Université de Poitiers et ENSMA
Site du Futuroscope. BP 109
86960 FUTUROSCOPE Cedex

Principes du Génie Logiciel

Y. AIT-AMEUR,
Université de Poitiers
05 49 49 80 77
yamine@ensma.fr

Plan du cours

- **I.** Introduction
- **II.** Éléments de terminologie
- **III**. Environnements de programmation
- **IV.** Les langages de programmation
- **V.** La Qualité
- **VI.** Modèles de développement de logiciels
- **VII.** Gestion de projets

Plan du cours

- **VIII.** Méthodes de conception de logiciels
- **IX.** Fiabilité du logiciel
- **X.** Test du logiciel
- **XI.** Gestion des versions
- **XII.** Réutilisation de logiciels
- **XIII.** Maintenance de logiciels
- **XIV.** Introduction aux méthodes formelles

Plan du cours

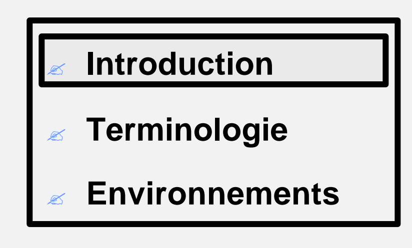
- ? **I. Introduction**
- **II.** Éléments de terminologie
- **III**. Environnements de programmation
- **IV.** Les langages de programmation
- **V.** La Qualité
- **VI.** Modèles de développement de logiciels
- **VII.** Gestion de projets

Introduction

Crise du Logiciel

- % du PNb des pays industrialisés
- Pénuries :
  - an 2000,
  - Euro
  - Nouvelles technologies et Internet
- Nouvelles architectures de machines

Introduction



Exemples de problèmes

- **Sonde Mariner vers Vénus :**
  - Erreur détectable à la compilation dans un programme Fortran
- **Navette spatiale :**
  - Retard de lancement de deux jours en 1981, navette lancée sans résolution du problème logiciel

Introduction

- **Socrate :**
  - Réutilisation inadaptée
- **Ariane 5 :**
  - Non vérification des types,
  - Réutilisation inadéquate (Ariane 4),
  - Dépassement de capacité,
  - Programme inutile

Introduction

- Problèmes
  - Logiciel = produit manufacturé complexe
  - Spécification des caractéristiques
  - Invisibilité du logiciel en cours de réalisation
  - Problème du test
  - «Flexibilité » du logiciel ==> erreurs

Introduction

- Nouveaux problèmes
  - Logiciels récents :
    - Plus fiables, plus satisfaisants
  - Situations critiques : nucléaire, transports
  - Logiciels complexes :
    - Navette spatiale = 50 millions de lignes,
    - Catia = 1500 a/h, 600 développeurs
  - Chez Microsoft :
    - {ingénieurs développeurs} = {ingénieurs testeurs}

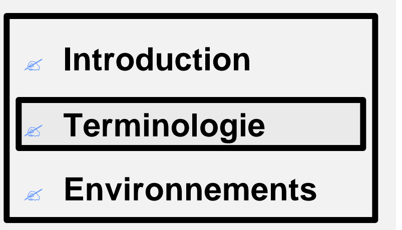
Introduction

- Quelques remèdes possibles :
  - Outils de génie logiciel
  - Environnements de programmation
  - Nécessité d 'une base théorique cohérente
  - Contrôle de qualité
  - Le logiciel est considéré comme un produit à part entière ==> Ingénieurs

Plan du cours

- **I.** Introduction
- **?** **<u>II. Éléments de terminologie</u>**
- **III**. Environnements de programmation
- **IV.** Les langages de programmation
- **V.** La Qualité
- **VI.** Modèles de développement de logiciels
- **VII.** Gestion de projets

Éléments de Terminologie



- La production de logiciel nécessite la définition d'outils de représentation et de communication.
  ==> Support
  - Syntaxe,
  - Sémantique,
  - Pragmatique
- Science du langage

Éléments de Terminologie

Si on se réfère à celui qui parle, ou en termes plus généraux aux usagers du langage, nous attribuons cette investigation à la *pragmatique*.
Si nous faisons abstraction des usagers du langage et si nous analysons seulement les expressions et leur signification, nous nous trouvons dans le domaine de la *sémantique*. Et si finalement, nous faisons abstraction des significations pour analyser uniquement les relations entre expressions, nous entrons dans la *syntaxe*.

Ces trois éléments constitue la science du langage ou *sémiotique*.

Éléments de Terminologie

- Syntaxe :
  - Grammaire formelle
  - Arbre syntaxique pour la représentation
  - Nécessaire à la sémantique et à la pragmatique
  - Exemple : les expressions arithmétiques

E ::=E + E |
E*E |
E/E |
E-E |
(E) |
id |
cst

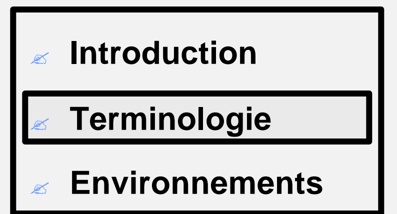
- Exemple :
  - Représentation de 2*(Pi*R)

- Sémantique :
  - Langages de programmation,
  - Langages de spécification,
  - Techniques,
  - Analyseurs de code,
  - Génération de tests,
  - Contrôle de types.

Éléments de Terminologie

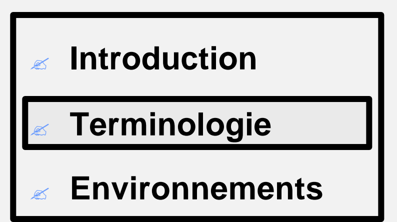
- Exemple
  - Sémantique de 2*Pi*R
    - R est de type Réel ou entier
    - PI est la constante 3.14….
    - Si R vaut 1, alors 2*Pi*R vaut 6.28….
    - Si R vaut ‘ aaa ’ alors erreur

Éléments de Terminologie



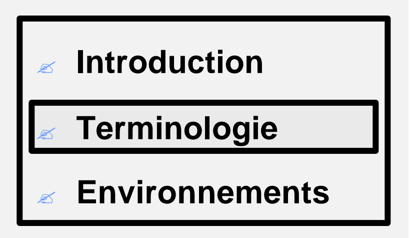
- Pragmatique :
  - Méthodes
  - Méthodologie
  - Notations
  - Règles
  - Patrons,
  - ….
- pour le développement, la maintenance, la spécification, le test, la documentation, ...

Éléments de Terminologie



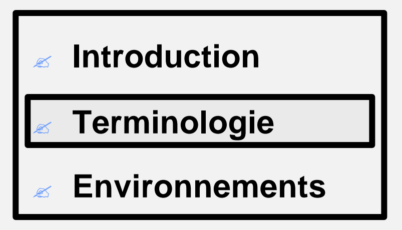
- Exemple
  - Pragmatique de 2*Pi*R
    - Circonférence d’un cercle de rayon R
    - Circonférence d ’une bille métallique de rayon R
    - Circonférence d’un disque de diamètre 2*R
    - ….

Éléments de Terminologie



- Méthodologie
  - Agrégation d'un ensemble de modèles, de techniques et de méthodes
- Modèle (lié à formalisme)
  - Interprétation de la compréhension d'une situation,
  - Description d'entités et de leur relations
  - Définition de Minsky
    - Pour un observateur **A**, **M** est un modèle de l 'objet **O**, si **M** aide **A** à répondre aux questions qu 'il se pose sur **O**

Éléments de Terminologie



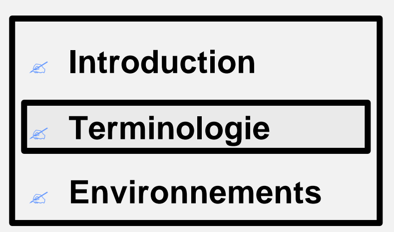
- **Méthode**

  Ensemble de règles biens définies
  qui conduisent pour le problème
  à une solution correcte.
  Importance de l ' enchaînement des règles

- **Technique**

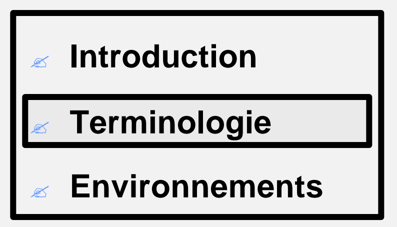
  Support de réalisation

Éléments de Terminologie



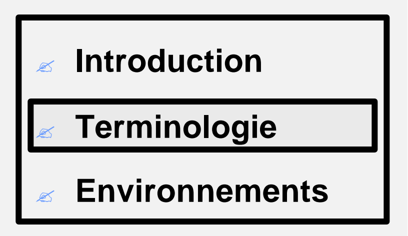
- **Naissance du génie logiciel**
  - Conférence de l'OTAN, 7 - 11 octobre 1968 : *software engineering*
- Terme « gènie »
  - Science de l 'ingénieur
  - Nécessité de :
    - bases théoriques
    - méthodes et outils validés par la pratique.

Éléments de Terminologie



- **Génie Logiciel :**
  - art de spécifier, concevoir, réaliser, et faire évoluer,
  - avec des moyens
  - et dans des délais raisonnables,
  - des programmes, des documentations et des procédures de qualité
  - en vue d'utiliser un ordinateur pour résoudre certains problèmes.

Éléments de Terminologie



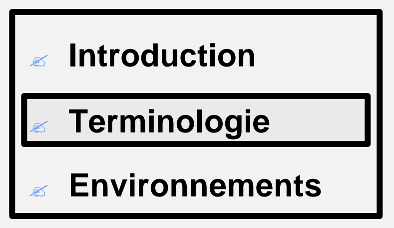
- Atelier de Génie Logiciel

  Progiciel supportant plusieurs activités du cycle de vie (i.e. plusieurs outils de génie Logiciel)

- Spécification

  Activité concourant à transformer l'expression des besoins (cahier des charges) en solution

Éléments de Terminologie



- Conception

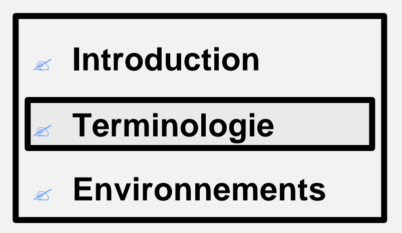
  Activités concourant à l'établissement de l'architecture technique du logiciel

- Rétro-conception

  Processus d'analyse de l'existant

  (en vue de sa reprise)

Éléments de Terminologie



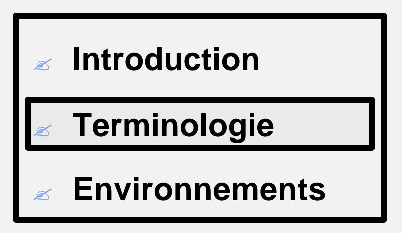
- Gestion de projet

  Activités permettant de maîtriser la mise en œuvre d'un produit (coûts, délais, méthodes et techniques, ressources, qualité...)

  Prototype

  Logiciel développé pendant la réalisation pour supporter des expérimentations

Éléments de Terminologie

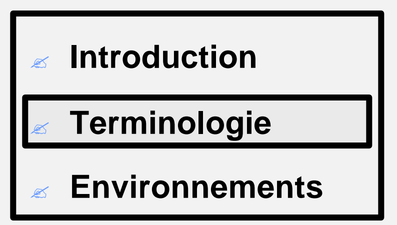


- Assurance Qualité

  Actions préétablies et systématiques pour donner la confiance appropriée en un produit ou un service

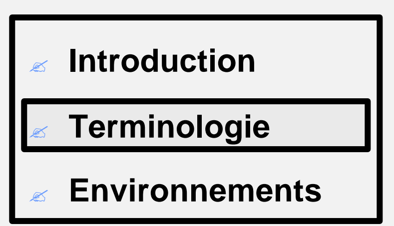
- Importance du service qualité
  - d'autant plus développé que l'entreprise est grande
  - 80% des services ont moins de cinq spécialistes

Éléments de Terminologie



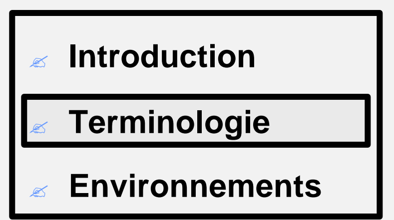
- Spécificité du processus de production
  - Reproduction = copie
  - Pas de notion d 'usure.
  - Évolution des spécifications
    - ==> Logiciel obsolète

Éléments de Terminologie



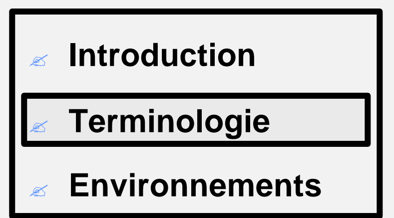
- Développement
  - Descriptions de plus en plus précises
  - Descriptions proches d 'un programme : raffinements
  - Itératif: retour sur la conception après vérification ou validation (tests, preuves, …)

Éléments de Terminologie



- Maintenance
  - Remise en cause du développement
- Exploitation
  - Mise en service du logiciel

Éléments de Terminologie

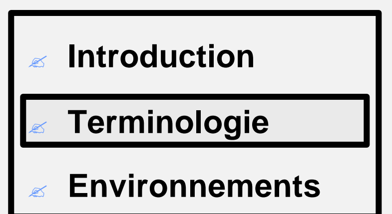


- Cycle de vie du logiciel :

Développement + Exploitation + Maintenance

- **Nécessité** de **modèles** de **développement**.

Éléments de Terminologie



- Caractéristiques des logiciels
  - Propriétés des logiciels
  - Deux catégories de paramètres
    - Externes : Facteurs
    - Internes : Critères
- Évaluations, mesures de ces caractéristiques
    - Vérification,
    - Validation,
    - Test,
    - Métriques, ...

Éléments de Terminologie
Facteurs

- Modifiabilité
- Extensibilité
- Modularité
- Ergonomie
- Facilité d'utilisation
- Compatibilité
- Portabilité
- Interopérabilité
- Documentation

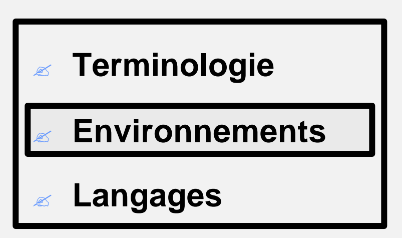
Éléments de Terminologie
Critères

- Rapidité
- Compacité
- Documentation
- Noms de variables, de fonctions et de procédures

Plan du cours

- ✍ **I.** Introduction
- ✍ **II.** Éléments de terminologie
- ✍? **<u>III. Environnements de programmation</u>**
- ✍ **IV.** Les langages de programmation
- ✍ **V.** La Qualité
- ✍ **VI.** Modèles de développement de logiciels
- ✍ **VII.** Gestion de projets

Les environnements de programmation



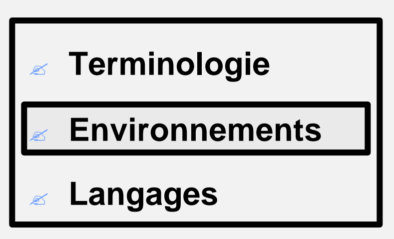
- Ensemble des outils permettant la conception, l'élaboration et la réalisation de programmes de qualité.

- Mis en œuvre pendant toutes les phases de développement de logiciels.

Les environnements de programmation

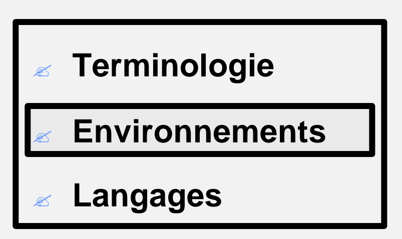
- Matériels
  - ordinateurs, souris, E/S
- Éditeurs
  - classiques, syntaxiques,
- Compilateurs
  - générateurs de code
- Interpréteurs
  - exécution

Les environnements de programmation



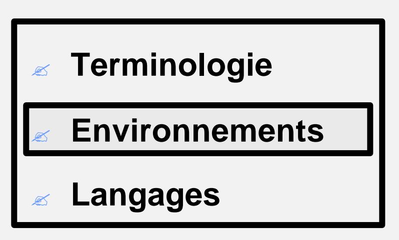
- Bibliothèques
  - ensemble de modules, de classes réutilisables
- Gui-Builders
  - Generate User Interface
- Générateurs de documentation
  - Javadoc : génère des documentations au format HTML

Les environnements de programmation



- Analyseurs : analyse de programmes ou de spécification.
  - comportements des programmes
  - détection d'erreurs
  - compréhension de code
  - optimisation de code
  - preuve de propriétés de programmes

Les environnements de programmation



- Analyseurs statiques
  - Exemples : évaluateurs partiels, contrôleurs de types,….
- Analyseurs dynamiques :
  - debbogueur,
  - générateurs de tests (boîte noire, boîte blanche)
    - assertions : package « assert » en ADA.

Les environnements de programmation

- Prototypage
  - exécution de programme dans un autre environnement,
  - langages dédiés au prototypage
- Maquettage
  - exemple : construction de maquettes à partir d'outils de génération d 'interfaces

Les environnements de programmation

- Environnements de programmation :
  - pour plusieurs langages de programmation
    - C++, Java, Ada, …
- Environnements de développement
  - support partiel du développement
    - conception et de génération de code
    - test
    - outils de spécification et de vérification de code puis génération

Plan du cours

- **I.** Introduction
- **II.** Éléments de terminologie
- **III**. Environnements de programmation
- **?** **IV. Les langages de programmation**
- **V.** La Qualité
- **VI.** Modèles de développement de logiciels
- **VII.** Gestion de projets

Les langages de programmation

Premiers langages :

- impératifs et séquentiels,
- fortement liés au matériel

« Von Neumann style »

Nouvelles approches :

- facilité d'expression et de compréhension
- « Von Neumann Style free »

Les langages de programmation

- **Nouvelles architectures :**
  - parallélisme,
  - synchronisation
  - partage

- **Atout pour développer du logiciel :**
  - langages bien définis syntaxiquement et sémantiquement,
  - bien adaptés au domaine d'application

Les langages de programmation

- **Points cruciaux :**
  - compréhension,
  - maintenance,
  - réutilisation :
    - modularité,
    - généricité,
    - encapsulation
  - vérification et preuves : origine de la programmation structurée, du typage, et des approches fonctionnelle et logique

Langages de programmation: Typage

Contrôle de type

- vérification du type des arguments des opérations

Synthèse de types

- calcul du type des arguments et des opérations

Contrôle statique des types :

- détection d'erreurs
- tests de types évités

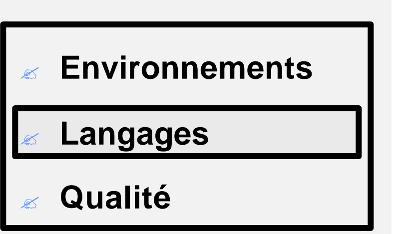
Langage fortement typé :

- contrôle de toutes les variables et opérations possibles

Langages de programmation: Typage

- **Langages non typés**
  - assembleur, Lisp, Prolog, Smalltalk
- **Langages typés**
  - Pascal, C
- **Langages fortement typés**
  - - ADA, ML
- **Langages à synthèse de types**
  - ML, CAML, OCAML,

Langages de programmation: Typage



- **Nécessité d'un typage souple :**
  - **1-** procédure pour des tableaux de bornes différentes (impossible en Pascal)
  - **2 -** compatibilité entiers, réels, etc.
  - **3 -** opérations utilisable pour plusieurs types :
- Exemple : tri d 'un tableau

Langages de programmation: Typage

Traitement de (1)

En ADA : *type* et *contrainte* :

- pour les tableaux : type
- pour leurs indices : contrainte

```
type TAB is array (NATURAL range <>) of FLOAT;

procedure tri (T : in out TAB; N in NATURAL) is
      ... T(I) ... -- texte de la procédure …
end tri ; ...
```

```
BTRIER: TAB(2..200);
...
BTRIER(I) := ...;
tri(BTRIER, 150);
```

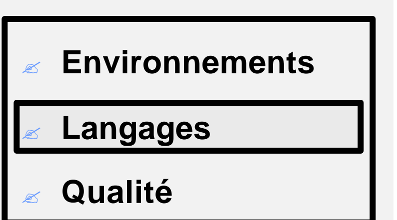
```
ATRIER: TAB(1..100);
...
ATRIER(I) := ...;
tri(ATRIER, 50);
```

Langages de programmation: Typage

Traitement de (2) :

- *surcharge* : plusieurs opérations homonymes (efficace si le choix est fait à la compilation)
- *conversions implicites* présentes dans Pascal et dans C

Langages de programmation: Typage



Traitement de (3) :

- paramétrage par un type, éventuellement avec une contrainte : *généricité*
- cas classique : tri paramétré

Langages de programmation: Typage

Types génériques + contrôle de type fort= *polymorphisme*

Surcharges et conversion = *polymorphisme ad hoc*

Langages de programmation: Typage

- Polymorphisme dû aux sous-types
  - Sous-typage :
    - *règle de substitution* : un type *t* est un sous-type d'un type *t*' si dans tout contexte, une expression du type *t*' peut être remplacée par une expression de type *t*.
- Exemples de sous-typage :
    - inclusion d'intervalles
    - sous-ensembles
    - ajout d'un champ à une structure :
      - exemple : point / point coloré

Modularité :

- Décomposition d'un logiciel en plusieurs parties lors de sa conception

- Exemples : *packages* en ADA, les classes et les packages en Java

Langages de programmation: Modularité

- **Description de l'interface d'un module :**
  - Partie visible ou publique d 'un module
  - Noms des opérations, variables, types *exportés*, *i.e.* utilisables par d'autres modules, qui les *importent.*
- **Corps d'un module :**
  - Entités(noms des opérations, variables, types) exportées
  - Entités (noms des opérations, variables, types) locales

Langages de programmation: Modularité

- Ecriture de programmes facilitée avec
  - l'encapsulation : on ne voit pas les structure de données « private »
  - la modularité: répartition des tâches de codage
- **Possibilité de passer outre :**
  - un seul grand module
  - interactions directes entre modules

Langages de programmation: Modularité

Modularité + encapsulation :

- indispensables pour la tolérance aux incidents dus à l'environnement ou au module lui-même
- programmation défensive : chaque module vérifie
  - ses données et
  - ses résultats
- redondance diversifiée ou non :
  - Plusieurs modules, plusieurs machines identiques ou non

Langages de programmation: Modularité

- **Type abstrait = interface de module** définition
  - de noms de types,
  - d'opérations et
  - de propriétés,
  - sans accès à la définition «*interne* »
    - Exemple : les ensembles.
- Type abstrait = classe de types de données
  - **Interface** : plusieurs corps possibles
  - **Types abstraits :** décomposition naturelle en modules

Langages de programmation: Modularité

Exceptions et récupérations

- **Cas classiques :**
  - traitement de fin de fichier
  - débordements mémoire
- **Généralisation des exceptions :**
  - avec l'apparition des types abstraits
  - dans l'interface des modules

- *Modules génériques*
- **Exemple en ADA :**

```
-- partie interface du module générique STACK
generic
     SIZE : POSITIVE;
type ITEM is private;
-- 2 paramètres : 1 entier positif et 1 type
package STACK is
      procedure PUSH(E : in ITEM)
      procedure POP(E : out ITEM)
OVERFLOW, UNDERFLOW : exception;
end STACK;
```

Utilisation

```
package STACK_INT is new
STACK(200, INTEGER) ;
        package STACK_BOOL is new STACK(100, BOOLEAN) ;
```

Objet ou instance = Entité nommée

- Possède une identité
- Possède des propriétés
  - descriptives : attributs
  - comportementales : opérations et méthodes
  - structurelles : objet père, fils, collatéral, ….
- Stocke un état
- Répond à des messages en exécutant des méthodes

Langages de programmation: Classes et objets

- Classification :
  - Regrouper tous les objets satisfaisant les mêmes propriétés descriptives, structurelles et procédurales.

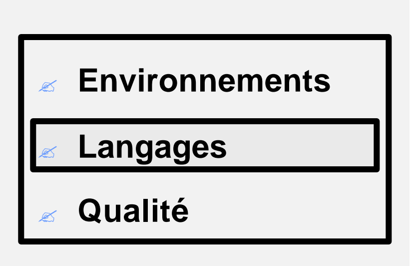
- **Classe :** Regroupement d 'objets obtenu par classification**.**

- **Objet = instance d'une classe**

Une classe contient

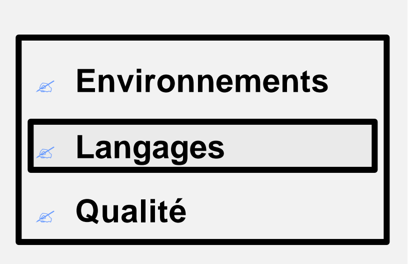
- Description des variables d'instance.
- Description des variables de classe
- Création (constructeurs) + destruction (destructeurs).
- Méthodes de la classe
- Ensemble des messages acceptés.
- Liens de structure avec les autres classes

Langages de programmation: Classes et objets



- Deux types de classes
  - une classe = machine à fabriquer des instances
    - Langages de classes : exemple C++
  - une classe = un objet représentant (clone)
    - Langages d 'objets : exemple SmallTalk
    - Les objets sont crées par recopie du représentant (clonage).

Langages de programmation: Classes et objets

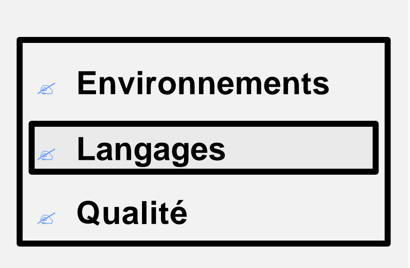


Différences entre classe et module

- création dynamique d'objets
- graphe d'utilisation avec d'éventuels circuits

Héritage :

- Liens structurels entre classes
  - sous-classe et super-classe
- Objets de la sous-classe
  - mêmes variables d'instance
  - mêmes méthodes
  - plus éventuellement d'autres attributs et/ou méthodes.
- Héritage simple et héritage multiple.
- Redéfinition d'une méthode lors de l'héritage
  - surcharge

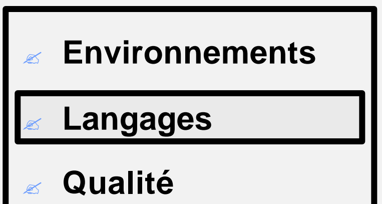


- **ATTENTION : héritage ? sous-typage**
  - pas de moyen de vérifier qu'un héritage est du sous-typage surtout avec l'héritage multiple et la généricité
  - Confusion souvent effectuée.

Langages de programmation. Classification

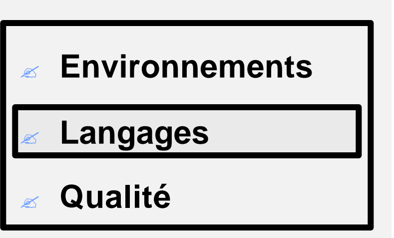
- Différentes classifications possibles
  - chronologie
    - ordre d 'apparition
  - génération
    - première, deuxième, …, cinquième
  - domaine d'application
    - calcul numérique, contrôle de procédés; temps réel, ….
  - **Sémantique et style de programmation**

Langages de programmation. Classification



- Programmation
  - Impérative
  - Déclarative
    - Fonctionnelle
    - logique
  - à objets

Langages de programmation. Classification

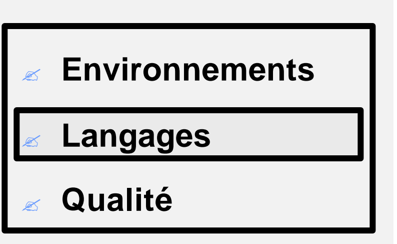


- Programmation impérative
  - « Von Neumann Style »
    - chargement, décodage, chargement opérandes, exécution, rangement des résultats
    - passage à l 'instruction suivante : séquence.
  - Affectation
    - instruction de base
    - création d'effets de bord

Langages de programmation. Classification

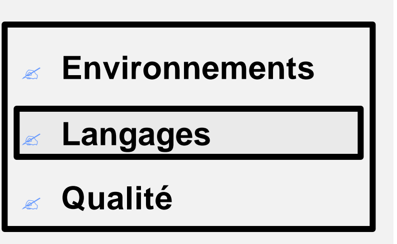
- Programmation impérative
  - Langages les plus courants, plus connus, plus utilisés
  - Ce sont les premiers
  - Structure de blocs
    - sans : Fortran
    - avec : Algol, PL1, C, Pascal, ADA, ….

Langages de programmation. Classification



- Programmation déclarative
  - Non Von Neuman Style
  - « Can programming be liberated from Von Neuman Style » article de *John Backus*
  - Pas d'ordre dans la déclaration des instructions d 'un programme
  - Un interpréteur simple pour ces langages
  - Deux types :
    - Programmation fonctionnelle
    - Programmation logique

Langages de programmation. Classification



- Programmation fonctionnelle ou applicative
  - Principe : déclarer des fonctions et composer.
  - Basée sur le lambda calcul
  - Évaluation d 'une expression par application de fonctions prédéfinies ou définies par le programmeur.
  - Ordre supérieur possible :
    - Les fonctions sont des variables
  - Typage et Polymorphisme possibles
    - fondés sur le lambda calcul typé

Langages de programmation. Classification

- Exemples de langages
  - LISP : non typé, précurseur, Scheme ….
  - ML, CAML : langage fonctionnel typé avec synthèse de type et polymorphisme.
- Exemple de programme.

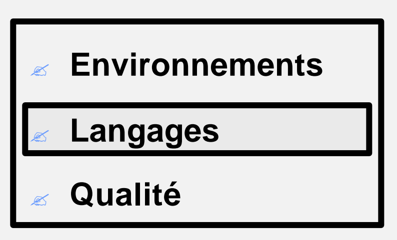
```
(Fact(3)+sum(8))/2

        WHERE

                sum(n) = If n== 0 Then 0 Else n + sum(n-1);

                fact(x) =  If n == 0 Then 1 Else n * fact(n-1);
        END;
```

Langages de programmation. Classification



- Programmation logique
  - Approche similaire à la programmation fonctionnelle
  - Principe : déclarer des règles et résoudre.
  - **Action de base : résolution**
    - Recherche des valeurs des variables qui rendent une expression vraie
  - Basé sur la logique des prédicats.
  - Ensemble de faits et de règles. La résolution permet de répondre à des questions exprimées par des formules logiques (ouvertes ou fermées).

Langages de programmation. Classification

- **Extensions :**
  - définition de fonctions
  - résolution de contraintes sur des types
- Exemples de langages
  - Prolog, Prolog III, ….
- Exemple de programme.

```
-- Règles
Pere(x,y) Pere(y,z) GrandPere(x,z)
Enfant(x,y) Pere(y,x)

-- Faits
Enfant(Paul, Henri)
Pere(Jacques, Henri)

-- Question ouverte
GrandPere(Jacques, x) ?

-- Question fermée
GrandPere(Jacques, Paul) ?
```

Langages de programmation. Classification

- Programmation à objets
  - Introduction de
    - classification, héritage, encapsulation, etc ..
    - communication par messages
  - **Premier langage :** SIMULA (années 60), pour la simulation et le maquettage
  - **Exemples :**
    - Smalltalk : non typé,
    - *Eiffel*, C++, Java …. : typés

Les langages de programmation

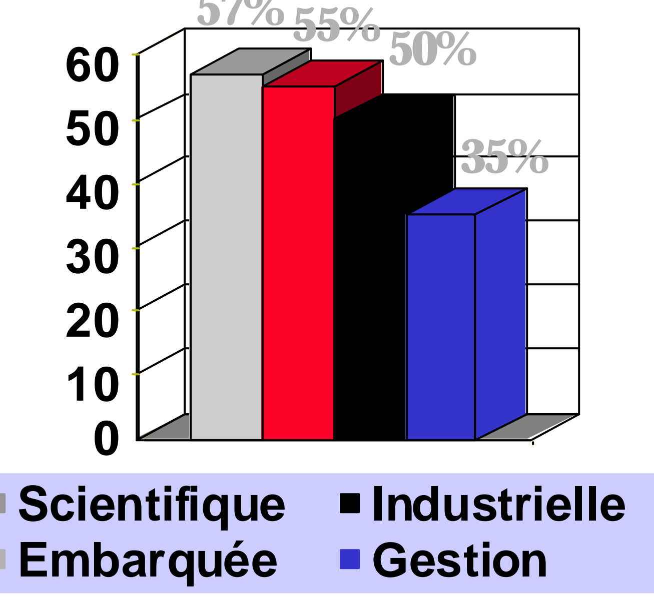
- Attention au choix du langage de programmation. En fonction
  - domaine d'application
  - nature de l 'application
    - temps réel, numérique, base de données, ….
  - logiciels déjà existants à réutiliser
  - pratique de l 'équipe de développement
  - adéquation avec les notations de conception et de développement

Plan du cours

- **I.** Introduction
- **II.** Éléments de terminologie
- **III.** Environnements de programmation
- **IV.** Les langages de programmation
- **?** **V. La Qualité**
- **VI.** Modèles de développement de logiciels
- **VII.** Gestion de projets

Pratique de la qualité

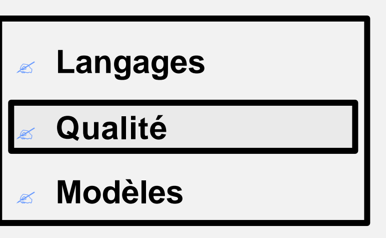
- Pratique en fonction des domaines d 'activité.



Pratique de la qualité

- Définition
  - Ensemble de
    - procédures,
    - règles et
    - conventions
  - approuvées et imposées par des organismes de normalisation
    - publics ou privés

Pratique de la qualité



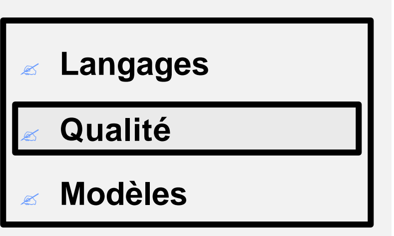
- Secteurs et organismes
  - Militaire : AQAP13, DoD 2167A, GAM 17 (DGA)
  - Aéronautique : DO 178B
  - Spacial : PSS ESA

Pratique de la qualité

- Exemples
  - ISO 9000

    Lignes directrices concernant l'organisation de l'entreprise
  - ISO 9000 est la plus employée
    - Forte croissance
    - 53% des entreprises ayant une structure qualité

Pratique de la qualité Classification



- Entreprises classées suivant différents points de vue.
  - Initial
  - Reproductible
  - Défini
  - Géré
  - Optimisé

Pratique de la qualité
Classification

Initial

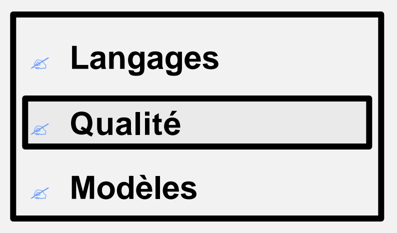
- sans méthode formelle, cohérence ni standard (concepteur/développeur = “artiste”).
- Problèmes
  - remplacement de personnel, de compétences, de machines, …
  - absence d’artiste

Pratique de la qualité
Classification

- **Reproductible**
  - consensus dans l'entreprise sur la "manière de faire" mais pas formation.
  - Gestion rigoureuse des coûts et des délais
  - mais repose sur des compétences individuelles
  - Problème
    - du remplacement de personnel, de compétences, de machines, ...

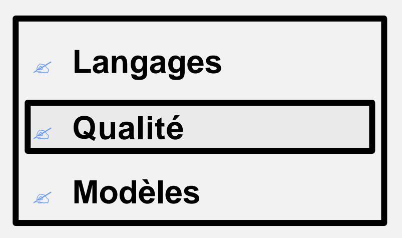
- **Défini**
  - Processus de développement formalisé et documenté.
  - Service de définition et de suivi des “méthodes” de l’entreprise.
  - Problèmes
    - Coût de définition du processus et de sa formalisation
    - mise aux normes des anciens développements
    - maintenance du processus de développement

Pratique de la qualité Classification



- Définition : Métrologie
  - **Science de la Mesure**
- **Géré**
  - Processus formel de collecte d'informations pour mesurer
    - le processus d'élaboration de systèmes ainsi que
    - les produits résultants (métrologie).

Pratique de la qualité
Classification

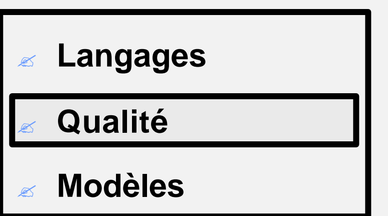


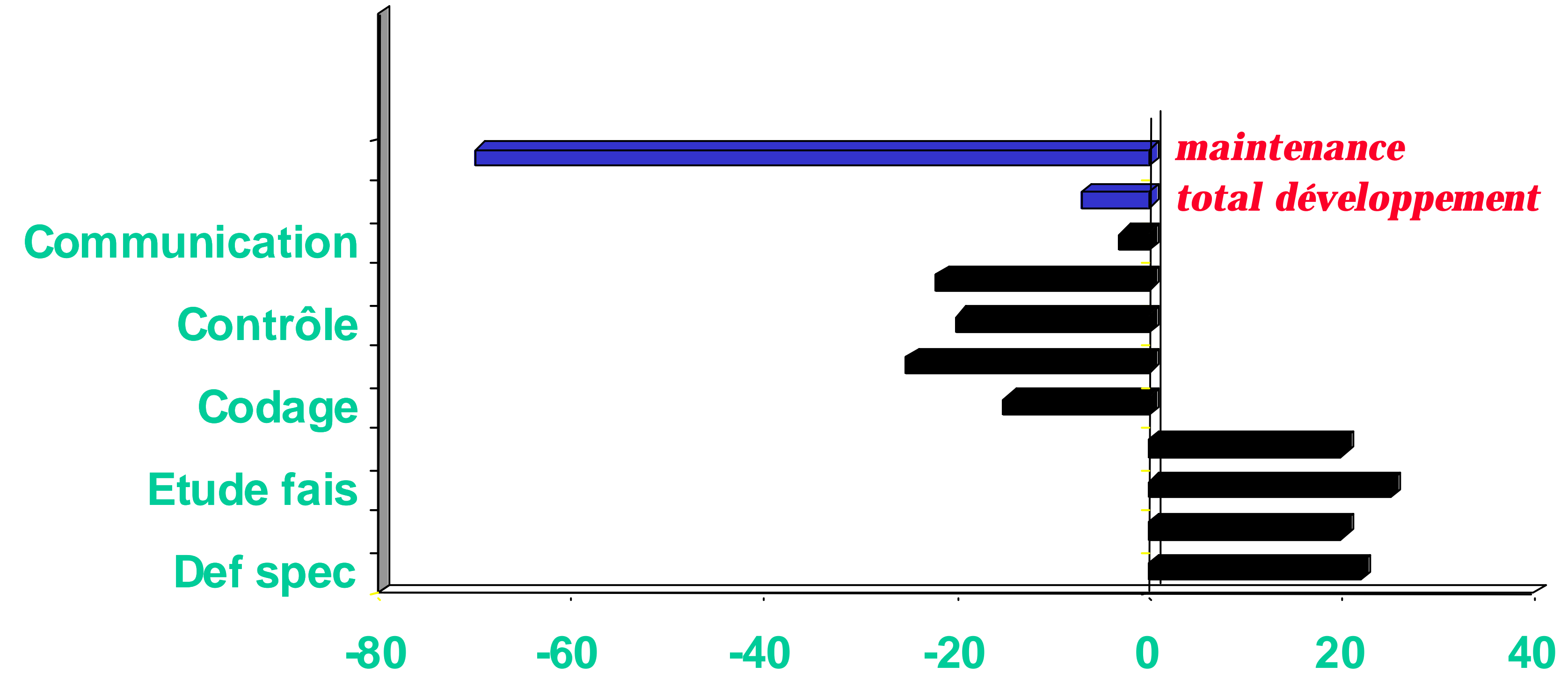
- Optimisé
  - Utilisation des résultats de la métrologie pour améliorer les méthodes.

Pratique de la qualité Classification

- Langages
- **Qualité**
- Modèles

Pratique de la qualité Classification





Evaluation des coûts de réalisation en fonction des étapes

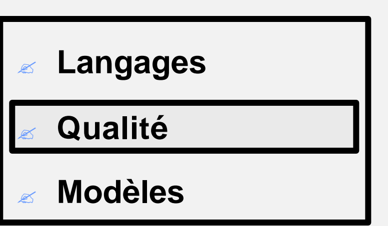
Pratique de la qualité
La certification ISO 9000

- OBJECTIFS:
  - Présenter et définir la norme
  - Présenter la procédure de certification

Pratique de la qualité
La certification ISO 9000

- Ensemble de textes définissant la qualité d 'organisation d 'une entreprise
- *17, 19 ou 20 points d'organisation à respecter*
- *Ne pas confondre avec la certification de produits ou de services*

Pratique de la qualité
La certification ISO 9000



- Un ensemble de textes
  - ISO 9000 : “chapeau” servant de guide aux autres
  - ISO 9003 : “de base”
  - ISO 9002 : “intermédiaire”
  - ISO 9001 : “la plus stricte”
  - ISO 9004 : guide de management de la qualité

Pratique de la qualité
La certification ISO 9000

Point à respecter	9001	9002	9003	Objet précis
Responsabilité de la direction	+	+	+	La direction du site certifié s'engage dans l'expression de la politique qualité et dans son organisation.
Système qualité	+	+	+	Les dispositions prises pour assurer la qualité des produits (système qualité) sont formalisées et rédigées (sous la forme d'un manuel qualité).
Revue de contrat	+	+	+	L'entreprise s'assure préventivement qu'elle peut satisfaire à ses engagements auprès de ses clients.
Maîtrise de la conception	+			Les données de planification, données d'entrée et de sortie, revue, vérification, validation sont maîtrisées.
Maîtrise des documents et des données	+	+	+	L'ensemble des documents qui conditionnent la qualité du produit sont sous contrôle : prévoir leur approbation, leur diffusion et la gestion des modifications.

Source : Management, Janvier 1996

Pratique de la qualité
La certification ISO 9000

Point à respecter	9001	9002	9003	Objet précis
Achats	+	+		La conformité des spécifications est contrôlée : évaluation des fournisseurs, données d'achat, vérifications.
Maîtrise du produit fourni par le client	+	+	+	L'entreprise veille à préserver le produit fourni par le client et destiné à être intégré dans le produit final.
Identification et traçabilité du produit	+	+	+	Les produits sont identifiés (s'ils s'y prêtent). Leur traçabilité est assurée (si les clients le spécifient).
Maîtrise des processus	+	+		L'entreprise doit être capable de définir l'ensemble de ses processus de production.
Contrôles et essais	+	+	+	Les contrôles et essais sont identifiés, à la réception, en cours de réalisation et en fin de processus.

