science informatique de Halstead

- Métriques pour la complexité d'un programme
- Fondées empiriquement
- Toujours considérées comme valides, contrairement à ses formules complexes de prédiction
- Entités de base
  - Opérandes : jetons qui contiennent une valeur
  - Opérateurs : tout le reste (virgules, parenthèses, opérateurs arithmétiques...)

### Mesures de base $\eta_1$ et $\eta_2$ (êta)

- $\eta_1$  : nombre d'opérateurs distincts
- $\eta_2$  : nombre d'opérandes distincts
- $\eta = \eta_1 + \eta_2$  : nombre total de jetons distincts
- Opérandes potentiels  $\eta_2^*$ 
  - Ensemble de valeurs minimal pour n'importe quelle implémentation
  - Pour comparer différentes implémentations du même algorithme
  - S'obtient généralement en comptant les valeurs qui ne sont pas initialisées à l'intérieur de l'algorithme
    - Valeurs lues par le programme
    - Valeurs passées en paramètres
    - Valeurs globales appelées depuis l'algorithme

### Exemple

```

z = 0;
while x > 0
    z = z + y;
    
```

```

    x = x - 1;
end-while
print (z);

```

- Opérateurs : = ; while/end-while > + - print ()  $\eta_1 = 8$
- Opérandes : = z 0 x y 1  $\eta_2 = 5$

### Longueur d'un programme

- $N_1$  : Nombre total d'opérateurs
- $N_2$  : Nombre total d'opérandes
- $N = N_1 + N_2$  : Nombre total de jetons

