4. EXPRESSION ET ANALYSE DU BESOIN

4.1. INTRODUCTION

Les phases d'expression et d'analyse du besoin permettent de décrire les fonctionnalités du logiciel et les contraintes sous lesquelles celui-ci doit être réalisé.

Les besoins sont exprimés par l'utilisateur dans le **cahier des charges rédigé** en langage naturel.

La réponse est formalisée dans le **dossier d'analyse (ou de spécifications fonctionnelles) .**

Dans le cycle de vie en V, c'est au cours de la phase d'analyse que sont décrits les **tests de validation et qualification,** c'est à dire les tests qui permettront de montrer que le logiciel construit répond aux **besoins** de l'utilisateur exprimés dans le cahier des charges. C'est également au cours de cette phase que doit être rédigée la première version du **manuel utilisateur** en tenant compte des interfaces homme-machine définies dans le dossier de spécifications fonctionnelles et des besoins et contraintes exprimées par l'utilisateur.

Le **plan qualité logiciel** est définitivement établi à l'issue de la phase d'analyse et sera mis en œuvre dans les suivantes. La conformité au plan qualité sera un élément contractuel du projet.

Le **plan projet** ou **plan de développement** est mis à jour si besoin est, en accord entre le client et le fournisseur.

4.2. LE CAHIER DES CHARGES

La rédaction du cahier des charges est la **première étape** de l'expression du besoin.

Le cahier des charges a pour but d'établir une description globale des **fonctions d'un nouveau produit** ou des **extensions d'un produit existant,** à partir de la spécification d'objectifs.

Le cahier des charges doit être **validé** pour s'assurer qu'il répond bien aux souhaits de l'utilisateur.

La description comprend

- l'**énoncé** du problème à résoudre
- la liste des **fonctions de base** requises
- les **caractéristiques techniques** du produit (limites, performances, nombre d'utilisateurs, ressources, interfaces avec d'autres produits, fiabilité, disponibilité, compatibilité logiciel/ matériel)
- les **priorités** éventuelles de réalisation

Le cahier des charges précise les **facteurs de qualité** (exemple : rapide) qui devront se traduire dans le produit final par des **critères de qualité** (exemple : temps de réponse inférieur à 2s) mesurables avec les métriques appropriées.

Les difficultés sont de plusieurs ordres :

- difficulté à exprimer les besoins et évaluer leur **faisabilité** ; une **maquette** pourra être utile ainsi qu'une étape de **simulation**
- difficulté à être **précis** et non ambigu en s'exprimant dans un langage naturel, d'où l'intérêt d' utiliser le même **formalisme** que pour les spécifications fonctionnelles.

Le cahier des charges est **rédigé par l'équipe marketing** ou par le **client** bien souvent en collaboration avec le responsable du développement.

C'est un **document technique** sans considérations économiques. Il ne faut pas par exemple chercher à justifier le développement du produit en termes de rentabilité...

Il s'adresse au **client et aux développeurs** et sera à la base du contrat.

Les besoins doivent être

- **précis :** problème bien délimité et caractéristiques techniques énoncées
- **cohérents** entre eux et avec l'environnement économique et technique
- **complets :** ils doivent tenir compte de tous les aspects,
  exemple : dans un système de contrôle d'avion les forces de Coriolis avaient été oubliées et on observait une déviation systématique
- **testables** par une métrique,
- **traçables :** possibilité d'identifier ce qu'il advient de ce besoin dans les phases ultérieures du cycle de vie
- **maintenables / flexibles**
  comment pourra-t-on prendre en compte les évolutions?

Attention à ne pas confondre but et besoin
exemple : système agréable à utiliser et commandes activables par menus

Critères de succès

Le processus d'analyse et d'expression des besoins demande un effort de **communication** entre le client (utilisateur) et le fournisseur (unité de développement).

La rédaction du cahier des charges est confiée à **l'analyste** (à prendre ici comme un rôle) qui doit posséder les **qualités** suivantes :

- capable d'**abstraction** et de **structuration ;** en effet les besoins de l'utilisateur sont souvent confus et contradictoires.
- capable de **connaître et anticiper le processus de développement** décrit dans le cycle de vie, par exemple il doit pouvoir prendre en compte des besoins liés à la maintenabilité du produit insoupçonnés par le client
- capable de **recul**, il ne doit pas se perdre dans les détails

On reconnaît un bon cahier des charges aux **critères** suivants

- il se place au bon niveau de **généralité**
- il **décrit bien le problème posé**
- il définit des **critères de validation**
- il permet d'exprimer facilement un **changement** dans les besoins

On sera particulièrement vigilant dans le cas où le cahier des charges est rédigé par un développeur. Un cahier des charges peut être constitué d'un prototype assorti de la description de contraintes de réalisation.

Attention : le cahier des charges ne doit décrire que les informations et traitements visibles de l'**extérieur du produit**.

Expression du besoin en langage naturel

La plupart du temps le cahier des charges est rédigé en **langage naturel,** c'est une bonne chose car le langage naturel est compréhensible par tous, **mais** ceci est une **source de problèmes majeurs** car le langage naturel induit des **imprécisions et des ambiguïtés** qui devront être levées lors de la phase d'analyse.

Il est conseillé de rédiger le cahier des charges en scindant le texte en **paragraphes qui** permettront une **traçabilité** plus grande du projet**.**

Toutefois , on peut facilement

- **mélanger** Besoins, Buts, Contraintes.
- inclure de **multiples concepts** dans un même paragraphe

Ces problèmes peuvent avoir des conséquences graves sur les phases ultérieures du cycle de vie (maintenabilité, traçabilité)

Exemples

Examinons un cahier des charges facilement accessible et qui a fait l'objet de nombreuses révisions : le CAHIER DES CHARGES STONEMAN [Buxton 1980] . Il s'agit de définir les besoins d'un environnement de programmation ADA (Ada Programming Support Environment).

Ce document est écrit en langage naturel et n'échappe pas aux travers engendrés par ce manque de formalisme.

> *4.C.1* Une interface virtuelle indépendante de la machine hôte doit être fournie pour l'interface utilisateur de l'APSE.

Le paragraphe 4. C. 1 est l'expression d'un **besoin**

> 4.C.2 L'interface virtuelle doit être basée sur un petit nombre de concepts généraux, simples, faciles à comprendre et à utiliser.

Le paragraphe 4.C.2 exprime un **but** certainement très louable, pas un besoin, car il est difficile d'évaluer généralité et simplicité.

> 4.C.8 Il doit être possible d'exprimer toute communication entre l'APSE et l'utilisateur à l'aide de l'ensemble des caractères standards Ada.

Le paragraphe 4.C.8 n'exprime pas une fonction du produit, mais une **contrainte** de réalisation.

Dans sa **version 1**, le CAHIER DES CHARGES STONEMAN comporte des exemples incluant de **multiples concepts** au sein d'un même paragraphe :

> 4.A.3 Une collection d'informations identifiables individuellement dans la base de données est appelée <<objet>>. On accède à chaque objet stocké dans la base de données par un nom unique. La base de données permet de maintenir des relations entre objets.

> 4.B.1 Objets : une collection d'objets identifiables individuellement dans la base de données est appelée un objet. Un objet a un nom qui l'identifie de manière unique dans la base de données. Il a des attributs et contient des informations. Par exemple, un objet peut contenir une unité de programme Ada, un fichier de données, un jeu de tests, un regroupement d'autres objets (une configuration), etc.

Dans la **version 2,** chaque paragraphe exprime une **notion unique :**

1. Base de données KAPSE
   - 1.1 La base de données KAPSE, appelée KDB, est constituée d'entités nommées appelées *objets.*
   - 1.2 La KDB doit permettre de stocker et de maintenir des relations entre objets.
   - 1.3 La KDB doit garantir que l'on accède aux objets par l'utilisation de noms distincts.
   - 1.4 Les objets de la KDB
     - 1.4.1 Un objet de la KDB est une collection d'informations identifiables individuellement. Des exemples d'informations maintenues dans des objets sont des unités de programme Ada, des fichiers de données, des jeux de tests, des regroupements d'autres objets (*configurations*), etc.
     - 1.4.2 Un objet de la KDB est constitué de deux composants, son *contenu et ses attributs.*
     - 1.4.3 Chaque objet de la KDB a un nom propre, qui l'identifie de façon unique

Approche pour la compréhension d'un produit

Un produit effectue un traitement sur de l'information qui existe dans un certain environnement.
Pendant la phase d'expression des besoins, seuls les **traitements visibles de l'extérieur** du produit ont un intérêt. Pendant cette phase, on s'intéressera donc

- à la **structure de l'information manipulée** par le produit
- au **flot d'informations** dans le produit

Pour cela,

1. On fera une brève description du monde extérieur (environnement) dans lequel le produit existera et on placera l'utilisateur dans cet environnement.
   Le but de cette étape est de déterminer l'information qui influence le produit.

2. On établira un modèle conceptuel qui permette de décrire le produit vu par l'utilisateur. On exprimera ici les relations entre ce qui se passe dans le produit et ce que perçoit l'utilisateur.
   Dans cette étape il est essentiel de se débarrasser du "bruit" pour aller à l'essentiel

Cette phase nécessite de réaliser des **interviews** exhaustives des différents **utilisateurs** du futur produit afin de recueillir des informations sur leur travail actuel (sans le nouveau produit) et des souhaits pour le futur environnement.

4.4. EXEMPLE

La suite du cours s'appuie sur l'étude d'un exemple : la gestion d'une médiathèque municipale. L'appel d'offres est fourni en annexe 1

Le devis retenu en réponse à l'appel d'offres est fourni en annexe 2.

Nous décrivons ici les résultats des interviews préalables à la réécriture du cahier des charges et à la réalisation du prototype.

Il s'agit de gérer les emprunts de livres et de cassettes dans une médiathèque municipale. (Environ 2000 livres et 1000 cassettes).

Le prêt de livres et de cassettes se fait au moyen d'une carte magnétique attribuée à chaque adhérent. Le matériel concernant les cartes magnétiques n'est pas choisi et fera l'objet d'une étude de marché.

On dispose d'un logiciel existant traitant les emprunts de cassettes vidéo, après examen, il s'avère que ce logiciel est réalisé sous excel et ne pourra être réutilisé dans un contexte élargi.

L'inscription de l'adhérent est faite par le gestionnaire qui enregistre les coordonnées de l'adhérent et encaisse sa caution (remboursable sauf en cas de non retour de livre ou cassettes) et sa cotisation annuelle.

Les livres sont disponibles en rayon afin de pouvoir être manipulés, par contre les cassettes sont rangées derrière un comptoir uniquement accessibles par le gestionnaire (vol).

Ainsi pour emprunter un livre, l'adhérent le choisit simplement sur l'étagère et le présente au gestionnaire qui enregistre le prêt.

Par contre pour les cassettes, l'adhérent consulte un catalogue et demande au gestionnaire la cassette de son choix. A chaque emprunt de cassette le système vérifie que l'adhérent est à jour de sa cotisation, qu'il reste un exemplaire de la cassette disponible et sort automatiquement la cassette du stock.

Quand un livre ou une cassette n'est pas disponible on peut les réserver.

A chaque retour de livre ou de cassette, le gestionnaire indique à l'adhérent s'il doit réapprovisionner son compte (paiement de cotisation annuelle).

En ce qui concerne la commande de nouvelles cassettes ou de nouveaux livres, le marchand fait son choix dans un catalogue, le système édite un bon de commande pour le fournisseur approprié. Lors de l'arrivée de la cassette ou du livre le gestionnaire l'introduit dans le stock.

Pour les livres, le catalogue de prêt inter-bibliothèques est disponible sur Internet et consultable par le gestionnaire au cas où un adhérent souhaite emprunter un livre non disponible.

La gestion de la comptabilité liée aux livraisons n'est pas traitée.

On suppose le système monoposte, c'est à dire qu'un seul gestionnaire peut enregistrer un emprunt à un instant donné.

Un adhérent peut réserver un film ou un livre s'il n'est pas disponible lorsqu'il passe à la médiathèque.

4.5 . LE DOSSIER D'ANALYSE

Il doit fournir une description complète et détaillée des fonctions du produit dans sa relation avec l'environnement. Il sera utilisé :

1. :Par le client

- pour avoir une description précise de ce qui sera réalisé, pour anticiper sur la mise en exploitation

2. Par les développeurs
   - comme référence précise et non ambiguë sur ce qu'il s'agit de réaliser
   - pour créer des jeux de tests

Le dossier d'analyse, le plan qualité et le plan projet définissent les termes du contrat passé entre le client et le fournisseur de logiciel.

Principes de base

L'auteur du dossier d'analyse appartient à l'unité de développement. Le dossier d'analyse contient une description du produit vu de l'extérieur, suffisamment précise pour pouvoir en dériver :

- la documentation client,
- les jeux de tests

L'architecture modulaire du produit ne doit pas y figurer. Seules sont décrites les fonctions visibles par l'utilisateur.

Exemple : Droits d'accès à telle fonction suivant le profil des utilisateurs. Tout doit se relier clairement à des objectifs/besoins exprimés dans la spécification d'objectifs et le cahier des charges (Critère de traçabilité).

Plan type du dossier d'analyse

1. Introduction

Objectifs, Fonctionnalités attendues, Environnement
Faisabilité, Justifications, Ressources nécessaires,
Éléments de coûts et d'échéancier

2.Fonctions du produit

2.1 CONCEPTS ET TERMINOLOGIE
Glossaire de l'application (même document que celui référencé dans le chapitre précédent)
2.2 DESCRIPTION FONCTIONNELLE EXTERNE
Entrées, Traitement, Sorties de chaque fonction
Préciser les conditions d'arrêt, d'exception, les points de reprise. Traitement des anomalies
Organisation logique des données, domaine de variation.
Interfaces homme - machine (écrans, états, tableaux....)

3 Description interne

3.1 ALLOCATION
Il s'agit de décrire l'interaction du système avec son environnement. Chaque tâche décrite en 2.2 est allouée à un élément du système. Le Hard et le Soft sont décrits séparément. En particulier l'utilisation de bases de données existantes doit être mentionnée.
3.2 CONTRAINTES
Contraintes de réalisation, ex : encombrement mémoire
Contraintes de qualité, ex : précision du calcul,
Performances, priorités, séquencement (temps réel)
Critères de vérification des contraintes

4. Questions ouvertes et réponses apportées par les développeurs.

5. Éléments de livraison

Cahier provisoire de Recette

4.6. MAQUETTAGE ET PROTOTYPAGE

Pourquoi prototyper ?

Après la réalisation des spécifications fonctionnelles, il est bon de montrer au client à quoi ressemblera le produit final. Boehm constate que 95 % du code à réécrire provient de modifications destinées à satisfaire des changements dus au client et que 12 % des erreurs totales proviennent de ce code. On utilise les techniques de maquettage et prtotypage pour disposer plus tôt dans le cycle de vie d'objets exécutables .

La maquette est un système incomplet dont l'aspect extérieur est le même que celui du système final à réaliser. Elle est destinée à tester l'ergonomie du produit final et permet d'instaurer le dialogue entre développeur et utilisateur. Le maquettage intervient dès l'écriture du cahier des charges lorsqu'il est nécessaire de préciser certains points entre l'utilisateur et le développeur. Il devient indispensable lors de la description de l' interface utilisateur comme on le verra au paragraphe suivant.

