CONCEPTION DES LOGICIELS

TOME 1: LES OUTILS DE LA CONCEPTION

Auteur: Bernard GIACOMONI

Version	Date	Objet
1.0	18/11/2021	Version initiale

Table des matières

I.INTRODUCTION:

I.1.OBJECTIF VISÉ:

L'objectif de l'ouvrage (dont le document présent constitue le premier tome) n'est en aucun cas d'initier le lecteur à une méthode de conception ou à un formalisme particulier, mais plutôt de le familiariser à la problématique générale et aux enjeux de l'activité de conception des logiciels.

Dans ce cadre, les différents aspects de cette activité et les concepts et techniques qui lui sont rattachés sont présentés et expliqués en s'efforçant de faire ressortir les similitudes et différences entre les différentes méthodes afin d'en tirer des enseignements généraux.

I.2.LES BESOINS DU DÉVELOPPEUR DE LOGICIELS:

I.2.1.CANALISER LA CRÉATIVITÉ SANS LA BRIDER:

D'une part, la conception des logiciels, comme toutes les activités de conception, s'accorde assez mal avec des contraintes méthodologiques trop fortes: les méthodes doivent apporter un soutien à la créativité sans l'étouffer sous des formalismes trop lourds à gérer. L'important n'est pas d'appliquer les règles à la lettre, mais de comprendre pourquoi ces règles ont été instaurées, ce que leur respect peut apporter ou garantir et les danger qui peuvent surgir lorsque l'on s'en écarte trop.

I.2.2.S'ADAPTER A DES CADRES MÉTHODOLOGIQUES DIFFÉRENTS:

D'autre part, un développeur de logiciel peut être amené à exercer dans différents contextes et dans différents cadres méthodologiques. Il doit donc pouvoir s'adapter à chacun de ces environnement avec un minimum d'efforts et dans un temps limité. De ce fait, la compréhension des différentes démarches (la MÉTHODOLOGIE de l'activité de conception des logiciels) lui est donc plus précieuse que la maîtrise de telle ou telle MÉTHODE ou formalisme particulier.

I.3.CONCEPTION ET MÉTHODES AGILES:

Certaines méthodes de développement actuelles ("méthodes agiles") contestent l'utilité d'une phase de conception effectuée EN PRÉLIMINAIRE aux activités de développement de chaque lot de réalisation, pour s'en remettre à une adaptation permanente de l'architecture au fur et à mesure de l'évolution du projet. Cette pratique n'aboutit pas à la disparition de l'activité de conception, mais plutôt à sa fragmentation en plusieurs itérations réparties tout au long du projet.

Dans ces contextes de travail, les compétences en matière de conception et d'architecture globale des logiciels sont d'autant plus précieuses qu'elles permettent d'éviter que ces

différentes étapes d'adaptations n'aboutissent à des architectures incohérentes et faiblement performantes.

I.4.REMARQUES SUR LE CONTENU ET LE PLAN DE L'OUVRAGE:

- Ce document n'est pas un ouvrage d'initiation. Il s'adresse à des lecteurs ayant une expérience du développement des logiciels (au minimum, une pratique de la programmation) ainsi qu'une connaissance au moins basique de la structure et du fonctionnement des Unités de Traitement Informatiques. Une expérience de la conduite de projets informatiques est également bienvenue. Cependant, le chapitre II effectue de nombreux rappels qui peuvent aider à une "mise (ou remise) à niveau" dans ces domaines;
- Si des formalismes existants sont parfois utilisés dans cet ouvrage pour expliquer ou illustrer les différents concepts, ces formalismes peuvent être empruntés à des méthodes différentes ou même à aucune méthode existante: le but est plus de faire comprendre au lecteur les tenants et les aboutissants des différentes activités de conception que de l'inciter à utiliser un formalisme ou une méthode particuliers.

Pour diminuer sa complexité, le présent ouvrage a été divisé en trois tomes dont voici les contenus respectifs:

PREMIER TOME:

- Le chapitre I, qui est commun aux deux tomes, s'attache à exposer les objectifs et le plan de l'ouvrage;
- Le chapitre II est consacré à des rappels ou des "mises à niveau" concernant les notions et techniques de base de la spécification des besoins et de la conception des logiciels;
- Le chapitre III présente les principaux outils et techniques spécifiques de l'activité de conception;
- La suite est composée d'annexes permettant aux lecteurs d'acquérir ou de revoir des compétences indispensables à la compréhension des trois premiers chapitres. Le lecteur pourra s'y référer en cas de besoin:
  - Annexe I: Structure et fonctionnement d'un ordinateur;
  - Annexe II: Traitements logiciels multitâches;
  - Annexe III: Communication réseau entre processus distants;
  - Annexe IV: Rappels sur la programmation objets;
  - Annexe V: Diagrammes de flux de données en analyse organique.

DEUXIÈME TOME:

Ce document est consacré à la présentation des démarches de conception en fonction des différents points de vue adoptés par le concepteur. Ces démarches sont appliquées à des exemples concrets. Sont ainsi abordées:

- Les démarches de conception au niveau du système informatique;
- Les démarches de conception depuis les points de vue comportementaux et logiques;
- La conception détaillée.
- Comme dans le cas du tome 1, la suite est composée d'annexes permettant aux lecteurs d'acquérir ou de revoir des compétences indispensables à la compréhension des premiers chapitres. Le lecteur pourra s'y référer en cas de besoin:
    - Annexe A: Consommation de la puissance CPU par une tâche;
    - Annexe B: Consommation de la mémoire vive par une tâche;
    - Annexe C: Fiabilité des composants électroniques digitaux;
    - Annexe D: Programmation en environnement multitâches.

TROISIÈME TOME :

Ce dernier document illustre la démarche exposée par le tome II en l'appliquant à un cas concret.

II.NOTIONS ET TECHNIQUES DE BASE:

II.1.PRÉAMBULE:

Dans ce chapitre, l'auteur s'efforce de présenter au lecteur un panorama des notions et techniques courantes utilisées dans l'activité de conception des logiciels . Celles-ci sont évoquées dans la plupart des ouvrages traitant des méthodes de développement, mais elles sont le plus souvent présentées dans le contexte restreint de la méthode traitée. Les termes utilisés pour chaque notion peuvent également varier d'une méthode à l'autre.

L'auteur a pris le parti de présenter chacune de ces notions et techniques indépendamment de toute méthode pour mieux faire ressortir leurs principes directeurs et leurs objectifs.

Les lecteurs qui estiment bien connaître tel ou tel de ces sujets peuvent très bien faire l'impasse sur leur lecture.

II.2.SYSTÈMES INFORMATIQUES, INFRASTRUCTURES ET PLATE-FORMES:

II.2.1.NOTION DE CPU:

L'acronyme CPU (Central Processing Unit en anglais) désigne l'unité Centrale de Traitement d'une machine de traitement informatique. Celle-ci peut comprendre un ou plusieurs EXÉCUTEURS FIXES, autrement appelés CŒURS (Processeur multi-cœurs).

Dans la littérature spécialisée française, l'acronyme CPU est considéré comme masculin, bien qu'il désigne en fait une entité appelée unité. Nous nous conformerons à cet usage.

II.2.2.NOTION DE SYSTÈME INFORMATIQUE:

On entend par SYSTÈME INFORMATIQUE (S.I) un ensemble d'ordinateurs et d'unités de stockage d'informations reliés entre eux par un réseau de communication et supportant les logiciels nécessaires à l'activité d'une entreprise ou le contrôle d'un processus complexe.

NOTA: ne pas confondre avec le SYSTÈME D'INFORMATION d'une entreprise qui désigne l'ensemble des ressources matérielles et humaines concourant au traitement de l'information dans cette entreprise:

SYSTÈME D'INFORMATION = SYSTÈME INFORMATIQUE + Personnel affecté à l'Info.

II.2.3.NOTION D'INFRASTRUCTURE MATÉRIELLE:

L'ensemble des éléments PHYSIQUES composant ce système (unités de traitement, médias de communisation, périphériques, routeurs, etc.) constitue l'INFRASTRUCTURE MATÉRIELLE du S.I.

II.2.4.NOTION D'INFRASTRUCTURE INFORMATIQUE:

Cette infrastructure, équipée des systèmes d'exploitation et des logiciels de traitement des protocoles d'échanges de données (drivers) est appelée parfois INFRASTRUCTURE INFORMATIQUE.

II.2.5.NOTION DE PLATE-FORME:

L'ensemble constitué par une machine munie de son système d'exploitation qui supporte l'exécution d'un logiciel est appelée PLATE-FORME D'ACCUEIL.

EXEMPLES:

- Un ordinateur composé d'un processeur I5 à 4 cœurs, de 6 Mo de RAM, d'un disque dur de 8 Go et d'un BUS PCI Express peut être appelé INFRASTRUCTURE à base INTEL;
- Le même ordinateur, équipée d'un O.S. Linux Ubuntu est une PLATE-FORME Linux.

II.2.6.VIRTUALISATION:

Les logiciels de virtualisation (tels que VIRTUAL BOX ou VMWARE) permettent de SIMULER sur une infrastructure ou une plate-forme quelconque une plate-forme d'un autre type. Ainsi, on peut installer sur une plate-forme WINDOWS plusieurs plate-formes virtuelles équipées de systèmes d'exploitation différents (distributions Linux, systèmes Windows de versions différentes, etc.).

II.3.RAPPELS SUR LE PHASAGE D'UN PROJET INFORMATIQUE:

II.3.1.LES DIFFÉRENTES PHASES D'UN PROJET:

Les méthodes de conduite de projets informatiques s'appuient sur une succession de PHASES (ou étapes). Le tableau ci-dessous en donne une description sommaire:

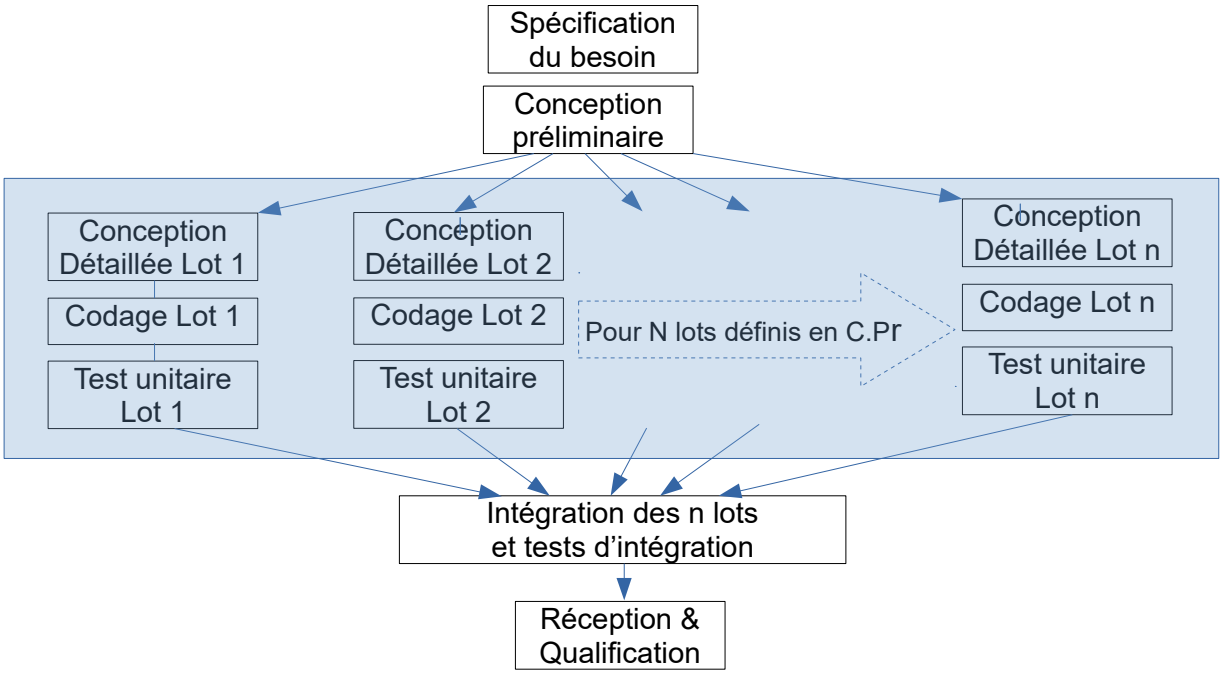
	PHASE et résultats	**DESCRIPTION DES ACTIVITÉS**
1	**Spécification du besoins** - Cahier des charges Fonctionnel, - S.T.B - Plan de réception	Recueil des besoins des différents utilisateurs, identification et caractérisation de leurs exigences vis à vis du produit. Lorsqu'à ce recueil sont ajoutées les spécifications techniques des interfaces physiques, le résultat est appelé Spécification Technique de Besoin (S.T.B).
2	**Conception préliminaire** - C.Pr - Plan d'intégration - Cahier de Réception Général (C.R.G)	- En fonction des exigences et contraintes impliquées par la S.T.B, élaboration des principes de l'architecture statique et du comportement dynamique du logiciel. - Décomposition de l'application en différents "lots" pouvant être réalisés séparément.
3	**Conception détaillée** - CD - Cahier des Tests Unitaire.	Conception de chacun des lots définis en C.Pr.
4	**Codage** - Code objet de l'appli.	Codage de chacun des lots définis en C.Pr conformément aux principes définis par leur C.D.
5	**Tests unitaires** - Dossier de tests Unitaires (D.T.U)	Validation de la conformité de chacun des lots réalisés par rapport aux spécifications définies pour chacun d'eux par la C.Pr, grâce à des procédures de tests unitaires.
6	**Intégration**	Construction du produit final par intégration progressive de chacun des lots réalisés. Chaque intégration est validée par un test d'intégration.
7	**Réception-Qualification** - Compte-rendu de réception; - Compte-rendu de qualification.	Vérification de la conformité du produit par rapport à la S.T.B (tests de réception) et vérification de l'aptitude du produit à être exploité dans des conditions opérationnelle.

II.3.2.LE PHASAGE SUIVANT LES TYPES DE MÉTHODES:

L'ordre dans lequel ces différentes phases s'enchaînent les unes aux autres dans une méthode donnée est appelé le "CYCLE DE DÉVELOPPEMENT". Trois sortes de cycles peuvent être distinguées:

II.3.2.1.LES CYCLES "EN CASCADE":

Cette sorte de cycle a pour caractéristique d'enchaîner les 7 phases une seule fois et sans retour en arrière:



Cycle en cascade

REMARQUE:

Ce type de cycle n'est jamais utilisé tel quel car cela supposerait que les résultats des activités correspondant à une phase donnée ne puisse pas être remis en cause par les phases suivantes. Ceci n'est pas tenable d'un point de vue pratique (par exemple, refuser de retoucher le codage en cas d'échec d'un tests unitaires entraîne le blocage du projet).

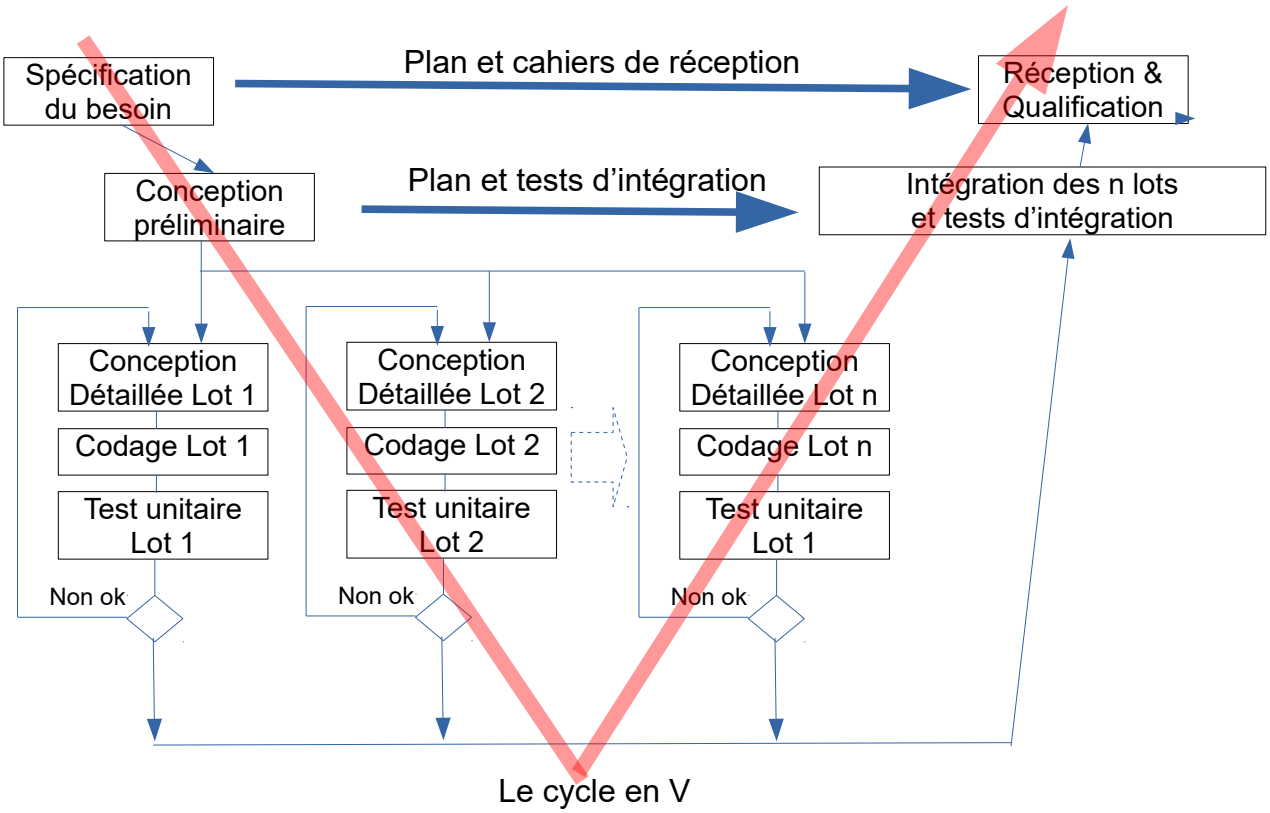
Le cycle en cascade est donc surtout cité parce qu'il est l'illustration d'un principe que l'on peut énoncer ainsi:

Dans un cycle en cascade, on ne peut pas démarrer les travaux d'une phase avant d'avoir entièrement validé les résultats de la phase précédente;

Nous verrons plus loin que ce principe l'oppose fondamentalement aux démarches "en spirales" ou "agiles".

II.3.2.2.LE CYCLE "EN V":

Le cycle en V est une amélioration du cycle en cascade qui permet de résoudre les cas de blocages signalés plus haut.



Le cycle en V

COMMENTAIRES:

Le principe qui consiste à ne démarrer aucune phase (sauf, bien sûr, la première) avant que les travaux de la phase précédente n'aient été entièrement validés n'est valable que pour la S.T.B, la Conception préliminaire et la Réception&Qualification. En revanche:

- Il est prévu que les phases de conception détaillée, codage et tests unitaires le chaque lot de production soient itérées jusqu'à ce que leur résultat soit validé (par les tests unitaires). Ceci correspond à un "sous-cycle" en spirale pour ces trois phases;
- L'intégration d'un lot au reste de l'application et les tests d'intégration correspondants peuvent intervenir avant la fin de tous les travaux de conception détaillée, codage et tests. Ils sont cependant soumis à un "plan d'intégration" issu directement des travaux de conception préliminaire ainsi que les tests d'intégration.
- Les travaux de la phase de Réception&Qualification sont préparés dès la phase de Spécification du besoin, qui élabore, en même temps que la liste des exigences, la planification et le contenu des tests de réception.

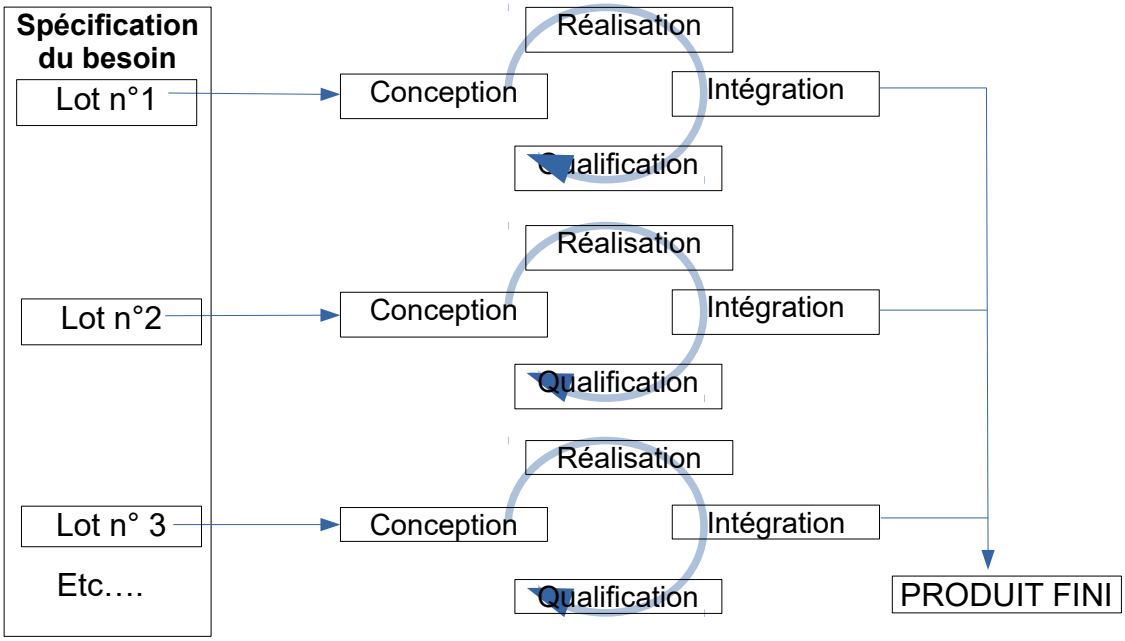
II.3.2.3.LES CYCLES EN SPIRALES ET LES MÉTHODES AGILES:

Dans les méthodes qui utilisent ce type de cycle de développement, le principe général des cycles en cascade est pris complètement à revers: il ne s'agit plus d'avancer par phases successives entièrement validées, chacune s'appliquant à la totalité de la réalisation. Le principe fondamental est plutôt:

Livrer au plus vite au client ce qui peut être livré (quitte à revenir sur ce qui a été livré pour pouvoir y intégrer ce qui reste à livrer).

De ce fait:

- Les exigences résultant de la S.T.B sont regroupées en lots dits "fonctionnels": ceci veut dire qu'ils correspondent à des parties de l'application qui peuvent être livrées au client indépendamment des autres et que celui-ci peut en théorie utiliser indépendamment du reste de l'application;
- Ces lots fonctionnels sont développés SANS TENIR COMPTE DU RESTE DE L'APPLICATION: il n'existe donc pas de conception préliminaire proprement dite: au fur est à mesure de la réalisation des lots, l'architecture générale est constamment réadaptée pour permettre l'intégration des nouveaux lots sans effet de bord.



Cycles itératifs (méthodes agiles)

Nous pouvons donc constater que dans ce type de méthode, la phase de conception préliminaire est remplacée par une activité de conception générale intégrée à la réalisation de chaque lot qui tend à "converger" vers une architecture logicielle globale prenant en compte toutes les exigences et contraintes pesant sur l'application.

II.3.3.L'ACTIVITÉ DE CONCEPTION SUIVANT LE TYPE DE MÉTHODE:

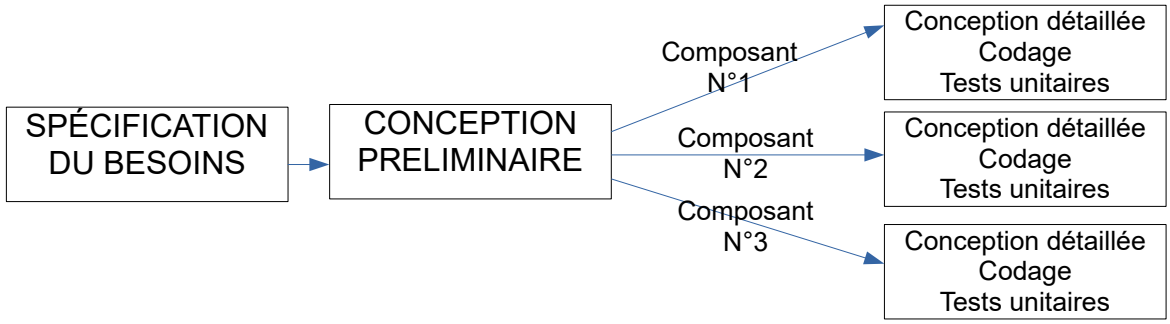
II.3.3.1.MÉTHODES DITES CLASSIQUES:

***Dans les cas des cycles "en cascade" et "en V"*:** la phase de conception recouvre deux activités distinctes et successives;

- La CONCEPTION PRÉLIMINAIRE du système à développer, dont le rôle est de définir l'ARCHITECTURE GLOBALE du système et son FONCTIONNEMENT;
- La CONCEPTION DÉTAILLÉE de chacun des composants identifiés lors de la conception préliminaire.

La conception préliminaire se situe EN AMONT de toute activité de conception détaillée.

Dans le cycle en V, la conception détaillée est souvent intégrée dans des "cycles" concernant la réalisation de chacun de composants définis en conception préliminaires et comprenant également le codage ces composants et les tests unitaires:



Conception préliminaire et détaillée (cycle en V)

II.3.3.2.CYCLES EN SPIRALES ET MÉTHODES AGILES:

Dans le cas des cycles "en spirale": les exigences issues de la spécification du besoin sont regroupées en lots dits "fonctionnels", car ils constituent des ensembles "livrables au client" (en théorie, celui-ci peut les utiliser indépendamment du reste de l'application).

La réalisation de chacun de ces lots comporte une phase de conception qui ne prend en compte que les exigences et contraintes de ce lot. Il n'y a donc pas, en théorie, de conception préliminaire (en fait, une conception préliminaire sommaire est souvent effectuée en amont).

Dans ce cadre, lors de la réalisation d'un lot donné, les éventuels conflits structurels ou dynamiques se révélant avec les lots déjà réalisés doivent être résolus, éventuellement par une adaptation de la conception de ces lots: la conception préliminaire est donc remplacée par une "ré ingénierie" permanente de la réalisation.

II.4.GÉNIE LOGICIEL ET MODÉLISATION:

II.4.1.LA NOTION DE POINT DE VUE:

II.4.1.1.DÉFINITION:

Dans son acception la plus générale, un point de vue est un "endroit où l'on se place pour voir un objet le mieux possible". Par analogie, dans le domaine du génie logiciel, un point de vue peut être défini comme correspondant à une description d'un logiciel mettant en valeur les objets correspondant aux préoccupations ou aux centres d'intérêt particuliers de l'observateur.

Par exemple:

- L'utilisateur d'un logiciel sera surtout intéressé par les services que ce logiciel peut lui offrir. Une vue d'utilisateur fera donc apparaître des éléments appelées "fonctions", "exigences", "scénarios d'utilisation" ou encore "cas d'utilisation";
- En revanche, le gestionnaire de configuration verra le logiciel du point de vue de sa description technique: ensemble des matériels, codes sources, structures et bases de données, documents, etc. qui le composent, classés par versions;
- Le développeur verra le logiciel comme un ensemble structuré de modules, classes, bibliothèques, processus, threads, sémaphores, ressources ...
- Etc.

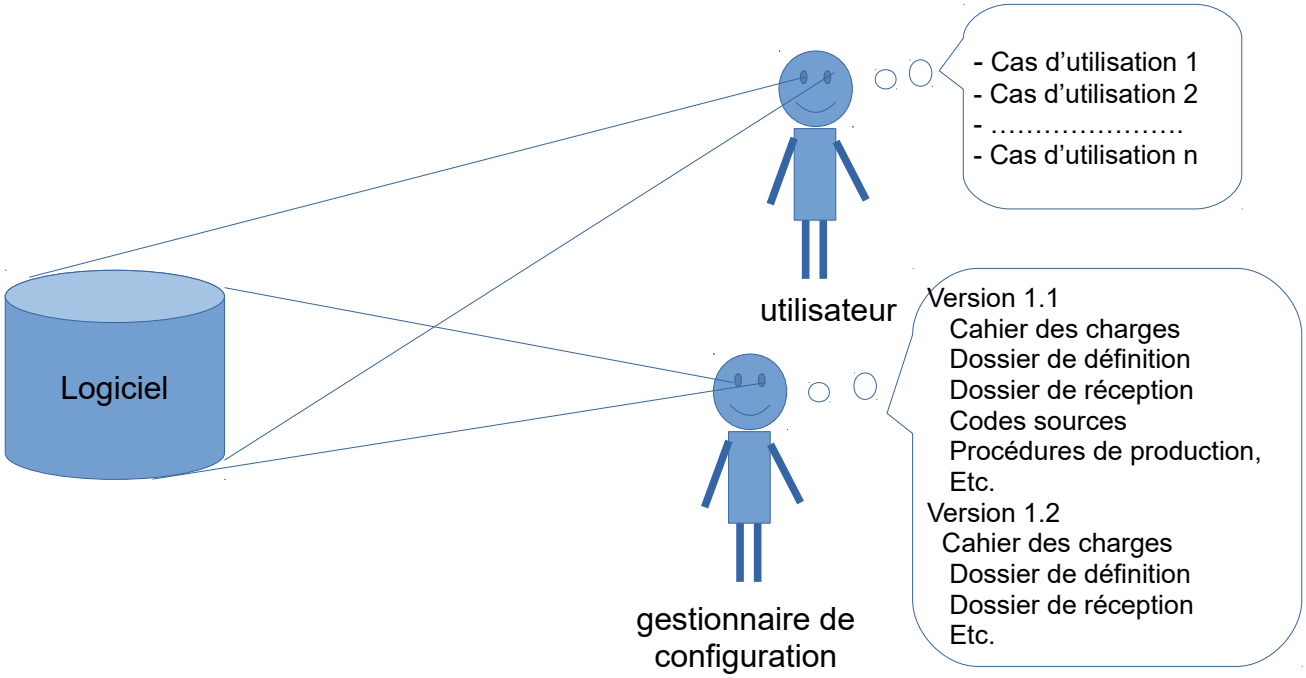


Illustration de la notion de point de vue

Deux points de vue différents conduiront donc à l'identification d'éléments de conception différents. Dans cet ouvrage, nous nous intéresserons surtout au point de vue LOGIQUE qui concerne plus particulièrement l'architecture et le comportement du logiciel.

II.4.1.2.EXEMPLE: LES DIFFÉRENTS POINTS DE VUE EN U.M.L:

La pratique qui consiste à étudier une réalisation suivant divers aspects n'est évidemment pas nouvelle:

- La réalisation d'un ouvrage de génie civil, par exemple, doit être étudiée suivant les points de vue des utilisateurs (cahier des charges), de l'architecte, de l'entrepreneur, de l'environnement, de la sécurité, etc.
- Les méthodes de développement des logiciel prévoient différentes phases qui correspondent implicitement à l'étude de l'application suivant un point de vue donné (la phase de S.T.B correspond au point de vue des utilisateurs, les phases de conception au point de vue du développeur de logiciels, etc.).

Le langage UML (**U**niversal **M**odeling **L**anguage) définit explicitement la notion de VUE pour structurer les travaux de développement d'un logiciel. U.M.L définit 5 vues, matérialisées par des diagrammes spécifiques:

1. La vue des CAS D'UTILISATION, qui correspond à la phase de spécifications fonctionnelles (UML utilise pour cela des diagrammes de cas d'utilisation);
2. La vue LOGIQUE qui s'intéresse à l'agencement des différents composants de l'architecture (définition des composants et des relations existantes entre eux). UML utilise pour cela des diagrammes d'OBJETS et de CLASSES;
3. La vue des PROCESSUS, qui s'intéresse à l'aspect DYNAMIQUE du fonctionnement (UML utilise pour cela des diagrammes d'activité, de séquences et d'états-transitions);
4. La vue de RÉALISATION qui présente d'une manière organisée les différents COMPOSANTS LOGICIELS (codes sources, fichiers de données, documentations, etc.) développés dans le cadre du projet. Cette vue est surtout utile dans le cadre de la GESTION DE CONFIGURATION du projet (UML utilise pour cela des diagrammes de composants);
5. La vue de DÉPLOIEMENT qui modélise le système dans son environnement d'exécution: postes utilisateurs, systèmes logiciels (clients, serveurs) mis en œuvre, périphériques utilisés (UML utilise pour cela des diagrammes de déploiement).

II.4.2.LA NOTION DE MODÈLE:

II.4.2.1.DÉFINITION:

Dans le domaine scientifique, le mot MODÈLE désigne une représentation simplifiée d'un système complexe obtenue en FAISANT ABSTRACTION des détails dont l'influence sur le comportement de ce système peut être négligée dans le cadre de l'étude. Il s'agit donc d'une représentation ABSTRAITE et SIMPLIFIÉE de ce système qui cependant conserve la capacité de reproduire tout ou partie de son comportement.

Pour créer un modèle, le concepteur se base sur une THÉORIE du comportement du système étudié. Ce sont les règles établies par cette théorie qui permettent de déterminer les écarts entre le comportement du système et celui du modèle (compte tenu des abstractions et simplifications effectuées) et de juger si elles restent acceptables dans le cadre de l'étude. Un modèle se compose donc toujours de deux éléments:

- Une représentation abstraite (éventuellement graphique) du système à étudier, sous la forme d'un certain nombre d'ENTITÉS représentant des éléments constituants du système réel et de RELATIONS entre ces entités;
- Un certain nombre de règles de comportement s'appliquant à cette représentation. Ces règles sont issues de la théorie du comportement du système.

II.4.2.2.REPRÉSENTATION D'UN MODÈLE:

En génie logiciel, un modèle est le plus souvent représenté sous la forme de GRAPHES dont les nœuds sont les entités déduites de la démarche d'abstraction et les arcs sont les relations définies entre ces entités. Ces graphes sont éventuellement accompagnés de commentaires textuels qui permettent de définir et de caractériser les entités et les relations et d'expliciter les règles de comportement.

La représentation sous forme de graphe présente, par rapport à une représentation entièrement littérale, l'avantage de permettre aux utilisateurs d'appréhender d'un seul coup d'œil l'ensemble des éléments "de premiers niveaux" du système. De plus, un graphe de ce type constitue un bon support pour une séance de travail en commun.

II.4.2.3.FINALITÉ DE LA MODÉLISATION:

Un modèle permet de simuler le comportement du système modélisé afin de valider la théorie ou de prévoir son évolution.

EXEMPLE: un modèle atmosphérique peut décrire l'atmosphère d'une région donnée comme un empilement de couches de cubes d'air. Chacun de ces cubes, qui constituent les cellules de base de la modélisation, peut être caractérisé par sa position, sa température, sa pression interne. Ces cellules interagissent les unes sur les autres conformément aux lois de la physique (thermodynamique des gaz, etc.), ce qui peut permettre de mettre en équations leurs interactions et ainsi, de faire des prévisions météorologiques.

II.4.3.RÔLE DE LA MODÉLISATION DANS UN PROJET INFORMATIQUE:

Au cours d'un projet informatique, la phase de conception permet de passer du MODÈLE FONCTIONNEL, représentatif des EXIGENCES DU CLIENT vis à vis du logiciel à réaliser, au MODÈLE LOGIQUE, représentatif de la SOLUTION PROPOSÉE. Le MODÈLE PHYSIQUE est, quant à lui, représenté par l'APPLICATION (le programme informatique) issue du modèle logique.

II.4.4.INFLUENCE DES OBJECTIFS DU MODÉLISATEUR:

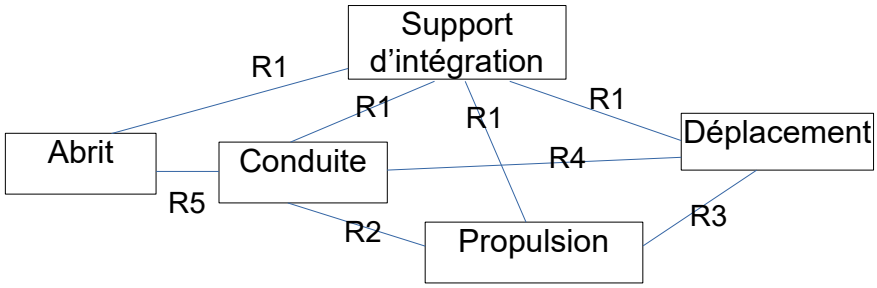
Nous avons vu plus haut que la modélisation d'un système s'obtient en faisant abstraction de détails dont le modélisateur estime pouvoir négliger l'influence sur le comportement de ce système dans le cadre de l'étude en cours. Ceci revient à dire que le modèle dépend des OBJECTIFS assignés au modélisateur, c'est à dire du POINT DE VUE qu'il adopte: a chaque point de vue correspondront des abstractions différentes, donc des modèles différents.

II.4.5.PRINCIPES DE CONSTRUCTION D'UN MODÈLE INFORMATIQUE:

II.4.5.1.EXEMPLE INTRODUCTIF:

Le graphe ci-dessous pourrait représenter un modèle sommaire pour un véhicule automobile, vu d'un point de vue LOGIQUE:

- R1 : Supporter
- R2 : Contrôler
- R3 : Entraîner
- R4 : Diriger
- R5 : Abriter



Modélisation logique d'un véhicule automobile

Les paragraphes suivants décrivent les étapes de la construction de ce modèle.

II.4.5.2.IDENTIFICATION DES ENTITÉS CONSTITUTIVES:

La première démarche consiste à identifier les différentes ENTITÉS intervenant dans le cadre du point de vue adopté. Dans notre exemple, nous avons identifié 5 entités, qui peuvent être qualifiées de LOGIQUES: elle ne correspondent ni à des besoins de l'utilisateur (entités fonctionnelles) ni à des composants physiques du véhicule (entités physiques). Nous caractériserons ces entités de la manière suivante:

- L'entité SUPPORT D'INTÉGRATION dont la finalité est de permettre l'intégration des autres entités;
- L'entité ABRIT dont la finalité est d'offrir un espace protégé et chauffé aux conducteurs et passagers du véhicule ainsi qu'aux équipements qui en ont besoin;
- L'entité CONDUITE qui permet de contrôler la vitesse et la direction du véhicule (par exemple, l'ensemble direction+accélération+freinage d'un véhicule);
- L'entité PROPULSION, qui fournit la puissance motrice;

- L'Entité DÉPLACEMENT qui permet d'utiliser la puissance motrice et les commandes de direction pour déplacer le véhicule sur le support prévu.

Les types d'entités, leur représentation graphique, leurs dénominations et les règles régissant la construction du graphe dépendent du type de modèle et de la méthode employée.

II.4.5.3.IDENTIFICATION DES RELATIONS ENTRE ENTITÉS:

la démarche suivante consiste à identifier les RELATIONS entre les ENTITÉS identifiées. Ici, nous avons identifié 5 types de relations, que nous caractériserons de la manière suivante:

- R1: **Supporter** (ex: Abrit est supporté physiquement par le support d'intégration)
- R2: **Contrôler** (ex: Propulsion utilise Conduite pour contrôler la puissance d'entraînement);
- R3: **Entraîner** (ex: Déplacement utilise Propulsion pour lui fournir la puissance d'entraînement);
- R4: **Diriger** (ex: Roulement utilise Contrôle pour diriger le véhicule);
- R5: **Abriter** (ex: Conduite utilise Abriter pour protéger les organes de conduite, le conducteur et ses passagers et les bagages).

Comme en ce qui concerne les entités, les types de relations, leur représentation graphique, leurs dénominations et les règles régissant la construction du graphe dépendent du type de modèle (fonctionnel, logique, physique, etc;) et de la méthode employée.

II.4.5.4.LES RÈGLES DE CONSTRUCTION:

Outre les symbolismes et les règles de représentation utilisés, la structuration de ces graphes doit satisfaire à un certain nombre de règles que l'on peut qualifier de SÉMANTIQUES, car elle dépendent en partie de la nature des entités.