### Exemple

```

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		

$$N_1 = 14, N_2 = 12 \text{ donc } N = 26$$

### Estimation de la longueur

- Estimation de  $N$  à partir de  $\eta_1$  et de  $\eta_2$

$$N^{est} = \eta_1 + \log_2(\eta_1) + \eta_2 \times \log_2(\eta_2)$$

- Dans notre exemple :  $N^{est} = 8 \times \log_2(8) + 5 \times \log_2(5) = 8 \times 3 + 5 \times 2.32 = 35.6$
- Ici,  $N^{est} \gg N$
- En pratique, *Si la différence entre  $N$  et  $N^{est}$  est supérieure à 30%, il vaut mieux renoncer à utiliser les autres mesures de Halstead*

### Volume

- Estimation du nombre de bits nécessaires pour coder le programme mesuré

$$V = N \times \log_2(\eta_1 + \eta_2)$$

- Dans notre exemple :  $V = 26 \times \log_2(13) = 26 \times 3.7 = 96,2$

### Autres mesures de Halstead

- Volume potentiel  $V^*$
- Taille minimale d’une solution au problème
- $V^* = (2 + \eta_2^*) \times \log_2(2 + \eta_2^*)$
- Niveau d’implémentation  $L$
- Exprime dans quelle mesure l’implémentation actuelle est proche de l’implémentation minimale
- $L = V^*/V$
- Mesures présentées pour leur intérêt historique
- Effort  $E$  en *emd* (elementary mental discrimination)
- Mesure de l’effort intellectuel nécessaire à l’implémentation d’un algorithme
- $E = V/L$
- Temps  $T$
- Temps nécessaire pour implémenter un algorithme
- $T = E/S$  où  $S$  est le nombre de Stroud. Halstead avait retenu  $S = 18$

## 6.3 Métriques de Henry-Kafura

### Flux d’informations d’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 noté  $poids_i$  calculé en fonction de son nombre de LOC et de sa complexité

$$HK_i = poids_i \times (out_i \times in_i)^2$$

- La mesure totale  $HK$  correspond à la somme des  $HK_i$

### Exemple

- 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

$$HK = 1110$$

## 6.4 Métriques Objet de Chidamber et Kemerer

### Métriques Objet de Chidamber

- Ensemble de métriques (Metric Suite for Object Oriented Design)
- Evaluation des classes d’un système
- La plupart des métriques sont calculées classe par classe
- Le passage au global n’est pas clair, une moyenne n’étant pas très satisfaisante



**M1 : Méthodes pondérées par classe**

- WMC : Weighted Methods per Class

$$WMC = \frac{1}{n} \times \sum_{i=0}^n c_i \times M_i$$

avec  $C$  un ensemble de  $n$  classes comportant chacune  $M_i$  méthodes dont la complexité (le poids) est noté  $c_i$

**M2 : Profondeur de l'arbre d'héritage**

- DIC : Depth of Inheritance Tree
- Distance maximale entre un nœud et la racine de l'arbre d'héritage de la classe concernée
- Calculée pour chaque classe

**M3 : Nombre d'enfants**

- NOC : Number Of Children
- Nombre de sous-classes dépendant immédiatement d'une classe donnée, par une relation d'héritage
- Calculée pour chaque classe

**M4 : Couplage entre classes**

- Dans un contexte OO, le couplage est l'utilisation de méthodes ou d'attributs d'une autre classe.
- Deux classes sont couplées si les méthodes déclarées dans l'une utilisent des méthodes ou instancient des variables définies dans l'autre
- Le relation est symétrique : si la classe A est couplée à B, alors B l'est à A
- CBO : Coupling Between Object classes
- Pour chaque classe, nombre de classes couplées
- Calculée pour chaque classe

**M5 : Réponses pour une classe (RFC)**

- $\{RS\}$  : ensemble des méthodes qui peuvent être exécutées en réponse à un message reçu par un objet de la classe considérée
- Réunion de toutes les méthodes de la classe avec toutes les méthodes appelées directement par celles-ci
- Calculée pour chaque classe

$$RFG = |RS|$$

**M6 : Manque de cohésion des méthodes**

- Un module (ou une classe) est « cohésif » lorsque tous ses éléments sont étroitement liés
- LCOM (Lack of COhesion in Methods) tente de mesurer l'absence de ce facteur
- Posons
- $I_i$  l'ensemble des variables d'instance utilisées par la méthode  $i$
- $P$  l'ensemble des paires de  $(I_i, I_j)$  ayant une intersection vide

- $Q$  l'ensemble des paires de  $(I_i, I_j)$  ayant une intersection non vide

$$LCOM = \max(|P| - |Q|, 0)$$

- LCOM peut être visualisé comme un graphe bi-partite
  - Le premier ensemble de nœuds correspond aux  $n$  différents attributs et le second aux  $m$  différentes méthodes
  - Un attribut est lié à une fonction si elle y accède ou modifie sa valeur
  - L'ensemble des arcs est  $Q$
  - Il y a  $n \times m$  arcs possibles, et  $|P| = n \times m - |Q|$