Source : Management, Janvier 1996

Pratique de la qualité
La certification ISO 9000

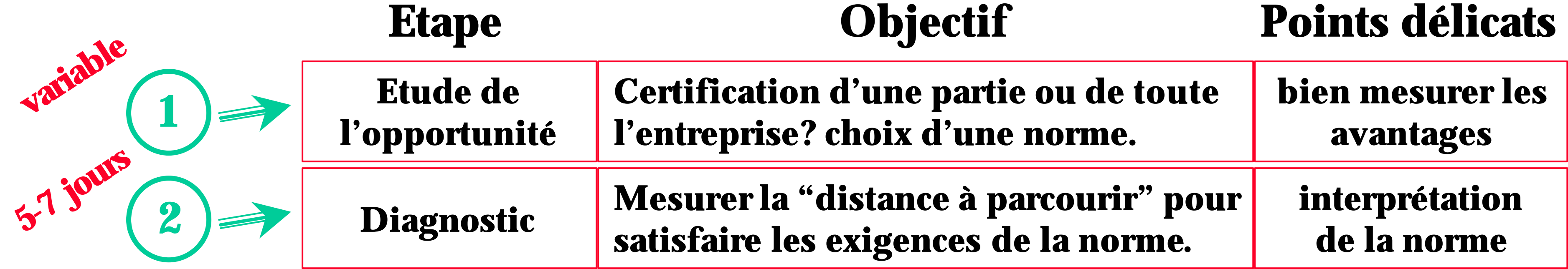
Point à respecter	9001	9002	9003	Objet précis
Maîtrise des enregis-trements relatifs à la qualité	+	+	+	L'entreprise s'assure que toutes les décisions, les actions et les mesures relatives à la qualité font l'objet de documents écrits, approuvés et diffusés.
Audits qualité internes	+	+	+	Des audits internes sont effectués pour vérifier l'état du système qualité et remédier à ses dysfonctionnements.
Formation	+	+	+	Le personnel est professionnel. Ses compétences sont réactualisées par le biais des actions de formation.
Prestations associées	+	+	+	Les prestations annexes, comme les actions de formation, sont maîtrisées par l'entreprise.
Techniques statistiques	+	+	+	Les techniques statistiques utilisées, notamment en matière de contrôles, sont clairement explicitées.

Source : Management, Janvier 1996

Pratique de la qualité
La certification ISO 9000

- Avantages
  - Accès à certains types d'appel d'offres
  - Audit forcé des mécanismes de production
  - Respect d'un "niveau minimum"
  - etc…
- Inconvénients
  - Processus lent (18 mois environ)
  - Cela coûte cher (12 à 17 KF par employé en 1995)

Définition des besoins et choix de la norme



Etape	Objectif	Points délicats
Etude de l'opportunité	Certification d'une partie ou de toute l'entreprise? choix d'une norme.	bien mesurer les avantages
Diagnostic	Mesurer la "distance à parcourir" pour satisfaire les exigences de la norme.	interprétation de la norme

Monter le projet

Etape	Objectif	Points délicats
Plan d'action	Planifier les étapes nécessaires pour élaborer et certifier le système qualité.	choix du "respon-sable qualité"

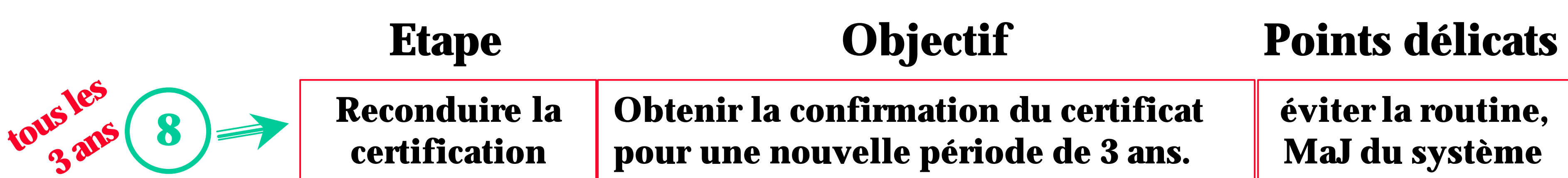
Répondre aux exigences de la norme

	Etape	Objectif	Points délicats
6-12 mois — 4	Conception du système qualité	Elaboration du système qualité (manuel, procédures et instructions).	mise à plat de l'entreprise
6-12 mois — 5	Application	Application du système qualité et mise sous contrôle des points critiques.	rapidement corri-ger les problèmes
2-3 jours — 6	Audit "à blanc"	Evaluation du système qualité (mesure des écarts par rapport à la norme)	

Certification

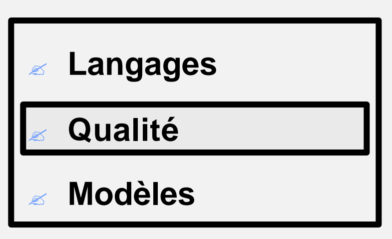
	Etape	Objectif	Points délicats
2-3 mois — 7	Certification	Reconnaissance par un organisme tiers de la conformité du système qualité.	prise en compte de l'audit à blanc

Après... le renouvellement



	Etape	Objectif	Points délicats
tous les 3 ans — 8	Reconduire la certification	Obtenir la confirmation du certificat pour une nouvelle période de 3 ans.	éviter la routine, MaJ du système

Pratique de la qualité
Processus de certification



- Conclusion sur la certification
  - **1-** Bons principes (aspects renouvellement)
  - **2-** Attention au rackett
  - **3-** Attention à l 'application (aspect commercial évident)
  - **4-** Parfois nécessaire (pour réponse aux appels d 'offres)

Pratique de la qualité
Exemple de norme

- Norme documentaire **DOD-STD-2167A**

- Objectifs
  - Présentation à travers d'un exemple des "structures types" de documentation d'un logiciel

Pratique de la qualité
Exemple de norme

Norme DOD-STD-2167A (Software Development Plan)

- 1) couverture
- 2) Page de titre
- 3) Table des matières
- 4) Domaines d'application
- 5) Documents cités en références
- 6) Gestion du développement du logiciel
- 7) Génie Logiciel
- 8) Test de qualification officielle
- 9) Evaluation du produit logiciel
- 10) Gestion des configurations du logiciel
- 11) Autres fonctions de développement du logiciel
- 12) Notes
- 13) Annexes

Utilisée pour planifier un plan de développement avec des sous-contractants

Pratique de la qualité
Exemple de norme

- Page de titre :
  - **Objectif : permettre une classification rapide du document**
    - Numéro de contrôle du document
    - Date
    - Numéro de révision et date
    - Titre
    - Référence du système (<identification>)
    - Référence du contrat (<identification>, <nom du client>)
    - Auteur
    - Authentifié par <donneur d'ordre>, <date>
    - Approuvé par <fournisseur>, <date>

Domaine d'application

- Identification
  - numéro d'identification du projet (éventuellement abréviation)
- Généralités sur le système
  - Rôle du système faisant l'objet du développement
- Généralités sur le document
  - Objectif et résumé du contenu du document

Pratique de la qualité
Exemple de norme

- Liens avec les autres plans
  - plan de gestion des configurations du système
  - plan d'archivage du logiciel
  - plan du programme qualité
  - plan de tests du logiciel
  - plan de gestion technique du système

gestion du développement du logiciel

- Organisation et ressources du projet
  - inventaire des ressources nécessaires (locaux, matériels et logiciels)
  - structure organisationnelle (rôle et responsabilité des acteurs)
  - personnels
- Echéanciers et jalons (activités (Gantt)) et réseau d'activités (Pert).

Pratique de la qualité
Exemple de norme

- Gestion des risques (domaines à risques, facteurs, procédures de contrôle et d'urgence)
- Interface avec les partenaires (contractants associés, sous-traitants)
- Revues formelles

Pratique de la qualité Exemple de norme

- Bibliothèque de développement du logiciel (procédures et méthodes)
- Processus d'actions correctives et rapports d'anomalies

génie logiciel

- Organisation des ressources
  - structure organisationnelle de développement (autorité et responsabilité des partenaires)
  - personnel (titre et qualification, exigences spécifiques)
  - environnement de génie logiciel (outils automatisés et matériel)

Pratique de la qualité
Exemple de norme

- Normes et procédures applicables
  - techniques et méthodologies pour chaque étape du cycle de vie
  - fichiers de développement du logiciel (normes et contraintes, documentation, échéancier, tests et jeux d'essais,...)
- Logiciel non développé
  - inventaire et justification des logiciels du marché et des logiciels réutilisés envisagés

Evaluation du produit logiciel

- Organisation et ressources
- Procédures et outils d'évaluation
- Produits de la sous-traitance
- Enregistrement des évaluations
- Évaluations de produits liées à une activité

Gestion des configurations du logiciel

- Identification et rédaction des caractéristiques fonctionnelles et physique des articles de configuration
  - organisation et ressources
  - identification de configuration (documents pour le référentiel)
  - procédures de maîtrise des configurations
  - suivi des états des configurations
  - audits de configuration
  - jalons de la gestion des configurations

Plan du cours

- **I.** Introduction
- II. Éléments de terminologie
- **III**. Environnements de programmation
- **IV.** Les langages de programmation
- **V.** La Qualité
- **?** **VI. Modèles de développement de logiciels**
- **VII.** Gestion de projets

Modèles de développement

- **Spécificité du processus de production :**
  - reproduction = copie
- **Intérêt :** analyse des besoins, conception, validation
- **Développement :**
  - descriptions de plus en plus précises et proches d'un programme exécutable : raffinements
  - itératif : retour sur la conception après test
  - invisible : importance de la documentation

Modèles de développement

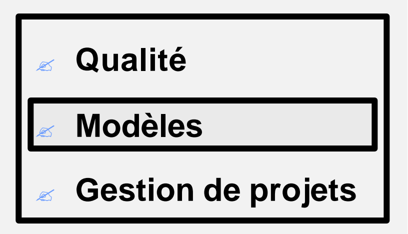
- **Maintenance :** remise en cause du développement
- ***Cycle de vie du logiciel*** =développement + exploitation + maintenance

D'où : modèles de développement

Modèles de développement

- **Premiers modèles de développement :**
  - programmation + traque d'erreurs
  - fin des années 60 : développement en étapes
- **Distinction entre activités et étapes**
  - Exemples :
    - étape de conception = activités de spécification, maquettage et validation
    - activité de documentation : sur plusieurs étapes

Modèles de développement



- **But des modèles :**
  obtenir un processus de développement rationnel, reproductible et contrôlable

- = vœu pieux ... dépendant de la bonne volonté des concepteurs

- Les activités
  - Utilisation et production de documents
  - Principales activités (globales)
    - Analyse des besoins
    - Spécification globale
    - Conception architecturale détaillée
    - Programmation
    - Gestion de configuration et intégration
    - Validation et vérifications
    - Maquettage et prototypage

Modèles de développement. Les activités

- Analyse des besoins
  - **But :** éviter de développer un logiciel inadéquat
  - **Étude :**
    - du domaine d'application,
    - des ressources,
    - des contraintes de coût et d'utilisation
  - **Données :**
    - fournies par des experts et des utilisateurs
    - entretiens, questionnaires, études, etc

Modèles de développement. Les activités

Spécification globale

- **But :**
  - établir une première spécification du système
- **Données :**
  - résultats de l'analyse des besoins
  - faisabilité informatique
- **Résultat :**
  - description de ce que fait le logiciel
  - point de départ du développement

Forte corrélation avec l'analyse des besoins

Modèles de dévelop-pement. Les activités

- Conception architecturale détaillée
  - Conception architecturale :
    - décomposer le logiciel
    - interfaces et fonctions des composants
  - Conception détaillée :
    - algorithmes
    - représentation des données

Modèles de dévelop-pement. Les activités

Programmation

- Passage de la conception détaillée à un ensemble de programmes

Activité	Temps
programmation	15 - 20 %
spécification et conception	40 %
validation et vérification	40 %

Gestion de configuration et intégration

- **Gestion des composants du logiciel :**
  - environnements intégrés, gestion homogène de programmes et de documentation
- **Intégration :**
  - assemblage pour produire un exécutable
  - variantes d'exécutables

- **Validation et vérification**
  - **Validation** = adéquation
    analyse / spécification
    - "A-t-on écrit un système qui répond à l'attente des utilisateurs ?"
  - **Vérification** = adéquation
    spécification / raffinement
    - "Le développement est-il correct par rapport à la spécification de départ ?"

Modèles de dévelop-pement. Les activités

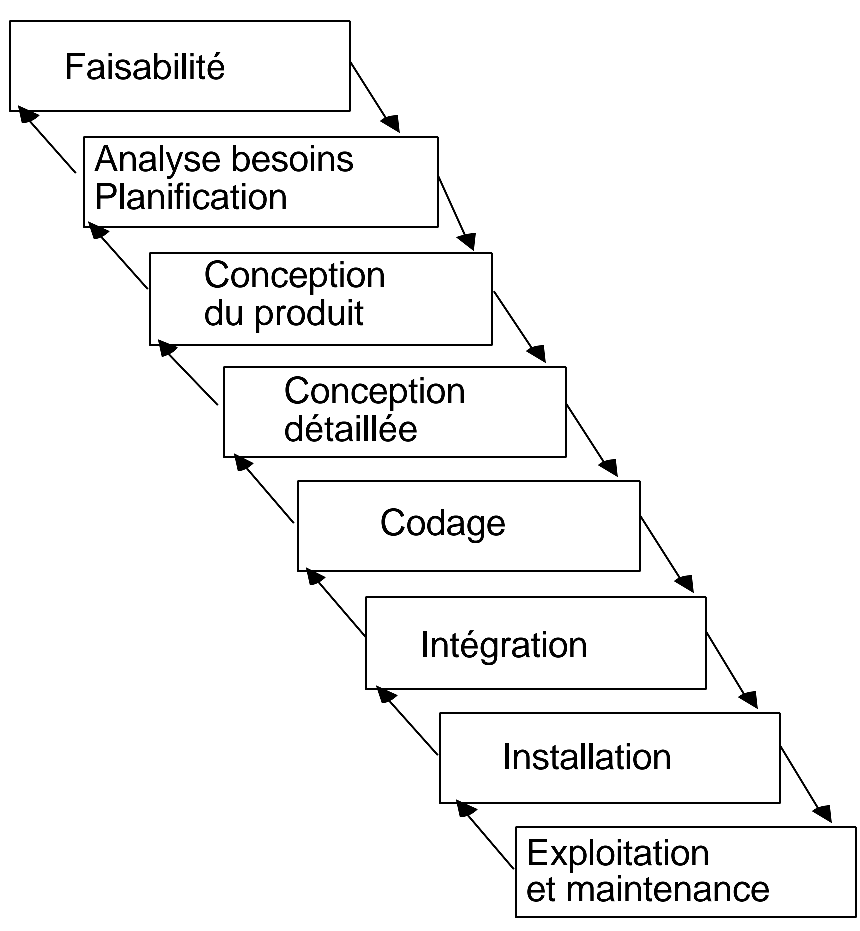
Validation et vérification

- **Validation :**
  - revues de spécifications ou manuels,
  - prototypage rapide
- **Vérification :**
  - revues de spécification ou programmes,
  - preuves et tests

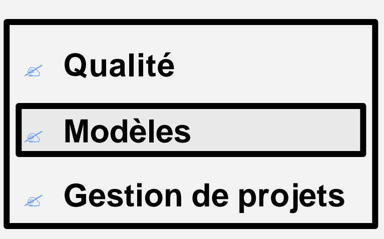
Modèles de dévelop-pement. Les activités

- Le **maquettage** et le **prototypage**
  - Imprécision des besoins
- maquettage, ou *prototypage rapide* = ébauche du futur système
  - **Maquettage exploratoire :**
    - test avec les utilisateurs
  - **Maquettage expérimental :**
    - pendant la phase de conception

Modèle en cascade



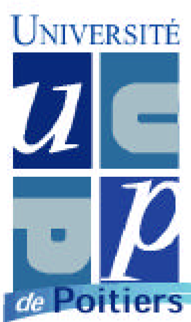
Modèle en cascade



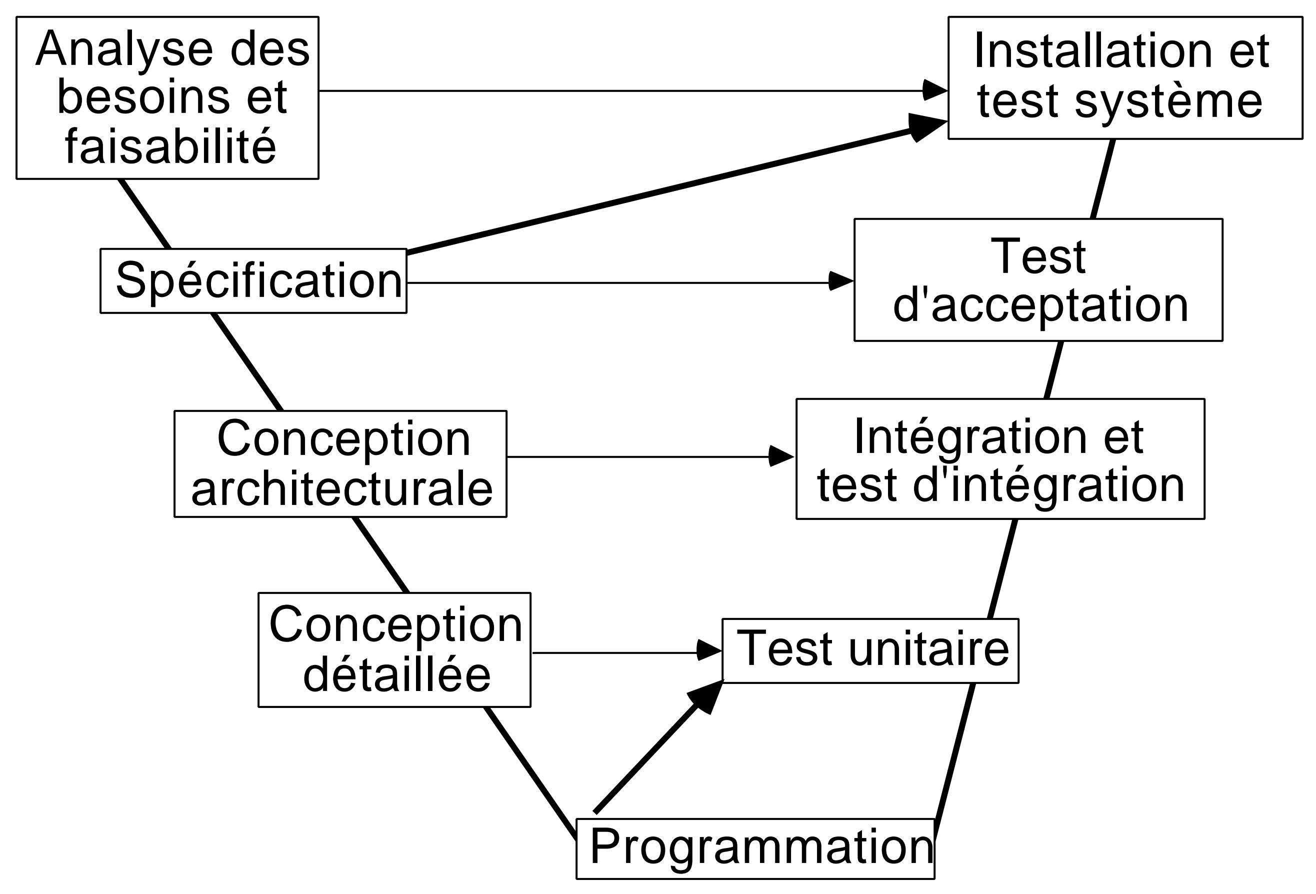
- Développement quasi-séquentiel
- Versions récentes : validation - vérification
- Chaque phase se termine par une activité V & V
- Retours arrière à la phase précédente possibles
- Modèle idéal
- Petite modification du cahier des charges implique grosses modifications.

Modèle en cascade

Analyse des besoins	Cahier des charges - Besoins - Contexte - Exigences	- Spécifications générales	- Identification - Fonctionnalités - Contraintes (N.F)
Spécification	Spécifications générales - Contexte	Spécifications - Fonctions - Contraintes	- Conversion des besoins en fonctions - Sélection des contraintes N.F.
Conception préliminaire	- Spécifications générales - Réutilisation - Contexte général - Systèmes semblables	Description de la structure (hard et Soft) su système.	- Mise à jour de la structure - Identifications des composants et de leurs relations
Conception détaillée	- Description architecture - Détails sur l'environnement de programmation	Structures des logiciels (graphes de programmes)	- Abstraction - Elaboration - Choix
Implémentation	Graphes	- Codes - Documents - Données + fichiers + programmes + manuels	- Codage - Test unitaire, mise au point - Intégration
Maintenance	- Documentation - Codes - Spécifications	Système amélioré	- Correction - Re-conception - Re-programmation



Modèle en V



Modèle en V

—— Enchaînement des opérations ou itérations

⟶ Une partie de résultats de départ est utilisée par l'étape d'arrivée

Modèle en V

- Dépendances séquentielles : V
- Dépendances fonctionnelles :
  - *Exemple* : conception architecturale
  - et protocole d'intégration
  - + jeux de test d'intégration

Modèle en spirale

- Qualité
- **Modèles**
- Gestion de projets

Modèle en Spirale

- 1- Détermination des objectifs, des alternatives pour les atteindre.
- 2- Analyse des risques, évaluation des alternatives, éventuellement maquettage.
- 3- Développement et vérification de la solution retenue.
- 4- Revue des résultats et planification du cycle suivant.

Risques majeurs

- Défaillance de personnel
- Calendrier et irréalistes
- Développement de fonctions inappropriées
- Développement d'interfaces utilisateurs inappropriées
- Produit « plaqué or »
- Volatilité des besoins
- Composants externes manquants
- Tâches externes défaillantes
- Problèmes de performances
- Exigences démesurées par rapport à la technologie

Modèles incrémentaux

- Développement d'un noyau
- Développements successifs autour du noyau
- **Avantages :**
  - chaque développement est moins complexe
  - intégrations progressives
  - livraison et mise en service après chaque incrément
  - efforts concernant les diverses étapes plus homogènes

Modèles incrémentaux

- **Utilisation :**
  - grands projets, fonctionnant par appels d'offre et sous-traitance
- **Risque :**
  - remise en cause du noyau

Modèles dynamiques de développement

- Modèles statique = cascade, V, ...
- Insuffisances des modèles statiques car les logiciels évoluent
- L'activité de V & V n'est pas rigoureuse
- Une modification mineure du cahier des charges entraîne une modification majeure des programmes
- Les changements sont permanents
- La complexité des applications est croissante
- Les programmes et la programmation évoluent

Modèles dynamiques de développement

- **Modèle évolutif par prototypage rapide**
  - cette étape est suivie d'une évaluation par l'utilisateur
  - puis transcodage

- **Inconvénient :**
  - Un prototype ne met en évidence que les aspects fonctionnels.

- **Modèle transformationnel et incrémental**
  - un concept est abstrait en une spécification structurée et/ou formelle qui est transformée par addition de propriétés non fonctionnelles.

  - **Inconvénient** :
  - Cette approche n'est vraiment intéressante que pour les systèmes nouveaux,
  - mais possibilité de rétro-conception.

Quelques modèles de développement

Cycle de vie de Balzer 1983

Quelques modèles de développement

Expression informelle des besoins

Analyse des besoins

Spécification formelle prototype

Validation

Maintenance

Décisions

Optimisations mécaniques

Développement formel

Programme source

Tuning

Cycle de vie de Balzer 1983

Quelques modèles de développement

Expression des besoins
Informels et/ou incomplets

Développement d 'une spécification formelle

- Validation de la spécification vis-à-vis des besoins
-Révisions des spécifications pour satisfaire les besoins

Expression des besoins
Informels et/ou incomplets

Maintenance du système par modification de la spécification formelle

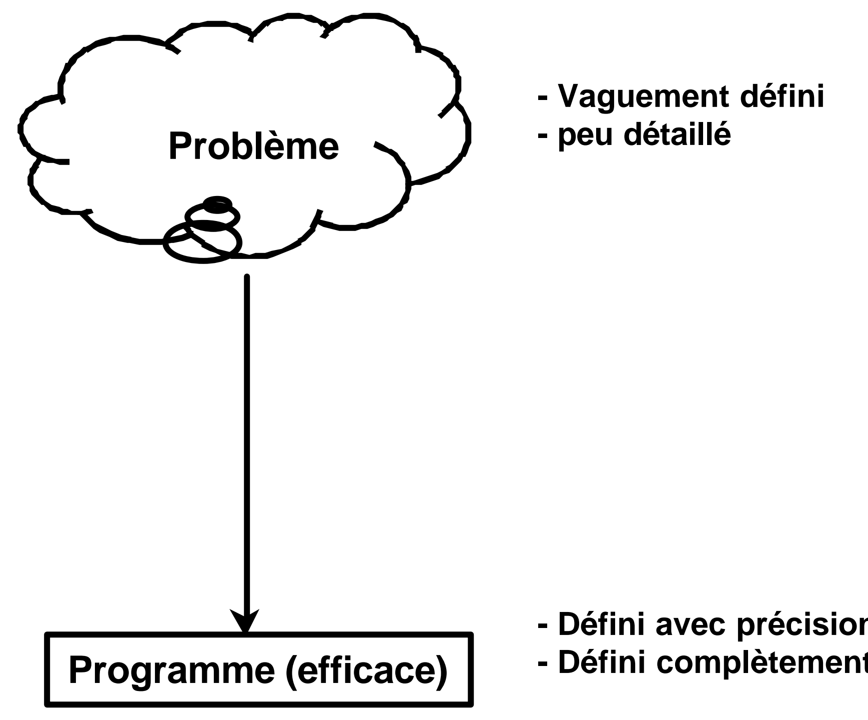
- Transformations simples
  . changement des représentations
  . sélection des algorithmes
  . optimisation
  . compilation
-Utilisation d 'un système automatique pour appliquer les transformations
- Utilisation d 'un système (expert) pour guider le choix des transformations

-Transformation de la spécification en un système concret

Expression des besoins
Informels et/ou incomplets

Modèle d 'Agresti 1986

Pratiques de développement



Pratiques de développement

Problème

- Vaguement défini
- peu détaillé

Analyse des besoins

Spécification rigoureuse

- Défini avec précision
- Peu détaillé

Développement de programmes

Programme (efficace)

- Défini avec précision
- Complètement détaillé

Pratiques de développement

Problème
- Vaguement défini
- peu détaillé

Analyse des besoins

Spécification rigoureuse
- Défini avec précision
- Peu détaillé

Introduction de détails par décisions d 'implémentation

Développement de programmes

Programme (efficace)
- Défini avec précision
- Complètement détaillé

Pratiques de développement

- Qualité
- Gestion de projets

Quelques modèles de développement

Problème
- Vaguement défini
- peu détaillé

Analyse des besoins

Spécification rigoureuse
- Défini avec précision
- Peu détaillé

Développement de programmes

- Programmation par transformation
  . Formellement vérifiées
  . Transitions qui préservent la sémantique

Introduction de détails par décisions d 'implémentation

Programme (efficace)
- Défini avec précision
- Complètement détaillé

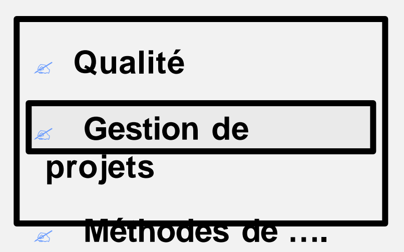
Plan du cours

- **I.** Introduction
- II. Éléments de terminologie
- **III**. Environnements de programmation
- **IV.** Les langages de programmation
- **V.** La Qualité
- **VI.** Modèles de développement de logiciels
- **?** **VII. Gestion de projets**

Gestion de projets

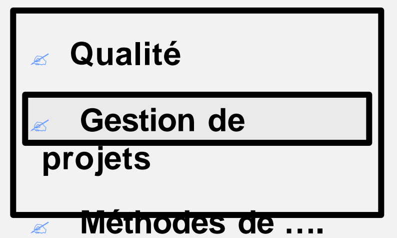
- OBJECTIFS:
  - Introduction de l'activité et de ses caractéristiques
  - Décrire et discuter l'activité de planification
  - Montrer l'utilisation de représentations graphiques dans cette activité
  - Description de l'activité : on apprend que sur le terrain (stages, emploi, …).

Gestion de projets
Définitions



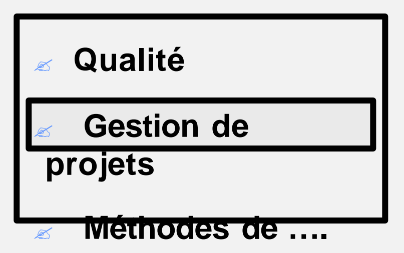
- Organisation, planification et établissement des échéances d'un projet (logiciel)
- Regroupement des activités assurant que
  - le logiciel est livré dans les temps
  - en accord avec les exigences des organismes impliqués
- Activité importante
  - le GL est une activité économique (qui donc implique des contraintes non techniques)
  - Les projets bien gérés échouent parfois, les projets mal gérés échouent toujours

Gestion de projets
Les activités



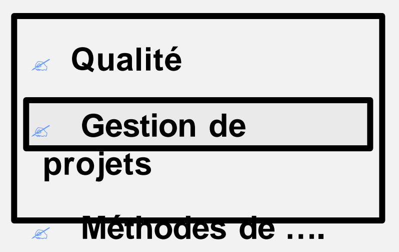
- Rédaction de propositions
- Evaluation du coût des projets
- Planification et construction de l'échancier du projet
- Pilotage et révision (contrôle des évolutions) du projet
- Recrutement et évaluation du personnel
- Rédaction de rapports et préparation de présentations
- Observations sur ces activités
  - **Elles ne sont pas propres à la gestion de projets logiciels**
  - **Les techniques d'ingénierie classique sont applicables au Génie logiciel et vice-versa**
  - **Les projets complexes (bâtiment etc) souffrent de problèmes similaires à ceux du Génie Logiciel**

Gestion de projets
Recrutement



- Problème: il est souvent difficile de recruter le personnel “idéal” sur un projet
  - Budget: impossible de payer les gens à la hauteur de leur qualification
  - Indisponibilité: impossible de trouver des gens ayant l’expérience appropriée sur le marché
  - Stratégie: l’organisation peur souhaiter développer un savoir-faire propre sur un projet logiciel

Gestion de projets
Planification



- Probablement l'une des activités les plus chronophages
- Activité continue depuis le démarrage du projet jusqu'à la mise à disposition du produit
  - Les planning doivent être régulièrement réactualisés
    - Nouvelles contraintes
    - Nouvelles données

Gestion de projets
Plans

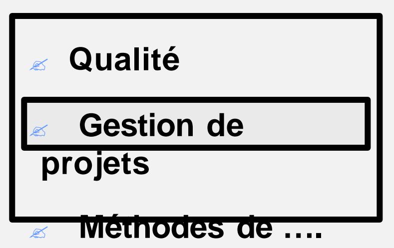
Plan	Description
Plan qualité	décrit les procédures et standards mis en œuvre pour assurer la qualité du logiciel
Plan de validation	décrit l'approche, les ressources, les procédures et les échéances (tests, recettes) relatives à la validation du système
Plan de gestion de la configuration	Décrit les procédures de gestion de la configuration
Plan de maintenance	Décrit et prévoit les besoins de maintenance du système, les coûts et les efforts requis
Plan de formation et de développement	Décrit comment le savoir faire et l'expérience des ingénieurs seront développés

D'après I.Sommerville ©1995

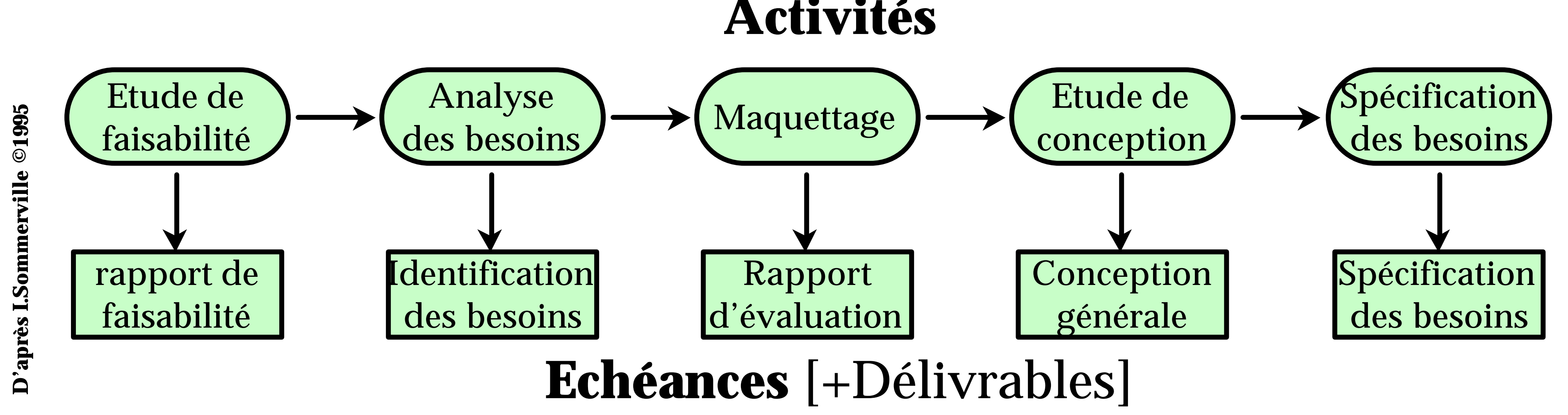
Gestion de projets
Processus de planification

Définir les contraintes qui pèsent sur le projet
Effectuer une première estimation des paramètres du projet (degrés de liberté)
Etablir des échéances et des délivrables
TANT QUE le projet n'est pas fini ou annulé FAIRE
    Etablir un planning du projet
    Démarrer les activités en fonction de ce planning
    ATTENDRE (durée déterminée)
    Faire une revue d'avancement du projet
    Re-estimer les paramètres du projet
    Appliquer ces révisions au planning du projet
    Re-négocier les contraintes et les délivrables (s'il y a lieu)
    SI problème ALORS
        démarrer une revue technique et une éventuelle révision
    FSI
FTQ

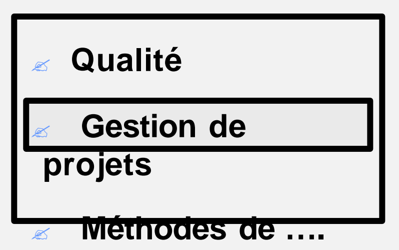
Gestion de projets
Organisation des activités



- Critères: L'organisation doit être effectuée en vue de produire des résultats tangibles du point de vue de l'évaluation
  - les échéances marquent la fin d'une activité
  - les délivrables sont des produits délivrés aux “clients”



Gestion de projets
Planification d 'un projet



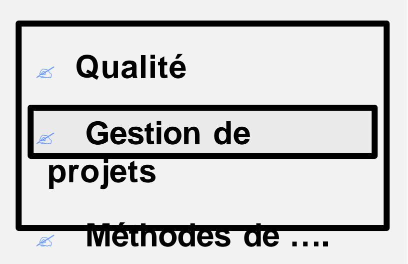
- Division du projet en tâches séparées + estimation :
  - des ressources requises pour les mener à bien
  - de la durée nécessaire pour les accomplir
- Organiser les tâches en parallèle afin d'optimiser la puissance de travail de l'équipe
- Minimiser la dépendance entre tâches afin de limiter le nombre de tâches critiques susceptibles de retarder le projet

Dépend de l'intuition et de l'expérience du chef de projet

Gestion de projets
Problèmes de la planification

- Estimation de la complexité d'un problème et du coût du développement de sa solution
- La productivité n'est pas proportionnelle à la taille de l'équipe
- Ajouter du personnel à un projet en retard risque fort d'engendrer un retard supplémentaire du à un surcroît de communications
- L'inattendu arrive toujours, il faut donc savoir le planifier (et ménager de la marge)

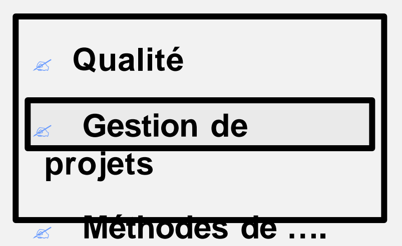
Gestion de projets Diagrammes et graphes d 'activités



- Notations graphiques "classiques" utilisées pour la planification d'un projet

- Un graphe d'activité indique:
  - les inter-dépendances entre tâches dans le projet:
    - les tâches ne doivent pas être trop courtes
    - ordre de grandeur: de quelques jours à deux semaines
  - le chemin critique entre les tâches

Gestion de projets Diagrammes et graphes d 'activités



- Le diagramme indique:
  - les responsables des tâches
  - les dates de début et fin de ces tâches
    - estimées (a priori)
    - réelles (mesurées)

Gestion de projets

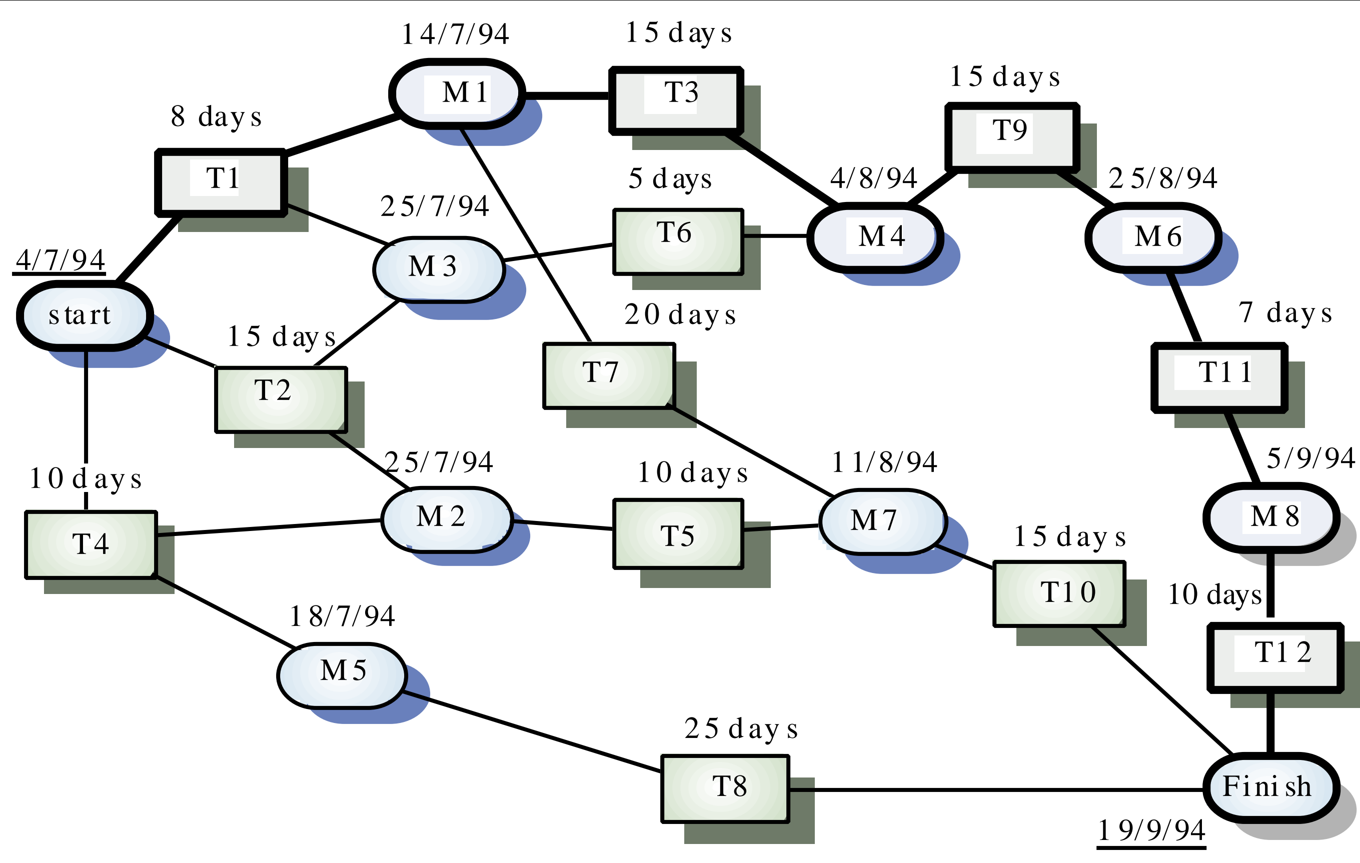
Exemple : durée et dépendances entre tâches

Task	Duration (days)	Dependencies
T1	8	
T2	15	
T3	15	T1
T4	10	
T5	10	T2, T4
T6	5	T1, T2
T7	20	T1
T8	25	T4
T9	15	T3, T6
T10	15	T5, T7
T11	7	T9
T12	10	T11

D'après I.Sommerville ©1995

Gestion de projets
Graphe des activités

- Qualité
- Gestion de projets
- Méthodes de ….