La maquette ne permet aucun test de performance ni de temps de réponse. Il s'agit essentiellement d'un produit jetable. (cf cycle de vie avec maquettage au chapitre 1)

Le **prototypage** intervient en phase d'analyse pour étudier la **faisabilité** du système. Le **prototype** est une esquisse de ce que sera le produit final. Il réalise les **fonctionnalités** du système final **sans respecter les contraintes de fiabilité, robustesse**...

Les intérêts du prototypage sont multiples :

- mettre en évidence les incompréhensions développeur-utilisateur,
- détecter les oublis de spécifications
- identifier les services difficiles à utiliser
- découvrir les contradictions
- évaluer rapidement la faisabilité
- servir de base à l'écriture de spécifications complètes

Le processus de prototypage permet d'établir la discussion entre développeurs et utilisateurs ; il permet de **valider les spécifications** et est en général jetable (se reporter aux différents types de cycle de vie au chapitre 2). Il permet de montrer que

- les besoins utilisateurs sont valides et réalisables par programme
- les spécifications ne sont pas contradictoires
- les spécifications sont complètes et réalistes

Techniques de prototypage :

Un prototype doit implémenter strictement les fonctionnalités du produit et doit donc **ignorer les spécifications non fonctionnelles :**

- temps de réponse, contraintes mémoire...
- prise en compte de traitements d'erreur,
- prise en compte des standards,
- prise en compte d'un certain niveau de fiabilité.

L'écriture d'un prototype peut se faire dans un langage quelconque, on distingue

- les **langages impératifs (ou procéduraux)** ayant un fort pouvoir d'expression, le plus souvent non typés, dans un environnement interactif. On trouve des prototypes écrits en C, en lisp, en shell ou en PERL.
  On gagne en productivité le système peut être réalisé en un temps très bref. Le produit obtenu n'a pas les qualités de fiabilité / efficacité du produit final.
- **les langages déclaratifs ou non procéduraux** on utilise alors un langage du même type pour décrire l'application. Le plus connu de ces langages est PROLOG. On ne décrit pas les algorithmes à exécuter pour produire les résultats mais les objets et les relations entre eux. On parle de base de faits et de bases de règles. A partir de faits et de règles, on cherche à établir si un ensemble de faits est réalisé ou non.
- **les langages spécifiques** , par exemple, dans le domaine du temps réel on peut citer les langages LUSTRE, SIGNAL ou ESTEREL qui sont à la fois langages de spécification, prototypage et réalisation

Cycle de vie et prototype

En ce qui concerne le devenir du prototype, nous avons vu au chapitre 1 que plusieurs choix sont possibles :

- **prototype jetable :** après avoir étudié la faisabilité du produit grâce à un prototype, on jette celui ci et on recommence toute la programmation.
- **prototype évolutif** : la première version du prototype est l'embryon du produit final. On itère jusqu'au produit final. On notera qu'avec cette approche, il est très difficile de mettre en œuvre des procédures de validation et de vérification.

Prototyper n'est pas spécifier

Il parait tentant d'utiliser le prototype comme dossier de spécifications, il faut absolument se garder de ce piège, en effet lors de la réalisation du prototype, les critères non fonctionnels sont absents (sécurité, fiabilité, rapidité...). D'autre part, l'utilisation d'un prototype n'est pas équivalente à celle de l'outil définitif. Enfin rappelons que le dossier de spécifications est la base du contrat passé entre développeur et client et qu'une implémentation n'est pas une base fiable pour un tel contrat.

Nous insistons sur le fait qu'une *ré-implémentation est toujours recommandable* même si pour des raisons de productivité elle réutilise des composants développés pour l'écriture du prototype (on notera alors l'intérêt d'utiliser un langage identique à celui de l'implémentation finale et l'impact des langage à objets dont nous rediscutons dans le chapitre sur la conception globale). Nous citons quelques bonnes raisons pour cette ré-implémentation :

- la prise en compte de contraintes non fonctionnelles n'est pas forcément facile dans un prototype non prévu pour ça au départ
- l'introduction de différents "patch" correspondant à la prise en compte de besoins utilisateurs conduit souvent à une dégradation de la structure générale du système.

4.7. SPECIFICATION D'INTERFACES

Introduction

La concurrence entre les applications se fait souvent au niveau de l'interface utilisateur. La demande du marché pour des interfaces WIMP (Windows Icons Menu Pointing Device) est de plus en plus pressante.

La spécification de l'interface intervient très tôt dans le cycle de vie, généralement au moment des spécifications fonctionnelles, elle donne lieu à de nombreuses discussions avec l'utilisateur puisqu'elle est en général le reflet des fonctionnalités du produit.

On aura soin d'utiliser une maquette pour valider l'interface car c'est le seul moyen efficace d'avoir une discussion profitable avec l'utilisateur.

Pour créer un premier modèle d'interface on repartira du modèle conceptuel des données et on s'attachera à définir un modèle conceptuel de dialogue. On cherchera ensuite à mettre en œuvre ce modèle en utilisant au mieux les outils mis à disposition tels menus (fixes, déroulants ou pop-up), boutons, boites de dialogue, inverse vidéo, cases à cocher, bip, icône, raccourcis clavier....

Ergonomie

Une interface ergonomique est à la fois conviviale, lisible, standard, efficace.

La **convivialité** peut se définir en terme de facilité d'utilisation et se mesurer en nombre de jours d'apprentissage.

La **lisibilité** dépend de la structure d'écran, on veillera à aligner les différents champs, regrouper entre eux les champs similaires, on privilégiera un sens de lecture habituel, on cherchera à garder une stabilité d'un écran à un autre. Un autre phénomène à prendre en compte est le degré d'excentricité des écrans successifs qui ne doit pas être trop grand.

La **standardisation** des interfaces se décline à la fois par rapport au marché, aux normes de l'entreprise et aux conventions de l'application.

L'**efficacité** des interfaces se vérifiera sur la productivité de ses utilisateurs ; il importe donc de **caractériser l'utilisateur** ou le type d'utilisateur, ses compétences, le but de son travail, les performances attendues, le contexte du travail.

Outils

Depuis la révolution apportée par le Macintosh au début des années 80, le nombre d'outils permettant de créer des interfaces conviviales n'a cessé de croître. Le standard X Window est aujourd'hui masqué par des boites à outils d'un niveau d'abstraction de plus en plus élevé, OSF Motif, Tcl/Tk plus facile d'utilisation. On trouve aussi des outils exclusivement dédiés au prototypage comme NSDK sur PC. Tous les outils permettant une approche RAD font partie des outils de prototypage (La gamme Visual, Delphi, tous les environnements Java…)

Enfin, beaucoup d'outils de spécification fournissent un environnement de prototypage d'interface (Oracle, O2...) qui sont utiles dans le contexte de l'utilisation de ces outils.

4.8 SIMULATION

Pendant la phase de spécifications fonctionnelles, le développeur peut être aidé par des outils de simulation.

Ceux-ci peuvent être utiles pour comparer plusieurs solutions envisageables, pour faire une étude de faisabilité, pour analyser plus précisément un élément critique du système.

Les résultats de ces simulations peuvent constituer des arguments décisifs pour les choix ultérieurs. Ces résultats doivent alors être présents dans le dossier de spécifications fonctionnelles

Parmi les méthodes utilisées on peut citer :

- théorie des réseaux de files d'attente (le plus utilisé)
- réseaux de Pétri (petits systèmes)
- chaînes de Markov (petits systèmes)

On peut également utiliser

- **des langages évolués** (C, C++, Java...)

On a alors plus de souplesse et de précision mais le temps d'analyse est plus long, (justifié si on a besoin de précision et d'efficacité)

- **des logiciels dédiés** qui permettent de simuler un système discret quelconque choisi parmi un ensemble de systèmes discrets ayant des caractéristiques communes

L'intérêt d'un tel logiciel est de permettre la description d'un système sans connaissance particulière d'un langage informatique.

Le contrôle est ensuite possible de façon interactive.

Ex : évaluation des performances des réseaux de données, productique

- **des langages de simulation.**

De tels langages comprennent généralement

des modèles préprogrammés de fonctions à réaliser

-un échéancier assurant les modifications de l'état des modèles en fonction des décisions prises

-des fonctions statistiques réalisant les calculs souhaités lors des simulations

Ils sont généralement associés à des interfaces graphiques facilitant la saisie des données, l'affichage des résultats, l'animation de la spécification

Deux approches sont possibles :

L'approche par événement : l'utilisateur définit les événements et les logiques de changement d'état correspondantes

L'approche par processus : l'utilisateur peut définir lui même les processus de son application à l'aide processus standards(consommation de temps, gestion de files d'attente, gestion de ressources).

Ex : SIMULA, QNAP2

Le langage peut également fournir un certain nombre de processus élémentaires qui suffisent dans la plupart des cas pour modéliser le système entier sans compétence informatique particulière.

On peut trouver des langages combinant les 2 approches (SLAM)

De façon générale on peut dire que de la qualité de la modélisation dépend la qualité de la simulation. Il n'est pas rare de trouver des couplages des 2 outils.

Par exemple ASA est à la fois un outil de spécification et de simulation, TEAMWORK fournit également cette facilité.

5. TECHNIQUES ET OUTILS DE SPECIFICATION

Les techniques de spécification sont utilisées pendant la phase d'analyse. Elles permettent la formalisation du cahier des charges. Pendant la phase d'analyse, on va modéliser le système en prenant en compte différents points de vue: données, traitements, utilisateur, service, qu'il convient de structurer. Les différents formalismes présentés découlent donc de ces différents points de vue. Un même formalisme de spécification peut être utilisé dans différentes méthodes d'analyse/conception qui seront vues dans les chapitres suivants.

Nous présentons ici les techniques les plus fréquemment utilisées.

5.1. PRESENTATIONS INFORMELLES

Il est toujours possible d'utiliser un énoncé des spécifications en langage naturel. Toutefois, même si les spécifications respectent un plan type standard (DOD 2167-A par exemple), l'utilisation d'un langage informel induit souvent un manque de cohérence et induit des ambiguïtés qui peuvent avoir des conséquences regrettables sur la conception du produit et l'organisation du projet surtout dans le cas de logiciels complexes mettant en œuvre plusieurs équipes.

5.2. PRESENTATIONS FORMATTEES

Les présentations formatées récapitulent dans des tables les informations extraites de l'analyse du cahier des charges, elles sont en général un complément d'information au texte tenant lieu de spécification. Elles peuvent également être le point de départ d'une analyse plus approfondie.

Modèle conceptuel

Le modèle conceptuel sert de point de départ à l'analyse et à la réalisation de l'interface homme-machine s'il y a lieu. Il peut être élaboré à la suite des interviews des utilisateurs du futur produit. Il permet de recenser les fonctionnalités attendues et les objets qui interviendront dans le futur logiciel.

Le modèle conceptuel définit pour chaque grande fonction du produit :

- les **objets (ou entités)** que le produit crée ou manipule
- les **attributs** de ces objets
- les **opérations** à réaliser sur ces objets

Les objets ainsi identifiés apparaîtront ensuite dans le dictionnaire des données vu plus loin.

EXEMPLE 1 : MESSAGERIE ELECTRONIQUE (xmh)

Il s'agit de décrire une messagerie électronique permettant à un utilisateur UNIX de

- lire les messages qui lui sont destinés
- ranger les messages dans des dossiers
- consulter les dossiers
- envoyer des messages, de les faire suivre...
- connaître les noms des autres abonnés du système

Objets
- message reçu
- message en cours
- dossier
- abonné

Relations
- un dossier possède une liste de messages
- un abonné possède un ensemble de dossiers
- un message est destiné à un abonné

Fonction du produit	Objet & Attributs	Opérations	Règles de comportement
gestion d'un dossier	une liste de messages	créer, détruire	
lecture de messages	messages reçu : non modifiable	lire, classer, répondre, faire suivre.	
création de messages	messages en cours : modifiable	éditer, envoyer.	message < 1K car., ne modifier que la dernière ligne.
adminis-tration	abonnés : nom	abonner, désabonner.	pas plus de 1000 abonnés

EXEMPLE 2 : TRAITEMENT DE TEXTE *(d'après P. Baudelaire)*

On montre par cet exemple que le modèle conceptuel sur lequel s'appuie le produit a une influence considérable sur le produit

A. MODELE DE SIMULATION

On essaie de recréer la vieille technologie en simulant la machine à écrire, le papier...
On effectue des actions très visibles sur l'écran en cours de traitement.

La page est une zone rectangulaire contenant
des cellules (caractères ou blancs),
un curseur.

Les actions s'expriment par la frappe d'un caractère à l'endroit où se trouve le curseur (Ecrire, Effacer, Copier, Déplacer)

Ce modèle conceptuel est simple et très limité.

B.MODELE STRUCTUREL

Le deuxième modèle manipule des entités abstraites laissant au logiciel le soin d'assurer la présentation du document.

Un document est vu comme une chaîne de caractères infinie, organisée hiérarchiquement en sections, paragraphes, mots...

Toute manipulation se fait à travers cette structure logique
Insérer, détruire, remplacer,
Déplacer, copier,
Enregistrer des règles de formatage

Ce deuxième modèle conceptuel permet :
la génération automatique de la présentation des documents,
des opérations d'édition beaucoup plus sophistiquées :

Exemple :
Changer le style de la présentation sans intervenir au niveau du contenu du document.

Dictionnaire des données

Il est constitué au fur et à mesure de l'analyse par l'ensemble des données du problème. Le modèle conceptuel vu précédemment peut en constituer une première ébauche. Le dictionnaire des données contient les noms de tous les objets utilisés (variables, fonctions, classes...) classés par ordre alphabétique ainsi que les synonymes ou alias éventuels.

Certains outils logiciels créent des dictionnaires de données qui permettent de faire des références croisées indiquant que telle procédure contient telles variables ou que telle fonction est référencée dans telle autre. Ces informations sont d'une grande utilité en maintenance lorsque l'on veut connaître les effets de bord potentiels d'une modification.

Tables de décision

Les tables de décision permettent de définir les valeurs de sortie d'un processus en fonctions de ses valeurs d'entrées et de leurs combinaisons.

Elles sont parfaitement adaptées à la spécification des systèmes dont les sorties ne dépendent que des entrées.

Tables de transitions

Comme les diagrammes états-transitions vus plus loin, elles représentent l'automate modélisant la dynamique du système. L'utilisation de ces tables est adaptée lorsque les sorties sont déterminées par les entrée et l'historique des états antérieurs.