## 6.5 Métriques MOOD

### Métriques MOOD


- Ensemble de métriques pour mesurer les attributs des propriétés suivantes :
  - Encapsulation
  - Héritage
  - Couplage
  - Polymorphisme


### Encapsulation

- MHF : Method Hiding Factor (10-30%)

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)} \quad \text{$$

avec

- $M_d(C_i)$  le nombre de méthodes déclarées dans une classe  $C_i$
- $M_h(C_i)$  le nombre de méthodes cachées
- $TC$  le nombre total de classes.
- AHF : Attribute Hiding Factor (70-100%) 

$$MHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)} \quad \text{$$

avec

- $A_d(C_i)$  le nombre d'attributs déclarés dans une classe  $C_i$
- $A_h(C_i)$  le nombre d'attributs cachés

### Facteurs d'héritage

- MIF : Method Inheritance Factor (65-80%)



$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

avec

- $M_i(C_i)$  le nombre de méthodes héritées (et non surchargées) de  $C_i$
- $M_a(C_i)$  le nombre de méthodes qui peuvent être appelées depuis la classe  $i$
- AIF : Attribute Inheritance Factor (50-60%)

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

avec

- $A_i(C_i)$  le nombre d'attributs hérités de  $C_i$
- $A_a(C_i)$  le nombre d'attributs auxquels  $C_i$  peut accéder

### Facteur de couplage

- CF : Coupling Factor (5-30%)
  - Mesure le couplage entre les classes sans prendre en compte celui dû à l'héritage

$$CF = \frac{\sum_{i=1}^{TC} \sum_{j=1}^{TC} client(C_i, C_j)}{TC^2 - TC}$$

avec

- $client(C_i, C_j) = 1$  si la classe  $i$  a une relation avec la classe  $j$ , et 0 sinon
- Les relations d'héritage ne sont pas prises en compte dans les relations

### Facteur de polymorphisme

- PF : Polymorphism Factor (3-10%)
  - Mesure le potentiel de polymorphisme d'un système

$$PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} M_n(C_i) \times DC(C_i)}$$

avec

- $M_o(C_i)$  le nombre de méthodes surchargées dans la classe  $i$
- $M_n(C_i)$  le nombre de nouvelles méthodes dans la classe  $i$
- $DC(C_i)$  le nombre de descendants de la classe  $i$

## 7 Analyse et gestion des risques

### Le risque

- Risque : probabilité qu'un événement indésirable ait lieu. Le risque implique des idées de
  - Incertitude : les événements ne se produiront pas de manière certaine
  - Perte : plus l'événement est indésirable, plus le risque est grand
- Une gestion proactive des risques peut aider à minimiser les effets négatifs d'événements susceptibles de se produire
- Types de risques :
  - Les risques de projet concernent le déroulement du projet
  - Les risques techniques portent sur la qualité du produit
  - Les risques commerciaux peuvent affecter sa viabilité

### Exemples de types de risques

Risque	Projet	Tech.	Com.	Propre
Matériel non disponible		×		×
Spécifications incomplètes	×			
Utilisation de méthodes spécialisées		×		×
Problèmes pour atteindre la fiabilité désirée		×		×
Départ d'une personne clé	×			
Sous-estimation des efforts nécessaires	×			
Le seul client potentiel fait faillite			×	×

### Calcul des risques

- Utilisation de probabilités élémentaires
  1. Estimer la probabilité du risque
  2. Estimer l'impact, le coût des conséquences
  3. Calculer le risque en multipliant ces deux valeurs
- Exemple : jeu à deux dés à six faces
  - L'obtention d'un 7 fait perdre 60 euros, quel est le risque ?
    1. Il y a 6 façons d'obtenir un 7, soit une probabilité de  $6/36 = 1/6$
    2. L'impact est de 60 euros
    3. Le risque vaut donc  $1/6 \times 60 = 10$  euros

### Atténuation des risques

- Stratégie proactive pour tenter de diminuer
  - L'impact d'un risque
  - La probabilité d'un risque
- *Pas de solution miracle !*
  - Identifier très tôt les risques les plus importants
  - Utiliser un cycle de vie incrémental et fondé sur les risques
  - Prototyper autant que possible

### Exemples de stratégies d'atténuation des risques

Risque	Réd. Proba	Réd. Impact
Matériel non disponible	Accélérer le dév. du matériel	Concevoir un simulateur
Spécifications incomplètes	Approfondir les contrôles des spéc.	
Utilisation de méthodes spécialisées	Former les équipes, engager des experts	
Problèmes pour atteindre la fiabilité désirée	Orienter la conception vers la fiabilité	
Départ d'une personne clé	Augmenter les salaires	Engager d'autres personnes
Sous-estimation des efforts nécessaires	Diagnostic par un expert externe	Respect des délais, estimations fréquentes
Le seul client potentiel fait faillite		Trouver d'autres clients potentiels