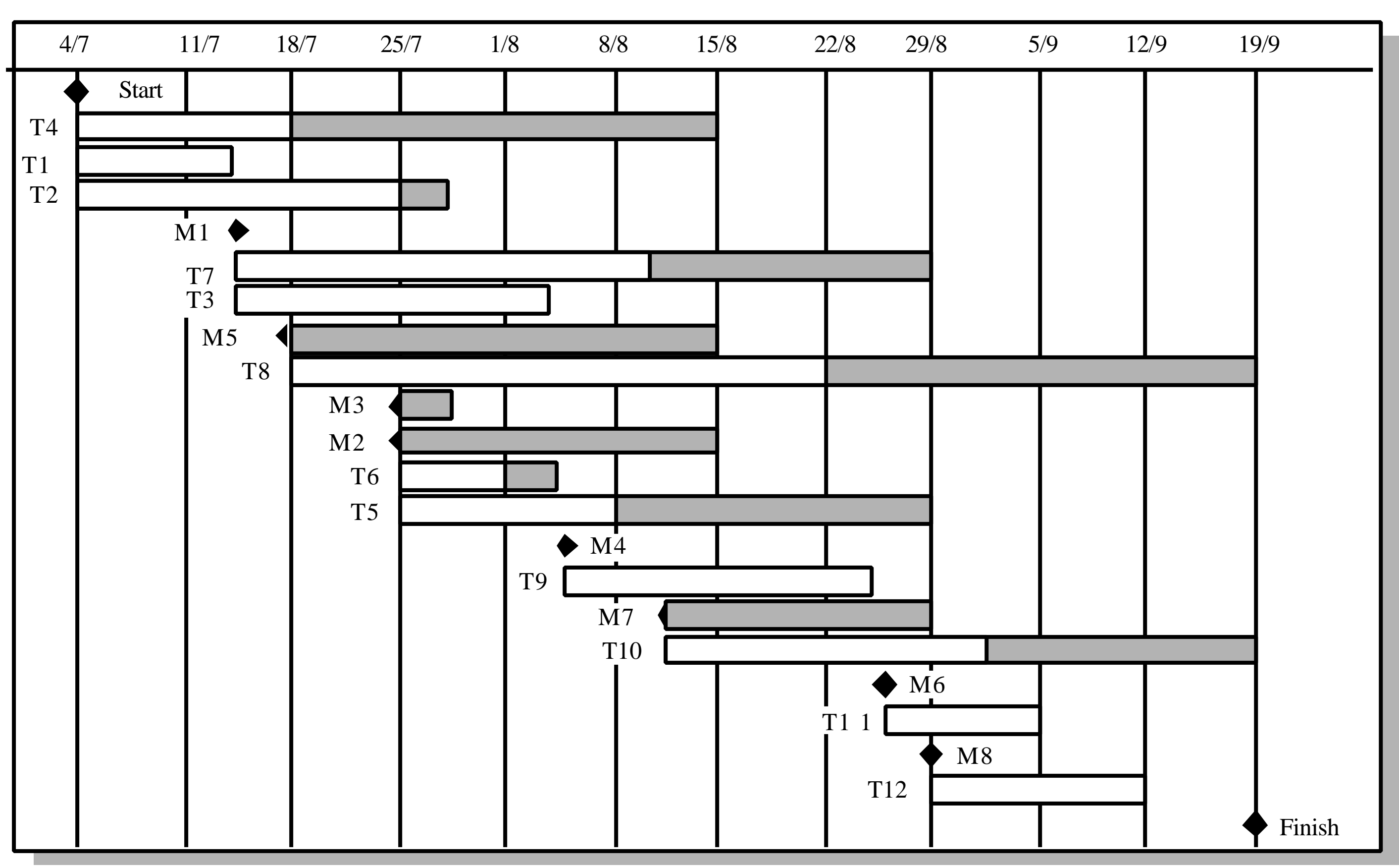


D'après I.Sommerville ©1995

Gros projets => hiérarchisation de stâches

Gestion de projets
Diagramme d 'activités

- Qualité
- **Gestion de projets**
- Méthodes de ….



D'après I.Sommerville ©1995

Gestion de projets

Affectation de personnels

4/7 11/7 18/7 25/ 1/8 8/8 15/8 22/8 29/8 5/9 12/9 19/9

Fred
T4
T8
T11
T12

Jane
T1
T3
T9

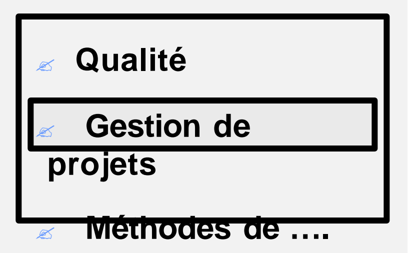
Anne
T2
T6
T10

Jim
T7

Mary
T5

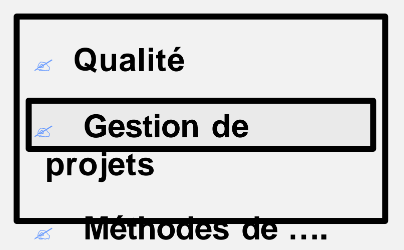
D'après I.Sommerville ©1995

Gestion de projets
Points clefs



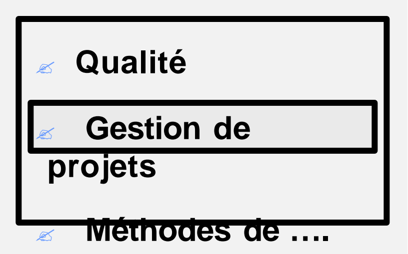
- Une bonne gestion de projet est essentielle pour réussir
- La nature du logiciel pose des problèmes particuliers de gestion
- Les chefs de projet ont différents rôles mais le plus important consiste en la planification, l’estimation et la mise en place d’échéances

Gestion de projets
Points clefs



- Planification et estimation sont des activités itératives et continues pendant toute la durée du projet
- Une échéance est une date “prévisible” pour la présentation d’un rapport à la hiérarchie
- Utilisation de techniques graphiques pour faciliter l’activité d’évaluation

Gestion de projets
Etre chef de projet !



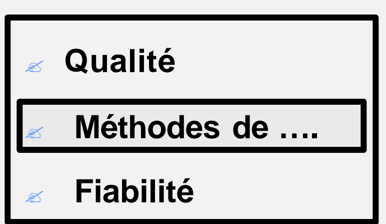
- Difficile
  - Compréhension des aspects techniques du projet (surtout s'il est vaste)
  - Rapport à la hiérarchie (gestion des délais, coûts, évaluations diverses etc.)
  - Beaucoup de "paperasse"

Gestion de projets
Etre chef de projet !

- Comment s'en sortir
  - Être un bon "public relation"
    - Maintenir la cohésion d'une équipe
  - Savoir se faire "respecter"
  - Avoir une bonne culture informatique
    - "sentir" les bonnes solutions
    - faire des choix techniques sans appréhender un problème dans les détails

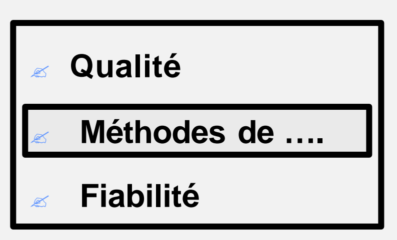
Plan du cours

- ? **VIII. Méthodes de conception de logiciels**
- **IX.** Fiabilité du logiciel
- **X.** Test du logiciel
- **XI.** Gestion des versions
- **XII.** Réutilisation de logiciels
- **XIII.** Maintenance de logiciels
- **XIV.** Introduction aux méthodes formelles



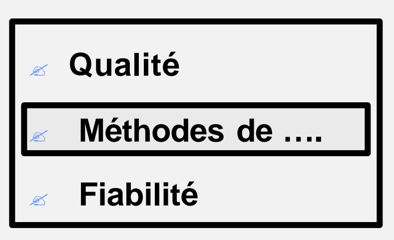
- 1- Énoncés informels
- Spécifications informelles.
  - Risques
    - incohérence,
    - ambiguïté,
    - incomplétude,
    - désorganisation,
    - redondance

- Présentations formatées
  - Dictionnaire de données
    - définition des termes, sigles, codes, symboles utilisés
    - informations sur les fichiers qui contiennent les données et les processus qui les utilisent



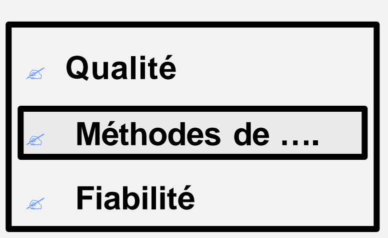
- **Table de décision :**
  - table de traitements exhaustifs
  - adaptées aux cas où les sorties sont directement déterminés par les entrées
- **Table états-transitions :**
  - tables avec les états du système, les transitions possibles
  - adaptées aux cas où les sorties sont déterminées par les entrées et les états

Méthodes de conception
Techniques de spécification



- Techniques graphiques ou semi-formelles
  - Les plus courantes
  - Utilisées comme documents de conception
  - Appréhendent souvent l'architecture
  - Intérêts pour les appels d'offres et les réponses à ces appels d'offres.
  - Exemples
    - Modèle entité-association
    - Diagrammes de flots de données

Méthodes de conception
Modèles de données



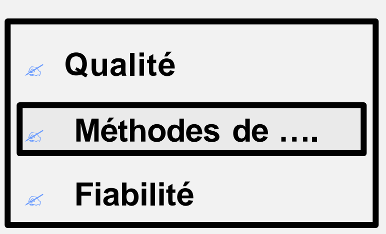
- Entité/association
  - Description de modèles de données
  - Utilisé pour la conception de bases de données, notamment relationnelles
  - Basé sur la description d'entités et d'associations entre entités

- Entité
  - Nom, classe,…
  - Propriétés, attributs,
  - Domaine de valeurs, ensemble de valeurs, types, occurrences, instances
  - Clé : propriété particulière permettant l 'identification d 'une unique occurrence

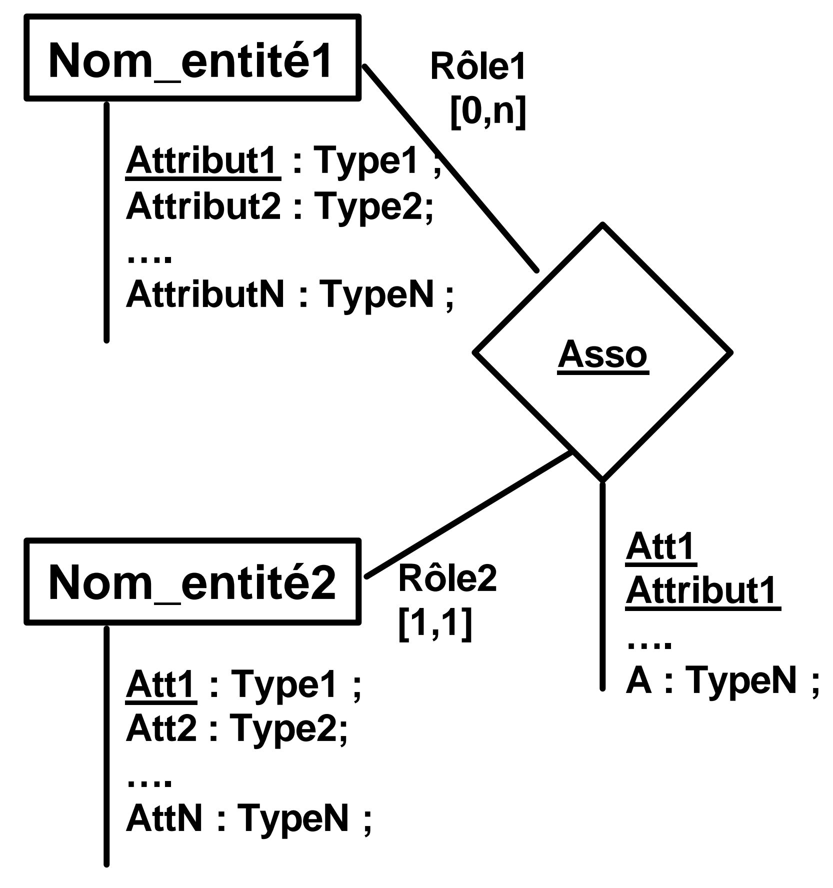
Nom_entité

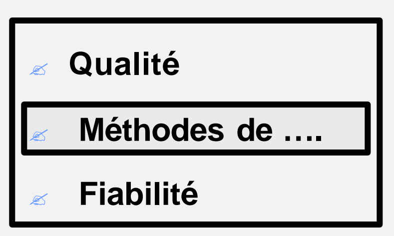
<u>Attribut1</u> : Type1 ;
Attribut2 : Type2;
….
AttributN : TypeN ;

Méthodes de conception
Modèles de données



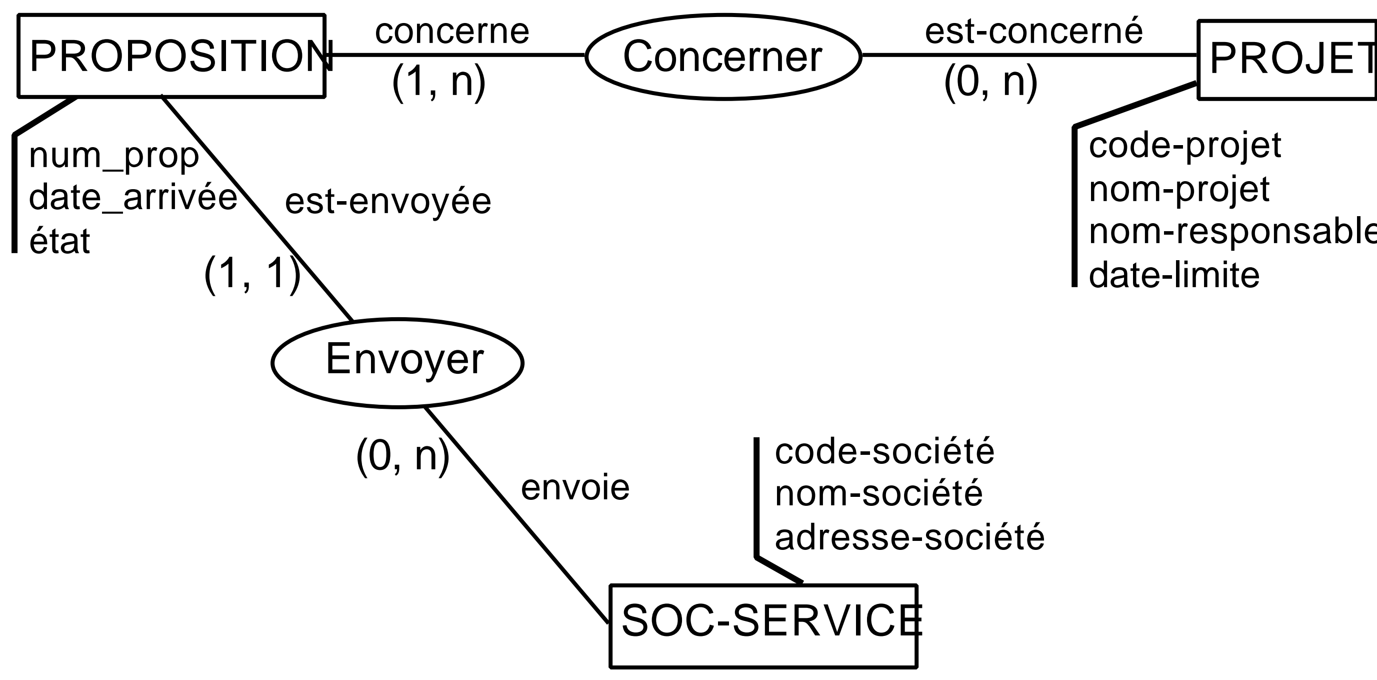
- Association
  - Verbe, flux ou échange d'information entre entités
  - Propriétés, attributs,
  - Rôle : effet de la présence d 'une entité dans une association.
  - Cardinalités : mesure d 'un rôle. **[0,1], [0,n], [1,1],[1,n]**
  - Clé : clés des entités reliées

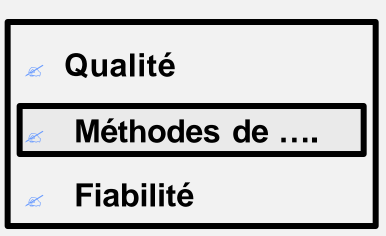




- **Exemple 1 :** une entreprise fournit des produits en grande quantité et un produit est fourni par une seule entre prise.

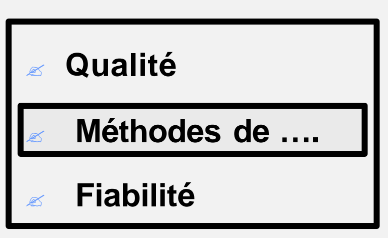
Schéma entité - association



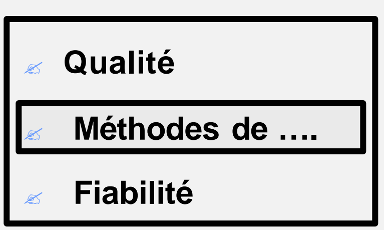


- **Exemple 2** : Étendre le schéma précédent pour prendre en compte les clients.

Méthodes de conception
Modèles de données



- Modèle : ensemble des entités et des associations.
  - Notion de chemin et de connexité
  - Contraintes d'intégrité
- Inconvénient :
  - Ne fait pas apparaître les traitements, les modules, l 'architecture, etc



- **Exercice** :
  - 1- Proposer un modèle entité/association
  - 2- Déterminer les traitements qui peuvent être effectués à travers ce modèle.

L 'agence de tourisme d 'une ville souhaite gérer le système d 'informations relatif à la gastronomie en répertoriant les restaurants. Les restaurants sont repérés avec tous les vins et les plats qu 'ils proposent.
Chaque plat, restaurant et vin est décrit par ses propriétés caractéristiques. Les touristes doivent pouvoir choisir un restaurant, un plat ou un vin en fonction du lieu, du prix et des horaires d 'ouverture.

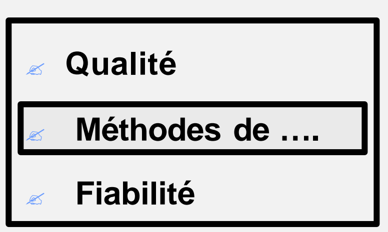
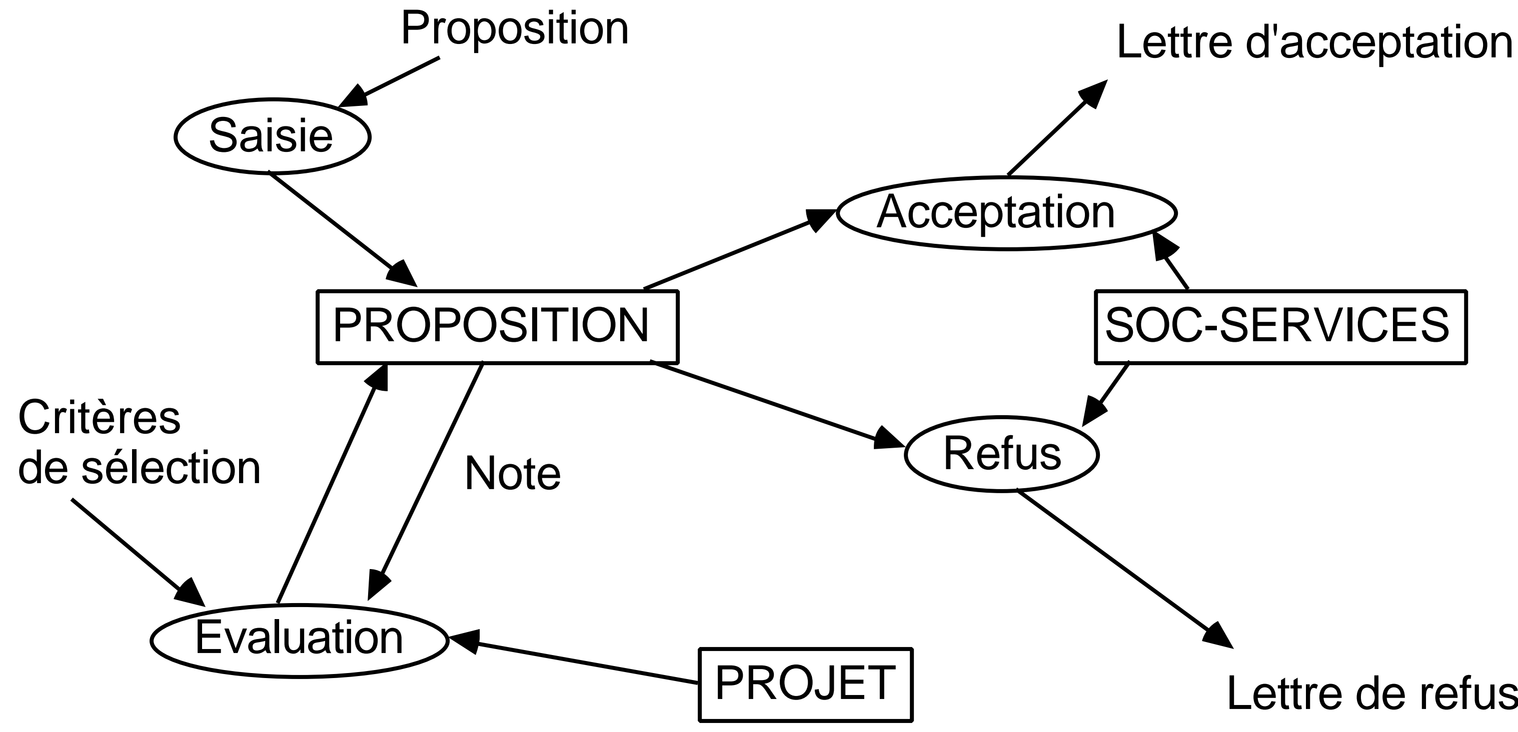
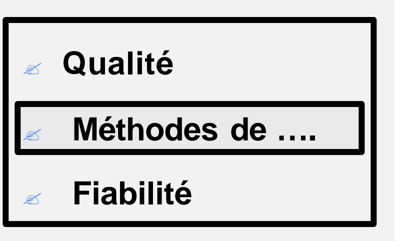
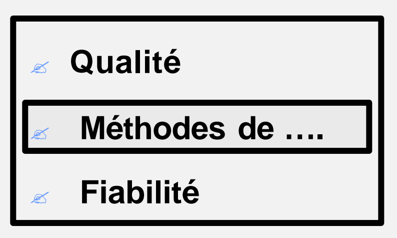


Diagramme de flots de données

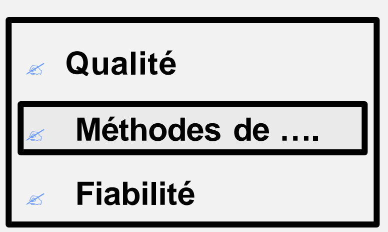




- Exercice : proposer un diagramme flots de données pour les entreprises, produits et clients

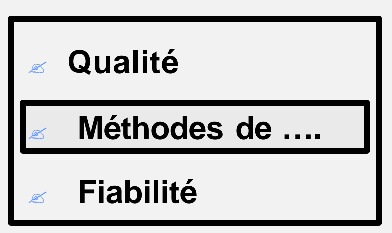


- Exercice : proposer un diagramme flots de données pour la gastronomie en ville.

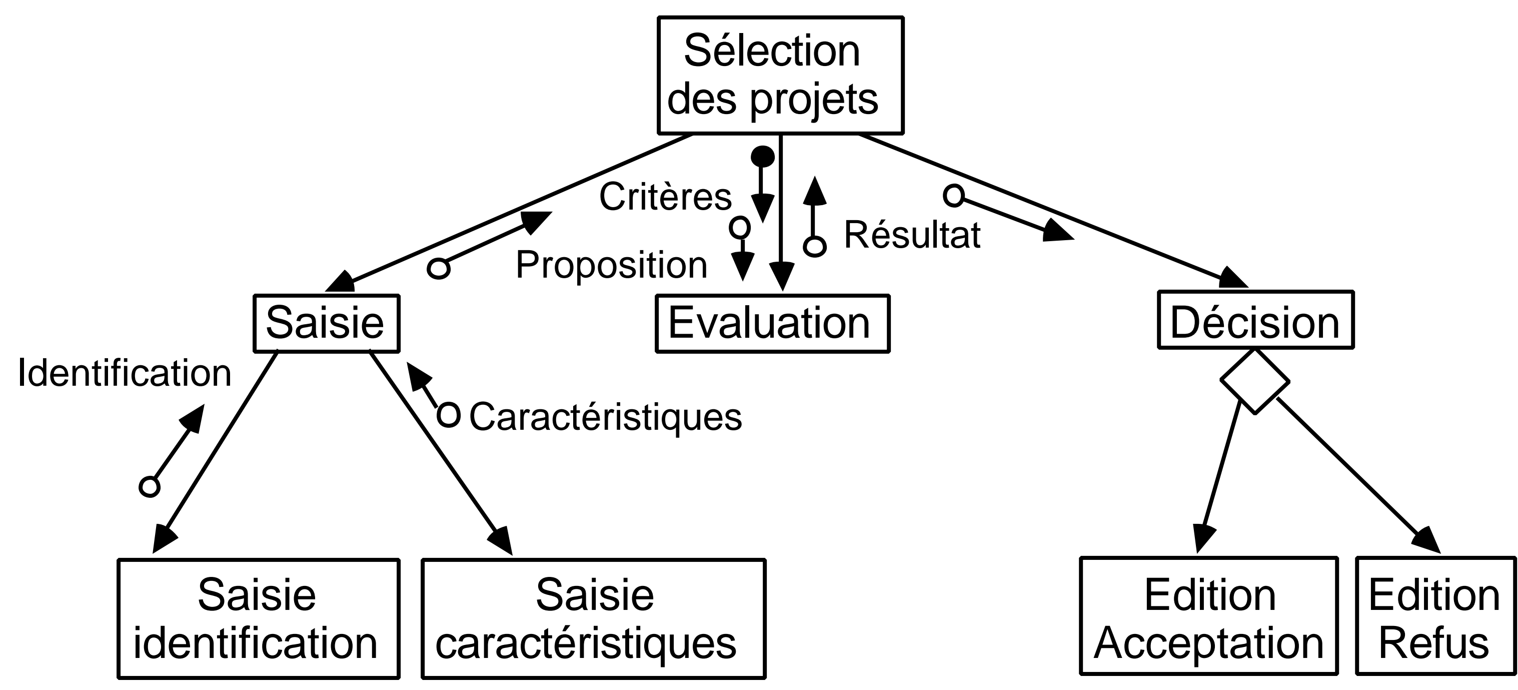


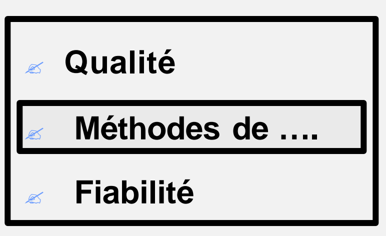
Diagrammes de structure

- description arborescente de la structure d'un système
- flèches entre deux fonctions
- distinction entre les paramètres de contrôle et les paramètres traités

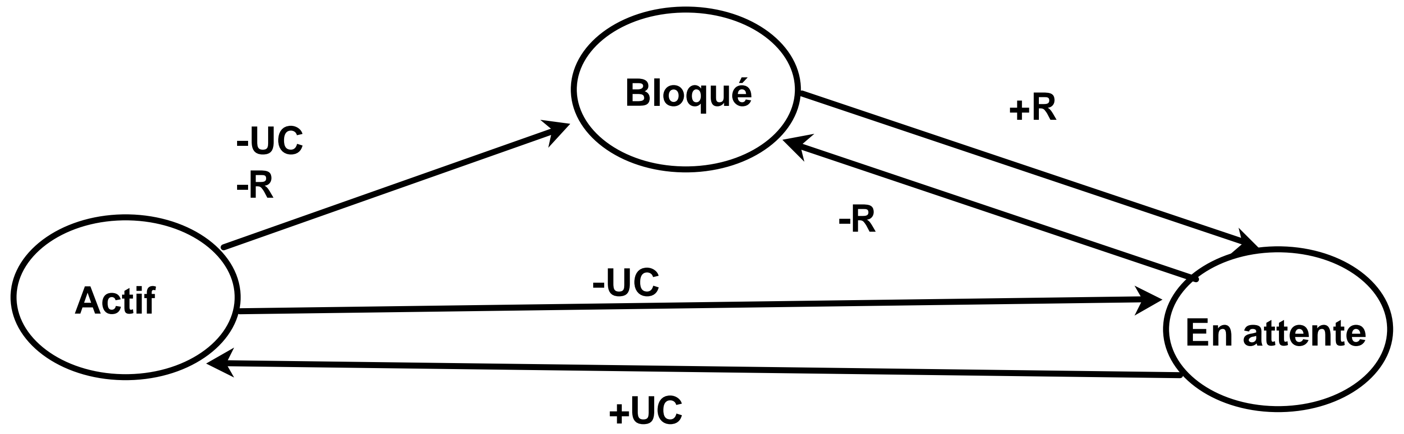


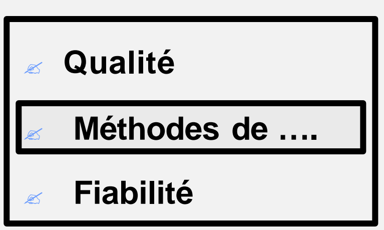
✍ **Exemple :** structure de la sélection des projets





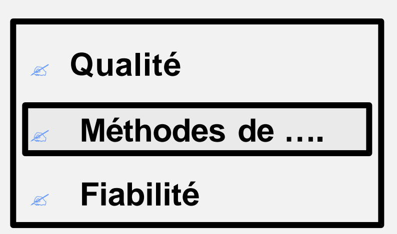
- **Diagrammes états-transitions**
  - utiles pour modéliser le cycle de vie d'un objet
- **Exemple :** état d'un processus
  - Bloqué, en attente, actif





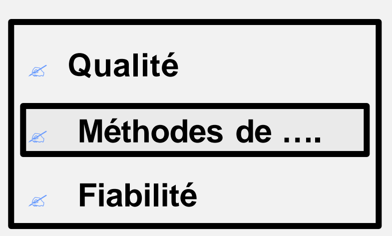
Exercices :

- déterminer le diagramme états-transition pour un feu de circulation
- déterminer un diagramme états-transition correspondant à une mémoire partagée.



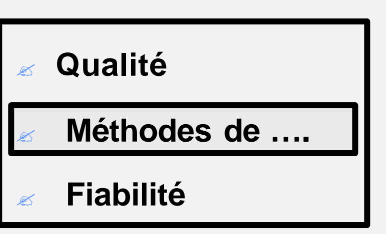
- **Autres systèmes :** modélisation dynamique d'un système a événements discrets
  - Systèmes de transitions
  - Réseaux de Petri

Méthodes de conception
Méthodes d 'analyse

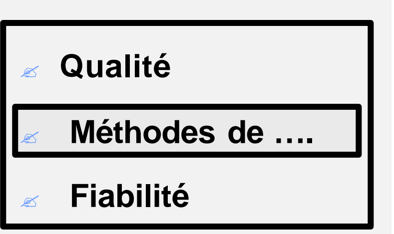


- Catégories possibles :
  - descendantes / ascendantes
  - fonctionnelles (dirigées par les traitements)
  - orientées objets

Méthodes de conception
Méthodes fonctionnelles



- Méthodes fonctionnelles
  - Origine : langages procéduraux
- Orientées traitements
- Approche hiérarchique, descendante, modulaire

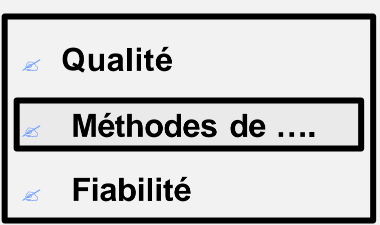


Méthode SADT

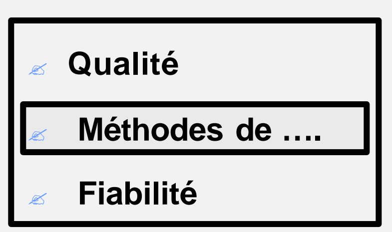
- = "Structured Analysis and Design Technique"
- Concerne la 1ère partie du cycle de vie
- Suite cohérente et hiérarchisée de diagrammes
  - *datagramme* : représente les données par des boîtes et les activités par des flèches
  - *actigramme* : décrit les activités par des boîtes et les données par des flèches

Analyse structurée

- Méthode descendante par raffinements successifs des traitements :
  - utilisation de DFD à chaque niveau
  - 1er DFD = diagramme de contexte
  - cohérence des flots entre les niveaux

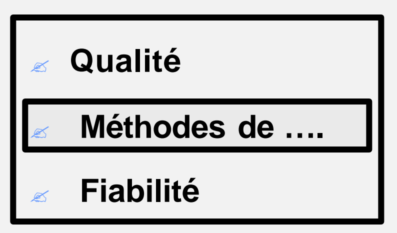


- **Conception structurée**
  - = analyse structurée+ types de liens
  - (simples, conditionnels, itératifs)



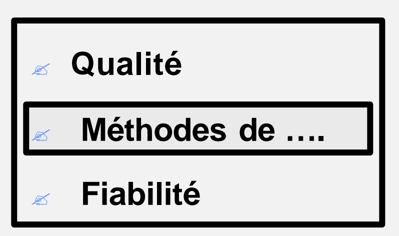
- **Analyse et conception "temps réel"**
  - = analyse structurée + diagrammes de flots de contrôle + spécifications de contrôle, pour décrire l'état des processus (activés ou non)

Méthodes de conception
Méthodes orientées objets

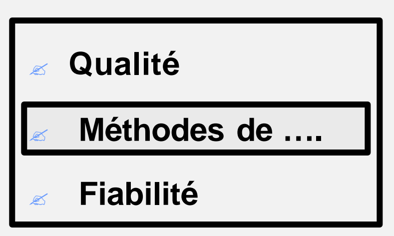


- Basées sur les mêmes concepts que les langages à objets
- Trois aspects pour toutes les méthodes :
  - *statique* ou *descriptif* : propriétés des objets, liens avec les autres objets
  - *dynamique* : comportement des objets, changements d'états
  - *fonctionnel* : fonctions réalisées par les objets par l'intermédiaire des méthodes

Quelques méthodes de conception OO

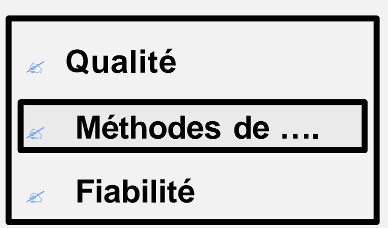


- Quelques méthodes souple, largement applicable
  - **Méthode Coad et Yourdon**
  - **Méthode Grady Booch**
  - **Méthode Shlaer et Mellor**
  - **Méthode OMT**
  - **MERISE** "ancêtre français d'OMT et d 'UML
- Ce cours étudie : UML.



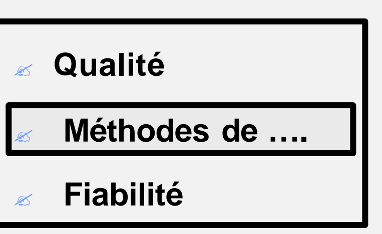
- **UML** : Unified Modeling Language
  - Notation unifiée, sensée regrouper les apports de chacune.
  - Fait office de standard de conception actuellement
  - Basée sur une représentation orientée objets.

Méthodes de conception
UML

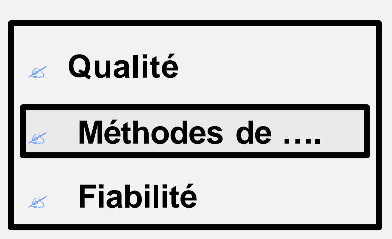


- Langage modélisation
  - Définition de modèles
- Représentation de la connaissance
  - Descriptive
  - Comportementale
  - Structurelle
- Langage graphique et textuel

Méthodes de conception
UML



- Techniques et notations pour représenter différents modèles
- Plusieurs diagrammes possibles
  - classes : structure statique du système.
    - Descriptions des objets et des liens entre objets.
  - fonctionnel : structure opératoire du système.
    - Définition des fonctions de base du système.
  - opérationnel : comportement du système.
    - Diagramme état - transition
- Autres : cas d'utilisation : les Use cases ……..

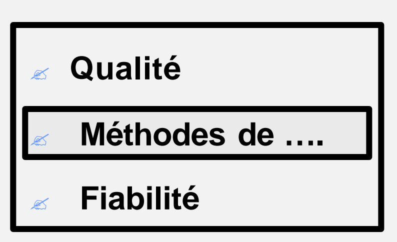


Les diagrammes d 'activité :

- représentation du comportement d'une opération en terme d'action

Les diagrammes de cas d 'utilisation :

- représentation des fonctions du système du point de vue de l'utilisateur.
- Correspond à un DFD

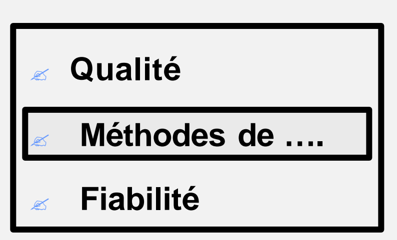


Les diagrammes de classes :

- représentation de la structure statique en terme de classes et de relations.

Les diagrammes de collaboration :

- représentation spatiale des objets, des liens et des interactions.

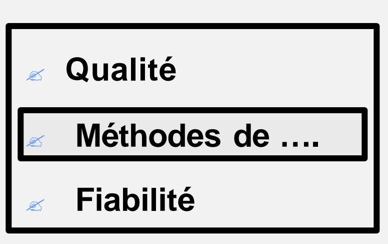


Les diagrammes de déploiement :

- représentation du déploiement des composants sur les dispositifs matériels.

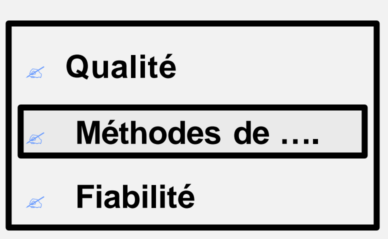
Les diagrammes d'états-transitions :

- représentation du comportement d'une classe en terme d'état.

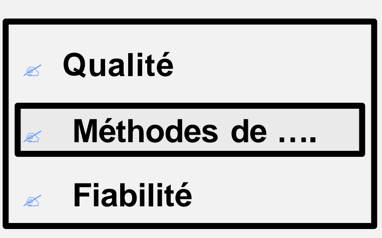


- **Les diagrammes d 'objet :**
  - représentation des objets et de leurs relations, correspond à un diagramme de collaboration simplifié, sans représentation des envois de message
- **Les diagrammes de séquence :**
  - représentation temporelle des objets et de leurs interactions.

Méthodes de conception
UML

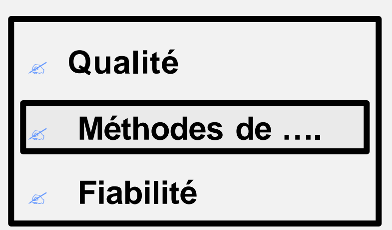


- Diagramme de classes
  - Classe
    - attributs, règles de visibilité, méthodes, invariants
  - Liens entre classes
  - Permet la description de l'architecture logicielle.
  - Structure des objets, des classes, des relations.
  - Éléments de construction :
    - Classes, attributs, relation, opération



- Classe : Regroupement d 'objets homogènes
  - attributs similaires, comportement et relations identiques
- Attribut : propriété caractéristique des objets d 'une classe
  - a valeur basique (simple, valeur objet non autorisée)
- Opération : fonction de transformation
  - manipule les objets de la classe
  - méthode = implantation d 'une opération
- État d'un objet= valuation des attributs d'une classe

Méthodes de conception
UML



- UML permet la *représentation* des :
  - classes, attributs,
  - règles de visibilité,
  - invariants
  - méthodes : comportements
  - relations structurelles ....

Nom_de_la_classe

<< Attributs>>
- attribut_privé : Type
+ attribut_public : Type
/ attribut_dérivé : Type
attribut_protégé : Type

<<**Opérations**>>
- méthode_1 ()
+ méthode_2(par_1 : Type_1 ; …; par_n : Type_n);
+ méthode_3(par_1 : Type_1 ; …; par_n : Type_n) : Type_retour;
…..

<<**Invariants**>>

Point

<< Attributs>>
- abs : Float
- ord : Float

<<**Opérations**>>
Point(x,y : Float); -- Constructeur
+ afficher();
+ changer_ord(y : float);
+ get_ord() : Float;
+ distance(P:Point) : Float;
....

<<**Invariants**>>
Le point ne sort pas de l ' écran

- Diagramme d 'objet :
  - Représentation d 'une instance ou d 'un objet

(Classe)

v_att1
V_att_2

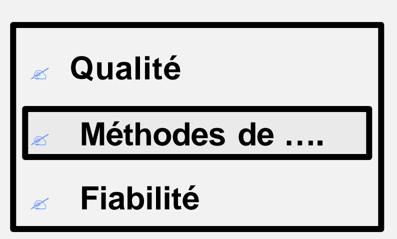
V_att_n

- Exemple
  - Souvent utilisé pour les use cases

(Point)

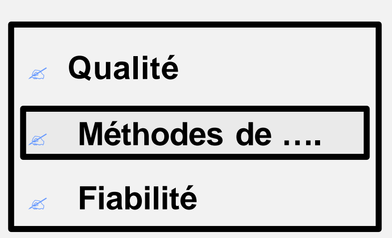
3.0
-2.45

Méthodes de conception
UML



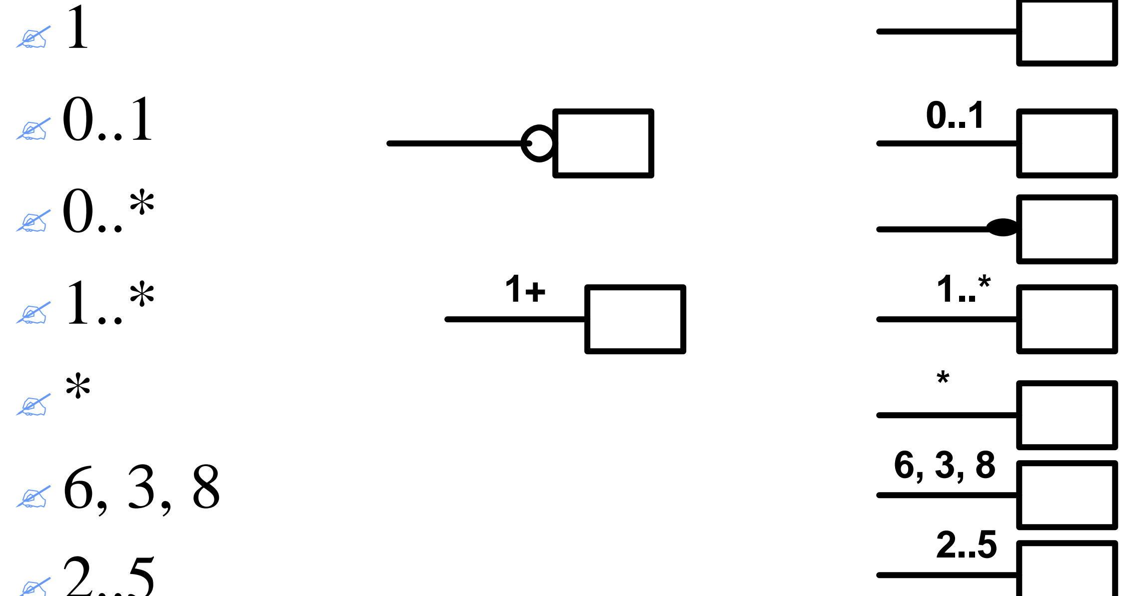
- Associations
  - Liens entre objets ou instances
  - Différentes catégories d'associations
    - Agrégation, composition, spécialisation, généralisation, abstraction, ….
- UML implante différents types d'associations.

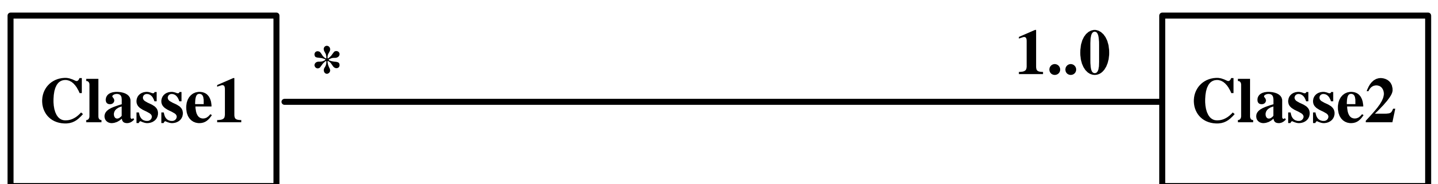
- Associations
  - Rôle
  - cardinalités
  - ....