RÈGLE FONDAMENTALE: A ce niveau de modélisation, il existe cependant une règle de construction applicable à tout type de modèle:

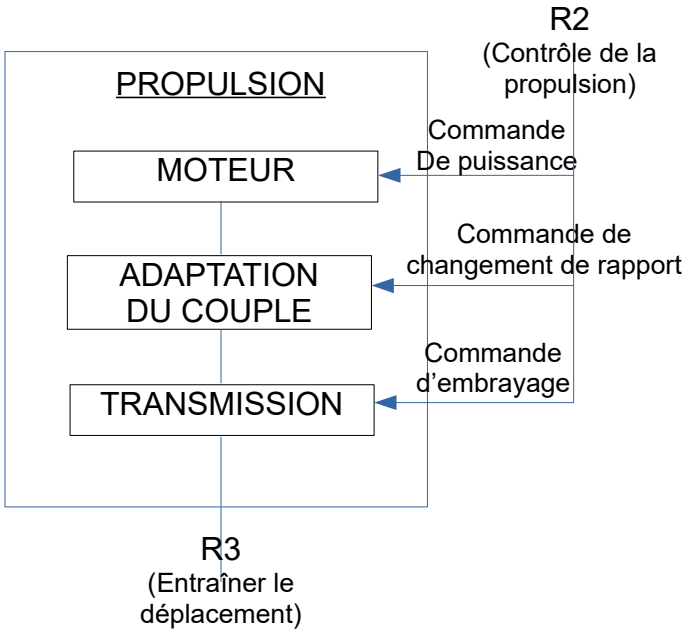
"Les entités doivent être choisies de manière a ce que la modification de l'une d'entre elles entraîne le minimum de modifications pour les autres entités".

Par exemple, dans le modèle pris en exemple, l'entité DÉPLACEMENT aurait pu s'appeler ROULEMENT, mais cela présupposerait que l'on cible un véhicule terrestre (à roues, à chenilles, etc.). Cependant, si l'on décide finalement de construire un bateau (qui est aussi un véhicule automobile), il n'y aura nul besoin de changer quoi que ce soit aux autres entités, ni aux relations: Les seuls changements concerneront l'entité DÉPLACEMENT (un gouvernail au lieu d'un essieu mobile, une coque au lieu d'un train de roulement, etc.).

II.4.5.5.DÉCOMPOSITION D'UNE ENTITÉ:

Lors de l'analyse d'un système d'une certaine complexité, il arrive très souvent que les entités identifiées lors d'une première modélisation soient elles-mêmes trop complexes pour que l'on puisse en appréhender facilement tous les détails. Dans ce cas, une nouvelle analyse s'impose afin d'obtenir une décomposition plus fine, composée d'entités moins complexes. Cette démarche est souvent appelée DÉCOMPOSITION.

EXEMPLE: L'entité PROPULSION peut être décomposée de la manière suivante:



II.5.RAPPELS SUR LA SPÉCIFICATION DES BESOINS:

II.5.1.INTRODUCTION:

Dans toutes les méthodes, la phase de spécification du besoin est celle qui précède la phase de conception. Les travaux de conception sont donc directement et majoritairement influencés par les résultats de cette phase. Il est donc nécessaire d'en rappeler ici les objectifs, les enjeux et les résultats attendus.

II.5.2.LES OBJECTIFS:

La phase de spécification du besoin a pour but de déterminer les EXIGENCES du client (que l'on appelle aussi "maître d'ouvrage", "donneur d'ordre", etc.) vis à vis du produit qu'il veut obtenir (ceci revient à répondre aux questions quoi faire? et pourquoi ?).

Ces exigences peuvent être de nature FONCTIONNELLE (quelles fonctions l'objet doit-il fournir aux utilisateurs?), mais aussi TECHNIQUES (interfaces mécaniques et électriques à respecter, ambiance de fonctionnement, fiabilité, résistance aux pannes, exigences sur le matériel ou les langages informatiques à utiliser, etc.), OPÉRATIONNELLES (quels types d'utilisateur, dans quelle ambiance de travail, etc.) ou encore ERGONOMIQUES, MÉTHODOLOGIQUES, NORMATIVES, etc.

II.5.3.LES ACTEURS CONCERNÉS:

Les exigences du CLIENT doivent comprendre les exigences de tous les intervenants agissant pour le compte ou à l'instigation de ce client: les utilisateurs, les administrateurs, les personnels de maintenance, les service chargés du respect de la qualité, de l'ergonomie et des normes en vigueur, etc.

II.5.4.RÉSULTAT DE LA SPÉCIFICATION DU BESOIN:

La phase de spécification du besoin aboutit en général à un document appelé SPÉCIFICATIONS TECHNIQUES DE BESOIN. Le contenu de ce document est l'énumération et la caractérisation des différentes EXIGENCES du client et leur caractérisation ainsi que la définition des interfaces externes.

II.5.5.EXPRESSION D'UNE EXIGENCE:

Une EXIGENCE se présente sous la forme d'un texte simple accompagné de critères quantifiés caractérisant cette exigence.

EXEMPLES:

Supposons que le problème consiste à mettre en place un système informatique de contrôle d'un réacteur chimique. Ce réacteur se présente comme une cuve dans laquelle divers produits sont amenés à interagir pour obtenir des produits de synthèse. La cuve est

dotée d'un certain nombre d'équipements de contrôle et de sécurité, parmi lesquels un capteur de pression. Une exigence sur le système de contrôle peut s'exprimer ainsi:

EXIGENCE N°1		
ÉNONCÉ	Le **personnel d'exploitation** doit pouvoir lire en temps réel les données saisies par le **capteur de pression**.	
Critère n°1.1	Équipement d'affichage	Écran de contrôle
Critère n°1.2	Délais d'affichage	Moins de 200 ms entre la date d'acquisition de la donnée par le capteur et son affichage sur l'IHM.
Critère n°1.3	Forme d'affichage	Numérique, en pascals, précision > 1/1000e.
Critère n°1.4	Signalisation d'anomalies	- Affiche "NULL" si la mesure n'est pas disponible; - Affichage en noir pour une pression { 50 P; - Affichage en rouge pour une pression ≥ 50 P.

Cette exigence est de nature FONCTIONNELLE, puisqu'elle concerne une fonction du produit destinée à une catégorie d'utilisateurs. De ce fait, elle met en relation des ACTEURS de l'environnement (le personnel d'exploitation) avec un ACTANT de cet environnement (le capteur de pression).

REMARQUE: Suivant les méthodes, une exigence peut être appelée "fonction", "user story", "scénario client", etc.

Une autre exigence sur le même système de contrôle pourrait s'exprimer comme suit:

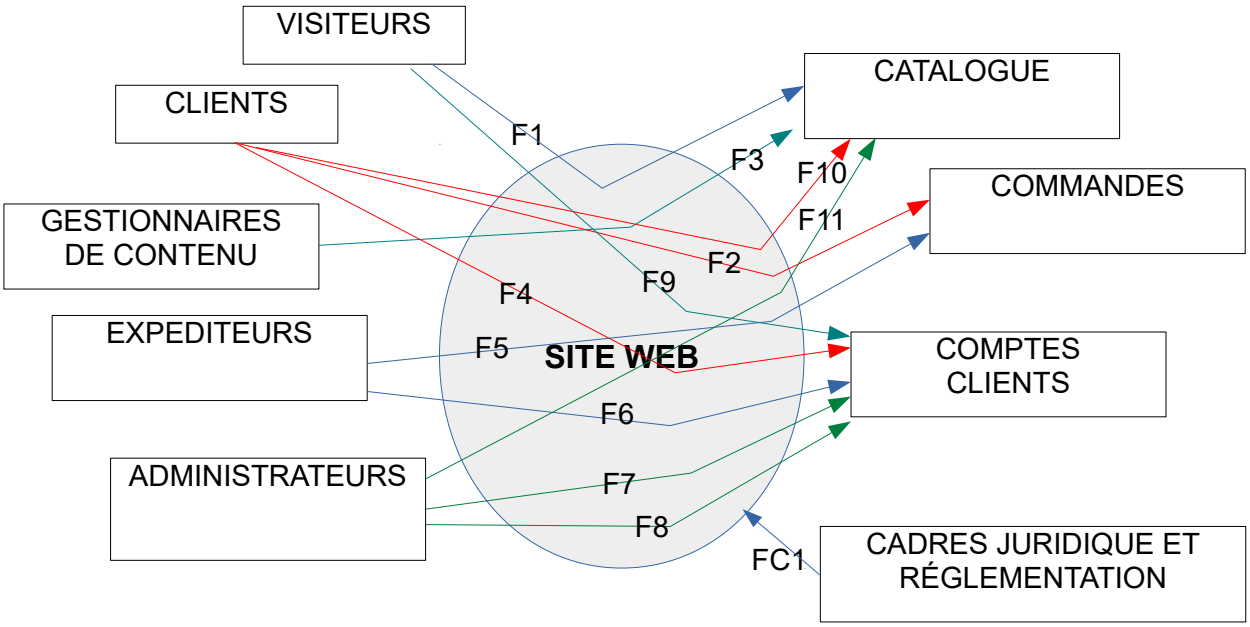
EXIGENCE N°2		
ÉNONCÉ	La probabilité d'une panne du système de contrôle entraînant l'impossibilité de surveiller la pression ou la température de la cuve doit être inférieur à 50% sur une période de 3 ans.	
Critère n°2.1	Fonctions concernées	- Visualisation de la pression interne de la cuve - Visualisation de la température interne de la cuve
Critère n°2.2	Tolérance sur la continuité de la surveillance.	Une perte de visualisation de moins de 500 ms n'invalide pas la continuité de la surveillance de la cuve.

Cette exigence est de nature OPÉRATIONNELLE, puisqu'elle ne définit pas une fonction supportée par le système mais un niveau de fiabilité exigé pour son exploitation. Une exigence de ce type, qui n'établit pas de relation entre deux acteurs ou actants de la périphérie est souvent nommée FONCTION CONTRAINTE.

II.5.6.EXEMPLE N°1 (MÉTHODE APTE):

II.5.6.1.CONSTRUCTION DU MODÈLE FONCTIONNEL:

La méthode APTE (**AP**plication aux **T**echniques d'**E**ntreprise), utilisée à partir des années 1970, permet d'évaluer les besoins et les contraintes relatives à un projet de nature quelconque (pas seulement informatique). Son application à la réalisation d'un site web marchand peut donner le résultat suivant:



FONCTION	DESCRIPTION
F1	Les visiteurs peuvent consulter le catalogue du site.
F2	Les clients peuvent passer une commande.
F3	Les gestionnaires de contenu peuvent créer et mettre à jour le catalogue du site.
F4	Les clients peuvent accéder à leur compte client (modifier, supprimer)
F5	Les expéditeurs peuvent consulter la liste des commandes.
F6	Les expéditeurs peuvent actualiser dans les comptes clients l'état des commandes en cours.
F7	Les administrateurs peuvent supprimer les comptes clients en déshérence.
F8	Les administrateurs peuvent consulter les commandes.
F9	Les visiteurs peuvent ouvrir un compte client.
F10	Les clients peuvent consulter le catalogue
F11	Les administrateurs peuvent consulter le catalogue
FC1	Respecter les cadres juridique et réglementaires concernant le commerce en ligne et la protection des données.

REMARQUE: *ceci ne constitue évidemment qu'une application sommaire des premières étapes de la méthode APTE: dans un projet, les fonctions obtenues seraient soit (si nécessaire), décomposées en fonctions plus simples, soit caractérisées par divers critères évaluables.*

Le graphe obtenu ci-dessus, accompagné des informations littérales contenues dans le tableau constitue un exemple de MODÉLISATION de l'aspect FONCTIONNEL du produit à développer: le produit est étudié du point de vue de son utilisation.

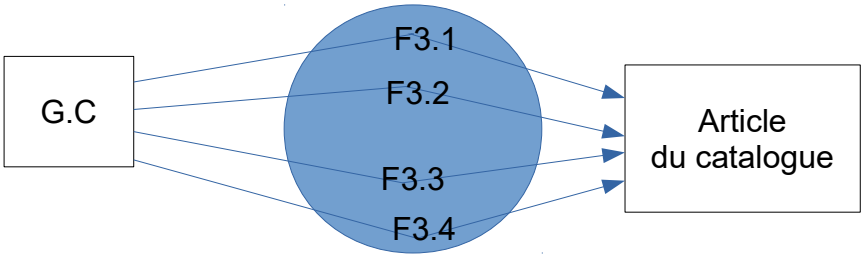
DÉCOMPOSITION D'UNE FONCTION:

Dans l'exemple précédent, la fonction F3 (Les gestionnaires de contenu peuvent créer et mettre à jour le catalogue du site) "encapsule" une réalité complexe qui mérite d'être précisée (raffinée).

Le raffinage peut par exemple conduire à la décomposition en 4 sous-fonctions:

- F3.1 Les G.C. peuvent ajouter un article au catalogue;
- F3.2 Les G.C. peuvent supprimer un article du catalogue;
- F3.3 Les G.C. peuvent modifier le libellé et le prix d'un article du catalogue;
- F3.4 Les G.C. peuvent déclarer un article "en promotion".

REMARQUE: Cette liste pourrait être représenté par le "sous-graphe" suivant:



La démarche de raffinage peut ainsi être itérée jusqu'à ce que les entités obtenues soient suffisamment simples.

II.5.6.2.IDENTIFICATION DES ENTITÉS CONSTITUTIVES:

La première démarche consiste à identifier les différentes ENTITÉS intervenant dans le cadre du point de vue adopté:

Dans notre exemple (point de vue des utilisateurs), nous avons identifié:

- Des ACTEURS, qui correspondent à des catégories d'utilisateurs (Visiteurs du site, Client (Visiteurs possédant un compte client), Gestionnaires de contenus, Administrateurs, Expéditeurs);
- Des ACTANTS, qui sont les entités gérées par le système (Catalogue, liste des commandes, liste des comptes clients, cadre juridique et réglementaire).

Les types d'entités, leur représentation graphique, leurs dénominations et les règles régissant la construction du graphe dépendent du type de modèle (fonctionnel, logique, physique, etc;) et de la méthode employée.

II.5.6.3.IDENTIFICATION DES INTERACTIONS:

La démarche suivante consiste à identifier les INTERACTIONS entre les ENTITÉS identifiées:

Dans notre type de vue, ces interactions interviennent entre:

- Les acteurs et les actants ("fonctions principales", correspondant aux exigences des utilisateurs);
- Les acteurs ou actants et le produit lui même (fonctions contraintes imposées au réalisateur par l'environnement d'utilisation).

Les types d'interactions, leur représentation graphique et leurs dénominations et les règles régissant la construction du graphe dépendent du type de modèle (fonctionnel, logique, physique, etc;) et de la méthode employée.

II.5.6.4.LES RÈGLES SÉMANTIQUES:

Nous avons vu qu'outre les symbolismes et les règles de représentation utilisés, la structuration des graphes de modèles doit satisfaire à un certain nombre de règles de nature "sémantiques", qui dépendent du type du modèle. Dans le cas du modèle fonctionnel, les différentes fonctions identifiées doivent être indépendantes les unes des autres: la disparition d'une des fonctions ne doit pas remettre en cause l'utilité d'une autre fonction.

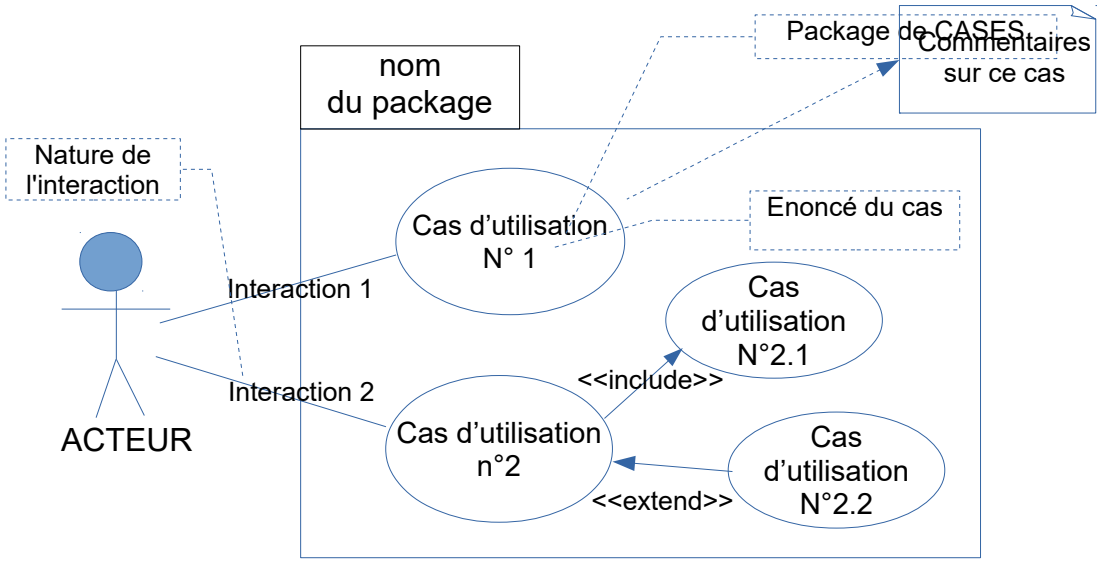
II.5.7.EXEMPLE N°2 (U.M.L):

II.5.7.1.CONSTRUCTION DES MODÈLES FONCTIONNELS (USE CASES):

Le langage de modélisation U.M.L utilise pour spécifier et caractériser les exigences des utilisateurs des DIAGRAMMES DE CAS D'UTILISATION (USE CASES). Chacun de ces diagrammes permet de représenter le comportement du logiciel correspondant aux attentes d'une classe d'utilisateur donnée (ce qui revient à représenter les EXIGENCES de cette classe d'utilisateur vis à vis du logiciel). Son application à la réalisation d'un site web marchand peut donner les résultat suivants:

II.5.7.2.MODÉLISATION D'UN CAS D'UTILISATION:

Dans sa forme la plus simple, un diagramme de cas d'utilisation se présente de la manière suivante:

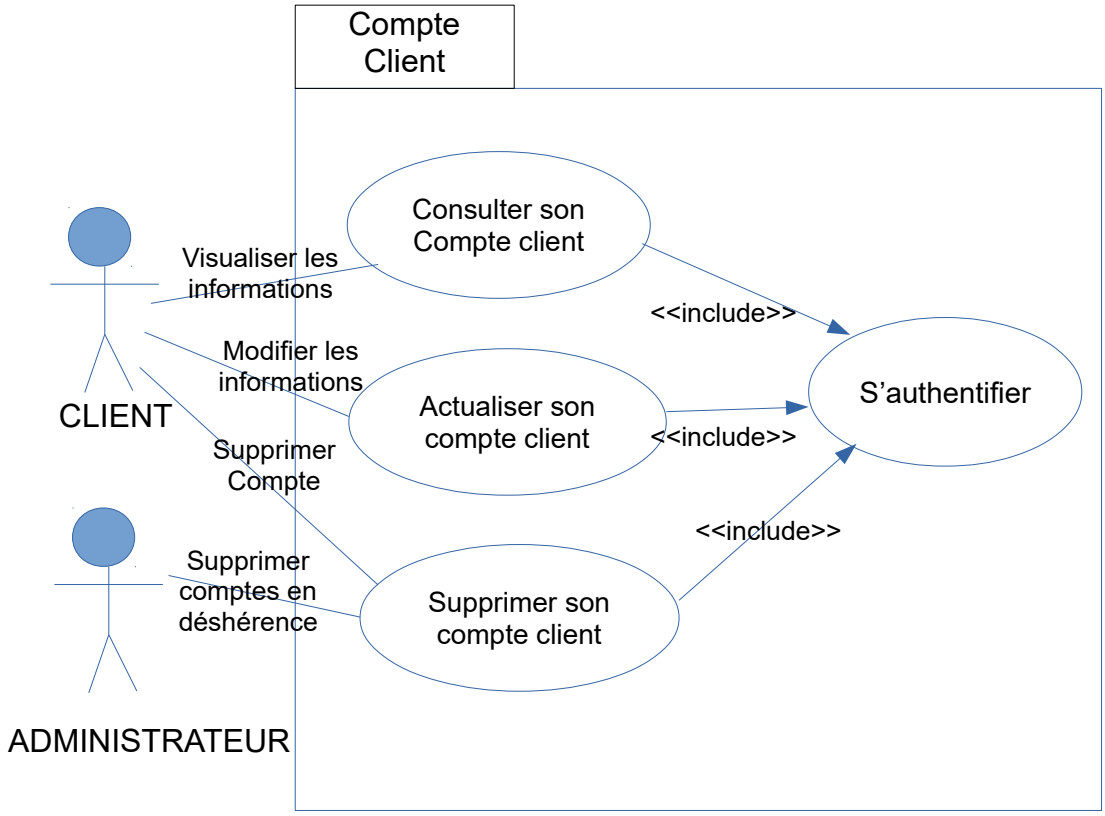


REMARQUES:

- Un ACTEUR est une entité interagissant avec le avec le système étudié. Ce peut être une personne (un utilisateur) ou un système matériel (un ordinateur en réseau, par exemple);
- Un CAS D'UTILISATION représente une action complexe que l'acteur doit être capable d'effectuer grâce au logiciel à développer (c'est une EXIGENCE). Par exemple: "Consulter mon compte en banque";
- Une INTERACTION précise l'action que l'acteur veut pouvoir effectue (par exemple: "visualiser mes débits");

- Les cas d'utilisation figurant dans un "package de cas" peuvent être reliés par deux relations: l'inclusion ("faire un retrait" inclue "Consulter mon compte" et "débiter mon compte") et l'extension ("retirer en euros" précise le cas "faire un retrait");
- Le diagramme peut être accompagné de commentaires littéraux précisant et caractérisant chacun des cas d'utilisation.

II.5.7.3.EXEMPLE-CAS D'UTILISATION LIES À UN COMPTE CLIENT:



NOTA: Chacun des cas d'utilisation du schéma doit être accompagné d'une fiche le caractérisant.

II.6.CONCEPTION PRÉLIMINAIRE ET CONCEPTION DÉTAILLÉE:

II.6.1.DIFFÉRENCE ENTRE LES DEUX PHASES DE CONCEPTION:

La plupart des méthodes de conception des logiciels séparent l'activité de conception en deux phases: la phase de CONCEPTION PRÉLIMINAIRE (ou CONCEPTION GLOBALE) et la phase de CONCEPTION DÉTAILLÉE. Le tableau suivant explicite les objectifs et les résultats de ces deux phases:

PHASE	OBJECTIFS	RÉSULTATS
CONCEPTION PRÉLIMINAIRE (ou GLOBALE)	• En fonction des exigences et contraintes exprimées par la S.T.B, élaboration des principes directeurs de l'architecture statique et du comportement dynamique du logiciel. • Décomposition de l'application en différents lots pouvant être réalisés séparément.	• Modèles architecturaux et comportementaux de l'application; • Spécifications techniques des entités logicielles composant les différents lots de réalisation (interfaces, traitements supportés, contraintes à respecter vis-à vis de l'environnement.
CONCEPTION DÉTAILLÉE	Conception de chacun des lots définis par la Conception préliminaire.	• Conception détaillée de chacune des entités logicielles composant les différents lots de réalisation (architecture et comportement internes). • Ces travaux de conception doivent être assez détaillés pour permettre de démarrer le codage de ces entités.

La phase de conception préliminaire étudie donc les aspects architecturaux et comportementaux du logiciel DANS SA GLOBALITÉ.

Lors de la phase de conception détaillée, chacune de équipes chargées de la conception de chacun des lots est censée travailler à partir des spécifications et contraintes définies en conception préliminaire pour les lots qui lui sont attribués, en partant du principe que ces travaux de conception préliminaires ont repéré et résolu tous les problèmes de coexistence des différents lots.

II.6.2.LES SOUS-PROJETS:

Dans le contexte de certains gros projets, la phase de conception préliminaire peut aboutir à la définition de lots tellement volumineux et complexes qu'ils constituent de véritables sous-projets, nécessitant eux-mêmes une étude de conception préliminaire.

II.6.3.CONCEPTION PRÉLIMINAIRE OU CONCEPTION GLOBALE?:

Nous avons vu que dans le cadre des méthodes agiles la phase de conception préliminaire n'existe pas (du moins en théorie): l'application est développée en un certain nombre de lots "fonctionnels", qui sont en fait des regroupements d'exigences. Chacun des lots est développé sans tenir compte des autres. Les problèmes d'incompatibilité ou d'effets de bord sont résolus au moment de l'intégration, par adaptations successives de l'architecture de l'application aux nouveaux lots intégrés.

On ne peut donc parler dans ce cas de conception "préliminaire". En revanche, les travaux d'adaptations successives de l'architecture concourent bien à ce que l'on peut appeler la CONCEPTION GLOBALE du logiciel. La conception "préliminaire" n'est autre que la conception globale quand, dans les méthodes "classiques", elle est effectuée PRÉALABLEMENT à tout travail de conception détaillée.

III.MÉTHODES ET OUTILS DE LA CONCEPTION:

III.1.CONCEPTION DES ALGORITHMES:

III.1.1.INTRODUCTION:

Quel que soit le paradigme de programmation utilisé pour la réalisation d'un projet (programmation procédurale, orientée objet, événementielle, séquentielle, etc.), la réalisation d'une application consiste toujours à traduire sous une forme exécutable par un ordinateur les traitements que cette application doit supporter.

Les algorithmes présentent l'avantage de représenter la logique et le comportement d'un traitement informatique quelconque en faisant abstraction du langage informatique qui sera adopté pour réaliser son code objet. D'autre part, ils s'adaptent à tous les niveaux de la conception (conception préliminaire, conception détaillée).

Il est donc souhaitable que toute personne amenée à concevoir des logiciels possède une connaissance et une pratique suffisantes de l'algorithmique.

III.1.2.DÉFINITION:

Dans son sens le plus général, le terme ALGORITHME désigne une méthode de résolution particulière caractérisée par les points suivants:

- Un algorithme se présente sous la forme d'une liste finie d'opérations dites «élémentaires» destinées à s'appliquer à un nombre fini de données;
- Les opérations élémentaires s'adressent à un exécutant (ou processeur) qualifié pour réaliser ces opérations une à une en les appliquant à des données d'une manière ordonnée dans le temps;
- L'exécution d'un algorithme doit conduire à la solution du problème après exécution d'un nombre fini d'opérations.

REMARQUE: l'exécutant dont il est fait mention au deuxième alinéa n'est pas forcément une machine physique: il peut s'agir d'une "abstraction de machine" pour laquelle le concepteur a défini des opérations élémentaires (par exemple, une CLASSE) ou même d'un opérateur humain: dans ce cas, l'algorithme décrit une procédure de travail.

Dans un ordinateur "classique", basé sur le modèle de VON NEUMAN (ce qui est le cas pour la quasi-totalité des ordinateurs actuels), un PROGRAMME est toujours la traduction d'un ou plusieurs ALGORITHMES dans le code d'ordre du processeur employé.

III.1.3.LES OUTILS DE LA CONCEPTION DES ALGORITHMES:

III.1.3.1.INTRODUCTION:

L'ALGORITHMIQUE définit un langage de description des algorithmes qui s'appuie sur des règles rigoureuses et peut éventuellement être utilisé pour décrire et analyser les traitements. Cependant, par son formalisme, ce langage se situe au niveau des langages de programmation impératifs (java, c, etc.), c'est à dire de l'instruction élémentaire (addition, soustraction, comparaison, etc.).

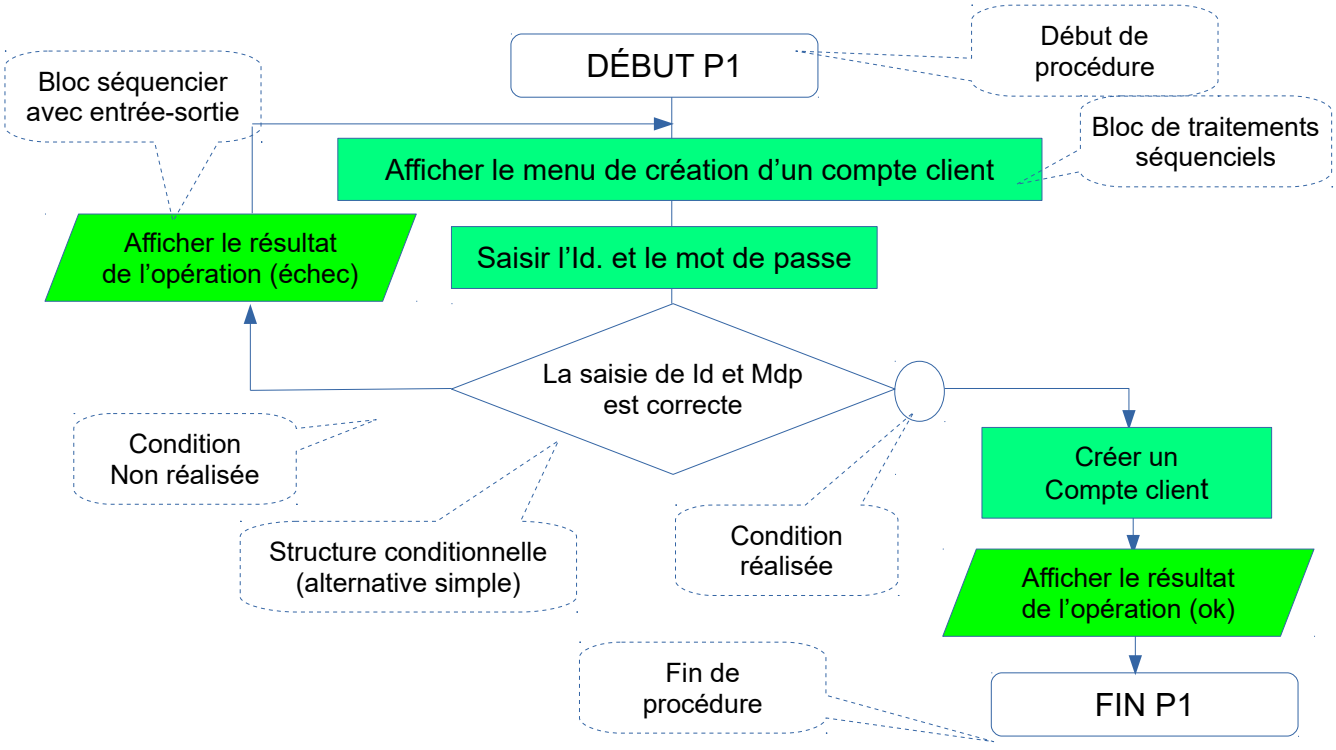
Ce niveau de détail n'est évidemment pas pertinent au début de la phase de conception ou l'on a besoin de faire abstraction des détails des traitements pour se concentrer sur leurs grandes lignes. De ce fait, les développeurs ont, dès les années 1960, préféré au langage de description des algorithmes deux outils qui permettent de s'adapter au niveau d'analyse auquel on se trouve, les ORGANIGRAMMES DE PROGRAMMATION et le PSEUDO CODE.

Ces deux outils permettent de décrire avec un minimum de formalisme la STRUCTURE LOGIQUE d'un algorithme, les traitements séquentiels étant décrits en texte libre.

III.1.3.2.LES ORGANIGRAMMES DE PROGRAMMATION:

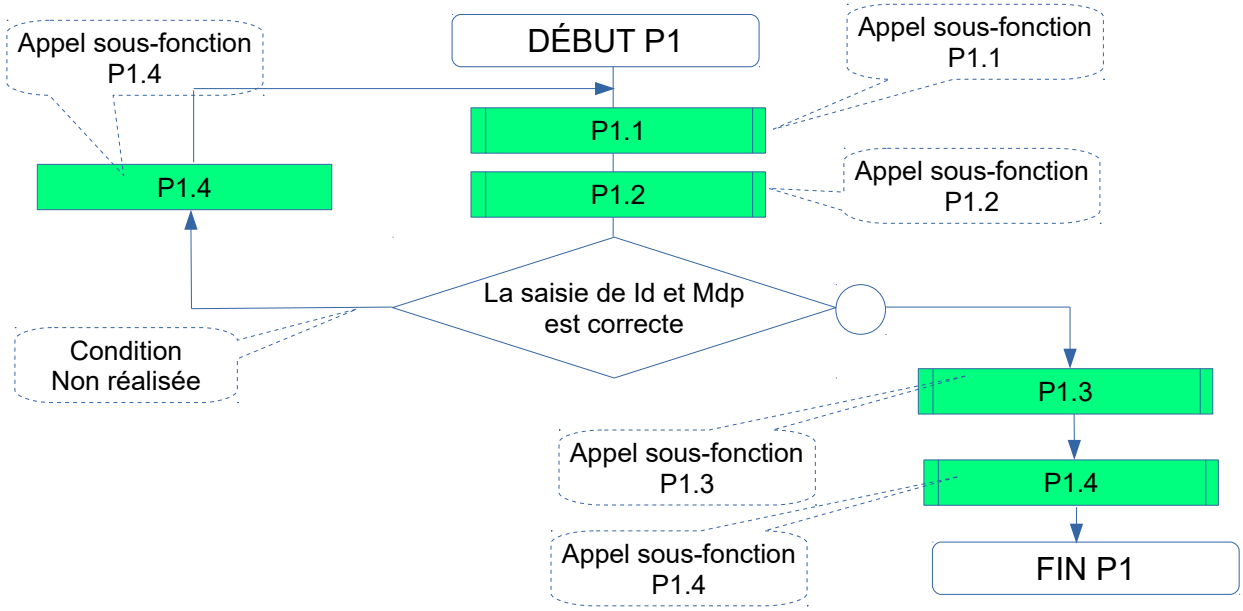
Les organigrammes de programmation ont probablement été les premiers outils utilisés pour le développement des logiciels. Ils ont l'avantage de donner une représentation graphique des algorithmes:

EXEMPLE: L'organigramme ci-dessous représente le résultat de l'analyse algorithmique de la création d'un compte client:



COMMENTAIRES:

Nous voyons que les 5 blocs (en vert) correspondant à 4 traitements dont on désire faire abstraction à ce niveau (deux correspondent en fait au même traitement: Afficher résultats). Cet organigramme permet donc de détecter la possibilité de faire abstraction de ces quatre blocs en les encapsulant dans les 4 sous-fonctions P1.1, P1.2, P1.3 et P1.4. Il est alors possible de remplacer ces blocs par des appels à des sous-procédures. On obtient alors l'organigramme suivant, dans lequel seule apparaît la structure logique de l'algorithme:



Les sous-fonctions P1.1, P1.2, P1.3 et P1.4 pourront être soumises à un raffinage de ce type, grâce à des sous-algorithmes. Le processus s'arrêtera lorsque toutes les sous-fonctions issues du raffinement pourront être codées directement.

REMARQUE:

Une conception utilisant les organigrammes de programmation se présente donc comme une arborescence de diagrammes allant de l'organigramme de principe (dans les niveaux supérieurs) à des organigrammes très détaillées pouvant préciser des détails de programmation. Les avantages de cet outil sont:

- Il est très intuitif et demande peu de formation;
- Il convient au phases de conception générales comme aux phases de conception détaillée.

Cependant, les organigrammes de programmation ont peu à peu été abandonnés en tant qu'outils de conception. En effet:

- Ils sont très difficiles à créer et surtout à maintenir sans un outil graphique;
- Ils permettent de concevoir des algorithmes qui ne suivent pas les règles de la programmation structurée sur l'enchaînement et l'imbrication des structures

alternatives et répétitives, ce qui peut conduire à des codes objets difficilement compréhensibles et peu maintenables.

III.1.3.3.LES PSEUDO-CODES:

Les pseudo-codes, également appelés L.D.A (Langage de Description d'Algorithmes) sont des "langages" qui permettent décrire un algorithme en utilisant le langage naturel simplement structuré par un petit nombre de mots clefs.

Il n'existe pas, à proprement parler, de convention d'écriture du pseudo-code. Le tableau suivant indique les notations les plus fréquentes:

CONVENTION	EXEMPLE(S)
SPÉCIFICATION DE TRAITEMENT	
En général, les blocs textuels délimités par des accolades spécifient un traitement dont on désire faire abstraction des détails à ce niveau.	{calcul de la mensualité du prêt à 3% sur 18 ans}
DÉCLARATIONS DE VARIABLES	

<Type> <Nom de variable> = <Valeur> [<Unité>]
<Type> = [entier/flottant/booleen/chaine/couleurs/...etc..]
	entier Compteur = 0;
	flottant Pression = 3,52 bars;
	chaine MotDePasse = '&3RFS';
	couleurs EtatFeuTricolore = rouge;

STRUCTURES ALTERNATIVES ET RÉPÉTITIVES	
SI <condition> ALORS {bloc alternative 1} SINON {bloc alternative 2} FINSI	SI EtatFeu = rouge ALORS {Attendre} SINON {Traverser} FINSI
TANT QUE <condition> FAIRE {bloc à itérer} FINFAIRE	TANT QUE {J'ai faim} ALORS {Je mange} FINFAIRE
DÉCLARATION DE FONCTION	
<Type Fonction> <Nom fonction> ([[{type> {arg1>[, {type> {arg2>[, {type> {arg3 [,...]]])	Flottant Puissance (Flottant X, entier N)
APPEL D'UNE FONCTION	
[<Nom variable> =] <nom fonction> ({Liste d'arguments>)	Résultat = TestExistance ("sources");
STRUCTURE D'UNE FONCTION	
<Déclaration de fonction> DEBUT {Corps de la fonction} [RETOURNER] [{nom de variable>/{valeur>]; FIN	Flottant Puissance (Flottant X, entier N) Flottant R; DEBUT { R = X puissance N}; FIN

NOTA: En général, les développeurs d'une même entreprise enrichissent ce "langage" d'autres mots clefs. Exemples:

Alternatives multiples:

```
COMMUTER <cas>
        CAS <cas 1>: {bloc cas 1}
        CAS <cas 2>: {bloc cas 2}
        ------------------------------------
        CAS <cas n>: {bloc cas n}
        DEFAUT:      {bloc cas défaut}
FINCAS
```

Contrôleur d'événement:

```
QUAND <événement> FAIRE <Nom de procédure>
ATTENDRE <événement>
etc.
```

Contrairement aux organigrammes, l'écriture en pseudo-code garantit une démarche structurée dans la description de l'algorithme, grâce aux mots clefs.

Le faible formalisme permet des descriptions plus ou moins fines: il est donc possible de faire abstraction de certains détails de traitements complexes pour les renvoyer à des sous-procédures.

EXEMPLE: Une procédure de création d'un compte client peut s'écrire en pseudo-code:

```
DÉBUT
      booleen: compte_créé = faux;
      chaîne: Id, mdp;

      TANT QUE ( création = faux ) FAIRE
            {Afficher le menu de création d'un compte client};
            ATTENDRE {saisie de l'Id. et du mot de passe dans Id et mdp}
            compte_créé = Créer un compte client ( Id, mdp) ;
            SI compte_créé = vrai ALORS
                  Afficher le message ("Création réussie");
            SINON
                  Afficher le message ("couple Id., mot de passe inacceptable:
                    recommencer.");
            FINSI
      FINFAIRE
      RETOURNER Compte_créé;
FIN
```

Les algorithmes correspondant aux blocs de traitement pour lesquels on a fait abstraction des détails seront réalisés dans une deuxième phase d'analyse.

EXEMPLE:

```
// Procédure Créer un compte client ( chaîne ID, chaîne MDP );
DEBUT
    entier Nombre_clients = 0;

    Nombre_Clients = {Compter clients tels que id = ID et pass = MDP };
    SI Nombre_clients = 0 ALORS     RETOURNER vrai;
    SINON                           RETOURNER faux.
FIN
```

III.2.NOTIONS DE MODULES LOGICIELS ET D'ARCHITECTURE MODULAIRE:

III.2.1.NOTION DE MODULE LOGICIEL:

III.2.1.1.DÉFINITION:

Du point de vue méthodologique, un MODULE LOGICIEL est un composant complexe ENCAPSULANT des traitements (et parfois les DONNÉES concernées par ces traitements). Ce regroupement peut être effectué en fonction de divers objectifs:

- Concourir à la gestion d'une même ressource (ex: une bibliothèque des fonctions réseau TCP-IP);
- Regrouper les traitements relatifs à la gestion d'une même structure de données complexe (ex: regrouper les traitements d'accès à une base de données);
- Regrouper des traitements de même nature (ex: bibliothèque de fonctions mathématiques);
- Minimiser les interfaces (ex: regrouper des fonctions liées par des dépendances procédurales pour ne faire apparaître que les points d'accès externes).
- Améliorer la RÉUTILISABILITÉ des logiciels;
- Découper la réalisation de l'application en LOTS dont on peut confier la réalisation à diverses équipes de développement;
- Etc

III.2.1.2.CARACTÉRISTIQUES D'UN MODULE LOGICIEL:

- L'existence d'un module doit autant que possible se justifier indépendamment de celle des autres modules de l'application (condition de réutilisabilité);
- Un module encapsule des traitements dont certains peuvent être activés par l'intermédiaire d'interfaces accessibles aux logiciel de l'environnement. De ce fait, ces traitements sont parfois appelés OPÉRATIONS EXTERNES du module. Ces opérations sont les MÉTHODES PUBLIQUES d'une classe ou les POINTS D'ENTRÉE d'une bibliothèque de fonctions;
- Un module peut également encapsuler des traitements inaccessibles pour les logiciels de l'environnement. Ces traitements, qui sont utilisées par les opérations externes, sont parfois appelés OPÉRATIONS INTERNES (ce sont les MÉTHODES PRIVÉES d'une classe);
- Pour utiliser un module, un développeur n'a besoin de connaître que le mode d'emploi des INTERFACES que ce module propose aux utilisateurs (prototypes des opérations externes);
- Lorsqu'un module est activé via une de ses interfaces externes, son fonctionnement et le résultat obtenu (service rendu, données retournées) ne dépendent que de la valeur des paramètres explicitement fournis par le logiciel appelant, à moins que ce module n'exploite des données ou des événements en provenance de la périphérie (par exemple: modules d'acquisition de données).