## 8 Tests logiciels

### Test logiciel

- Tester un logiciel : Exécuter le logiciel avec un ensemble de données réelles

« *Un programme sans spécifications est toujours correct* »

- Il faut confronter résultats de l'exécution et résultats attendus
- Impossibilité de faire des tests exhaustifs
  - Ex : 2 entiers sur 32 bits en entrée :  $2^{64}$  possibilités. Si *1ms* par test, plus de  $10^8$  années nécessaires pour tout tester
- Choix des cas de tests :
  - Il faut couvrir *au mieux* l'espace d'entrées avec un nombre réduit d'exemples
  - Les zones sujettes à erreurs nécessitent une attention particulière

### 8.1 Tests fonctionnels

#### Tests fonctionnels

- Identification, à partir des spécifications, des sous-domaines à tester
  - Produire des cas de test pour chaque type de sortie du programme
  - Produire des cas de test déclenchant les différents messages d'erreur
- Objectif : Disposer d'un ensemble de cas de tests pour tester le programme complètement lors de son implémentation
- Test « boîte noire » parce qu'on ne préjuge pas de l'implémentation
  - Identification possible des cas de test pertinents avant l'implémentation
  - Utile pour guider l'implémentation

#### Exemple

- Problème : Créer un ensemble de cas de test pour un programme qui
  1. Prend trois nombres a, b et c
  2. Les interprète comme les longueurs des côtés d'un triangle
  3. Retourne le type de triangle

Sous-domaines	Données de test
<i>Scalènes :</i>	
Longueurs par ordre croissant	(3, 4, 5) — scalène
Longueurs par ordre décroissant	(5, 4, 3) — scalène
Côté le plus long en second	(4, 5, 3) — scalène
<i>Isocèles :</i>	
$a = b$ et $c$ plus long	(5, 5, 8) — isocèle
$a = c$ et $b$ plus long	(5, 8, 5) — isocèle
$c = b$ et $a$ plus long	(8, 5, 5) — isocèle
$a = b$ et $c$ plus court	(8, 8, 5) — isocèle
$a = c$ et $b$ plus court	(8, 5, 8) — isocèle
$c = b$ et $a$ plus court	(5, 8, 8) — isocèle
<i>Équilatéraux :</i>	
Tous les côtés égaux	(5, 5, 5) — équilatéral
<i>Pas un triangle :</i>	
Côté le plus long en premier	(6, 4, 2) — pas un triangle
Côté le plus long en second	(4, 6, 2) — pas un triangle
Côté le plus long en dernier	(1, 2, 3) — pas un triangle
<i>Entrées incorrectes :</i>	
Une entrée incorrecte	(-1, 2, 4) — ent. incorrectes
Deux entrées incorrectes	(3, -2, -5) — ent. incorrectes
Trois entrées incorrectes	(0, 0, 0) — ent. incorrectes

### Matrices de test

- Les matrices de test permettent de
  - Formaliser l'identification des sous-domaines à partir des conditions des spécifications
  - Indiquer systématiquement si les combinaisons de conditions sont vraies ou fausses
  - Toutes les combinaisons possibles de V et de F seront examinées

### Exemple

- À partir des spécifications, on identifie des conditions d'exécution du programme qui permettront de produire les sorties voulues :
  1.  $a = b$  ou  $a = c$  ou  $b = c$
  2.  $a = b$  et  $b = c$
  3.  $a < b + c$  et  $b < a + c$  et  $c < a + b$
  4.  $a > 0$  et  $b > 0$  et  $c > 0$

Condition	Cas 1	Cas 2	Cas 3	Cas 4	Cas 5	Cas 6	Cas 7	Cas 8
(1)	V	V	V	V	V	F	F	F
(2)	V	V	F	F	F	F	F	F
(3)	F	V	F	F	V	F	F	V
(4)	F	V	F	V	V	F	V	V
Entrée	(0, 0, 0)	(3, 3, 3)	(0, 4, 0)	(3, 8, 3)	(5, 8, 5)	(0, 5, 6)	(3, 4, 8)	(3, 4, 5)
Sortie	Erron.	Equi.	Erron.	Pas Tri.	Iso.	Erron.	Pas Tri.	Scal.

- NB : Il est impossible d'avoir (3)=V et (4)=F, ou encore (2)=V et (1)=F

## 8.2 Tests structurels

### Tests structurels

- Tests « boîte blanche »: détermination des cas de test en fonction du code
- Critère de couverture : Règle pour sélectionner les tests et déterminer quand les arrêter
  - Au pire, on peut sélectionner des cas de test au hasard jusqu'à ce que le critère choisi soit satisfait
- Oracle : Permet de déterminer la sortie attendue associée aux cas sélectionnés
  - Difficile à mettre en place si on veut automatiser complètement le processus de test

### Couverture de chaque instruction (C0)

- Selon ce critère, chaque instruction doit être exécutée avec des données de test
- Sélectionner des cas de test jusqu'à ce qu'un outil de couverture indique que toutes les instructions du code ont été exécutées

### Exemple

Noeud	Ligne de code	(3, 4, 5)	(3, 5, 3)	(0, 1, 0)	(4, 4, 4)
A	read a,b,c	×	×	×	×
B	type="scalène"	×	×	×	×
C	if (a==b  b==c  a==c)	×	×	×	×
D	type="isocèle"		×	×	×
E	if (a==b&&b==c)	×	×	×	×
F	type="équilatéral"				×
G	if (a>b+c  b>a+c  c>a+b)	×	×	×	×
H	type="pas un triangle"			×	