- **Rôle** : effet de l'intervention d'une classe dans une autre classe
  - Représenté par un nom ou un verbe
  - mon_de_rôle

- **Cardinalité** : quantification du rôle ou du degré d'intervention
  - 1
  - 0..1
  - 0..*
  - 1..*
  - *
  - 6, 3, 8
  - 2..5





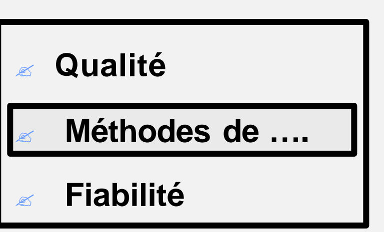
Représentation complète

Méthodes de conception

UML

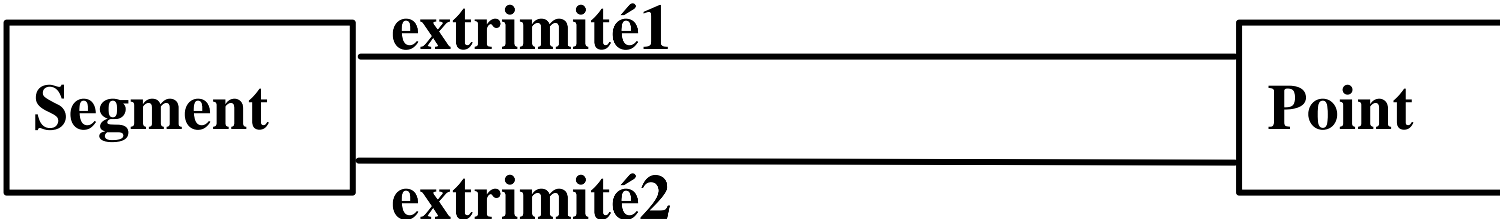
- Fonctionnel

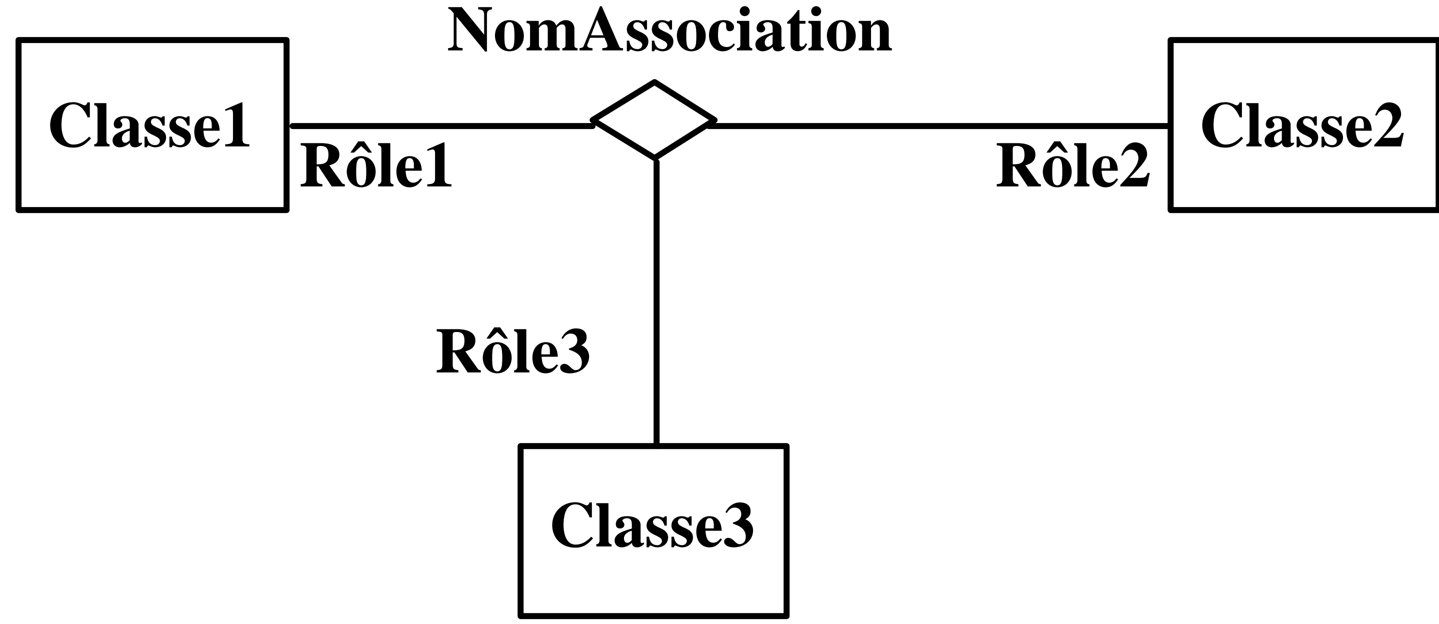
Méthodes de conception
UML



- Exemple : entreprise produisant des produits.

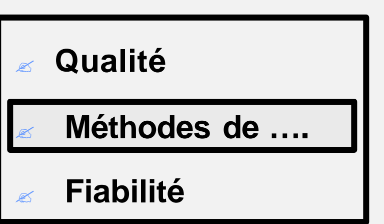
- Exemple segment reliant deux points





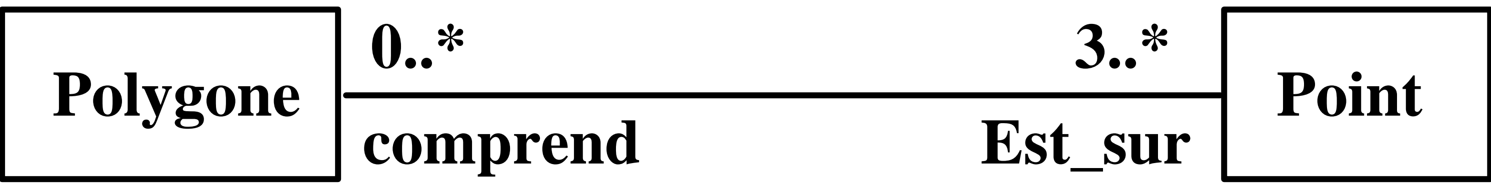
NomAssociation
Classe1
Rôle1
Rôle2
Classe2
Rôle3
Classe3

Méthodes de conception
UML

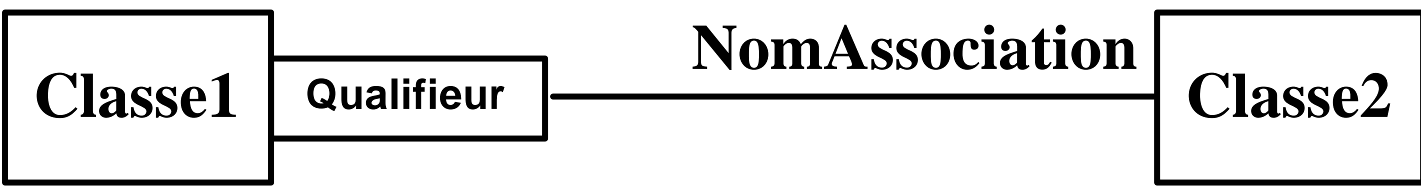


- Relationnel

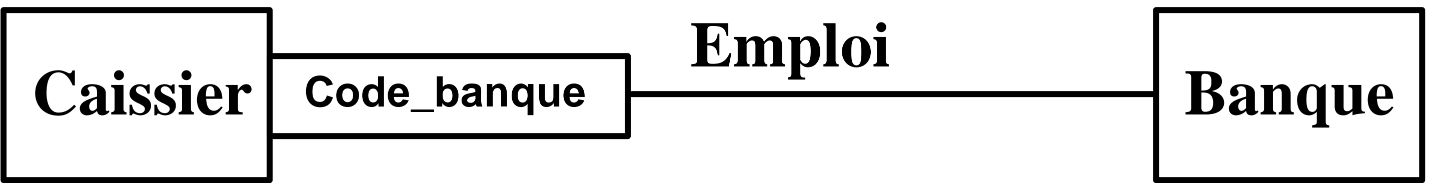
- Représentation d 'un polygone



- Associations qualifiées



- Exemple



- Attributs et méthodes d 'association.

Classe1 — NomAssociation — Classe2
Rôle1, Rôle2

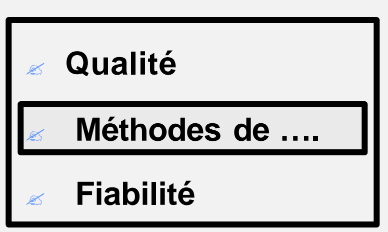
Classe_A
- Attributs
- Opérations

- Exemple

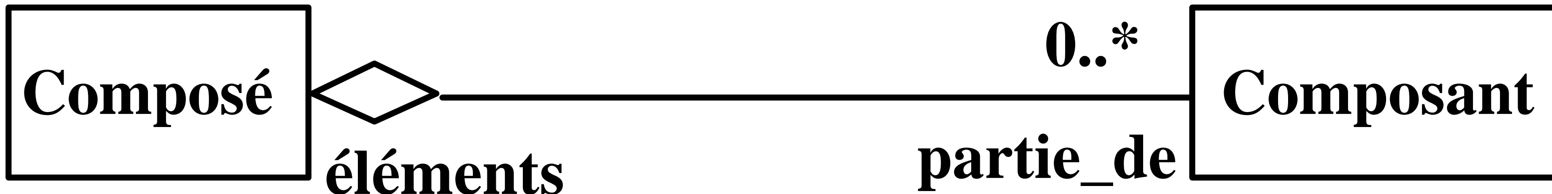
Fichier — accessible — Utilisateur

Propriété
- Droits : …..
- Changer_droits(….)...

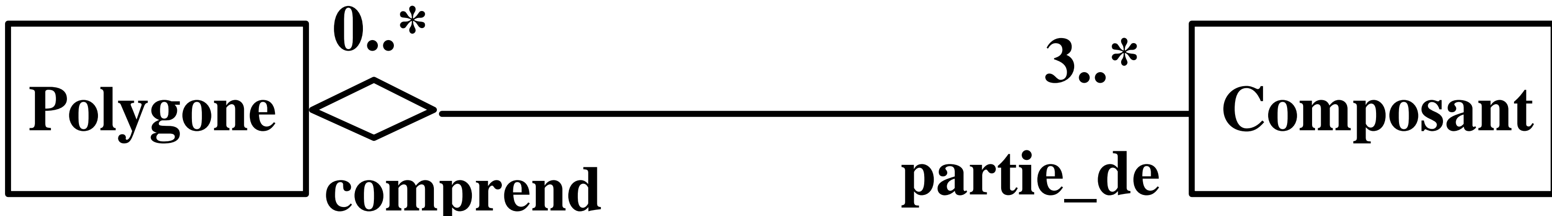
Méthodes de conception



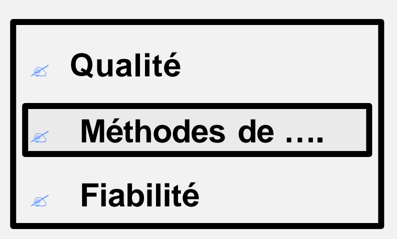
- Relation de composition, ou d'agrégation



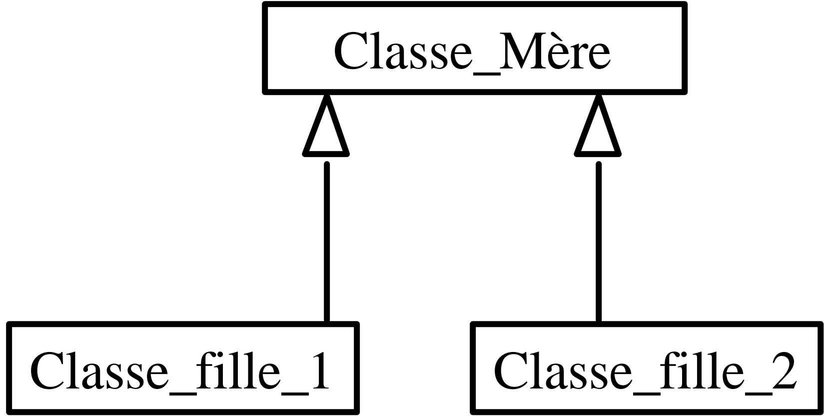
- Exemple : le polygone



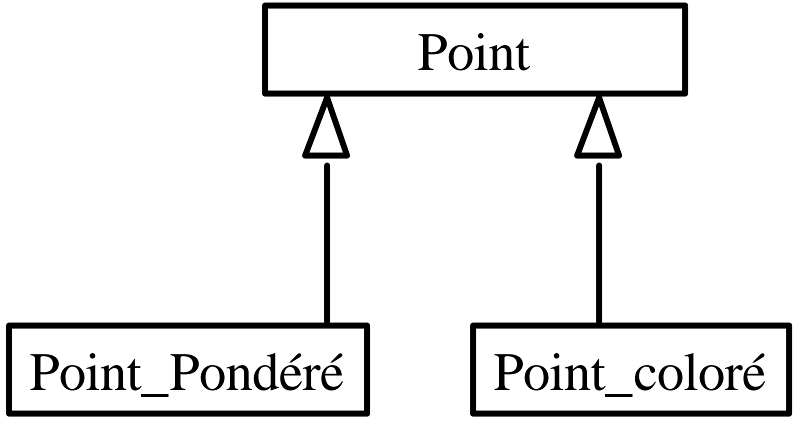
Méthodes de conception



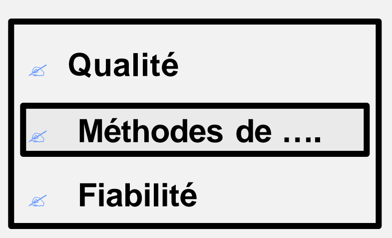
- Relation de généralisation/spécialisation : Héritage
  - Les attributs et les méthodes de la classe mère sont hérités.
  - De nouveaux attributs et méthodes peuvent être définis.



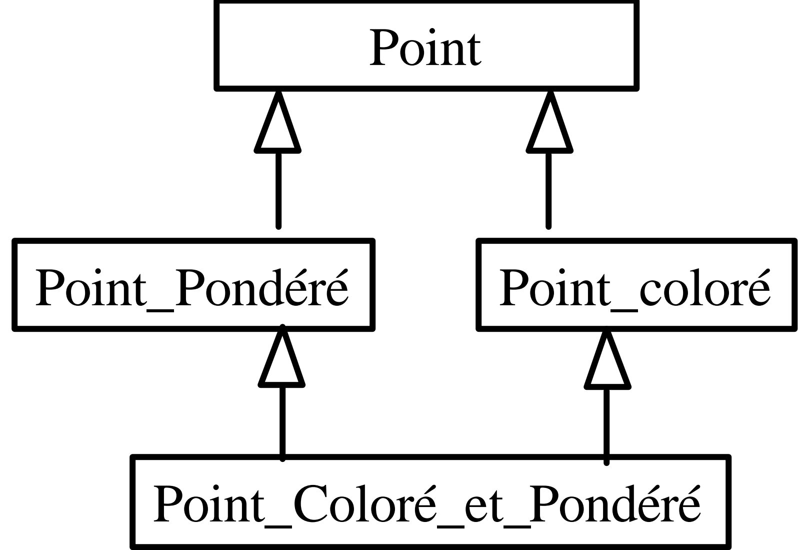
- Exemple : les points.



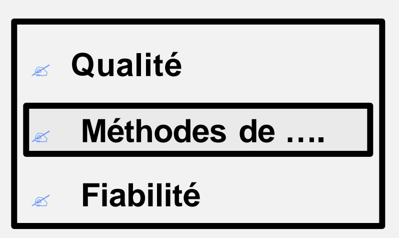
Méthodes de conception



- Héritage multiple.
  - Héritage à partir de plusieurs classes mères.
  - Exemple : les points.



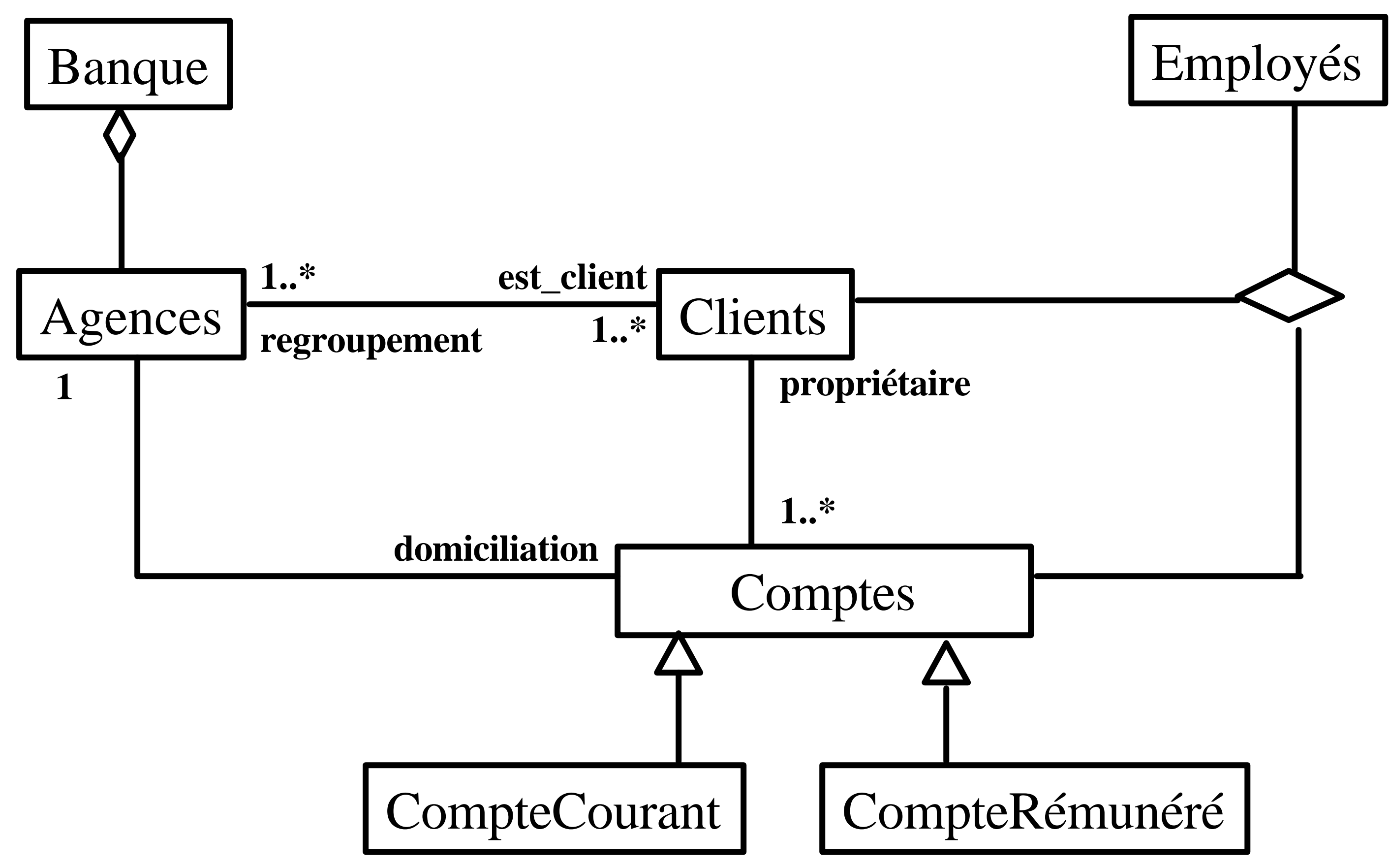
Méthodes de conception

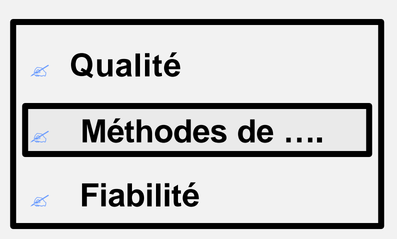


Une banque comprend plusieurs agences réparties sur le territoire. Les agences ont des employés, qui gèrent les comptes de clients. Ces comptes sont de nature différente selon qu 'il sont rémunérés ou non.

- **Exercice** :
  - Proposer un modèle de classes avec les attributs, les opérations ainsi que les associations et les rôles correspondants.

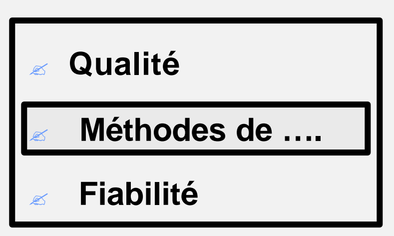
- Une solution : un système bancaire.





Exercice :

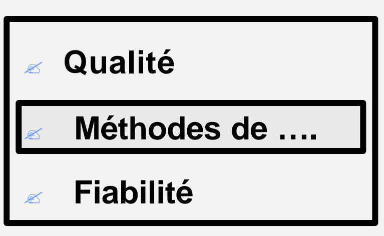
- 1- Reprendre l 'exercice sur la gastronomie d 'une ville et proposer un modèle de classes
- 2- Reprendre l 'exercice des banques et proposer un modèle entité/association.
- 3- Conclure



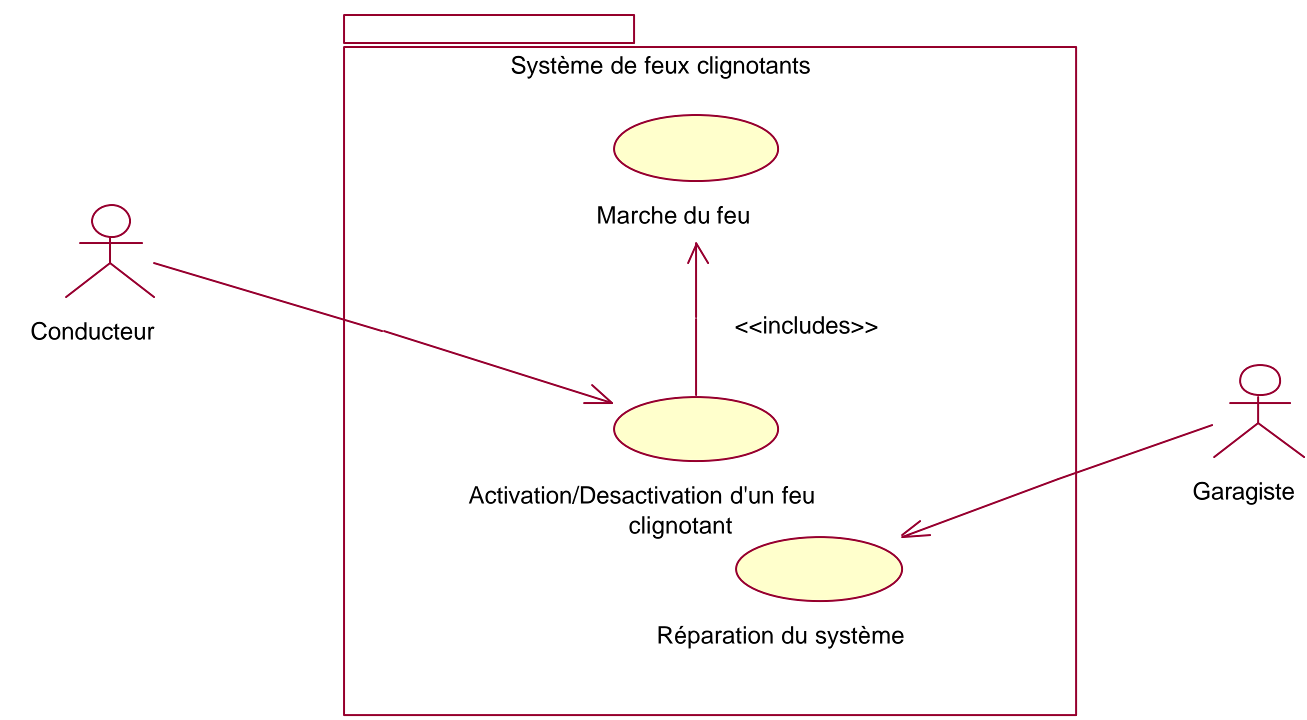
- Également appelés « use case diagrams »
  - Représentent l'aspect fonctionnel du système du point de vue de l'utilisateur.
  - Correspondent à un diagramme de flots de données.

Méthodes de conception

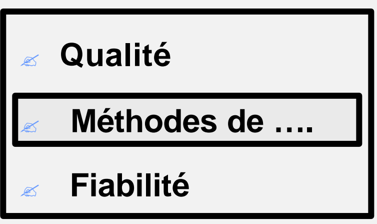
Les diagrammes de cas d 'utilisation



- Représentation très générale du système considéré.
- Introduction des
  - *acteurs* (utilisateurs humains et/ou système externe au logiciel que l'on développe)
  - et des *cas d'utilisations* (représentés par des ovales) du système (le système est représenté par un rectangle qui englobe les cas d'utilisations).
- Ce diagramme établit de plus des liens entre les acteurs et les cas d'utilisations

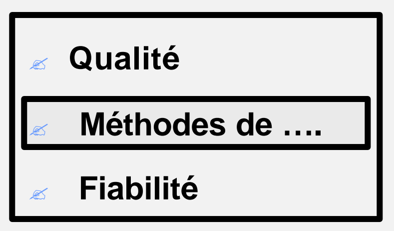


Exemple de diagramme de cas d 'utilisation : les feux clignotants

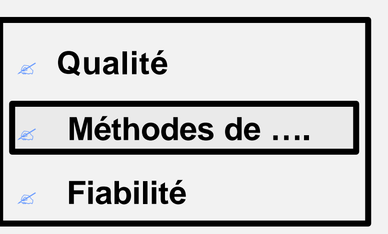


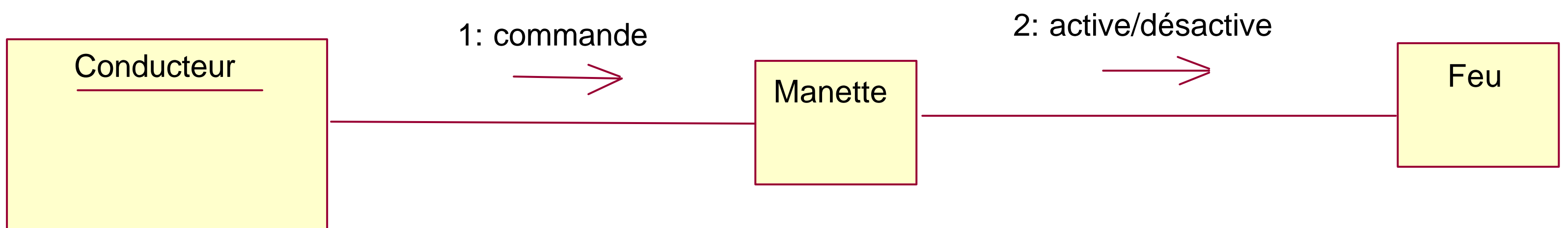
- Représentation spatiale des
  - objets,
  - liens
  - interactions
- Montrer les interactions entre les objets du système

Méthodes de conception

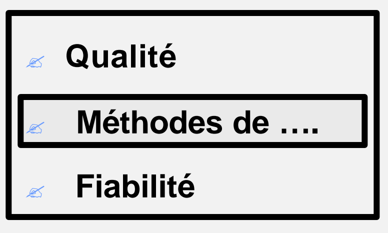


- Représentation des interactions entre objets (instances de classes ou acteurs).
- Les interactions sont réalisées par des échanges de messages entre les objets.
- Les messages sont symbolisés par des flèches.
- On peut préciser le contexte d'une interaction en mentionnant l'état des objets qui interagissent.

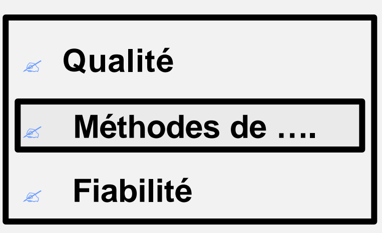




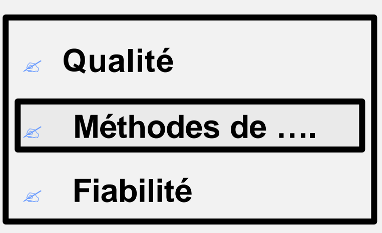
Exemple de Diagramme de collaboration : feux clignotants



- UML permet de définir de manière précise l'ordre et les conditions d'envoi des messages.
- Utilisation d 'un langage de *synchronisation des messages* :
  - Les expressions de synchronisation sont associées aux noms des messages.



- Un *modèle dynamique* décrit le comportement des objets actifs d’un système en termes
  - de *flots de contrôle*,
  - d’*interactions*
  - et des *séquences d'opérations*.
- Un modèle dynamique est défini par des **diagrammes d'états-transitions** :
  - UML propose de définir un diagramme d'états-transitions (en anglais“ *Statechart* ”) par objet du système;
  - mais on peut aussi, définir un diagramme d'états-transitions par système ou par groupe d’objets.

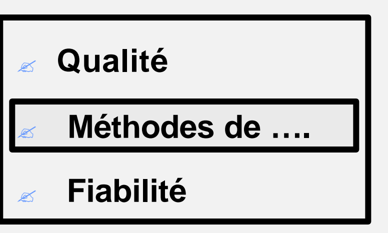


- UML propose en fait 3 types de diagrammes pour décrire le comportement d'un système et de ses composants :
  - les **diagrammes de séquence** qui permettent de réaliser une analyse préalable d'un système et de sa dynamique par la définition de *scénarios*.
  - les **diagrammes d'états-transitions** qui sont au cœur de la modélisation de la dynamique des systèmes.
  - les **diagrammes d'activités** qui détaillent les modèles précédents et se rapprochent de la programmation.

- Les diagrammes de séquence sont fondés sur la notion de scénarios.

- Un ***scénario*** est une séquence d'événements se déroulant durant une exécution particulière (un cas d’utilisation) du système modélisé.

- Une séquence est une suite dotée d ’un ordre. Ici la chronologie des événements définit cet ordre.

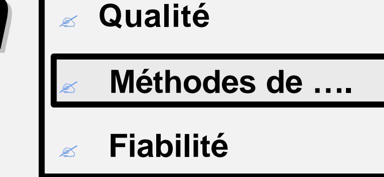


Un scénario d 'utilisation des clignotants

- *.au départ la manette est en position neutre*
- *.la manette est poussée vers le haut* ? *le feu clignotant gauche s'allume*
- *.envoi d'un signal d'horloge du système de commande électrique*
  ? *.le feu clignotant gauche s'éteint*
- *.envoi d'un signal d'horloge* ? *le feu clignotant gauche s'allume*
- *.envoi d'un signal d'horloge* ? *le feu clignotant gauche s'éteint*
- *.envoi d'un signal d'horloge* ? *le feu clignotant gauche s'allume*
- *.la manette est poussée vers le bas* ? *le feu clignotant gauche s'éteint*
- *.envoi d'un signal d'horloge (sans effet)*
- *.la manette est poussée vers le bas* ? *le feu clignotant droit s'allume*
- *.envoi d'un signal d'horloge* ? *le feu clignotant droit s'éteint*
- ....

Méthodes de conception

Les diagrammes de séquence

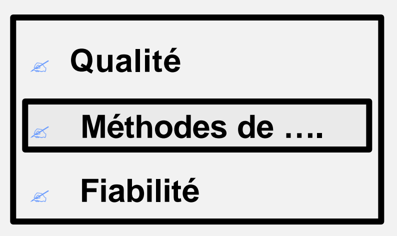


- Un diagramme de séquence est représentatif d'un scénario où apparaissent
  - les intervenants, c'est-à-dire les objets émetteurs et récepteurs des événements.
- Un diagramme de séquence permet de commencer à identifier
  - quels objets sont mis en œuvre dans le système à développer.

Méthodes de conception

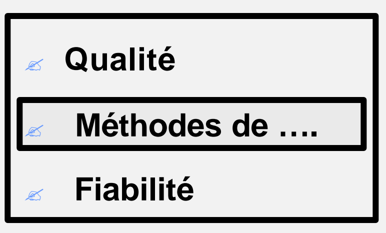
Les diagrammes de séquence exemple des clignotants

Diagramme de séquence : les feux clignotants

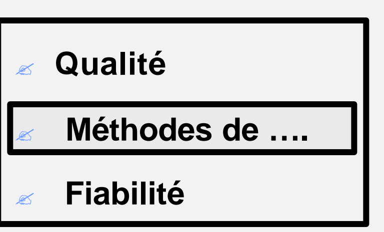


- Les objets intervenant dans le scénario sont représentés par des lignes verticales
- Les parties de lignes avec une double épaisseur représentent des phases d'activité des objets.
- Les communications (événements) entre objets sont représentées par des flèches horizontales. Le temps s'écoule verticalement, de haut en bas.

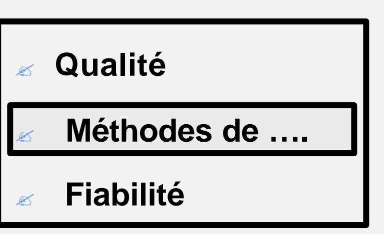
- Un **Diagramme d’états-transitions**
  - appelé aussi **automate**, en anglais : ***Statechart***
  - décrit le comportement des objets d’une classe en terme d’états et de transitions entre états.
- Un automate est associé à chaque classe :
  - tous les objets (instances) d'une classe
    - ont le même comportement
    - et partagent le même automate,
    - mais chacun fonctionne à son propre rythme.



- Un diagramme d'états-transitions est un graphe orienté dont
  - les nœuds sont des ***états***
  - et les arcs sont des ***transitions***.
- Éléments de base d 'un diagramme d'états-transitions sont :
  - les ***états***.
  - les ***transitions***.
  - les ***évènements***.



- les ***états***.
  - Un objet est à tout instant dans un état donné.
  - L'état d'un objet est constitué
    - par les valeurs de ses attributs
    - et de tous ses liens avec d'autres objets.



- les ***transitions***.
  - Une transition représente un changement d'état ;
  - elle est déclenchée par l'arrivée d'un événement.

- les ***événements.***
  - Un événement est un stimulus d'un objet vers un autre objet.
  - Certains événements
    - sont de simples signaux ;
    - d'autres transportent de l'information entre les objets ; l'information est alors représentée par des *attributs* en paramètres.

Méthodes de conception
Les diagrammes états-transitions.

Notations

Etat_I — événement(attributs) → Etat_J

- Propriétés
  - un événement étant considéré comme instantané, la traversée d'une transition est également instantanée ; par contre le passage dans un état a une certaine durée directement dépendante du traitement effectué dans l'état
  - le passage dans un état correspond à l'intervalle entre deux événements reçus par un objet.

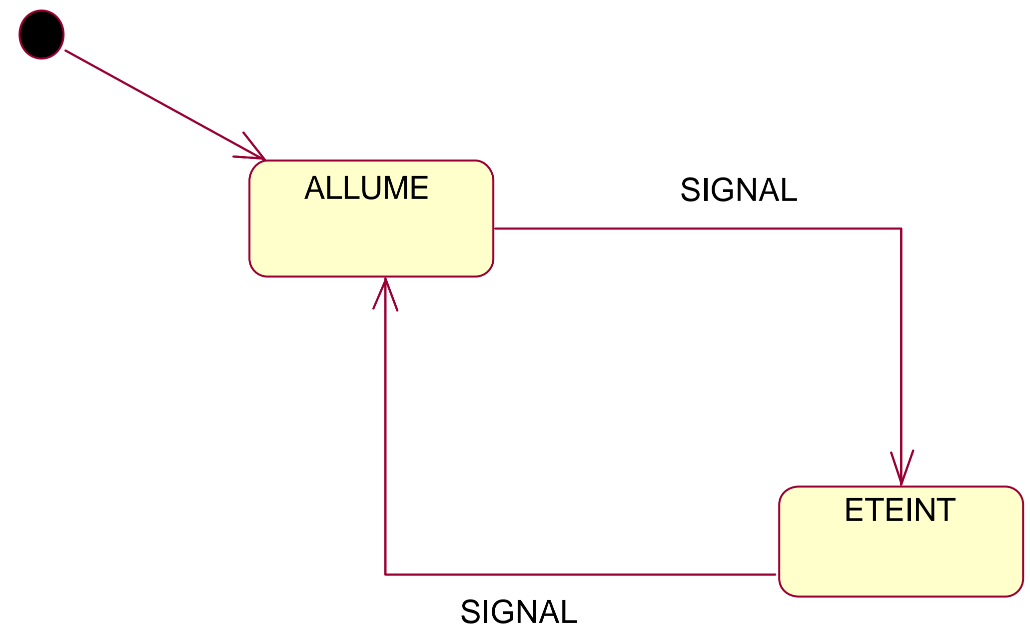
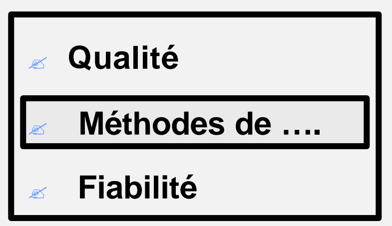


Diagramme état-transition : un feu clignotant

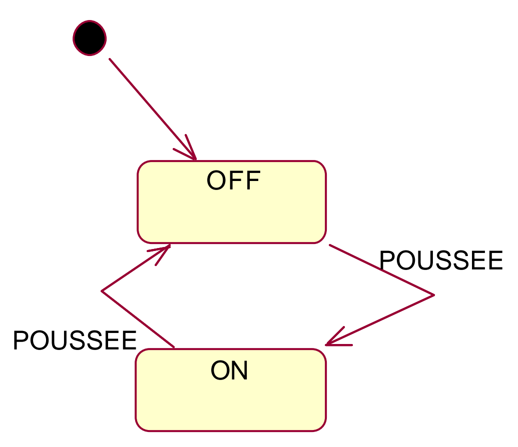


- Structuration d 'un diagramme état-transition
  - Il est souhaitable de structurer les diagrammes pour les rendre plus compréhensibles.
  - La " ***Structuration OU*** " consiste en une hiérarchie d'états :
    - un état correspond à un sous-diagramme qui comprend lui-même des états.

- Deux notations (équivalentes) sont possibles

  - ***Diagrammes imbriqués*** : les sous-diagrammes sont définis séparément.
  - ***Généralisation d'états*** : les sous-diagrammes sont définis ensemble (“ emboîtés ” par des *contours*).

Digrammes d 'états-transitions avec imbrication.



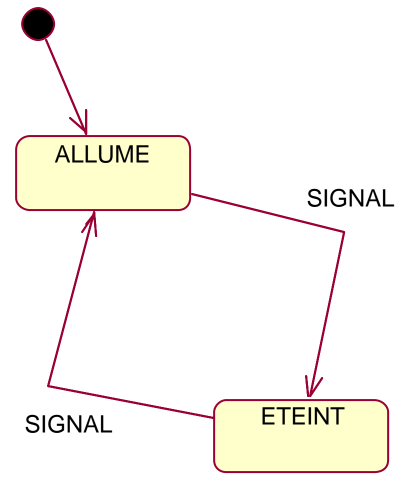
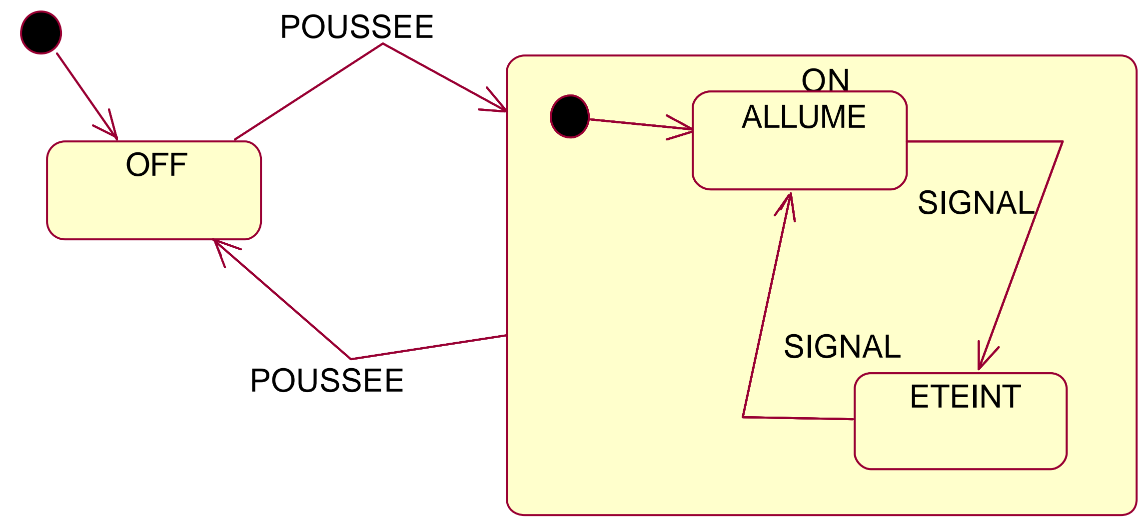


Diagramme général

Sous-diagramme ON

Digrammes d 'états-transitions avec généralisation.