REMARQUE: si la programmation OBJETS est forcément modulaire, l'inverse n'est pas vrai car un module n'encapsule pas forcément les données sur lesquelles il agit. Ceci entraîne diverses conséquences:

- Deux MODULES d'une même application peuvent agir sur la même structure de données (ce qui peut entraîner des effets de bord), à moins d'adopter des conventions de codage contraignantes à ce sujet ou de choisir la programmation objets;
- Il est impossible de parler de l'état d'un module comme on parle de l'état d'un objet.

III.2.1.3.POINT DE VUE LOGIQUE:

Du point de vue LOGIQUE, un module réalise une ABSTRACTION des traitements informatiques qu'il renferme. Ceci veut dire qu'il permet de DISSIMULER la manière dont ces traitements sont réalisés (algorithmes utilisés, particularités d'implémentation, etc.). On dit que le module ENCAPSULE ses traitements internes: pour l'utiliser, le développeur n'a besoin de connaître que le mode d'emploi des INTERFACES que ce module propose aux logiciels utilisateurs (types des paramètres à fournir, types des paramètres retournés) et les SERVICES fournis par les différents points d'accès.

III.2.1.4.POINT DE VUE LOGICIEL:

Du point de vue INFORMATIQUE, un module se présente sous la forme d'un ensemble de FICHIERS pouvant renfermer:

- Des déclarations de CLASSES (programmation objet), de FONCTIONS (programmation procédurale), de STRUCTURES DE DONNÉES ou d'INTERFACES (prototypes de fonctions ou de méthodes);
- Des procédures de production du code exécutable du module (procédures de compilation, édition de liens, etc);
- Le code exécutable du module.

EXEMPLE:
Dans un logiciel rédigé en langage C, les fichiers sources d'une bibliothèque de fonctions se présente en général sous la forme d'un fichier doté de l'extension ".c" qui contient le code source des différents fonctions encapsulées par la bibliothèque, et d'un fichier doté de l'extension ".h" qui contient les déclarations des prototypes (interfaces) de ces différentes fonctions (ainsi que des déclarations de constantes en rapport avec l'objet de la bibliothèque). Le fichier d'extension .h suffit pour utiliser le module car il explicite entièrement les interfaces des différentes fonctions. Une telle bibliothèque est un exemple de module logiciel.

D'après ce qui précède, une classe (en programmation objets) constitue un module logiciel.

III.2.2.NOTION D'ARCHITECTURE MODULAIRE:

III.2.2.1.DÉFINITION:

L'architecture d'un logiciel est dite MODULAIRE lorsqu'elle se présente sous la forme d'un certain nombre d'unités de programmation satisfaisant à la notion de MODULE.

III.2.2.2.INTERACTIONS ENTRE MODULES:

Dans une architecture modulaire, les échangent entre modules sont souvent appelés MESSAGES: ce terme très général permet de ne rien préjuger de la nature et du protocole de ces échanges. Il peut s'agir:

- D'échanges de données ou de commandes en mode SYNCHRONE (le module récepteur est mis en attente du message de l'émetteur);
- D'échanges de données ou de commandes en mode ASYNCHRONE (l'émetteur dépose les données ou commandes dans une "boîte aux lettres" (un puits de données rémanentes, une file d'attente ou une pile), le récepteur va les "relever" quand il en a besoin;
- D'échanges de SIGNAUX (l'émetteur émet un signal d'interruption logicielle vers le récepteur, activant chez ce dernier un "contrôleur de signal" qui va permettre de déclencher un traitement (fonction ou méthode) de ce module;
- De l'activation directe par l'émetteur d'une méthode ou fonction du récepteur situé dans la machine locale (appel procédural).
- De l'activation de procédures ou méthodes à distance (entre deux machines), grâce à des MIDDLEWARES tels que DCOM (Distributed Component Object Model) de Microsoft, CORBA (Common Object Request Broker Architecture) ou RMI (Remote Method Invocation) de Java.
- Plus généralement, de toute autre cause permettant d'activer le code du récepteur (comme la libération par l'émetteur d'une ressource dont le récepteur est en attente de disponibilité).

III.2.2.3.UTILITÉ DE LA DÉMARCHE MODULAIRE:

Du point de vue architectural, "encapsuler" les traitements dans un MODULE présente plusieurs avantages:

- La programmation modulaire permet de dégager deux concepts indépendants l'un de l'autre: l'INTERFACE d'un module et l'ALGORITHME utilisé pour réaliser les traitements. Il suffit de définir l'Interface d'un module pour que le développement d'autres modules de l'application puisse être mené EN PARALLÈLE avec son développement, même si ces autres modules doivent l'utiliser: il suffit pour cela de

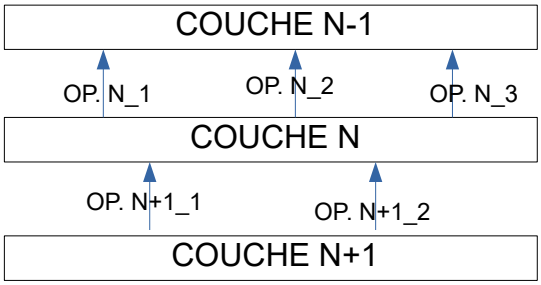
remplacer le module manquant par un composant simulant ses traitements internes (ces composants peuvent être appelés stubs, mocks, muets ou bouchons suivant leur rôle et leur complexité);

- L'encapsulation permet de FAIRE ABSTRACTION de détails de réalisation dont on veut repousser l'étude à plus tard;
- Une fois développé, un module pourra être RÉUTILISÉ dans la même application ou dans des applications différentes en occasionnant une économie de ressources et de temps de travail;
- Les modules d'une application représentent autant de LOTS de réalisation qui peuvent être confiée à diverses équipes qui peuvent les réaliser indépendamment les unes des autres: ceci permet de paralléliser les tâches de conception détaillée, codage et tests unitaires, donc de raccourcir les délais. Chacun des lots peut également être aisément sous-traité.

III.3.L'ARCHITECTURE EN COUCHES LOGICIELLES:

III.3.1.DÉFINITION:

La structuration en couches consiste à répartir les composants logiciels constituant une application en différentes COUCHES (LAYERS). Dans une telle organisation, chaque couche de logiciel se comporte comme une MACHINE VIRTUELLE qui offre aux logiciels de la couche supérieure un jeu d'OPÉRATIONS VIRTUELLES (les fonctions ou méthodes publiques de ses composants) et utilise, pour réaliser les traitements associés à ces opérations, les opérations offertes par la couche inférieure:



L'interface de chaque couche constitue donc une ABSTRACTION des traitements de cette couche et des couches inférieures. La couche la plus profonde est celle qui dialogue directement avec le système d'exploitation de la machine physique.

III.3.2.PRINCIPES GÉNÉRAUX:

Une couche ne constitue pas une entité logicielle unique et indivisible. Elle est au contraire constituée de différents MODULES dont la répartition et les interactions sont régies par les règles suivantes:

RÈGLE N° 1 (Répartition):

- La couche supérieure doit accueillir les modules chargés des interfaces homme-machine. Elle doit également relayer les requêtes des utilisateurs vers la couche inférieure;
- Dans les autres couches, les composants logiciels sont répartis en fonction de la plus ou moins grande spécificité des traitements qu'ils supportent par rapport au "métier" de l'application: MOINS un composant est spécifique à l'application, PLUS il doit être relégué dans une couche inférieure.

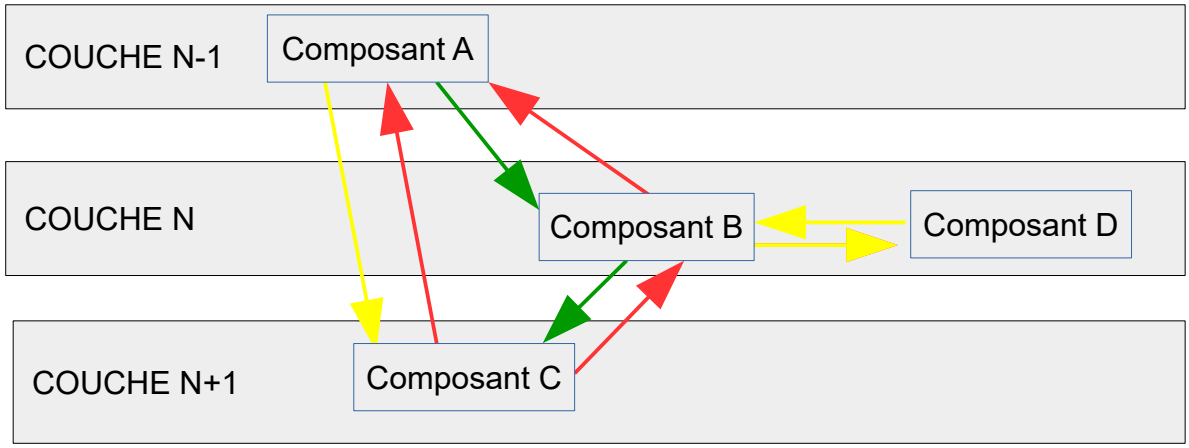
EXEMPLE: *dans une application bancaire, un module réalisant un transfert de compte à compte (opération spécifique est plus spécifique qu'un module modifiant le solde d'un compte dans une base de données, qui lui-même est plus spécifique qu'un module lisant un fichier sur un disque dur.*

RÈGLE N° 2 (interactions): En principe, une COUCHE ne peut interagir qu'avec une couche ADJACENTE: la couche supérieure envoie une REQUÊTE à la couche inférieure, qui exécute le traitement demandé et renvoie un MESSAGE DE COMPTE-RENDU (qui peut contenir des données). Ce fonctionnement correspond au modèle CLIENTS-SERVEUR: le client (couche supérieure) active un serveur (couche inférieure). Celui-ci lui répond par un SERVICE (par exemple, l'émission de données vers la périphérie), puis peut retourner un compte rendu d'exécution (par exemple, les données acquises par la lecture d'un périphérique). Un composant ne peut donc pas interagir avec un composant de sa propre couche.

REMARQUE: des transgressions à ces principes sont admises dans certains cas (flèches jaunes):

- Une couche N peut communiquer directement avec la couche N+2 si l'application stricte de la règle exigerait de développer dans N+1 un composant vide (c'est à dire un composant qui ne fait que relayer les requêtes et réponses entre N et N+2);
- La conception par objets s'accommode mal de l'interdiction de communiquer entre deux modules d'une même couche. De ce fait, cette interdiction est parfois transgressée dans les modèles basés sur ce style de conception (voir le pattern MVC).

Le schéma suivant résume ces principes:



Principes de la structuration en couches

COMMENTAIRES:

- Les flèches représentent la relation "UTILISER". Il peut s'agir d'un appel procédural classique (à l'intérieur d'une même machine) ou d'un appel de procédure ou méthode à distance de type CORBA, DCOMP, etc. Cette généralisation permet de traiter les cas où les modules sont localisés dans des machine différentes;

- Les flèches vertes représentent les requêtes autorisées;
- Les flèches jaunes représentent les requêtes qui sont parfois tolérées;
- Les flèches rouges représentent les requêtes strictement interdites;
- Les réponses à ces requêtes ne sont pas représentées dans ce schéma. Tout logiciel recevant une requête autorisée peu y répondre.

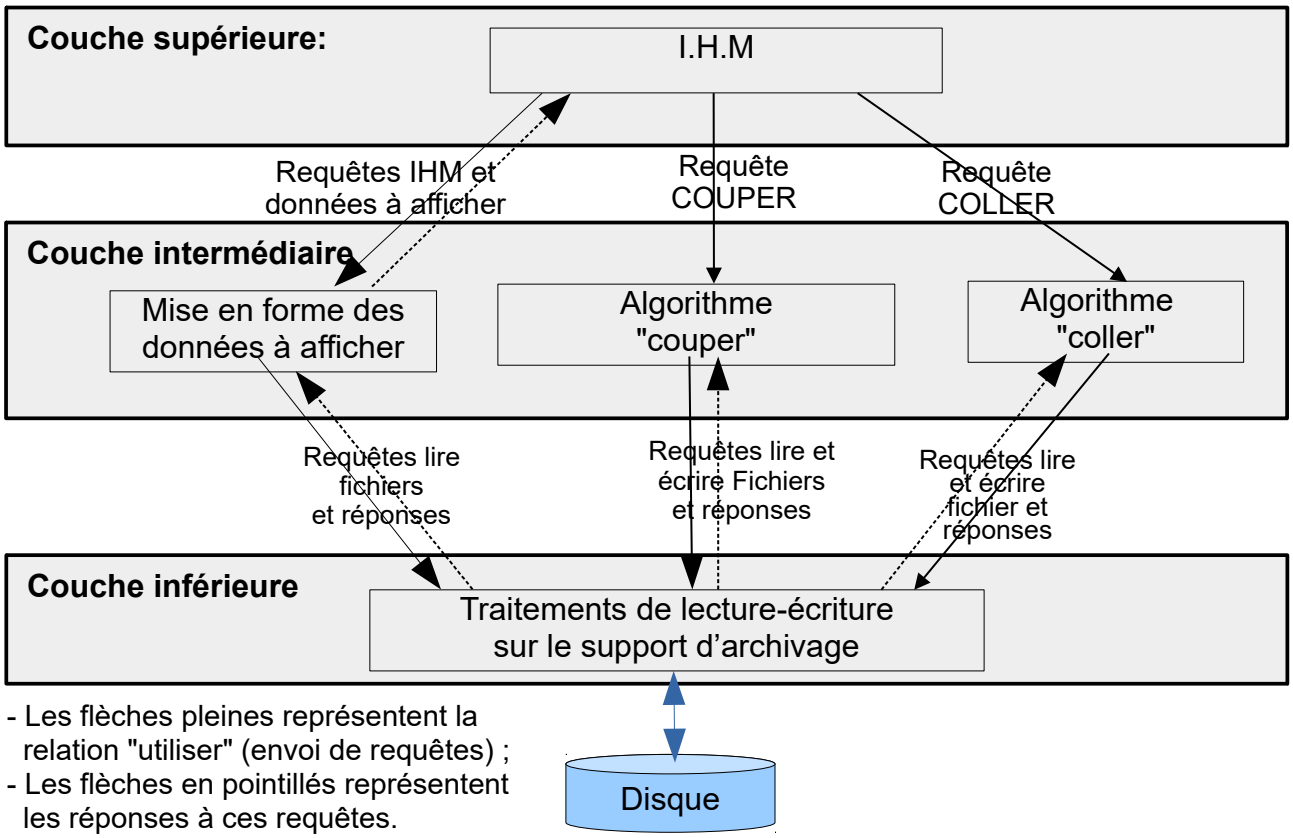
III.3.3.AVANTAGES DE LA STRUCTURATION EN COUCHES:

La structuration en couches permet :

- **De maîtriser la complexité des algorithmes**: l'imposition de règles strictes dans les interactions entre composants permet de limiter la complexité du "graphe des appels" en imposant une structure hiérarchique et en évitant les rebouclages de dépendances;
- **De favoriser la communication à l'intérieur d'une application**, en structurant les échanges entre les différentes couches.
- **D'optimiser la durée du développement** en faisant ressortir des opportunités de réutilisation;
- **De rationaliser et d'unifier l'architecture:** chaque couche produit des services pour la couche supérieure, éventuellement en utilisant pour cela la couche inférieure (communication de type Client-Serveur);
- **La conception d'architectures OUVERTES, facilitant les évolutions:** par exemple, si l'on veut changer de technologie de bases de données, il suffit d'adapter la couche qui gère les accès aux S.G.B.D.

III.3.4.EXEMPLE DE STRUCTURATION EN COUCHES:

L'exemple suivant représente une partie de la structuration en couches d'un logiciel de traitement de textes:



Structuration en couches - Exemple d'un éditeur de textes.

III.3.5.MODÈLES DE CONCEPTION EN COUCHES:

Il existe plusieurs modèles de décomposition en couches logicielles, chacun d'eux étant adapté à un type d'application et au degrés de complexité de cette application. Nous allons étudier plus particulièrement les modèles à 3 et 5 couches.

III.3.5.1.MODÈLE A TROIS COUCHES:

C'est le modèle MINIMUM. Il peut très bien convenir à une application de volume faible à moyen, faiblement ou pas du tout répartie.

MODÈLE 3 COUCHES	TRAITEMENTS ASSOCIES
PRÉSENTATION	• Affichage de l'interface graphique utilisateur et gestion des I.H.M; • Interception des événements utilisateurs et appel des traitements de contrôle de ces événements; • Gestion des impressions.
MÉTIER	• Gestion des sessions, habilitations, droits d'accès, gestion des erreurs. • Contrôle et gestion des événements utilisateurs; • Elle implémente la logique "métier" de l'application; • Elle assure la sécurité de l'application; • Elle assure le déroulement et la sécurité des transactions.
DONNÉES ou PERSISTANCE	• Elle gère les données persistantes de l'application (En particulier, les Bases de Données).

NOTA: *Il est quelquefois pratique d'adopter le modèle à 3 couches comme "architecture provisoire", en début de conception, car les détails des traitements à effectuer et les contraintes liées à ces traitement n'ont à cet instant de l'étude, pas encore été suffisamment analysées. Cela ne pose pas de problème dans la mesure où dans les modèles à plus de 3 couches, les couches supplémentaires sont en fait des subdivisions de la couche MÉTIER du modèle à trois couches.*

III.3.5.2.MODÈLE A CINQ COUCHES:

Ce modèle est probablement le plus utilisé. La couche MÉTIER du modèle à 3 couches est dans ce cas subdivisée en trois couches: CONTRÔLE, SERVICE et DOMAINE: cette subdivision permet une approche plus fine de la complexité de certaines applications et favorise la réutilisation des composants d'un domaine professionnel donné.

COUCHE	TRAITEMENTS ASSOCIES
PRÉSENTATION	• Affichage de l'interface graphique utilisateur; • Gestion des I.H.M utilisateurs; • Interception des événements utilisateurs et appel des traitements de contrôle de ces événements; • Gestion des impressions.
CONTRÔLE (ou COORDINATION)	• Gestion des sessions, habilitations, droits d'accès, gestion des erreurs. • Contrôle et gestion des événements utilisateurs;
SERVICES	• Elle implémente la logique "métier" de l'application (Services "métier", enchaînements des règles métier). Ex: virement de compte à compte, souscription d'un prêt en ligne, etc; • Elle assure la sécurité de l'application; • Elle assure le déroulement et la sécurité des transactions liées au métier.
DOMAINE	• Elle implémente les services de base du domaine d'activité utilisés par la logique "métier". Ex: La gestion d'un compte bancaire et un service de base du métier bancaire qui peut être invoqué par la couche SERVICES dans le cadre d'un transfert de compte à compte.
DONNÉES (ou PERSISTANCE)	• Elle gère les données persistantes de l'application (Fichiers, bases de données).

EXEMPLE D'APPLICATION STRUCTURABLE EN 5 COUCHES:

L'étude des applications liées au domaine bancaire (les logiciels de banque en ligne, par exemple), montre que les traitements liés au MÉTIER de la banque peuvent être classés en trois catégories collaborant entre elles:

1. Les traitements gérant les accès des utilisateurs (ouverture et fermeture de session, identification, gestion des droits d'accès, etc.);
2. Les traitements concernant les TRANSACTIONS DE COMPTE À COMPTE (transferts de fonds internes, virements externes, etc.);
3. Les traitements gérant les COMPTES CLIENTS: versements, retraits, visualisation des opérations, etc.

la transaction consistant en un transfert de fonds d'un compte client A identifié vers un autre compte B sera contrôlé par un module de la catégorie 1, assurant le déroulement sécurisé de cette transaction et la reprise en cas d'erreur.

Le transfert de fond, qui appartient à la catégorie 2, s'appuie au moins sur deux traitements de la catégorie 3: un retrait depuis le compte A suivi d'un versement sur le compte B.

Les catégories 1, 2 et 3 correspondent donc aux couches CONTRÔLE, SERVICE et DOMAINE du modèle à 5 couches.

III.4.APPLICATION RÉPARTIE ET DÉPLOIEMENT D'APPLICATION:

III.4.1.DÉFINITION:

On appelle APPLICATION RÉPARTIE une application composée de modules logiciels indépendants et autonomes, pouvant être installés sur des unités de traitement distantes géographiquement, et coopérant sans notion de hiérarchie grâce à un réseau de communication qui peut être hétérogène du point de vue des matériels et des protocoles.

L'absence de hiérarchie entre composants implique que les fonctions de surveillance du fonctionnement, de détection des erreurs, d'ordonnancement des traitements, soient elles-même réparties entre les composants. Il y a donc coopération et "suspicion mutuelle" des composants.

L'hétérogénéité des supports matériels implique des mécanismes de communication de haut niveau pour assurer la transparence des différences de structure entre ces composants (mécanismes d'adaptation, de traduction des données, etc.. entre systèmes différents).

L'Autonomie des composants implique l'utilisation de mécanismes de synchronisation entre les divers processus logiciels supportés par des composants différents et distants.

III.4.2.NOTION DE DÉPLOIEMENT D'UNE APPLICATION:

Dans son acception la plus stricte, le terme DÉPLOYER, concernant une APPLICATION logicielle, est une ACTION qui consiste à INSTALLER les différents composants de cette application (fichiers, bases de données, exécutables, etc.) dans un environnement matériel donné, en les adaptant aux caractéristiques de cet environnement (en général, d'une manière automatique, par l'utilisation de procédures de déploiement).

Le plus souvent, une application est, dans un premier temps, conçue et mise au point sur une "plate-forme de développement" appartenant à l'entreprise en charge de la création de cette application. Cette plate-forme est constituée d'un certain nombre d'unités de traitement reliées par un réseau local, lui-même comprenant des systèmes de liaison (filaires, optiques, wifi, etc;) et des unités de routage).

Lorsque cette application est terminée (développée et testée "en usine"), elle peut être installée chez les différents clients qui s'en portent acquéreurs.

Le déploiement d'une application sur une plate-forme donnée est en général effectué au moyen d'un certain nombre de procédures automatiques (PROCÉDURES DE DÉPLOIEMENT) dont le but est de transférer les différents composants de cette application de la plate-forme de développement vers la plate-forme cible.

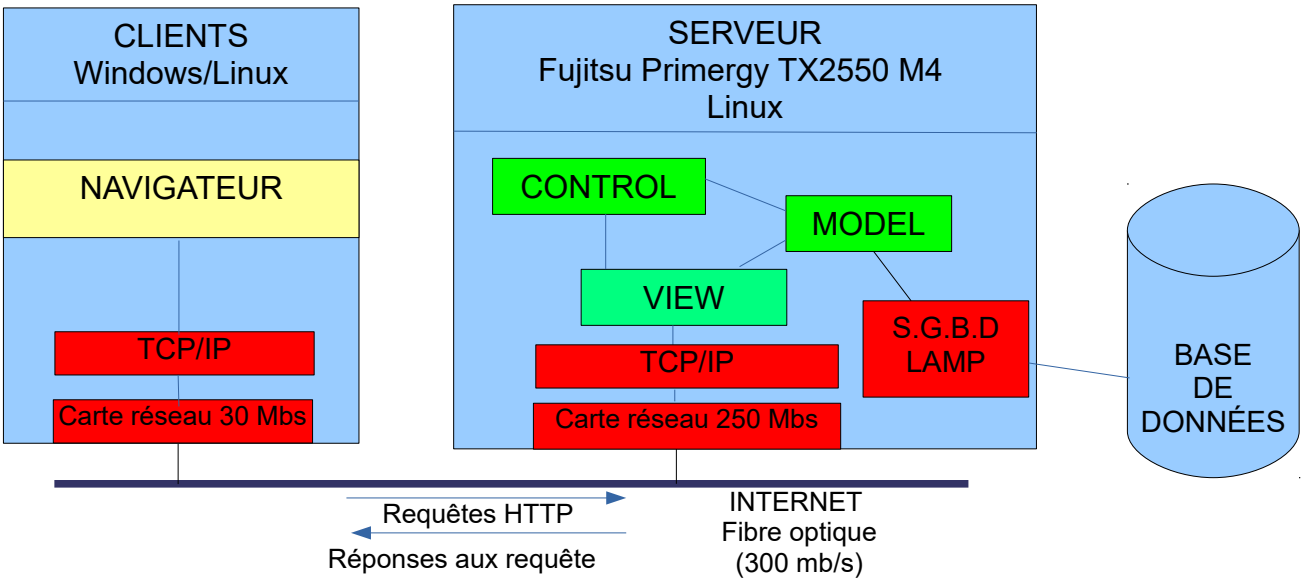
Les plates-formes d'accueil des différents clients pouvant être différentes de la plate-forme de développement (matériels, systèmes d'exploitation, etc.), ces transferts

s'accompagnent le plus souvent d'adaptations (recompilation des sources, changement de représentation des données, etc.).

Par extension, le DÉPLOIEMENT d'une application peut désigner la manière dont une application logicielle est RÉPARTIE sur sa plate-forme d'accueil. C'est dans ce sens là que la notion de déploiement est la plus utile en phase de conception.

EXEMPLE:

Le schéma ci-dessous peut représenter le déploiement d'une application répartie sur une plate-forme comprenant un serveur et des clients HTTP:



III.4.3.LES DIAGRAMMES DE DÉPLOIEMENT U.M.L:

Le langage de modélisation U.M.L définit des diagrammes de déploiement assez formalisés qui permettent de modéliser les applications suivant la "vue de déploiement".

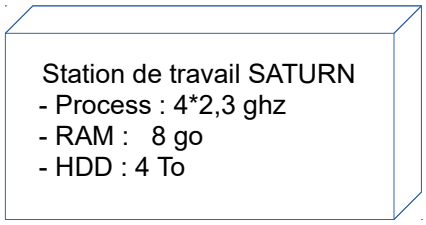
Ces diagrammes permettent de décrire à la fois les ressources matérielles et la répartition "spatiale" du logiciel dans ces ressources. Plus précisément, ils font apparaître:

- La disposition et la nature physique des matériels qui composent l'infrastructure d'accueil (unités de traitement, liaisons physiques, routeurs, etc) ainsi que les performances de ces matériels;
- L'implantation des principaux modules logiciels sur les unités de traitement et les dépendances existant entre eux;
- Les exigences en terme de performances (temps de réponse, fiabilité, tolérance aux fautes, M.T.B.F, etc.).

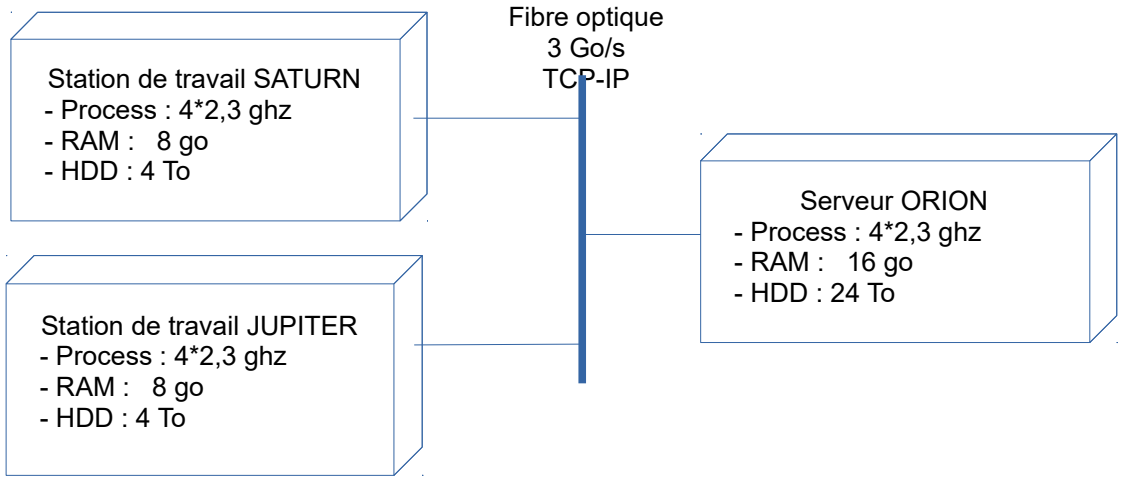
PRINCIPALES RÈGLES DE REPRÉSENTATION:

Un diagramme de déploiement U.M.L se présente comme un réseau dont les nœuds sont les différentes unités de traitement et les arcs représentent les "chemins de communication" entre les nœuds.

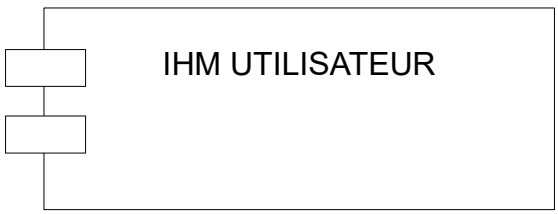
Les **nœuds** sont représentés par des parallélépipèdes rectangles. Ils possèdent un nom et des attributs qui caractérisent leurs performances. Exemple:



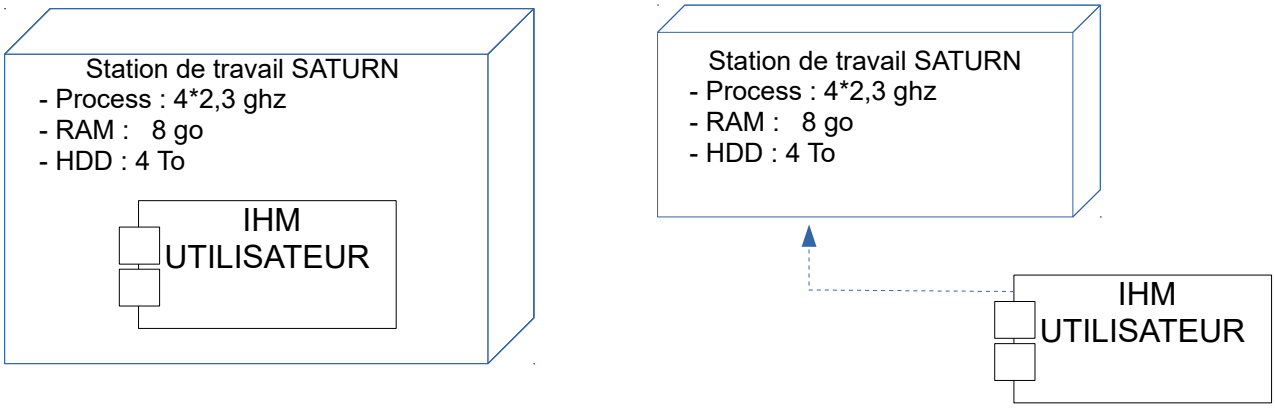
Les **chemins de communication** sont représentés par des lignes auxquelles peuvent être associés différents attributs (nature du média, protocole de communication, débit, etc):



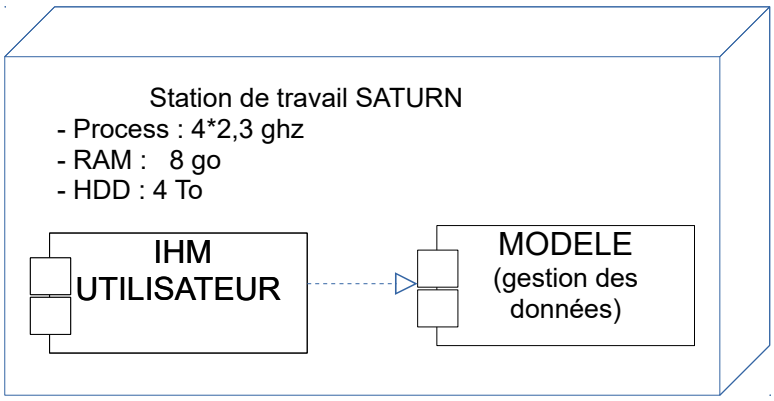
Les **composants logiciels** de l'application sont représentés par des rectangles auxquels sont adjoints deux petits rectangles en bordure. EXEMPLE:



Les **composants logiciels** peuvent être représentés à l'intérieur des nœuds ou à l'extérieur:



Les **relations de dépendances entre composants logiciels** peuvent être représentées par des flèches pointillées à pointe évidée:



III.5.LA CONCEPTION VUE AU NIVEAU DU SYSTÈME INFORMATIQUE:

III.5.1.PRISE EN COMPTE DE L'ENVIRONNEMENT DE L'APPLICATION:

La conception d'une APPLICATION LOGICIELLE doit nécessairement tenir compte de l'environnement qui va la supporter. Cet environnement est à la fois PHYSIQUE (c'est l'INFRASTRUCTURE d'accueil), mais aussi LOGICIEL (ce sont les système d'exploitation des machines hôtes et les logiciels de traitement des protocoles d'échange qui, avec l'infrastructure, constituent la PLATE-FORME d'accueil).

III.5.2.RÉPARTITION D'UNE APPLICATION :

Une grande partie des projets de développement ou d'évolution de logiciels concerne des applications S'EXÉCUTANT SUR UNE SEULE MACHINE équipée d'un certain type de système d'exploitation. C'est le cas, par exemple, du développement d'un pilote de périphérique sous Linux, d'une application pour téléphone portable Android ou d'un site web sur un serveur Apache. Dans ce cas, l'environnement matériel et logiciel doit être considéré comme une CONTRAINTE DE CONCEPTION.

D'autres projets, en général plus volumineux et complexes, concernent l'élaboration ou l'évolution d'APPLICATIONS RÉPARTIES sur plusieurs unités de traitement dans l'infrastructure d'accueil. La détermination de cette répartition fait partie, dans ce cas, des travaux de conception globale.

III.5.3.COMMUNICATION DANS UNE APPLICATION RÉPARTIE:

III.5.3.1.MODÈLES DE COMMUNICATION:

Les mécanismes de la communication entre deux partie d'une application situées sur des unités de traitement différentes appartiennent souvent à deux modèles principaux: le modèle CLIENTS-SERVEUR et le modèle PAIR À PAIR (souvent noté P2P).

Ces modèles sont surtout utilisés dans le cadre des applications et services WEB. En revanche, les applications industrielles (conduite de processus en temps réel, automatismes, applications embarquées, simulation, etc) les utilisent beaucoup moins car elles satisfont mal aux contraintes de délais et de déterminisme liées à leur domaine.

III.5.3.2.LE MODÈLE CLIENTS-SERVEUR:

PRINCIPE:

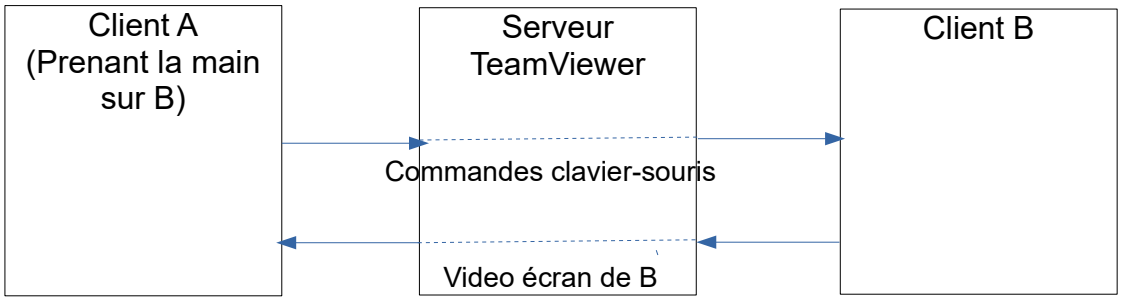
Ce modèle est caractérisé par la dissymétrie des rôles des logiciels communicants: un SERVEUR reçoit des requêtes en provenance d'un certain nombre de CLIENTS. Il répond à ces requêtes en lui fournissant le SERVICE sollicité par la requête. Ce service peut être:

- ***La fourniture d'informations au client:*** ces informations peuvent être le contenu d'une page web (serveur HTTP), d'un fichier de données (serveur FTP), d'une liste de mails (serveur POP), ou même des données correspondant à l'exécution d'un traitement particulier (acquisition des données d'un capteur, calcul de prêts, etc;
- ***L'exécution d'un traitement:*** enregistrement d'un fichier sur le serveur (serveur FTP), mise à jour d'une table dans une base de données (serveur de bases de données), etc.

Les CLIENTS sont des entités éphémères: ils sont crées, puis sollicitent le serveur après s'être connectés et peuvent disparaître dès que leur rôle est terminé. Les SERVEURS, en revanche, sont des entités permanentes dotées d'une adresse logique fixe.

Le modèle CLIENTS-SERVEUR est un exemple d'architecture répartie CENTRALISÉE, car dans ce modèle, le SERVEUR (unique et non éphémère) est au centre d'un schéma de communication en étoile dont les branches sont les liaisons entre clients et serveur;
Le serveur, de par sa position, est à même de relayer les informations émises par un client vers un autre client.

EXEMPLE: un exemple de ce fonctionnement est l'application TeamViewer:



Le principe de fonctionnement de cette application est le suivant:

- Chacun des clients se connecte au serveur qui en retour leur transmet un identificateur et un mot de passe;
- Un client A qui veut "prendre la main" sur la machine d'un client B demande à celui-ci (par téléphone, mail, sms, etc.) ses données de connexion, puis les transmet au serveur par l'intermédiaire de l'interface client;
- Le serveur relaie alors en permanence la vidéo d'écran du client B vers le client A pour que celui-ci l'affiche sur l'écran de A dans une fenêtre. En même temps, les événements clavier et souris créés sur A sont relayés vers B et substitués aux événements clavier et souris locaux.

REMARQUES:

- Le protocole TCP établit de fait une communication de type CLIENTS-SERVEUR. En effet, les bibliothèques de fonctions TCP/IP ne permettent au développeur de créer que deux sortes de connexions: des connexions de type CLIENT (qui doivent, en préalable à toute autre action, se connecter à un serveur TCP/IP la fonction "connect") et des connexions de type SERVEUR (qui traitent les requêtes de connexion par la fonction "accept"). Pour autant, le modèle CLIENTS-SERVEUR n'implique pas forcément l'utilisation de TCP/IP: certaines applications (comme FTP) utilisent U.D.P en gérant elles-mêmes les mécanisme de connexion et de fragmentation/défragmentation).
- Le modèle CLIENTS-SERVEUR s'applique à des PROCESSUS LOGICIELS communicants et non à des machines. Une même machine peut abriter des clients et des serveurs. Ce n'est que par abus de langage qu'un poste de travail est qualifié de CLIENT ou de SERVEUR: il ne l'est que relativement à une application donnée.

UTILISATION:

Le modèle CLIENTS-SERVEUR, est très utilisé par les applications gérant des TRANSACTIONS (banques) où celles qui peuvent être utilisées par un grand nombre d'utilisateurs volatiles (qui peuvent se créer et disparaître d'une manière aléatoire, comme les sites web). En revanche, il convient assez mal au traitement de mesures ou à la conduite de processus.

III.5.3.3.LE MODÈLE PAIR À PAIR (P2P):

PRINCIPES:

Ce modèle correspond à une architecture RÉPARTIE DÉCENTRALISÉE (ou DISTRIBUÉE): Les processus distants communiquent entre eux directement, sans passer par un élément central. Chacun de ces processus, relativement à un échange donné, peut jouer les rôles de CLIENT ou de SERVEUR où encore les deux rôles en même temps.

Dans une architecture distribuée, les NŒUDS qui la composent sont en relation d'égal à égal (d'où l'expression PAIR A PAIR). On utilise parfois le terme SERVENT pour les désigner. Chacun des nœuds peut héberger tout ou partie des informations gérées par l'application.

D'autre part:

- Il n'existe aucun système ou organe de coordination globale;
- Tous les nœuds d'une application répartie en P2P peuvent être VOLATILES. Il n'existe pas de nœud permanent comme peuvent être les serveurs d'un système Clients/serveur.