5.3. TECHNIQUES GRAPHIQUES OU SEMI FORMELLES

Ces techniques de spécification sont les plus couramment utilisées dans les méthodes modernes d'analyse/conception. Elles favorisent la communication entre développeurs et utilisateurs en utilisant une représentation graphique des données et des traitements. Le langage graphique est normalisé et sa sémantique est non ambiguë. Toutefois les diagrammes sont en général accompagnés de texte écrits en langage naturel ce qui amène à les qualifier de techniques de représentation semi-formelles. Les textes informels peuvent être enrichis de prédicats logiques d'entrée et de sortie qui favorisent la cohérence et la complétude des spécifications. Nous verrons plus loin les techniques de spécifications formelles qui s'appuient

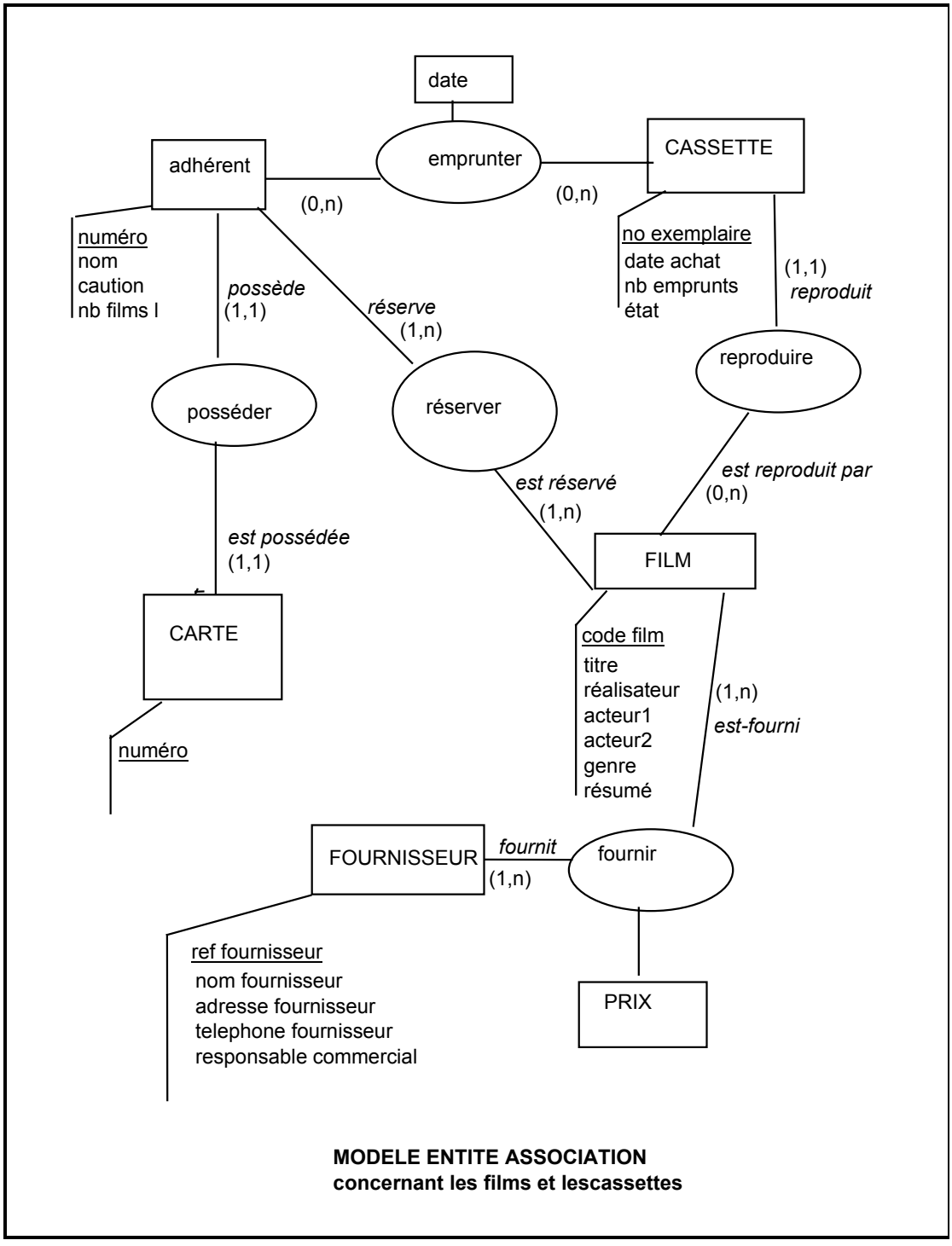
quant à elles uniquement sur ce type de prédicats et d'équations; mais nous tenons à signaler que des retombées de ces méthodes peuvent être introduits lors de l'utilisation de spécifications semi-formelles.

Modèle entité-associations

Le modèle entité-association connu sous le nom anglais de *Entity-Relationship model* est du à Peter Chen [CHE76 : Peter Chen The entity relationship model Toward a unified view of Data ACM transactions on Data base system 1976]. Il est aujourd'hui le plus couramment utilisé pour représenter les données d'un système et les relations les liant. Il était autrefois utilisé dans les méthodes de conception reposant sur une *approche par les données* que nous ne présentons pas en détail ici mais dont nous dirons toutefois quelques mots dans le chapitre "méthodes de conception".

Une *entité* est un objet que l'on sait distinguer d'un autre (ex : livre, employé, contrat, client..)
Chaque entité a des *attributs ou propriétés* (ex : titre, nom, numéro..).
Chaque attribut prend ses *valeurs sur un domaine* de valeurs autorisées (ensemble des chaînes de caractères, ensemble des entiers de 1 à 150...).
Les entités peuvent être regroupées en *ensemble d'entités* ayant les mêmes attributs avec des valeurs différentes (ex : les livres d'une bibliothèque, les clients d'une entreprise, les films d'un catalogue). Une entité devant être parfaitement caractérisable, on définit pour chaque entité ou ensemble d'entités une *clé ou identifiant* permettant de la distinguer d'une autre (exemple : le titre d'un livre ou son numéro si plusieurs exemplaires du même livre)
Une *relation* entre différentes entités ou ensemble d'entités est une association entre ceux-ci.
exemples :
un livre possède un auteur, un film est reproduit sur des cassettes
un client d'une banque possède un ou des comptes en banque.
Ce dernier exemple nous amène à définir la *cardinalité* d'une relation qui permet de savoir combien d'occurrences d'une relation existent (au minimum et au maximum) pour un ensemble d'entités

Exemple de la médiathèque :



MODELE ENTITE ASSOCIATION
concernant les films et lescassettes

Les entités ADHERENT, FOURNISSEUR, FILM, CASSETTE, CARTE sont représentées par des rectangles. Ces entités ont des attributs qui les décrivent plus précisément, par exemple un FILM est décrit par son titre, son réalisateur, son résumé...L'*identifiant* de l'entité est souligné, par exemple l'identifiant d'un CLIENT est le numéro client , attribut artificiel rajouté lors des spécifications pour les besoins de la gestion.

On notera les cardinalités (1,1) , (0,n), (1,n) décrites sur chaque relation . Par exemple un film peut être reproduit sur 0 ou plusieurs cassettes alors qu'une cassette ne reproduit qu'un seul film.. Une carte et un client sont liés par une relation biunivoque.

On remarquera l'attribut état pour une cassette qui peut prendre les valeurs disponible, empruntée, commandée, réservée..

Utilisation

Le modèle Entité-Association est bien adapté à la conception des bases de données relationnelles, ou à l'enrichissement du modèle de données d'une entreprise lors de l'introduction de nouveaux produits. Il est également utilisé sous une forme enrichie dans la plupart des méthodes de conception orientées objets actuelles étudiées plus loin .

Diagrammes de flots de données

Dans le domaine des *logiciels industriels* les méthodes les plus couramment employées depuis les années 80 et encore aujourd'hui sont celles reposant sur la *modélisation des traitements***.** Les diagrammes de flots de données montrent comment chaque processus transforme ses entrées en sorties (flot entrant, flot sortant). Les gisements d'information sont également modélisables.

Les diagrammes de flots de données sont en général complétés par un *diagramme de contexte* qui permet de faire figurer les *agents extérieurs* intervenant sur le produit

Exemple

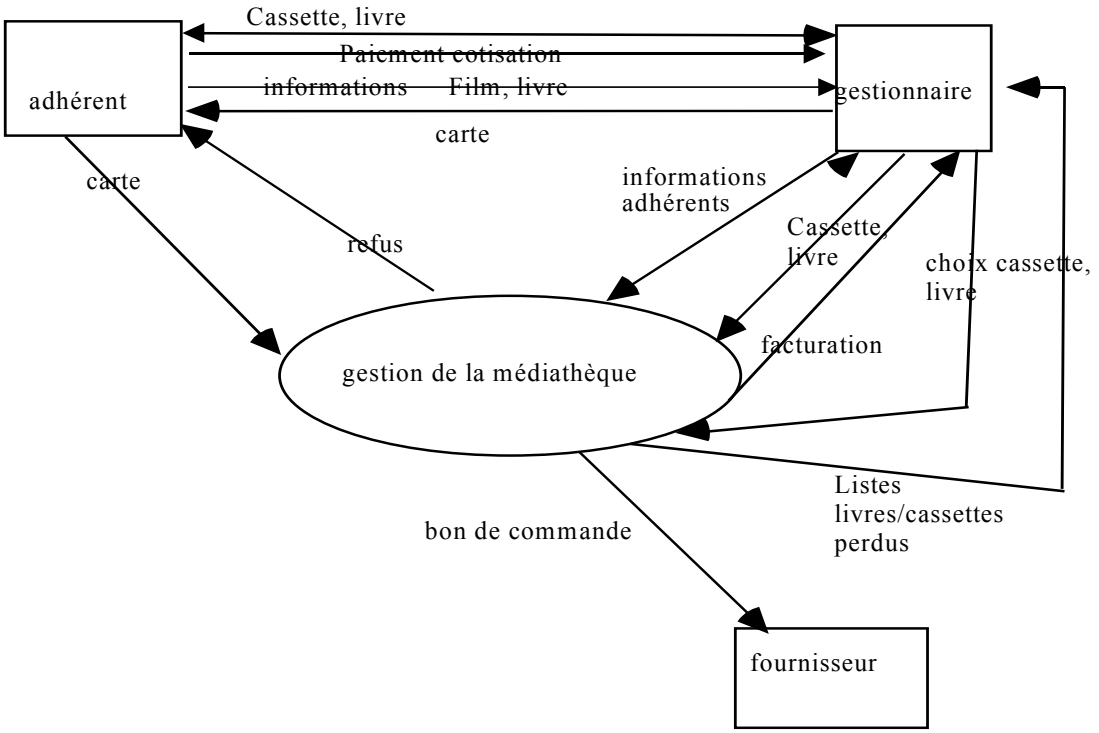


DIAGRAMME DE CONTEXTE

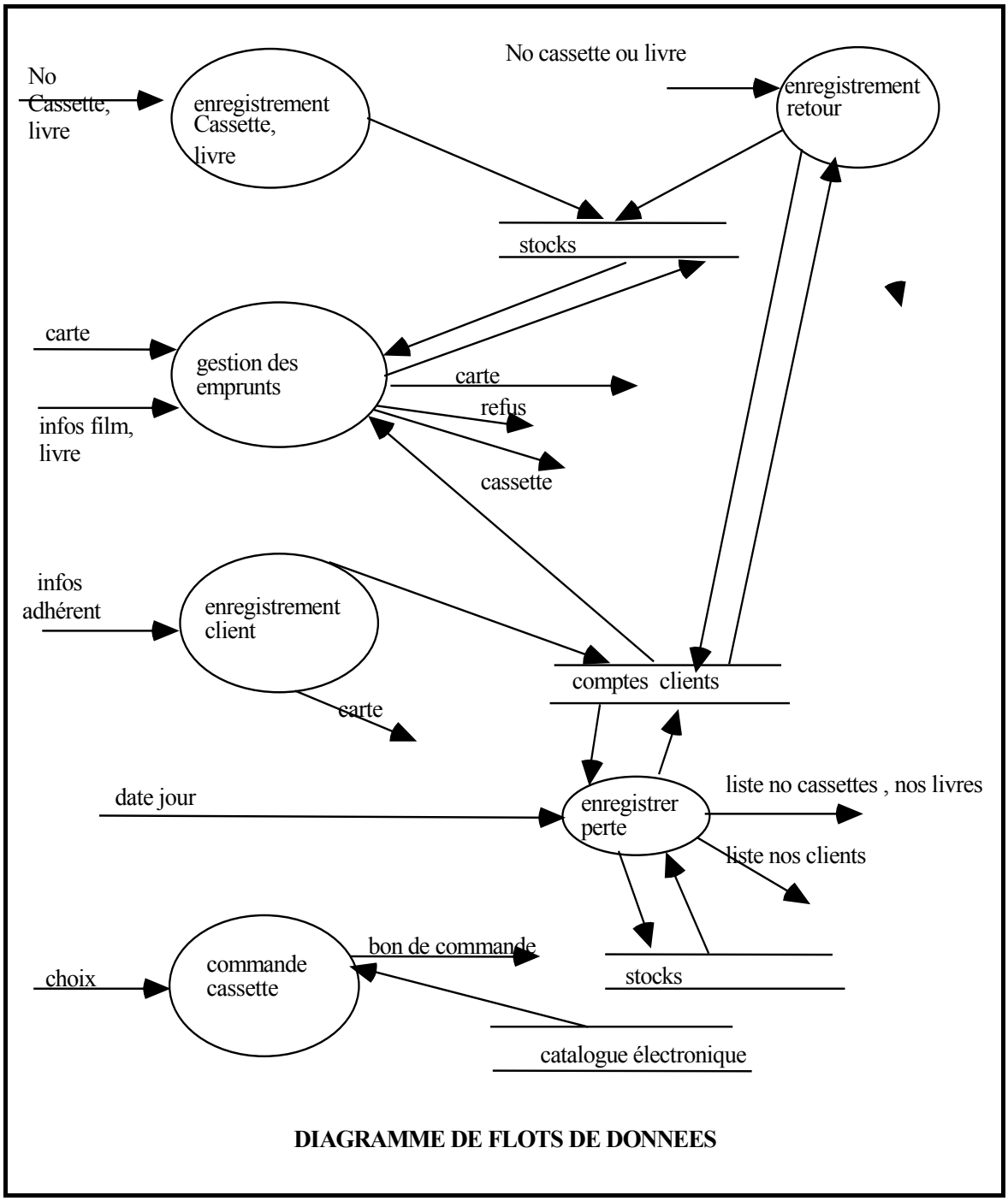


DIAGRAMME DE FLOTS DE DONNEES

Affinements successifs

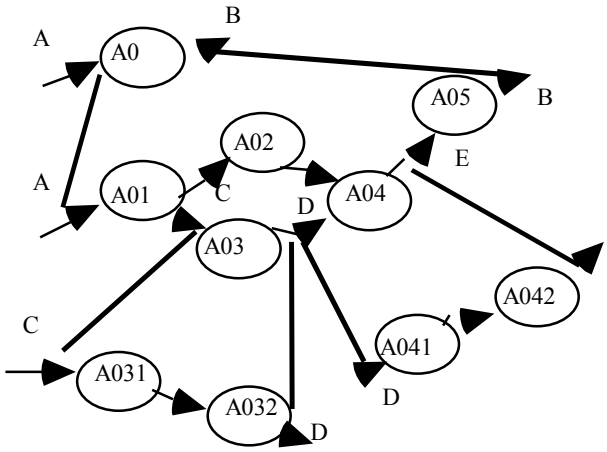
Le diagramme de flots de données correspond à la décomposition de la "boite" *gestion des cassettes* du diagramme de contexte.

La plupart des méthodes utilisant les diagrammes de flots de données s'appuient sur une approche par affinements successifs. La description du produit se fait à plusieurs niveaux.

Les différents niveaux sont numérotés de façon *arborescente.*

Les entrées et les sorties doivent être *cohérentes* entre les différents niveaux.

Le schéma ci-dessous représente un exemple de décomposition.