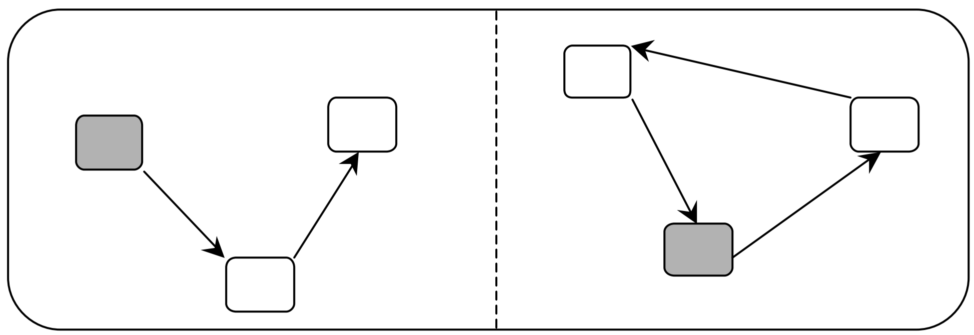


Exemple avec généralisation d'états

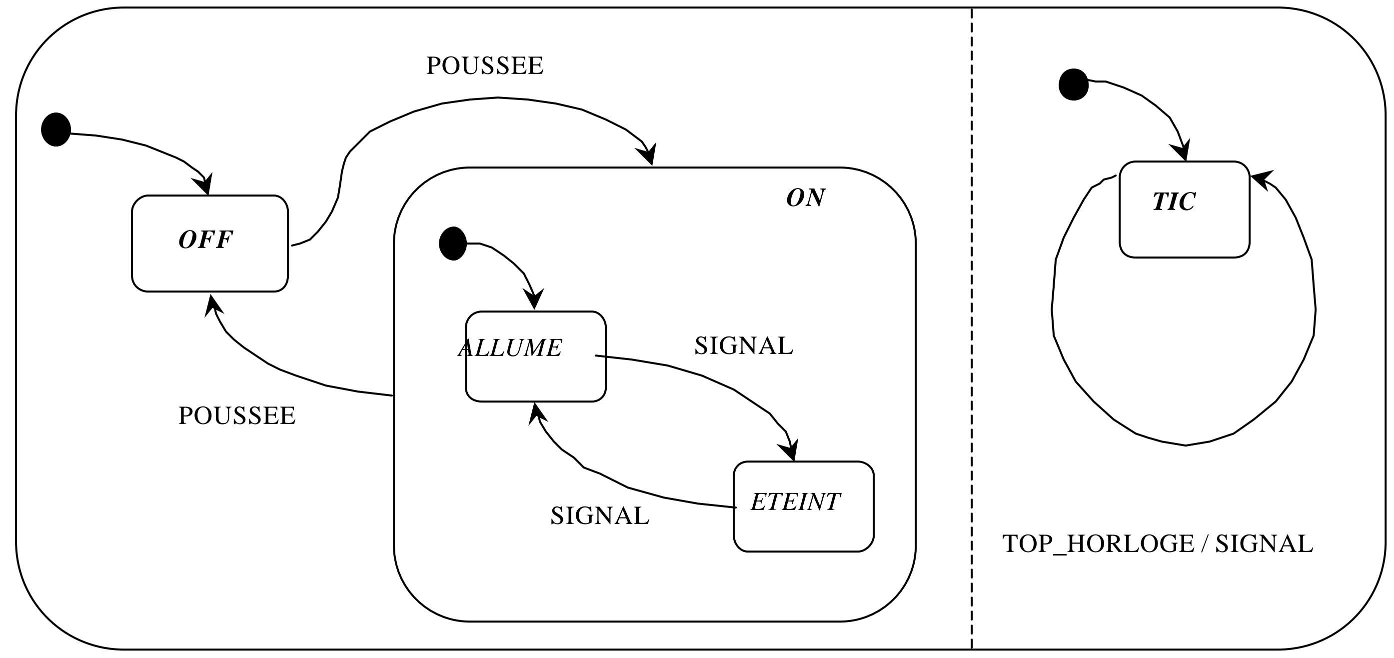
Concurrence - agrégation d ’états

- Un diagramme d’états-transitions peut représenter le comportement d’un système ou d’un groupe d’objets.
- L’état d’un système est composite :
  - il est constitué de l’ensemble des états des objets qui composent le système.

- *Agrégation d'états* (" *Structuration ET* ")
  - Représentation de l'état composite d'un système.
  - Regroupement de plusieurs diagrammes d'états-transitions qui évoluent simultanément



Structuration ET : Exemple



La transition dans le sous-diagramme de l'horloge (qui boucle sur l'état *TIC*) est déclenchée par un événement externe TOP_HORLOGE. Elle lance une action SIGNAL, perçue ici comme un événement envoyé par l'horloge et reçu par le feu clignotant.

- Forme générale d 'un digramme états-transitions
  - Autres notions dans les diagrammes d'états-transitions
    - ***Conditions*** : utilisées comme *gardes* sur les transitions
    - Opérations : ***activités*** et ***actions***.

Méthodes de conception

Les diagrammes états-transitions.

- Opérations : *activités* **et** *actions*.

- *Conditions* : utilisées comme ***gardes*** sur les transitions

- *Notation générale*

  - **Attributs, conditions, activités et actions sont facultatifs.**

ETAT_I

do/
entry/
exit/
event Undefined/

EVENEMENT(attributs)[condition] / action

ETAT_J

Quelques expressions utiles pour les transitions

Expressions d'événements

- `en(S)` : on est entré dans l'état S
- `tr(C)` : la condition C est devenue vraie
- `ex(S)` : on est sorti de l'état S
- `fs(C)` : la condition C est devenue fausse
- `tm(E,n)` : timeout ? n clocks d 'horloge depuis le dernier l'événement E *(notion temporelle)*

Quelques expressions utiles pour les transitions

Expressions de conditions

- *expressions booléennes classiques (opérateurs* `not, and, or` *comparaisons* `<, >, =< )`
- `in(S)` : on est dans l’état S
- `ac(A)` : l ’activité A est active

Quelques expressions utiles pour les transitions

- Expressions d'actions
  - `E` : on génère l'événement E
  - `tr!()` : la condition C est mise (forcée) à «vrai»
  - `st!(A)` : on lance l'activité A
  - `fs!()` : la condition C est mise (forcée) à «faux»
  - `sp!(A)` : on stoppe l'activité A
  - `...`

- Une *activité*
  - est notée **do/** sur l'ETAT_I
  - opération qui nécessite un certain temps d'exécution
  - est associée à un état.
  - se termine
    - d'elle-même au bout d'un certain temps
    - ou bien une opération qui dure indéfiniment.
- Une activité peut être interrompue par l'arrivée d'un événement pris en compte dans l'état actif.

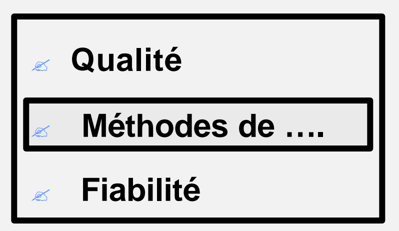
Une *action*

- est notée action sur la transition
- est une opération instantanée (ou de durée non significative).
- Elle est généralement associée à une transition.
- Elle est exécutée lors du déclenchement de la transition : elle est alors perçue comme l'émission d'un événement qui sera reçu par une transition dans une autre partie du diagramme
- **Exemple** : action SIGNAL dans l'exemple du feu clignotant contrôlé par une horloge

- Certaines actions particulières sont cependant associées à un état :
  - *action d'entrée* (notée entry/ dans l'ETAT_I) : action exécutée lors de l'entrée dans l'état,
  - *action de sortie* (notée exit/ dans l'ETAT_I): action exécutée au moment de quitter l'état,
  - *actions internes* (notée event Undefined/ dans l'ETAT_I) : action exécutée sur une occurrence d'*événement* sans changer d'état.

Remarque

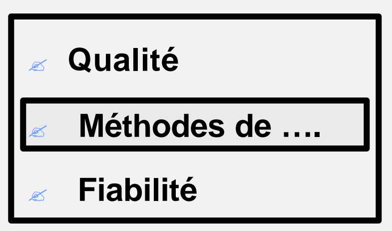
- Dans l’exemple du feu clignotant contrôlé par une horloge,
- l’action SIGNAL aurait pu être définie comme une action interne à l’état TIC,
- déclenchée sur une occurrence de l’événement TOP_HORLOGE.



- Les *diagrammes d'activités*
  - permettent de représenter le comportement des *opérations* ou des *cas d'utilisation* en terme d'actions.
  - Ces diagrammes reprennent, en les détaillant, les modèles de comportements exprimés dans les diagrammes d'états-transitions.
  - on se rapproche de la programmation (ou implémentation).

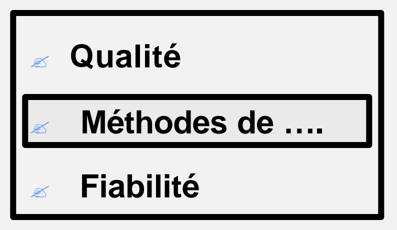
Méthodes de conception

Les diagrammes états-transitions.



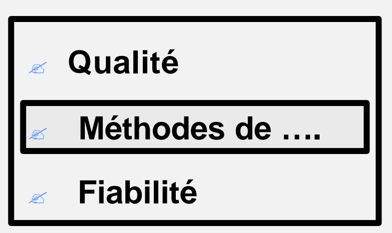
- La notation des diagrammes d 'activités
  - est proche du formalisme ***LDS*** (langage de spécification)
  - très utilisé dans le domaine des télécoms et des réseaux pour la spécification de protocoles.
- *Les diagrammes d 'activités ne sont pas détaillés dans ce cours.*

Méthodes de conception



- Diagrammes de classes sont conçus par raffinement
- Les raffinements correspondent à différents niveaux du modèle de développement.
  - Spécification
  - Conception
  - Conception détaillée
- Descriptions des diagrammes états-transitions et les diagrammes de séquence dès que le comportement peut être capturé

Méthodes de conception



- Spécification
  - Représentation des classes par leurs noms.
  - On représente une ébauche de diagramme de classes.
  - On y trouve également :
    - Les rôles
    - Les multiplicités
    - Les associations
  - Descriptions des diagrammes de cas d'utilisation.

Conception

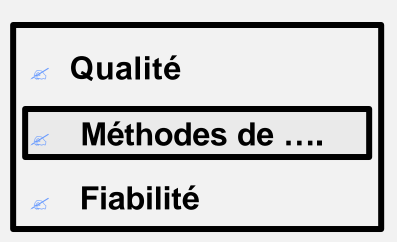
- Description des diagrammes de classes avec :
  - Les attributs
  - Les opérations
  - Les règles de visibilité
  - Les rôles
  - Les multiplicités
  - Les associations
- Les diagrammes états-transitions qui font intervenir
  - Les attributs
  - Les opérations.

Conception détaillée

- Les associations,
- les rôles,
- les multiplicités

- disparaissent du diagramme de classes.
  - seules les classes subsistent.
- Le diagramme de classes contient :
  - Les attributs,
  - Les opérations
  - Les règles de visibilité
  - Les attributs qui représentent les assocations.
  - Les opérations : modificateurs, accesseurs, constructeurs, testeurs, destructeurs, … pour les attributs (pour réaliser l 'encapsulation).

Méthodes de conception

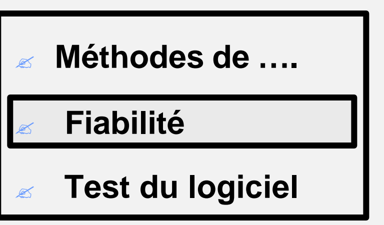


- Différents diagrammes
- Différentes utilisations
- Tentative d 'unification avec UML
- Une application ne nécessite pas l'utilisation de tous les diagrammes obligatoirement.
- Le domaine d'application détermine les diagrammes à utiliser
- Attention : la méthodologie propre au concepteur

Plan du cours

- VIII. Méthodes de conception de logiciels
- ? **IX. Fiabilité du logiciel**
- **X.** Test du logiciel
- **XI.** Gestion des versions
- **XII.** Réutilisation de logiciels
- **XIII.** Maintenance de logiciels
- **XIV.** Introduction aux méthodes formelles

Fiabilité du logiciel



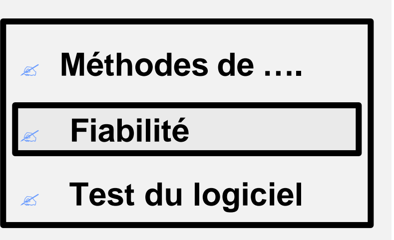
- Définition et mesure de cette notion
- Processus de test statistique
- Évaluation a priori de la sûreté et tests statistiques

Fiabilité du logiciel

- Méthodes de ....
- Fiabilité
- Test du logiciel

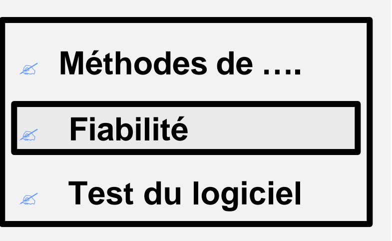
- Définition
  - Probabilité d'avoir une opération sans défaillance pendant un temps déterminé pour un environnement déterminé pour un objectif déterminé
  - Cela implique des points de vue différents selon que l'on considère les usagers du système ou le système lui-même
- Mesurer la confiance des usagers dans les services rendus par un système

Fiabilité du logiciel



- Impossible à définir de manière objective
  - un ensemble de “mesures” sorties de leur contexte ne signifie rien
- Nécessite un “profil opérationnel” pour être définie “proprement”
  - un profil opérationnel définit les types d’utilisation attendus

Fiabilité du logiciel



- Il faut considérer les conséquences d'une défaillance
  - Certaines défaillances sont plus dangereuses que d'autres
  - Plus une défaillance est importante, plus la confiance dans le système est réduite.

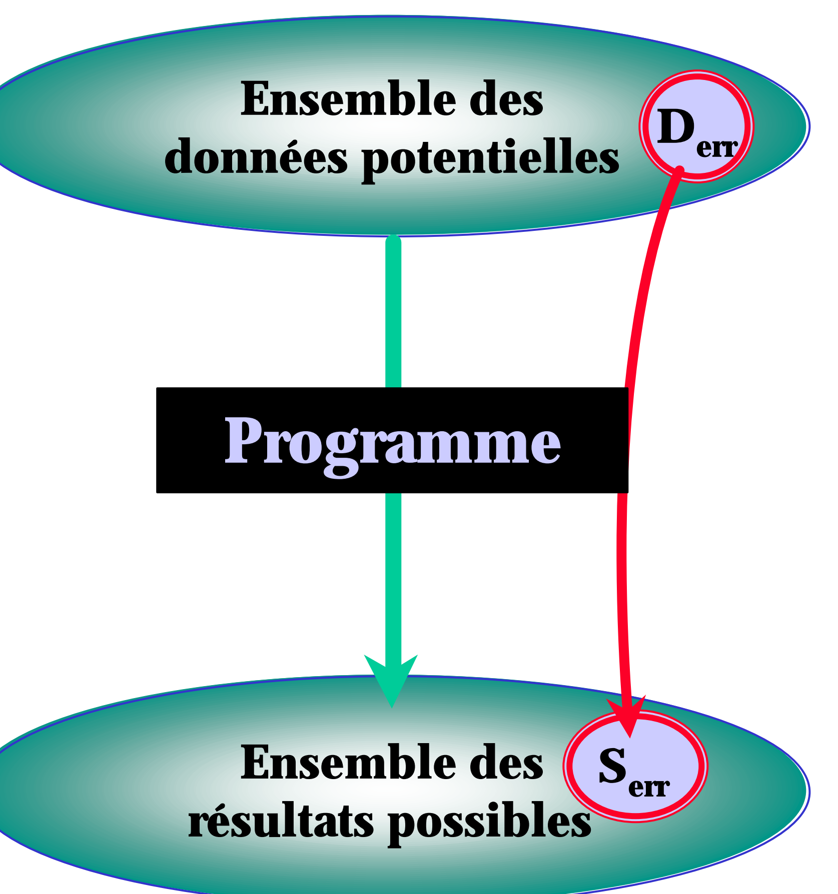
Qu'est ce qu'une *défaillance*? qu'est-ce qu'une *erreur*?

Fiabilité du logiciel

- Défaillance
  - Comportement inattendu observé par un utilisateur
  - Lié à l'exécution d'un système
- Erreur
  - Caractéristique statique qui peut dégénérer en défaillance
  - Ne débouche pas forcément sur une défaillance

Fiabilité du logiciel

- Conséquences
  - Une erreur devient une défaillance si la partie "erronée" du logiciel est exécutée
  - Une défaillance non constatée en est-elle une?



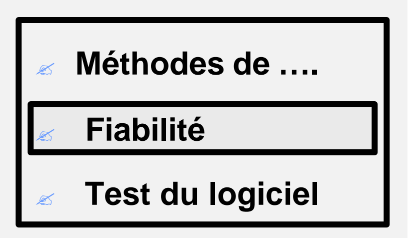
Fiabilité du logiciel
Classification des défaillances

Type de défaillance	Description
Episodique	N'apparaît que pour certaines entrées
Permanente	Apparaît pour toutes les entrées
Rattrapable	Le système peut se "rattraper" sans intervention d'un opérateur
Irrattrapable	Un opérateur doit intervenir pour rattraper la défaillance
Non-corrompante	La défaillance ne corrompt pas l'état du système ni des données qui sont manipulées
Corrompante	La défaillance corrompt l'état du système et/ou des données qui sont manipulées

Fiabilité du logiciel

D'après I.Sommerville ©1995

Fiabilité du logiciel
Augmenter la fiabilité ?



- La fiabilité est acquise lorsque les erreurs contenues dans les composants les plus utilisés ont été corrigées
  - On ne peut dire d'un logiciel qu'il est "sans erreur"!
- Supprimer x% des erreurs n'accroîtra pas forcément la fiabilité d'un facteur correspondant
  - Exemple : "la suppression de 60% des erreurs n'a augmenté la fiabilité du logiciel que de 3%"

Objectif: supprimer les erreurs ayant les conséquences les plus "importantes"

Fiabilité du logiciel

Fiabilité et efficacité

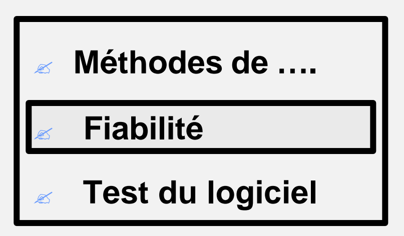
- Fiabilité et efficacité sont contradictoires
  - fiabilité implique “code redondant” (contrôles à l’exécution etc.)
  - code redondant implique, exécution plus lente
- En général
  - la fiabilité est plus importante que l’efficacité (sauf cas particuliers)
    - les machines sont de plus en plus rapides
    - un logiciel non fiable n’est pas utilisé
    - le coût d’une défaillance peut dépasser le coût du logiciel en lui-même

Fiabilité du logiciel

Mesurer la fiabilité

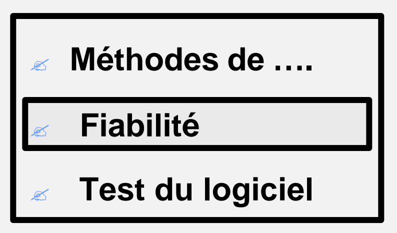
- Les techniques utilisées pour mesurer le matériel <u>ne sont pas adaptées</u>
  - liées à la défaillance de composants
  - liées au remplacement de composants défaillants
  - La conception est <u>considérée comme correcte</u>
- Les défaillances logicielles sont <u>toujours</u> liées à des erreurs de conception
- Un système peut continuer à fonctionner malgré l'apparition d'une défaillance
  - ne pas confondre "défaillance" et "observation de la défaillance"

Fiabilité du logiciel
Unités de mesure



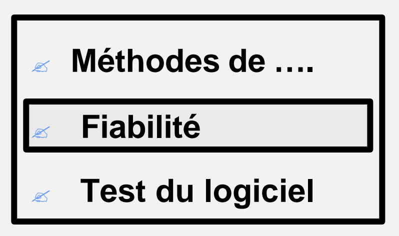
- Probabilité d'obtenir une défaillance sur "requête" (Probability of failure on demand, **POFOD**)
  - Mesure permettant d'évaluer la propension du système à défaillir sur sollicitation
  - POFOD = 0,001 signifie "il y a une chance sur 1000 pour qu'une sollicitation aboutisse à une défaillance"
  - Significative pour les systèmes critiques ou fonctionnant en permanence (démons)

Fiabilité du logiciel
Unités de mesure

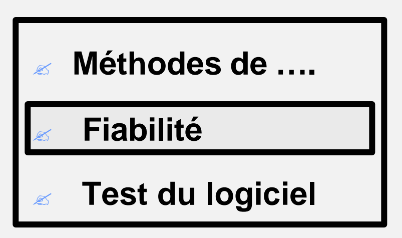


- Taux d'apparition des erreurs (Rate of fault occurrence, **ROCOF**)
  - Fréquence d'apparition d'un comportement inattendu
  - **ROCOF = 0,02** signifie "deux défaillance toutes les 100 unités de temps opérationnel"
  - Significative pour les OS et les systèmes basés sur la notion de transaction

Fiabilité du logiciel
Unités de mesure



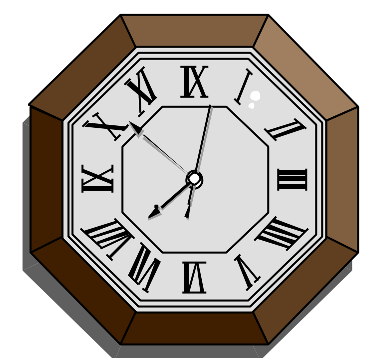
- Temps moyen entre deux défaillances (Mean time to failure, **MTBF**)
  - Mesure le temps écoulé entre deux défaillances observées
  - **MTBF = 500** signifie "il s'écoule 500 unités de temps entre deux défaillances"
  - Significatif pour les systèmes ayant de "longs traitements"



- Disponibilité (Availability, **AVAIL**)
  - Mesure la disponibilité du système en tenant compte des temps de réparation et de redémarrage suite à une défaillance
  - **AVAIL = 0.998** signifie “le logiciel est opérationnel 998 pour 1000 unités de temps”
  - Significatifs pour les systèmes fonctionnant perpétuellement (autocommutateur téléphonique)

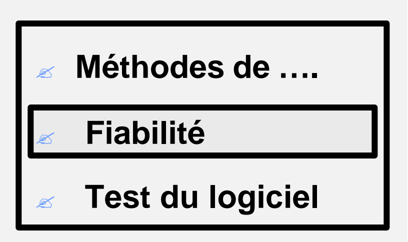
Fiabilité du logiciel
Comment mesurer la fiabilité

- **POFOD**
  - mesurer le nombre de défaillances sur un ensemble de sollicitations
    - Constitution de jeux de tests de grande taille
- **ROCOF** et **MTBF**
  - mesurer le temps (ou le nombre de “transactions” entre deux défaillances)
    - introduction de mécanismes de surveillance dans le code
    - surveillance externe
- **AVAIL**
  - Mesurer le temps nécessaire à la remise en route du système



- Attention au “temps”!!!
  - bien sélectionner les unités de temps (en fonction du système)
    - temps d’exécution (démons)
    - temps “calendaire” (systèmes à exécution “éphémère” utilisés régulièrement)
    - nombre de transactions (systèmes transactionnels)

Fiabilité du logiciel
Spécifier la fiabilité



- La cahier des charges en matière de fiabilité est rarement exprimé de manière quantitative et vérifiable
- Il est nécessaire de définir un profil opérationnel en tant que partie intégrante du plan de tests
- La fiabilité est une notion “dynamique”
  - (NE PAS la lier aux sources)

- Proposition de procédure
  - Analyser pour chaque sous-système les conséquences d'une défaillance
  - A partir de cette analyse, classer les défaillances en fonction des classes identifiées
  - Pour chaque défaillance, établir a priori les critères de fiabilité en utilisant l'unité (les unités) de mesure la (les) plus appropriée(s)

Fiabilité du logiciel
Exemple : distributeur de billets

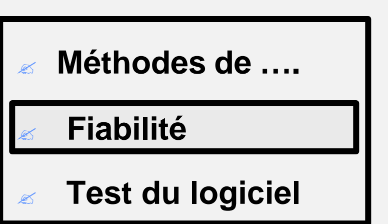
- Identification des caractéristiques d'utilisation
  - Chaque machine du réseau est utilisées environ 300 fois par jour
  - la banque possède 1000 machines
  - la durée de vie du logiciel embarqué sur ces machines est de 2 ans
  - chaque machine devra exécuter 300*365*2 =± 219000 transactions
  - Le système complet supporte environ 300000 transactions par jour

Fiabilité du logiciel
Exemple : distributeur de billets

Type de défaillance	Exemple	Unité de mesure
Permanente, non corrompante	L'appareil refuse toutes les cartes, il fa redémarrer pour que ça reparte	ROCOF 1 / 1000 jours
Episodique, non corrompante	Certaines cartes correctes ne peuvent être lues	POFOD 1 /1000 transactions
Episodique, Corrompante	Une séquence type de transactions cor rompt la base de données	non quantifiable JAMAIS

D'après I.Sommerville ©1995

Fiabilité du logiciel
Validation



Il est impossible de valider empiriquement des critères de fiabilité contraignants

(1)

Pas de corruption de la base de données signifie

POFOD < à 1/200 000 000

(2)

Si simuler une transaction dure 1 seconde,

simuler les transactions sur une journée prend 3,5 jours

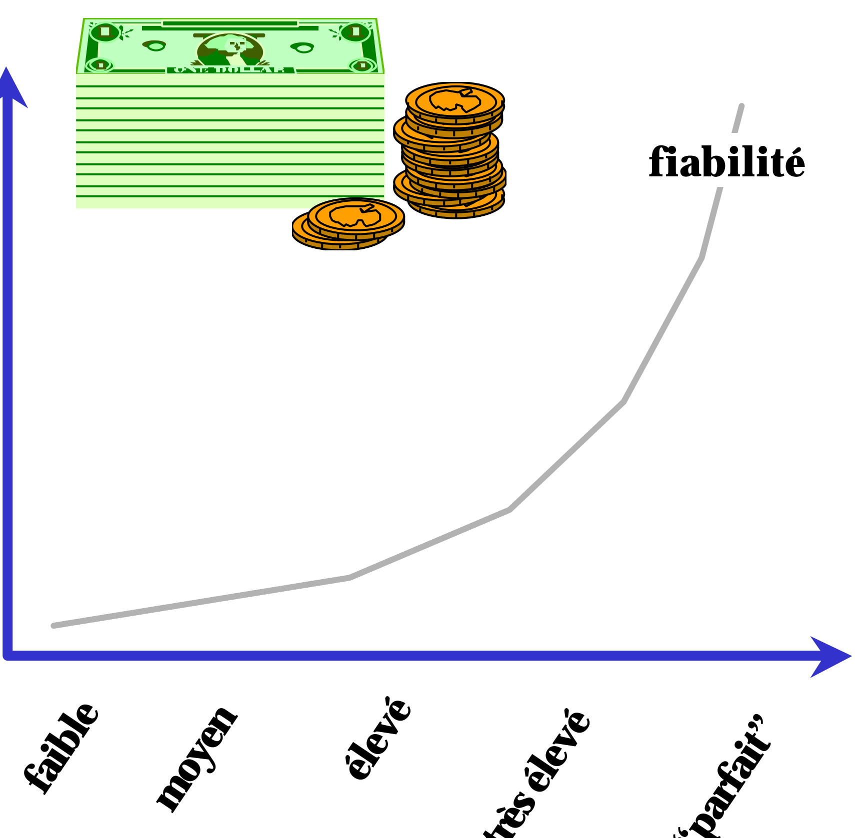
(3)

Dans ces conditions, tester la fiabilité du système sur ce critère

prend plus de temps que la durée de vie estimée du système!

Fiabilité du logiciel
Aspects économiques

- Le prix de la fiabilité est extrêmement élevé
- Il peut être intéressant d'accepter certaines défaillances et d'en payer le prix a posteriori
  - Analyse statistique
  - Analyse du risque
- Attention, cela implique
  - Des facteurs "politiques" ou "sociaux" (réputation)
  - Du type de système (donc, des risques encourus)



D'après I.Sommerville ©1995

Fiabilité du logiciel
Exemple : distributeur de billets

- On teste la fiabilité du logiciel plutôt que de rechercher ses erreurs
- La sélection des données fournies en entrée doit respecter le profil d'utilisation identifié
- Le nombre d'erreurs observées donne une idée de la fiabilité du logiciel
- Le niveau de fiabilité "accepta-ble" est déterminé a priori et le logiciel testé jusqu'à ce qu'il soit atteint

Procédure type

- Déterminer le profil opérationnel type du logiciel
- Générer un ensemble de données correspondant à ce profil
- Appliquer les tests et mesurer le nombre d'exécutions entre deux défaillances
- Une fois qu'un nombre "statisti-quement valide" de tests a été effectué, on a une idée de la fiabilité du logiciel testé

Fiabilité du logiciel
Fiabilité et techniques formelles

- L'usage de "méthodes formelles" permet d'obtenir des systèmes fiables
  - Conception détaillée d'un modèle (analyse poussée très tôt)
  - Preuve sur ce modèle (analyse structurelle et comportementale)
  - Différentes approches (model checking, évaluation, démonstration assistée etc.)
- Mais...
  - Opération de modélisation TRES délicate
  - Formalismes complexes et délicats à manipuler
  - Il faut éviter de bâtir une preuve sur des hypothèses non vérifiables

Une technique du futur?

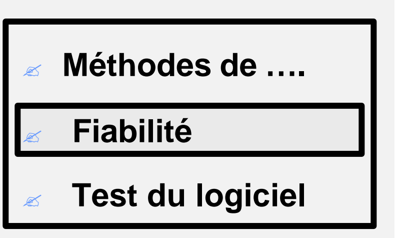
Fiabilité du logiciel

La fiabilité intervient à tous les niveaux
du cycle de vie du logiciel.

Elle est prise en compte dans chaque activité.

Problème :
Comment augmenter la fiabilité ?

Fiabilité du logiciel
Techniques pour la fiabilité

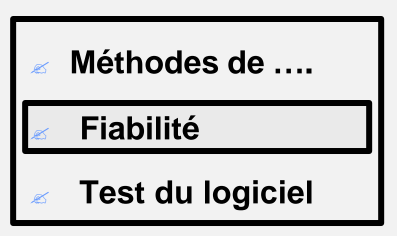


- Étudier les techniques de programmation “fiables”
- Étude des constructions “dangereuses”
- Catalogue de techniques utilisées et reconnues

- Lien avec la fiabilité logicielle.
  - Les utilisateurs supposent un logiciel fiable. Cependant, ils peuvent accepter les défaillances pour des applications non critiques
  - D'autres applications requièrent cependant un niveau de fiabilité impliquant l'usage de techniques de programmation spéciales

- Trois approches pour obtenir la fiabilité
  - Éviter les erreurs
    - Développement du logiciel de manière à éviter l'introduction d'erreurs
  - Détecter les erreurs
    - Organisation du processus de développement en vue de détecter et réparer les erreurs avant la livraison
  - Tolérances aux fautes
    - Conception du logiciel en vue d'éviter que les erreurs ne soient fatales au système complet

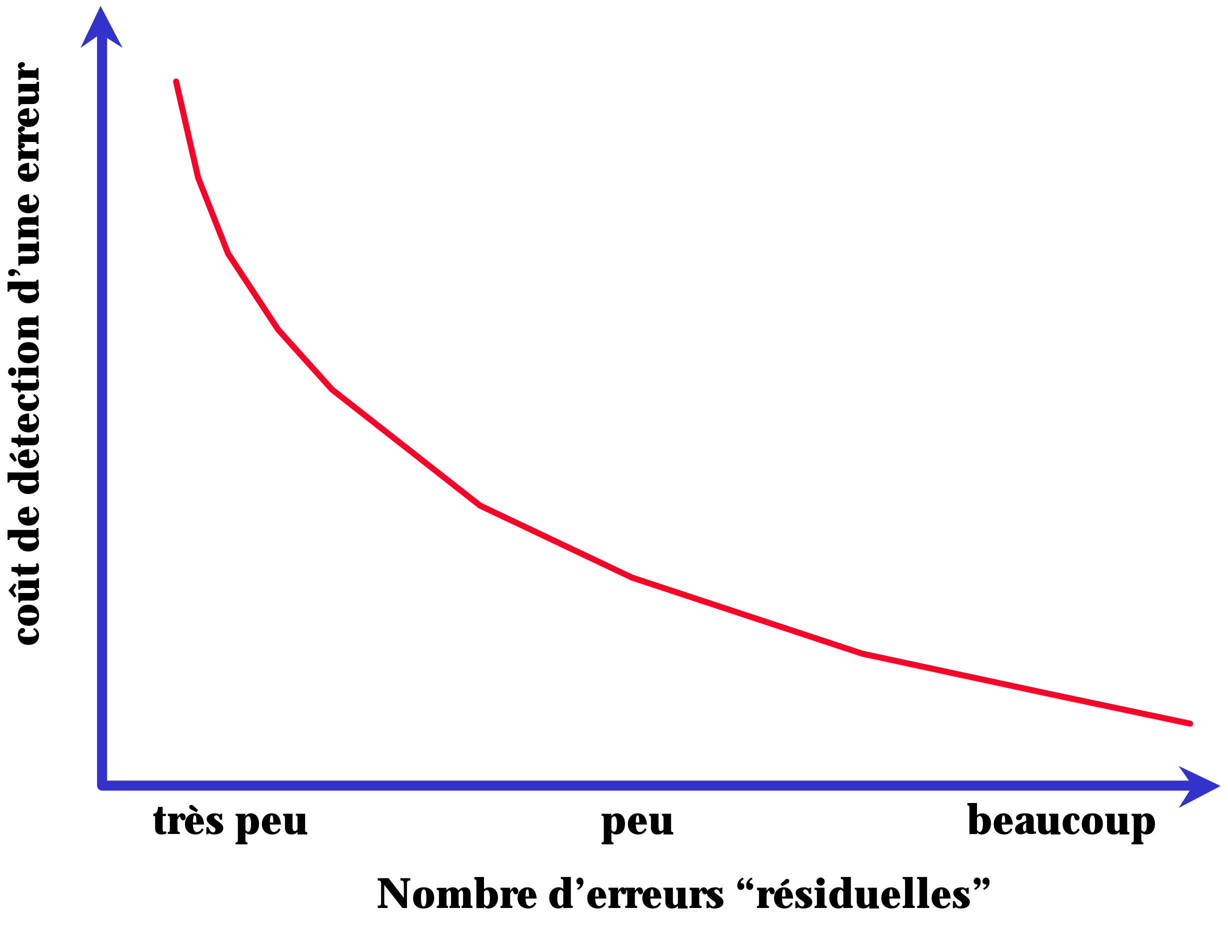
Fiabilité du logiciel
Logiciels «zéro défaut»



- Les techniques de Génie Logiciel permettent la production de “logiciel zéro-défaut”
- “Logiciel zéro-défaut”
  - Signifie “logiciel conforme à ses spécifications”
  - Ne signifie PAS “logiciel fonctionnant toujours parfaitement”
  - Des erreurs peuvent avoir été introduites dans la phase de spécification!!!
- Ces techniques ont en général un coût élevé
  - Utilisation dans un contexte exceptionnel
  - Il est parfois plus “rentable” d’accepter l’existence de défaillances.

Fiabilité du logiciel

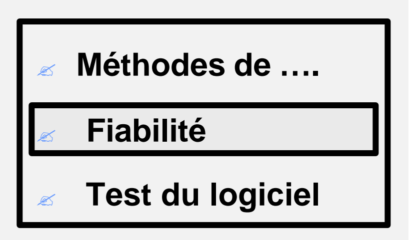
Logiciels «zéro défaut»



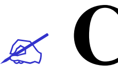
D'après I.Sommerville ©1995

Fiabilité du logiciel

Logiciels «zéro défaut»



- Développement de logiciels «zéro défaut»
  - Requiert des spécifications précises (par exemple formelles)
  - La protection de l'information à travers l'encapsulation des différentes notions est essentielle
  - L'usage de langages basés sur un fort typage et assurant des vérifications pendant l'exécution est vivement conseillé
  - L'usage de processus de relecture à différentes étapes est intensif
  - L'implication dans un processus de qualité est importante
  - Des systèmes de tests fins et intensifs sont toujours mis en œuvre

- **COBOL :** problèmes ...
- **FORTRAN**
  - Efficace à l’exécution
  - Constructions syntaxiques ambigües (ça s’arrange)
  - Beaucoup de notions très basiques
  - Utilisation probable de branchements
-  **C**
  - Efficace à l’exécution
  - Constructions syntaxiques ambigües (la, c’est désespéré)
  - Fonctions de réécritures dangereuses (macro, pas de vérification)
  - Typage faible (évolution dans la version ANSI)
  - Obligation d’utiliser des constructions dangereuses

Ada (95)

- Conçu pour le développement d'applications de grande taille (embarqué)
- Très strictement typé, langage objet
- Une tradition sur les grands systèmes (versions domaine public depuis 1995)
- Excellente portabilité au niveau source

C++

- Combine l'efficacité de C (langage de bas niveau) et les notions Objet
- Mieux typé que C mais moins bien qu'Ada
- Portabilité: attention aux fonctions évoluées (portabilité via l'OS)

Java

- Repose sur la notion de machine virtuelle => portabilité à l'exécution
- langage objet “pur et dur” (constructions réduites et bon typage)

- Notion apparente dès les annés 1970
  - Programmation sans "goto" (Disjkstra)
  - Préférer les boucles de type "while" (Hoare)
  - Eviter la modification d'indices dans les boucles énumératives
  - Conception descendante (diviser pour régner)
- Important: a ouvert un débat sur les techniques de programmation
  - Nouveaux langages
  - Constructions sûres
  - Chartes de programmation
- Identification des "constructions dangereuses" à éviter (si possible)

- Nombres décimaux
  - Imprécision inhérente aux formats de représentation choisis
  - Calculs absurdes, notion de convergence informatique
  - Comparaisons aberrantes
- Pointeurs
  - Permet la corruption de données (problème d'initialisation)
  - Évolution de données par “effet de bord”
  - Lisibilité réduite des programmes
- Allocation dynamique de mémoire
  - Contrôle de l'espace mémoire disponible
  - Problème de la désallocation

Fiabilité du logiciel

Encapsulation et typage

- Il est inutile de rendre une information accessible de manière trop aisée pour les raisons suivantes:
  - Réduire la probabilité d'une corruption "accidentelle" de données
  - Encapsuler les données de manière à concentrer leur manipulation en local (et les problèmes avec)
  - Localiser l'information pour réduire la probabilité de faire une erreur et accroître la lisibilité des programmes

1 **Les composants ne doivent accéder qu'au "minimum" strictement nécessaire pour leur implémentation**

2 **La représentation des données doit être cachée aux "utilisateurs" de ces données**

3 **Le système de typage permet d'accroître la lisibilité des programmes s'il est correctement utilisé**

4 **Il faut utiliser si possible les possibilités du langage dans ce domaine (Ada et C++)**

Fiabilité du logiciel
Objets et types abstraits

- Implémentations des objets
  - C++ => objets
  - Ada95 => paquetages
    système de classe avec héritage simple
- Les méthodes sont déclarées sous la forme de procédures et fonctions
- Le type est défini dans la partie “privée” de l’objet ou du paquetage

File d'entiers en Ada (Spécification)

```
package Queue is
        type T is private ;
        procedure Put (IQ : in out T; X: INTEGER);
        procedure Remove (IQ : in out T; X : out INTEGER);
        function Size (IQ : T ) return NATURAL;
private
        type Q_RANGE is range 0..99 ;
        type Q_VEC is array ( Q_RANGE ) of INTEGER ;
        type T is record
                The_queue: Q_VEC ;
                front, back : Q_RANGE ;
        end record;
end Queue;
```

D'après I.Sommerville ©1995

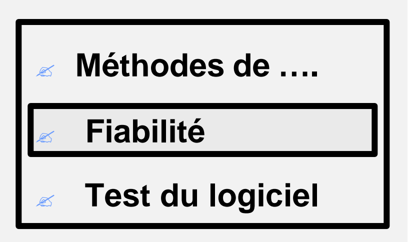
File d'entiers en C++ (Spécification)

```
class Queue {
public:
       Queue () ;
      ~Queue () ;
      void Put ( int x ) ; // adds an item to the queue
      int Remove(); // this has side effect of changing the queue
      int Size( ) ;  // returns number of elements in the queue
private:
           int front, back ;
           int qvec [100] ;
} ;
```

D'après I.Sommerville ©1995

Fiabilité du logiciel

Généricité



- Le mécanisme de macro est “dangereux”... on lui préfère le mécanisme de généricité qui est “mieux contrôlé”
- Caractéristiques de la généricité:
  - Mécanisme de “macro” avec contrôle de type
  - Permet le paramétrage d’unités de programmes (paquetages ou objets)
  - Accroître la réutilisabilité de composants dont le comportement est indépendant des données manipulées
  - Particularisation à travers une opération: l’instanciation
  - Des langages comme Ada ou C++ disposent de ce type de mécanismes

File générique en Ada (Spécification)

```
generic
        type ELEM is private ;
        type Q_SIZE is range <> ;
package Queue is
        type T is private ;
        procedure Put (IQ : in out T; X: ELEM );
        procedure Remove (IQ : in out T; X : out ELEM );
        function Size (IQ : in T ) return NATURAL ;
private
        type Q_VEC is array (Q_SIZE) of ELEM ;
        type T is record
                The_queue: Q_VEC ;
                Front : Q_SIZE := Q_SIZE'FIRST ;
                Back: Q_SIZE := Q_SIZE'FIRST ;
        end record;
end Queue;
```

D'après I.Sommerville ©1995

File générique en C++ (Spécification)

```
template
        <class elem>
class Queue {
public
        Queue ( int size = 100 ) ;
                // default to queue of size 100 elements
        ~Queue () ;
        void Put ( elem x ) ;
        elem Remove ( ) ; // this has side effect of changing queue
        int Size ( ) ;
private
        int front, back ;
        elem* qvec ;
} ;
```

D'après I.Sommerville ©1995

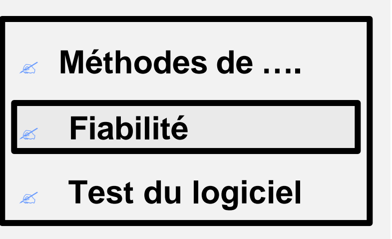
- Instanciation
  - Opération réalisée pendant la compilation, ce qui rend possible la vérification des types et prototypes
  - **en Ada**

```
type IQ_SIZE is range 0..49 ;  type LQ_SIZE is range 0..199 ;
package Integer_queue is new Queue (ELEM => INTEGER,
                                    Q_SIZE => IQ_SIZE ) ;
package List_queue is new Queue (ELEM => List.T,
                                 Q_SIZE => LQ_SIZE ) ;
```

  - **en C++**

```
//Assume List has been defined elsewhere as a type
Queue <int> Int_queue (50) ;
Queue <List> List_queue (200) ;
```

Fiabilité du logiciel



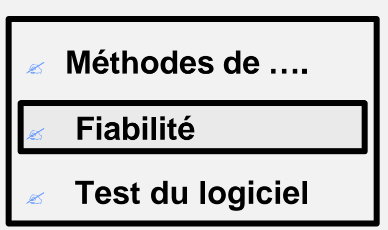
- Le logiciel doit, dans des situations critiques, être résistant aux pannes
  - Le système doit continuer à fonctionner malgré la défaillance
  - Même un système sans “erreur” (ou considéré comme tel) doit être tolérant aux défaillances
    - elles peuvent être liées à des erreurs de spécification
    - *la validation qui a abouti à ce niveau de confiance peut être mal faite*

Fiabilité du logiciel

Exercice

- *Décrire une pile générique en Ada puis en C++*
- *Instancier la pile sur des documents ou lettres dont vous définirez le type*

Fiabilité du logiciel
Actions en cas de défaillance

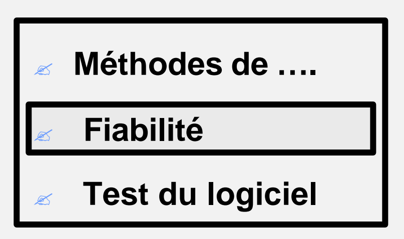


- (1) Détection de la défaillance
  - Le système doit être capable de détecter l'apparition de la défaillance
- (2) Évaluation des "dégâts"
  - Identifier les éléments du système qui sont affectés par la défaillance
- (3) Rattrapage de la défaillance
  - Restauration d'un état reconnu comme "sûr"
- (4) Réparation de l'erreur
  - Modification du système pour éviter d'autres apparitions de l'erreur
  - Lorsque les erreurs sont épisodiques, cela peut être inutile

D'après I.Sommerville ©1995

Fiabilité du logiciel

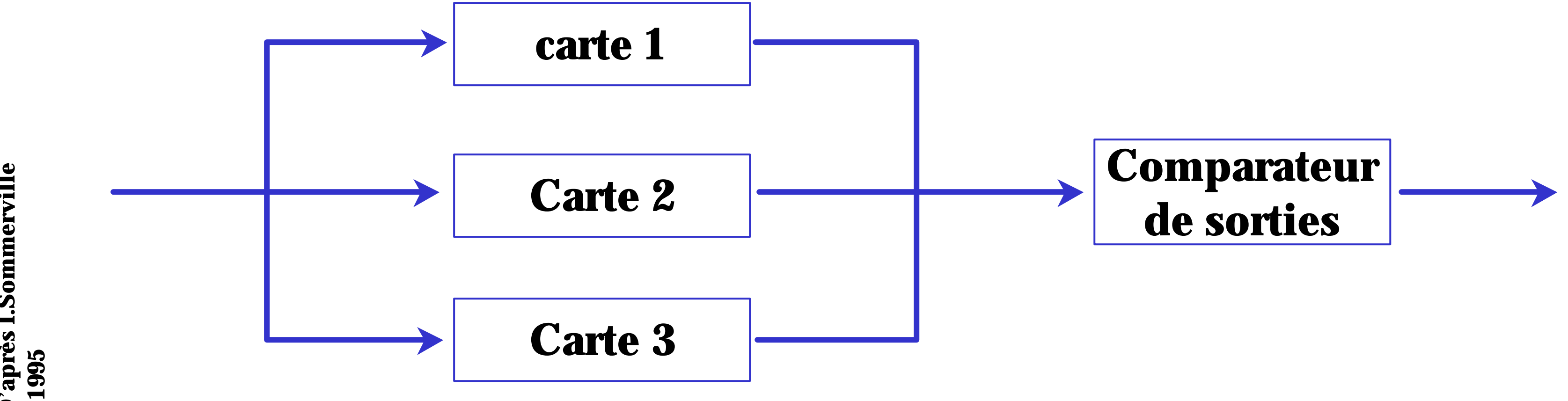
Occurrences



- La plupart des défaillances logicielles épisodiques dépendent d'une configuration de données particulière. On peut décider de redémarrer le système.