- Chacun des nœuds du réseau ne connaît que les nœuds qui lui sont voisins dans le réseau;
- En revanches, toutes les informations distribuées sont accessibles à partir de n'importe quel nœud.
- Les informations échangées entre deux nœuds sont souvent nommées OBJETS. Ces informations peuvent être des fichiers ou encore des flux de données comme la VoIP ou le steaming video.

FONCTIONNEMENT:

- Chacun des nœuds participant à l'application est équipé du même logiciel P2P;
- Lorsqu'un nœud A souhaite accéder à un OBJET qui n'est pas hébergé par lui-même, il émet vers ses voisins une requête.
  - Si l'un de ces voisins héberge l'objet en question, il répond à A en lui renvoyant l'objet en question (en fait, une copie, s'il s'agit d'un fichier).
  - Si le voisin B n'héberge pas l'objet demandé, il émettra vers ses voisins une requête, et ainsi de suite, la requête sera répétée de proche en proche jusqu'à ce que l'objet ait été trouvé;
  - La copie de cet objet est alors renvoyée vers A en suivant le chemin inverse.
  - ***NOTA:*** Ce fonctionnement est très proche de celui du service des noms de domaines (DNS) d'internet.
- Le protocole de transport est majoritairement TCP (protocole connecté), mais certaines applications utilisent UDP (en traitant elles-même le contrôle) ou RTP (Real-Time Transport Protocol – Application utilisant VoIP).

AVANTAGES:

- Cette architecture permet de d'utiliser au mieux les ressources de la plate-forme. En effet, tous les nœuds mettent à disposition de l'application leurs ressources propres (puissance CPU, volume de stockage, bande passante, etc.). Ceci permet de répondre au mieux aux "pics de charge" (mieux qu'une architecture clients-serveur où les nœuds sont spécialisés);
- La distribution et l'absence de nœud "central" permet d'augmenter la robustesse de l'application en cas de panne d'une partie des nœuds, à condition que les données soient en permanence répliquées sur plusieurs nœuds.

INCONVÉNIENTS:

- Le "P2P" et très associé aux applications de téléchargement illégal car sa structure complique les investigations des autorités. En revanche, il est peu utilisé dans les applications industrielles;

- Si le P2P permet d'optimiser l'utilisation des ressources et de résister aux pannes et aux investigations, la fragmentation des données et le mécanisme d'accès à ces données dégradent sérieusement les temps de réponse;
- Il est pratiquement impossible de contrôler globalement le fonctionnement de l'application et les accès aux données.

UTILISATION:

Les inconvénients énumérés ci-dessus expliquent que le P2P soit peu utilisé en informatique industrielle. La plupart des applications concernent les partages de fichiers (musique, vidéo) ou de streaming, le calcul distribué, ou les jeux vidéo multijoueurs.

III.5.3.4.LA COMMUNICATION DANS LES APPLICATIONS INDUSTRIELLES:

III.5.3.4.1. CARACTÉRISTIQUE FONDAMENTALES D'UNE APPLICATION INDUSTRIELLE:

Les applications industrielles ont pour objet principal de CONTRÔLER un PROCESSUS INDUSTRIEL: la fabrication d'un produit, la production d'électricité dans une centrale (thermique, nucléaire, éolienne), le pilotage automatique d'un drone, etc.

Le CONTRÔLE d'un processus implique de prendre en compte "en temps réel" l'ÉTAT COURANT de ce processus et d'en déduire les COMMANDES qui doivent lui être appliquées pour que l'évolution de son état reste conforme à ce que l'on attend de lui (la "consigne").

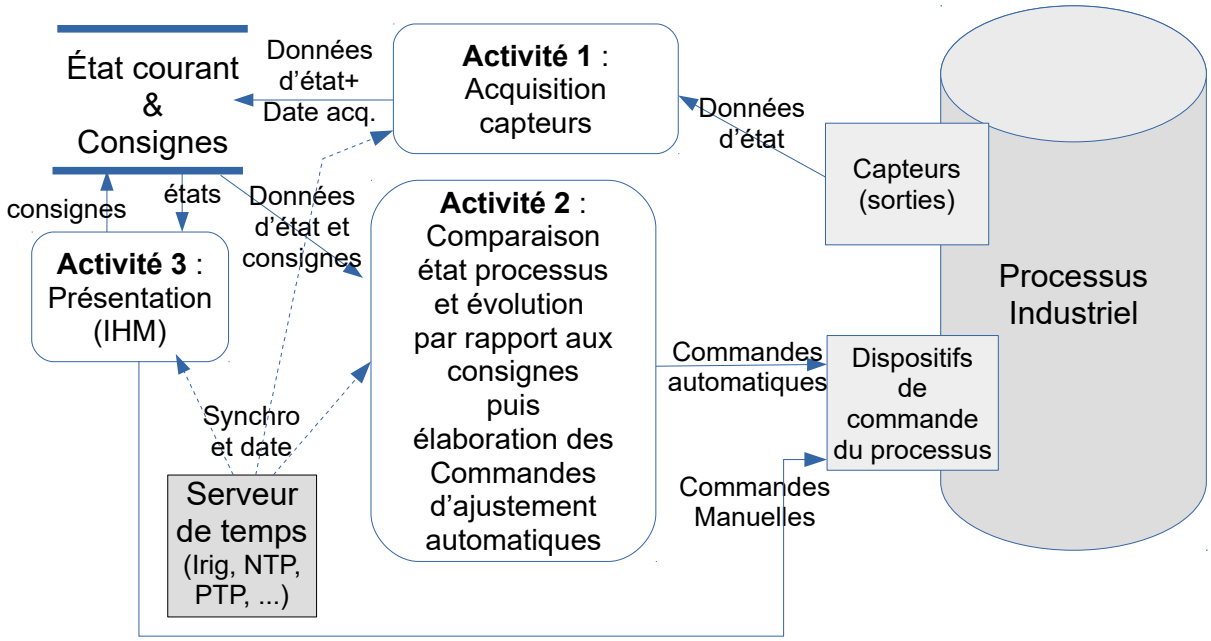
Cette activité peut être assurée automatiquement, par le logiciel de l'application, ou manuellement, par un opérateur utilisant une Interface Homme-Machine. Cependant, le contrôle de processus industriels complexes exige une rapidité d'analyse et de réaction que ne peut fournir un opérateur humain. Dans ce cas, le mode automatique est le seul qui puisse assurer une réaction "en temps réel" assez rapide, l'opérateur humain se contentant de saisir des "consignes".

D'autre part, le contrôle d'un processus industriel exige très souvent une synchronisation précise des traitements avec le temps universel. Ceci implique de disposer de moyens de datation et de signaux horaires beaucoup plus précis que ceux qui sont délivrés par les fonctions classique des systèmes d'exploitation (date, time, etc.). De ce fait, une installation industrielle comprend très souvent un "serveur de temps" capable de fournir aux applications les informations nécessaires pour se synchroniser d'une manière fine:

- Une datation à la milliseconde près par rapport au temps universel;
- Des signaux horaires (1 Hz, 10 hz, etc.) calés sur cette datation.

Ces serveurs utilisent diverses sources de synchronisations telles que Inter-Range Instrumentation Group (IRIG B IRIG H, etc.), Precision Time Protocol (PTP), Network Time Protocol (NTP).

Le schéma ci-dessous modélise le fonctionnement général d'une telle application:



COMMENTAIRES:

- Les DONNÉES D'ÉTAT sont les valeurs instantanées de certaines grandeurs caractéristique du fonctionnement du processus (température, pression, position GPS, etc). L'état du processus est l'ensemble de ces valeurs;
- En général, la fonction principale de l'application consiste à piloter le processus industriel de façon à ce que son état instantané reste le plus proche possible des CONSIGNES définies par les utilisateurs. Pour ce faire, l'application acquiert périodiquement les valeurs d'état instantanées du processus, les compare aux consignes et en déduit automatiquement des commandes d'ajustement qu'elle applique aux dispositifs de commande du processus (il s'agit donc d'un "CONTRÔLE EN BOUCLE FERMÉE"). Ces traitement sont supportés par les activités 1 et 2;
- L'activité 3 (I.H.M) permet de visualiser en permanence l'état du système et quelquefois de prévoir les évolutions à plus ou moins long terme. L'IHM permet également de définir ou modifier les consignes ou encore de commander manuellement le processus.
- Le processus de contrôle peut être:
  - Entièrement automatique (l'application réagit automatiquement aux évolutions d'état en tenant compte de "consignes" saisies au préalable par les opérateurs);

- Entièrement manuel (grâce àl' IHM, un opérateur peut surveiller l'état courant et piloter le processus par des commandes manuelles);
- Mixte (L'application possède un mode automatique et un mode manuel).

REMARQUES:

- La notion de "consigne" est ici beaucoup plus extensive qu'elle ne l'est en automatisme. Si une consigne peut être seulement une valeur à respecter (température, pression), il peut s'agir aussi d'une entité beaucoup plus complexe: une trajectoire à suivre (guidage d'un véhicule), une suite d'opérations à effectuer (conduite d'un processus), la description d'un comportement, etc.
- Très souvent, le terme CONTRÔLE-COMMANDE est utilisé plutôt que CONTRÔLE pour exprimer cette activité).

III.5.3.4.2.L'ACQUISITION DES DONNÉES D'ÉTAT:

Ces données d'état sont fournies par des CAPTEURS intégrés au processus industriel. A ces capteurs sont associés des dispositifs électroniques plus ou moins complexes qui permettent de récupérer les mesures du capteur et de les injecter sous forme digitale sur un média adapté (liaison digitale point à point, réseau, liaison wifi, etc.). Ces dispositifs d'interfaçage sont parfois appelés SERVEURS D'ACQUISITION (surtout quand ils interfacent un réseau).

REMARQUES:

- Nous pourrons ainsi avoir affaire à des capteurs de température, de pression, de vitesse (anémomètres, tachymètres), de position et vitesse (GPS), etc.
- La liaison n'est pas forcément unidirectionnelle car beaucoup de capteurs peuvent être paramétrés à distance (échelle, unités, fréquence de fourniture des mesures, etc.).

III.5.3.4.3.LA COMMANDE DU PROCESSUS:

Un processus industriel intègre un certain nombre de dispositifs qui permettent d'influer sur le déroulement de ce processus.

Tous ces dispositifs, lorsqu'ils sont connectés à un dispositif d'interfaçage permettant de les régler à distance par l'intermédiaire d'une liaison digitale, permettent de COMMANDER le processus à partir d'une application.

REMARQUE:

Nous pourrons rencontrer:

- Des vannes qui permettent de contrôler un débit de fluide;
- Des soupapes qui permettent de faire baisser une pression;
- Des commutateurs qui permettent par exemple de sélectionner un mode de fonctionnement;

- Des rhéostats qui permettent de régler des tensions électriques;
- Des dispositif de contrôle de la température (climatisation réglable);
- Des ailerons hypersustentateurs, des gouvernes de direction ou de profondeur, etc. (si le processus et un avion télécommandé.);
- Etc.

III.5.3.4.4.LES ÉCHANGES D'INFORMATION DANS UNE APPLICATION INDUSTRIELLE:

En général, les échanges d'informations qui ont lieu dans la BOUCLE DE CONTRÔLE (acquisition des états du processus, comparaison avec la consigne, élaboration et émission des commande vers le processus) concernent des messages COURTS (quelques centaines d'octets), soumis à des CONTRAINTES ASSEZ STRICTES concernant la DURÉE de la boucle et le DÉTERMINISME de cette durée et surtout les dates de délivrance de ces messages.

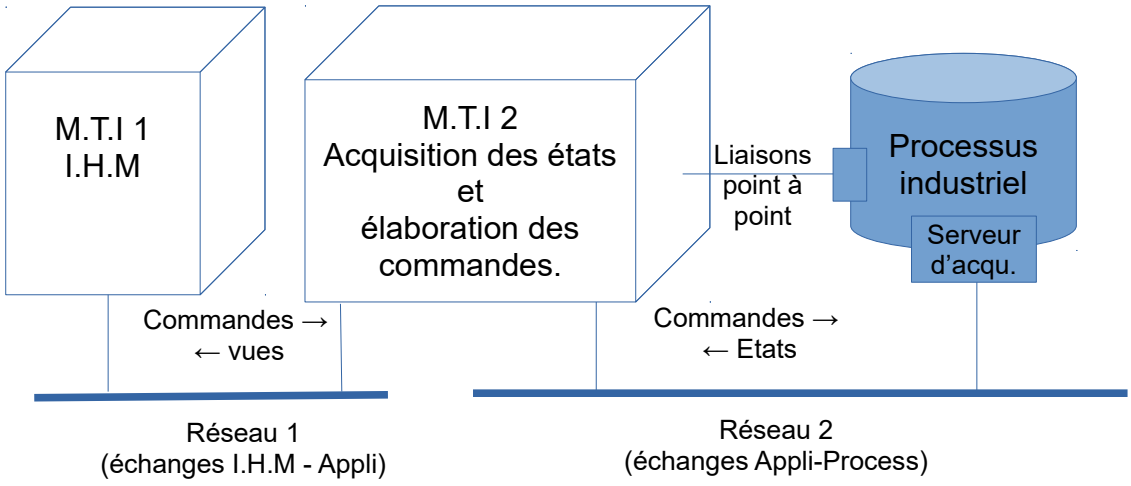
EXEMPLE DE CONTRAINTE: *la réaction du logiciel à l'émission d'un nouvel état du processus doit parvenir à ce processus 50 ms ± 2 ms après cette émission (la durée d'exécution la boucle de contrôle doit être de 50 ms à 2 ms près).*

Ces contraintes, caractéristiques des applications fonctionnant en TEMPS RÉEL, ne peuvent être respectées que si les durée des échanges d'acquisition des états et d'émission des commandes sont elles-même suffisamment déterministes.

De ce fait, il est difficile de faire cohabiter ces échanges très contraint avec d'autres échanges impliquant des messages beaucoup plus longs, circulant de manière apériodique et pouvant donner lieu à des réémission, comme c'est le cas pour les échanges entre un client et un serveur HTTP. Ces considérations justifient très souvent la séparation physique des médias transportant ces deux catégories d'échange.

Les solutions peuvent être:

- Soit d'utiliser pour les liaisons à contrainte temps réel des liaisons physiques point à point dédiées, à plus ou moins haut débit, utilisant des protocoles tels que HDLC, PPP, SLIP, ou encore RS432;
- Soit d'utiliser deux (ou plusieurs) réseaux de communication séparés. Le schéma ci-dessous illustre cette solution:

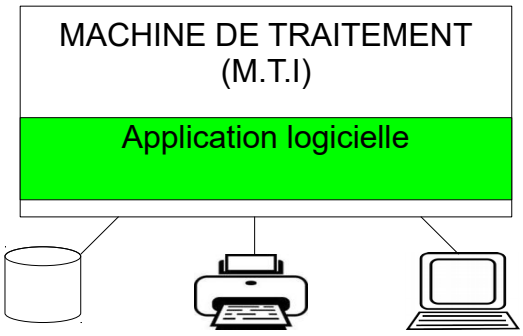

M.T.I 1
I.H.M
M.T.I 2
Acquisition des états
et
élaboration des
commandes.
Liaisons
point à
point
Processus
industriel
Serveur
d'acqu.
Commandes →
← vues
Commandes →
← Etats
Réseau 1
(échanges I.H.M - Appli)
Réseau 2
(échanges Appli-Process)

III.6.ACTIVITÉ DE CONCEPTION ET LOGICIELS RÉPARTIS:

III.6.1.INTRODUCTION:

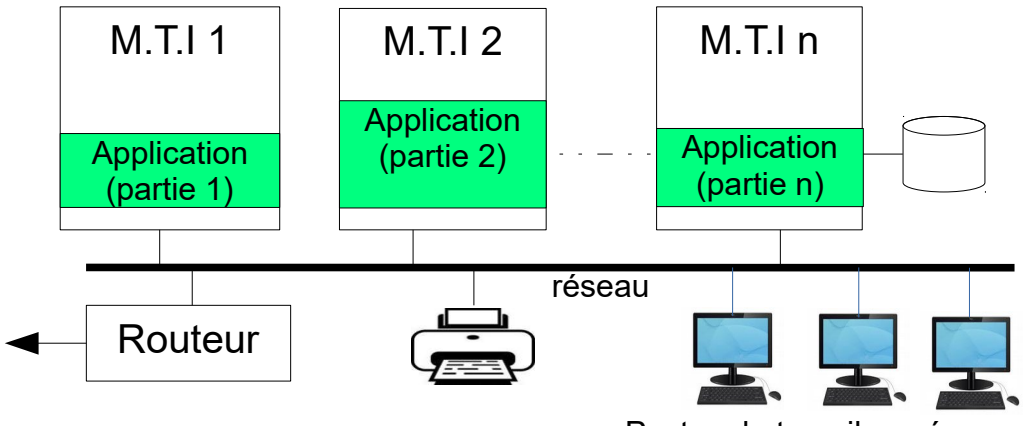
REMARQUE PRÉLIMINAIRE: *dans le texte, le sigle M.T.I est employé pour désigner une Machine de Traitement Informatique, c'est à dire un équipement physique capable d'exécuter des programmes informatiques. Une M.T.I peut être mono ou multi processeurs.*

Au début de l'évolution de l'informatique (années 1950-1960), les systèmes informatiques étaient dans leur immense majorité, composés d'un seul ordinateur, connecté à divers périphériques. Les applications s'exécutaient donc dans une seule machine (Architecture MAIN FRAME):



Architecture "main frame" : l'application est supportée par une seule unité de traitement.

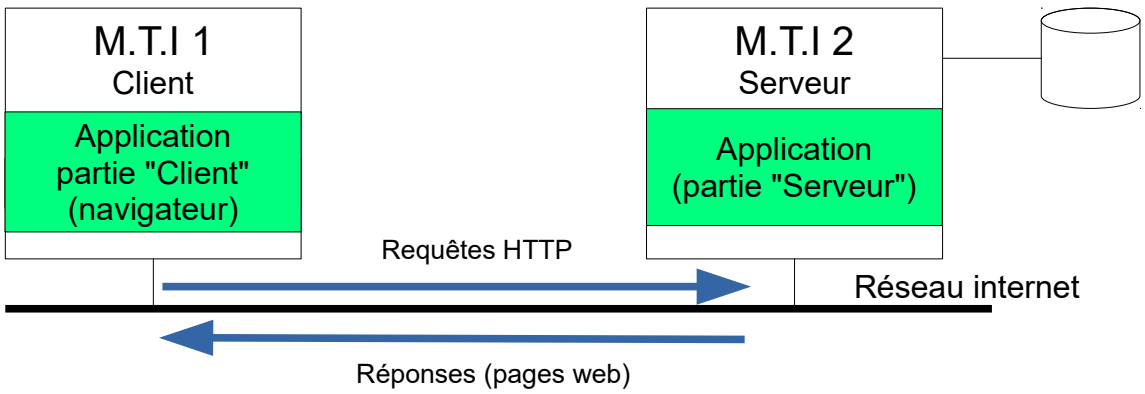
Avec l'apparition des RÉSEAUX LOCAUX et des RÉSEAUX ÉTENDUS (années 1980), l'architecture des systèmes informatique a évolué de plus en plus vers des SYSTÈMES dits RÉPARTIS, composés de plusieurs unités de traitement communicant entre elles par le réseau:



Architecture "Répartie" :
l'application est composée de plusieurs parties installées sur des unités de traitement différentes et communicant entre elles par le réseau.

EXEMPLE:

Les SITES WEB sont des exemples d'applications réparties: une application web est répartie au moins sur deux unités de traitement: le Client WEB (navigateur) et le Serveur WEB:



Exemple d'architecture "Répartie" : les site WEB

III.6.2.VUE LOGIQUE ET VUE EN ÉTAGES DE SERVICES:

De ce qui précède, il résulte que la conception architecturale d'un logiciel doit être abordée sous différents aspects:

- La VUE LOGIQUE de l'application, c'est à dire l'agencement logique des différentes ENTITÉS LOGICIELLES, indépendamment des considérations liées notamment à leur localisation physique dans le système informatique;
- La VUE EN ÉTAGES DE SERVICE (TIER VIEW), qui représente l'architecture PHYSIQUE, tenant compte de la RÉPARTITION des entités logicielles sur les différentes UNITÉS DE TRAITEMENT composant le réseau. Cette vue est proche de la vue de déploiement d'U.M.L.

III.6.3.LA VUE EN ÉTAGES DE SERVICES (TIER VIEW):

III.6.3.1.PRISE EN COMPTE DE L'ARCHITECTURE PHYSIQUE:

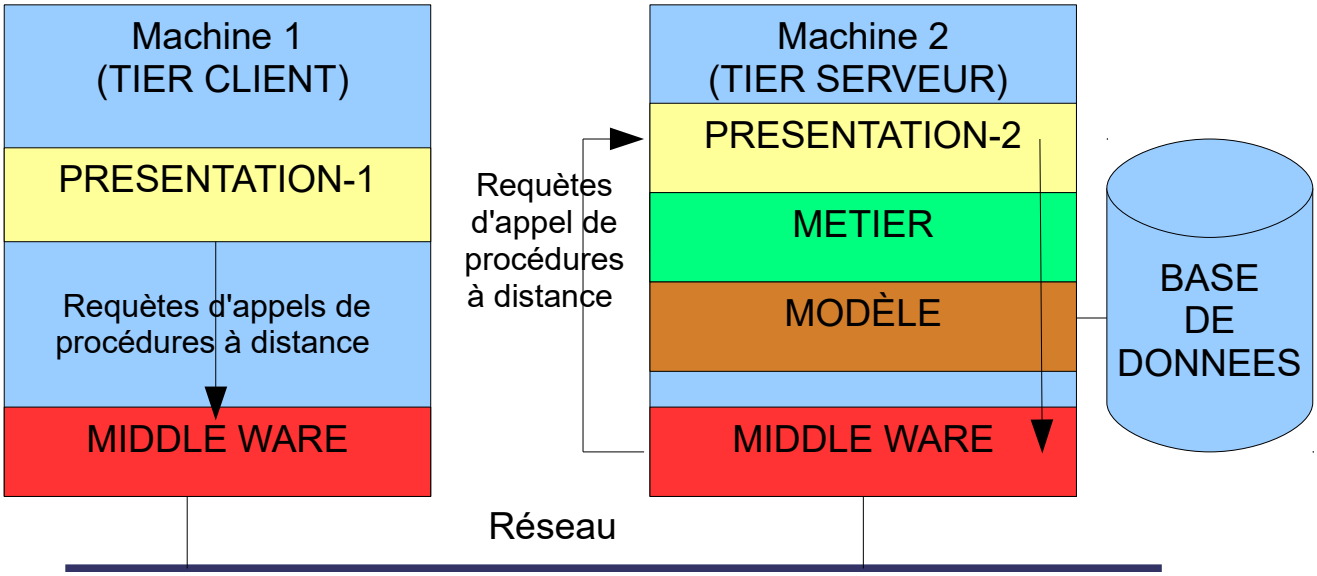
Lorsqu'une application est destinée à s'exécuter sur un seul ordinateur (architecture MAINFRAME), le point de vue LOGIQUE et la décomposition en couches qui en résulte suffisent pour définir l'architecture générale.

Dans le cas contraire (architecture RÉPARTIE), il faut prendre en compte la structure physique du système informatique, afin de RÉPARTIR AU MIEUX LES TRAITEMENTS sur différentes machines. Dans cette répartition, certaines couches logicielles peuvent se retrouver supportées par plusieurs machines. Les composants d'un logiciel qui concourent à réaliser un NIVEAU DE SERVICE donné constituent un TIER ou ÉTAGE.

La liaison entre les TIER est effectuée par des logiciels appelés MIDDLEWARES. Ceux-ci peuvent utiliser:

- Soit des ÉCHANGES DE MESSAGES (Ex: échange de requêtes HTTP);
- Soit des APPELS DE PROCÉDURES A DISTANCE (Ex: Un composant déclenche l'exécution d'une procédure dans la machine distante). C'est le cas de DCOM (Distributed Component Object Model) de Microsoft;
- Soit l'INVOCATION DE MÉTHODES A DISTANCE (Ex: Un composant déclenche l'exécution d'une méthode d'un objet situé dans la machine distante). C'est le cas de CORBA(Common Object Request Broker Architecture) ou RMI (Remote Method Invocation de Java).

EXEMPLE: architecture en 3 "couches" répartie sur 2 ordinateurs ou architecture 2 TIER (Client-Serveur):



Architecture en 3 couches sur 2-tier

III.6.3.2.REMARQUES SUR LA NOTION DE TIER:

- Nous pouvons constater dans le schéma précédent que la couche PRÉSENTATION est répartie sur les deux TIER (client et serveur). La notion de COUCHE est donc différente de la notion de TIER;
- On peut dire qu'un TIER désigne un NIVEAU (ou ÉTAGE) de service, un peu comme dans un bâtiment où chaque étage serait affecté à un certain type d'activités: par exemple, l'accueil au rez-de-chaussée, les finances au premier, les ressources humaines au deuxième, etc.

- Un logiciel donné peut être réparti sur plusieurs TIERs (de la même manière que le service ressources humaines d'une entreprise peut occuper plusieurs étages, si cette disposition permet une meilleure gestion).

III.6.3.3.CONCLUSION:

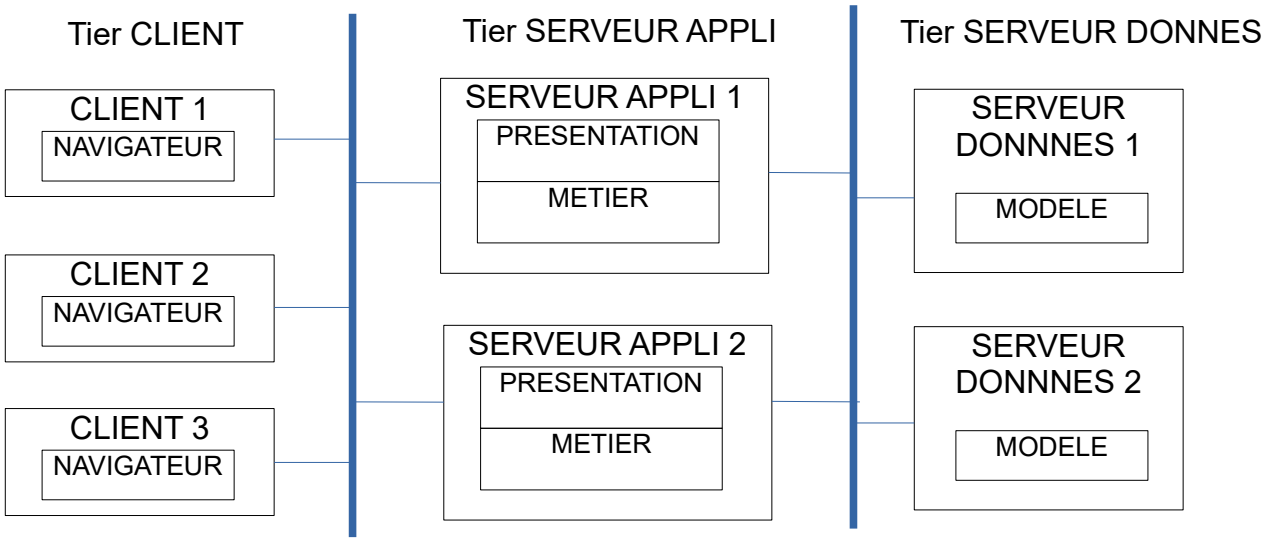
Bien que les deux notions soient souvent confondues dans la littérature, la notion de TIER est différente de la notion de COUCHE logique. Un TIER ne désigne ni une couche logicielle ni une machine particulière d'un système informatique. Il désigne plutôt "un ensemble de machines d'un S.I. qui jouent le même rôle pour une application donnée".

Nous pourrons trouver ainsi:

- Un tier CLIENTS, constitués de l'ensemble des postes de travail des utilisateurs d'une application (ces tiers hébergent l'interface homme-machine de l'application);
- Un tier SERVEURS D'APPLICATIONS qui hébergent les logiciels "métier" de l'application);
- Un tier SERVEURS DE DONNÉES qui hébergent les S.G.B.D de l'application;
- Un tier SERVEUR D'ACQUISITION, qui permet d'acquérir les données d'un capteur et de les injecter sur un réseau;
- etc.

Un TIER peut donc être représenté par plusieurs M.T.I (par exemple, l'ensemble des CLIENTS d'une application de type CLIENT-SERVEUR constituent un seul et même TIER).

D'autre part, une couche logicielle donnée peut être répartie sur plusieurs tiers: c'est le cas pour un site web où les postes clients n'hébergent que le navigateur du client (Client "léger": affichage et capture des interactions) alors que le serveur d'application héberge les autres traitements de PRÉSENTATION. Le schéma ci-dessous (site web en architecture 3-TIER) résume ces différents points:



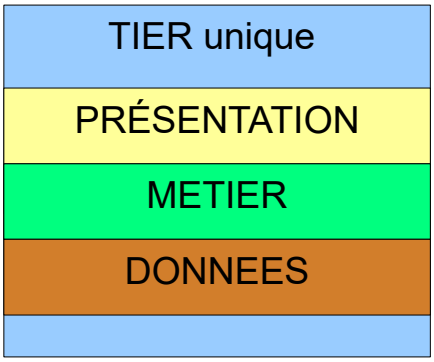
REMARQUE: lorsqu'un TIER autre que le TIER CLIENT accueille plusieurs machines, il est indispensable de mettre en place un mécanisme de répartition de charge (pour étaler la charge sur les différents serveurs du tier) ou de reprise en cas de panne s'il existe une redondance entre serveurs (pour augmenter la sécurité de fonctionnement).

III.6.4.PRINCIPAUX MODÈLES D'ARCHITECTURE REPARTIE:

III.6.4.1.ARCHITECTURE 1-TIER:

L'architecture 1-TIER correspond au cas où l'application est supportée par un seul ordinateur.

EXEMPLE: architecture en trois couches sur un seul TIER:

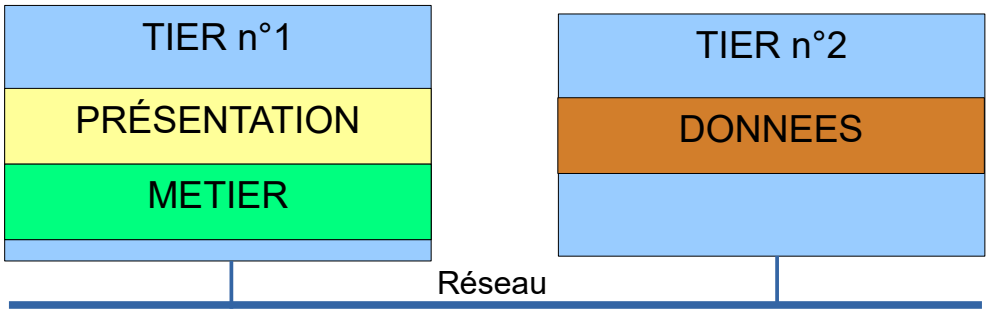


Architecture en 3 couches sur 1-tier

III.6.4.2.ARCHITECTURE 2-TIER:

L'architecture 2-TIER classique a été créée à l'origine pour soulager les MAIN FRAMES de la gestion des DONNÉES. La répartition était:

- TIER n°1: couches PRÉSENTATION et couches MÉTIER
- TIER n° 2: couche DONNÉES (gestion des données rémanentes – SGBD).



Architecture en 3 couches sur 2-tier

C'est cette architecture qui est appelée CLIENT-SERVEUR. Remarquons que le TIER CLIENT est en général représenté par plusieurs postes CLIENTS (Un serveur est fait pour gérer plusieurs clients)

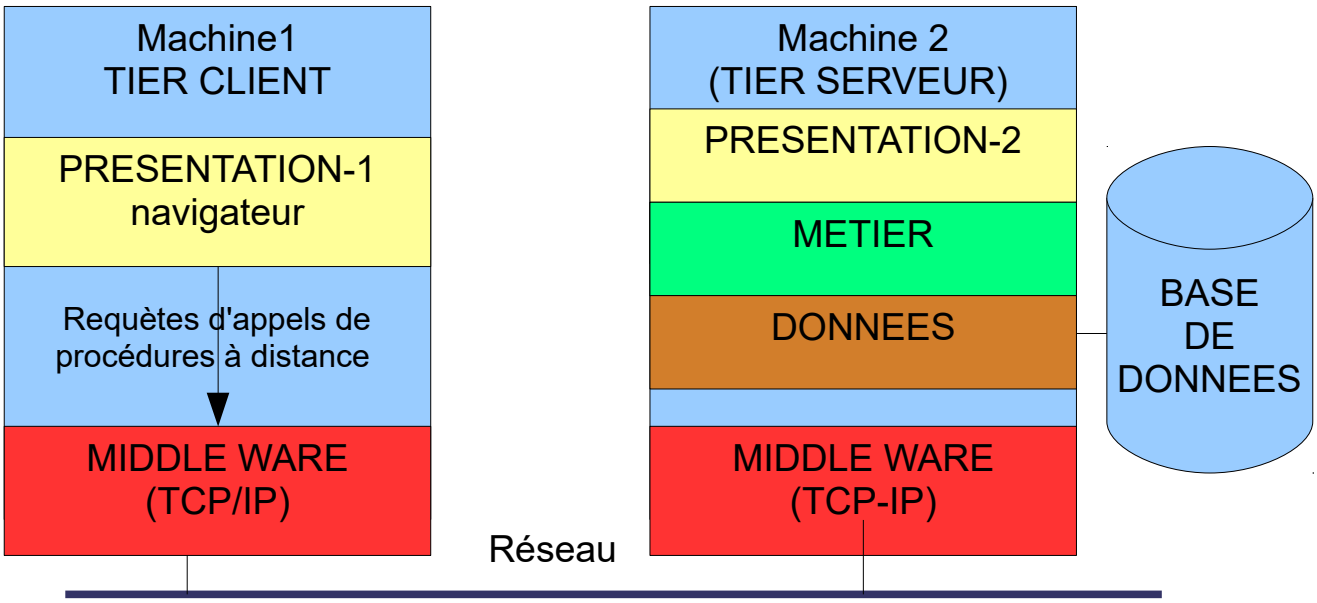
L'architecture CLIENT-SERVEUR présente les défauts suivants:

- Le POSTE CLIENT supporte l'essentiel des traitements;
- La cohabitation de la couche PRÉSENTATION et des COUCHES MÉTIER sur le CLIENT peut poser des problèmes de cohabitation et complique les évolutions;
- Le dialogue CLIENT-SERVEUR consomme beaucoup de bande passante.

Cependant, l'architecture CLIENT-SERVEUR permet l'implantation d'IHM dits "riches" car plus complets que ne le permet un navigateur seul. On parle de CLIENT LOURDS ou CLIENTS RICHES.

REMARQUE: L'exemple du chapitre précédent (IV.4.1) représente une architecture 2-TIER différente, souvent utilisée dans le cas des SERVICES WEB:

- Le TIER CLIENT (Poste client web) ne supporte que le NAVIGATEUR (CLIENT LÉGER);
- Le TIER SERVEUR supporte le reste de la couche PRÉSENTATION et les deux autres couches. Il s'agit donc également d'une architecture 2-TIER:



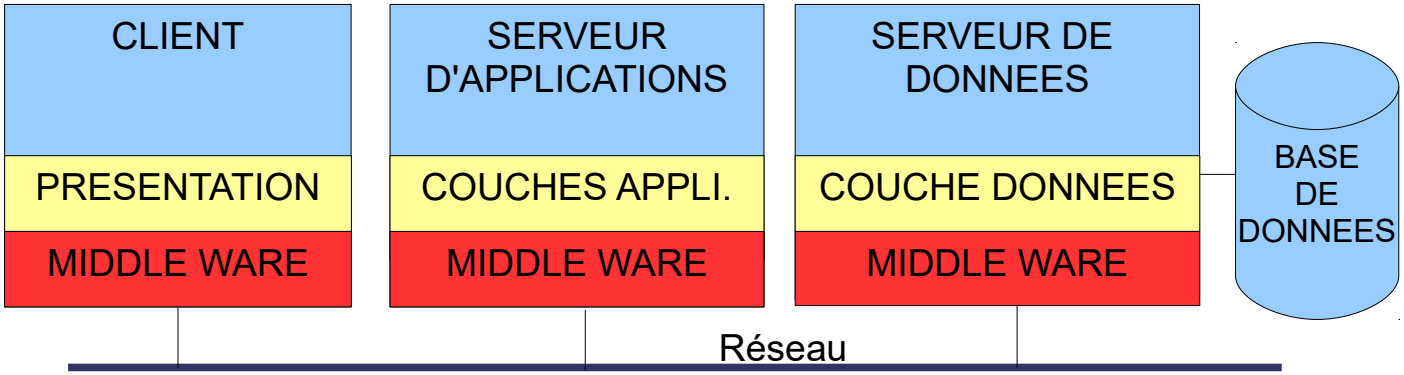
Architecture en 3 couches sur 2-tier

III.6.4.3.ARCHITECTURE 3-TIER:

Dans l'architecture 3-TIER, les trois types de traitements:

- PRÉSENTATION
- MÉTIER (TRAITEMENTS APPLICATIFS)
- DONNÉES

Sont répartis sur trois machines différentes. La couche PRÉSENTATION est donc séparée physiquement des couches APPLICATIVES:



Architecture 3-TIER

REMARQUE: chacun des trois étages peut être représenté par plusieurs machines. En effet

- Le TIER CLIENT est supporté à tout moment par tous les POSTES CLIENTS connectés à l'application;
- Chacun des clients connectés peut être pris en compte par une machine SERVEUR D'APPLICATION choisi parmi le parc des serveurs d'application disponibles;
- Les données rémanentes peuvent être distribuées sur les machines du parc de SERVEURS DE DONNÉES (bases de données réparties), avec des possibilités de redondance.

UTILISATION D'UN D.A.O: Pour assurer l'indépendance de la couche de données avec la technologie de représentation et d'enregistrement des données persistantes, l'utilisation d'un OBJET D'ACCÈS AUX DONNÉES (Data Access Object – DAO) est recommandée. Le D.A.O traite les structures de données (qu'on lui fournit sous la forme d'OBJETS) et les transforme en fonction du S.G.B.D de destination.

CONCLUSION: Cette structure permet donc d'assurer une grande fiabilité de fonctionnement tout en s'adaptant avec souplesse aux variations de charge.

III.7.LA CONCEPTION VUE AU NIVEAU DU LOGICIEL:

III.7.1.UNE ACTIVITÉ GUIDÉE PAR LES CONTRAINTES:

Considérons l'exigence suivante:

EXIGENCE N°1		
ÉNONCÉ	Le **personnel d'exploitation** doit pouvoir lire en temps réel les données saisies par le **capteur de pression**.	
Critère n°1.1	Équipement d'affichage	Écran de contrôle
Critère n°1.2	Délais d'affichage	Moins de 200 ms entre la date d'acquisition de la donnée par le capteur et son affichage sur l'IHM.
Critère n°1.3	Forme d'affichage	Numérique, en pascals, précision > 1/1000e.
Critère n°1.4	Signalisation d'anomalies	- Affiche "NULL" si la mesure n'est pas disponible; - Affichage en noir pour une pression { 50 P; - Affichage en rouge pour une pression ≥ 50 P.

Nous pouvons facilement déduire à la lecture de ce tableau que ce qui représente une EXIGENCE pour le CLIENT va se traduire par un certain nombre de CONTRAINTES À RESPECTER pour le CONCEPTEUR. Ce sont les différentes contraintes induites par les exigences du client qui vont constituer les principaux facteurs dimensionnants de l'activité de conception.

En effet, si la lecture du seul énoncé de l'exigence suggère immédiatement au concepteur un certain nombre de solutions pour lire le capteur, transférer les données vers un poste informatique et afficher les mesures sur un support lisible par les exploitants, la liste des critères contribue beaucoup à limiter ce nombre aux seules solutions répondant aux contraintes induites par ces critères:

- Le respect des critères n° 1.1, 1.3 et 1.4 vont contraindre la conception de l'architecture générale du système (choix du matériel d'affichage, conception de l'IHM);
- Le respect du critère n° 1.2 va contraindre le fonctionnement dynamique de cette architecture en induisant des contraintes de vitesse de transmission et d'exécution.

La conception des logiciels est donc en très grande partie GUIDÉE PAR LES CONTRAINTES. Celle-ci peuvent résulter de l'étude des EXIGENCES du client, mais aussi de celles du réalisateur (contraintes méthodologiques, politique de réutilisation, etc.) ou encore des normes imposées par les administrations.