Le diagramme de contexte représente la décomposition au niveau 0. La boite gestion des emprunts de cassettes peut être décomposée comme suit au niveau 2

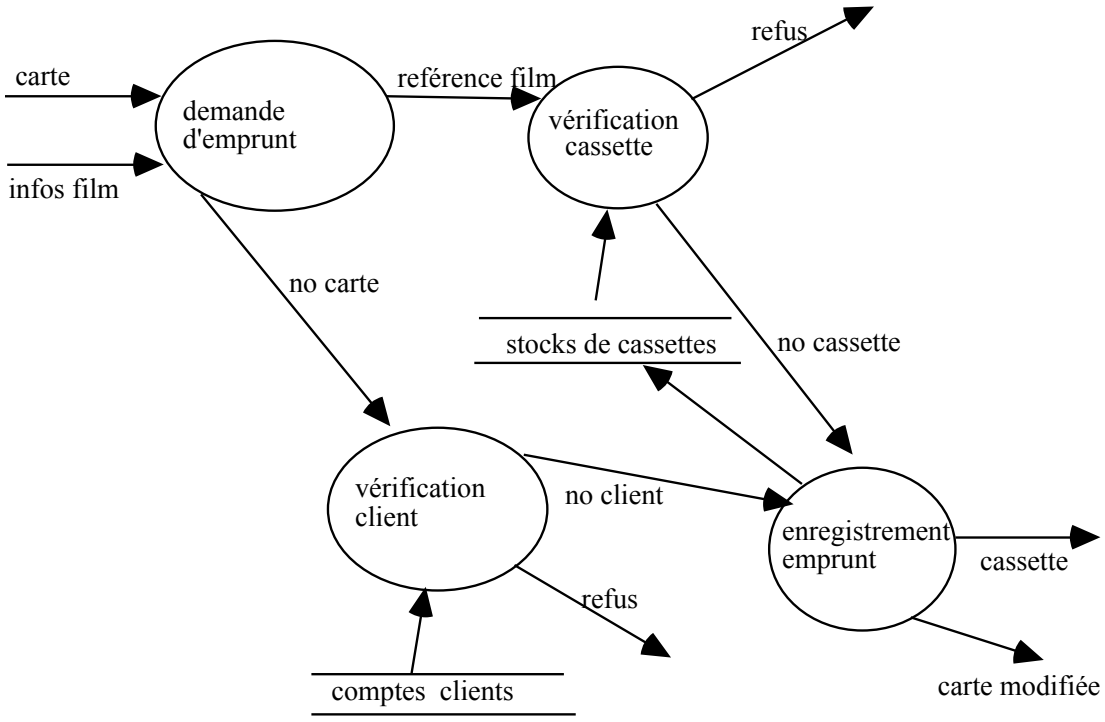
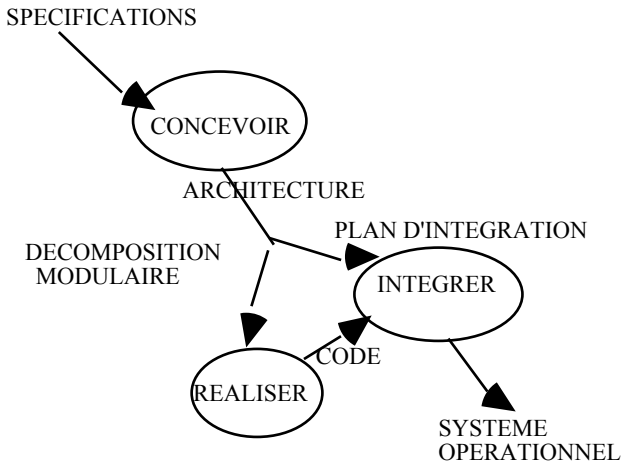


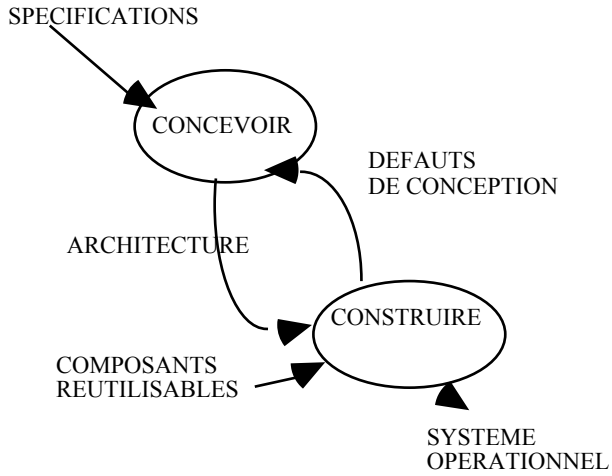
DIAGRAMME DE FLOTS DE DONNEES
NIVEAU 2 GESTION DES EMPRUNTS

Précisions sur les DFD

Les flèches sont des câbles qui peuvent :
se partager en câbles plus petits,
se regrouper en câbles plus grands :
Exemple



Le réseau de flèches peut exprimer des *contre-réactions* :
Exemple



Conseils de présentation des DFD

On conseille de définir entre 2 et 6 activités par diagrammes: Si > 6, DFD trop chargé.
Avoir un flot général de données allant de gauche à droite et de haut en bas, privilégier la diagonale
Éviter les croisements de flèches
Décrire chaque fonction au moyen d'un langage pseudo naturel (Program Design Language) , compromis entre la langue naturelle et un langage algorithmique structuré.

Utilisation des DFD

Les DFD sont particulièrement utiles lorsque le système à modéliser est très réactif, c'est à dire lorsque le système est toujours prêt à réagir à l'arrivée de nouvelles données.
Toutefois, il est à noter que les DFD ne permettent pas d'exprimer des *flots de contrôle* (par exemple des boucles), ce ne sont pas des organigrammes.

Diagrammes états-transitions

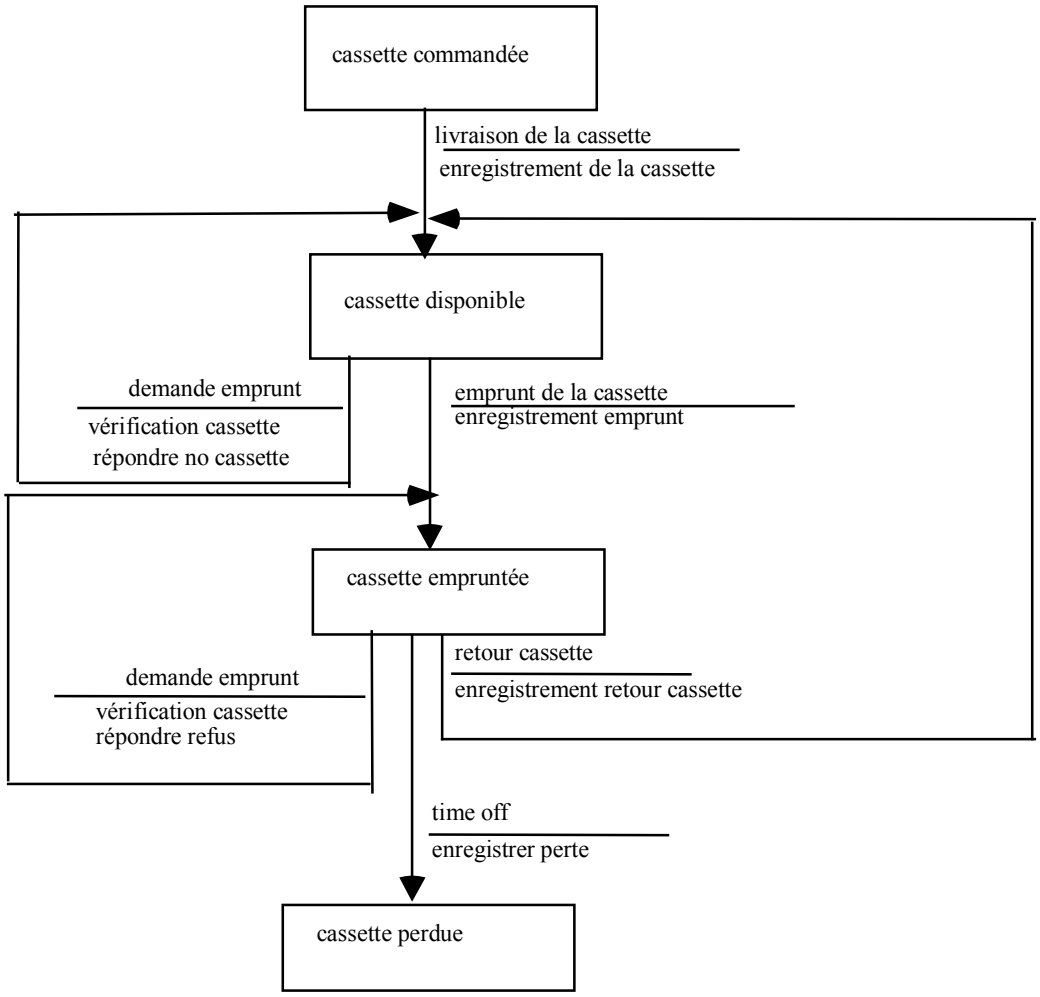
Les diagrammes de flots de données ne rendent pas compte de la dynamique des traitements et de leur enchaînement dans le temps. Cet inconvénient est particulièrement redoutable dans le cas de la modélisation de *systèmes temps réel* ou ayant des interfaces utilisateurs complexes dans leurs enchaînements d'écrans par exemple.

Les diagrammes états-transitions permettent de matérialiser l'incidence des événements sur le comportement du système et indiquent les actions à effectuer.

Ils servent de base à la réalisation des tables (ou matrices) états transitions.

Ils sont très utilisés dans la modélisation du comportement d'un objet réactif dans les méthodes orientées objet actuelles.

Exemple



Ce diagramme permet de préciser les valeurs de l'attribut état associé à l'entité CASSETTE en fonction de différents événements. La commande d'une cassette déclenche la création d'un nouvel objet CASSETTE avec l'état commandé. La réception d'une cassette fait passer dans l'état disponible, un emprunt dans l'état emprunté si elle est disponible, réservé si elle est empruntée.

Un retour client la fait passer de empruntée à disponible.

Au bout d'un certain temps la cassette empruntée est considérée comme perdue.

Réseaux de Pétri / Grafcet

La modélisation des systèmes dynamiques nécessite d'étudier l'ordonnancement des activités Il faut distinguer la partie contrôle de la partie opératoire Les réseaux de Pétri permettent la modélisation et la simulation de systèmes parallèles à événements discrets

Un réseau de Pétri est un graphe biparti constitué de places représentées par des cercles et de transitions représentées par des traits.

L'ensemble des places représente l'ensemble des états du système. Les transitions correspondent aux changements d'états du système.

On définit un marquage qui permet de localiser des marques (jetons) dans certaines places du réseau, on représente ainsi l'état du système à un instant donné.

La présence de jetons dans les places "entrée" (pré-conditions) d'une transition permettent le déclenchement de la transition. Lorsqu'une transition est déclenchée, elle provoque la décrémentation du nombre de marque de ses places entrée et l'incrémentation du nombre de marques dans ses places sortie (post conditions)

Caractéristiques

-l'*indéterminisme* du fonctionnement : quand une condition est satisfaite, on n'a pas forcément déclenchement de l'activité correspondante.

-l'*atomicité* du fonctionnement : lorsque plusieurs pré-conditions sont remplies, une et une seule activité débute. On notera que certaines actions précédemment déclenchables peuvent ne plus l'être ;

Les réseaux de Pétri permettent de modéliser la concurrence, la coopération.
Exemples : Exclusion mutuelle, Producteur/Consommateur.

L'étude des réseaux de Pétri consiste à démontrer certaines propriétés du réseau en utilisant différentes théories :

-*l'algèbre linéaire* :
on s'intéresse alors aux séquences de franchissements représentées par un vecteur caractéristique.

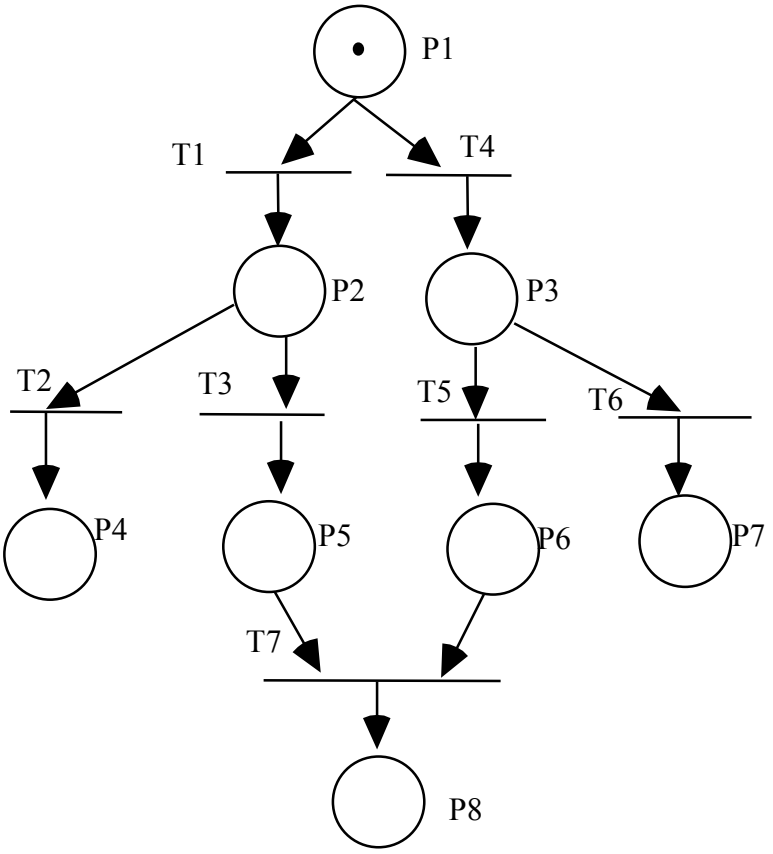
On étudie l'équation caractéristique du réseau : $M= M_0+ C\ s$

où M est un marquage,
M_0 est le marquage initial et
s le vecteur caractéristique d'une séquence de franchissements.

-la théorie des graphes :
on s'intéresse alors à l'étude du graphe des marquages atteints

Exemple

Modélisons par un réseau de Pétri la délivrance d'une cassette



P1 : emprunt demandé
P2: vérification cassette
P3: vérification adhérent
P4 : emprunt refusé
P5 :emprunt possible
P6 :emprunt possible
P7 : emprunt refusé
P8 : emprunt autorisé

T1 : début vérification cassette (action)
T2 : cassette empruntée (condition)
T3 : cassette disponible (condition)
T4: début vérification adhérent (action)
T5 : client OK (condtion)
T6: adhérent non ok (condition)
T7 : autoriser emprunt (action)

L'emprunt sera autorisé si et seulement si les deux critères de disponibilité de la cassette et de solvabilité de l'adhérent sont satisfaits.

Notons que la modélisation par réseau de Pétri permet de représenter la concurrence des deux opérations de vérifications travaillant sur des données distinctes.

Si l'on souhaite introduire plusieurs postes dans le magasin, la modélisation par réseau de Pétri permettra d'éviter que la même cassette soit empruntée par deux personnes.

Extensions

De nombreuses extensions au modèle original ont été proposées, on citera

-*réseaux de Pétri colorés* :

Les marques ont une "couleur" différente au sein d'une même place permettant de les différencier.

- *réseaux de Pétri à arcs inhibiteurs* :

une marque dans une place entrée d'une transition empêche la transaction d'avoir lieu.

Le Grafcet est un outil de spécification d'automates utilisant également des places et des transitions mais dont la signification est duale à celle des réseaux de Pétri. Le grafcet est devenu une norme internationale en 1987.