- En cas d'impossibilité, une reconfiguration dynamique peut-être intéressante (les composants sont reconfigurés sans arrêt du système)

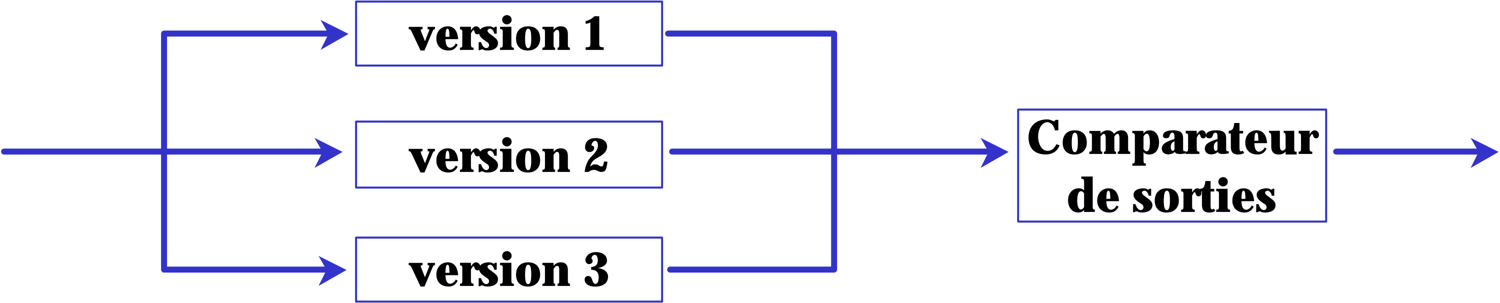
- La Triple-Modular Redundancy (TMR)
  - Trois composants répliqués identiques recevant les mêmes données
  - Comparaison des sorties, si une sortie est différente, le composant est écarté et on identifie une défaillance
  - Principalement utilisé pour les composants qui tombent en panne (pas d'erreurs de conception)



D'après I.Sommerville ©1995

Fiabilité du logiciel
Solution : N-versions Programming

- Principe
  - La même spécification est implémentée plusieurs fois, traitée en parallèle et la sortie majoritaire est sélectionnée.
  - Approche très commune (Exemple: Airbus A320)



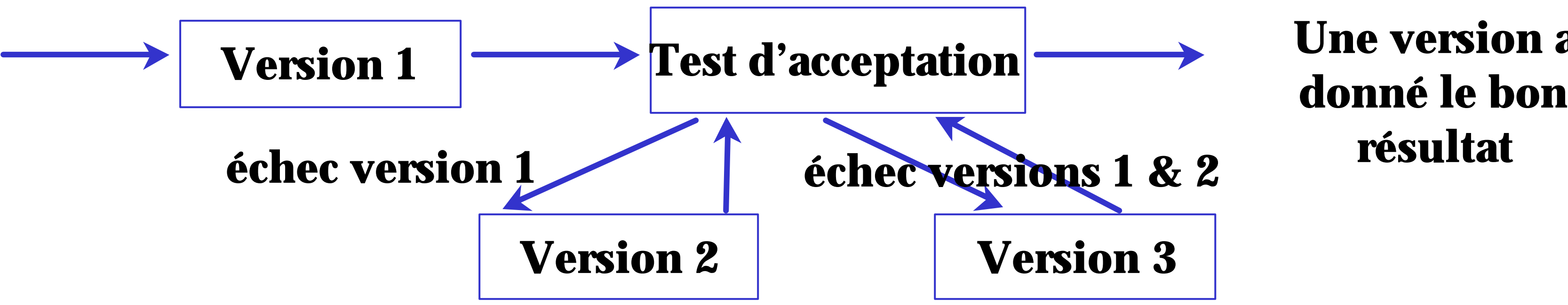
- Conception par différentes équipes
  - Attention aux erreurs de spécifications ou aux points mal définis qui peuvent être <u>mal interprétés de concert</u> par les différentes équipes

D'après I.Sommerville ©1995

Fiabilité du logiciel

Solution : Recovery Blocks

- Principe
  - Les versions sont éxécutées en séquence tant qu'un test de conformité sur la sortie refuse le résultat



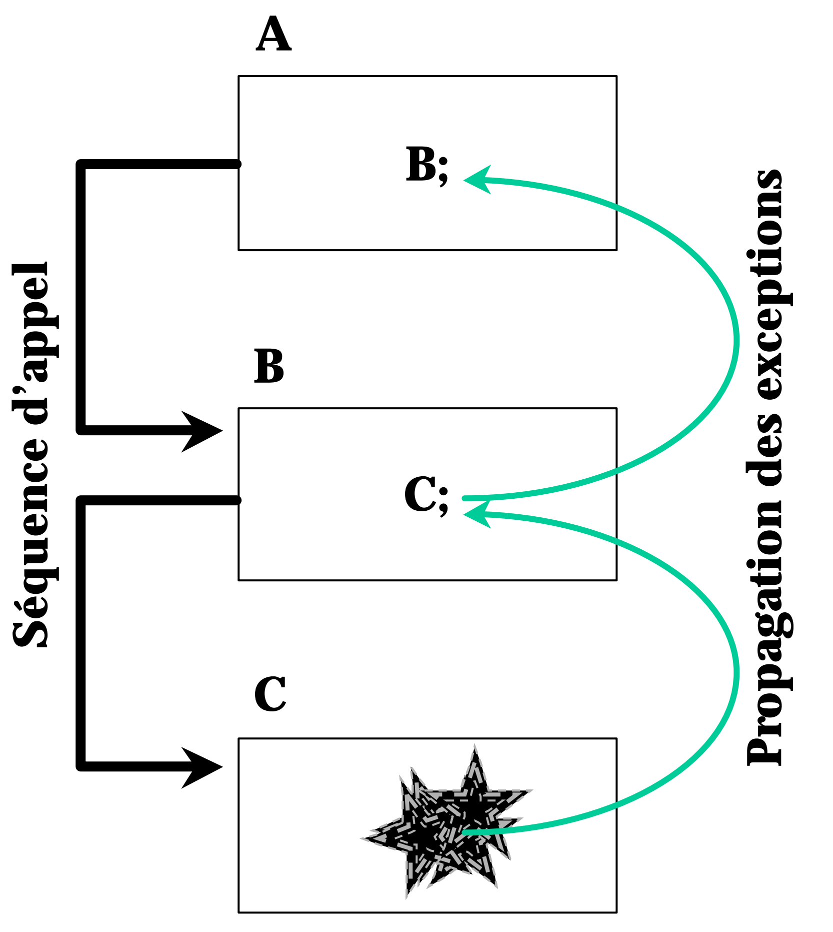
  - Différentes versions ? différents algorithmes
  - Le test d'acceptation est délicat à concevoir car il doit être indépendant des techniques utilisées dans les différentes versions
  - Les erreurs de spécification restent possibles

D'après I.Sommerville ©1995

Fiabilité du logiciel

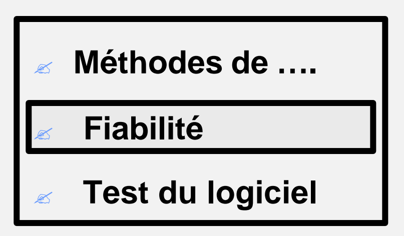
Solution : les exceptions

- Exception = “événement rattrapable”
- Il est possible de les rattraper lorsqu’elles surviennent
- Les points de rattrapages peuvent s’empiler les uns sur les autres
- Permettent de gérer le rattrapage des erreurs avec élégance et sans altérer l’écriture des programmes
- Implémenté en Ada, en C++ et en Java
- Inconvénient
  - Coûteux à poser



D’après I.Sommerville ©1995

Fiabilité du logiciel
Solution : forward recovery

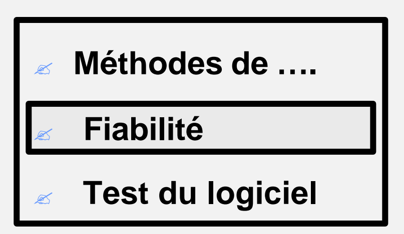


- Principe: essayer de “réparer” l’état du système après la défaillance
- Caractéristiques
  - Spécifique à l’application
  - la connaissance du domaine d’application est nécessaire pour effectuer les corrections nécessaires
- Moyens:redondance dans les structures de données
  - Rattrapage possible si la défaillance n’affecte pas le système au delà d’un certain seuil
  - Exemple type: les Systèmes de Gestions de Fichiers

D’après I.Sommerville ©1995

Fiabilité du logiciel

Solution : Backward recovery



- Principe: retrouver un état reconnu fiable du système
- Caractéristiques
  - Plus simple à mettre en œuvre que le “Forward recovery”
  - Moyens: sauvegarde d’états dit “fiables”
    - Systèmes transactionnels
    - Journalisation
    - Points de reprise

Fiabilité du logiciel

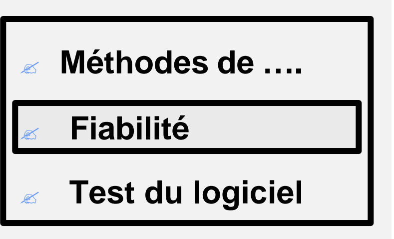
Autres solutions

- Programmation “défensive”
  - techniques de développement où l’existence d’erreurs non détectées peuvent exister dans le programme
  - Le programme contient du code permettant de détecter de telles fautes
    - test des paramètres
    - contrôles divers
- Prévention des défaillances
  - Le mécanisme de typage permet d’insérer des tests qui sont effectués lors de la compilation
  - Ajout de mécanismes fonctionnant pendant l’exécution du programme
    - Test sur des intervalles de valeurs
    - Définition d’assertions insérées dans le code et effectuant des vérifications pendant l’exécution

Évaluation des dégâts

- Analyse de l'état du système en vue d'évaluer l'étendue de la corruption liée à la défaillance
- Utilisation de "fonctions de validité" qui peuvent être appliquées aux éléments à évaluer
  - *"Checksums" pour évaluer si une donnée est corrompue ou non*
  - *redondance dans les pointeurs (ou les structures de données)*
  - *"Watch dog" pour surveiller les processus qui ne se terminent pas (ou se terminent mal)*

Fiabilité du logiciel



- La fiabilité du logiciel implique un travail actif des équipes de développement

- De nombreuses techniques existent

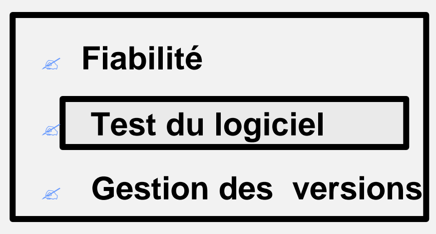
- Leur mise en œuvre engendre un coût supplémentaire sur le logiciel produit

- Il faut que ca en vaille la peine!!!

Plan du cours

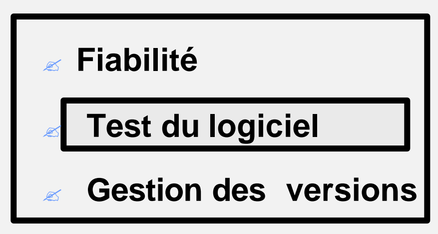
- VIII. Méthodes de conception de logiciels
- IX. Fiabilité du logiciel
- ? **X. Test du logiciel**
- **XI.** Gestion des versions
- **XII.** Réutilisation de logiciels
- **XIII.** Maintenance de logiciels
- **XIV.** Introduction aux méthodes formelles

Test du logiciel



- Définition de l'activité
- Identification de cas types à tester

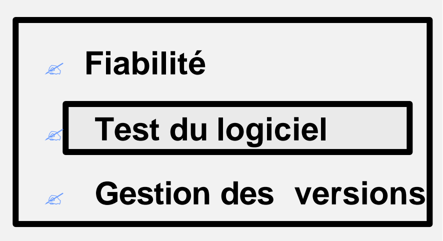
Test du logiciel



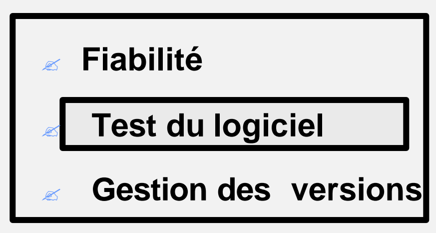
- Tester?
  - Moyen d'établir une certaine confiance dans un programme
  - Rechercher les cas de défaillances dans un programme (causes)
- Ce qu'il faut tester
  - les tests liés aux capacités d'un système ou à un composant
  - les régressions du logiciel (fonctions anciennement assurées qui ne le sont plus)
  - plus les "situations d'utilisation normale" que les "conditions limites"
- Seul un test exhaustif (impossible!!!) permet de garantir l'absence <u>d'erreurs</u> (avec pour conséquence, des <u>défaillances</u>) dans un programme

D'après I.Sommerville ©1995

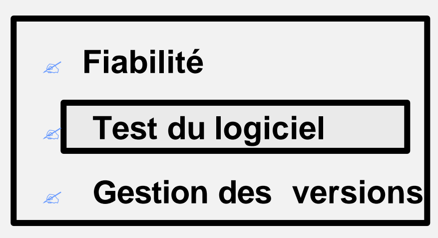
Test du logiciel



- Objectif du test
  - détecter des fautes ou des inadéquations d'un logiciel.
- Référence de test
  - spécifications
  - normes / règles concernant le code
  - documentation
- Définition
  - recherche d'inadéquations d'un logiciel avec des références établies

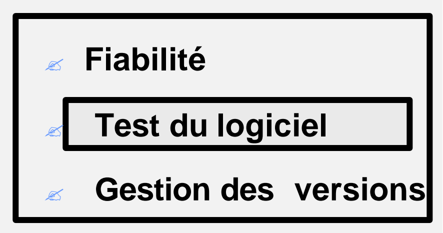


- Tests statiques
  - traitement du texte du logiciel sans exécution
- Tests dynamiques
  - exécution du logiciel sur un sous-ensemble choisi du domaine des entrées possibles
- Méthodes statiques
  - lecture croisée et inspection : vérification "collégiale" d'un programme ou d'une spécification
  - analyse d'anomalies : typage impropre, incohérence des interfaces de modules, portions de code isolées, mauvais usage de variables, etc. (assistance d'outils)
  - évaluation symbolique : simulation de l'exécution d'un programme sur des données symboliques



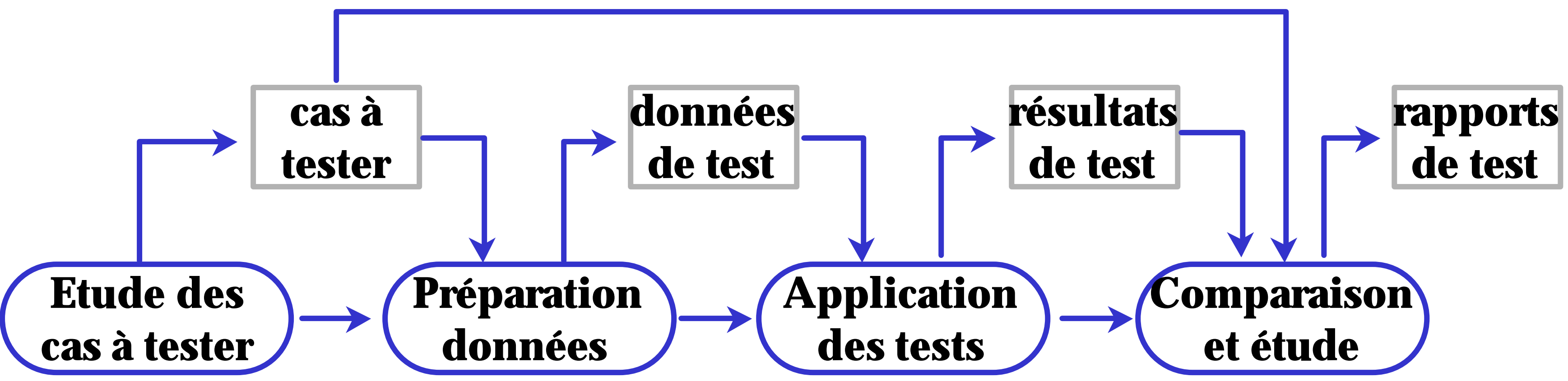
- **Test dynamique**
  - sélection d'un sous-ensemble des entrées possibles du logiciel = jeu de tests
  - soumission du jeu de tests pour exécution
  - dépouillement des résultats pour décider du succès ou de l'échec d'un jeu de tests : problème, pas toujours décidable, de l'oracle
  - évaluation de la qualité des tests effectuée, pour décider ou non l'arrêt du test

Test du logiciel
Techniques



- **Types de méthodes dynamiques :**
  - basées sur la structure et le texte du logiciel
    - test «boîte blanche» ou «boîte transparente»
  - basées sur les spécifications :
  - test fonctionnel
    - test "boîte noire"
  - basées sur les statistiques :
    - tirage aléatoire de jeux de tests

Test du logiciel
Procédure de test



- Tout au long du cycle de vie du logiciel
- Les tests ne sont jamais perdus
- Il faut toujours analyser le résultat d'un test
  - au moins une fois (la première)
  - étude "diférentielle" par la suite

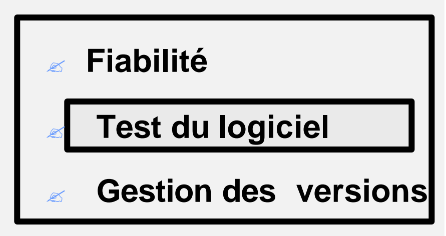
D'après I.Sommerville ©1995

Test du logiciel
Approche

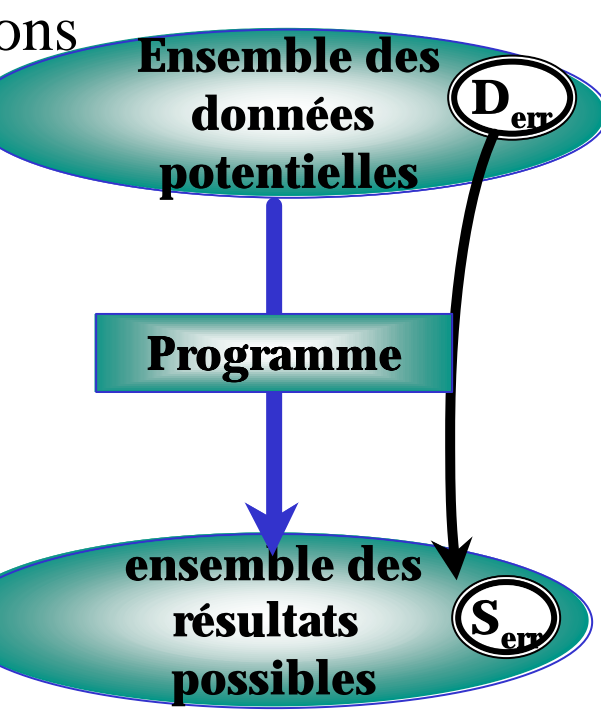
- Fiabilité
- **Test du logiciel**
- Gestion des versions

D'après I.Sommerville ©1995

Test du logiciel
Test en boîte noire



- Approche de test dans laquelle les programmes sont considérés comme des “boîtes noires”
- Les tests sont générés à partir des spécifications
  - interfaces
  - contraintes
  - etc...
- Ce type de test peut commencer très tôt dans le cycle de développement du logiciel
- Objectif:
  - Identifier les jeux de données et appels de primitives aboutissant à des résultats (erronés?)

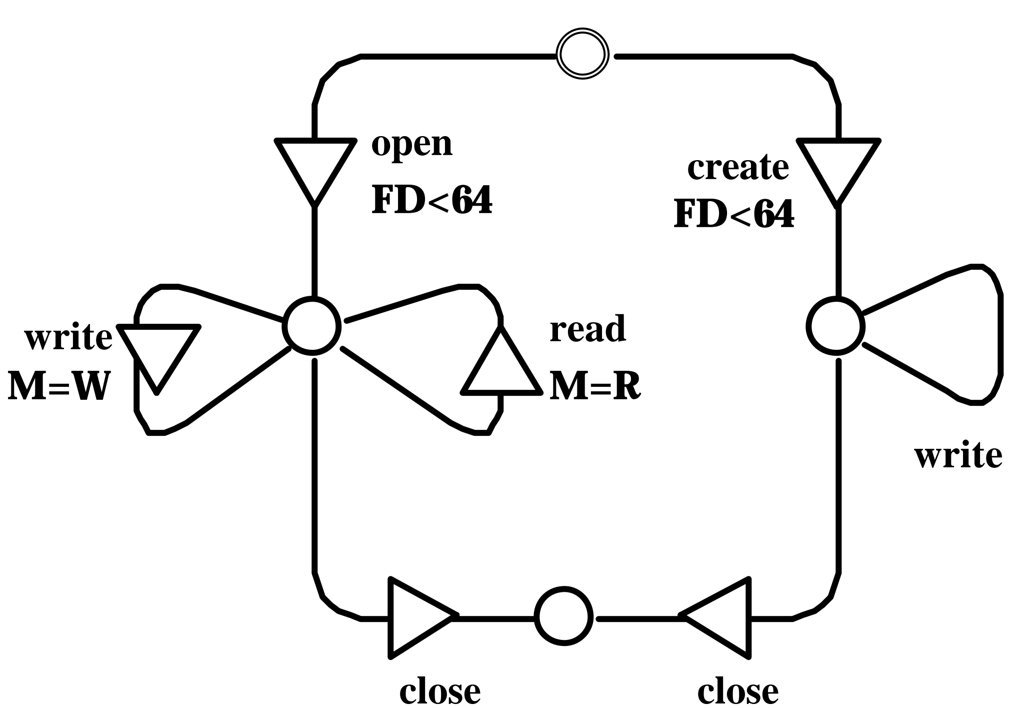


D’après I.Sommerville ©1995

Test du logiciel

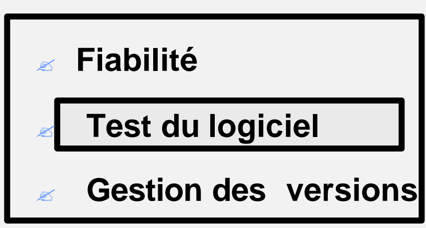
Exemple de test en boîte noire

- Utilisation de l'automate du "modèle opératoire"
  - Génération de séquences licites
  - Génération de séquences illicites
  - Repérer les bornes
- Séquences licite
  - open (W)/write()/close
  - open(R)/read()/close
  - create()/read()/close
- Séquences illicites
  - open (W)/read()/close
  - open(R)/write()/close
  - write()/close()
  - 65 open de suite

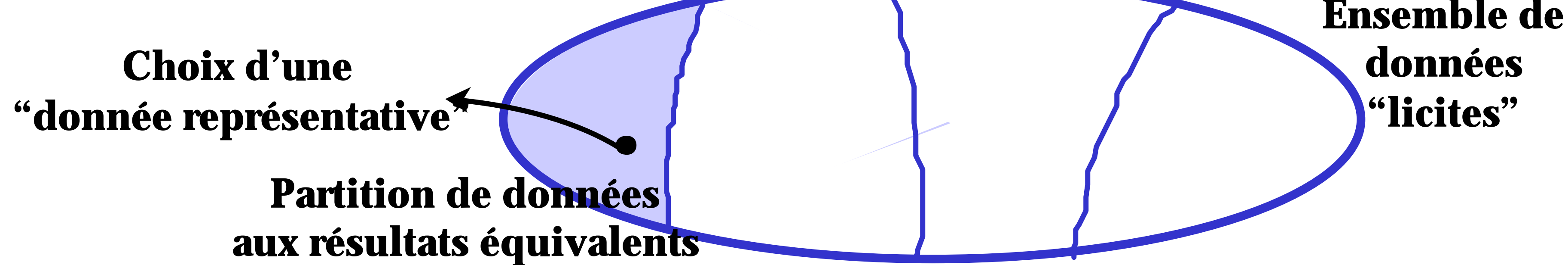


Test du logiciel

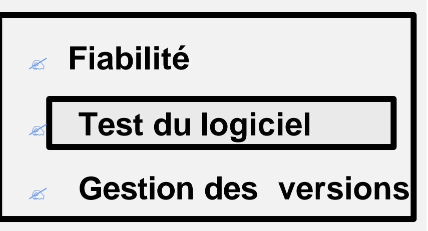
Données de test : ensemble minimal



- Recherche d'équivalences pour partitionnement



- Limitation de la taille du jeu de tests avec les critères:
  - cas de figures différents (dans les primitives)
  - types de comportements différents définis dans les spécifications
  - utilisation des informations de type invariant ou conditions permettant de vérifier qu'un traitement s'est "correctement" déroulé
  - etc..



Procedure de recherche d'un élément

- Prototype

```
procedure Cherche (Elem   : in ELEM ;
                   T      : in TAB_ELEM;
                   Trouve : in out BOOLEAN;
                   Indice : in out INDICE_TAB) ;
```

- Précondition
  - le tableau contient au moins un élément ^ T'First < T'last
- Postcondition
  - élément trouvé : trouve ^ T(indice) = Elem **OU**
  - élément non trouvé : not(trouve) ^ ? i ? ? T'First .. T'Last / T(i)]= Elem
- Intérêt des pré et post absentes dans UML ...

D'après I.Sommerville ©1995

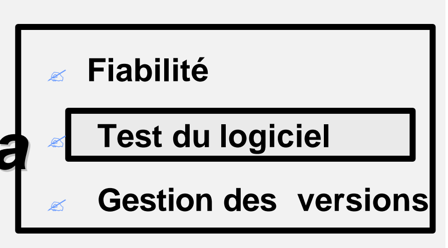
Test du logiciel
Application des règles du test

- Entrées conformes à la précondition
  - Elem = 3, T = [1 2 3 4] Trouve=TRUE, Indice=3
- Entrées non conformes à la précondition
  - Elem = 3, T = ? ?Trouve=FALSE, Indice=??
- Entrés avec l'élément dans le tableau
  - Elem = 3, T = [1 2 3 4] Trouve=TRUE, Indice=3
- Entrées avec l'élément absent du tableau
  - Elem = 12, T = [1 2 3 4] Trouve=FALSE, Indice=??

D'après I.Sommerville ©1995

Test du logiciel

Exploitation de la connaissance de la structure



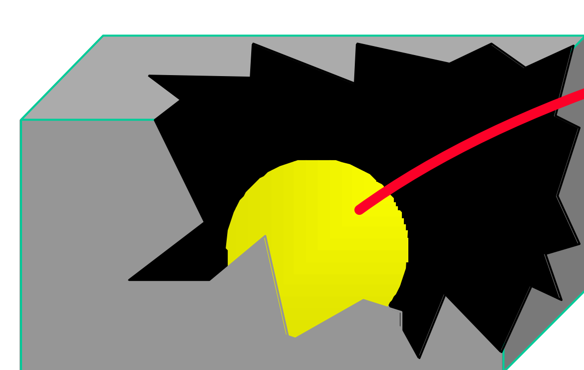
On cherche dans un tableau!!!

- Elément au "milieu" du tableau
  - Elem = 3, T = [1, 2, 3, 4] Trouve=TRUE, Indice=3
- Elément au début du tableau
  - Elem = 1, T = [1, 2, 3, 4] Trouve=TRUE, Indice=1
- Elément à la fin du tableau
  - Elem = 4, T = [1, 2, 3, 4] Trouve=TRUE, Indice=4
- Tableau avec un seul élément
  - Elem = 1, T = [1] Trouve=TRUE, Indice=1
  - Elem = 2, T = [1] Trouve=FALSE, Indice=??

D'après I.Sommerville ©1995

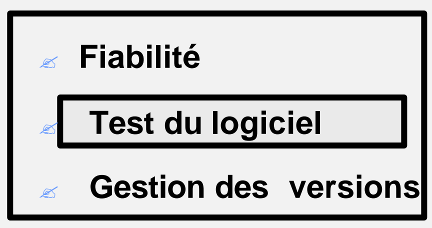
Test du logiciel
Test boîte blanche

- Aussi appelés “tests structurels”
- Exploitation de la connaissance des programmes
- Objectif: couvrir toutes les portions de code
  - Exemple (fonction de recherche: la recherche est implémentée selon la méthode “dichotomique”



D’après I.Sommerville ©1995

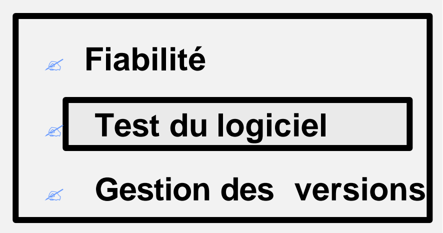
Test du logiciel
Application à l'exemple



- Tableau non trié
  - Elem = 3, T = [4,3,2,1] Trouve=FALSE, Indice=?? (ou erreur)
- Tableau trié
  - Elem = 3, T = [1, 2, 3, 4] Trouve=TRUE, Indice=3
- Donnée absente, supérieure à l'élément le plus lourd
  - Elem = 5, T = [1, 2, 3, 4] Trouve= FALSE , Indice=??
- Donnée absente, inférieure à l'élément le plus léger
  - Elem = 3, T = [1, 2, 3, 4] Trouve= FALSE , Indice=??
- Donnée présente, exactement au milieux (pair et impair)
  - Elem = 3, T = [1, 2, 3, 4, 5] Trouve=TRUE, Indice=3
  - Elem = 2, T = [1, 2, 3, 4] Trouve=TRUE, Indice=2
  - Elem = 3, T = [1, 2, 3, 4] Trouve=TRUE, Indice=3

D'après I.Sommerville ©1995

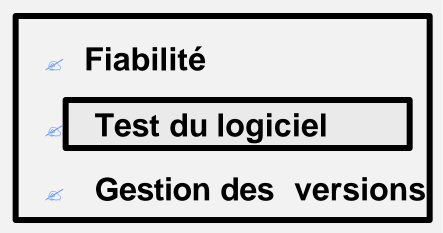
Test du logiciel
Quelques règles



- Tester le fait que les paramètres d'une procédure/fonction soient valeurs limites
- Toujours tester les paramètres de type pointeur avec des valeurs "nulles" (usage de bibliothèques style mallocdebug)
- Etudier les cas de figures pouvant amener un composant à se planter (comment se comporte-t-il face à une mauvaise utilisation)
- Insérer des délais supplémentaires (et variables) dans le test des programmes parallèles
- Tester à limite de saturation les systèmes parallèles basés sur la communication de messages
- Changer l'ordre d'activation des composants dans les systèmes répartis à communication à mémoire partagée

D'après I.Sommerville ©1995

Test du logiciel



- L'activité de test est une activité à part entière dans le développement d'une application
- De la qualité des tests, dépendra la qualité du produit (Qualité et quantité)
- La conception doit prendre en compte des considérations relatives au test du système
- On ne jette jamais un test on doit automatiser le passage des tests en grand nombre

Objectif :

Vers une technique d'application systématique de tests

Test du logiciel

(1)

Définition d'un jeu de données en entrée

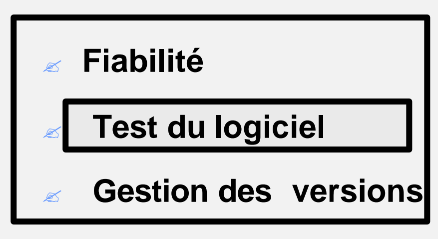
(2)

Application de ce jeu de données à un programme

(3)

Observation du résultat

Test du logiciel
Observation des résultats



- Décider du statut des résultat produits
  - Opération délicate
  - Nécessite une connaissance sémantique du système étudié
  - Difficilement automatisable
- Simplifier le problème de la décidabilité
  - Identifier les changements de comportement
    - Ce sont des “régressions” potentielles
  - Prévenir le responsable du test

Test du logiciel
Utilisation de l'oracle

(1)
Définition d'un jeu de données en entrée

(2)
Application de ce jeu de données à un programme

(3)
Comparaison du résultat à un <u>résultat de référence</u>

(4)
En cas de changement, prévenir le <u>responsable</u> qui <u>décide</u>

Alerte

Données en entrée
Responsable
"programme"
identique
Sorties produites
Référence
différent

Test du logiciel

Exemple d'une application sous Unix

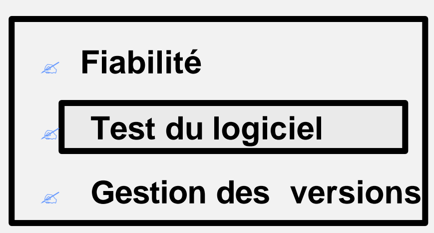
- Le “programme” à tester
  - ma_commande -input fichier_i -output fichier_o
- Hypothèses
  - Pas de sortie standard
  - Erreurs sur la sortie d'erreur standard (stderr)
- Ce que l'on teste
  - La conformité à la sortie de référence
  - L'absence d'éléments sur la sortie standard (hypothèse de base)
  - la conformité à la référence sur stderr
- On dispose
  - Référence de sortie, référence de sortie standard, responsable du test

Test du logiciel

Ecriture de l'oracle en shell


```
# lancement de la commande
ma_commande -i reference_i -o resultat > r_stdout 2> r_stderr
diff resultat reference_resultat >tmp
# evaluer le fichier resultat
if [ $? -eq 1 ]; then
   signal_change "changement dans le resultat" <responsable
fi
# evaluer la sortie erreur standard
diff r_stderr reference_stderr >tmp
if [ $? -eq 1 ]; then
   signal_change"changement dans sortie erreur" <responsable
fi
# verifier que rien n'est ecrit dans stdout
set `wc -c r_stdout`
if [ $1 -ne 0 ]; then
   signal_change "une écriture sur stdout" <responsable
fi
```

Test du logiciel
Application en série de l'oracle

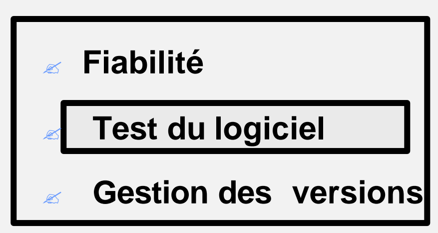


- Principe:
  - Déposer les fichiers de références et celui identifiant le propriétaire dans un répertoire différent pour chaque test
  - Construire un “applicateur” de l'oracle qui passe de répertoire en répertoire
- Cela donne:

```
for rep in *; do
   if [ -d $rep ]; then
      cd $rep
      sh oracle
      cd ..
   di
done
```

Test du logiciel

Actions du responsable de test

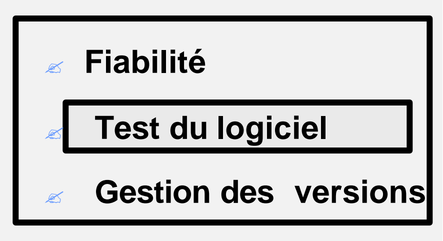


- Si le changement ne correspond pas à une régression
  - Mettre à jour la référence
- Si le changement correspond à une régression
  - Identifier la régression (numéro origine etc)
  - Corriger la régression
  - Construire un jeu de tests permettant de détecter la disparition de l'erreur (ou ajout de la raison à ce jeu de tests)
  - Réappliquer l'oracle à tous les tests

1> un test a toujours une raison

2> on repasse toujours tous les tests

Test du logiciel



- Ce principe est très souvent appliqué sur de gros projet
- Cette technique est appliquée (projet GNAT)
- Cela suppose que les tests soient automatisables
- Il faut prendre en compte cet aspect lors de la conception de l'application (prévoir un mode "batch"?)

Plan du cours

- VIII. Méthodes de conception de logiciels
- IX. Fiabilité du logiciel
- X. Test du logiciel
- ? **XI. Gestion des versions**
- **XII.** Réutilisation de logiciels
- **XIII.** Maintenance de logiciels
- **XIV.** Introduction aux méthodes formelles

Gestion de versions

- Pourquoi gérer différentes versions d'un programme
- Comment faire?
- Illustration des problèmes à travers des exemples

Gestion de versions

Introduction

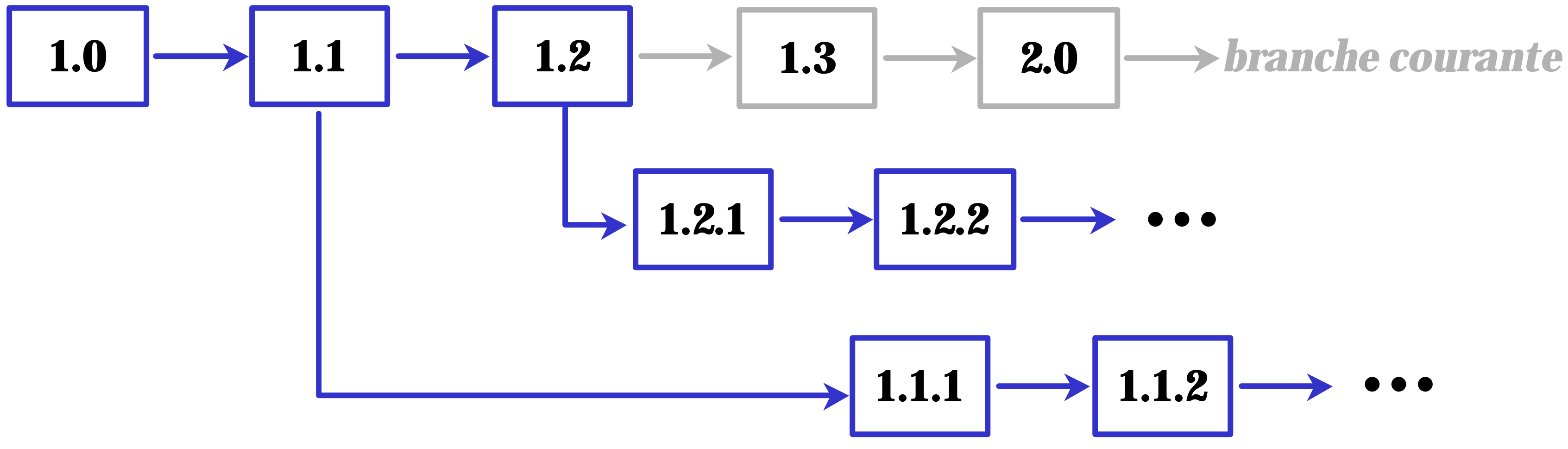
- Version d'une application
  - Ses <u>sources</u> et sa <u>documentation</u> à un <u>instant donné</u>
  - Les gros logiciels se construisent par évolution de versions successives
- A quoi ca sert?
  - Définir des versions "stables"
  - Revenir à des états stables antérieurs
  - Reproduire des phénomènes identifiés dans des situations particulières

Gestion de versions

Organisation des versions

- Fiabilité
- **Gestion des versions**
- Réutilisation

- Suite linéaire de versions
- Arbres de dérivations



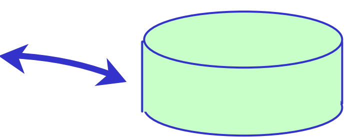
Gestion de versions

Opérations sur les versions

- Enregistrer (check in)
  - Insérer une nouvelle version d'un élément en lui attribuant une nouvelle version
  - Possibilité de déverrouillage (autoriser les extractions)
- Extraire (check out)
  - Extraire une version (la dernière?) d'un élément
  - Possibilité de verrouillage (pas d'autre extraction)

Ces opérations se justifient!!!

Base de référence



- Autres commandes...
  - Extraire l'historique des modifications sur un élément
  - Verrouiller/déverrouiller
  - etc...