III.7.2.ASPECT LOGIQUE ET ASPECT COMPORTEMENTAL:

D'une manière générale, certaines contraintes influeront sur l'architecture LOGIQUE (Détermination des composants logiciels principaux et des relations entre ces composants), tandis que d'autres influeront sur les choix concernant son

COMPORTEMENT (priorités et enchaînement des tâches, synchronisation des échanges d'informations entre ces tâches et avec la périphérie, gestion des accès aux ressources partagées).

Remarquons que ces deux aspects ne sont pas indépendants l'un de l'autre. Par exemple:

- Si le choix architectural de privilégier la réutilisation de composants peut être judicieux du point de vue de la fiabilité et de la maîtrise des coûts, il peut entraîner une majoration de la durée d'exécution (un composant réutilisable est toujours plus "lourd" qu'un logiciel "taillé sur mesures");
- Le choix d'une architecture répartie plutôt qu'une architecture "main frame" majore forcément certains délais de réaction dans la prise en compte de certains événements, du fait de la durée de transmission entre machines;
- Etc.

Il appartient au concepteur de faire les compromis nécessaires en fonction des exigences exprimées par le client.

III.8.LA MODÉLISATION EN CONCEPTION:

III.8.1.GÉNÉRALITÉS SUR LES MODÈLES DE CONCEPTION:

Comme la plupart des modèles informatiques, ceux qui sont utilisés pour la conception des logiciels peuvent être représentés sous la forme de graphes dont les nœuds sont des entités participant à la structure ou au comportement du logiciel modélisé et dont les arcs représentent les relations ou interactions existant entre ces entités.

Ces graphes sont en général accompagnés d'informations textuelles permettant de bien caractériser les différents éléments identifiés par la conception.

Les modèles correspondants à l'activité de conception sont le MODÈLE LOGIQUE (architectural), et le MODÈLE COMPORTEMENTAL (dynamique).

REMARQUES :

- Nous dirons qu'une relation entre composants logiciels est de nature STATIQUE lorsqu'elle constitue un élément structurel de l'architecture globale dont le rôle se justifié indépendamment de l'état d'exécution du logiciel. Par exemple, les relations d'héritages entre classes ou de dépendance procédurale entre deux fonctions sont de nature statique ;
- A contrario, une relation entre composants logiciels est de nature DYNAMIQUE lorsque son rôle se justifie uniquement pendant l'exécution du logiciel. Par exemple, un échange d'informations entre deux composants est une relation d'interaction de nature dynamique.

III.8.2.MODÈLE ARCHITECTURAL D'UN LOGICIEL:

III.8.2.1.DÉFINITIONS :

Ce modèle correspond à la VUE LOGIQUE du langage UML. Il s'intéresse aux différents COMPOSANTS qui constituent l'ARCHITECTURE du logiciel et aux relations de nature STATIQUE existant entre ces éléments.

III.8.2.2.OUTILS GRAPHIQUES DU MODÈLE LOGIQUE :

III.8.2.2.1.INTRODUCTION :

Il existe de nombreux outils graphiques de représentation du modèle logique. Certains obéissent à des règles de présentation relativement simples et intuitives (graphes structurels, organigrammes de programmation, par exemple), d'autres sont très formalisés et demandent un apprentissage important (les vues statiques d'U.M.L). Cependant, ils présentent des similitudes dans la manière de représenter la vue logique d'un logiciel :

III.8.2.2.2.LES NŒUDS DU MODÈLE:

A ce niveau de description, les nœuds de ces diagrammes représentent des entités LOGIQUES et non PHYSIQUES: ce sont des "boîtes noires" dont le rôle dans l'architecture globale et les interfaces avec les autres composants sont identifiés et caractérisés, mais le concepteur FAIT ABSTRACTION de leur "mécanisme interne".

NOTA : suivant le type d'outil et la méthode de conception adoptée, ces nœuds peuvent représenter des modules, des objets, des classes ou des fonctions.

III.8.2.2.3.LES RELATIONS ENTRE NŒUDS:

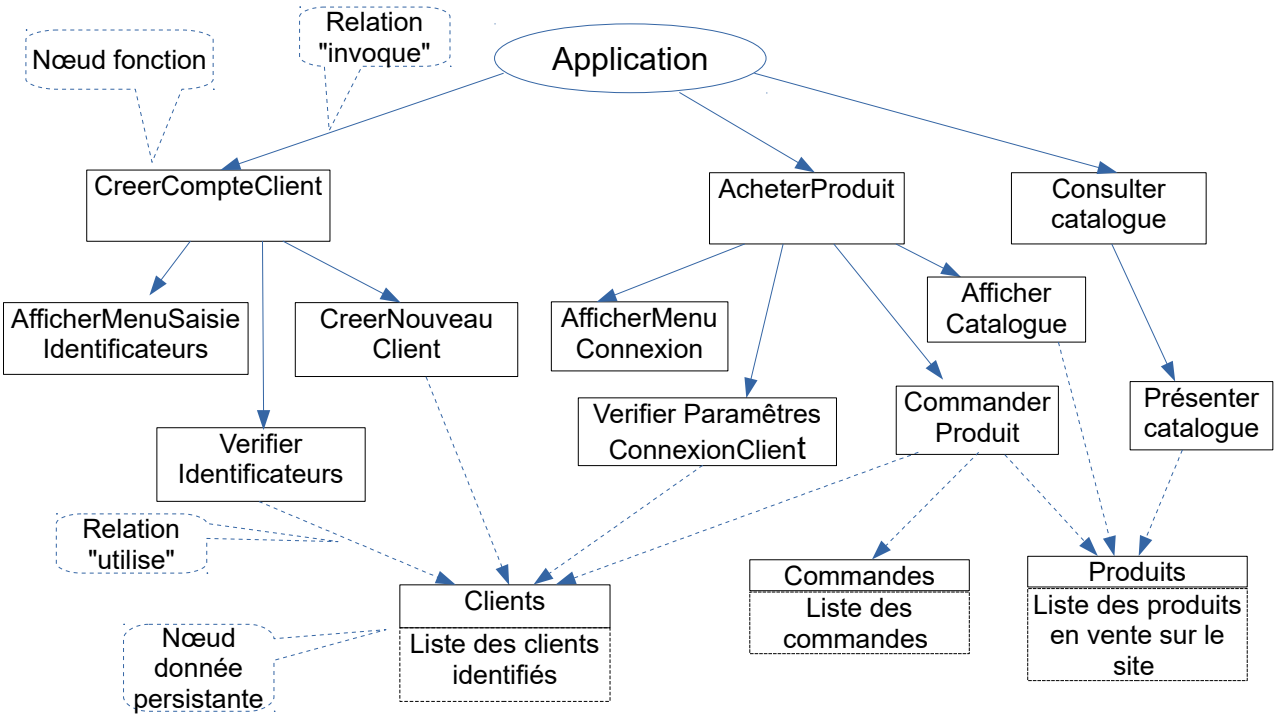
Les relations qui existent entre deux nœuds A et B traduisent une dépendance entre les deux composants qu'ils représentent. Il ne faut pas oublier que ces relations sont de type "statique" et non "dynamique". Par exemple, "A utilise B" n'indique qu'une dépendance fonctionnelle de A vis à vis de B (A a besoin de B) sans préjuger de la manière dont A sollicite B.

III.8.2.2.4.EXEMPLES :

Les paragraphes suivants donnent une description succincte des principaux diagrammes utilisés. Une présentation plus complète et plus rigoureuse est donnée en annexe.

A.EXEMPLE 1 - GRAPHE STRUCTUREL :

Ce type de diagramme, qui est souvent appelé GRAPHE STRUCTUREL est probablement le plus ancien des outils utilisés pour modéliser l'architecture d'un logiciel. Il modélise l'architecture statique d'un site web suivant la démarche procédurale.

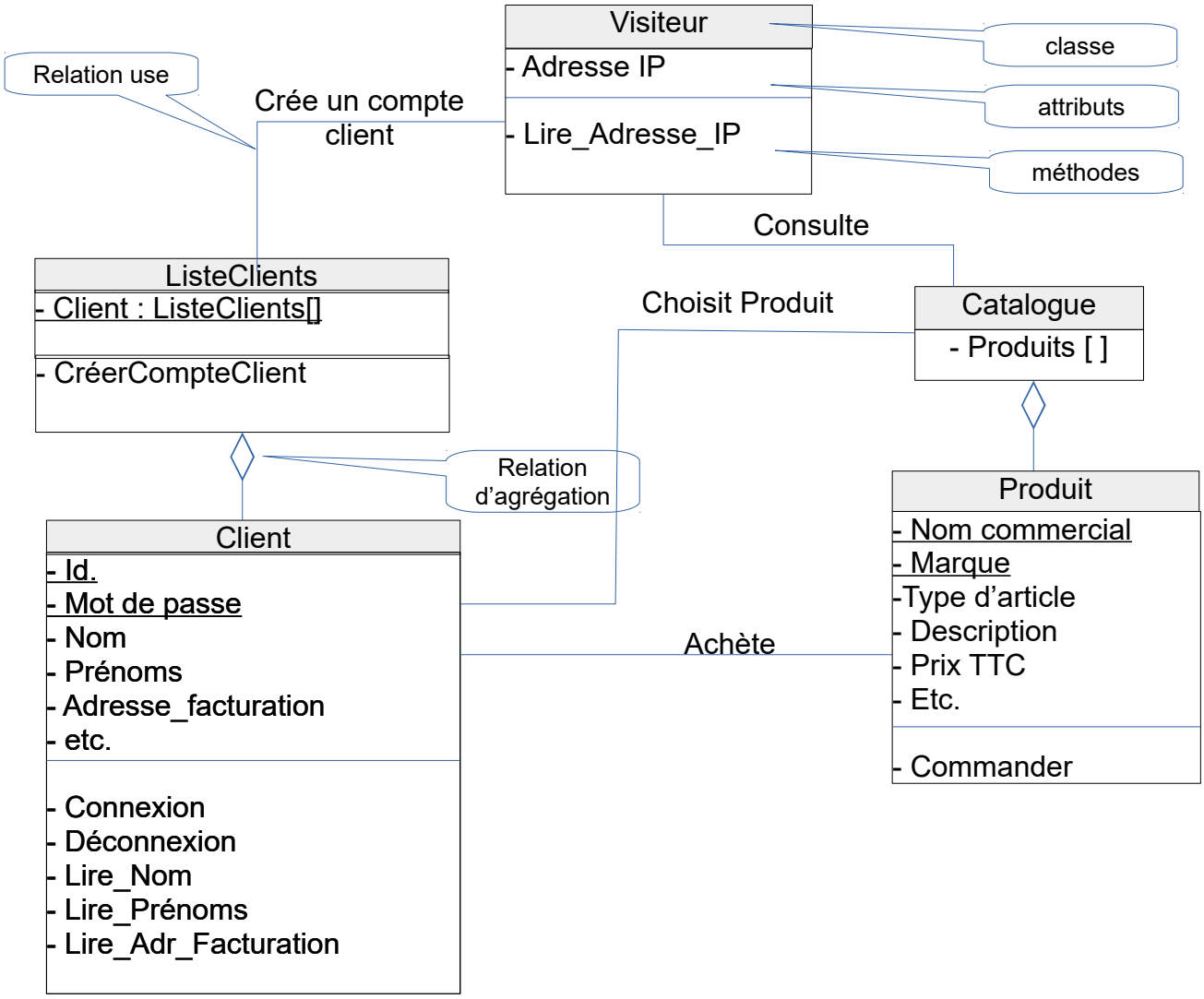


COMMENTAIRES :

- Ce graphe utilise deux sortes de nœuds : les nœuds "fonctions" qui représentent des procédures logicielles, et des nœuds "données" qui représentent des structures de données persistantes ;
- Les arcs orientés du graphe indiquent une DÉPENDANCE FONCTIONNELLE : la relation A → B indique que A a besoin de recourir aux services de B. Ce recours ne se traduit pas forcément à l'exécution par un appel procédural : il peut s'agir d'un échange de signaux ou de messages. De plus, dans un environnement multitâches, l'appel à une ressource partagée (procédure, donnée rémanente) n'est pas forcément garanti. De ce fait, l'expression "A invoque B" est souvent utilisée pour qualifier ce type de relation "statique";
- Les relations entre fonctions et structures de données se distinguent par le fait que l'arc est représenté en pointillés.

B.EXEMPLE 2 – DIAGRAMME DE CLASSES :

Le diagramme suivant correspond à la vue logique (vue des classes) d'un site WEB :



COMMENTAIRES :

- Ici, les NŒUDS représentent des CLASSES, c'est à dire des types d'OBJETS ;
- Les RELATIONS représentent :
  - Soit des ASSOCIATIONS entre classes (au sens du modèle entités-associations, une classe pouvant être considérée comme une ENTITÉ dotée d'ATTRIBUTS).
  - Soit d'autres types de relations entre classe, comme la généralisation, l'héritage, l'agrégation ou la composition ;

III.8.3.MODÈLE COMPORTEMENTAL D'UN LOGICIEL:

III.8.3.1.INTRODUCTION :

Le MODÈLE COMPORTEMENTAL d'un logiciel représente la manière dont celui-ci réagit aux différentes sollicitations qu'il reçoit de son environnement d'exécution. Il correspond sensiblement à la VUE DES PROCESSUS du langage U.M.L.

Son objectif est de concevoir à partir des spécifications fonctionnelles et techniques de l'application à développer un "schéma de fonctionnement" mettant en évidence :

- Les différentes entités à caractère dynamique intervenant dans l'exécution du logiciel (processus, tâches, threads, données persistantes partagées) ;
- Les différentes interactions qui se produisent entre ces entités ou avec la périphérie ;
- Les risques de conflits induits par la concurrence des différentes tâches pour l'accès aux ressources partagées (y compris les CPU).

III.8.3.2.OUTILS GRAPHIQUES DU MODÈLE COMPORTEMENTAL :

*III.8.3.2.1.*LES NŒUDS DES MODÈLES :

A la différence de la conception architecturale, dont les nœuds et les arcs représentent des entités indépendantes de l'état d'exécution du logiciel (fonctions, classes, objets, relations d'utilisation ou d'instanciation, etc.), les nœuds des graphes comportementaux représentent des concepts qui n'existent que durant l'exécution du logiciel (états, activités, objets, processus et threads).

*III.8.3.2.2.*LES RELATIONS ENTRE NŒUDS:

Les arcs des modèles comportementaux représentent des interactions qui se produisent du fait de l'exécutions du logiciel. Il s'agit donc de relations DYNAMIQUES. La nature de celles-ci dépend de la nature des nœuds du diagramme , c'est à dire de l'outil de modélisation utilisé.

*III.8.3.2.3.*DIFFÉRENTS POINTS DE VUE COMPORTEMENTAUX :

Les méthodologies SA-SD (Structured Analysis/Structured Designe), SA-RT ((Structured Analysis for Real Time), DARTS (Design Approach For Real-Time System) ainsi que le langage de modélisation U.M.L proposent de nombreux diagrammes de modélisation du comportement des logiciels. Nous pouvons classer ces outils en trois catégories, choisies en fonction de la manière de caractériser le comportement des logiciels ;

- Les diagrammes d'états-transitions ;
- Les diagrammes activités/flux d'informations ;
- Les diagrammes d'interactions entre objets .

Les paragraphes suivants donnent une description succincte de chacune de ces catégories ainsi que des exemples concrets. Le but est surtout rappeler les principales notions. Une présentation plus complète et plus rigoureuse de certains outils est donnée en annexe pour les lecteurs peu familiers du sujet.

*III.8.3.2.4.*LES DIAGRAMMES D'ÉTATS/TRANSITIONS :

A.PRÉSENTATION :

Les diagrammes d'états/transitions permettent de représenter graphiquement le comportement d'un logiciel en assimilant celui-ci à un AUTOMATE À ÉTATS FINIS:

- Les nœuds du diagramme représentent les différents ÉTATS que peut prendre le logiciel. A un instant donné, le logiciel se trouve dans un état et un seul ;
- Les arcs représentent les TRANSITIONS possibles entre les états (passage d'un état à un autre) ;
- Les transitions sont provoquées par la survenue d'ÉVÉNEMENTS (signaux externes, commandes des exploitants, exceptions,etc). Des indications textuelles renseignent sur l'événement déclencheur de chaque transition ;
- Certaines transitions (transitions réflexives) peuvent conserver le même état tout en provoquant une action particulière (ex : une transition déclenchée par un signal horaire conserve l'état courant tout en déclenchant l'historisation de certaines données).

REMARQUES: une présentation plus détaillée des automates à états finis est donnée en annexe du présent document.

B.EXEMPLE :

L'exemple suivant modélise sous la forme d'un diagramme d'états-transitions le comportement du logiciel embarqué dans un sélecteur de boissons. Les états de cette machine sont :

- En service, attente sélection ;
- Personnalisation boisson (+ ou – de sucre) ;
- Paiement ;
- Préparation ;
- Présentation de la boisson .

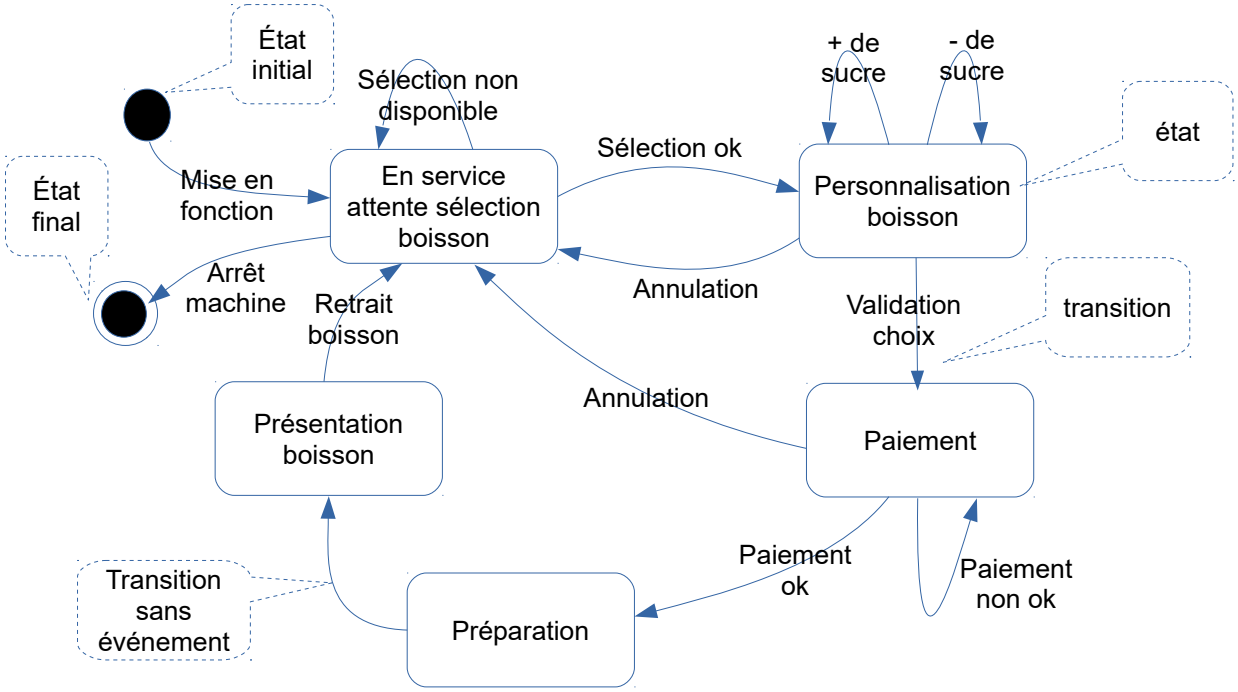
Chacune de ces tâches constitue une activité durable représentative d'un état stable du système. Ces états sont représentés dans le diagramme par des rectangles à bords arrondis.

Les événements susceptibles d'agir sur l'état du système sont :

- La mise en fonction machine ;
- L'arrêt machine ;
- La sélection d'une boisson ;
- La non disponibilité de la boisson sélectionnée;
- L'annulation de la transaction ;

- Les signaux + ou moins de sucre ;
- La validation des choix ;
- Le paiement OK ;
- Le paiement non ok ;
- La fin de préparation de la boisson ;
- Le retrait de la boisson

Ces événements déclenchent des transitions qui sont représentées par des lignes fléchées. Les transitions pour lesquelles aucun événement déclencheur n'est indiqué sont déclenchées automatiquement par la fin de la tâche associée à l'état de départ de cette transition (exemple: la flèche sortante de l'état préparation et la transition déclenché par la fin de la préparation).



Ce diagramme permet, connaissant l'état courant, de déterminer la modification de cet état provoquée par la survenue d'un événement. Il permet donc de déterminer le comportement du logiciel en réponse à un événement donné ou à une suite d'événements ;

C.UTILISATION :

- Les diagrammes d'états/transitions sont souvent utilisés en début de conception pour préciser les modalités de traitement d'un cas d'utilisation ou d'une exception. A ce niveau de conception, les ÉTATS et les TRANSITIONS pris en compte correspondent à des notions de nature FONCTIONNELLES.

- Ils peuvent également être utilisés lors de la conception détaillée d'un composant : par exemple, une CLASSE peut être représentée sous la forme d'un automate à états finis dont les états correspondent aux valeurs possibles de l'ensemble de ses attributs et dont les transitions sont provoquées par l'activation de ses différentes méthodes.

*III.8.3.2.5.*LES DIAGRAMMES ACTIVITÉS/FLUX D'INFORMATIONS:

A.PRÉSENTATION :

Ce type de diagramme décrit le comportement d'un logiciel par le biais des ACTIVITÉS qui se déroulent pendant son exécution ainsi que des flux d'exécution et d''informations que ces activités échangent entre elles et avec les différentes ressources qu'elles partagent.

NOTIONS D'ACTIVITÉ ET D'ACTION:

- D'un point de vue fonctionnel, une ACTIVITÉ représente un ensemble cohérent de traitements informatiques s'exécutant suivant une logique algorithmique donnée. Elle est caractérisée par un événement initiateur (événement qui "lance" le flot d'exécution), par les différents traitements qui la composent et par son flot d'exécution sortant. A l'intérieur d'une activité, le flot d'exécution qui la parcourt peut se subdiviser localement en un plusieurs flots s'exécutant parallèlement ;
- Une ACTION désigne en général un traitement informatique que l'on peut considérer comme ATOMIQUE au niveau de l'application, c'est à dire dont les mécanismes internes n'ont pas besoin d'être détaillés au niveau de la démarche de conception (par exemple : une affectation de valeur, un calcul simple, l'émission ou la réception d'un signal ou d'un message, etc.) ;
- Une ACTIVITÉ correspond à l'exécution

NOTION DE FLUX D'INFORMATIONS:

Les activités interagissent entre elles ou avec d'autres entités de leur périphérie d'exécution en échangeant des FLUX D'INFORMATIONS. Un FLUX représente des informations "en mouvement", de nature fugace car elles n'existent que pendant leur transmission, par opposition à des informations dites PERSISTANTES qui restent accessibles tant qu'elle n'ont pas été détruites par une activité.

Nous pouvons distinguer les types de flux suivants:

- Les FLUX DE DONNÉES ;
- Les FLUX DE COMMANDES;
- Les SIGNAUX;
- Les EXCEPTIONS.

NOTION DE RESSOURCE:

Une RESSOURCE est une entité logicielle qu'une ou plusieurs ACTIVITÉS ont besoin d'utiliser pour s'exécuter.

Les autres types de ressources sont :

- Les données persistantes (mémoires partageables, fichiers, bases de données);
- Les ressources logicielles (bibliothèques de fonctions, modules, classes, etc.).
- Les connexions réseaux (exemple : connexion TCP/IP);
- Les PILOTES des équipements matériels (imprimantes, mémoires de masse, etc). En effet, l'utilisation d'une ressource matérielle par un logiciel s'effectue toujours par l'intermédiaire d'un pilote logiciel intégré au système d'exploitation ;
- etc.

REMARQUE : quelle que soit l'activité considérée, son exécution nécessite toujours la consommation d'une partie de la puissance du CPU (ou du pool de CPU dans le cas d'un système multiprocesseurs). L'accès au CPU s'effectue par l'intermédiaire des fonctions de traitement des taches du système d'exploitation.

NOTION DE CONCURRENCE ENTRE ACTIVITÉS :

Chaque activité d'une application entraîne nécessairement l'exécution de processus ou de threads. Dans un environnement d'exécution multitâches, ces entités dynamiques se trouvent fatalement en concurrence pour l'accès aux ressources qui leurs sont nécessaires.

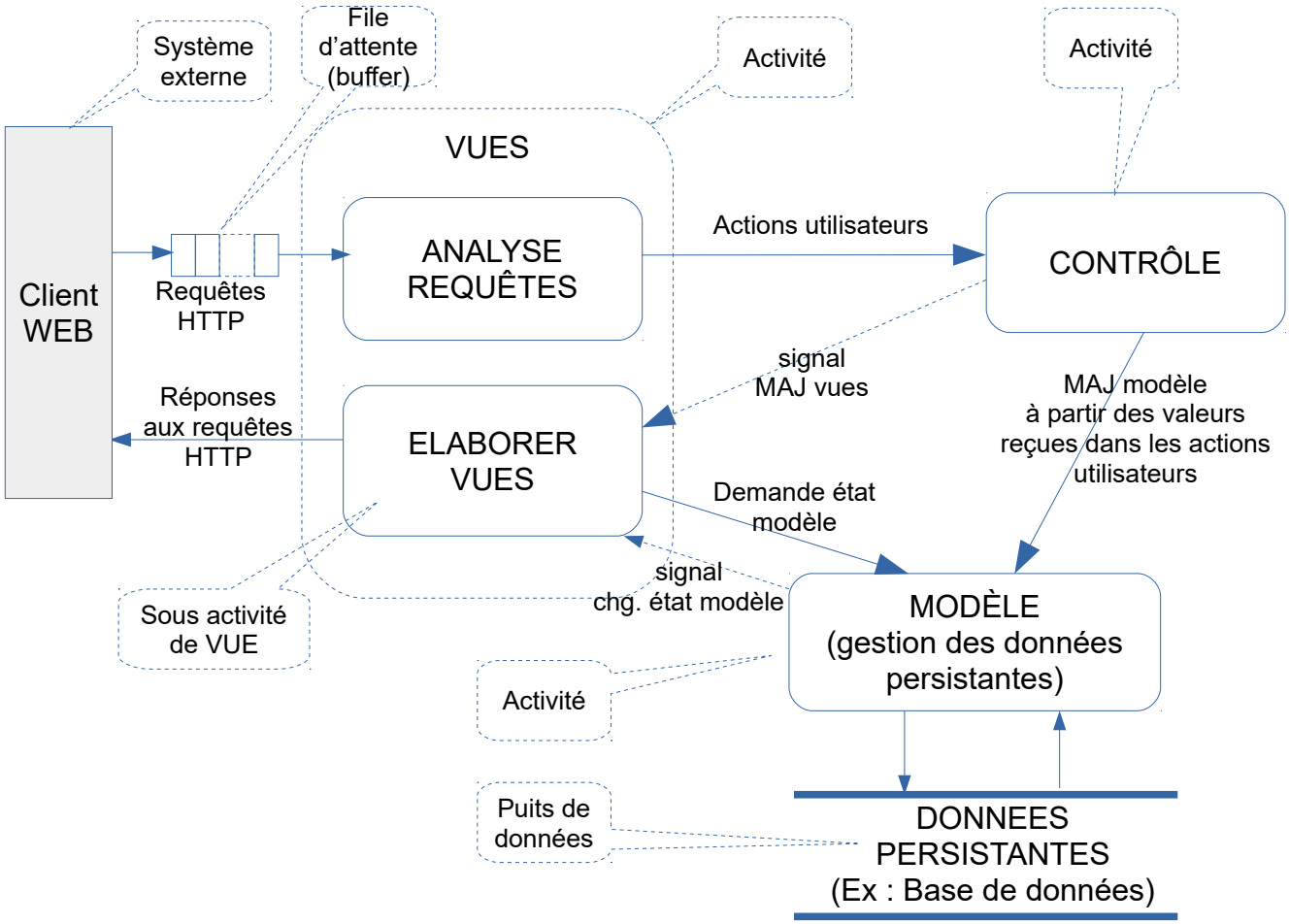
La concurrence de plusieurs activités pour l'accès à une même ressource est potentiellement source de conflits entraînant des dysfonctionnements. De ce fait, des mécanismes de résolutions doivent être intégrés à l'application pour permettre leur résolution. Ces mécanismes peuvent figurer sur certains diagrammes sous la forme de nœuds particuliers appelés nœuds de contrôle.

B.EXEMPLE 1 : DIAGRAMME SA/RT

Le diagramme ci-dessous représente le comportement du logiciel gérant un site web à l'intérieur d'un serveur HTTP. Le symbolisme graphique utilisé est celui des diagrammes de flux SA/RT :

- Les activités sont représentées par des rectangles à bords arrondis ;
- Les données persistantes (puits de données) sont représentées par des barres parallèles épaisses ;
- Les flux d'informations sont représentés par des flèches entre deux activités ou entre une activité et un acteur externe ;
- D'autres types de nœuds (nœuds de contrôle) permettent de spécifier la manière dont les flux d'informations sont contrôlés (dans le schéma ci-dessous, un nœud "file d'attente est utilisé pour "bufferiser" les requêtes HTTP entrantes).

NOTA :Une description plus détaillée des diagrammes SA/RT est fournie en annexe.



COMMENTAIRES SUR LE SYMBOLISME SA/RT :

Le symbolisme de représentation des flux dans les diagrammes SA/RT est très réduit :

- Les flèches pleines représentent des FLUX DE DONNÉES destinées à être transformées par l'activité destinataire ;
- Les flèches pointillées représentent des SIGNAUX, destinés à informer l'activité destinataire de la survenue d'un événement. Une donnée de type "atomique" (un entier naturel, par exemple) peut être associée à un signal : elle permet de spécifier au destinataire l'action à effectuer à la réception du signal. Il s'agit alors d'une "commande";

Ce symbolisme très sommaire ne permet pas de décrire le détail des interactions entre activités (échanges synchrones, asynchones, bufferisés, etc.). De ce fait, les diagrammes SA/RT sont souvent accompagnés de descriptions textuelles. Notons toutefois que des "nœuds de contrôle" (files d'attente, sémaphores) permettent de préciser graphiquement la gestion et le contrôle des flux dans une certaine mesure;

COMMENTAIRES SUR LE DIAGRAMME :

- La survenue d'une requête HTTP en provenance d'un CLIENT WEB (navigateur) déclenche la sous-activité ANALYSE REQUÊTES de l'activité VUES qui va en déduire les ACTIONS UTILISATEURS à transmettre à l'activité CONTRÔLE. Ces actions utilisateurs véhiculent les valeurs d'argument des requêtes ;
- L'activité CONTRÔLE, déclenchée par les messages "actions utilisateur" va lancer les traitements nécessaires pour répondre à ces actions:
  - Mettre à jour les données persistantes à partir des valeurs d'arguments transmises par les requêtes HTTP reçues, par appel de l'activité MODÈLE;
  - Déclencher la mise à jour des VUES à afficher sur l'IHM, grâce à l'émission du signal "chg état modèle" vers la sous-activité ÉLABORER_VUES de l'activité VUES;
- Enfin, la sous-activité ÉLABORER_VUES va déclencher l'activité MODÈLE pour récupérer les données rémanentes nécessaires à l'élaboration des vues (message "demande état modèle". ÉLABORER_VUES et ANALYSE_REQUÊTES peuvent se dérouler en parallèle.

RÉCURSIVITÉ DE LA DÉMARCHE:

Cette démarche peut être appliquée d'une manière itérative : Chacune des activités identifiées à un certain niveau d'analyse est elle-même décomposée en un certain nombre de sous-activités mises en relation par des flux d'informations, et ainsi de suite. La démarche s'arrête lorsque les sous-activité ainsi identifiées peuvent être considérées comme "atomiques".

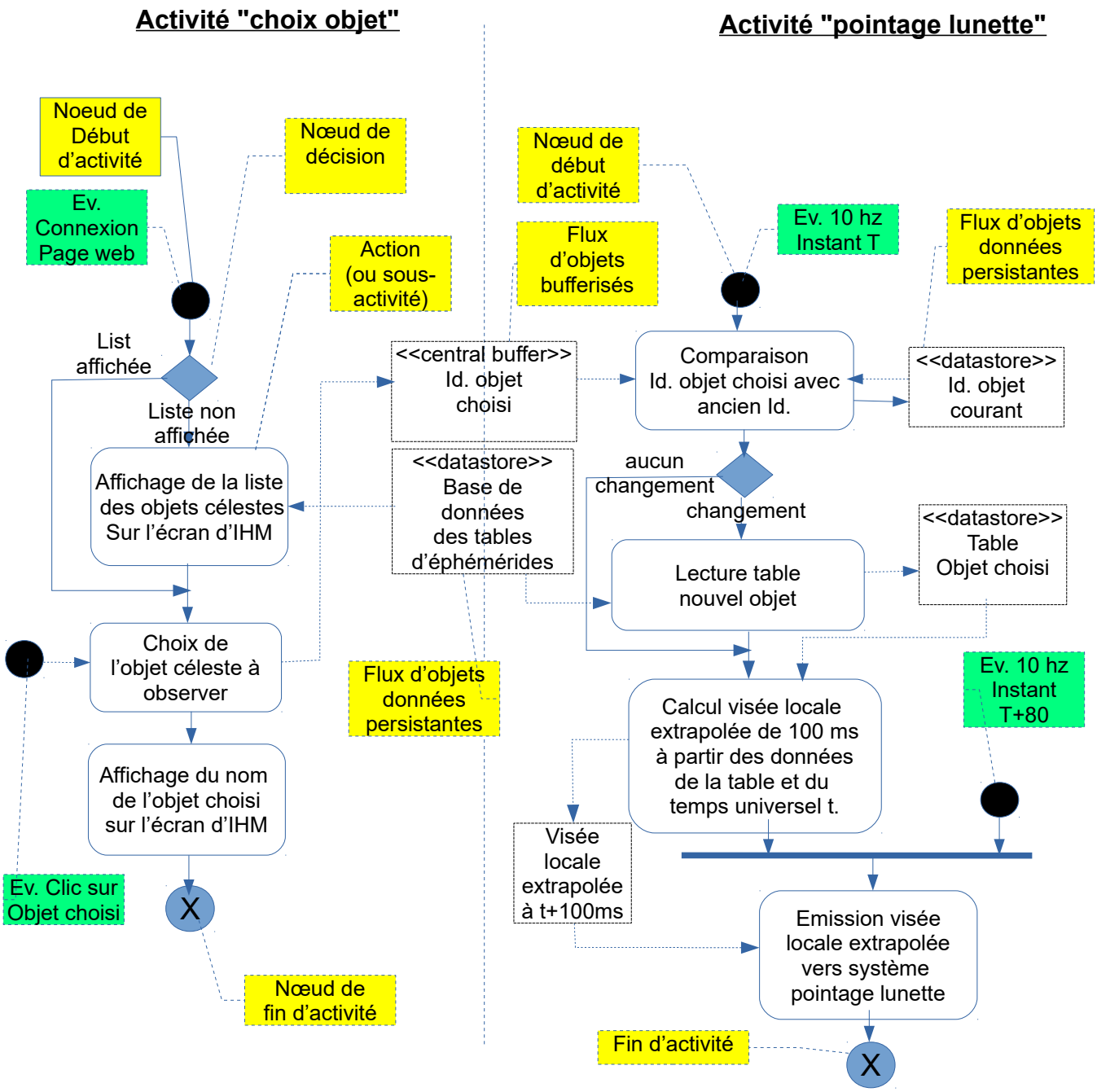
C.EXEMPLE 2 : DIAGRAMME D'ACTIVITÉ U.M.L :

Un diagramme d'activité U.M.L permet de décrire en détail une ou plusieurs activités d'une application ainsi que les interactions entre ces activités :
Nous pouvons distinguer deux catégories de nœuds :

- Les "nœuds exécutables" (ACTIONS "atomiques" ou SOUS-ACTIVITÉS que l'on peut considérer comme atomiques au niveau d'analyse courant) ;
- Les "nœuds d'objets" qui permettent de spécifier les échanges de données entre activités) ;
- Les "nœuds de contrôle" qui permettent de représenter le contrôle du flux d'exécution : nœuds début et fin d'activité, nœuds de décision ou d'union nœuds de bifurcation, de fusion, etc.

NOTA : Une description plus détaillée des diagrammes d'activité est donnée en annexe.

L'exemple ci-dessous concerne une application permettant de pointer une lunette astronomique sur un objet céleste appartenant à une liste d'objets répertoriés dans une base de données. Une I.H.M permet de choisir cet objet tandis qu'une autre activité se charge de pointer en permanence la lunette vers la position courante de l'astre. Les éléments de pointage (azimut et élévation) sont rafraîchis toutes les 100 ms :



COMMENTAIRES SUR LE DIAGRAMME:

-
- Le diagramme suivant fait apparaître deux activités non liées de façon procédurale. Elles sont représentées dans deux "couloir d'exécution". Cette disposition permet de faire figurer les données persistantes communes à ces deux activités:

- L'identificateur de l'objet céleste à viser ;
- La base de donnée des tables d'éphémérides ;

- D'autre part, l'étude de l'activité "pointage lunette" nous amène à définir plusieurs autres données persistantes:
  - La donnée "Id. objet courant"" (qui permet de détecter le changement d'objet) ;
  - La "Table d'éphémérides de l'objet choisi" (qui permet d'éviter de rechercher cette table dans la base de données à chaque cycle de pointage) ;
  - La "Visée locale extrapolée à t+100 ms" qui permet de stocker les coordonnées de la visée, issues du calcul, en attendant l'événement d'émission (10 hz + 80 ms).
- Le calcul de la visée locale extrapolée est un calcul relativement complexe recourant à plusieurs changements de repères spatiaux et des ajustements de trajectoires par des courbes polynomiales en trois dimensions ;
- Les activités issues de l'exigence "Choix objet" comprennent deux cas d'exécution, suivant que la page web incluant la listes des objets sélectionnables est affichée sur l'IHM de l'utilisateur ou non;
- Les activités issues de l'exigence "pointage lunette" comprennent deux cas d'exécution, selon que l'on change ou non d'objet céleste. Lorsqu'on change d'objet céleste, le cas d'exécution comprend un accès en lecture à la table d'éphémérides du nouvel objet. Ces tables sont des données volumineuses (de l'ordre du Mo) : ces accès disques peuvent donc durer un temps non négligeable par rapport au cycle de récurrence des pointages (100 ms), surtout si la base de données n'est pas localisée sur la même machine que l'application.

D.APPLICATIONS :

DIAGRAMMES D'ACTIVITÉS :

- Dans le contexte d'une démarche de conception s'appuyant sur U.M.L, les diagrammes d'activités sont très utilisés pour décrire des cas d'utilisation complexes, car leur formalisme graphique les rend, dans ces cas, plus intelligibles que les diagrammes d'états-transition ;
- Les diagrammes d'activités permettent également de mettre en évidence les flux d'informations entre les activités et les "objets" persistants qu'elles partagent ;
- L'existence de nœuds de contrôle permettant de représenter l'union, la bifurcation, l'union, des flux d'exécution, mais aussi leur aiguillage en fonction de conditions permet également de spécifier assez finement des traitements algorithmiques, ce qui les rend utilisables en phase de conception détaillée ;
- De par leur formalisme rigoureux et complet, l'utilisation des diagrammes d'activité demande un apprentissage assez conséquent ;

DIAGRAMMES DE FLUX SA/RT :

Par comparaison avec les diagrammes d'activité U.M.L, les diagrammes de flux SA/RT sont surtout orientés vers la modélisation des interactions entre TÂCHES et des situations de concurrence pour l'accès aux ressources. En revanche, ils ne permettent pas de faire ressortir la logique algorithmique qui préside à l'exécution des activités.

De par leur formalisme assez sommaire, les diagrammes SA/RT n'exigent que peu d'apprentissage. Le revers de la médaille est qu'ils ne permettent pas d'atteindre le niveau de rigueur des modélisations par diagrammes d'activités.

CONCLUSION :

Les diagrammes SA/RT permettent de modéliser rapidement le comportement des logiciels. Ils s'appliquent bien aux applications de conduite de processus et au "temps réel". En revanche, en raison de leur manque de rigueur et de précision, il est plus sûr de leur préférer les diagrammes d'activités d'U.M.L dans le cas d'applications supportant une logique complexe.