6. METHODES D'ANALYSE FORMELLES

6.1. INTRODUCTION

Nous avons vu dans le chapitre précédent quelques **méthodes semi-formelles** de spécification. Celles-ci sont largement répandues dans l'industrie grâce aux **nombreux outils** qui les implémentent. Elles présentent cependant l'inconvénient que l'on signalait en introduction : leur **manque de rigueur mathématique.**

Nous abordons ici les **méthodes formelles** de spécification qui bien que **moins répandues** représentent à notre avis l'**avenir** de la spécification pour les raisons suivantes :

- Elles nécessitent une approche plus **profonde** du problème posé et permettent donc de mieux le maîtriser.
- Etant donné un système formel de spécification et un langage de programmation formel, on peut éventuellement **prouver** que le programme est **conforme à sa spécification**. (vérification formelle)
- On peut construire des **outils automatiques** pour assister le **développement**, la **compréhension**, le **debugging** des spécifications formelles.
- Suivant le langage formel utilisé, on peut **animer** les spécifications pour générer automatiquement un **prototype** de l'application.
- Les spécifications formelles sont des objets mathématiques et peuvent être analysés avec des **méthodes mathématiques**
- Les spécifications peuvent être utilisées comme **guide** dans la construction des jeux de tests.

Elles sont pourtant **peu utilisées** pour ces autres **raisons** essentiellement d'ordre **économique :**

- Le management est fortement **conservateur**, il n'est pas encore prêt à adopter des techniques dont la rentabilité n'est pas encore prouvée. En effet le coût de développement d'un système formel est très cher et il n'est pas clair que ça réduise fortement le coût global du logiciel.
- La plupart des ingénieurs logiciels **manquent de formation** à ces techniques (essentiellement des mathématiques et de la logique).
- Les chefs de projets sont peu enclins à imposer une méthode qu'ils ne maîtrisent pas.
- Certaines **classes de problèmes** sont **peu facilement spécifiables** avec les méthodes actuelles (interfaces homme-machine, systèmes temps réel...)
- La plupart des professionnels pensent que peu de problèmes peuvent être spécifiés formellement. On a pourtant des exemples de gros systèmes spécifiés formellement (ex : la gestion de fichiers Unix, un environnement de programmation ...)
- La plupart des **efforts** dans ce domaine ont porté sur le développement de notations et de nouvelles techniques, **rarement sur des outils.** Ces outils sont pourtant tout à fait nécessaires pour pouvoir spécifier de gros systèmes.
- La plupart des chercheurs qui travaillent dans ce domaine ont une connaissance approximative du métier d'ingénieur logiciel et pensent naïvement que le développement de logiciel peut être ramené à la seule adoption de méthodes formelles.

Les deux classes de spécifications formelles

Les spécifications formelles reposent sur le principe de l'*approche transformationnelle.*

On évolue de la spécification vers le programme par une série de transformations minimes permettant de passer d'une description du problème à une autre description suffisamment proche de la précédente pour qu'il soit aisé de prouver que les deux descriptions sont équivalentes.

On arrive ainsi à prouver que le programme réalise effectivement les spécifications.

Il ne faut pas néanmoins se cacher la difficulté de trouver les bonnes transformations et de prouver leur équivalence.

Les spécifications formelles peuvent être réparties en 2 classes :

- Spécifications opérationnelles
  On s'intéresse à la description de l'enchaînement des actions.
  Les outils les plus utilisés sont : les automates, les systèmes de transition
  Le modèle le plus commun est la machine à transitions d'états, où l'on décrit les états, les transitions, les invariants d'états, les invariants de transitions
  On s'intéresse également à la description des aspects fonctionnels et propriétés : on s'appuie en général sur la logique de Hoare et le calcul de prédicats.

- Spécifications axiomatiques
  On décrit ici les propriétés que doivent satisfaire les implémentations
  exemple : spécifications algébriques (ASSPEGIC Université d'Orsay)

Les **langages de spécification** actuellement les plus connus empruntent aux deux classes.

Le langage **VDM** [JON 80] met en œuvre la logique de Hoare en l'enrichissant avec des types. Il semble que certains industriels soient enclins à l'adopter. (Bull) On peut regretter sa difficulté d'apprentissage du fait d'un grand nombre de symboles particuliers présents dans sa syntaxe.

Le **langage Z** [HAY 87] semble d'un abord plus facile du fait de son usage du graphisme dans la représentation des structures.

Le langage Larch introduit des mnémoniques qui facilitent la compréhension et permettent d'utiliser un clavier standard contrairement à VDM.

Introduction à la logique de Hoare

Dans un article de 69, Hoare voulant démontrer que l'informatique avait des fondements formels proposait la définition d'un programme à l'aide de pré et post conditions.

Cet article revêt une grande importance historique alors même que le but recherché était tout autre.

Définition

{P} S {Q}

S est un Programme
P est un ensemble de pré conditions :Prédicat logique à vérifier avant toute exécution
Q est un ensemble de post conditions : Prédicat logique à vérifier après toute exécution

Ces prédicats sont exprimés à l'aide de quantificateurs

Il est à noter que dans son article de 69, Hoare ne propose
pas de langage
pas de structuration
pas de types
pas de fonctions

Démarche de spécification

Etablir le domaine de variation des paramètres d'entrée
Spécifier cette contrainte comme un prédicat
Spécifier un prédicat définissant une condition sur les sorties en cas de comportement correct
Etablir les changements des paramètres d'entrée et les spécifier (différence entre fonction mathématique et informatique, passage par référence)
Combiner ces contraintes en pré et post conditions

Exemple

Soit une **fonction cherche** qui prend en **entrée**
un **tableau d'entiers X**
une **clé de recherche Cle**
La fonction **retourne**
L'index dans le tableau de l'élément égal à la clé.
Le tableau d'entrée est **inchangé.**

```
function Cherche (X : in TABLEAU_D_ENTIERS, Cle : INTEGER) return INTEGER ;

Pre :   Il existe I in X'FIRST..X'LAST tel que X (I) = Cle

Post :          X" (Cherche(X, Cle)) = Cle and X" = X

Erreur :        Cherche (X, Cle) = X'LAST+1
```

6.2. SPECIFICATIONS OPERATIONNELLES

On distingue essentiellement deux langages représentatifs Z et VDM
Nous étudierons les principes de VDM sans entrer trop en détail dans la syntaxe.

Types
simples : entiers, réels énumérés
ensembles
enregistrements
listes
maps

Opérations
sur les ensembles
sur les listes
sur les maps :

domain, range, superposition, restrictions

exemple

composition = map couleur to R
composition1 : composition
composition1= {bleu->0.5, vert ->0.5, rouge ->0.0}

dom composition1= {bleu, vert,rouge}
composition1 { rouge ->1.0}={bleu->0.5, vert ->0.5, rouge ->1.0}

composition1(bleu) =0.5

{bleu, vert} composition1 = {bleu->0.5, vert ->0.5}

composition1 {0.5} = {bleu vert}

Pour qu'une spécification soit cohérente chaque opération doit
avoir au moins un résultat possible
préserver les invariants

Une implémentation est soit
une *décomposition opérationnelle* prouvée en utilisant la logique de Hoare étendue
une *approche par les données* prouvée en utilisant des fonctions de retour.

La méthode utilise une construction pas à pas

EXEMPLE
Portefeuille d'actions boursières
commencer avec portefeuille vide
acheter et vendre N actions de la société X
donner le nombre d'actions d'une société.

Spécification en VDM

1 Construire les données

```
Portefeuille : map Société to INTEGER
```

2 Opérations

```
Vide () P : Portefeuille
Post P = {}

Acheter (N : Positif, X : Société, P : Portefeuille)
    P' : Portefeuille
Post if X  dom P
    then P'=P {X -> P(X) + N}
    else P' =P {X ->N}

Vendre (N : Positif, X : Société, P : Portefeuille)
    P' : Portefeuille
```

```
Pre X dom P et P(X) >= N
Post P'=P {X -> P(X)-N}

Nombre (X : Société, P : Portefeuille) N : INTEGER
Post if X dom P
    then N=P(X)
    else N=0
```

Les idées développées dans les langages VDM et Z et plus généralement dans les méthodes de spécifications opérationnelles sont reprises aujourd'hui dans des méthodes semi-formelles. En particulier les notions de prédicats d'entrée et de sortie sont reprises dans le langage EIFFEL et deviennent des assertions. La méthode orientée objet OMT que nous verrons plus loin préconise également de définir les conditions d'invocation d'un module sous forme d'assertion d'entrée, de même les conditions de sortie sont décrites dans un prédicat de terminaison.

6.3. SPECIFICATIONS ALGEBRIQUES

Idées de base :
- spécifier à partir des propriétés
- structurer en modules indépendants réutilisables

Il existe de nombreux langages de spécifications algébriques (OBJ [FUT 85] , LARCH [GUT 85] ...). Ces langages sont en général peu utilisés dans l'industrie. Il faut toutefois mentionner que des langages s'inspirent de la théorie des types abstraits algébriques sans l'implémenter complètement. Ceci facilite l'écriture de programmes "propres" c'est à dire lisibles, vérifiables et réutilisables. Nous citerons en particulier Eiffel (B Meyer) et dans une moindre mesure C++ et Ada.

Concepts :
- modules
- héritage
- importation
- généricité

FORMAT D'UNE SPECIFICATION ALGEBRIQUE
- -partie syntaxique
  - types
  - fonctions
- -partie sémantique
  - propriétés liant les différentes opérations
  - ces axiomes définissent implicitement le type de données

SYNTAXE GENERALE

sort <nom>

imports <Liste de noms de spécifications>

Description informelle du type et de ses opérations

Signature des opérations décrivant le nom et le type des paramètres des opérations définies pour le type

Axiomes définissant les opérations sur le type.

Exemple : Le type abstrait Pile

Type Pile
Sorts Pile

Constructors
Vide-> Pile
Empiler :(Element, Pile) -> Pile

Operators
Depiler : Pile->Pile
Sommet :Pile ->Element

Variables
N : Element
P : Pile

Equations
1 **:** Depiler (Empiler (N,P)) == P
2 : Sommet (Empiler (N,P)) == N

End Pile

A noter : On distingue les **constructeurs** des autre opérations. :ce sont les opérations retournant le type de base(pile), telles que n'importe quelle pile peut être construite en les utilisant.

Exemple 2 : gestion de portefeuille

Type Portefeuille
with Société
Sorts Portefeuille

Constructors
Vide : ->Portefeuille
Acheter : (Société,Positif,Portefeuille) -> Portefeuille
Nombre : (Société, Portefeuille) ->Integer

Variables
PF : Portefeuille
N, N1, N2 : Positif
S,S1, S2 : Société

Equations
Nombre (S,Vide) = 0
Nombre (S1, Acheter (S2,N,PF)) == N + Nombre (S1,PF) if S1=S2
Nombre (S1, Acheter (S2,N,PF)) == Nombre (S1,PF) if S1/=S2

Vendre (S1, N1,Acheter (S2,N2,PF))==
Acheter (S2,N2, Vendre (S1,N1,PF)) if S1/=S2
Vendre (S1, N1,Acheter (S2,N2,PF))==
Acheter (S1,N2-N1, PF) if S1=S2 and N1<N2
Vendre (S1, N1,Acheter (S2,N2,PF))== PF
if S1=S2 and N1 = N2

```
Vendre ( S1, N1,Acheter (S2,N2,PF))==
            Vendre (S1,N1-N2,PF)) if S1=S2 and N1>N2

Acheter (S1,N1,Acheter (S2,N2,PF))==
            Acheter (S2,N2,Acheter (S1,N1,PF)) if S1/=S2
Acheter (S1,N1,Acheter (S2,N2,PF))==
            Acheter (S1,N1+N2,PF) if S1=S2

End Portefeuille
```

Compléments

Il est possible de définir un type abstrait à partir d'un autre en **l'enrichissant** de nouvelles opérations **: héritage**

Les erreurs dans les opérations peuvent être spécifiées en rajoutant un symbole *indéfini*

Les spécifications algébriques sont recommandées en cas d'approche objet.

On note toutefois une difficulté : *Comment est on sur d'avoir toutes les propriétés?*

6.4. COMPARAISON SPECIFICATIONS ALGEBRIQUES/ VDM - Z

Inconvénients de VDM

-Les données ne sont pas abstraites

un même concept : 2 représentations

P1 = {X1 ->0} P2= {}

-Une donnée n'est pas associée à ses opérations

Avantages de VDM

- Spécification d'avantage intuitive

Inconvénients des types abstraits

- A-t-on toutes les propriétés?

Actuellement VDM et Z sont plus répandus que les langages implémentant la notion de types abstraits au sens strict du terme.

6.5 BIBLIOGRAPHIE

JF Monin
Comprendre les Méthodes Formelles, Panorama des outils et méthodes
Masson , 1996

7. METHODES D'ANALYSE ET CONCEPTION DE LOGICIEL

Les méthodes d'analyse et de conception structurées ont constitué une première réponse à la crise du logiciel. Apparues dans les années 70 elles sont fortement pratiquées depuis le début des années 80 et le seront encore vraisemblablement pendant quelques années encore.

Elles reposent sur une approche **hiérarchique** du problème basée sur les **données ou les traitements**.

Ces méthodes s'appuient largement sur des **langages graphiques** plus ou moins standardisés. Elles permettent de gagner en **précision** et constituent un bon **langage de communication** entre les différents membres d'une équipe. que ce soit entre eux ou dans les discussions avec le client. Elles ne nécessitent **pas de connaissances particulières**, mais leur pouvoir d'expression est limité et on ne peut parler de rigueur au sens mathématique du terme.

Les outils mettant en œuvre ces méthodes permettent souvent de **dériver semi automatiquement** une **architecture globale** à partir des spécifications. Ils assurent en général la **cohérence** des spécifications et aident à vérifier leur **complétude**.

7.1. INTRODUCTION

7.1.1. Définition

La phase de conception suit immédiatement la phase d'analyse. Elle est prise en charge par l'équipe de développement. Elle vise à définir une architecture *modulaire* du logiciel ou encore une décomposition en *modules* qui facilitera la maintenance et permettra le développement parallèle par différents programmeurs.

On regroupe en général sous le terme conception deux étapes distinctes :

- **Conception globale** : elle permet de définir les choix fondamentaux de l'architecture du logiciel en liaison avec l'architecture matérielle.
- **Conception détaillée :** elle décrit précisément chaque module, les algorithmes mis en œuvre et les traitements effectués en cas d'erreur. La documentation produite sera très utile en phase de maintenance. La phase de conception détaillée n'est pas traitée dans ce cours. Elle a souvent tendance aujourd'hui à être escamotée car la taille des modules étant réduite il n'est souvent pas nécessaire de décrire précisément les algorithmes mis en jeu.

La phase de conception globale se termine avec la rédaction du **document de conception globale** et du **plan d'intégration** qui ne sera pas traité ici.

7.1.2. Influence de la conception sur la maintenance et la qualité du logiciel

La conception globale du logiciel aura une importance considérable sur sa **qualité.** Rappelons quelques facteurs de qualité

- **validité** : aptitude du produit à réaliser les tâches spécifiées
- **robustesse :** aptitude du produit à fonctionner dans des conditions anormales

- **extensibilité :**	aptitude du produit à s'adapter aux changements de spécification
- **réutilisabilité** :	aptitude du produit à resservir dans de nouvelles applications
- **compatibilité :**	aptitude des logiciels à être combinés ensemble
- **efficacité**	bonne utilisation des ressources
- **portabilité :**	facilité d'adaptation à différents environnements
- **vérifiabilité**	facilité de validation
- **intégrité**	aptitude du logiciel à protéger ses composants contre des modifications non autorisées
- **convivialité**	facilité avec laquelle les utilisateurs font l'apprentissage du produit

Il est à noter que bien souvent ces facteurs de qualité s'excluent mutuellement et que par conséquent des compromis sont souvent nécessaires lors de la conception d'un logiciel. La conception globale du logiciel aura une importance considérable sur sa **réalisation.** La bonne répartition du travail dans les phases de conception détaillée et codage passe par une conception globale précise et efficace.