I	if (a<=0  b<=0  c<=0)	×	×	×	×
J	type="données erronées"			×	
K	print type	×	×	×	×

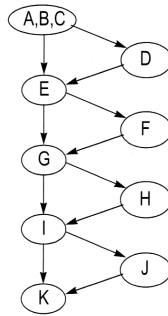
- Après le quatrième test, toutes les instructions sont exécutées
- *Il est rare que le jeu minimal soit bon d'un point de vue fonctionnel*

### Test de toutes les branches (C1)

- Plus complet que C0
- Selon ce critère, il faut emprunter les deux directions possibles au niveau de chaque décision
- Nécessite la création d'un graphe de contrôle et de couvrir chaque arc du graphe

### Exemple



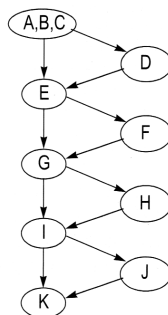


Arcs	(3, 4, 5)	(3, 5, 3)	(0, 1, 0)	(4, 4, 4)
ABC-D		x	x	x
ABC-E	x			
D-E		x	x	x
E-F				x
E-G	x	x	x	
F-G				x
G-H			x	
G-I	x	x		x
H-I			x	
I-J			x	
I-K	x	x		x
J-K			x	

### Test de tous les chemins

- Encore plus complet
- Selon ce critère, il faut emprunter tous les chemins possibles dans le graphe
- Chemin : suite unique de nœuds du programme exécutés par un jeu spécifique de données de test
- *Peu adapté au code contenant des boucles, le nombre de chemins possible étant souvent infini*

### Exemple



Chemin	V/F	Données	Sortie
ABCEGIK	FFFF	(3, 4, 5)	Scalène
ABCEGHIK	FFVF	(3, 4, 8)	Pas un triangle
ABCEGHIJK	FFVV	(0, 5, 6)	Données erronées
ABCDEGIK	VFFF	(5, 8, 5)	Isocèle
ABCDEGHIK	VFVF	(3, 8, 3)	Pas un triangle
ABCDEGHIJK	VFVF	(0, 4, 0)	Données erronées
ABCDEFGIK	VFVV	(3, 3, 3)	Equilatéral
ABCDEFGHIJK	VVVV	(0, 0, 0)	Données erronées

### Couverture des conditions multiples

- 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
- Très complet, ne pose pas de problème en cas d’itérations

### Exemple

- Pour `if (a==b || b==c || a==c)`

Combinaison	Données de test	Branche
V??	(3, 3, 4)	ABC-D
FV?	(4, 3, 3)	ABC-D
FFV	(3, 4, 3)	ABC-D
FFF	(3, 4, 5)	ABC-E

## 8.3 Test de flot de données

### Test de flot de données

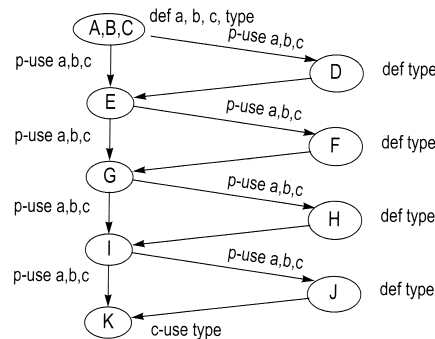
- Test structurel
- S’appuie sur la circulation des données à l’intérieur du programme
  - Elles circulent depuis l’endroit où elles sont définies jusqu’à celui où elles sont utilisées
    - def : définition d’une donnée, correspond à l’attribution d’une valeur à une variable
    - c-use : utilisation pour calcul, la variable apparaît à droite d’un opérateur d’affectation
    - p-use : utilisation comme prédicat dans une condition
- A noter :
  - L’utilisation p-use est attribuée aux deux branches conditionnelles
  - un chemin sans définition, noté def-free, va de la définition d’une variable à une utilisation sans passer par d’autres définition

### Critères de tests de flots de données

- dcu : chaque définition doit être reliée à un c-use en aval par un chemin sans définition
- dpu : chaque p-use doit être reliée à une définition en amont par un chemin sans définition
- du : combinaison des deux précédents

- all-du-paths : le plus exigeant, qui veut que tous les chemins possibles entre une définition et une utilisation doivent être sans définition

### Example



- dcu : le seul c-use porte sur type et se trouve au nœud K
  - Depuis ABC jusqu'à K : chemin ABCEGIK
  - Depuis D jusqu'à K : chemin DEGIK
  - Depuis F jusqu'à K : chemin FGIK
  - Depuis H jusqu'à K : chemin HIK
  - Depuis J jusqu'à K : chemin JK
- dpu : les p-use portent sur les variables a,b et c qui ne sont définies que dans le nœud ABC
  - Depuis ABC jusqu'à l'arc ABC-D
  - Depuis ABC jusqu'à l'arc ABC-E
  - Depuis ABC jusqu'à l'arc E-F
  - Depuis ABC jusqu'à l'arc E-G
  - Depuis ABC jusqu'à l'arc G-H
  - Depuis ABC jusqu'à l'arc G-I
  - Depuis ABC jusqu'à l'arc I-J
- du : tous les tests de dcu et de dpu combinés
- all-du-paths : mêmes tests que pour du

## 8.4 Tests orientés objet

## Spécificité des tests orientés objet

- Habituellement :
  - Couverture des instructions
  - Couverture des branches
  - Couverture du flot de données
- Ces techniques s'appuient sur un diagramme de contrôle
- Ces diagrammes sont mal adaptés pour représenter la structure d'un logiciel objet
  - Les complexités de ces logiciels objet résident plutôt dans les interactions entre les méthodes
- Il faut couvrir les appels de méthodes

**Couverture pour les tests objet**

- Tests MM (méthode, message)
  - Tester tous les appels de méthode
  - Si une méthode en appelle une autre plusieurs fois, chaque appel est testé séparément
- Couverture des paires de fonctions
  - Tester tous les enchaînements de deux méthodes possibles
  - Utilisation d'expressions régulières pour identifier les paires à couvrir