Gestion de versions

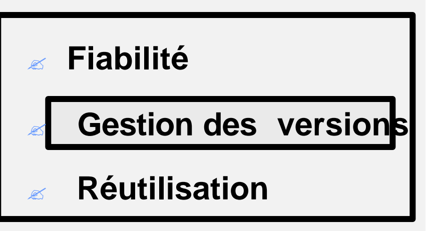
Versionnement des grosses applications

- Version d'une application = état d'un ensemble d'éléments
  - Et s'il y a beaucoup de fichiers?
    - sources de programmes
    - documentation
    - autres...
- Qu'est-ce que la version d'un élément
  - Un état "stable" ou non
- Qu'est ce que la version d'une application
  - L'état de tous ses éléments

Version d'une application ? "somme" de versions d'éléments

Gestion de versions

RCS : Revision Control System



- Numéro de version d’un élément = quadruplet <v1.v2.v3.v4>
  - Stockage des éléments (fichiers) dans une base de référence
- opération CI (check in) pour enregistrer un source
  - Incrémente automatiquement le dernier quadruplet
  - On peut forcer un nouveau numéro (pour effectuer une dérivation)
  - Édition d’un descriptif de la version introduite
- opération CO (check out) pour extraire un source
  - Récupère par défaut la dernière version de la branche courante
  - On peut spécifier une autre version
- Système de verrouillage élémentaire
  - On peut verrouiller avec un CO et déverrouiller avec un CI

Gestion de versions

RCS : Revision Control System

- Insertion automatique d'informations dans un fichier
  - Insertion de directives dans les fichiers
  - Remplacement de ces directives au niveau des CI et CO
  - Exemples:
    - -- $Author$ → -- $Author: toto $
    - -- $Revision$ → -- $Revision: 1.3.5.1 $
    - -- $Date$ → -- $Date: 1998/02/15 16:28:52 $
- Autres commandes
  - Construction d'une "version logique" (état "figé" de la base de fichiers)
  - Historique des évolutions d'un fichier
  - Différences entre deux versions
  - Manipulation spécifiques de verrouillage ou de déverrouillage
  - Synthèse de deux révisions (issues d'arbres différents) :-|!!!

Gestion de versions

RCS : Revision Control System

- Technique utilisée:
  - Stockage différentiel : $V_n = V_0 + ?\ V_1 + ?\ V_2 + ?\ V_3 + \ldots ?\ V_{n-1}$
  - Deux états pour un fichier:
    - **fichier** **?** la révision extraite (courante?)
    - **fichier,v** **?** le référentiel (original + modifications + raisons)
- Avantages
  - Simple, “canonique” et souple
- Inconvénients
  - Lourd quand on gère beaucoup de fichiers
  - Pas de règles de structuration précises (on peut “tout” faire)

Gestion de versions
CVS : Cooperative Versionning System

- Surcouche de RCS
  - Grosses applications multi-utilisateur
- Intègre et “cache” des “règles de bon usage”
  - Une base de référence unique organisée en répertoires
  - Chaque développeur possède une propre copie de cette base
    - il peut la modifier à loisir
    - il peut gérer des versions “localement” hors resynchronisation avec la base principale
  - Mécanisme de conciliation a posteriori
    - Hypothèse: les modifications simultanées sur un même fichier sont rares

Gestion de versions

CVS : Cooperative Versionning System

- Une seule commande:
  - CVS [options] fonction [options] [liste de fichiers]
- Fonctions
  - checkout ? extraire une copie locale pour un utilisateur
  - commit ? publication des modifications effectuées dans la base locale
  - update ? mise à jour dans la copie locale des modifications des autres développeurs (publiées par commit)
  - remove ? Suppression d'un fichier dans la base locale (et dans la base de référence lors du prochain commit)
  - add ? Ajout d'un nouveau fichier dans la base locale (et dans la base de référence lors du prochain commit)

Gestion de versions

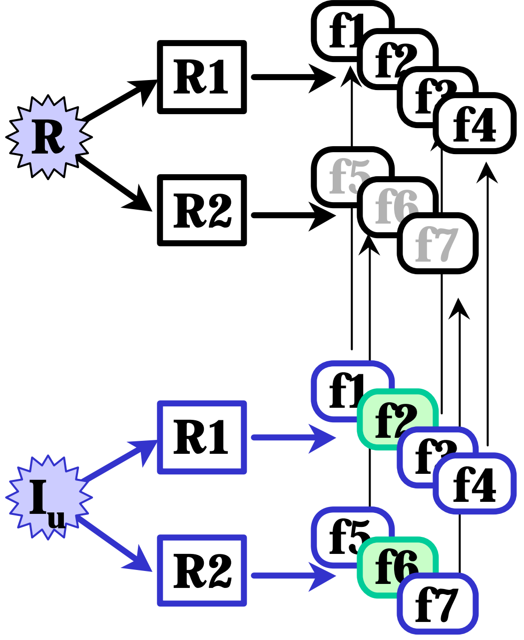
CVS : Cooperative Versionning System

- Avantages
  - Plus facile d'utilisation que RCS
  - Philosophie d'utilisation "encapsulée" à travers une utilisation simple
  - Très adapté au développement d'applications sur plusieurs sites
- Inconvénients
  - Décidabilité de certaines modifications croisées (même si c'est rare)
  - Adapté à la gestion de sources ASCII

Gestion de versions

FVS : Framekit Versionning System

- Surcouche de RCS
  - Grosses applications multi-utilisateur centralisées sur un site
- Intègre et “cache” des “règles de bon usage”
  - Une base de référence unique structurée en répertoires
  - Version = <v1.v2.v3> (v4 de RCS utilisé en interne)
  - Chaque développeur possède “image” de cette base
    - “liens” vers des fichiers de référence
    - copie locale de fichiers importés et verrouillés
  - Exclusion par verrouillage, pas de synchronisation a posteriori
    - Hypothèse: les modifications simultanées sur un même fichier ne doivent PAS pouvoir exister

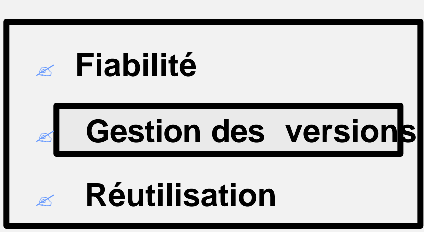


Commandes

- **get_file** ? importer un fichier depuis la base de référence le fichier est verrouillé pour les autres développeurs
- **push_file** ? sauver une version dans la base de référence sans publication aux autres usagers (ou création)
- **publish_file** ? sauver une version dans la base de référence avec publication aux autres usagers (ou création)
- **make_version** ? Construire une nouvelle version stable pour l'application
- **add_user** ? ajouter un usager certifié
- **add_directory** ? ajouter un répertoire à la structure (sur la référence et chez tous les usagers)

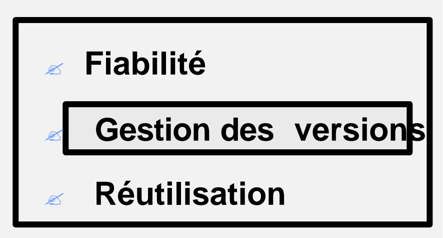
Gestion de versions

FVS : Framekit Versionning System



- Avantages
  - Plus facile d'utilisation que RCS
  - Philosophie d'utilisation "encapsulée" à travers une utilisation simple
  - Très adapté au développement d'applications sur un site (accès de la base de référence via NFS)
- Inconvénients
  - Verrouillage strict
  - Adapté à la gestion de sources (ASCII)

Gestion de versions



- La gestion de version est un travail <u>nécessaire</u> et <u>important</u> dans la vie d'un projet
- La gestion de version implique un travail actif des équipes de développement (documentation et publication)
- De nombreux outils existent (intégrés dans un AGL)
- Il faut impérativement s'appuyer sur de tels outils

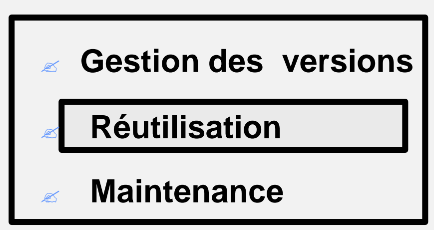
Plan du cours

- VIII. Méthodes de conception de logiciels
- IX. Fiabilité du logiciel
- X. Test du logiciel
- XI. Gestion des versions
- ? **XII. Réutilisation de logiciels**
- **XIII.** Maintenance de logiciels
- **XIV.** Introduction aux méthodes formelles

Éléments pour la réutilisation

- Qu'est-ce que la réutilisation de composants ?
- Approche de réutilisation
- Avantages et inconvénients

Éléments pour la réutilisation



- Objectif: construction de logiciel à partir de composants réutilisables
- Proposition de classification
  - Réutilisation d'applications
    - S'applique à l'intégralité d'un système
    - Généralement appelé "portage"
  - Réutilisation de sous-systèmes
    - Intégration de composants logiciels de grande taille
  - Réutilisation de modules ou de classes
    - Le composant réutilisé offre des fonctions de base
  - Réutilisation de sous-programmes (fonction/procédures)
    - exploitation d'algorithmes déjà éprouvés et efficaces (bibliothèques mathématiques)

D'après I.Sommerville ©1995

Éléments pour la réutilisation

- Réutilisation d'application (portage)
  - C'est une pratique commune
  - On en parle de plus en plus au niveau du développement (cf Metrowerks
- Réutilisation de sous-systèmes ou de modules
  - Pratique moins systématique et plus informelle
  - Certaines sociétés/services disposent de bibliothèques de composants “de base”
- Réutilisation de sous-programmes
  - Très commun dans certains domaines (existence de bibliothèques spécifiques)
  - C'est une des raisons de la persistance de langages comme FORTRAN ou COBOL

D'après I.Sommerville ©1995

Éléments pour la réutilisation

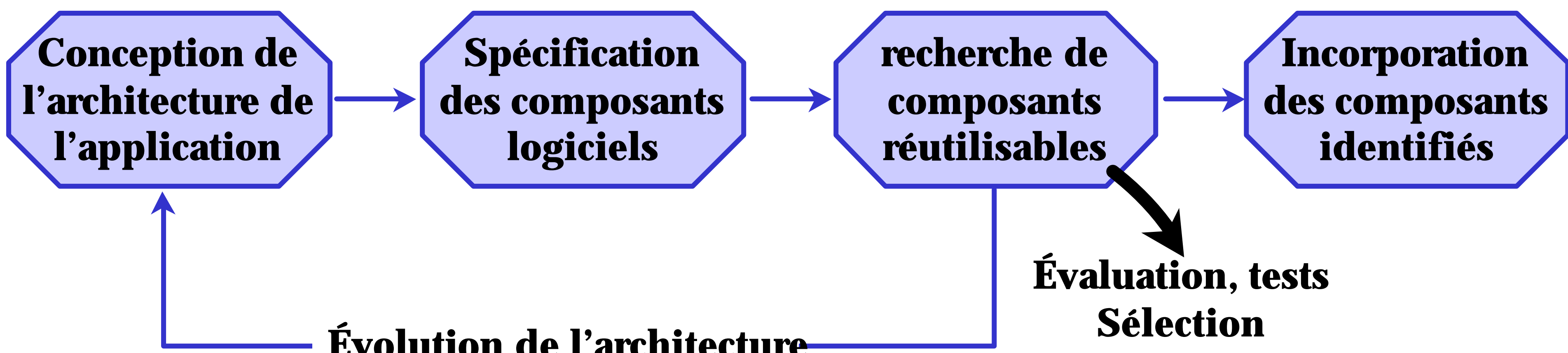
- Développement de logiciel par réutilisation de composants
  - Besoin de classification des composants
  - Besoin de Bibliothèques de composants (commerce?)
- Développement de logiciel réutilisable
  - Comment faire (généricité, système de macros etc...)
  - Bien penser son composant
  - Imaginer son usage dans un contexte complètement différent
- Portage d'application
  - Comment penser la structure du système en vue d'accroître sa portabilité

D'après I.Sommerville ©1995

Éléments pour la réutilisation

- Avantages
  - Moins de composants nouveaux à spécifier
  - Importation de technologie (pas à développer)
  - Diminution du coût de développement
  - Diminution de la durée du développement
  - Meilleure confiance dans la fiabilité du système produit
  - Réduction des risques
- Inconvénients
  - Quand réutiliser, que réutiliser
  - Attention aux conditions de réutilisation
  - Attention à l'adaptation du logiciel réutilisé (cf Socrate)
  - Maîtrise de l'ouvrage moindre (contrôle sur les composants?)

Éléments pour la réutilisation



D'après I.Sommerville ©1995

- Attention au “rebouclage”
  - L'architecture globale du système ne doit pas être touchée
  - Les hypothèses de fonctionnement des composants réutilisés peuvent cependant avoir un impact sur la conception détaillée de certaines parties du système

Éléments pour la réutilisation

- Utilisation intensive de base de données de composants réutilisables
  - de plus en plus sur Internet
  - aussi via des entreprises spécialisées
- Ceux qui réutilisent doivent comprendre le composant
- Ceux qui réutilisent doivent avoir confiance dans le composant (fonctionnement, adéquation aux besoins)
- Les composants doivent être documentés
  - Comment s'en servir
  - Énoncé clair des limites et des contraintes du système

D'après I.Sommerville ©1995

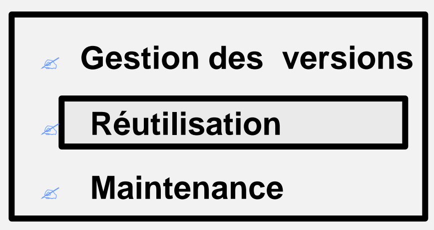
Éléments pour la réutilisation

Évaluation, tests
Sélection

identification des fonctions du système → recherche de composants réutilisables → conception architecturale → spécification des composants originaux

recherche de composants réutilisables ⇄ modification des fonctions d'après les composants identifiés

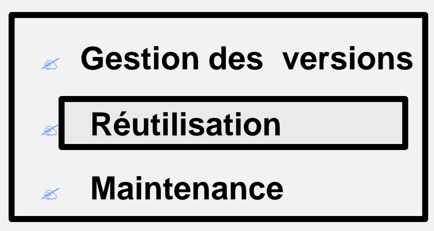
Éléments pour la réutilisation



- Les bénéfices de la réutilisation sont difficiles à quantifier
- les AGL ne supportent en général pas la réutilisation
  - Ou alors, de composants déjà définis dans l'AGL
  - Intégration avec un système de bibliothèque délicate
- Les développeurs aiment "réinventer la roue"
- Les techniques de classification actuelles sont encore immatures. Le coût de recherche d'un composant reste élevé
- Cependant...
  - Technique utilisée dans l'industrie embarquée (automobile, électroménager etc...) ? domaines proches du matériel?
  - Technique utilisée dans des domaines à valeur scientifique élevée

D'après I.Sommerville ©1995

Éléments pour la réutilisation



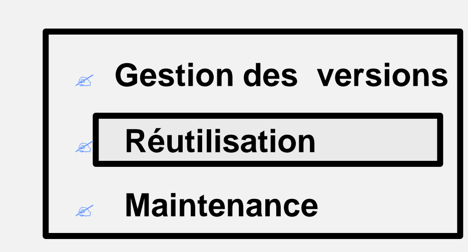
- La réutilisation est une technique
- La réutilisation doit-elle devenir un objectif en soi?
- La réutilisation est encore assez restreinte
- C’est sans doute une technique d’avenir

Éléments pour la réutilisation

- Problème : faire du logiciel réutilisable
  - Qu'est ce que développer du logiciel réutilisable ?
  - La notion de middleware
  - éléments d'architecture

Éléments pour la réutilisation

Développer en vue de réutiliser

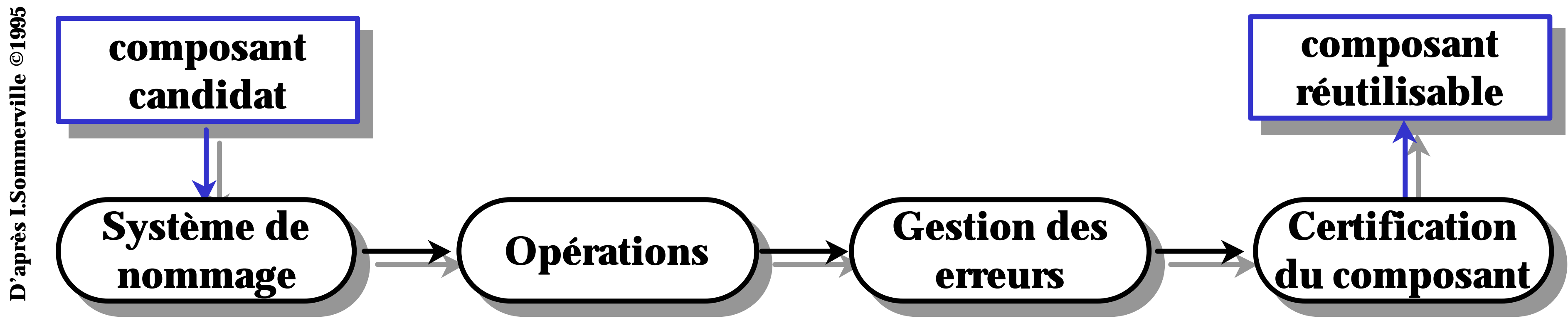


- Les composants logiciels ne sont en général pas “directement” réutilisables
- Développer du logiciel réutilisable implique
  - Penser différemment
    - généraliser l’usage du composant
    - documenter en pensant aux limites
    - prévoir différents domaines d’application
- A qualité égale, le coût d’un composant logiciel réutilisable est plus élevé que celui d’un composant logiciel non réutilisable
- La réutilisabilité peut se faire au détriment d’autres critères
  - rapidité d’exécution ou taux d’occupation mémoire

Éléments pour la réutilisation

Que faire pour rendre un composant réutilisable ?

- Système de nommage
  - modifier les noms pour les rendre indépendants de la "réalité" initiale
- Opérations
  - Certaines opérations doivent souvent être ajoutées pour étendre les fonctions du composant au delà de l'idée initiale
- Gestion des erreurs
  - La gestion des erreurs doit être beaucoup plus fine et prendre en compte des cas d'erreurs liés à l'ignorance des choix de réalisation
  - Certification
  - Le composant doit être certifié



D'après I.Sommerville ©1995

Éléments pour la réutilisation

Généraliser un composant

- Implique en général l'ajout d'opérations pour assurer que le domaine d'application (étendu et généralisé) est bien couvert
- En général, on entoure des structures de données abstraites par

Constructeurs/Destructeurs

Opérations de manipulation

Ensemble de types abstraits exportés

Structure de données abstraite

Itérateurs

Opérations de comparaison

Opérations d'E/S

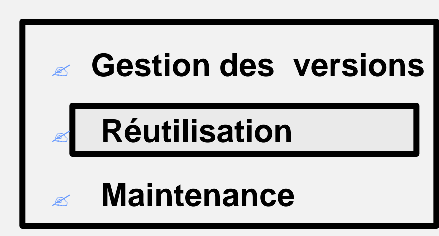
D'après I.Sommerville ©1995

Éléments pour la réutilisation

Quelques règles

- Exploiter la généricité (templates) pour implémenter les structures de données
- Interdire l'affectation autrement que par des opérations spécifiques (exportées)
- Interdire la comparaison d'égalité autrement que par des opérations spécifiques (exportées)
- Penser à un mécanisme de diagnostic sur les opérations proposées (comment se sont-elles déroulées)
- Minimiser la quantité d'informations exportées
  - Utilisation des exceptions (si possible)
  - Implémenter les opérations pouvant échouer sous forme de procédures avec code de retour

Éléments pour la réutilisation
Exemple de composant



- Unité générique
  - ce que contiennent les noeuds est passé en paramètres
- Principes
  - L'arbre est une structure opaque:
    - pointeur sur un nœud de position courante
    - liens vers fils et père d'un nœud
  - Manipulation de "chemins" pour se repérer dans les arbres
  - Exportation d'exception pour signaler les problèmes (Ada)
- Deux types de primitives de manipulation
  - primitives "sûres" avec duplication entre paramètres et résultats
  - primitives "avec effet de bord" qui modifient les paramètres
    - Préfixée "SE_"

Paramétrage (généricité)

```
-- $Author: fko $
-- $Revision: 1.4.1.1 $
-- $Date: 1995/04/26 16:29:55 $

----------------------------------------------------------------------
-- Ce paquetage permet la gestion d'arbre N-aires. Des liens de pere
-- a fils existent mais aussi des liens du fils vers le pere.
----------------------------------------------------------------------
-- Dans la structure de donnee decrivant un arbre N-aire, on a un
-- "pointeur sur l'element courant". Un certain nombre de fonctions
-- concernent cet element courant. Un certain nombre de fonctions permettent
-- de modifier l'element courant
----------------------------------------------------------------------

generic
   -- Le type contenu dans un noeud
   type ELEMENT is private;
   -- Un pointeur sur ce qui est contenu dans le noeud
   -- (pour effectuer des modifications sans prendre-retirer)
   type PT_ELEMENT is access ELEMENT;
package GENERIC_N_ARY_TREE_MANAGER is ...
```

Types abstraits & exceptions exportés

```
package GENERIC_N_ARY_TREE_MANAGER is

   N_ARY_TREE_ERROR : exception;

   -- Le type Arbre N-aire

   type N_ARY_TREE is private;


   -- le type chemin dans un arbre (utile pour les primitives de
   -- parcours avec actions).

   type PATH_TREE is array (NATURAL range <>) of NATURAL;

   EMPTY_TREE : constant PATH_TREE (0 .. 0) := (0 => 0);

   PATH_TREE_ERROR : exception;
```

Éléments pour la réutilisation

Exemple de composant

- Gestion des versions
- **Réutilisation**
- Maintenance

Manipulation de chemins

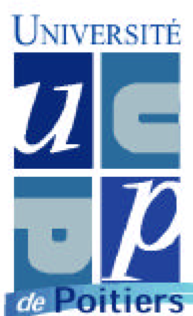
```
-----------------------------------------------------------------------
-- primitive de gestion du chemin pour se positionner
-- directement dans un endroit de l'arbre.

-- verifie si un chemin est vide
-- attention ce n'est pas le chemin pour aller a la racine
-- il s'agit d'une valeur "de repos" sans sens
function IS_EMPTY (PATH : in PATH_TREE) return BOOLEAN;

-- verifie si un chemin est valide, c.a d. qu'il ne provoque pas d'erreur
-- lors d'un positionnement ( un chemin vide n'est pas valide )
function IS_VALID (ON_TREE : in N_ARY_TREE;
                   PATH    : in PATH_TREE) return BOOLEAN;

-- positionne le noeud courant par rapport au chemin depuis la racine
function SET_CURRENT (TREE      : in N_ARY_TREE;
                      WITH_PATH : in PATH_TREE) return N_ARY_TREE;

function SE_SET_CURRENT (TREE      : in N_ARY_TREE;
                         WITH_PATH : in PATH_TREE) return N_ARY_TREE;
```



Constructeurs/Destructeurs & manipulation d'arbres

```
  -- Creation, destruction d'un arbre N-aire
function CREATE_EMPTY_N_ARY_TREE return N_ARY_TREE;
function CREATE_N_ARY_TREE (E : in ELEMENT) return N_ARY_TREE;
procedure DESTROY_N_ARY_TREE (TREE : in out N_ARY_TREE);
 -- Duplication d'un arbre.
 -- REMARQUE : la position de l'element courant reste coherent
 -- (meme noeud) dans l'arbre recopie.
function DUPLICATE_TREE (THE_TREE : in N_ARY_TREE) return N_ARY_TREE;
-- ajoute un element (forcement feuille) ou un arbre comme sous-arbre de
-- l'autre. L'ajout se fait a droite (ADD_SON_TO_CURRENT_NODE) ou
-- a gauche (ADD_FIRST_SON_TO_CURRENT_NODE)
-- ATTENTION, la position du noeud courant reste inchangee dans l'arbre resultat.
function ADD_SON_TO_CURRENT_NODE (THE_TREE       : in N_ARY_TREE;
                              THE_SON_VALUE : in ELEMENT) return N_ARY_TREE;
function ADD_SON_TO_CURRENT_NODE (THE_TREE : in N_ARY_TREE;
                              THE_SON   : in N_ARY_TREE) return N_ARY_TREE;
function ADD_FIRST_SON_TO_CURRENT_NODE (THE_TREE       : in N_ARY_TREE;
                              THE_SON_VALUE : in ELEMENT) return N_ARY_TREE;
function ADD_FIRST_SON_TO_CURRENT_NODE (THE_TREE : in N_ARY_TREE;
                               THE_SON   : in N_ARY_TREE) return N_ARY_TREE;
```

Manipulation d'arbres (suite)

```
-- Meme chose mais avec "Side Effect" (sans duplication des parametres)
function SE_ADD_SON_TO_CURRENT_NODE (THE_TREE       : in N_ARY_TREE;
                                     THE_SON_VALUE : in ELEMENT) return N_ARY_TREE;
function SE_ADD_SON_TO_CURRENT_NODE (THE_TREE : in N_ARY_TREE;
                                     THE_SON   : in N_ARY_TREE) return N_ARY_TREE;
function SE_ADD_FIRST_SON_TO_CURRENT_NODE (THE_TREE       : in N_ARY_TREE;
                                     THE_SON_VALUE : in ELEMENT) return N_ARY_TREE;
function SE_ADD_FIRST_SON_TO_CURRENT_NODE (THE_TREE : in N_ARY_TREE;
                                     THE_SON   : in N_ARY_TREE) return N_ARY_TREE;

-- Cree au dessus de la racine de l'arbre un noeud dont l'etiquette sera
-- celle donnee dans le parametre ROOT_VALUE
-- ATTENTION, le noeud courant est ensuite positionne a la nouvelle racine
-- (celle qui cient d'etre cree).
function ADD_ROOT (THE_TREE    : in N_ARY_TREE;
                   ROOT_VALUE : in ELEMENT) return N_ARY_TREE;
-- Meme chose mais avec "Side Effect" (sans duplication des parametres)
function SE_ADD_ROOT (THE_TREE    : in N_ARY_TREE;
                      ROOT_VALUE : in ELEMENT) return N_ARY_TREE;
```

Manipulation d'arbres (suite)

```
-- Cree un nouvel arbre dont la racine a l'etiquette ROOT_VALUE et
-- qui a deux fils : le premier est le sous-arbre designe par TREE_1,
-- le second celui designe par TREE_2
-- ATTENTION, le noeud courant est ensuite positionne a la nouvelle racine
-- (celle qui cient d'etre cree).
function MERGE_TWO_TREES (TREE_1      : in N_ARY_TREE;
                          TREE_2      : in N_ARY_TREE;
                          ROOT_VALUE : in ELEMENT) return N_ARY_TREE;
-- Meme chose mais avec "Side Effect" (sans duplication des parametres)
function SE_MERGE_TWO_TREES (TREE_1      : in N_ARY_TREE;
                             TREE_2      : in N_ARY_TREE;
                             ROOT_VALUE : in ELEMENT) return N_ARY_TREE;

----------------------------------------------------------------------
-- Acces au contenu du noeud d'un arbre N-aire (contenu ou
-- pointeur sur le contenu) de l'element courant dans l'arbre

function CURRENT_CONTENT (TREE : in N_ARY_TREE) return ELEMENT;
function CURRENT_CONTENT_ACCESS (TREE : in N_ARY_TREE) return PT_ELEMENT;
```

Manipulation d'arbres (suite)

```
-- Tester l'existance d'un fils ou du pere pour l'element courant. La
-- valeur FALSE signifie qu'il n'y a pas de fils au numero indique ou pas
-- de pere (le noeud courant est alors le noeud racine))
function CURRENT_TEST_FATHER (TREE : in N_ARY_TREE) return BOOLEAN;
function CURRENT_TEST_SON (TREE    : in N_ARY_TREE;
                           SON_POS : in POSITIVE := 1) return BOOLEAN;
-----------------------------------------------------------------------
-- Avoir des informations diverses sur l'arbre. Est-il vide (on n'a rien),
-- contient-il des fils, si oui combien...
function IS_TREE_EMPTY (TREE : in N_ARY_TREE) return BOOLEAN;
function NUMBER_OF_NODES (TREE : in N_ARY_TREE) return NATURAL;

-----------------------------------------------------------------------
-- L'element courant est-il situe a la racine? est-il une feuille?
-- conbien a-t-il de fils?
-- Ces primitives levent N_ARY_TREE_ERROR si l'arbre est vide
function IS_CURRENT_A_ROOT (TREE : in N_ARY_TREE) return BOOLEAN;
function IS_CURRENT_A_LEAF (TREE : in N_ARY_TREE) return BOOLEAN;
function CURRENT_NUMBER_OF_SON (TREE : in N_ARY_TREE) return NATURAL;
```

Manipulation d'arbres (suite)

```
   -- Changer la position de l'element courant dans l'arbre
   function CURRENT_GOTO_ROOT (TREE : in N_ARY_TREE) return N_ARY_TREE;
   function CURRENT_GOTO_FATHER (TREE : in N_ARY_TREE) return N_ARY_TREE;
   function CURRENT_GOTO_SON (TREE     : in N_ARY_TREE;
                              SON_POS : in POSITIVE := 1) return N_ARY_TREE;
----------------------------------------------------------------------
   -- Extraction du sous-arbre a partir du noeud courant (qui devient
   -- la racine de l'arbre rendu)
   function EXTRACT_SUBTREE (TREE : in N_ARY_TREE) return N_ARY_TREE;
   -- meme chose mais sans recopie du sous-arbre extrait
   function SE_EXTRACT_SUBTREE (TREE : in N_ARY_TREE) return N_ARY_TREE;
----------------------------------------------------------------------
   -- Supression d'un sous-arbre par dont la racine est le noeud courant. Le
   -- noeud courant est également supprime et le noeud courant devient
   -- le noeud pere. Ces primitives rendent N_ARY_TREE_ERROR dans le cas ou
   -- le noeud courant n'est pas correctement positionne.
   function CURRENT_SUPPRESS_SUB_TREE (TREE : in N_ARY_TREE) return
                     N_ARY_TREE;
   function SE_CURRENT_SUPPRESS_SUB_TREE (TREE : in N_ARY_TREE) return
                     N_ARY_TREE;
```

Manipulation d'arbres (suite & fin)

```
------------------------------------------------------------------------
-- Comparaison de deux arbres. Attention aux effets de bord si les
-- types contenus contiennent des pointeurs...
function EQUAL (X, Y : in N_ARY_TREE) return BOOLEAN;

------------------------------------------------------------------------
-- Calcul de la profondeur d'un arbre (0 => arbre vide).
-- ATTENTION : ici, on définit la profondeur d'un harbre comme étant le
-- nombre maximum de nœuds depuis la racine jusqu'à la feuille la plus
-- "lointaine".
function DEPTH (X : in N_ARY_TREE) return NATURAL;

private ...

end GENERIC_N_ARY_TREE_MANAGER;
```

on met des choses ici mais c'est comme si "l'utilisateur" de l'unité ne savait rien

Exemple d'itérateur

```
-- Avant tout, des primitives de parcours generiques. Ces primitives
-- appliquent a chaque noeud une primitive fournie par l'utilisateur en
-- parametre generique formel.
-- Ces primitives disposent de deux informations :
--     - un pointeur vers le contenu du noeud (de type PT_ELEMENT fourni
--       par l'utilisateur). L'utilisation d'un pointeur permet la
--       modification par effet de bord du contenu de l'arbre.
--     - la profondeur du noeud transmis en parametre (profondeur
--       0 = racine).
--     - L'ordre du noeud par rapport au noeud pere (0 = racine car
--       elle n'a pas de pere).
--     - le nombre total de fils par rapport au noeud pere (0 =
--       racine car il n'y a pas de noeud pere).
--     - savoir si le noeud que l'on visite est une feuille ou non
generic
   with procedure FUNCTION_TO_APPLY (PT_CONTENT           : in PT_ELEMENT;
              DEPTH               : in NATURAL; SON_NUMBER: in NATURAL;
              TOTAL_SON_IN_FATHER : in NATURAL; IS_A_LEAF : in BOOLEAN);
procedure PREFIX_RUN (TREE : in N_ARY_TREE);
```

Éléments pour la réutilisation

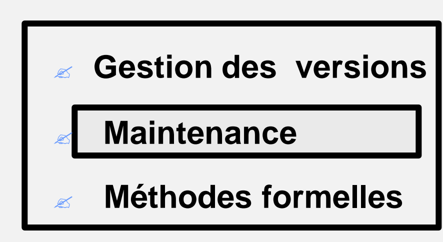
Exemple de composant

- Rendre un composant utilisable est une activité complexe
- Langages objets: l'utilisation de l'héritage est évident
- Autre approche, les générateurs de programmes `flex/yacc`
  etc...
- Un bon composant réutilisable est un facteur important de gain de temps

Plan du cours

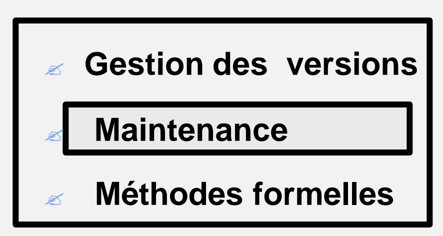
- VIII. Méthodes de conception de logiciels
- IX. Fiabilité du logiciel
- X. Test du logiciel
- XI. Gestion des versions
- XII. Réutilisation de logiciels
- ? **XIII. Maintenance de logiciels**
- **XIV.** Introduction aux méthodes formelles

Éléments pour la maintenance



- Analyse d’un cas d’étude réel
- Discussion sur le choix du langage
- OS et Middleware
- Présentations de “bonnes” architectures logicielles

Éléments pour la maintenance



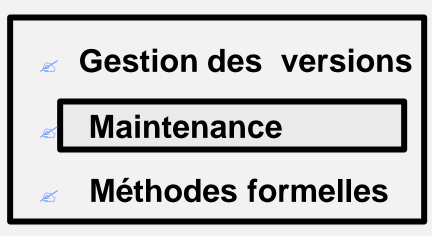
- La maintenance se prépare dès la phase de conception ET pendant le développement
  - Choix de l'architecture
  - Réutilisation de composants et maîtrise de ces composants
  - Choix du langage de programmation
  - Mise en place de procédures de tests
  - Documentation
  - etc...

Procédures de collecte

Bonne architecture logicielle

Middleware (interface OS)

Éléments pour la maintenance



- En général, on utilise des formulaires
  - Identifier l'auteur
  - Identifier les informations primordiales sur l'anomalie
    - sa reproductibilité
    - la séquence d'actions qui y aboutit
    - toutes autres informations utiles
  - C'est un guide pour les utilisateurs lorsqu'ils doivent identifier un problème
- Deux exemples
  - FrameKit (rapport d'anomalie sur WWW)
    - aspect "externe"
  - Proteus (rapport sur feuille)
    - aspects "externes" et "internes"

Éléments pour la maintenance

- Gestion des versions
- **Maintenance**
- Méthodes formelles

Netscape: FrameKit/Macao: Bug report form

Miscellaneous information

This bug report concerns : Macao / FrameKit / A tool — AMI-Net, AMI-Net Verifier

This bug report comes from — Name / E-mail / Organization

Problem Report | Extension proposal

Importance (if it is a problem) — Blocking / Major / Minor

Reproductibility (if it is a problem) — Systematic / Some times

Full Description (french or english)

Submit Bug Report — Reset

Identifier le composant incriminé

Identifier l'origine (réponse)

Suggérer un degré d'urgence

Anomalie ou extension?

Définir la reproductibilité

Information brute

Soumission

Éléments pour la maintenance

Circuit d'une fiche d'anomalie

- (1) Identification de la fiche
- (2) Identification de l'erreur associée à l'anomalie
- (3) Correction de l'erreur
- (4) Tests de non régression
- (5) Diffusion d'une nouvelle version

Éléments pour la maintenance
Une fiche d'anomalie

Change Request Form

Project: Proteus/PCL-Tools **Number:** 23/94
Change requester: I. Sommerville **Date:** 1/12/94
Requested change: When a component is selected from the structure, display the name of the file where it is stored.

Change analyser: G. Dean **Analysis date:** 10/12/94
Components affected: Display-Icon.Select, Display-Icon.Display

Associated components: FileTable

Change assessment: Relatively simple to implement as a file name table is available. Requires the design and implementation of a display field. No changes to associated components are required.

Change priority: Low
Change implementation:
Estimated effort: 0.5 days
Date to CCB: 15/12/94 **CCB decision date:** 1/2/95
CCB decision: Accept change. Change to be implemented in Release 2.1.
Change implementor: **Date of change:**
Date submitted to QA: **QA decision:**
Date submitted to CM:
Comments

même type d'info qu'avant

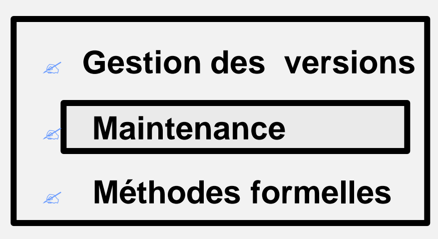
Identification de l'erreur

Fiche de tâche associée à la correction

Acquittement de l'erreur

D'après I.Sommerville ©1995

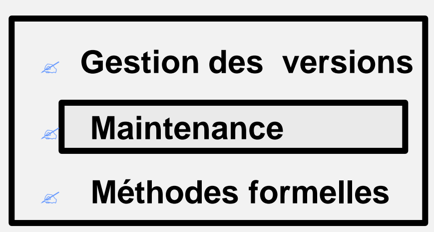
Éléments pour la maintenance



- Un langage structuré et fortement typé apporte beaucoup
- Ce n'est qu'un critère parmi d'autres
  - Conception & architecture
  - Gestion de la configuration
  - Méthodes de test
  - Support logiciel pour le développement
  - Savoir-faire des programmeurs
  - Savoir-faire du “management”
- mais c'est un critère

Éléments pour la maintenance

Etude et analyse d'un long projet
(Effectuée par Stephen. Zeigler, Rational Soft Corp)



- Étude sur une ligne de produits (de 1983 à 1994, date de l'étude)
  - VADS (Verdix Ada Development System)
    - Environnement Ada multi-cibles
    - essentiellement sous Unix
    - Développements en C et en Ada
  - Un gros volume de sources communs à toutes les cibles
  - Début des développements en C (avant 1986)
  - Développements suivants en Ada (à partir de 1986)
  - Caractéristiques du projet (de "bonnes conditions")
    - faible "turnover"
    - niveau de spécialisation élevé dans les équipes
    - produits stables, pas de va et vient coté management

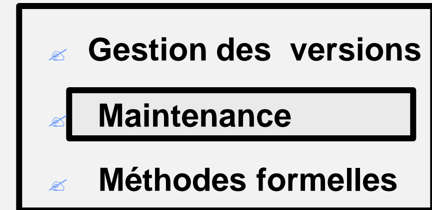
Analyse des données du projet (entre autres, comparaisons entre C et Ada)

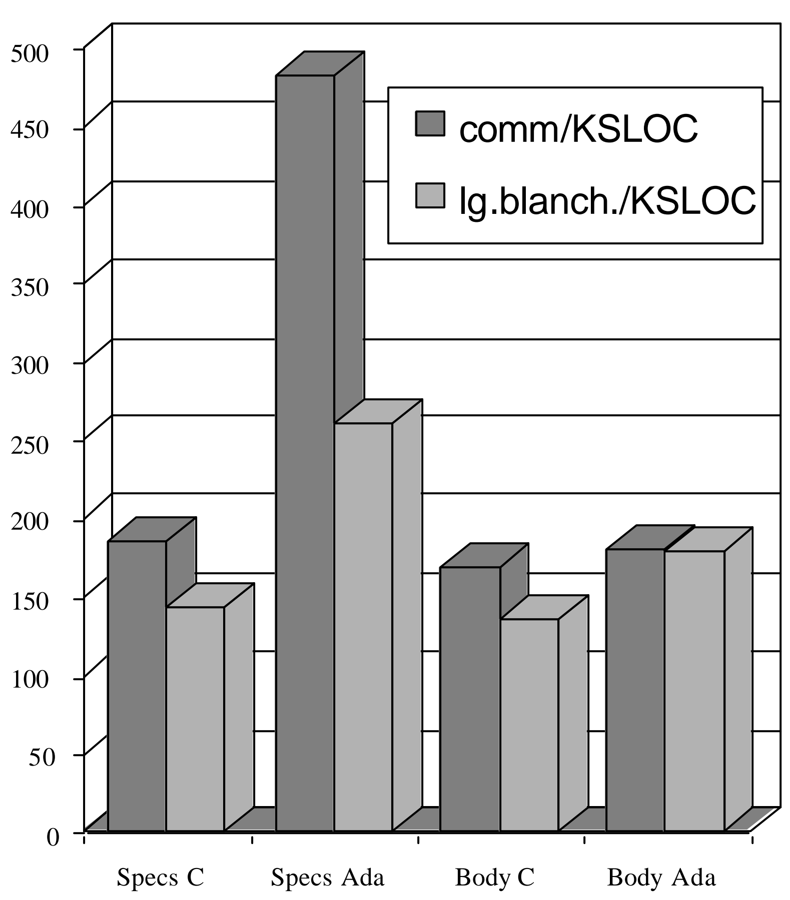
Éléments pour la maintenance
Etude : mesure de la fiabilité

	C	Ada (83)	Scripts	Autres	Total
toutes lignes	1 925 523	1 883 751	117 964	604 078	4 531 316
lignes d'instructions (sans com.)	1 508 695	1 272 771	117 964	604 078	3 505 508
Fichiers	6 057	9 385	2 815	3 653	21 910
M = Mises à jours	47 775	34 516	12 963	12 189	107 443
E = Extensions	26 483	23 031	5 594	6 145	61 253
Correction de bugs	13 890	5 841	4 603	1 058	25 392
Modification par extension (E/M)	0.52	0.25	0.82	0.17	0.41
Corrections par K lignes	9.21	4.59	39.02	1.75	7.25
Coût de développement ($)	15 873 508	8 446 812	1 814 610	2 254 982	28 389 856
Coût de la ligne de programme ($)	10.52	6.82	15.38	3.72	8.10
Bugs identifiés "à l'extérieur"	1020	122	/	/	1242
Bugs identifiés à l'extérieur/K lignes	0.676	0.096	/	/	0.355