La **facilité de maintenance** du produit dépendra de sa conception globale. Trop souvent l'activité de maintenance n'est pas suffisamment prise en compte au moment de la réalisation d'un produit ; cette activité en regroupe en fait au moins deux : adaptation à des besoins nouveaux et correction des erreurs. Une analyse des coûts de maintenance fait ressortir les chiffres suivants :

modifications dans les exigences des utilisateurs :	42%
modification du format des données	18%
correction d'erreurs en urgence	13%
correction d'erreurs de routine	9%
modification du matériel	6%
documentation	5%
amélioration de l'efficacité	4%
autres	3%

On voit clairement que le coût le plus important provient des modifications de spécifications ou du format de données (ce qui est en somme la même chose). La conception globale devra permettre d'effectuer ces modifications de la façon la plus simple qui soit et doit mettre en évidence des éléments permettant la traçabilité des spécifications exprimées lors de l'analyse des besoins.

7.1.3. Modularité

7.1.3.1. Définitions

Nous définissons ici les notions de *modules et modularité* vues précédemment et qui vont présider à la phase de conception globale. Ces notions sont à l'origine des méthodes de conception structurées et plus tard de l'approche objets.

Un module peut être vu comme *un bloc de code pouvant être appelé*. Cette définition permet de considérer une *procédure ou fonction* comme un module ainsi qu'un perform COBOL bien que celui-ci n'ait pas de variables propres. Cette définition n'est pas suffisante car elle exclut par exemple un include de déclarations en C ou C++ qui n'est pas appelé au plein sens du terme.

Yourdon élargit la définition d'un module en 1979 : "*toute portion de code entre deux éléments servant d'encadrement (begin ...end, (), {}, ...) et pouvant être nommé.*"

Cette définition permet d'inclure l'ensemble des éléments que tout programmeur est en droit de considérer comme un module. On peut assimiler modularité et mécanisme d'abstraction de programmation que nous décrivons un peu plus loin.

Toutefois notons que pour que la modularité soit bien choisie elle doit répondre à certaines règles, et en particulier être adéquate (correspondre aux besoins), assurer une forte cohésion à l'intérieur de chaque module et un couplage faible entre les modules. Ce sont ces qualités qui permettront extensibilité et réutilisation. Le bon sens nous donne quelques indicateurs sur une bonne abstraction ; elle est forcément

7.1.3.2. Critères de modularité

La décomposabilité modulaire permet de décomposer un problème complexe en problèmes plus petits solubles isolément . Ce procédé est souvent récursif (redécomposition des sous systèmes)

exemple : conception descendante

contre exemple : un module d'initialisation (aller retours permanents avec les autres modules)

La composabilité modulaire favorise la production indépendante de modules pouvant être combinés dans des contextes différents de ceux dans lesquels ils ont été initialement développés. On utilise ici des éléments existant pour faire un nouveau système au lieu de partir des spécifications comme pour la décomposabilité. Ce principe est directement lié à la réutilisabilité

exemple : bibliothèque de sous programmes, langage shell

contre exemple : préprocesseurs (extensions d'un langage incompatibles les unes avec les autres.)

Les deux critères précédents sont indépendants et s'opposent souvent entre eux.

La compréhensibilité modulaire, très utile en maintenance, la compréhension d'un module n'exige pas celle des autres.

contre exemple : dépendances séquentielles, si un ensemble de modules a été conçu de façon à fonctionner dans un certain ordre, ils seront difficiles à comprendre individuellement.

La continuité modulaire est assurée si des petites modifications de spécifications n'amènent pas à revoir l'ensemble de l'architecture. Les modifications ne doivent affecter que quelques modules. Ce critère est lié à l'extensibilité.

exemple : constantes symboliques, référence uniforme (stockage ou calcul, notation insensible à ce critère dans le langage d'implémentation)

contre exemples : utilisation de représentations physiques des données, tableaux statiques

La protection modulaire est assurée si une condition anormale se produisant pendant l'exécution d'un module reste localisée à ce module ou bien ne se propage qu'à quelques modules voisins.

exemple : validation des entrées à partir de la source

contre exemple : exceptions indisciplinées (séparation des erreurs du lieu de traitement)

7.1.3.3. Principes de modularité

A partir des critères précédemment définis, on peut déduire cinq principes à respecter.

Unités modulaires linguistiques : les modules doivent correspondre à des unités syntaxiques du langage que ce soit un langage de programmation de conception ou de spécification. Les modules doivent être compilables séparément.

Ceci permet de vérifier la décomposabilité, la composabilité, la protection. Ceci implique la prise en compte du langage de programmation dès la conception.

Exemples :

types abstraits : ce sont des objets complexes qui représentent à la fois des données et des opérations (packages ADA, classes C++...)

Contre exemples :

- sous-programmes en Pascal, Fortran, Assembleur qui représentent des actions mais ne sont pas compilables séparément
- types en Pascal, C... qui représentent des structures de données mais ne sont pas compilables séparément contrairement à ADA où l'on peut compiler seulement des spécifications.

Minimisation des interfaces : Tout module doit communiquer avec aussi peu d'autres que possible. Si un système est composé de n modules, le nombre d'interconnexions doit être plus près du minimum (n-1) que du maximum (n * n-1/2)

Ce principe permet de vérifier les règles de protection et de continuité.

Simplification des interfaces : Les modules qui communiquent doivent échanger aussi peu de données que possible. Ceci permet d'assurer les critères de continuité et de protection.

contre exemple : les common en fortran, variables globales au niveau le plus externe, portée des identificateurs dans les langages à structure de blocs où tout bloc a accès à des données déclarées dans les niveaux supérieurs dont toutes ne sont pas indispensables.

Interfaces explicites : lorsque deux modules communiquent, ceci doit se voir clairement dans le texte de l'un et/ou de l'autre. Ceci permet de satisfaire les critères de composabilité et décomposabilité.

contre exemple : les modules qui communiquent par partage de données sans communication apparente (comme par exemple un appel de procédure).

Masquage de l'information **:** toute information d'un module est privée sauf l'interface. Ceci afin de vérifier le critère de continuité : le contenu d'un module (calculs...) peut changer sans que ses appels soient changés.

Exemple Recherche dans une table ne tenant aucun compte du mode de stockage (séquentiel, hash code, triée...) mais seulement de la nature de la clé de recherche.

Nous examinons ici la notion de modularité pour analyser les méthodes de construction de logiciel ; nous essayons en particulier de définir ce que peut être un module en pratique. On a coutume de l'associer à un sous-programme. Nous verrons que cette notion est peu satisfaisante et nous définirons des formes plus élaborées de modules.

7.1.4. Mécanismes d'abstraction

Les langages de programmation classiques fournissent des mécanismes d'abstraction favorisant la modularité. On trouvera ci-après quelques exemples.

7.1.4.1. Types abstraits : masquage de l'information

Dans l'exemple ci-dessous, la représentation de la file d'attente n'est pas rendue visible aux utilisateurs du package. Il est notamment impossible d'accéder à des éléments intermédiaires de la pile, la représentation interne de la pile est sans influence sur l'utilisation du package. Ceci a un impact sur la maintenance puisque

- il est possible de changer facilement de représentation
- il n'est pas nécessaire de recompiler les modules utilisant le package en cas de changement de celui ci.

Un type abstrait permet de créer n objets similaires.

Exemple : structure de données abstraites

```
package FILE_D_ATTENTE is

        X_VIDE : exception ;
        procedure INSERER (I : INTEGER) ;
        procedure RETIRER (I : out INTEGER) ;
        function EST_VIDE return BOOLEAN ;

end FILE_D_ATTENTE ;
```

-- Que l'on utilise de la façon suivante

```
FILE_D_ATTENTE.INSERER (X) ;
```

```
package FILES_D_ATTENTE is
        type FILE_D_ATTENTE is private ;
        X_VIDE : exception ;

        procedure INSERER
                (    I : INTEGER ;
                     DANS : in out FILE_D_ATTENTE) ;

        procedure RETIRER
                (    I : out INTEGER ;
                       DE : in out FILE_D_ATTENTE) ;

        function EST_VIDE (F : FILE_D_ATTENTE)
                        return BOOLEAN ;

end FILES_D_ATTENTE ;
```

7.1.4.2. Activités abstraites

De même, des activités abstraites constituées d'actions complexes pourront être représentées par des tâches en ADA.

Exemples

process utilisateur dans un système d'exploitation
contrôle des moteurs dans un avion

Les tâches permettent de

créer n activités identiques
décrire une activité indépendamment des autres
nommer une activité globalement

7.1.4.3. Objets actifs/passifs

Les objets actifs sont des objets autonomes qui évoluent indépendamment du reste du système.

Par contre, un serveur est un objet passif, il définit un service c'est à dire un ensemble d'opérations qui peut être utilisé par plusieurs agents.

7.1.4.4. Exemple SERVEUR (en Ada)

```
package RÉSEAU is
        type PAQUET is private ;
        type STATION is private ;
        type UTILISATEUR is private ;

        procedure ENVOYER_PAQUET
              (  P : PAQUET ;
                        S : STATION,
        U : UTILISATEUR) ;


        procedure RECEVOIR_PAQUET
              (  P : out PAQUET ;
                        TIME_OUT : DURATION) ;
        ...
    end RÉSEAU ;
    package body RESEAU is
        task ENVOYEUR is
            entry ENVOYER (...) ;
            entry STATUS (...) ;
        end ENVOYEUR ;

        task RECEVEUR is
            entry RECEVOIR (...) ;
        end RECEVEUR ;

        procedure ENVOYER_PAQUET(...) is
        begin
            ...ENVOYEUR.ENVOYER(...) ; ...
        end ENVOYER_PAQUET ;

        procedure RECEVOIR_PAQUET(...) is
        begin
            ...RECEVEUR.RECEVOIR(...) ;...
        end RECEVOIR_PAQUET ;
    end RESEAU ;
```

7.1.5. Conclusion

En conclusion nous dirons qu'une bonne abstraction est définie avec précision, elle renforce la cohésion au sein de chaque module, elle minimise les interfaces.

Elle peut être implémentée par des mécanismes d'abstraction dont les types abstraits sont un exemple. Nous reviendrons sur d'autres mécanismes dans le chapitre consacré à la conception orientée objets.

7.2. APPROCHE FONCTIONNELLE DESCENDANTE

L'approche fonctionnelle descendante est un moyen de dériver une architecture modulaire du logiciel à partir d'un diagramme de flots de données.

A partir d'un diagramme de flots de données on dérive la structure du programme selon une méthode décrite par Yourdon et Constantine. Le système est vu comme un ensemble de fonctions. La méthode consiste à :

-Concevoir les fonctions de haut niveau

-Affiner jusqu'à obtenir une conception suffisamment détaillée

L'exemple ci-dessous décrit comment à partir d'un diagramme de flots de données dériver un modèle d'architecture.

7.2.1. Exemple : Vérificateur d'orthographe :(d'après I. Somerville)

Ce programme recherche chaque mot d'un document dans un dictionnaire. Si le mot apparaît dans le dictionnaire, il n'y a pas d'autre traitement à réaliser. Sinon, le mot est affiché. Si l'utilisateur décide que le mot est correctement orthographié, il est entré dans le dictionnaire, sinon il est entré dans la liste des mots mal orthographiés.

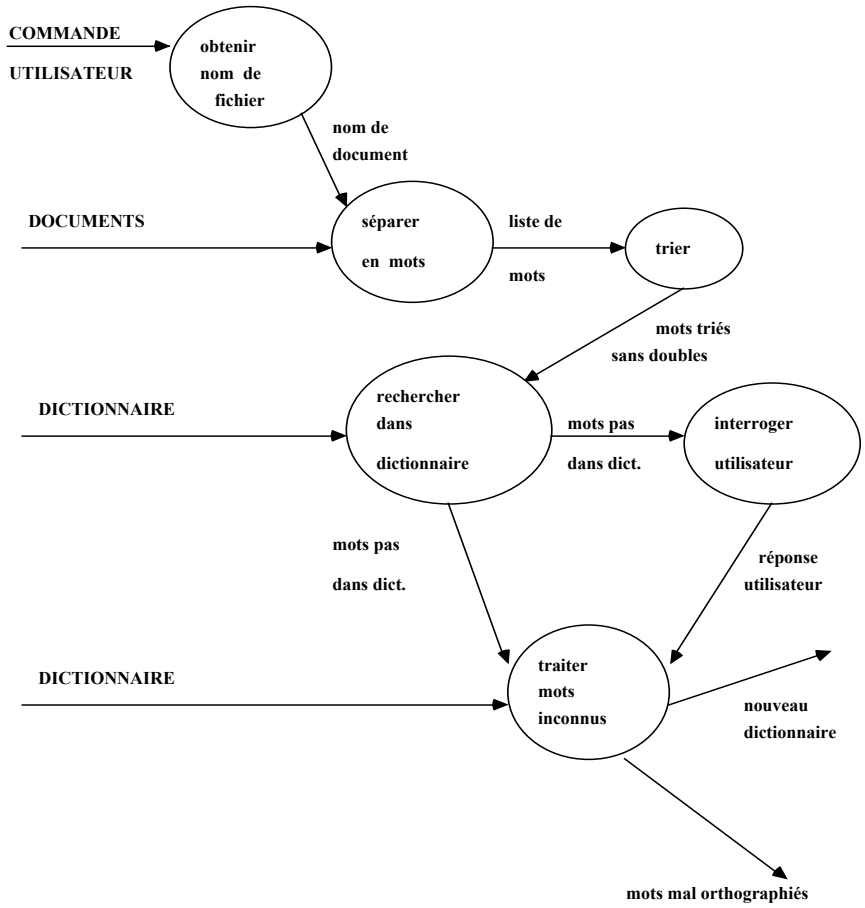
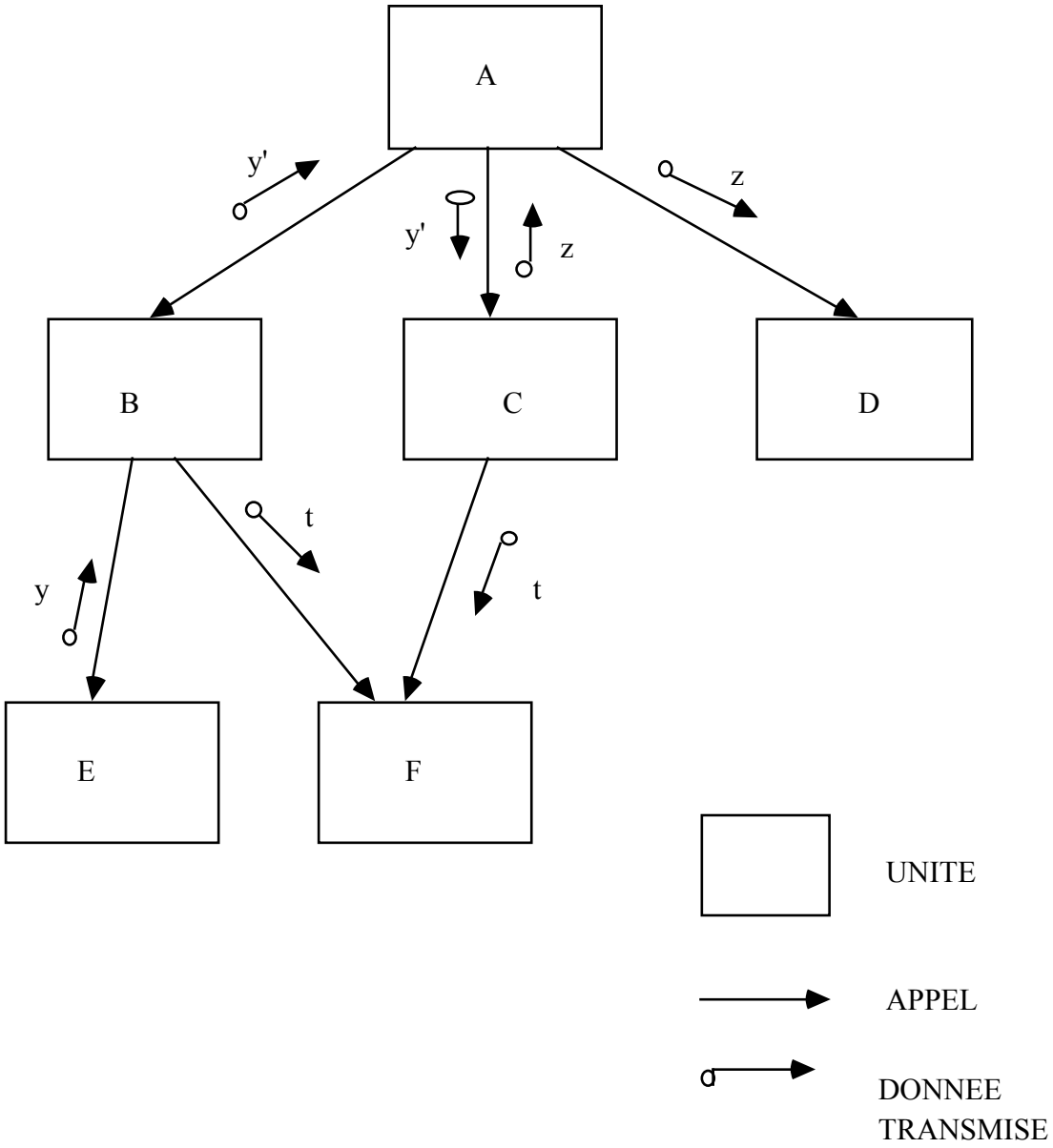