III.8.3.2.6.DIAGRAMMES D'INTERACTIONS:

A.PRÉSENTATION :

Nous avons vu précédemment que l'architecture LOGIQUE d'un logiciel bien construit repose sur une ou plusieurs hiérarchies de modules reliés entre eux par des relations de dépendance "statique" (c'est à dire non liées à l'exécution de ces modules).

Si nous nous plaçons maintenant d'un point de vue DYNAMIQUE et dans un environnement multitâches, l'objet de la conception est d'étudier la manière dont les différents flux d'exécution vont activer ces modules afin de prévoir les conflits d'accès éventuels et de mettre en place les mécanismes de contrôle adéquats.

Les diagrammes d'interaction ont pour but d'étudier la manière dont les différentes entités logicielles "dynamiques" collaborent entre elles pendant l'exécution de l'application. Ces entités peuvent être des objets (instances de classes exécutables), des modules logiciels ou encore des "acteurs" (dans le sens admis pour décrire les "use cases"). Ces entités collaborent entre elles grâce à des échanges de messages.

Le langage de conception UML propose divers diagrammes d'interaction :

- Les DIAGRAMME DE COLLABORATION, qui permettent de modéliser les interactions entre les "objets" intervenant lors de l'exécution (instances de classes ou acteurs) dans un contexte d'exécution donné;
- L'extension de ces diagrammes aux notion d'OBJETS ACTIFS et NON ACTIFS ;
- Les DIAGRAMMES DE SÉQUENCE qui permettent de modéliser le déroulement de la collaboration entre objets du point de vue temporel.

B.NOTIONS D'OBJETS ACTIFS ET PASSIFS:

DÉFINITION U.M.L:

Le langage U.M.L fait la distinction entre les classes et objets dits ACTIFS et les classes et objets dits PASSIFS (ou non-actifs):

- Dans le premier cas, il s'agit de classes ou d'objets qui peuvent contrôler leur propre flux d'exécution;
- Dans le deuxième cas, il s'agit de classes ou d'objets qui, pour s'exécuter, ils doivent attendre qu'un autre objet les appelle.

DÉFINITION DANS LE CONTEXTE DES LANGAGES DE PROGRAMMATION :

Cette définition U.M.L peut sembler au premier abord assez obscure car elle adopte une formulation très abstraite. Cependant, les notions d'objets actifs et passifs peuvent être facilement implémentées dans les langages objet : Par exemple, dans le langage Java, un objet actif (appelé AGENT) est une instance d'une classe implémentant l'interface Runnable (interface doté d'une seule méthode, la méthode run). Exemple :

```
Class agent implements Runnable
{
  public void run (
                        for (i = 0; i > 10; i++)  System.out.println("" + i);
                    );
}

Class Test
{
   public static void main()
   {
     agent Ag = new Agent ();
     Thread t = new Thread(Ag);       // Le thread t contrôle l'exécution de Ag
      t.start();                      // Le thread t lance l'exécution de la méthode run de Ag
    }
}
```

Le lancement du programme (point d'entrée main) provoque la création d'un thread t dont la cible d'exécution est l'objet ag (objet agent, synonyme d'objet actif). Le thread t contrôle donc l'exécution de la méthode run de l'agent.

DÉFINITION PLUS CONCRÈTE :

De ce qui précède, nous pouvons déduire cette définition plus concrète des objets actifs :

"Un objet actif est un objet dont la seule méthode publique est la cible de lancement d'un thread. Par opposition, un objet passif ne peut être activé que par l'appel d'un autre objet."

GÉNÉRALISATION DE LA NOTION

Le concept peut être généralisé aux modules d'une application quelconque, même si celle-ci n'est par programmée en "objets" :

- Un module peut être dit ACTIF si l'une de ses méthodes publique est la cible de lancement d'un thread ;
- Un module peut être dit passif si son exécution ne peut être déclenchée que par un autre module.

C.EXEMPLE 1: DIAGRAMMES DE COLLABORATION (U.M.L):

PRÉSENTATION :

Les diagrammes de collaboration permettent de modéliser les interactions entre les objets d'une application en cours d'exécution. Dans ce contexte, le mot "objet" peut représenter une instances de classe ou un acteurs (dans le sens des "use cases").

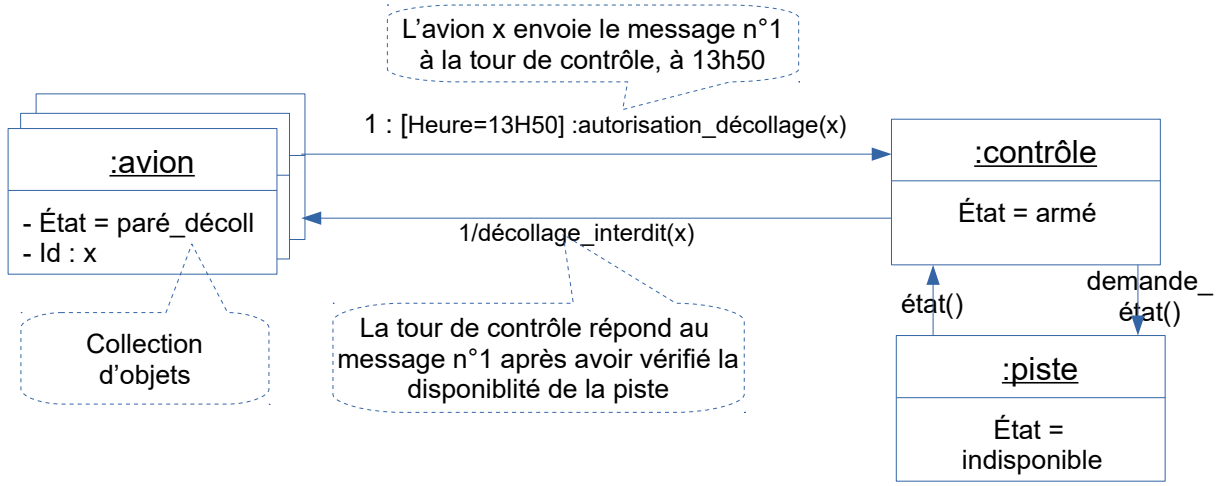
Le formalisme de représentation des objets dans ces diagrammes permet de préciser leurs états au moment des interactions.

La syntaxe d'U.M.L permet de décrire très précisément les messages échangés. Ainsi, pour chaque message, il est possible de préciser :

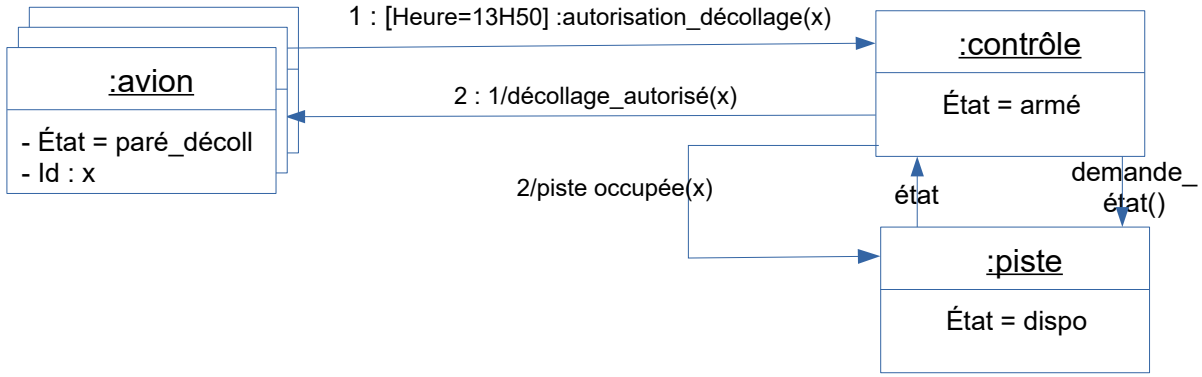
- Les "clauses" qui conditionnent son envoi,
- Son numéro d'ordre par rapport aux autres messages ;
- Éventuellement, sa récurrence ;
- Les arguments qu'il véhicule vers le récepteur.

EXEMPLES :

Cas n° 1 : un avion fait sa demande de décollage alors que la piste est indisponible :



Cas n° 2 : l'avion x fait sa demande de décollage alors que la piste est libre:



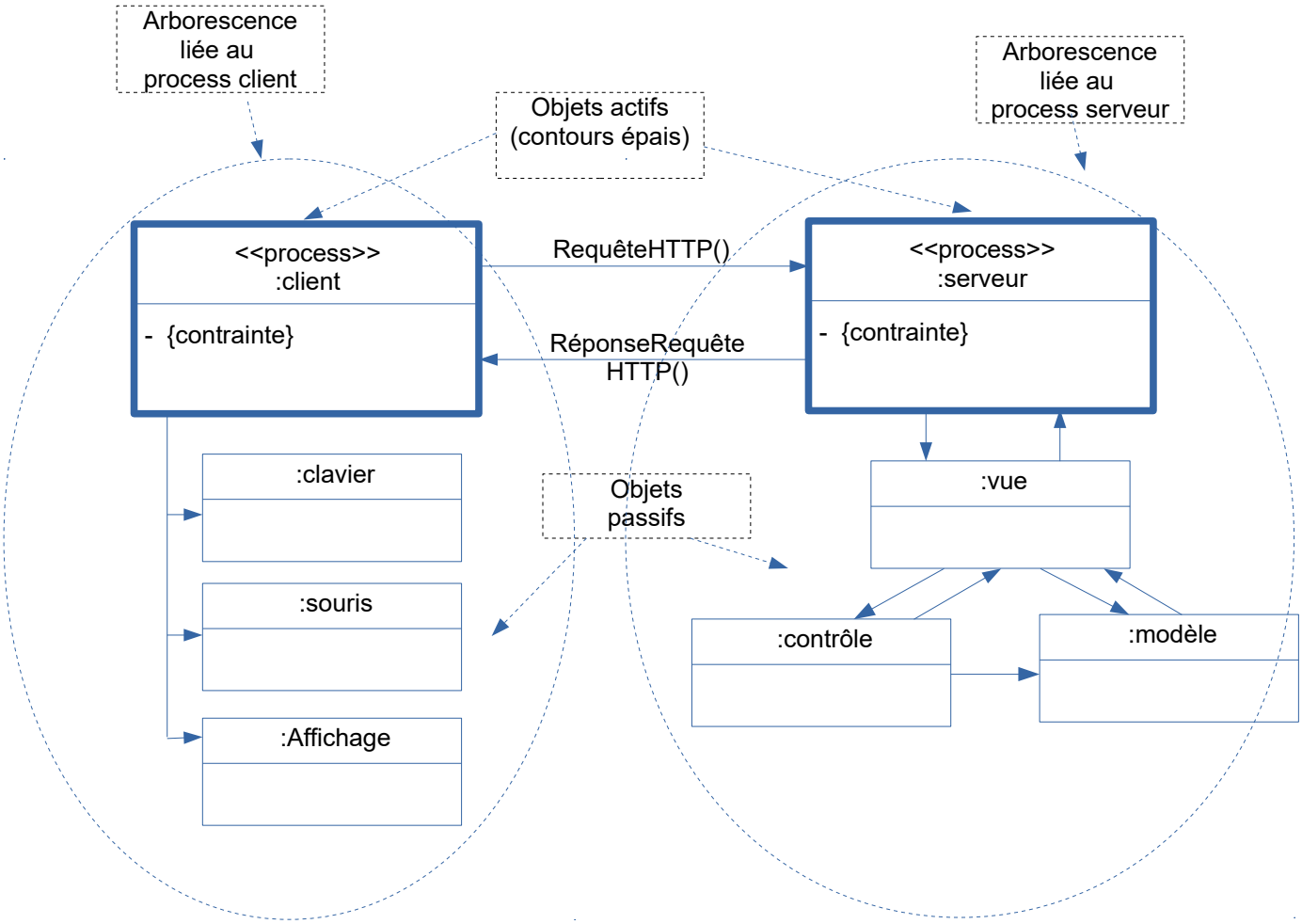
D.EXEMPLE 2 : DIAGRAMMES DE COLLABORATION ET OBJETS ACTIFS (U.M.L) :

PRÉSENTATION :

La distinction entre objets ACTIFS et PASSIFS permet de mettre en évidence dans les diagrammes de collaboration les notions de parallélisme et de concurrence. En effet, un objet actif est par définition associé à un THREAD, puisque c'est lui qui initie et contrôle son propre flux d'exécution. Les objets PASSIFS ne pouvant être activés que par d'autres objets (et en particulier, les objets actifs), chaque objet ACTIF occupe le sommet d'un "arbre des appels".

EXEMPLE :

Le diagramme ci-dessous décrit la collaboration entre les objets résultant de l'exécution d'une application "site web" (partie client et partie serveur) :



E.EXEMPLE 3 : DIAGRAMMES DE SÉQUENCES (U.M.L) :

PRÉSENTATION :

Les diagrammes de séquence représentent également les interactions entre objets dans une application. La différence avec les diagrammes de collaboration réside dans le fait que la dimension temporelle y est explicitement représentée.

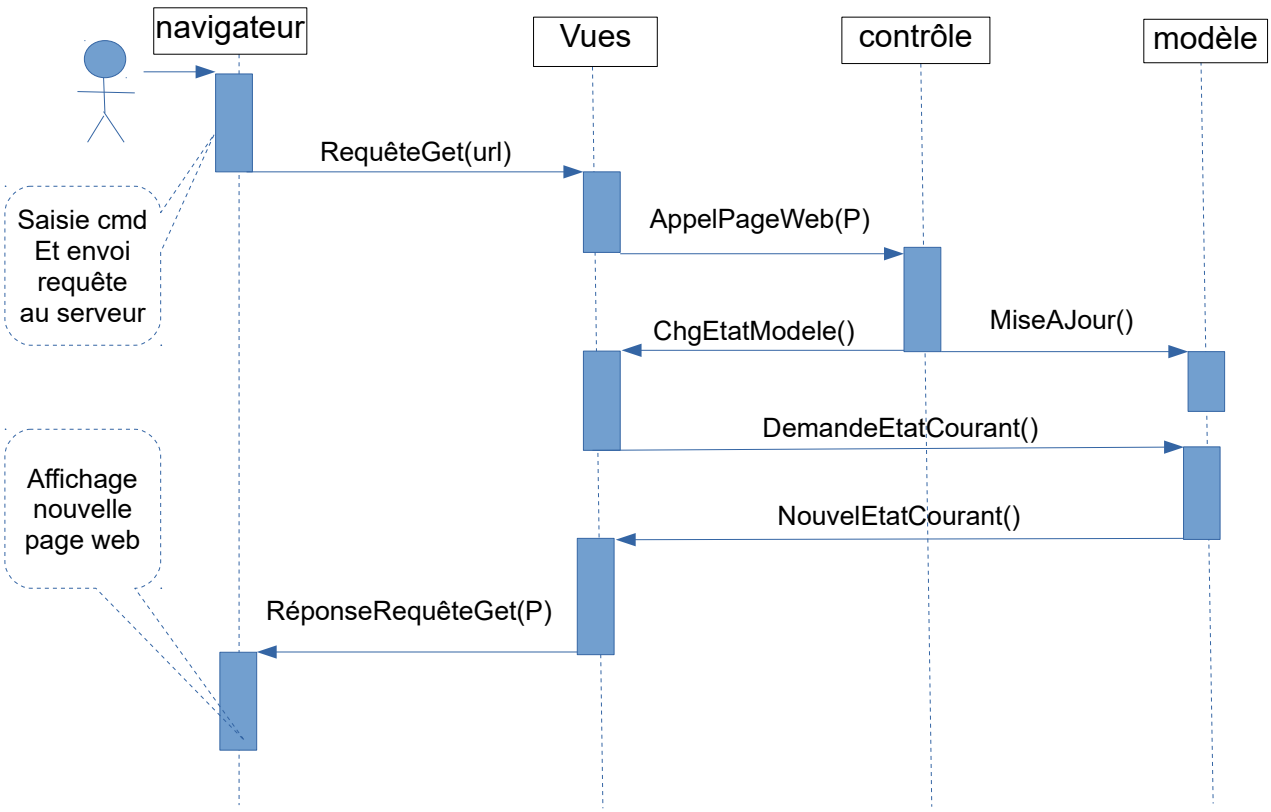
A chacun des différents objets étudiés (dans l'exemple : navigateur, vue, contrôle, modèle) est associée une LIGNE DE VIE, représentée par des rectangles allongés dans le sens vertical indiquant les périodes d'activité de l'objet en question. Ces rectangles sont alignés sur une ligne pointillée verticale représentant l'écoulement du temps.

Les interactions entre objets sont représentées par des flèches de différentes formes en fonction des caractéristiques liées à leur envoi ou à leur réception (dans le schéma, la flèche pleine utilisée indique qu'on ne veut spécifier aucune caractéristique).

L'écoulement du temps est sensé s'effectuer de haut en bas.

EXEMPLE :

Le diagramme ci-dessous est sensé représenter la séquence de collaboration entre un navigateur et les différents composants d'un site web (architecture M.V.C) lors de l'appel d'une nouvelle page :



COMMENTAIRES SUR LE DIAGRAMME :

Le diagramme modélise la séquence d'exécution déclenchée par l'activation d'un lien sur le navigateur :

- L'activation du lien sur l'écran du navigateur provoque l'envoi vers le serveur HTTP d'une requête GET transportant l'URL de la page (RequeteGet(Url)) ;
- Sur le serveur, cette requête est dirigée vers le module VUES du site web. Celui-ci analyse cette requête et la convertit en un message AppelPageWEB qui est redirigé vers le module CONTRÔLE ;
- A la réception du message AppelPageWEB(P), le module CONTRÔLE récupère les arguments de la requête, puis déclenche la mise à jour du modèle à partir de ces données en expédiant le message MiseAJour (arguments) vers le module MODÈLE. Puis, le module CONTRÔLE prévient le module VUES du changement d'état par le message ChgEtatModèle() ;
- Le module VUES, sur réception du message ChgEtatModèle(), demande au module MODÈLE de lui fournir le nouvel état courant (Message DemandeEtatCourant() ;
- Le module MODÈLE répond à ce message en renvoyant à VUES le nouvel état courant (message NouvelEtatCourant()) ;
- Sur réception de ce message, le module VUES élabore les nouvelles vues, les intègre dans la page web à renvoyer au navigateur et répond à la requête de celui-ci (RéponseRequeteGet(P)).

F.APPLICATIONS :

Contrairement aux diagrammes d'états-transition et d'activités/flux qui modélisent le comportement des logiciels sous la forme d'entités abstraites (états, activités, flux d'exécution ou de données), les diagrammes de collaboration font apparaître les interactions entre les composants "réels" de l'application, ceux qui sont identifiés par la vue LOGIQUE : classes, objets, modules.

De ce fait, ce type de diagramme permet de faire le lien entre les points de vue logiques et comportementaux.

IV.ANNEXE I-STRUCTURE ET FONCTIONNEMENT D'UN ORDINATEUR:

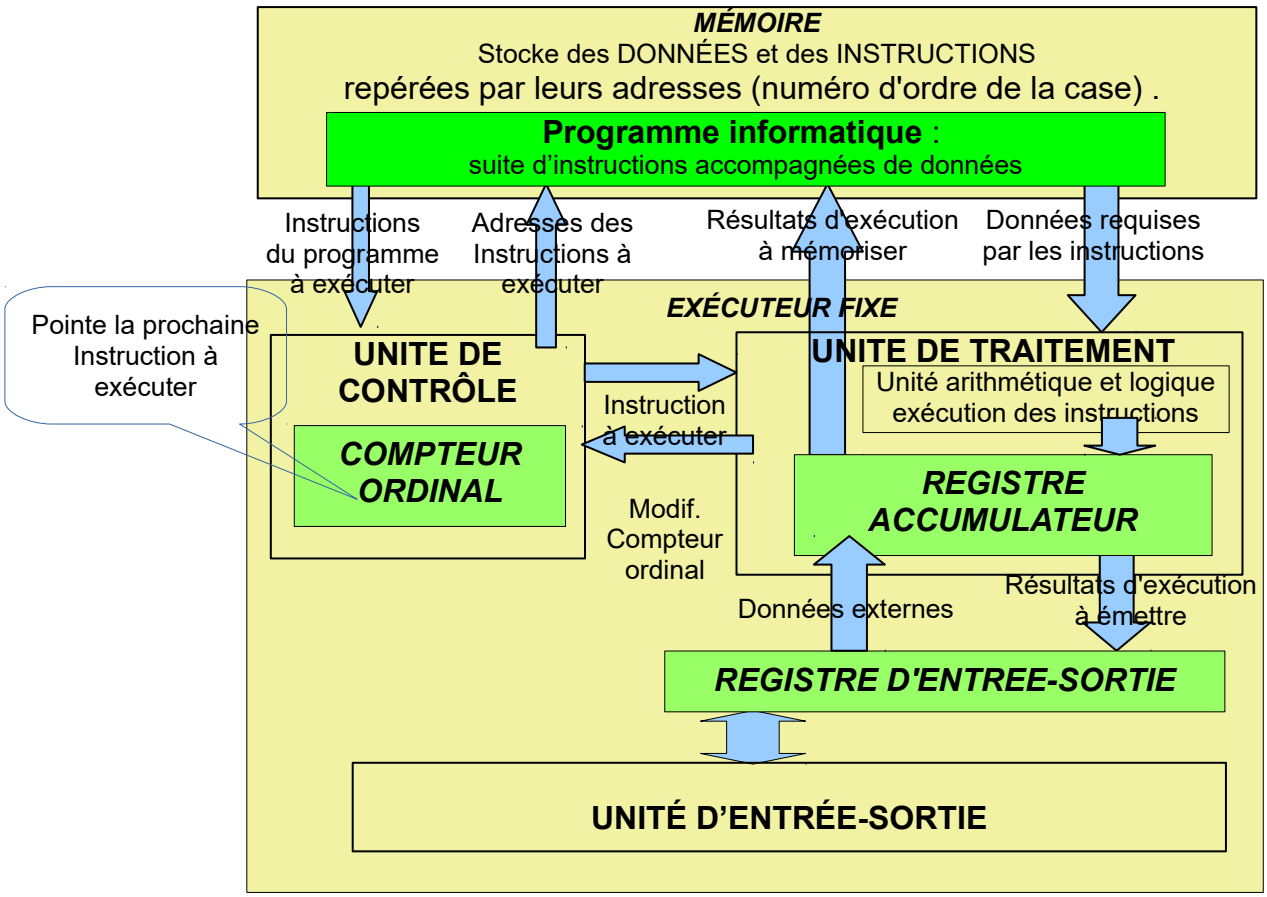
IV.1.INTRODUCTION:

Ce sous-chapitre rappelle les différentes notions nécessaires à la compréhension du fonctionnement de base d'un ordinateur.

IV.2.LE MODÈLE DE VON NEUMAN:

IV.2.1.SCHÉMA DE PRINCIPE:

Le schéma ci-dessous représente le modèle dit de VON NEUMAN. Ce modèle, qui représente une architecture LOGIQUE et non PHYSIQUE, date de 1945. Il sert de base à la quasi-totalité des CŒURS des processeurs physiques utilisés dans les unités de traitement informatiques "grand public" actuelles (micro-ordinateurs, tablettes, téléphones portables) et dans les matériels professionnels classiques (stations de travail, "supercalculateurs", etc):



Modèle de VON NEUMAN

IV.2.2.DESCRIPTION DÉTAILLEE DU MODÈLE:

IV.2.2.1.LA MÉMOIRE:

La MÉMOIRE est composée d'un certain nombre de CASES dans lesquelles peuvent être enregistrées des DONNÉES (valeurs numériques, valeurs logiques, textes) ou des INSTRUCTIONS interprétables et exécutables par l'exécuteur fixe. Chaque case mémoire est repérée par son ADRESSE, qui est un simple numéro d'ordre dans la liste des cases mémoires.

IV.2.2.2.EXÉCUTEUR FIXE ET JEUX D'INSTRUCTIONS:

IV.2.2.2.1.GÉNÉRALITÉS:

L'EXÉCUTEUR FIXE est capable d'interpréter et d'exécuter des INSTRUCTIONS correspondant à des opérations arithmétiques (+, -, *, /) ou logiques (comparaison de deux valeurs, par exemple).

Une INSTRUCTION est composée d'un CODE D'ORDRE identifiant l'opération à exécuter et d'un seul ARGUMENT D'ENTRÉE qui correspondent à une VALEURS numérique ou logique où à l'ADRESSE DE CETTE VALEUR dans la mémoire:

INSTRUCTION = <CODE D'ORDRE> <UN SEUL ARGUMENT D'ENTREE>
<ARGUMENT> = [<Valeur numérique ou logique> / <Adresse mémoire>]

REMARQUE: *le modèle de VON NEUMAN est donc basé sur un jeux d'instructions de type SISD (Simple Instruction and Simple Data).*

JEUX D'INSTRUCTIONS: Aucune description d'un jeux d'instruction particulier n'est donnée par modèle. Cependant, en général, les instructions ont besoin d'au moins un deuxième argument d'entrée (par exemple, une addition se fait entre deux valeurs). Dans ce cas, les valeurs des autres argument de l'opération seront trouvées dans l'un des registres (registre accumulateur, registre d'entrée-sortie ou encore compteur ordinal).

Ce qui suit est un exemple de code d'ordre compatible avec le modèle:

- Si le code d'ordre "LA" signifie "charger le contenu d'une case mémoire dans le registre accumulateur, l'instruction "LA 300" va charger le contenu de la mémoire d'adresse 300 dans le registre accumulateur;
- Si le code d'ordre "AVA" signifie "Ajouter une valeur à l'accumulateur", l'instruction "AVA 1000" va additionner la valeur 1000 au contenu du registre accumulateur. Le résultat est stocké dans le registre accumulateur;

- Si le code d'ordre "SA" signifie "ranger le contenu de l'accumulateur dans une cas mémoire, l'instruction "SA 301" va charger le contenu de l'accumulateur dans la case mémoire d'adresse 301;
- Si le code d'ordre "IFG" signifie "Si la valeur de l'accumulateur est supérieure à 0, aller exécuter l'instruction dont adresse est donnée en argument", l'instruction "IFG 3000" va effectuer l'une ou l'autre des actions suivantes:
  - Si la valeur de l'accumulateur est supérieure à 0, l'instruction va charger l'adresse 3000 dans le compteur ordinal, ce qui permettra de continuer l'exécution à cette adresse.
  - Sinon, l'exécution continuera à l'adresse suivant l'instruction if.

IV.2.2.2.2.DÉTAIL DE L'EXÉCUTEUR FIXE:

L'EXÉCUTEUR FIXE se divise en 3 parties: l'UNITÉ DE TRAITEMENT, l'UNITÉ DE CONTRÔLE et l'UNITÉ D'ENTRÉE-SORTIE:

- L'UNITÉ DE CONTRÔLE a pour rôle de lire les INSTRUCTIONS du programme informatique contenu dans la mémoire. Le COMPTEUR ORDINAL contient à tout instant l'adresse de la prochaine instruction à lire;
- L'UNITÉ DE TRAITEMENT a pour rôle d'exécuter les instructions fournies par l'unité de contrôle, de ranger les résultats en mémoire ou de les présenter à l'unité d'entrée-sortie, puis de charger dans le compteur ordinal l'adresse de la prochaine instruction à lire en mémoire. Le REGISTRE ACCUMULATEUR a pour rôle de stocker les valeurs des cases mémoires lues ou les résultats de calcul avant de les ranger en mémoire ou de les utiliser pour d'autres étapes de calcul.
- L'UNITÉ D'ENTRÉE-SORTIE remplit deux fonctions distinctes:
  - Émettre les données fournies par l'unité de traitement dans le REGISTRE D'ENTREE-SORTIE, afin de les acheminer vers des équipements périphériques (SORTIE d'informations);
  - Acquérir les informations fournies par les équipements périphériques (données ou signaux) et les placer dans le REGISTRE D'ENTRÉE-SORTIE (ENTRÉE d'informations).

IV.2.2.3.INSTRUCTIONS ET PROGRAMMES:

IV.2.2.3.1.ORDRE D'EXÉCUTION DES INSTRUCTIONS:

Par défaut, les suites d'instructions représentant le programme en mémoire sont exécutées dans l'ordre croissant de leurs adresses (l'instruction d'adresse N+1 est donc exécutée immédiatement à la suite de l'instruction d'adresse N). Cependant, certains codes d'ordre permettent de modifier cet ordre séquentiel en introduisant dans le COMPTEUR ORDINAL l'adresse de l'instruction qui doit être exécutée après elles. Il s'agit:

- Des instructions de RUPTURE DE SÉQUENCE INCONDITIONNELLE, dont le seul rôle est de charger la nouvelle adresse dans le compteur ordinal (le schéma logique de ces instructions est: ALLER_A <nouvelle adresse>);
- Des instructions de RUPTURE DE SÉQUENCE CONDITIONNELLE, qui ne modifient le compteur ordinal que si une condition est réalisée. Le schéma logique de ces instructions est: SI (telle condition est réalisée) ALORS aller à <nouvelle adresse> SINON on conserve l'ordre d'exécution par défaut.

Les instructions de rupture de séquence sont donc capables de modifier l'ordre d'exécution des instructions en fonction de l'état de l'environnement d'exécution.

IV.2.2.3.2.NOTION DE PROGRAMME INFORMATIQUE:

Dans le cadre du modèle de VON NEUMAN, un PROGRAMME peut être défini comme un ensemble cohérent d'INSTRUCTIONS dont l'exécution est censée aboutir à un résultat déterminé.

EXEMPLE: Supposons que la case mémoire 300 renferme une valeur supérieure à 0. Le programme suivant décrémente la valeur de la case mémoire 300 jusqu'à ce qu'elle vaille 0 (l'exemple de code d'ordre utilisé plus haut a été réutilisé ici):

Adresse	Instruction	Effet
1000	LA 300	Charge la valeur de la case 300 dans l'accumulateur.
1001	AVA -1	Retranche 1 à la valeur de l'accumulateur.
1002	SA 300	Range la valeur de l'accumulateur dans la case 300.
1003	IFG 1000	Si l'accumulateur contient une valeur plus grande que 0, renvoyer l'exécution à l'adresse 1000. Sinon, continuer par l'instruction suivante (1004).
1004		

Nous verrons par la suite que les spécifications de modèle de VON NEUMAN ne sont adaptées qu'à la MONOPROGRAMMATION. Ceci signifie qu'à un instant donné, un seul programme peut être en cours d'exécution et celui-ci ne peut être interrompu avant sa fin.

IV.2.3.NOTIONS DE TRACE D'EXÉCUTION ET DE FLUX DE CONTRÔLE:

IV.2.3.1.INTRODUCTION:

L'étude du modèle de VON NEUMAN va nous permettre de dégager deux notions très importantes pour la compréhension du fonctionnement des ordinateurs: il s'agit de la TRACE D'EXÉCUTION d'un exécuteur fixe et du FLUX DE CONTRÔLE d'un programme. La première notion caractérise l'activité de l'EXÉCUTEUR FIXE tandis que la deuxième est liée à la LOGIQUE ALGORITHMIQUE du programme.

IV.2.3.2.TRACE D'EXÉCUTION D'UN EXÉCUTEUR FIXE:

Nous avons vu, lors de l'étude du modèle de VON NEUMAN, que l'EXÉCUTEUR FIXE est l'entité logique capable d'interpréter et d'exécuter les instructions contenues dans la mémoire de la machine. Dans les CPU actuels, la notion de CŒUR est sensiblement équivalente à celle d'EXÉCUTEUR FIXE dans le modèle de VON NEUMAN.

Dans la logique du modèle, l'exécuteur fixe ne peut exécuter qu'une instruction à la fois. De ce fait, les instructions exécutées par un exécuteur fixe pendant un créneau temporel donné forment une liste ordonnée que nous appellerons TRACE D'EXÉCUTION.

IV.2.3.3.FLUX DE CONTRÔLE D'UN PROGRAMME:

Nous avons vu également que par défaut, un exécuteur fixe exécute les instructions dans l'ordre croissant de leurs adresses mémoires, mais qu'il existait des instructions, dites de "ruptures de séquence", capable de rompre cet ordre d'exécution de manière conditionnelle ou inconditionnelle.

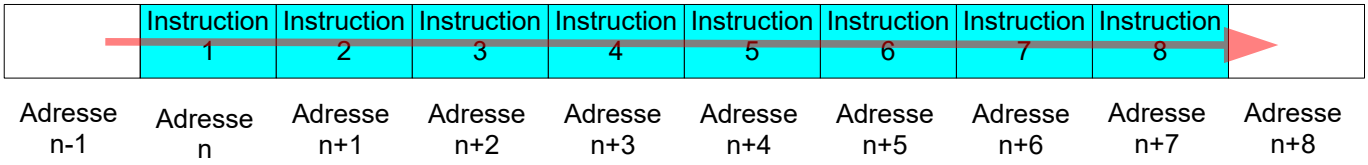
Nous appellerons FLUX ou FLOT D'EXÉCUTION (ou encore FLUX ou FLOT DE CONTRÔLE) l'ordre dans lequel les instructions d'un programme doivent être exécutées en fonction de l'état du contexte d'exécution. Cet ordre tient donc compte du comportement des instructions de rupture de séquence en fonciton des valeurs du contexte d'exécution.

EXEMPLE: supposons qu'un programme de 8 instructions soit chargé dans les adresses n à n+7 de la mémoire et que la 4e instruction de ce programme soit une instruction de rupture de séquence conditionnelle dont le schéma est:

SI (Accumulateur < 10) ALORS Aller à l'adresse n+7

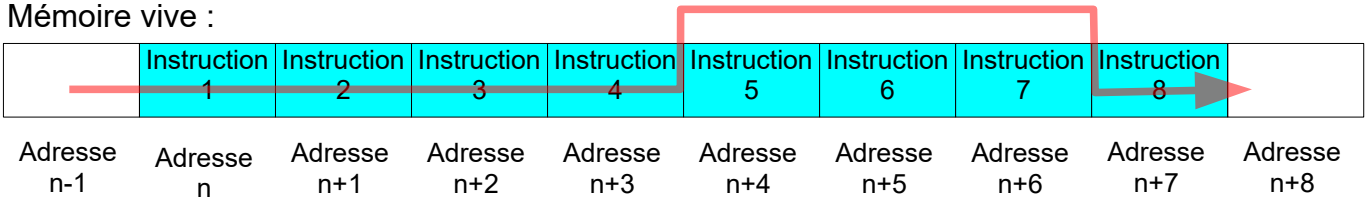
Dans le cas ou l'accumulateur vaut 10 ou plus lors de l'exécution de l'instruction n°4, la trace d'exécution de l'exécuteur fixe sera matérialisée par la ligne rouge:

Mémoire vive :



En revanche, dans le cas ou l'accumulateur vaut moins de 10 lors de l'exécution de l'instruction n°4, la trace d'exécution de l'exécuteur fixe sera matérialisée par la nouvelle ligne rouge:

Mémoire vive :



Le FLUX D'EXÉCUTION (ou de CONTRÔLE) D'UN PROGRAMME dépend donc de l'état du contexte d'exécution. L'ensemble des flux d'exécution possibles est souvent appelé GRAPHE DE CONTRÔLE du logiciel.

IV.2.3.4.REMARQUE:

En monoprogrammation, la TRACE D'EXÉCUTION se confond avec un des flux du GRAPHE DE CONTRÔLE: celui qui correspond à l'état de l'environnement d'exécution. Nous verrons que ce n'est pas forcément le cas en multiprogrammation.

IV.2.4.FONCTIONNEMENT DU MODÈLE DE VON NEUMAN:

IV.2.4.1.CHARGEMENT DU PROGRAMME EN MÉMOIRE:

Le modèle suppose que le programme à exécuter a été préalablement chargé en mémoire. Dans une machine réelle, la mémoire est généralement chargée à partir d'un fichier situé sur un disque dur.

IV.2.4.2.LANCEMENT DU PROGRAMME:

Le modèle suppose qu'il suffit d'indiquer à l'exécuteur fixe la première instruction du programme à exécuter. Pour cela, on peut considérer qu'il suffit de charger dans le COMPTEUR ORDINAL l'adresse mémoire de cette première instruction.

Remarquons que le modèle de VON NEUMAN n'aborde pas le fait que l'exécution puisse être démarrée ou arrêtée une fois qu'elle est lancée. Pourtant, on peut supposer que si l'exécuteur dépasse les limites de la zone allouée aux instructions du programme, il va être amené à essayer d'exécuter des cases dont le contenu n'est pas forcément une instruction. Sur une machine réelle, cela peut être la cause de dysfonctionnements très graves (appelés familièrement "plantages"), dont certains sont exploités par les hackers pour infiltrer des virus (alarme stack overflow, par exemple).

De ce fait, dans une machine "réelle", l'exécution est toujours aiguillée vers une "boucle infinie" à la fin d'un programme (une boucle infinie est une instruction de rupture de séquence inconditionnelle qui reboucle l'exécution sur elle-même).

IV.2.4.3.DÉROULEMENT DE L'EXÉCUTION D'UNE INSTRUCTION:

Il correspond à l'algorithme suivant:

DÉBUT BOUCLE D'EXÉCUTION:

L'exécuteur fixe va récupérer l'instruction pointée par le compteur ordinal dans la mémoire et la traiter. Pour cela, il exécute les opérations suivantes:

- Il récupère les valeurs des ARGUMENTS attachés à l'instruction: ceux-ci sont soit directement des DONNÉES à utiliser, soit des ADRESSES de données. Dans le dernier cas, il va récupérer ces données dans la mémoire;
- Il récupère le CODE D'ORDRE de l'instruction (c'est à dire le type d'opération à appliquer aux valeur correspondant aux arguments);
- Il incrémente le COMPTEUR ORDINAL de 1 (par défaut, passage à l'instruction suivante);
- Il exécute alors l'instruction en utilisant les valeurs d'arguments et éventuellement les valeur contenues dans les REGISTRES et suivant le cas, il place les résultats dans le REGISTRE ACCUMULATEUR, le REGISTRE D'ENTRÉE-SORTIES ou le COMPTEUR ORDINAL (instructions de rupture de séquence) afin qu'elles soient émises par l'unité d'entrée-sorties.

RETOUR AU DÉBUT DE LA BOUCLE.

IV.2.5.PANORAMA DES ÉVOLUTIONS PAR RAPPORT AU MODÈLE DE VON NEUMAN:

IV.2.5.1.CONCERNANT L'ARCHITECTURE MATÉRIELLE:

Le modèle de VON NEUMAN date de 1945. Les premiers ordinateurs réellement opérationnels (construits dans les années 1945 à 1955) s'inspiraient très fidèlement de ce modèle. Au fil du temps et des progrès de la technologie, de nombreux dispositifs matériels et logiciels ont permis d'améliorer les fonctionnalités et les performances des ordinateurs sans vraiment remettre en cause les principes de base. Nous citerons:

- La multiplication des registres intermédiaires (registres accumulateurs supplémentaires), qui permet d'optimiser le temps d'exécution des calculs numériques en évitant le stockage en mémoire des résultats intermédiaires;
- Les modes d'adressage sophistiqués (adressage indirect, indexé, indirect-indexé, etc.) qui, entre autres, permettent de mieux gérer les traitements itératifs;
- Les mécanismes de MÉMOIRES CACHES (ou antémémoires) permettant d'optimiser le temps d'accès aux données en mémoire;
- Les mécanismes de PIPELINES consistant à découper le déroulement des instructions en plusieurs phases, ce qui permet de commencer l'exécution d'une instruction sans attendre que la précédente soit terminée;
- Les jeux d'instructions admettant plusieurs arguments explicites (par exemple, les Simple Instruction, Multiple Data: SIMD, facilitant les traitements "vectoriels" (ordinateurs vectoriels);
- Les CPU hébergeant plusieurs exécuteurs fixes (machines multi-cœurs), qui permettent de PARALLÉLISER l'exécution des tâches;
- Les machines dites MULTIPROCESSEURS car elles sont dotées de plusieurs CPU pouvant travailler simultanément et permettre ainsi l'exécution simultanée de plusieurs programmes;
- Ils faut ajouter à cette énumération les progrès accomplis dans les domaines de la commutation dans les circuits intégrés, les temps de latence des mémoires vives, les mécanismes d'entrée-sortie, etc.

Ces ajouts, qui concernent l'amélioration des performances des matériels et, dans une démarche de conception, les possibilités qu'ils offrent doivent être examinées en regard des performances demandées.

IV.2.5.2.CONCERNANT LA GESTION DES ÉVÉNEMENTS:

IV.2.5.2.1.INTRODUCTION:

Le modèle de VON NEUMAN ne décrit aucun mécanisme permettant d'interrompre l'exécution du programme en cours pour aller traiter au plus tôt la survenue d'un événement. Lorsque l'exécution d'un programme est lancée, celle-ci suit l'ordre logique des instructions en fonction du contexte jusqu'à la fin de l'algorithme.