Ligne Ada: moins cher, -70% de bug internes, -90% de bugs externes

Éléments pour la maintenance
Etude : mesure de la fiabilité



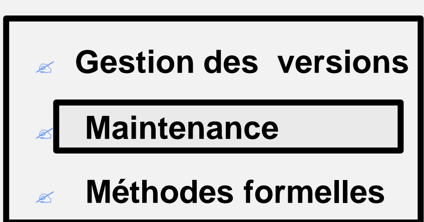


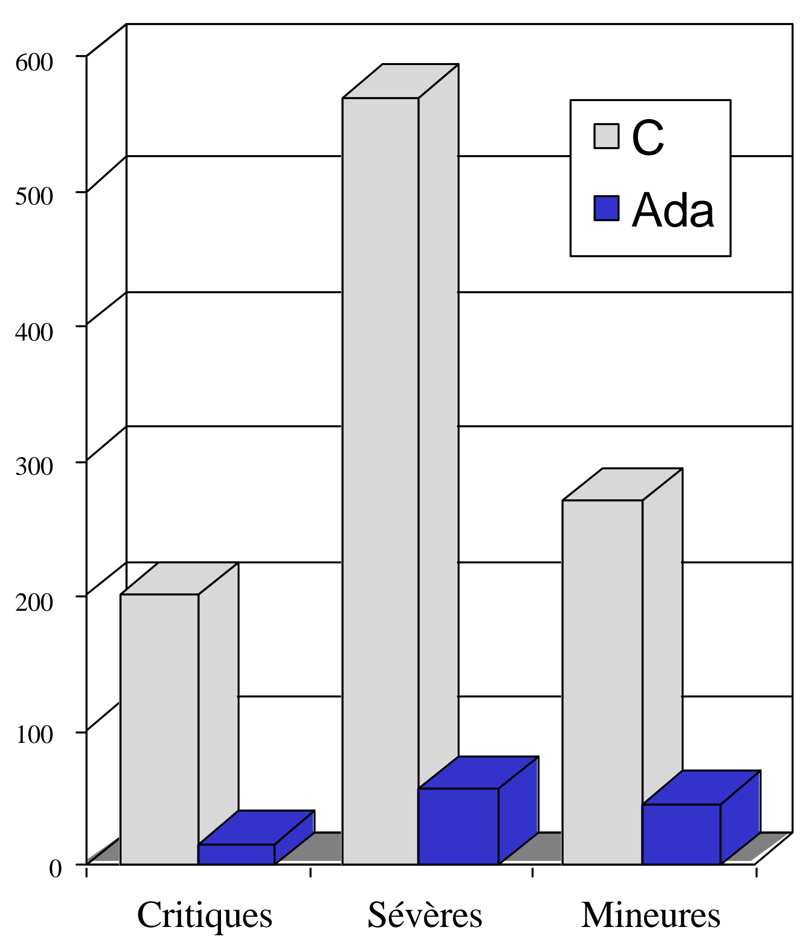
- Equivalences lignes C/lignes Ada?
- Nombre de lignes de commentaires par lignes de code
  - Les spécifications en Ada tendent à servir de base documentaire
  - le programmeur C va directement voir les sources (body C)

	CS	AS	CB	AB
Modification par extension	**0.18**	**0.16**	**0.60**	**0.33**
Bugs identifiés à l'extérieur/K lignes	**5.60**	**0.25**	**9.63**	**4.81**

Toujours un avantage à Ada

Éléments pour la maintenance
Etude : répartition des bugs entre C et Ada

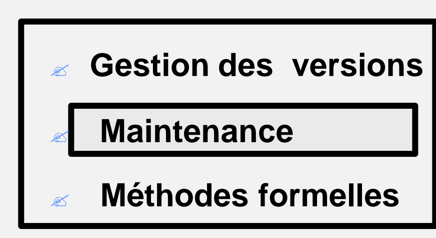




- L'utilisation de standards pour éviter (en C) a permis de réduire le taux d'erreurs "bêtes" comme:
  - problème avec les "parties else"
  - "=" au lieu de "= ="
  - "/=" au lieu de "!="
  - utilisation abusive de macros
  - utilisation de "ruses en C" illisibles
  - utilisation d'entiers pour des pointeurs (et vice-versa)

Éléments pour la maintenance
Répartition des coûts de la maintenance

Éléments pour la maintenance
Autres remarques



- L'étude porte principalement sur les coûts de suivi des produits concernés
- Selon l'auteur: 1$ dépensé en évolutions signifie 3$ dépensés pour
  - Support client
  - Documentation
  - Administration
  - Marketting
  - Packaging
  - Calmer/aider/"dédommager" les usagers malencontreusement affectés par des bugs du produit

Éléments pour la maintenance
Définir une bonne architecture

- Utiliser des middleware
- Penser l'interfaçage/encapsulation
  - Approche "bibliothèque"
  - Approche "communication"
    - par messages
    - par mémoire partagée
  - Autres mécanismes sophistiqués
    - EdL dynamique (systèmes d'exploitation)

Une solution :Application Programming Interface (API)

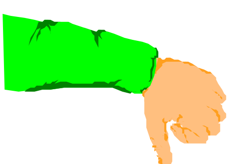
Éléments pour la maintenance
Interfaçage : approche bibliothèque

Un seul exécutable

"transportabilité"

Transparence des évolutions (EdL)

Produits utilisant une bibliothèque



Mises à jour
plus "délicates"

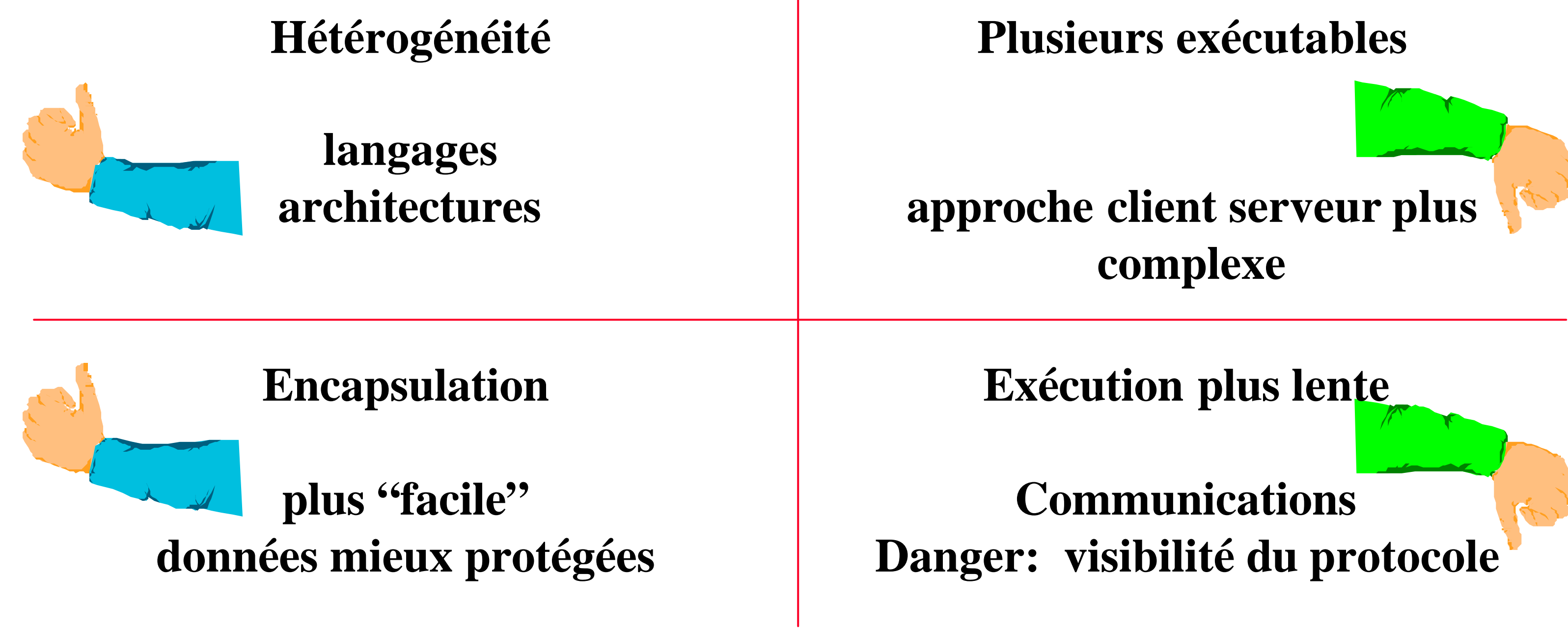
Encapsulation

plus délicate
(visibilité des structures)

Hétérogénéïté

langage, architecture

Éléments pour la maintenance
Interfaçage : approche message

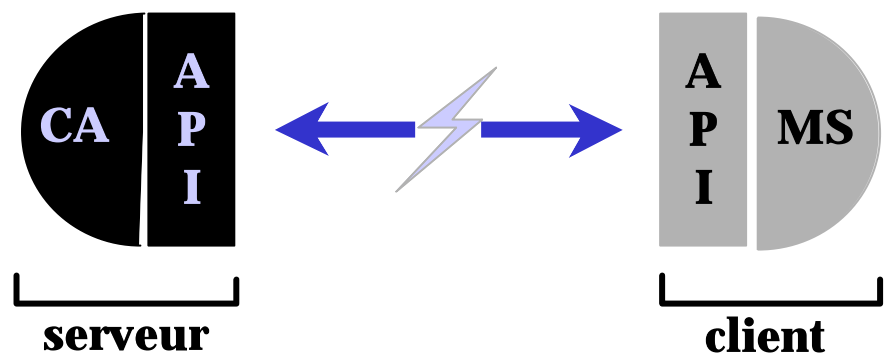


Attention au langage de communication

Attention au protocole

Éléments pour la maintenance
Exemples d'architectures

Architecture chez Sligos
(terminaux de saisie + serveur)



phase1: identification du client (version)

phase 2: envoi de masques dans un format de haut niveau (gérés par le moteur de saisie)

PVM
(Parallel Virtual Machine)

Couche de communication

Multiples implémentations

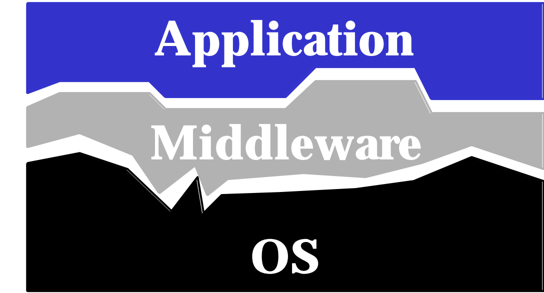
suivi d'exécution, debug

Éléments pour la maintenance
Encapsulation de l'OS

Famille Unix
BSD, SystemV

Famille Windows
NT, 95

MacOS
v7.5.x, copland



- Points de relation avec l'OS
  - PrésentationX-MOTIF, Toolbox Mac toolbox Windows...
  - SGF Nommage "Unix", nommage "IBM", nommage "Mac"...
  - Communication Socket (TCP, UDP), fifo, signaux, ligne série...

⚠ Tout se paye... ici, en temps d'exécution

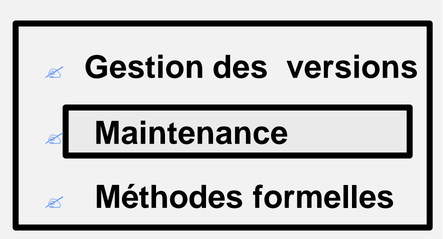
Cas de l'OS Unix

- Ouvert
  - Sources diffusés publiquement
  - Extensibilité
- Complet
  - Gestion des tâches
  - Gestion de fichiers
  - Communication/Réseau
  - Notion de shell (extrêmement sophistiquée)

...Mais...

Unix est toujours «installable» !

- Hétérogénéité des versions
  - Philosophie BSD versus philosophie System V
  - POSIX...
  - Pas d'uniformisation, pas de règles de conduite dans le développement
    - Interfaces graphiques disparates (X-MOTIF, OpenWin, HP...)
    - look and feel
    - paramètres des outils
- Un "système par et pour les ingénieurs systèmes"???
  - Administration complexe (mais évolution dans le bon sens)
  - difficulté de maintenance
    - beaucoup de bugs,
    - trous de sécurité...

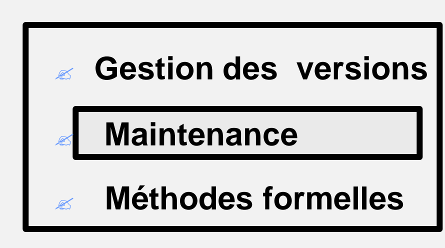


Unix : observations

- Les problèmes d'Unix sont explicables historiquement
  - Pas d'approche GL, un système en extension continuelle...
- La portabilité est somme toute "relative"
  - ...et pourtant, c'est un précurseur dans le domaine
- Beaucoup d'éléments "non standard"
  - Rajouts à faire
  - Choix à effectuer (très gros marché du freeware cependant)

Raison d'une certaine "désaffection" pour Unix...

Éléments pour la maintenance
Nouveaux OS : approche



- A la base, un système fermé (boîte noire)
- Exemple: MacOS
  - Des primitives définies et documentées
    - défaut: trop bas niveau (graphique)
  - Des règles de développement
    - des volumes de milliers de pages
- Notion de “signature” systématique
  - classification gérée par l’OS

Éléments pour la maintenance
Exemple 1 : MacOS Copier/Coller

Application source

Couper Copier

exportation de plusieurs formats (prédéfinis ou “maison”)

presse papier

Coller

Choix de l’un des formats disponibles

Application cible

Éléments pour la maintenance
Exemple 2 : MacOS Impression

imprimante sélectionnée

Interface de construction

laser PostScript

laser PCL

laser bitmap

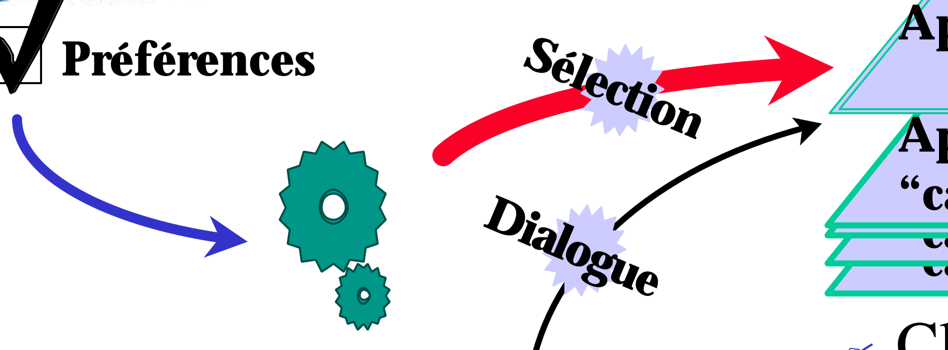
Jet d'encre

Drivers

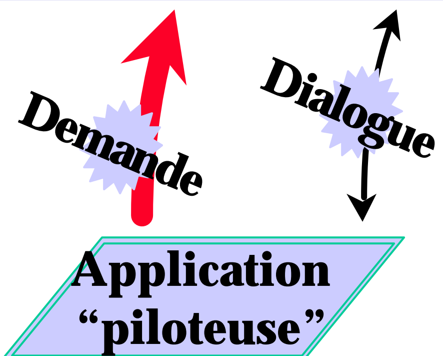
Application source

- Mise à jour aisée
- Service uniforme
- Applicatif plus simple

Éléments pour la maintenance
Exemple 3 : Mac Event

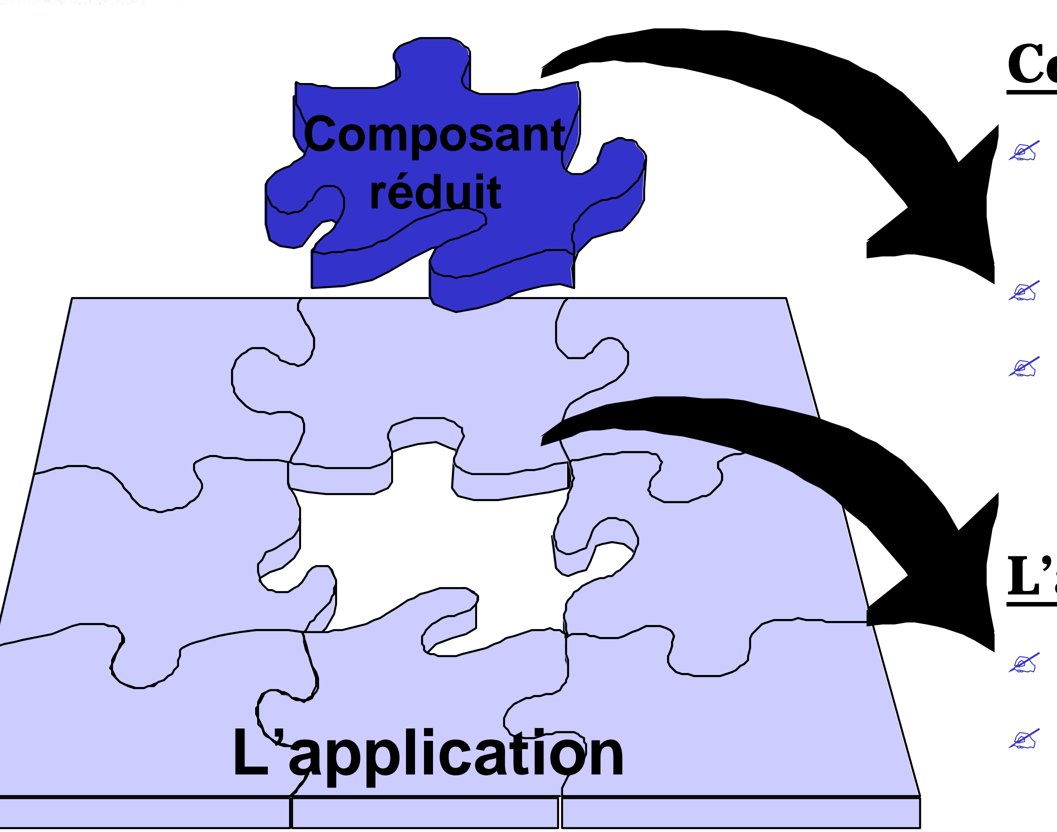


Interface de dialogue



- Classification des applications
  - 1) j'accepte/ je n'accepte pas
  - 2) minimum requis
  - 3) services dédiés
    - éditeur de texte
    - tableur
    - ...
  - 4) place pour des services spécifiques

Éléments pour la maintenance
Exemple 4 : Open Doc



Composant Open-Doc

- Gère des "morceaux de documents"
- Sous-ensemble des fonctions
- "emmené" avec un copier/coller

L'application

- Ensemble complet de services
- Peut faire appel à des composants open-doc

Client/serveur

Client

Serveur (httpd)

Types de services différents

Conventions : le protocole

Comme les architectures cibles (site clients)

URL

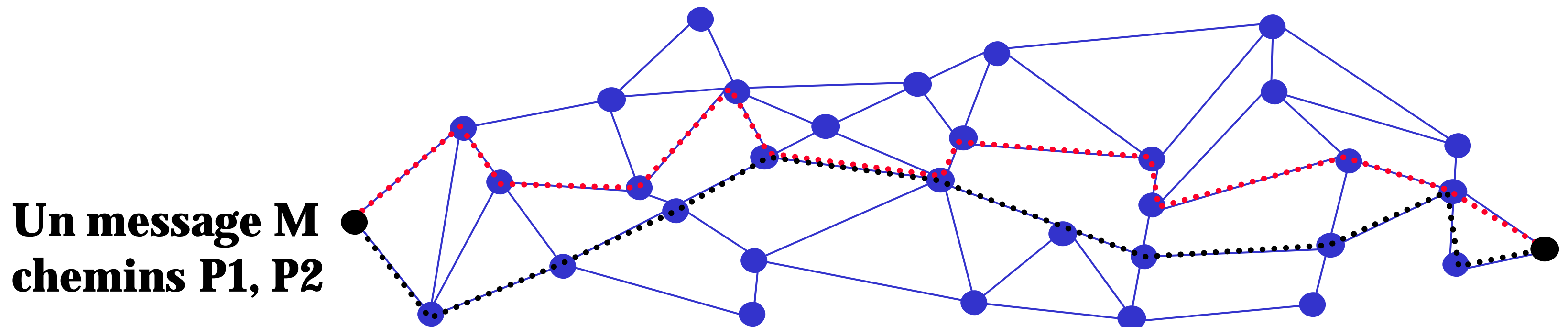
Différents protocoles

- *E-mail*
- telnet
- ftp
- news
- gopher
- wais
- http
- ...

Éléments pour la maintenance
Architecture du WWW

- Mécanisme transparent à l'utilisation
- Réseau maillé dont la gestion est décentralisée (WWW)

Un message M chemins P1, P2



- Contrôle complètement décentralisé

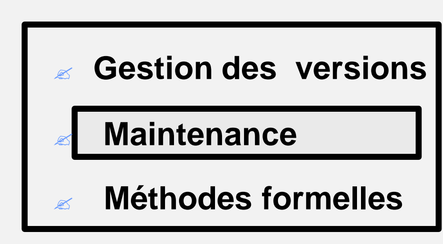
- Comprennent plusieurs protocoles
- URL : Uniform Resource Locator

```
protocole : //serveur[:port]/[chemin-d'accès]fichier[indicateurs]
```

- Les clients

u	u	u
ftp news gopher http telnet mailto ...	identification du serveur à joindre	données recherchées sur le serveur

Éléments pour la maintenance
Architecture du WWW
Serveur HTTP



- Hyper Text Transfert Protocol
- Basé sur un langage HTML (Hyper Text Markup Language)
  - “Simplification” de SGML et de XML
  - Une DTD particulière
- inventeur de HTML : le CERN
- Lien hyper texte
  - “ancre” référençant un autre texte ou portion de texte

Lien hyper texte = URL

Éléments pour la maintenance
Architecture du WWW
Le langage HTML

- Structuration
- un document est un ensemble de pages
- Langage ASCII de balisage de texte
  - balise = constructions types
- Construction d'une balise

```
<type_de_balise [attributs]>
```

  - Les balises se ferment presque toujours

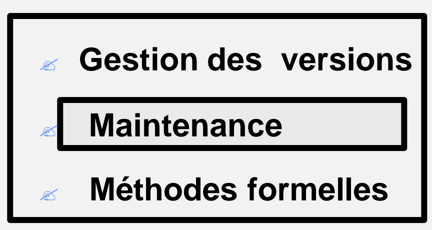
Plusieurs versions

HTML-1

HTML-2

HTML-3

etc...



Client/serveur

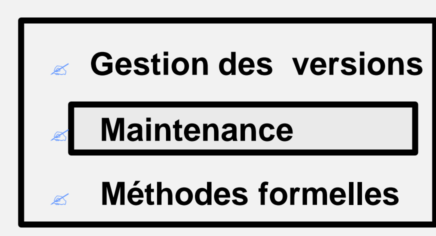
client Web
“transmet” les rubriques d’un formulaire

Serveur (httpd)
invoque le programme CGI associé au formulaire avec les données reçue

Ce programme produit des “données WEB”
ASCII
GIF/JPEG...

Affiche le résultat

Éléments pour la maintenance
Architecture du WWW
Le langage Java



- Java est un langage objet
- Idée majeure: il repose sur une machine virtuelle
- Cette machine virtuelle est implémentée par tous les OS
- Avantages
  - Portabilité au niveau du “byte-code”
  - Possibilités de migration du code en cours de son exécution
- Inconvénients
  - Une certaine lenteur
  - Problèmes potentiels de droits d’accès
  - Pas utilisable pour des systèmes critiques (spécification SunMicroSystems)

Est-ce un langage pour développer de grosses applications ?

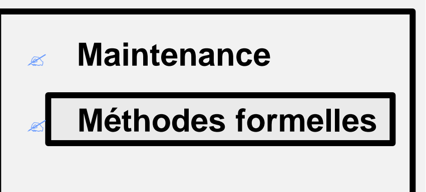
Plan du cours

- VIII. Méthodes de conception de logiciels
- IX. Fiabilité du logiciel
- X. Test du logiciel
- XI. Gestion des versions
- XII. Réutilisation de logiciels
- XIII. Maintenance de logiciels
- ? **XIV. Introduction aux méthodes formelles**

Techniques formelles

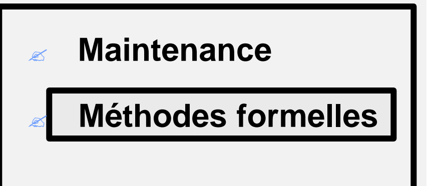
- Méthodes semi-formelles :
  - libre interprétation laissée au lecteur
  - fournissent un schéma général informel
- Exemples :
  - DFD : traitements non définis
  - UML : méthodes définies informellement
- Méthodes formelles :
  - Fondements mathématiques
    - expression des objets du développement
    - expression des propriétés
    - expression des preuves

Techniques formelles



- Preuve sur une spécification = *validation*
- Preuve sur une propriété = *vérification*
- Vérification formelle : preuve de propriété utilisant un système formel.
- Il faut pour cela :
  - Un langage (et sa sémantique) d'expression du système
  - Un langage (et sa sémantique) d'expression de propriétés
  - Un système formel permettant le codage de preuve
    - Preuve automatique
    - Preuve semi automatique

Techniques formelles



- Système formel permet de représenter des preuves formelles
  - langage de formules
  - sous-ensemble de ces formules = axiomes, vrais par définition
  - règles de déduction = règles d'inférence : formalisent les étapes élémentaires licites de raisonnement

Exemple de système formel

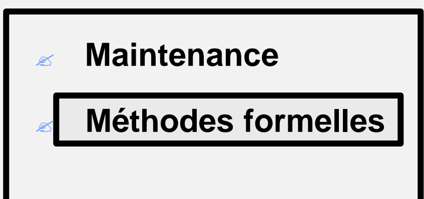
- calcul des propositions : A ? ?? ??   ??
- exemple d'axiome : f ?   (g ?   f)
- règle d'inférence unique : modus ponens

f, f ?   g   (MP)   (prémisses)
g   (conclusion)

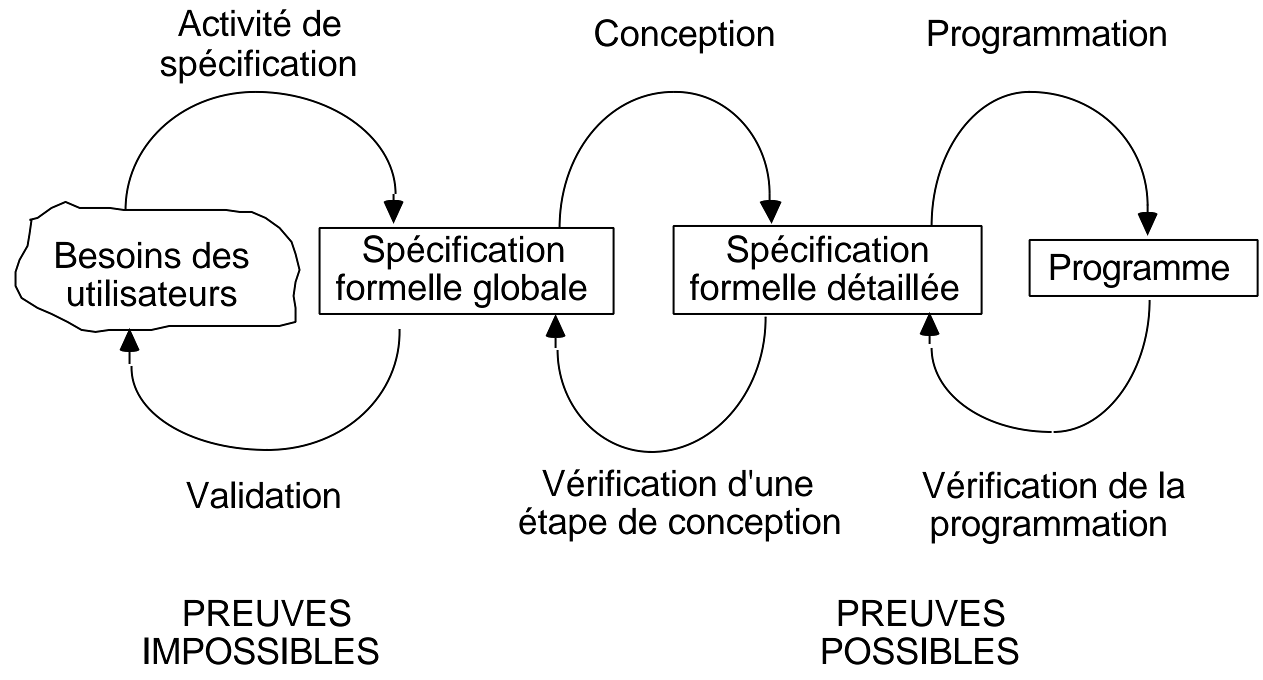
Techniques formelles

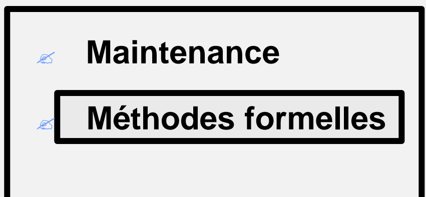
- Preuve d'une formule
  - dérivation
  - séquence de formules
  - dernière formule = formule à prouver
  - première formule = axiome ou hypothèse
- Théorème
  - formule provenant d'une dérivation
  - Prouver formellement des propriétés = prouver des théorèmes exprimant les propriétés
- Preuves formelles : éventuellement vérifiables automatiquement
- Preuves "non formelles" : tests statiques, inspections poussées
- Remarque : existence de preuves quasi-formelles

Techniques formelles



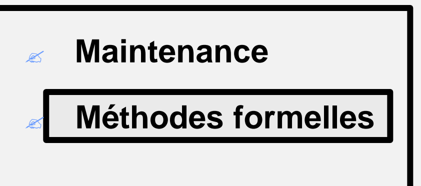
- Preuves formelles ? spécifications formelles
- Pas de preuve formelle possible à partir des besoins des utilisateurs informellement exprimés





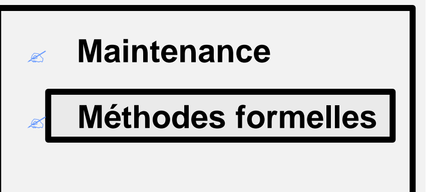
- Preuve de l'étape de conception
  - correction de la spécification détaillée par rapport à la spécification globale
- Preuve de l'étape de programmation
  - correction du programme par rapport à la spécification détaillée (logique de Hoare)
  - Possibilité de preuves "internes"
  - Complétude des spécifications
  - Absence d'interblocage
- Preuve de spécification globale par preuve de propriétés

Techniques formelles



- Spécification formelle =
  écriture avec une *syntaxe* bien définie, comme celle d'un langage de programmation : BNF
  - *sémantique* rigoureuse, définissant les modèles mathématiques de la spécification
  - règles de déduction pour démontrer des propriétés de la spécification
- **Remarque**
  - Comme pour les langages de programmation, pas de langage de spécification universel
  - Dépendance par rapport à l'existant, au domaine d'application, aux propriétés à établir, etc…

Techniques formelles

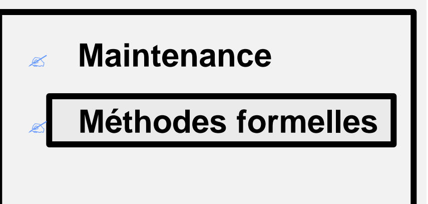


- Logique classique = cadre général
- A inclure :
  - types,
  - fonctions partielles
  - changements d 'états
  - raffinements
- Nécessité de langages de spécification adaptés

Techniques formelles

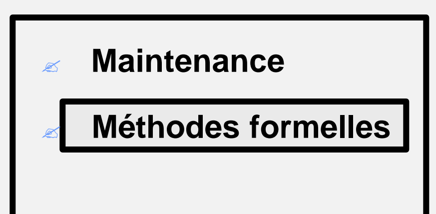
Formalisme	Sémantique	Logique
Spécifications algébriques	Algèbres hétérogènes	Logique équationnelle typée
LOTOS	Arbres de transitions	Logiques temporelles
VDM	Domaines (fonctions partielles)	LPF (logique des fonctions partielles)
Z	Théorie des ensembles	Logique des prédicats

Techniques formelles



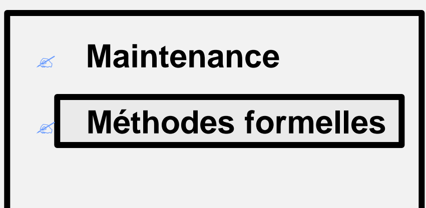
- **Approches avec environnements :**
  - réseaux de pétri : parallélisme
  - spécifications algébriques : types abstraits
  - LOTOS : types abstraits + parallélisme
  - Z, VM, B : manipulation d'états

Techniques formelles Intérêts des approches formelles



- Compréhension approfondie du logiciel : levée des ambiguïtés
  - Preuves, évaluation symbolique : validation formelle de la spécification
  - Prototypage
  - Raffinements (ou réification, implantation abstraite)
- Systèmes critiques : preuves de raffinements
- Génération automatique de jeux de tests

Techniques formelles Exemple : techniques algébriques



- Types abstraits algébriques
- Type de données
  - = ensemble de valeurs muni d'opérations permettant de calculer avec ces valeurs
  - **Exemples** : booléens, entiers, etc.
- Type abstrait de données=classe de types de données
  - *nom* des ensembles de valeurs
  - *en-tête* des opérations
  - pas de représentation des valeurs
  - propriétés des opérations : *axiomes*
- Remarque
  - types abstraits = base des *classes* et *objets*

Techniques formelles Exemple : techniques algébriques

- Noms d'ensembles de valeurs = *sortes*
  - Opérations typées : *profil* = liste des sortes des arguments
  - Sortes et noms d'opérations = *signature*

Exemple

sortes : Ens, Nat, Bool ;

opérations :

? : ? ?Ens
ajout _ _ : Nat ? Ens ? Ens
_ ? _ : Nat ? Ens ? Bool
0 : ? ?Nat
_ +1 : Nat ? Nat
eg _ _ : Nat ? Nat ? Bool
vrai, faux : ? Bool

Techniques formelles

- Maintenance
- **Méthodes formelles**

- Spécifications classiques
  - une seule sorte
  - réutilisation des spécifications
- axiomes=équations, ou axiomes conditionnels
  - compatibilité des sortes
  - contraintes hiérarchiques / *inconsistance*

Exemple :

spec ENS_NAT
utilise NAT, BOOLEENS
sorte Ens
opérations :
? : ? ?Ens
ajout _ _ : Nat ? Ens ? Ens
_ ? _ : Nat ? Ens ? Bool
axiomes :
(N ? ?) = faux
eg(N, N') = vrai ? (N ? ajout(N', E)) = vrai
eg(N,N') = faux ? (N ? ajout(N', E)) = (N ? E)
(N ? E) = vrai ? ajout(N, E) = E
ajout(N, ajout(N', E)) = ajout(N', ajout(N, E))
avec :
N, N' : Nat ; E : Ens ;
fin ENS_NAT.

Techniques formelles
Types algébriques génériques

- Paramétrage d'une spécification par d'autres spécifications

```
generic spec ENS(ELEM)
        utilise BOOLEENS
        sorte Ens
        opérations :
                ?  : ?  ?Ens
                ajout _ _ : Elem ? Ens ?  Ens
                _ ?  _ :  Elem ? Ens ?  Bool
        axiomes :
                (L ?  ? ) = faux
                eg(L, L') = vrai ?  (L ?  ajout(L', E)) = vrai
                eg(L,L') = faux ?  (L ?  ajout(L', E)) = (L ?  E)
                (L ?  E) = vrai ?  ajout(L, E) = E
                ajout(L, ajout(L', E)) = ajout(L', ajout(L, E))
        avec : L, L' : Elem ; E : Ens
fin ENS.
```

Spécification de ELEM :

- - sorte Elem
- - propriétés de eg

```
par ELEM
  utilise BOOLEENS
  Sorte Elem
  opérations :
        eg _ _ : Elem ? elem ?  Bool
  axiomes :
        eg(L, L) = vrai
        eg(E, E') = eg(E', E)
        eg(E, E')=vrai ? eg(E', E")=vrai
                ?  eg(E, E")=vrai
        avec E, E', E" : Elem
fin ELEM.
```

Techniques formelles

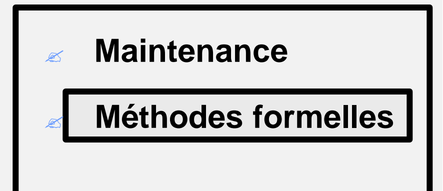
- Utilisation d'une spécification générique
- Exemple : ensemble de propositions.

spec ENS_PROP =
ENS(ELEM => PROPOSITION avec eg => egal)

spec ENS_PROP
utilise PROPOSITION, BOOLEENS

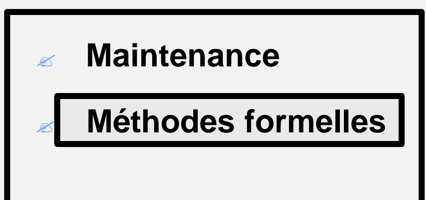
spec PROPOSITION
utilise PROJET, SOC_SERVICE, NAT, ...
sorte Proposition
opérations :
ref _ : Nat ? Proposition
egal _ _ : Proposition ? Proposition ? Bool
origine _ : Proposition ? Société
proj_concerné : Proposition ? Projet
montant : Proposition ? Nat
durée _ : Proposition ? Nat
...
axiomes :
egal(ref(N), ref(N')) = eg(N, N')
% il faut **PROUVER** que egal correspond à eg
avec N, N' : Nat
fin PROPOSITION

Techniques formelles



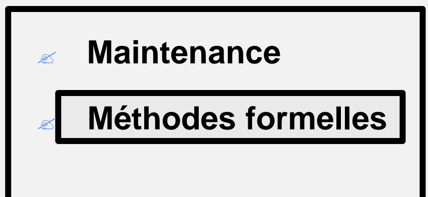
- Sémantique d'une spécification = classe de types de données
- Associations :
  - sorte / ensemble de valeurs
  - opérateur / opération (algorithme)
  - correspondance des profils
  - satisfaction des axiomes

Techniques formelles



- Exemples
  - spécification ENS_NAT, Ens / séquences, arbres, tableaux
- Logique associée aux spécifications
    - = logique du 1er ordre typée avec égalité
  - Axiomes conditionnels positifs
  - exécutabilité des spécifications par réécriture (orientation des axiomes de gauche ˆ droite), ou résolution équationnelle
- Résolution équationnelle
  - preuve de théorèmes déductifs

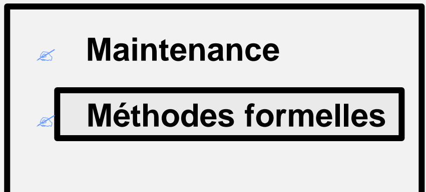
Techniques formelles



- Induction
  - Récurrence sur les entiers naturels :
    - entiers définis par "0" et "successeur »
  - Spécification
    - Fondé sur la distinction des opérations qui engendrent toutes les valeurs possibles : *générateurs*

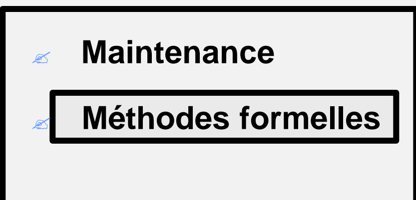
- Exemple
  - ensembles : ? et ajout.
  - Expression des autres opérations en fonction des générateurs
- Exemple :
  - union ensembliste
    - E ? ? = E
    - E ? ajout(N, E')) = ajout(N, E ? E')
  - Contrainte : *?* et *ajout* doivent engendrer toutes les valeurs possibles.

Techniques formelles



- Preuve par induction structurelle
- Exemple : démonstration d'une propriété inductive P(E) sur les ensembles :
  - démonstration de P(?)
  - démonstration de P(E) ? P(ajout(N, e))

Techniques formelles
Raffinements



- Passage à une spécification plus détaillée :
  - représentation d'un type par un autre
  - Correspondance entre types : axiomes

Exemple : ensembles d'entiers / listes

spec LISTE_NAT

utilise NAT, BOOLEENS

sorte Liste

générateurs :

lvide : %*représente ?*

cons _ _ : Nat ? Liste ? Liste %*reprŽsente ajout*

opérations :

car _ : Liste ? Nat %
premier élément

cdr _ : Liste ? Liste % reste de la liste

présent _ _ : Nat ? Liste ? Bool

concat _ _ : Liste ? Liste ? Liste

axiomes :

car(cons(N, L) = N

cdr(cons(N, L) = L

présent(N, lvide) = faux

eg(N, N') = vrai ? présent(N, cons(N', L)) = vrai

eg(N, N') = faux

? présent(N, cons(N', L)) = présent(N, L)

concat(lvide, L) = L

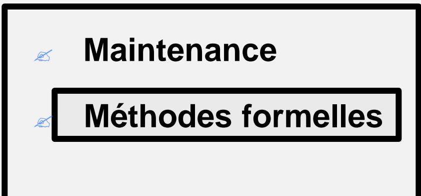
concat(cons(N, L), L') = cons(N, concat(L, L'))

avec

N, N' : Nat

L, L' : Liste

fin LISTE_NAT



Fonction d'abstraction *A*, des listes dans les ensembles

opération :

A _ : Liste ? Ens

axiomes :

? = *A*(lvide)

ajout(N, *A*(L)) = *A*(cons(N, L))

N ? *A*(L) = prŽsent(N, L)

A(L) ? *A*(L') = *A*(concat(L, L'))

avec

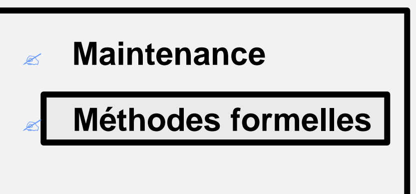
N : Nat

L, L' : Liste

Techniques formelles
Raffinments

- Représentants multiples :
  - {1} est représenté par <1>, <1, 1>, ...
- Représentation surjective :
  - " E, $ L / *A*(L) = E
- Démonstration de la correction de *A* :
  - propriétés de la spécification d'origine conservées

- Exemple :
  - pour l'axiome :
    - E ? ajout(N, E')) = ajout(N, E ? E')
    - il faut prouver que :
    - *A*(L) ? ajout(N, *A*(L'))) = ajout(N, *A*(L) ? *A*(L')))
- Preuve par réécriture
- Représentation de l'égalité :
  - N ? *A*(L) = vrai ? *A*(cons(N, L)) = *A*(L)
  - *A*(cons(N, cons(N', L))) = *A*(cons(N', cons(N, L)))



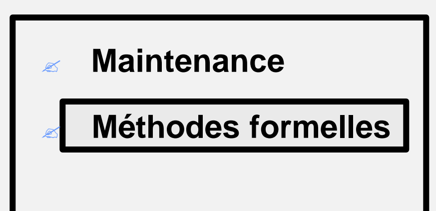
- Chaque construction S est définie par :
  - S : décrite dans le langage
  - P : Propriété satisfaite avavt exécution de S
    - Précondition
  - Q :
  - Propriété satisfaite après exécution de S
    - Postcondition
  - Triplet de Hoare {P}S{Q}

Techniques formelles
Règles

Techniques formelles exemples

Techniques formelles
Un développement. Obligations de preuve

Techniques formelles
Plus faible précondition de Dijkstra



- Principe : on cherche la précondition la plus faible qui établit une postcondition donnée Q après exécution de S
  - Notation [S]Q équivaut à {P}S{Q} où P est la plus faible précondition

Techniques formelles
Règles

Techniques formelles exemples

Techniques formelles Un développement. Obligations de preuve

Techniques formelles