DIAGRAMME DE FLOT DES DONNEES

7.2.2 Diagramme structuré

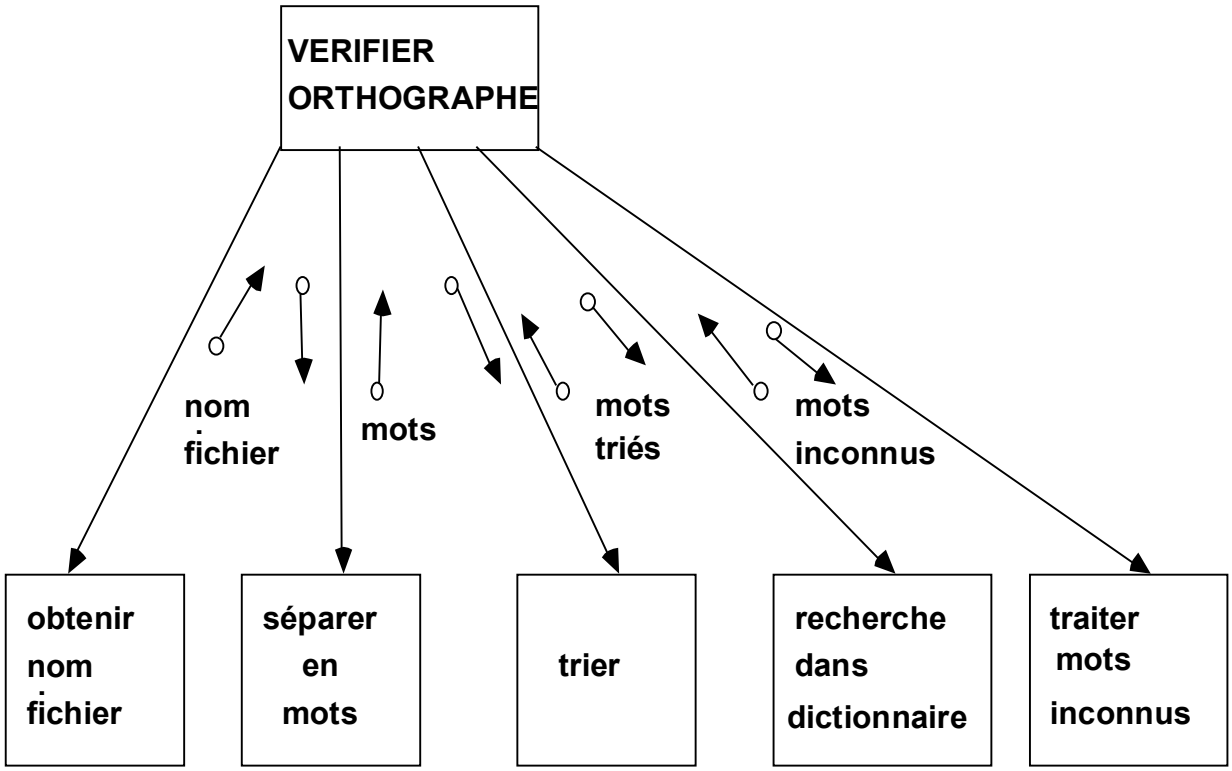
Un diagramme structuré constitue le graphe d'appel d'un logiciel. La méthode décrit comment les éléments d'un diagramme de flot de données peuvent être dérivés en une hiérarchie d'unités de programme (CONSTANTINE & YOURDON).



à partir d'un diagramme de flot,

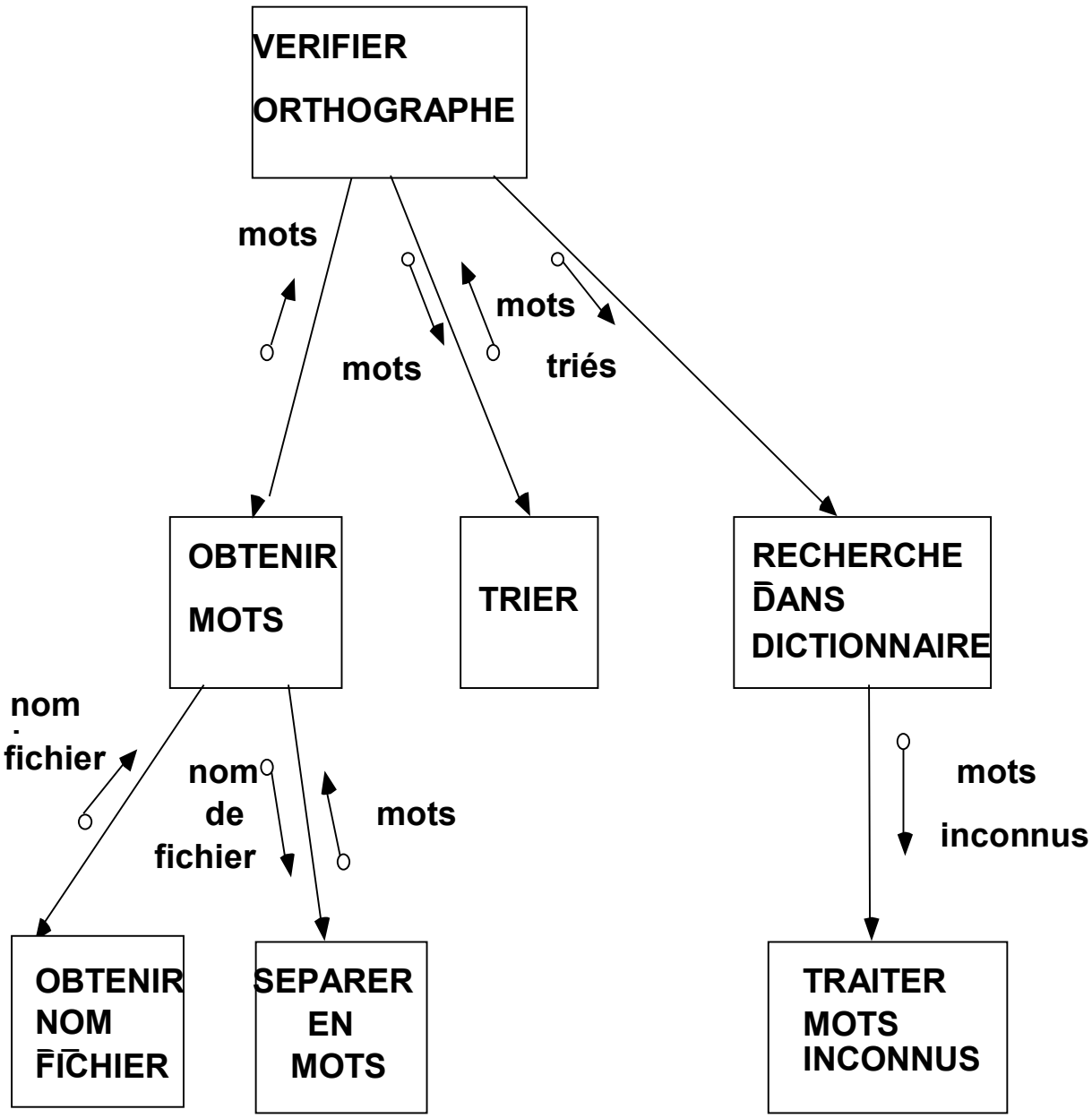
on peut construire plusieurs

diagrammes structurés.



VERIFICATEUR D'ORTHOGRAPHE

STRUCTURE N° 1



VERIFICATEUR D'ORTHOGRAPHE

STRUCTURE N° 2

comment dériver la structure la plus appropriee ?

7.3.3. Définitions

On classe les différentes unités en :

- Unités d'entrée, dites "afférentes" : acceptent des données d'un niveau plus bas, les transmettent vers un niveau plus haut.
- Unités de sortie, dites "efférentes" : acceptent des données d'un niveau plus haut, les transmettent vers un niveau plus bas.
- Unités de transformation acceptent des données d'un niveau plus haut, les transforment, les retransmettent vers un niveau plus haut.
- Unités de coordination : servent à contrôler d'autres unités.

7.2.4. Méthode de dérivation

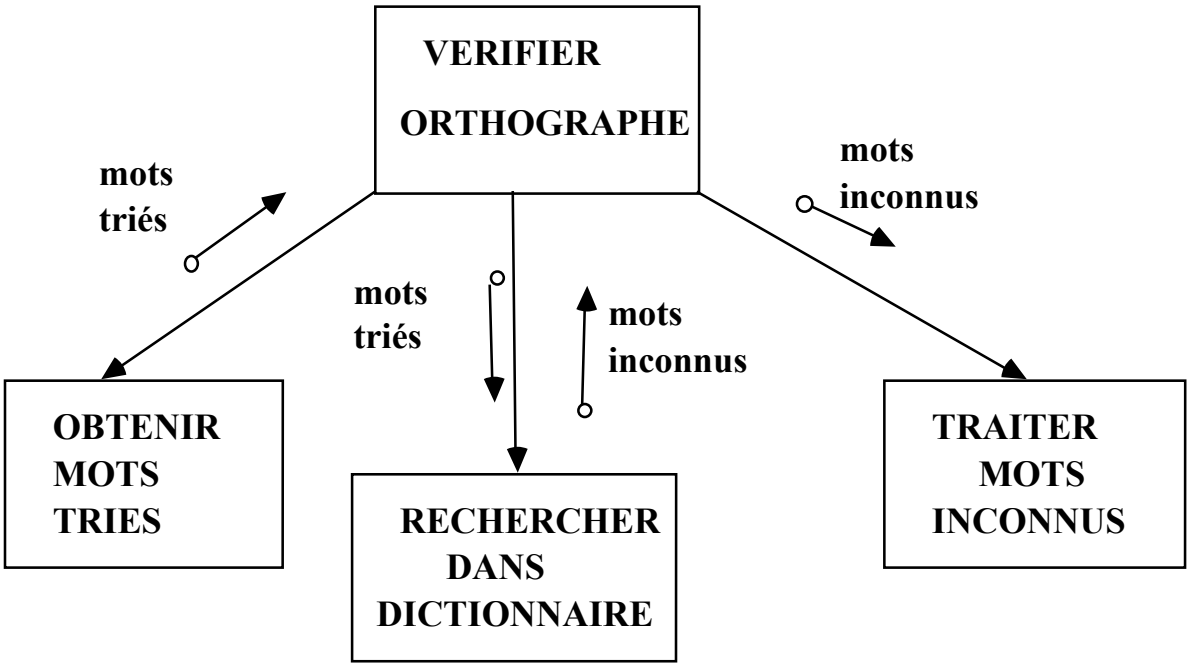
1. Identifier l'unité d'entrée de plus haut niveau :
   Tracer les entrées jusqu'à une bulle telle que sa sortie ne puisse être immédiatement dérivée de ses entrées.
   L'unité précédente est l'unité d'entrée de plus haut niveau.