Ceci implique que la prise en compte de la survenue d'un événement externe (la réception de données, par exemple) ne peut être traité que de manière SYNCHRONE par rapport à l'exécution du programme en cours. De ce fait,

- Soit le traitement de l'événement doit être prévu par ce programme;
- Soit il doit attendre le fin du programme en cours et le lancement d'un autre programme consacré à ce traitement.

Ce mode de fonctionnement est appelé MONOPROGRAMMATION.

IV.2.5.2.2.MÉCANISME DES INTERRUPTIONS PRIORITAIRES MATÉRIELLES:

A.DÉFINITION:

D'un point de vue fonctionnel, le mécanisme des INTERRUPTIONS PRIORITAIRES MATERIELLES consiste à suspendre temporairement l'exécution du flux de contrôle d'un programme informatique afin d'aller exécuter un programme plus prioritaire que le premier (ce nouveau programme est appelé SERVICE D'INTERRUPTION). En fin d'exécution de ce dernier, l'exécution du premier programme peut être reprise au point où il avait été interrompu.

B.MÉCANISME PHYSIQUE:

D'un point de vue technique, le mécanisme consiste, lorsqu'un événement survient, à charger dans le compteur ordinal (en "écrasant" l'adresse d'instruction qui s'y trouve), l'adresse d'un programme dont la fonction est de traiter cet événement (service d'interruption, encore appelé gestionnaire d'événement). De ce fait, à la fin de l'exécution de l'instruction en cours, la trace d'exécution de l'exécuteur sera dérouté vers ce service d'interruption.

Comme leur appellation l'indique, les interruptions prioritaires MATÉRIELLES sont provoquées par des causes liées au fonctionnement des équipements matériels internes ou périphériques de la machine. Il peut s'agir:

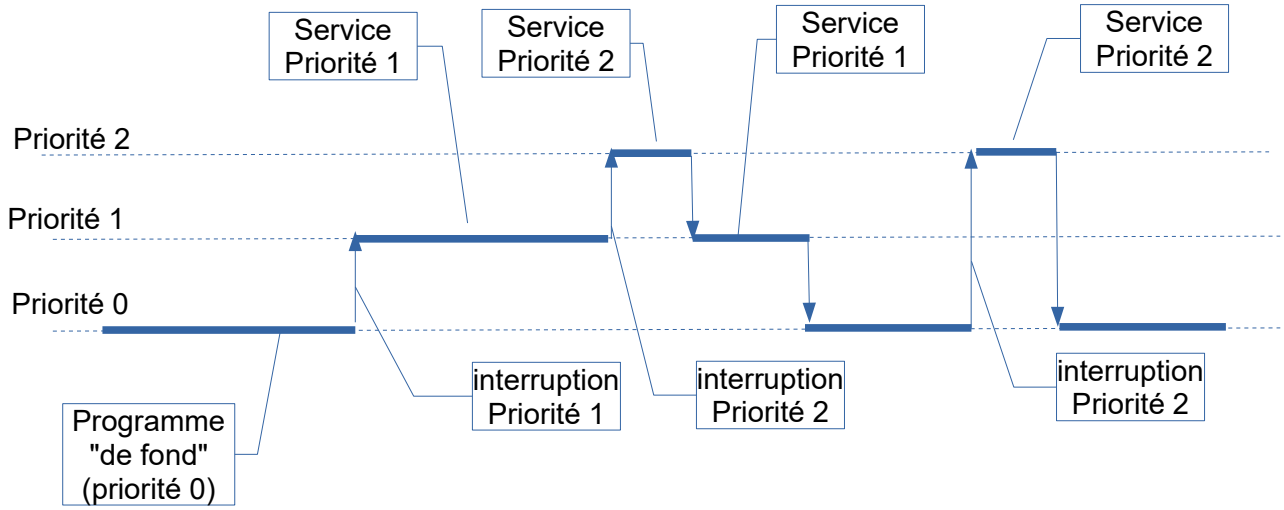
- De la réception de données par l'unité d'entrée-sortie de la machine;
- De la détection de signaux d'état en provenance d'un péripériques;
- De la détection d'alarmes matérielles en provenance de la machine elle-même (ex: signalisation d'un défaut d'alimentation électrique);

- Etc.

C.HIÉRARCHIE DES PRIORITÉS:

Le mécanisme physique des interruptions matérielles affecte à chaque cause d'interruption une PRIORITÉ (dite "priorité matérielle", car la priorité d'une cause d'interruption dépend de la connexion physique qui transmet cette cause à la machine). Cette hiérarchisation justifie le terme d'INTERRUPTION PRIORITAIRE). De ce fait, un service d'interruption peut lui-même être interrompu pour aller exécuter un service d'interruption plus prioritaire que lui.

Le schéma ci-dessous donne une idée de ce mécanisme:



COMMENTAIRES:

- Un "programme de fond" s'exécute en priorité 0;
- Une interruption de priorité 1 interrompt le programme de fond avant sa fin pour lancer un service de priorité 1;
- Le service de priorité 1 s'exécute jusqu'à ce survienne une interruption de priorité 2;
- Le service de priorité 1 est alors interrompu pour aller exécuter un service de priorité 2.
- Celui-ci se déroule jusqu'à sa fin. L'exécuteur retourne alors au service de priorité 1 et le termine. A sa fin, l'exécuteur retourne à la tâche de fond;
- Cette tâche de font est de nouveau interrompue avant sa fin par une interruption de priorité 2. Ce service est alors lancé;
- En fin d'exécution de ce service, l'exécution du programme de fond est reprise jusqu'à sa fin.

REMARQUE: INTERRUPTIONS LOGICIELLES: Nous verrons plus tard qu'il existe d'autres interruptions qui peuvent être provoquées par les logiciels eux-mêmes ou par les systèmes d'exploitation. C'est le cas, par exemple pour les mécanismes d'I.P.C (Inter-

Process Communication). Dans ce cas, le mécanisme est entièrement logiciel et géré par les ordonnanceurs de tâches des systèmes d'exploitation.

D.GESTION DU CONTEXTE D'EXÉCUTION

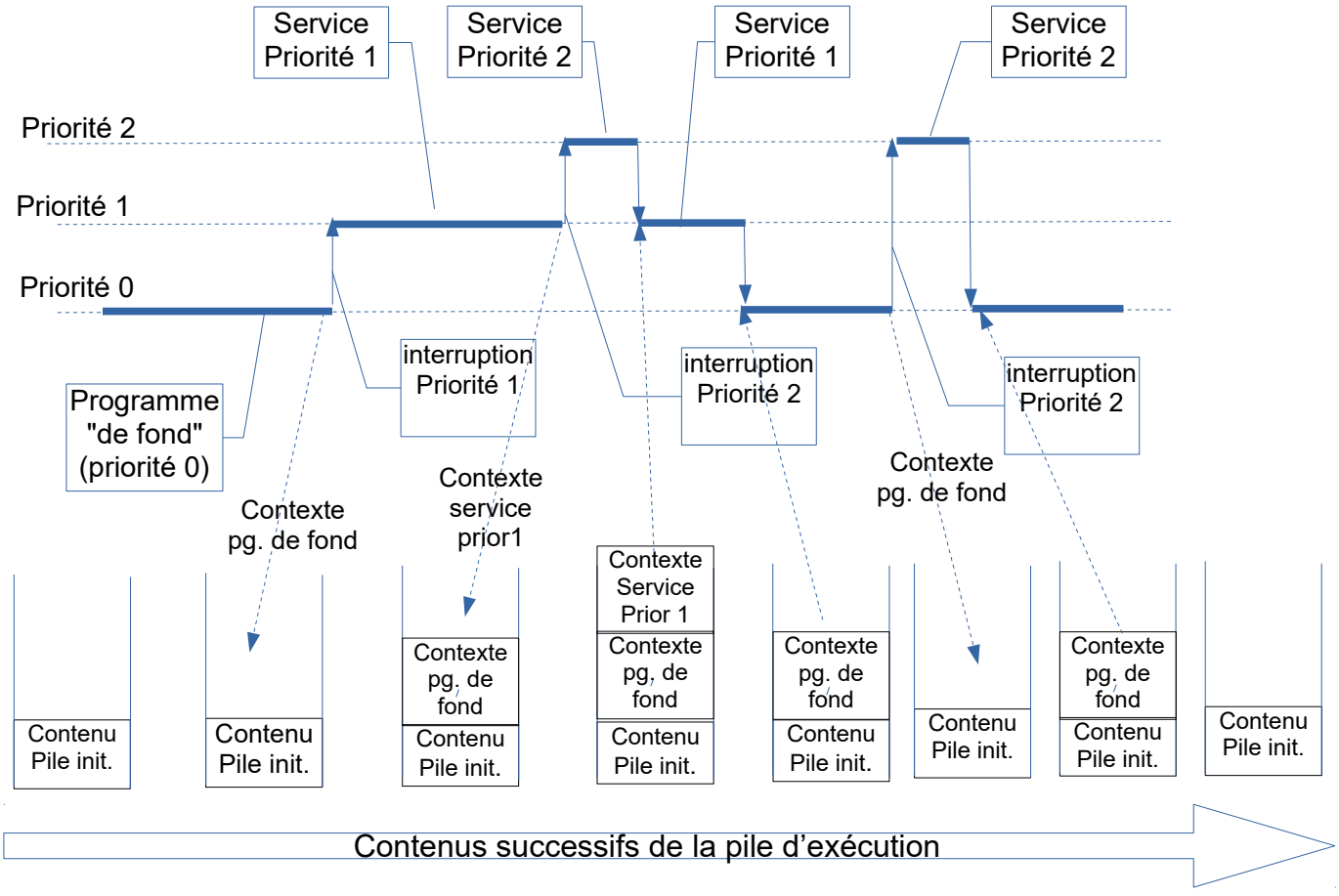
NOTA: *Le contexte d'exécution représente les valeurs courantes des registres de travail.*

Quelle que soit la nature de l'interruption, le service d'interruption devra, à la fin de son exécution, renvoyer l'exécuteur vers l'adresse où elle a été abandonnée dans le programme interrompu, tout en restaurant le CONTEXTE D'EXÉCUTION aux valeurs qui étaient les siennes au moment de l'interruption. Ceci implique que ces différentes valeurs aient été sauvegardées au moment de l'interruption.

Cette gestion du contexte d'exécution est assurée grâce à la PILE D'EXÉCUTION. Il s'agit d'une pile en mémoire vive gérée en mode FI/FO (First in, first out):

- Lorsqu'une interruption survient, le contexte d'exécution du programme interrompu est sauvegardé en haut de la pile avant de démarrer le service d'interruption;
- Lorsqu'un service d'interruption se termine, le contexte d'interruption situé en haut de pile est dépilé et placé dans le contexte d'exécution courant avant de redémarrer l'exécution.

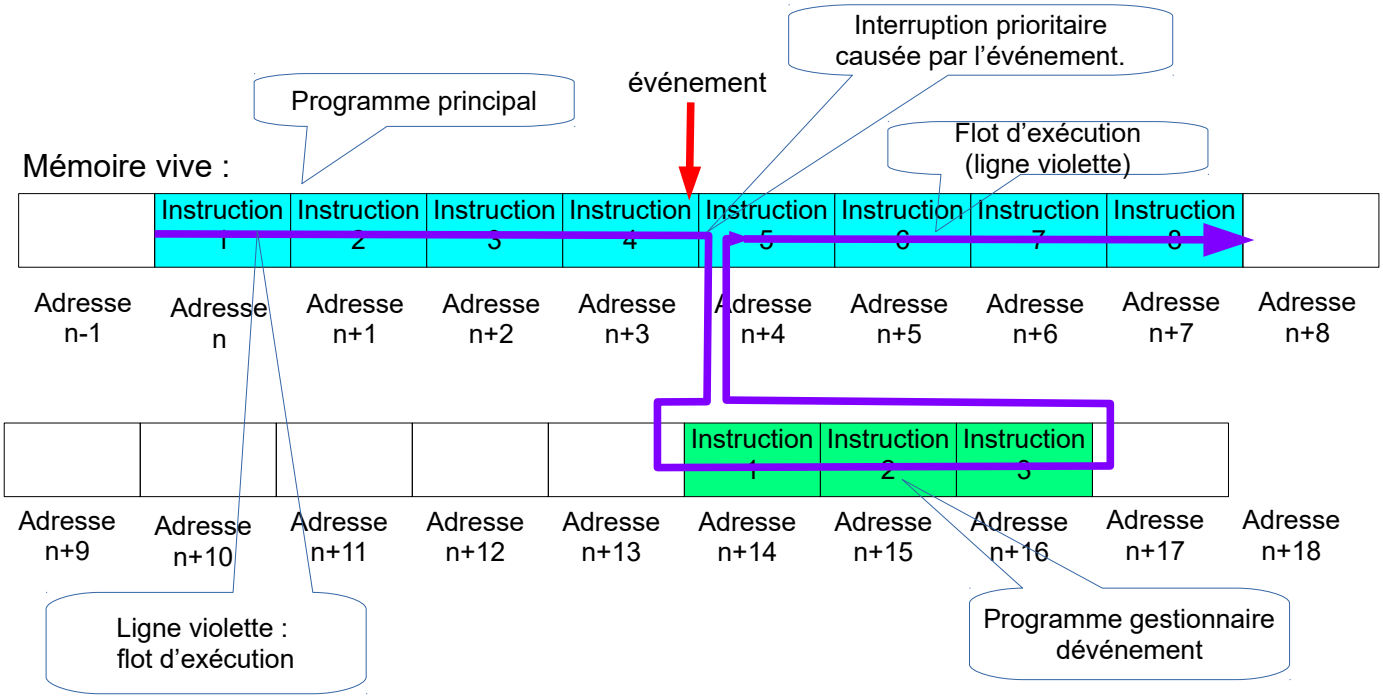
Le schéma suivant représente l'évolution du contenu de la pile d'exécution en fonction des appels prioritaires traités par le logiciel:



IV.2.5.2.3.NOTION DE MULTIPROGRAMMATION:

Contrairement aux ruptures de séquence provoquées par les instruction conditionnelle, celles qui sont provoquées par les INTERRUPTIONS PRIORITAIRES ne sont pas prévisibles par la logique interne du programme en cours puisqu'elles résultent de l'apparition d'un événement sans lien avec celui-ci. L'exécution est provisoirement ou définitivement abandonnées au profit d'un autre programme destiné à traiter l'événement (gestionnaire d'événement).

De ce fait, l'exécuteur passe d'un programme à l'autre (on peut également dire que le système d'exploitation réattribue la ressource "exécuteur fixe" au second programme). Le schéma ci-dessous illustre ce type de fonctionnement:



REMARQUES:

- Dans ce cas, le chemin d'exécution de l'exécuteur fixe n'est plus identique au FLUX DE CONTRÔLE du programme;
- C'est ce mode de fonctionnement que l'on appelle MULTIPROGRAMMATION, car il permet à un exécuteur fixe de mener d'une manière apparemment simultanée l'exécution de deux ou plusieurs programmes (un programme "de fond" et des programmes à caractères prioritaires). C'est lui qui permet à un ordinateur de réagir "en temps réel" aux sollicitations extérieures au programme en cours (réception de données ou de signaux, requêtes en provenance d'utilisateurs, etc.).

IV.2.5.3.CONCERNANT LES SYSTÈMES D'EXPLOITATION:

IV.2.5.3.1.GENÈSE DES SYSTÈMES D'EXPLOITATION:

Les premiers ordinateurs s'inspirant du modèle de VON NEUMAN étaient livrés sans système d'exploitation. De ce fait, au démarrage de la machine, la mémoire était vide et aucun système de chargement automatique n'était proposé.

Pour charger en mémoire le contenu du programme qu'ils voulaient exécuter, les utilisateurs devaient introduire dans cette mémoire, grâce à des dispositifs de saisie manuelle très sommaires, une courte séquence d'instruction appelée AMORCE ou BOOTSTRAP qui permettait de lire ce programme sur un support physique (dans un premier temps, il s'agissait de lecteurs de cartes perforées ou de rubans perforé, plus tard, de disques durs), puis de lancer l'exécution de ce programme.

Ce procédé, qui devait être répété à chaque changement de programme, se révélait évidemment fastidieux et rigide. De ce fait, avec le développement de l'utilisation de disques durs pouvant stocker de nombreux programmes, l'idée se fit jour de charger à l'aide du bootstrap non pas le programme à exécuter, mais un programme interactif (interface en ligne) dont le but était de permettre à l'utilisateur de choisir le programme qu'il voulait charger à partir du disque dur: ces programmes "chargeurs" étaient les ancêtres des premiers systèmes d'exploitation.

REMARQUE: *Ces interfaces en ligne utilisaient au début une TÉLÉTYPE munie d'un clavier (pourla saisie des commandes des utilisateurs) et pouvant imprimer les messages en réponse aux commande sur un support papier. Plus tard, les télétypes furent remplacées par des consoles vidéo alphanumériques, puis graphiques.*

Plus tard, les systèmes d'amorçage automatique (B.I.O.S), ont permis de charger automatiquement, au démarrage et à partir de mémoires de masse à accès rapide, des logiciels qui non seulement assuraient l'interfaçage graphique avec les utilisateurs, mais permettaient également de gérer l'exécution simultanée de plusieurs applications, ainsi que les autres activités liées à la gestion des événements externes et internes, à la gestion des alarmes, etc. Ces logiciels ont alors mérité l'appellation "SYSTÈMES D'EXPLOITATION".

IV.2.5.3.2.RAPPELS SOMMAIRES SUR LES SYSTÈMES D'EXPLOITATION

Un système d'exploitation (Operating System: OS) est un LOGICIEL qui est chargé en mémoire et lancé au démarrage de la machine par le système d'amorçage automatique (BIOS).

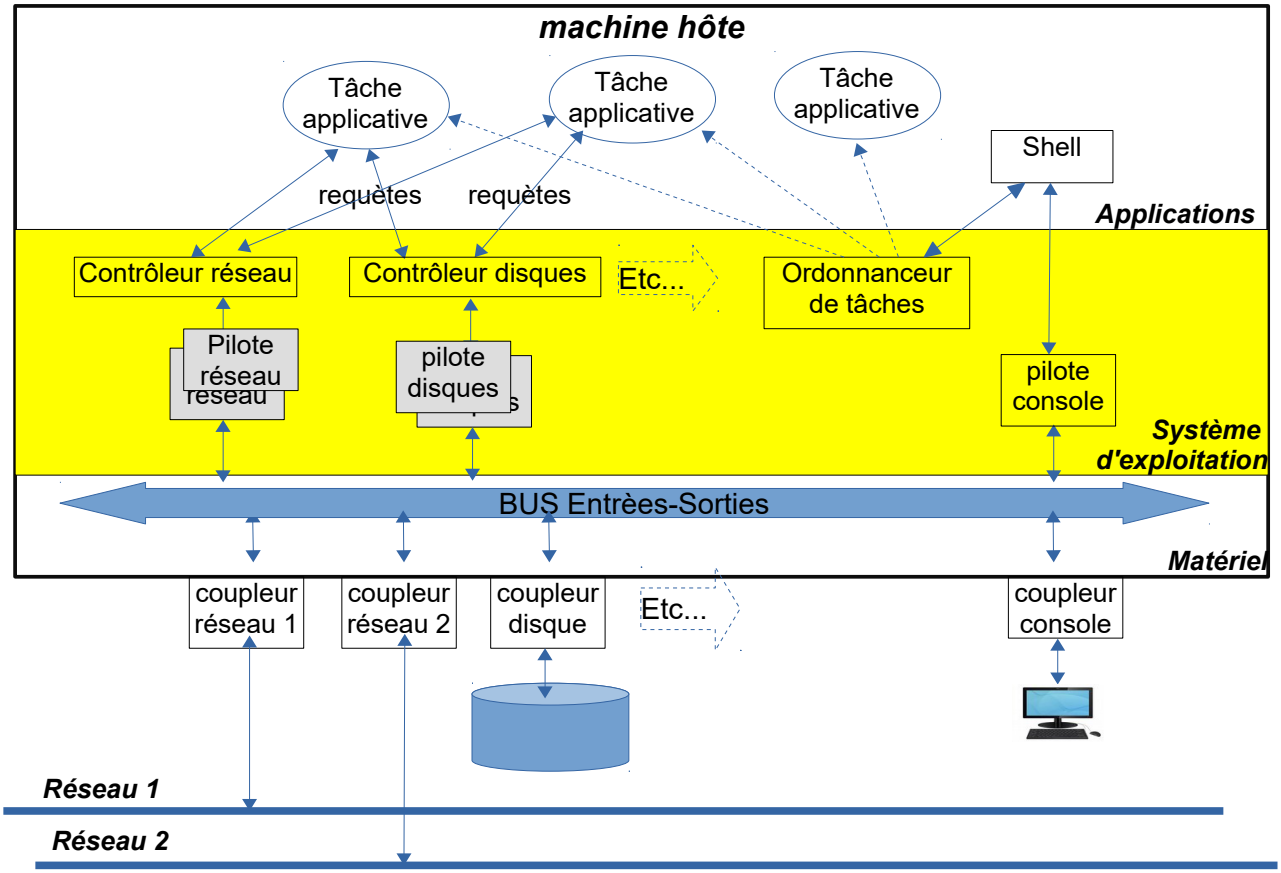
Ses deux principales missions sont:

- L'ordonnancement de l'exécution des différentes TÂCHES correspondant aux applications des utilisateurs;

- La gestion de la communication entre les applications et les équipements composant l'infrastructure matérielle.

Les utilisateurs communiquent avec le système d'exploitation grâce à des logiciels d'interfaçage (shells ou invites de commande) qui leur permet de saisir des COMMANDES SYSTÈME qui leur permettent de lancer des applications ou de paramétrer les dispositifs.

Le schéma ci-dessous représente l'architecture logique d'un système d'exploitation à l'intérieur de sa machine hôte:



COMMENTAIRES:

- Le composant du système d'exploitation qui gère l'ordonnancement des TÂCHES s'appelle ORDONNANCEUR en français, ou encore SCHEDULER en anglais;
- La gestion des communications entre les applications et les équipements de l'infrastructure matérielle est assurée par les systèmes d'exploitation à l'aide de deux niveaux (ou couches) de composants logiciels: les CONTRÔLEURS et les PILOTES:

- Les CONTRÔLEURS supportent les traitements communs à la gestion d'un certain type d'équipements: réseaux, pool de disques durs, périphériques SCSI, etc. Ils gèrent les protocoles de communication communs à tous les équipements de ce type (protocole TCT/IP, RAID, etc.);
- Les PILOTES permettent d'assurer la liaison entre le CONTRÔLEUR et un équipement particulier (par exemple: une "carte" réseau ETHERNET, une "carte" graphique VGA ou HDMI etc.).

IV.2.6.ARCHITECTURE INTERNE D'UN PROCESSEUR ACTUEL:

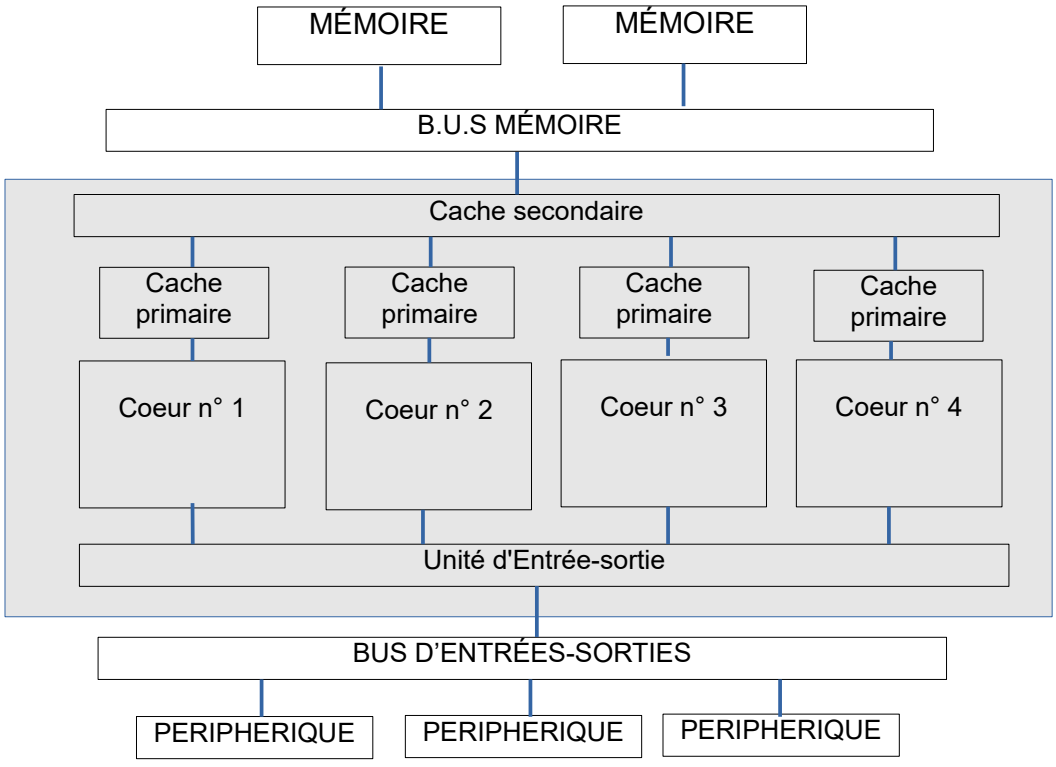
IV.2.6.1.LES CPU MULTICŒURS:

IV.2.6.1.1.INTRODUCTION:

Jusqu'au début des années 2000, les progrès constants dans la conception et la réalisation des microprocesseurs se traduisaient par une augmentation rapide des capacités de traitement de ceux-ci (de quelques dizaines de millions d'instructions par seconde en 1980 à plus de trois milliards au début de la décennie 2000). La plupart des machines ne comprenaient alors qu'un seul exécuteur fixe. Cependant, ces progrès n'étaient acquis qu'au prix d'une augmentation importante de la dissipation d'énergie. De ce fait, la progression finit par se heurter à des obstacles insurmontables. La plupart des constructeurs de CPU se tournèrent alors vers des architectures de CPU comprenant plusieurs exécuteurs fixes (plusieurs "cœurs") de moindre puissance, mais dont les actions pouvaient être conjuguées pour augmenter la puissance d'exécution de l'ensemble.

IV.2.6.1.2.ARCHITECTURE GÉNÉRALE:

Dans un ordinateur actuel, un PROCESSEUR (encore appelé CPU pour Central Processing Unit) est un équipement qui supporte un ou plusieurs EXÉCUTEURS FIXES (les CŒURS) et assure leur liaison avec les autres organes de cet ordinateur (mémoire, BUS d'entrée-sorties, etc.). Le schéma ci-dessous décrit une des architectures les plus répandues actuellement:



COMMENTAIRES:

- Le schéma ci-dessus représente un processeur à 4 cœurs que nous supposerons dotés d'un jeux d'instructions de type RIS (Reduced Instruction Set). La plupart des ordinateurs "grand public" correspondent actuellement (en 2020) à ces caractéristiques;
- Dans un tel processeur multicœur, chacun des cœurs peut exécuter une instruction indépendamment des autres cœurs. Ce processus peut donc exécuter simultanément 4 instructions;
- En revanche, le choix d'un jeu d'instruction de type RIS ne permet à chaque cœur de traiter qu'une donnée à la fois;
- Notons que chaque cœur est doté d'une mémoire cache primaire et qu'il existe un cache secondaire commun. Les mémoires caches, qui sont des mémoires à accès très rapide, permettent de stocker les données les plus fréquemment utilisées par chaque cœur dans les caches auxquels il a accès, ce qui réduit le nombre des accès de ces cœurs à la mémoire centrale ainsi que les problèmes dus aux accès simultanés de ces cœurs sur la même donnée. Les caches ont donc pour effet d'accélérer la vitesse d'exécution.

Actuellement, les processeurs multicœurs sont très répandus dans les applications "grand public" car ils permettent de pallier le plafonnement des progrès en matière de vitesse d'exécution des "puces": 4 processeurs de 1,5 Gh d'horloge présentent potentiellement une puissance CPU beaucoup plus importante qu'un seul processeur de 3,4 Ghz (à condition que les traitements à exécuter s'y prêtent et que la programmation soit adaptée). D'autre part, la dissipation d'énergie est moindre: en effet, la dissipation d'énergie d'un cœur quadruplant quand sa puissance double, deux cœurs de 1,5 Ghz dissipent à eux deux 2 fois moins d'énergie qu'un cœur de 3,0 Ghz.

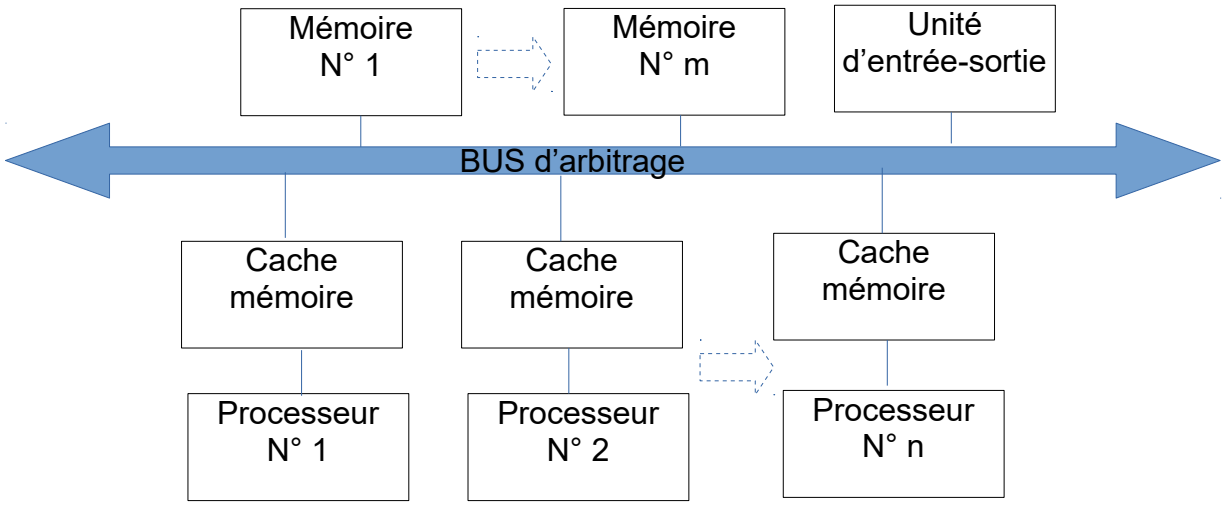
IV.2.6.2.LES UNITÉS DE TRAITEMENT MULTI PROCESSEURS:

IV.2.6.2.1.CARACTÉRISTIQUES:

Ce qui caractérise un ordinateur multiprocesseurs est qu'il est équipé de plusieurs PROCESSEURS PHYSIQUES (CPU), chacun de ces processeurs pouvant être équipé d'un ou plusieurs CŒURS.

IV.2.6.2.2.STRUCTURE GÉNÉRALE:

De nombreuses architectures MULTIPROCESSEURS sont proposés par les constructeurs. Le schéma ci-dessous représente une architecture multiprocesseurs SYMÉTRIQUE, caractérisée par le fait que tous les processeurs sont à égalité en ce qui concerne les possibilités d'accès aux différentes mémoire vives et aux entrées-sorties:



REMARQUES:

- Le BUS D'ARBITRAGE peut être un bus industriel (V.M.E, P.C.I Express, etc.). Dans ce cas, les processeurs, mais aussi les unités de mémoire ou les périphériques se présentent comme des cartes électroniques enfichable sur ce BUS, ce qui permet de construire des machines "sur mesures", dont la puissance peut facilement être augmentée en cas de besoin;
- Dans ce type d'architecture, la communication entre processus peut être assurée par mise en commun d'une mémoire vive (système de BOÎTE AUX LETTRES) ou par communication à travers le BUS;
- Les systèmes multiprocesseurs sont en général des solutions coûteuses qui conviennent pour des applications critiques en termes de puissance de calcul ou de rapidité d'exécution.

V.ANNEXE II - TRAITEMENT LOGICIEL MULTITACHES:

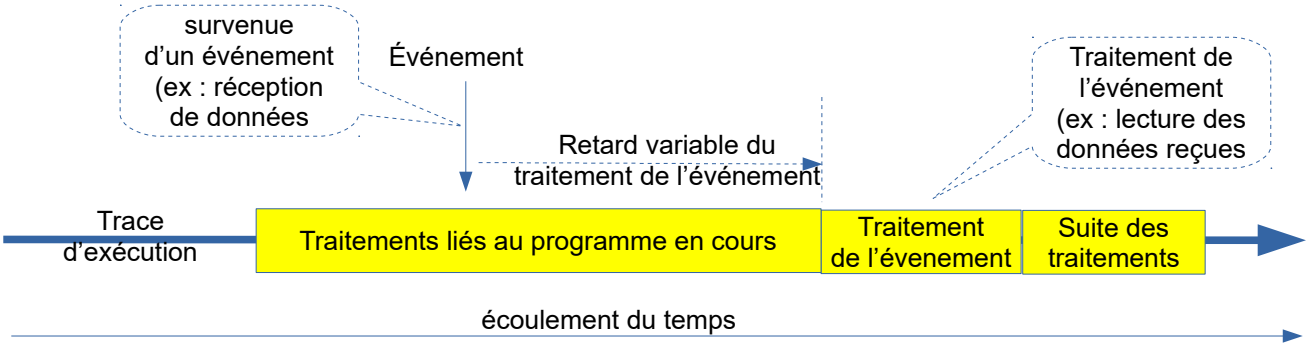
V.1.DIFFÉRENCE ENTRE TRACE D'EXÉCUTION ET FLUX D'EXÉCUTION:

V.1.1.CAS DE LA MONOPROGRAMMATION:

Nous avons vu précédemment que la MONOPROGRAMMATION correspond à un mode de fonctionnement dans lequel la survenue des événements ne provoque pas d'interruption de la trace d'exécution en cours. Les événements sont traités d'une manière SYNCHRONE par rapport à l'exécution du programme.

De ce fait, en mode monoprogrammation, la trace d'exécution de l'exécuteur fixe se confond avec le flux d'exécution (ou de contrôle) du programme en cours.

EXEMPLE: le schéma ci-après illustre une exécution en monoprogrammation:



Le schéma matérialise l'exécution d'un programme (en jaune) alors que survient un événement (par exemple, la réception de données): l'événement n'est traité qu'avec un certain retard, lorsque le flux de contrôle du programme atteint le code destiné à ce traitement. Le gestionnaire d'événement doit donc appartenir au programme en cours.

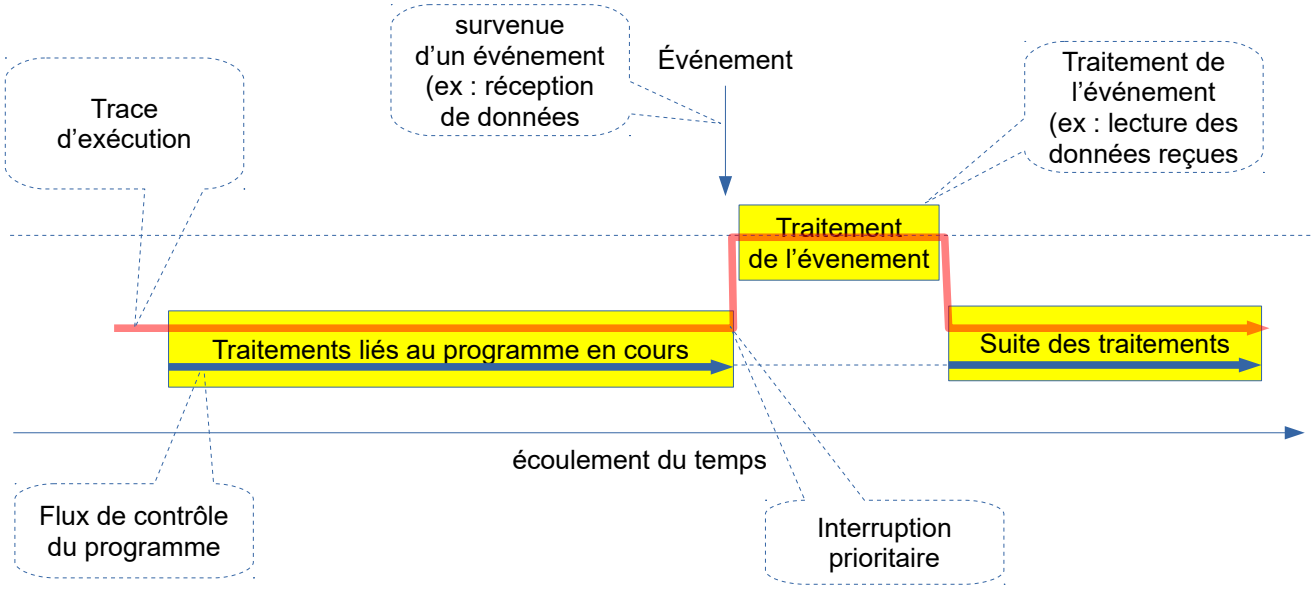
REMARQUE: Dans le cas où l'événement n'est par encore survenu lorsque le flux de contrôle du programme atteint le code destiné à le traiter, l'événement peut même être ignoré, sauf si le code objet du programme effectue une itération de ce code jusqu'à ce que l'événement soit présent:

```
BOUCLE DE TRAITEMENT
        SI (l'événement a eu lieu) ALORS
                Traiter l'événement;
                Sortir de la boucle;
        FINSI
FIN BOUCLE
```

V.1.2.CAS DE LA MULTIPROGRAMMATION:

En mode MULTIPROGRAMMATION, la survenue des événements provoque des INTERRUPTIONS PRIORITAIRES qui rompent la séquence d'exécution en renvoyant celle-ci vers une adresse mémoire correspondant au début du programme de traitement de l'événement. L'événement peut donc être traité d'une manières ASYNCHRONE par rapport au programme principal, et potentiellement "au plus tôt". De ce fait, la trace d'exécution ne se confond plus forcément avec le flux de contrôle de celui-ci.

EXEMPLE: le schéma ci-après illustre une exécution en multiprogrammation:



Dans ce cas d'exécution, la trace d'exécution (en rouge) ne se confond plus avec le flux de contrôle du programme (en bleu sombre). La survenue de l'événement provoque une INTERRUPTION PRIORITAIRE qui renvoie l'exécution vers un autre programme destiné à traiter l'événement (gestionnaire d'événement). L'événement est donc traité EN PRIORITÉ par rapport au programme en cours et AU PLUS TÔT.

Lorsque le traitement de l'événement a été effectué, l'exécution du programme en cours est en général reprise à l'endroit où elle avait été interrompue, sauf si l'événement correspond à la détection d'un dysfonctionnement rendant impossible cette reprise.

Lorsque le traitement est effectué dans le cadre d'un système d'exploitation monotâche (comme par exemple, les premiers systèmes MS/DOS), les interruptions prioritaires sont déclenchées uniquement par des événements d'origine matérielle (composants de la périphérie matérielle) ou liés au systèmes d'exploitation (Timers, détection d'anomalies ou d'exceptions). Dans ce cas, les logiciels de traitement de ces événements sont appelés PILOTES (drivers). Par exemple, lors de la réception de données en provenance d'un réseau, la trace d'exécution sera déroutée vers le PILOTE RÉSEAU correspondant au type de liaison. Le gestionnaire d'événement n'appartient donc plus forcément au programme en cours, ce qui justifie la qualification de MULTIPROGRAMMATION.

V.2.PROGRAMMATION MULTITÂCHES:

V.2.1.NOTION DE TÂCHE:

En informatique, le terme TÂCHE désigne l'ensemble des activités logicielles nécessaires à l'obtention d'un résultat donné.

Par exemple:

1. L'ensemble des traitements nécessaires pour imprimer un document donné est qualifié de tâche d'impression. Le logiciel contrôleur d'une imprimante exécute les tâches d'impression en attente d'impression dans l'ordre de la file d'attente;
2. La tâche d'acquisition des mesures d'un capteur regroupe les activités informatiques nécessaires à la récupération des mesures de ce capteur dès que celles-ci sont disponibles et à leur mise à disposition des autres tâches de l'application.

La notion de tâche se rapporte donc à un niveau d'abstraction élevé, dans le cadre duquel on fait encore abstraction de la manière dont les logiciels seront exécutés par les processeurs.