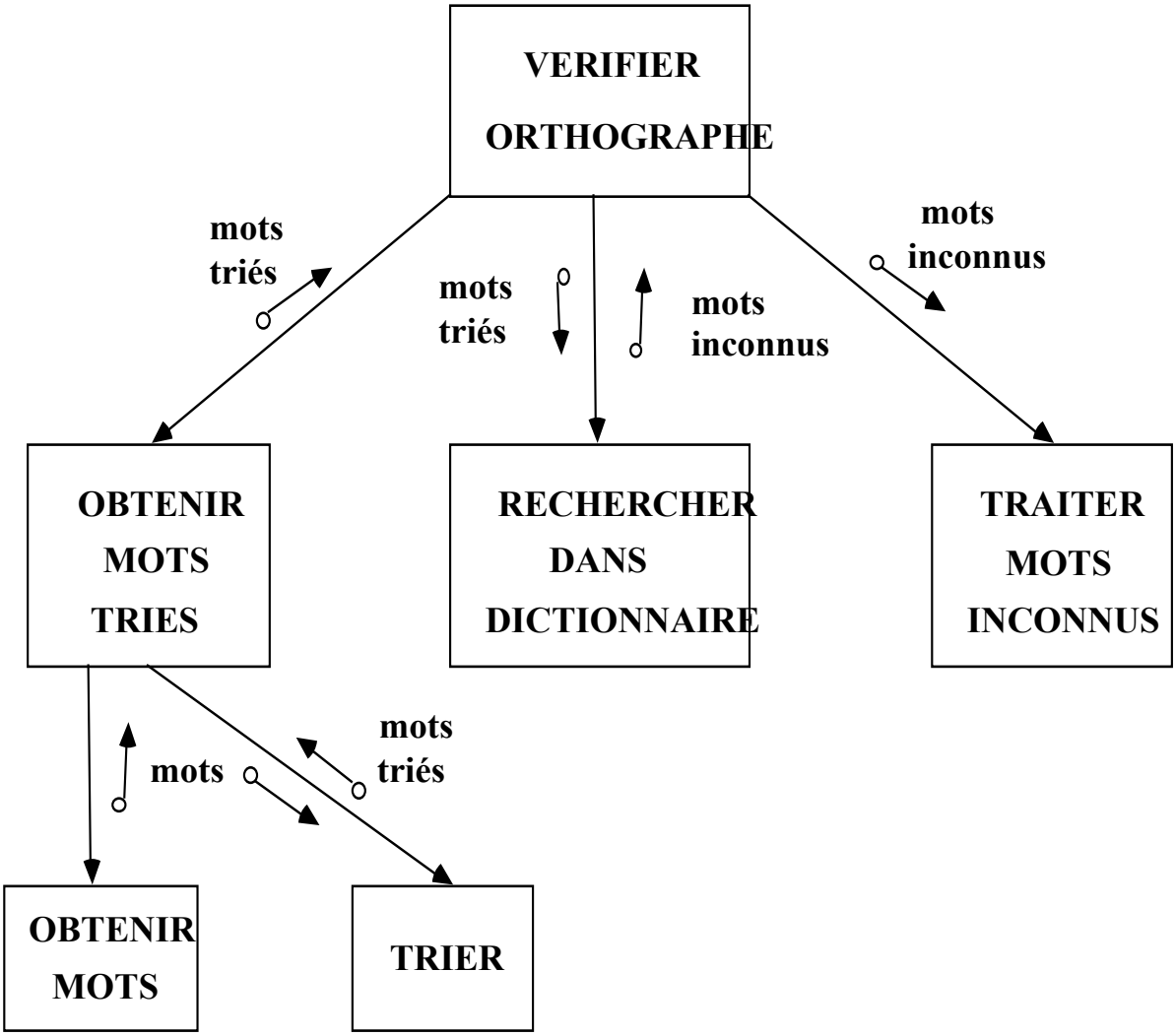
2 Identifier l'unité de sortie de plus haut niveau :
   Même méthode, à partir des sorties.

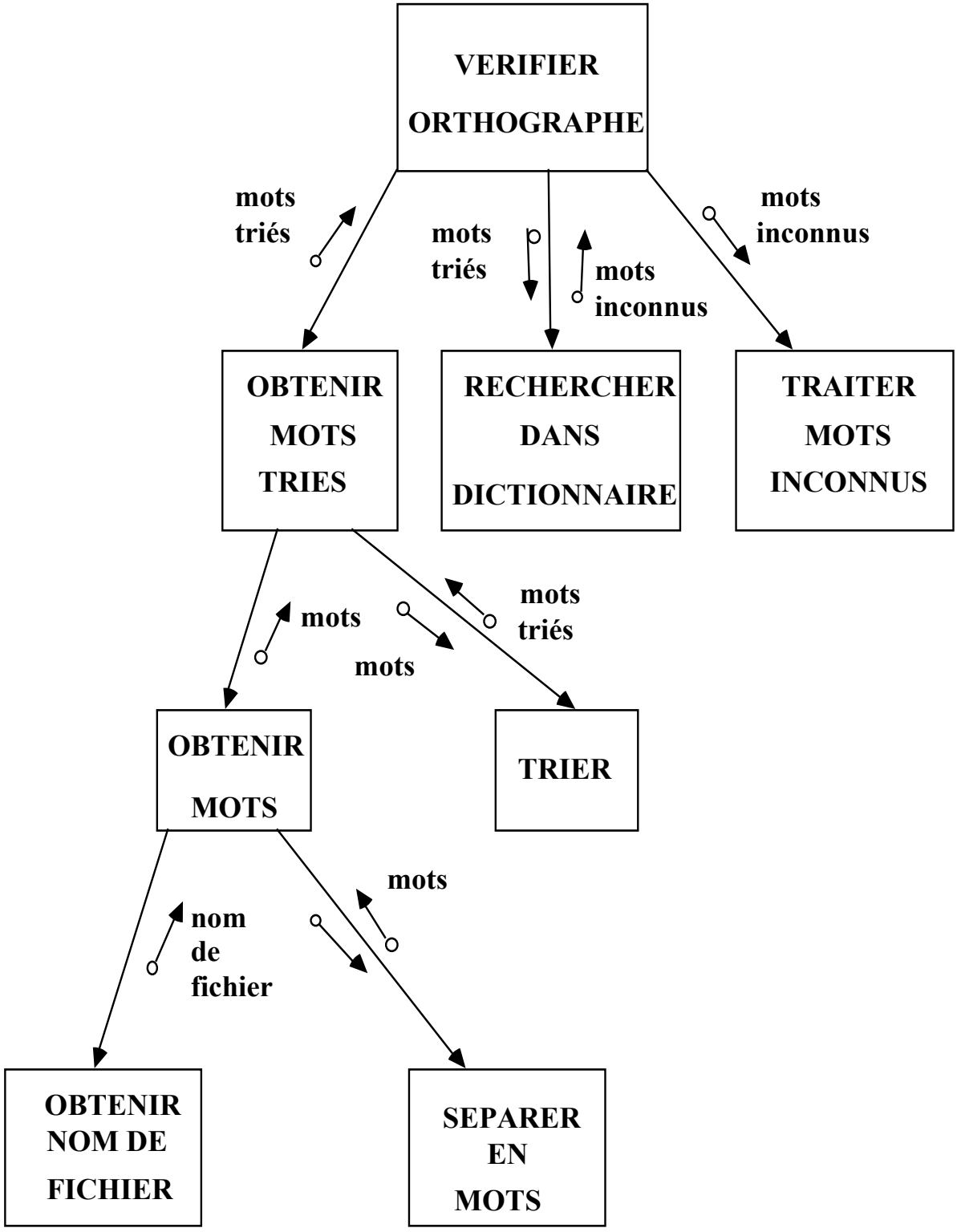
3 Les unités non visitées sont les unités de transformation de plus haut niveau.

4 Répéter ce processus récursivement à partir des unités de premier niveau.

Exemple : VÉRIFICATEUR D'ORTHOGRAPHE




VERIFIER
ORTHOGRAPHE
mots
triés
mots
triés
mots
inconnus
mots
inconnus
OBTENIR
MOTS
TRIES
RECHERCHER
DANS
DICTIONNAIRE
TRAITER
MOTS
INCONNUS
mots
mots
triés
OBTENIR
MOTS
TRIER


VERIFIER ORTHOGRAPHE
mots triés
mots triés
mots inconnus
mots inconnus
OBTENIR MOTS TRIES
RECHERCHER DANS DICTIONNAIRE
TRAITER MOTS INCONNUS
mots
mots
mots triés
OBTENIR MOTS
TRIER
nom de fichier
mots
OBTENIR NOM DE FICHIER
SEPARER EN MOTS

Une fois l'architecture globale décrite, les données des différents modules doivent être précisées afin que les différents programmeurs qui coderont les différents modules soient bien d'accord sur le type et l'ordre des paramètres à échanger.

Exemple

```
procedure VERIFIER_ORTHOGRAPHE is
   procedure OBTENIR_MOTS_TRIES
     (MOTS_TRIES :out LISTE_DE_MOTS)
     is separate ;

   procedure RECHERCHER_DANS_DICTIONNAIRE
   (MOTS_TRIES in LISTE_DE_MOTS ;
      MOTS_INCONNUS : out LISTE_DE_MOTS)
     is separate ;

   procedure TRAITER_MOTS_INCONNUS
     (MOTS_INCONNUS :in LISTE_DE_MOTS)
     is separate ;

   MOTS_TRIES, MOTS_INCONNUS : LISTE_DE_MOTS

begin
   OBTENIR        (MOTS_TRIES) ;
   RECHERCHER_DANS_DICTIONNAIRE (MOTS_TRIES, MOTS_INCONNUS) ;
   TRAITER_MOTS_INCONNUS (MOTS_INCONNUS) ;
end VERIFIER_ORTHOGRAPHE ;
```

7.2.5. Inconvénients de l'approche descendante

Cette approche
- ne permet pas de mettre en œuvre les types abstraits et le masquage de l'information
- ne permet pas de traiter de façon évidente les problèmes naturellement concurrents
- ne permet pas de répercuter simplement de petits changements du problème à résoudre

Il est difficile avec cette méthode
- d'isoler des sous systèmes communs
- de réutiliser des composants disponibles

Cependant c'est l'approche la plus utilisée dans l'industrie essentiellement pour des raisons historiques.

On peut utiliser quelques heuristiques pour améliorer le résultat obtenu par application stricte de la méthode.
- Réduire le couplage, améliorer la cohésion
- Minimiser les structures à grand fan out (par factorisation de modules)
- Garder l'impact de l'effet d'un module dans les modules sous contrôle de celui ci

Simplifier les interfaces entre modules

ex :liste de données apparemment non corrélées passées en paramètre indique un manque de cohésion du module
- Définir des modules à comportement prévisible
  ex : pas de mémorisation interne
- Éviter les points d'entrée et sortie pathologiques dans un module
  ex : ENTRY de fortran

7.2.6 Outils mettant en œuvre la méthode :

STP, TeamWork, IEW, ASA...

7.3. APPROCHE DIRIGEE PAR LES DONNEES

La méthodes d'analyse de Jackson et celle de Warnier et Orr débouchent sur des méthodes de conception que nous ne décrivons pas ici.

Méthode :
- Décrire la structure des données en entrée et en sortie
- En déduire la structure des modules

Caractéristiques de la méthode :
- Utilisée principalement en gestion

Ne sera pas décrite ici car n'est plus guère utilisée, il est toutefois à noter que de nombreux programmes aujourd'hui remis en cause par les problèmes Euro/an 2000 ont été conçus en utilisant cette méthode.

7.4 RETOUR SUR LA METHODE MERISE (MCD)

La méthode Merise, vue au chapitre 1 est essentiellement utilisée pour sa partie concernant la modélisation des données au niveau conceptuel.

Le modèle utilisé est très apparenté au **modèle entité-relation** de Chen même si le vocabulaire diffère légèrement (le paragraphe précédent donne les deux types de vocabulaire).

La méthode aide à définir les entités et les relations à partir de l'analyse du monde réel. Elle s'intéresse ensuite aux **dépendances** existant entre les entités ou classes d'entités et propose des règles de normalisation reprises de la théorie des bases de données relationnelles.

La règle d'**énumération** impose qu'une valeur de l'identifiant détermine une et une seule valeur pour chaque attribut (on sait distinguer une entité d'une autre à la seule vue de son identifiant)

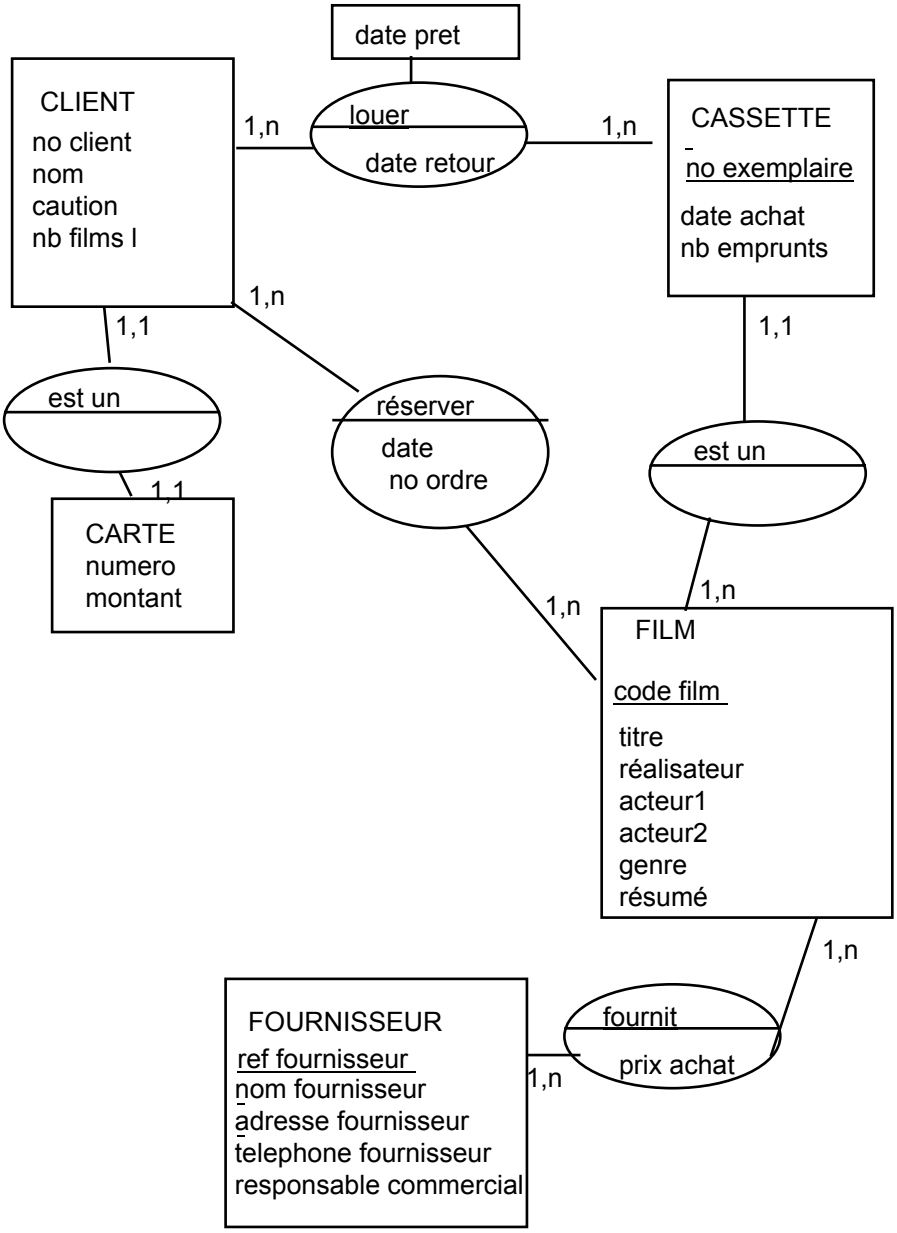
La règle de **dépendance directe** impose que toutes les propriétés dépendent directement de l'identifiant

exemple : le titre d'un film est un attribut de l'entité film et non pas de l'entité cassette.

La règle de **pleine dépendance** s'applique aux relations, elle impose que toutes les propriétés d'une relation dépendent de la totalité des entités que cette relation associe.

exemple :la date de retour est un attribut de la relation louer car elle dépend à la fois de l'entité cassette , du client et de la date de prêt

Exemple : Modélisation des données dans la gestion des prêts d'un vidéo club



La partie modélisation des traitements (MCT) de la méthode Merise repose sur une approche Réseaux de Pétri et n'est pas aussi utilisée que la partie MCD. Nous ne la traiterons pas dans ce cours.

Références

7.5 REFERENCES :

L CONSTANTINE & E.YOURDON,
Structured Design,
Prentice-Hall, 1979

Peter CHEN
The entity relationship model
ACM Transaction on Data Base System 1976

HATLEY, PIRBAI
Stratégies de spécification de systèmes temps réel, SA/RT
Masson 1991

WARD & MELLOR
Structured Development for real time systems
Prentice Hall 1985

Didier Banos et Guy Malbosc
Merise Pratique
Eyrolles 1990

M. Bouzeghoub, G Gardarin, P Valduriez
De C++ à Merise Objets
Eyrolles 1994