V.2.2.NOTION DE PARALLÉLISME D'EXÉCUTION:

L'exécution de deux ou plusieurs TÂCHES est qualifiée de PARALLÈLE lorsqu'un observateur a l'IMPRESSION que celles-ci s'exécutent SIMULTANÉMENT, sachant que cette simultanéité peut être RÉELLE ou simplement APPARENTE. Un système d'exploitation qui permet d'exécuter des tâches en simultanéité réelle ou apparente est dit MULTITÂCHES.

V.2.3.NOTION DE THREAD:

V.2.3.1.DÉFINITION:

En anglais, le terme THREAD évoque, dans son acception la plus fréquente l'idée de fil (fil, filin, filetage, etc.). En informatique, le terme thread est souvent traduit en français par FIL D'EXÉCUTION par analogie avec le terme "fil de discussion" qui rassemble les échanges successifs entre interlocuteurs traitant d'un même sujet. On peut donc définir un thread comme une entité à caractère DYNAMIQUE correspondant à l'exécution d'un FLUX DE CONTRÔLE attaché à une TÂCHE donnée.

V.2.3.2.ÉTATS D'UN THREAD:

L'exécution d'un thread peut être interrompue par un autre thread exécutant une tâche "plus prioritaire" ou parce qu'une ressources nécessaire à l'exécution de ce thread est temporairement inaccessible. De ce fait, à un instant donné, l'état d'un thread est égal à une des valeurs suivante:

- ***PRÊT***: le thread est prêt à être lancé (son code exécutable vient d'être chargé en mémoire vive);
- ***ACTIF***: un exécuteur fixe est en train d'exécuter ce thread;
- ***EN ATTENTE***: le thread est en cours d'exécution mais il est en attente:
  - De la libération d'une RESSOURCE (équipement, zone mémoire partagée, etc.);
  - De la disponibilité d'un exécuteur fixe (si tous les exécuteurs sont occupés à d'autres tâches;
  - De la réception d'un MESSAGE en MODE BLOQUANT (le thread sera réactivé par la réception du message;
  - De la réception d'un SIGNAL (signal horaire, par exemple);
  - Etc..
- ***TERMINÉ***: l'exécution du thread s'est terminée naturellement ou bien elle a été arrêtée définitivement par un signal extérieur (ex: signal "kill" sous unix/linux);

V.2.3.3.RELATIONS ENTRE THREADS, TÂCHES ET APPLICATIONS:

- L'utilisation d'une APPLICATION nécessite l'exécution d'une ou plusieurs TÂCHES;
- L'exécution d'une TÂCHE nécessite l'exécution d'un ou plusieurs THREADS.

Du point de vue de l'ORDONNANÇEUR d'un système d'exploitation multitâche, les threads représentent donc les UNITÉS D'EXÉCUTION qu'il a à gérer.

V.2.4.NOTION DE PROCESSUS LOGICIEL:

V.2.4.1.DÉFINITION:

En informatique, dans un environnement multitâches, le concept de PROCESSUS LOGICIEL désigne une OCCURRENCE D'EXÉCUTION d'un logiciel donné, s'exécutant d'une manière indépendante par rapport aux autres processus en cours. Ceci implique qu'il n'existe aucune dépendance procédurale entre les processus : par défaut, ils s'exécutent de manière complètement ASYNCHRONE les uns des autres.

Un processus logiciel est donc une entité à caractère DYNAMIQUE et TEMPORAIRE qui est créée lors du lancement de l'exécution d'un logiciel. Il peut être supprimée par lui-même (terminaison normale) ou par la survenue d'un signal déclenché par un autre processus ou par la détection d'une anomalie empêchant son exécution.

REMARQUE: Un processus ne peut être assimilé à un programme car un même programme lancé plusieurs fois crée à chaque fois un ou des processus différents. Il peut donc exister à un moment donné plusieurs processus exécutant le même programme (par exemple, un traitement de texte peut être lancé plusieurs fois, ouvrant à chaque fois une fenêtre d'exploitation différente: les deux processus ainsi crées sont indépendants et peuvent traiter deux textes différents).

V.2.4.2.CARACTÉRISATION:

Un processus est caractérisé par:

- Un ESPACE D'ADRESSAGE en mémoire vive qui lui est propre, protégé de toute intrusion des autres processus "contemporains". Cet espace d'adressage contient la PILE D'EXÉCUTION et les données de travail nécessaires à l'exécution des traitements attachés au processus;
- Un ou plusieurs THREADS qui exécutent les traitements attachés à ce processus. Ces threads ont accès sans restriction à l'espace d'adressage du processus. Pour que le processus soit actif, un au moins de ces threads doit être en cours d'exécution;
- Un certain nombre d'autres ressources peuvent être affectées à un processus. C'est le cas en particulier des PORTS D'ENTREE-SORTIE (caractéristiques des échanges réseaux) ou des CONNEXIONS aux fichiers.

REMARQUES:

- Dans un système multitâches, le lancement d'un fichier exécutable (par exemple, un fichier .exe sous Windows) se traduit par la création d'un processus dont le premier thread exécute ce fichier;
- Les threads d'un même processus peuvent communiquer facilement en partageant les données contenues dans l'espace d'adressage commun ainsi que les ressources rattachées au processus. En revanche, les threads appartenant à des processus différents ne peuvent accéder aux mêmes données. L'encapsulation des traitements dans différents processus permet donc de limiter les "effets de bord" dans les applications multitâches.

V.2.4.3.PROCESSUS APPARENTÉS:

Un processus donné peut créer un processus FILS grâce au mécanisme généralement appelé FORK (fourche). Le fils partage l'état de l'espace d'exécution du père antérieur à sa création. La terminaison du père entraîne la terminaison du fils.

V.2.5.NOTION DE PRIORITÉ LOGICIELLE:

V.2.5.1.DÉFINITION :

La notion de PRIORITÉ LOGICIELLE permet de spécifier la priorité que le MONITEUR DE TÂCHES d'un système d'exploitation alloue à une tâche donnée pour l'accès à la ressource CPU. Le principe général est qu'une tâche de priorité élevée disposera pour son exécution d'un "temps CPU" (c'est à dire d'un temps d'exécution) plus important qu'une tâche de priorité moindre.

L'utilité de cette notion réside dans le fait qu'elle permet, dans un environnement multitâches, de favoriser l'exécution de certaines d'entre elles au détriment des autres, de

façon à rendre plus rapide l'exécution de tâches soumises à des contraintes d'ordre temporel strictes.

V.2.5.2.TRAITEMENTS DES PRIORITÉS :

Supposons qu'une application traite des demandes de différentes natures émises par divers utilisateurs. Ces demandes surviennent d'une façon aléatoire. Le traitement de chacune de ces demandes constitue une TÂCHE dont l'exécution peut, suivant sa nature, exiger plus ou moins de puissance CPU (pour simplifier, nous supposerons que l'infrastructure ne dispose que d'un seul CPU simple cœur.

Au fur et à mesure de l'arrivée des demandes des utilisateurs, les tâches correspondantes sont introduites dans une file d'attente. Nous pouvons distinguer trois modes principaux de gestion des tâches par le moniteur :

- SANS NOTION DE PRIORITÉ: En l'absence de priorité logicielle déclarée, ces différentes tâches pourront être exécutées :
  - Soit suivant un ordonnancement FIFO (exécution en file, dans l'ordre de leur arrivée). L'impression générale est celle d'une file d'attente ;
  - Soit en temps partagé (a chaque tâche est alloué un quantum de "temps CPU" (par exemple 100 ms). Ces quantums sont allouées à tour de rôle aux différentes tâches en cours d'exécution (l'impression générale est alors d'une exécution "en parallèle").
- ROUND ROBBIN : si des priorités différentes sont accordées à chaque tâche, le moniteur pourra choisir de lancer l'exécution des tâche présentes dans la file d'attente non pas suivant l'ordre d'arrivée, mais suivant leur niveau de priorité ;
- PRIORITÉS PRÉEMPTIVES : Dans ce cas, lorsqu'une tâche est introduite dans la file d'attente, le moniteur pourra (en plus du fonctionnement en round robbin), INTERROMPRE la tâche en cours d'exécution si cette dernière est de priorité inférieure, pour lancer la nouvelle tâche. (La tâche interrompue sera reprise lorsque aucune tâche de priorité supérieure ne sera présente dans la file d'attente. Ce type de fonctionnement revient à PRÉEMPTER IMMÉDIATEMENT LE CPU pour exécuter en priorité le code le plus prioritaire.

V.2.5.3.MANIPULATION DES PRIORITÉS LOGICIELLES:

Concrètement, la notion de priorité logicielle est un attribut qui s'applique aux THREADS, car ce sont eux qui sont les "unités d'exécution" manipulées par les moniteurs de tâches.

Dans le cadre de la plupart des systèmes d'exploitation multitâches, lors de la création d'un PROCESSUS (lancement d'un fichier exécutable), l'exécution du PROGRAMME correspondant à ce fichier constitue le premier THREAD du nouveau processus (quelquefois appelé "thread initial"). Une priorité PAR DÉFAUT lui est attribuée par le

moniteur de tâches. Le développeur peut alors utiliser des primitives système pour modifier la priorité de ce thread.

Cependant, lors de la création d'un processus, une priorité PAR DÉFAUTleur est souvent attribuée. Dans ce cas, les threads créés à l'intérieur de ce processus "héritent" de cette priorité jusqu'à ce que celle-ci soit modifiée (par programmation ou par commande système). Un processus "fils" hérite de la priorité de son père.

La priorité d'un processus ou d'un thread se traduit en général par un nombre entier relatif :

- De -20 à 19 pour un linux classique (-19 étant la priorité la plus haute, 0 étant la priorité par défaut) ;
- De 0 à 99 pour un linux muni de la capacité "temps réel souple" (0 état la priorité par défaut) ;
- Jusqu'à 256 pour des OS temps réel comme RT Linux, Lynx OS, Irix, Etc.

V.2.6.DIFFÉRENTS MODES DE TRAITEMENT DES TÂCHES DANS UN O.S MULTITÂCHES:

V.2.6.1.EXÉCUTION EN PARALLÉLISME RÉEL:

Pour que plusieurs threads puissent réellement s'exécuter en parallèle, il faut pouvoir disposer d'autant d'EXÉCUTEURS que de threads. Pour cela, il faut:

- Soit que le PROCESSEUR de la machine soit équipé d'au moins autant de CŒURS que de threads à exécuter (machines MULTICŒURS)
- Soit que cette machine soit équipées de plusieurs PROCESSEURS indépendants (machine MULTIPROCESSEURS).

V.2.6.2.EXÉCUTION EN PARALLÉLISME APPARENT:

A un instant donné, plusieurs threads peuvent être actifs dans un ordinateur (par exemple, une fenêtre peut afficher l'heure courante tandis qu'un utilisateur consulte un site web). Un EXÉCUTEUR FIXE ne peut exécuter qu'un seul thread à la fois. Cependant, il est possible de partager le temps d'exécution de cet exécuteur entre plusieurs threads. Ce partage est effectué par le SYSTÈME D'EXPLOITATION. Deux techniques de partage peuvent coexister:

V.2.6.2.1.LE TEMPS PARTAGÉ:

La technique, appelée TEMPS PARTAGÉ (TIME SHEARING en anglais),) consiste en un partage du temps d'exécution de l'exécuteur entre les threads: une certaine durée DT est allouée par roulement à chaque thread actif, comme l'illustre le schéma ci-dessous. Comme DT est choisie faible par rapport à la perception humaine (par exemple, DT=100 ms), l'utilisateur a l'impression d'une exécution simultanée:

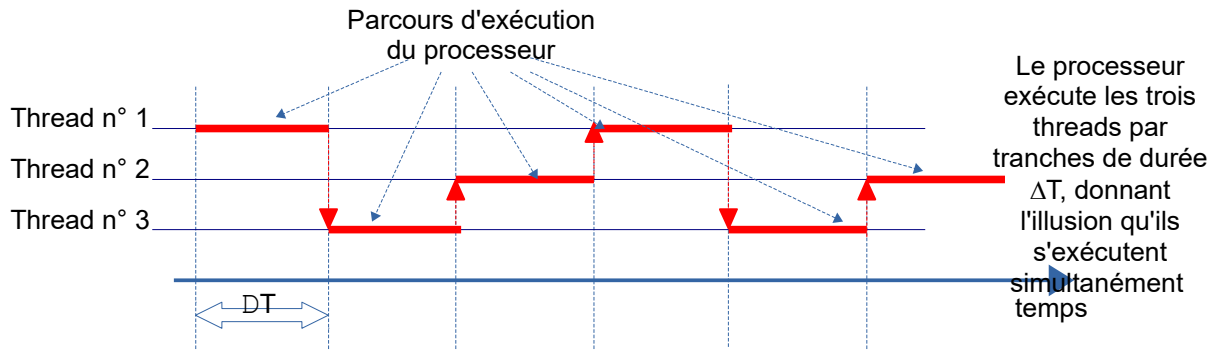


Fig. VIII.6.2.1-a: Exécution en temps partagé

V.2.6.2.2.LA GESTION PAR PRIORITÉS PRÉEMPTIVES:

Dans ce cas, un niveau de priorité est attribué à chaque THREAD (il s'agit d'un nombre entier positif ou nul). Lorsque plusieurs threads sont prêts à être exécutés par le même

exécuteur, c'est le plus prioritaire qui est lancé par le système d'exploitation. Ce thread occupe cet exécuteur jusqu'à la fin de son exécution, sauf dans deux cas:

- Si un autre thread plus prioritaire devient prêt à être lancé, le système d'exploitation interrompt immédiatement le thread en cours pour exécuter ce nouveau thread. On dit qu'il y a PRÉEMPTION de l'exécuteur par le thread le plus prioritaire. Dès que l'exécuteur peut être réaffecté au premier thread (c'est à dire quand celui-ci redevient le plus prioritaire), l'exécution reprend à l'endroit ou elle avait été interrompue.
- Si, pour pouvoir continuer son exécution, le processus en cours doit attendre un événement (un message de validation, par exemple), il passe à l'état EN ATTENTE. A ce moment là, le processeur sera réaffecté à l'exécution du thread prêt de priorité immédiatement inférieure. Lorsque l'événement attendu par le processus mis en attente se produit, le système d'exploitation interrompt immédiatement le thread en cours et reprend l'exécution du premier thread à l'endroit ou celle-ci avait été interrompue.

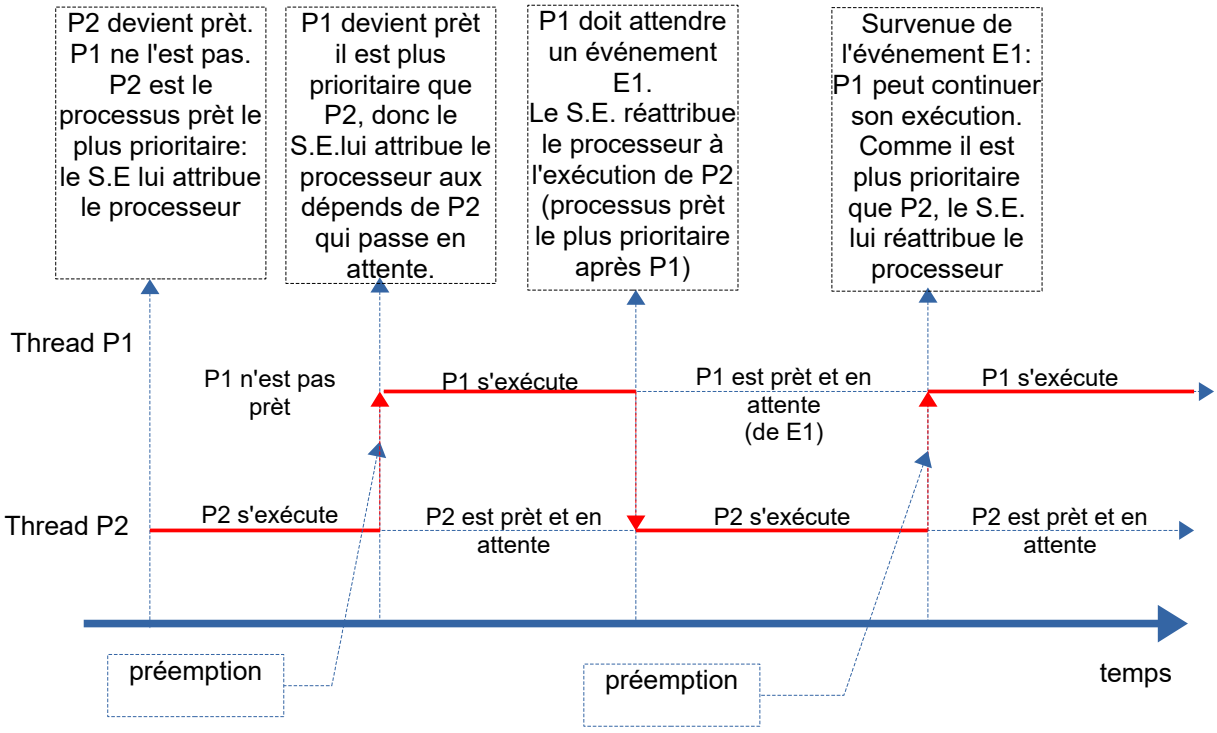


Fig. VIII.6.2.2.a : Execution par priorités préemptives

Le schéma ci-dessus donne une image de ce fonctionnement: P1 est P2 sont deux threads. La priorité de P1 est supérieure à celle de P2, mais P1 a besoin, à un certain moment de son exécution, d'attendre un événement E1 (on peut par exemple supposer qu'il a besoin d'une information qu'un utilisateur va entrer par le clavier). Les traces rouges horizontales matérialisent les périodes d'exécution de chaque thread par le processeur.

Les traces horizontales en pointillés matérialisent les périodes d'attente des processus (S.E. Signifie: Système d'Exploitation).

V.2.6.3.CHOIX DU MODE DE GESTION:

La plupart des systèmes supportent les deux modes de gestion: dans une même machine, certains processus peuvent être gérés par priorités préemptives alors que d'autres peuvent être gérés en temps partagé. Dans ce cas:

- Deux tâches préemptives, mais possédant des priorités identiques se voient allouer les ressources sur lesquelles elles sont concurrentes suivant la règle du temps partagé;
- Les tâches gérées en temps partagé sont considérées comme ayant toutes la même priorité, inférieure à celles de toutes les taches préemptives. De ce fait, une tâche gérée en temps partagé ne peut se voir attribuer une ressource (et en particulier, le processeur) que si aucune tâche préemptive n'est prête et non en attente.

Le mode de gestion en temps partagé est le mode «par défaut» de la plupart des systèmes d'exploitation. La gestion par priorités préemptive est surtout utilisée par les applications industrielles qui fonctionnement en «temps réel» ou par certaines fonctions systèmes qui doivent s'exécuter en priorité par rapport aux taches d'applications.

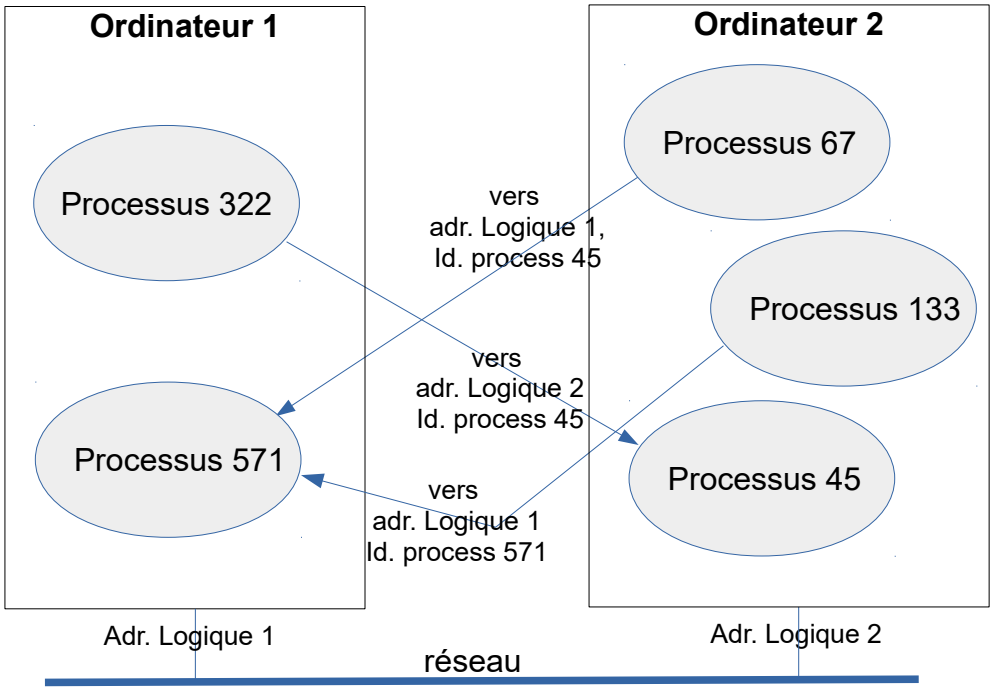
VI.ANNEXE III- COMMUNICATION EN RÉSEAU ENTRE PROCESSUS DISTANTS:

VI.1.NOTION DE CONNECTEUR RÉSEAU (OU SOCKET):

Une application répartie est supportée par plusieurs ordinateurs connectés à un réseau de communication. Chacun de ces ordinateurs supporte donc une partie de l'application qui a besoin de communiquer avec les autres parties pour échanger des données ou bien requérir ou fournir des services.

Comme nous l'avons noté en introduction, la communication entre deux parties de l'application répartie est un fait une communication entre PROCESSUS LOGICIELS.

EXEMPLE:



COMMENTAIRES:

- Le schéma ci-dessus représente l'exécution d'une application répartie sur deux ordinateurs connectés à un réseau de communication;
- La partie de l'application supportée par l'ordinateur 1 engendre, en s'exécutant dans cette machine, deux processus (auxquels nous attribuerons les identificateurs 322 et 571 dans cette machine);
- La partie de l'application supportée par l'ordinateur 2 engendre, en s'exécutant dans cette machine, trois processus(auxquels nous attribuerons les identificateurs 67, 133 et 45 dans cette machine).

- Les flèches représentent les communications entre processus: par exemple, le processus 322 de l'ordinateur 1 émet vers le processus 45 de l'ordinateur 2.

La communication entre processus logiciels est assurée par la couche 4 de l'ISO (couche transport). Cette couche correspond à la couche TCP de TCP/IP. Comme le schéma le suggère, lorsqu'un de ces processus, supporté par un ordinateur donné, doit émettre par le réseau vers un autre processus supporté par un autre ordinateue, il identifie le destinataire par un couple d'identificateurs:

- L'un représente l'adresse logique de l'ordinateur destinataire (c'est l'adresse IP pour un réseau TCP/IP);
- L'autre identifie le processus destinataire dans la machine distante (c'est le numéro de port pour le protocole TCP).

Ce couple de données est appelé CONNECTEUR RÉSEAU dans un cadre général ou SOCKET dans la terminologie TCP/IP. Deux processus distants communiquent entre eux par l'intermédiaire de deux CONNECTEURS RÉSEAU (ou SOCKETS).

VI.2.LES MODES DE TRANSMISSION ENTRE PROCESSUS:

Au niveau de la couche transport nous pouvons distinguer deux "modes" de transmission entre processus logiciels:

- Le MODE CONNECTÉ, caractérisé par le fait que toute transmission doit être précédée par l'établissement d'une CONNEXION LOGIQUE entre le processus émetteur et le récepteur. Cette connexion permet de s'assurer que le récepteur est présent sur le réseau et prêt à recevoir des informations. Elle permet également au récepteur d'identifier l'émetteur afin de pouvoir lui transmettre un compte-rendu. Ce mode correspond au protocole TCP de TCP/IP;
- Le MODE NON CONNECTÉ (ou DATAGRAM), caractérisé par l'absence de toute procédure de connexion entre émetteur et récepteur: l'émetteur émet sans être assuré que le récepteur existe ni qu'il est en état de recevoir les informations. Ce mode correspond au protocole UDP de TCP/IP (appelé quelquefois UDP/IP).

Ces deux modes d'échanges correspondent à des besoins de communication très différents:

VI.2.1.EMPLOI DU MODE CONNECTÉ:

La CONNEXION permet de s'assurer que le récepteur est en état de recevoir les informations et que l'on ne va pas émettre "dans le vide": les protocoles connectés permettent au récepteur de renvoyer un message d'acquittement (ACK) à l'émetteur pour confirmer la réception de chaque message ou segment de message. Ce mécanisme permet de répéter l'envoi d'un message non reçu ou incorrect. La connexion permet donc de sécuriser les échanges de données.

Additionnellement, les protocoles connectés (par exemple, TCP), intègrent des mécanismes permettant de s'affranchir des erreurs de transmission, même lorsque les données sont très volumineuses: les messages volumineux peuvent être découpés en segments envoyés individuellement, puis réassemblés par le récepteur. Chaque segment reçu est acquitté par le récepteur. En cas de non réception ou de message erroné, une réémission est effectuée par l'émetteur.

Ces mécanismes conviennent très bien lorsque la sécurisation des échanges est un priorité (pour les transactions bancaires, par exemple). En revanche, elles ralentissent les transmission et ne permettent pas d'assurer des délais de réception déterministes.

VI.2.2.EMPLOI DU MODE NON CONNECTÉ:

L'ABSENCE DE CONNEXION avec le destinataire présente évidemment le risque d'émettre vers un destinataire inexistant ou hors d'état de recevoir le message, sans que l'on puisse en être prévenu. d'autre part, il ne permet pas de détecter les erreurs de transmission dues à des perturbations ponctuelles.

De plus, le mode DATAGRAM, du fait de l'absence d'acquittement des réceptions, ne permet pas la segmentation des messages volumineux, ce qui limite la taille de ces messages aux possibilités du réseau en la matière.

Le mode non connecté présente cependant l'avantage de rendre beaucoup plus déterministes les délais de transmission (du moins quand le réseau est peu utilisé).

De plus, le mode non connecté permet d'émettre un même message vers un groupe de destinataires et non vers un seul: c'est la transmission en MULTICAST, qui permet d'éviter la multiplication des envois lorsqu'un même message doit être envoyé à plusieurs postes différents.

C'est pour ces raison que le mode DATAGRAM est très utilisé dans les applications ou les messages échangés sont peu volumineux et où la perte de quelques messages n'est pas critique par rapport à la garantie du déterminisme des dates de réception (la VoIP ou les applications à contraintes "temps réel", par exemple).

VI.2.3.REMARQUE SUR LA FRAGMENTATION DES MESSAGES:

RAPPELS:
La taille maximale d'un "paquet" transporté par le protocole IP de TCP/IP est de 65535 octets, y compris les octets de protocole. De ce fait:

- Cette taille est la taille maximale pouvant être transportée message UDP de UDP/IP
- Le protocole TCP fragmente les messages de taille supérieure à 65535 octets en plusieurs paquets.

Cependant, la couche transport n'est pas la seule couche de communication qui utilise la fragmentation des messages. En effet, chaque technologie de liaison est limitée dans la longueur des trames qu'elle peut transmettre d'un seul tenant (la technologie Ethernet, par exemple, limite cette longueur à 1518 octets, y comprise les octets de protocole). Au delà de cette longueur maximale, les trames sont découpées en plusieurs fragments. De ce fait, un message de 65535 octets subit au niveau de la couche liaison une fragmentation en 44 fragments.

VII.ANNEXE IV - RAPPELS SUR LA PROGRAMMATION PAR OBJETS:

VII.1.REMARQUE:

Cette présentation de la programmation objets est restreintes aux notions qu'il est indispensable de connaître pour comprendre la démarche de conception objets.

VII.2.LA NOTION D'OBJET:

RAPPELS: principes structurels des langages procéduraux:

- Les langages de programmation PROCÉDURAUX décrivent les applications sous la forme d'arborescences de ROUTINES. Le sommet de cette arborescence est constitué par une routine particulière (souvent appelée main) qui définit le point d'entrée du logiciel;
- Suivant le langage, les ROUTINES peuvent aussi être appelées fonctions, sous-programmes, subroutines, etc.);
- Une routine peut être définie comme une entités encapsulant une portion de code capable d'effectuer un traitement spécifique et bien identifié à partir de valeurs qui lui sont fournies (variables ou arguments d'entrée);
- Une ROUTINE possède une interface (et une seule) permettant de lui fournir des arguments et d'activer son code interne et de récupérer les résultats de son exécution.
- Dans un langage procédural, les données RÉMANENTES de l'application sont déclarées EN DEHORS des routines et constituent des entités différentes.

A la différences des routines (ou fonctions) de la programmation procédurale, un OBJET est une entité logicielle qui encapsule à la fois:

- **Des procédures de traitement appelées MÉTHODES**: ce sont des sous-programmes pouvant être activés de diverses façons par l'objet lui-même ou par les autres objets de l'application (si ces méthodes sont déclarées PUBLIQUES). Un objet peut donc être considéré comme une BIBLIOTHÈQUE DE MÉTHODES;
- **Des données associées à ces traitement, appelées ATTRIBUTS**. Les méthodes de l'objet, mais aussi les autres objets de l'application (si ces attributs sont déclarés PUBLICS) peuvent accéder aux données contenues dans ces ATTRIBUTS. Un attribut peut être lui-même un objet.

En programmation OBJET, une donnée est toujours déclarée à l'intérieur d'un objet (ou plutôt de la classe de cet objet, comme nous le verrons plus tard.

VII.3.STRUCTURE D'UNE CLASSE D'OBJET:

La Programmation Orientée Objets (P.O.O) cherche à produire des composants RÉUTILISABLES. Pour ce faire, elle ne crée par des composants UNIQUES mais des MODÈLES DE COMPOSANTS destinés à être déclinés dans plusieurs application. Ces modèles sont appelés des CLASSES d'objets.

Une CLASSE n'est pas exécutable en elle-même. Pour créer à partir de cette classe un OBJET exécutable, il faut l'INSTANCIER. L'instanciation crée un code objet exécutable dans la mémoire de l'ordinateur.

EXEMPLE: Soit la classe Éditeur:

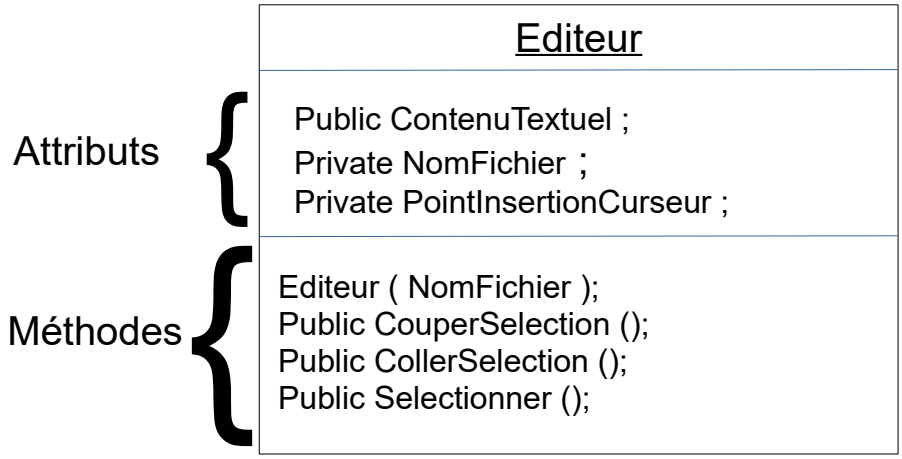


Fig. VII.3-a : Classe Editeur (exemple)

La méthode Editeur, qui a la particularité d'avoir le même nom que l'objet est le CONSTRUCTEUR de la classe. C'est son activation qui crée un objet à partir de cette classe. Le constructeur est donc appelé lors de l'instanciation d'un objet, avec des paramètres qui permettent de déterminer quel objet doit être crée. Ainsi, si l'on veut éditer le fichier texte c://MesDocuments/MonTexte.txt, on écrira une instruction de ce type (ex: en PHP):

```
$MonEditeur = new Editeur ( c:/MesDocuments/MonTexte.txt );
```

L'instruction ci-dessus crée l'objet $MonEditeur à partir de la classe Editeur. L'opérateur new est l'opérateur d'INSTANCIATION dans de nombreux langages.

Le mot clef PUBLIC placé devant la déclaration d'une MÉTHODE ou d'un ATTRIBUT indique que cette méthode ou cet attribut sont accessibles depuis d'autres objets que l'objet instancié par cette classe. Le mot clef PRIVATE interdit l'accès aux autres objets. Le CONSTRUCTEUR est évidemment toujours PUBLIC.

VII.4.CODAGE D'UNE CLASSE D'OBJET:

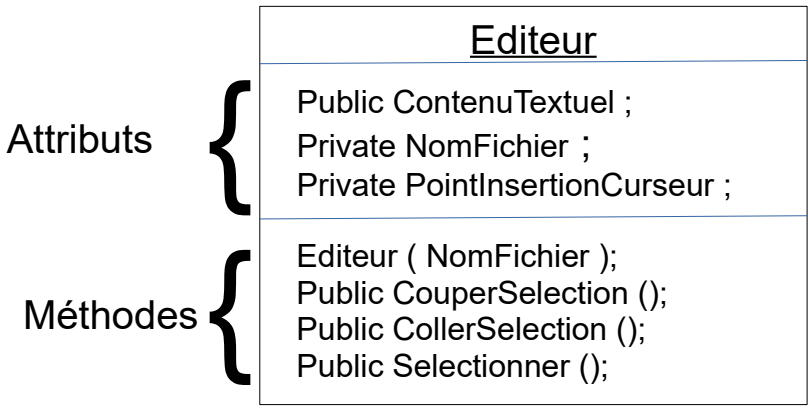


Fig. VII.4-a :Codage d'une classe d'objets

COMMENTAIRES:
Le mot clef **class** indique que ce qui suit est la déclaration de la classe (Ici, classe Editeur) ;

Dans le code objet qui utilisera un objet de cette classe, il faudra d'abord instancier l'objet :

$MonObjet = new Editeur ("./MonDocument.txt");

Une fois l'objet instancié, je pourrai appeler les méthodes grâce à l'opérateur "→" .

EXEMPLES :
Appel méthode CouperSelection :
$MonObjet->CouperSelection();

Accès à la valeur de ContenuTextuel :
$MonObjet->ContenuTextuel;

En PHP :

```
<?php
class Editeur
{
      public $ContenuTextuel ;
      private $NomFichier ;
      private $PointInsertion ;

      // constructeur
      function Editeur ( $nomfichier )
      {
      // corps de la methode
      }

      // methode couperselection
      function CouperSelection ()
      {
      // corps de la methode
      }

      // methode collerselection
      function CollerSelection ()
      {
      // corps de la methode
      }
      etc....
}?>
```

VII.5.CLASSES DÉRIVÉES, NOTION D'HÉRITAGE:

La programmation OBJET permet de créer une CLASSE D'OBJETS à partir d'une classe déjà existante, en rajoutant des méthodes ou des attributs ou en redéfinissant des méthodes ou des attributs existants.

La CLASSE ainsi dérivée HÉRITE de tous les attributs et toutes les méthodes de la classe d'origine (éventuellement modifiés), plus les attributs et méthodes rajoutés ou redéfinis.

Le procédé consistant à redéfinir une méthode ou un attribut s'appelle la SURCHARGE.

EXEMPLE:

Imaginons qu'on veuille dériver de la classe Editeur une autre classe (Editeur1) héritant de Editeur, mais possédant en plus la méthode InsererCaractere ($Caractere) Il suffira d'écrire en PHP:

En PHP :

```
<?php
class Editeur1 extends Editeur
{
     // methode InsererCaractere
     function InsererCaractere ( $Caractere)
     {
       // corps de la methode
     }
     etc....
}?>
```

Le mot-clef **Extends** permet à la classe Editeur1 d'HERITER de la classe Editeur.

Editeur1 récupère donc tous les attributs et toutes les methodes de Editeur. De plus, il est doté d'une nouvelle méthode InsererCaractere.

VII.1.REMARQUE:

Si l'on se place au niveau du codage, une méthode n'est rien d'autre qu'une routine un peu particulière (car elle fait référence à une classe). Ceci explique que beaucoup de langages objets (java, c++, etc.) nomment "fonctions" les méthodes d'une classe;

VIII.ANNEXE V – OUTILS DE MODÉLISATION COMPORTEMENTALE:

VIII.1.INTRODUCTION:

Cette annexe donne un aperçu de certains outils et des formalismes utilisés pour l'étude comportementale des logiciels. Un diagramme comportemental ne donne qu'une vue partielle du comportement d'un logiciel. De ce fait, pour être certain d'avoir identifié et caractérisé tous les aspects de ce comportement, le concepteur doit souvent recourir à plusieurs de ces outils abordent le problème suivant des points de vue différents.

VIII.2.LES DIAGRAMMES D'ÉTATS-TRANSITIONS:

VIII.2.1.LES AUTOMATES A ÉTATS FINIS :

VIII.2.1.1.DÉFINITION:

VIII.2.1.1.1.DÉFINITION RIGOUREUSE :

Un automate déterministe fini peut être défini par un "quintuplet" M = (K, E, Δ, ε , k) où :
- K est un ensemble fini (non vide) d'états
- E est un alphabet (ensemble non vide de lettres)
- Δ et une fonction appelée fonction de transition, définie comme suit :

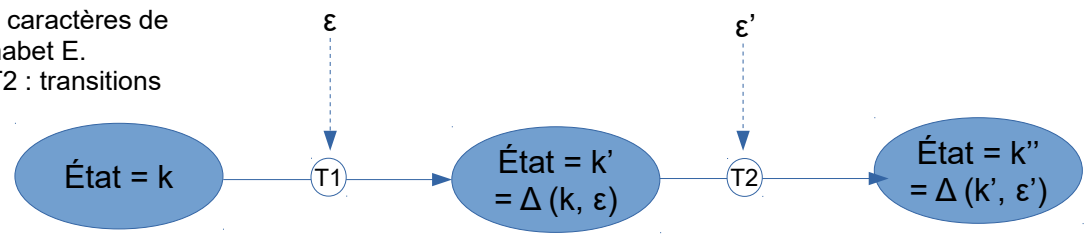
  K x E → K

  Δ (k, ε) → k'
- ε est un élément de E
- k est un élément de K
- k_0 est l'état initial.

VIII.2.1.1.2.ASPECT OPÉRATIONNEL :

Cette définition mathématique sous-tend le fonctionnement opérationnel suivant : si l'état courant est k, la lecture d'un élément ε dans l'alphabet provoque la TRANSITION de l'état k à l'état k' = Δ (k, ε). Le schéma ci-dessous représente ce fonctionnement :

- k, k', k" : états
- ε, ε' : caractères de l'alphabet E.
- T1, T2 : transitions



COMMENTAIRES: la "réception" du caractère ε déclenche la transition T1 qui fait passer le système de l'état k à l'état k' par application de la fonction de transition Δ avec les arguments k et ε ;

- De même, la "réception" du caractère ε' déclenche la transition T2 qui fait passer le système de l'état k' à l'état k'' par application de la fonction de transition Δ avec les arguments k' et ε' ;
- Etc.

VIII.2.2.APPLICATION A L'ÉTUDE COMPORTEMENTALE DES LOGICIELS :

VIII.2.2.1.ASSIMILATION D'UN LOGICIEL À UN AUTOMATE D'ÉTATS FINIS :

L'assimilation d'un logiciel à un automate d'états finis repose sur les considérations suivantes:

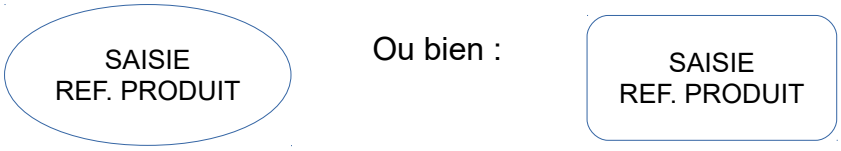
- L'ÉTAT COURANT d'un logiciel peut être défini par une conjonction momentanée et durable des valeurs de certains des paramètres de ce logiciel (par exemple, certains des attributs d'un objet), conduisant au maintien dans la durée d'un certain type d'activité jusqu'à ce qu'un événement le fasse passer dans un autre état (ainsi, un logiciel peut se trouver à l'état "en réception" et peut quitter cet état sur réception de l'événement "fin de transmission"). Un état ne correspond donc pas forcément à une période d'inactivité du logiciel ;
- Lorsqu'un événement quelconque amène le changement d'état du logiciel, ce changement est produit par le déclenchement d'un traitement spécifique (gestionnaire d'événement) qui, en provoquant la modification des paramêtres d'état du logiciel, entraîne la modification des activités de ce logiciel, c'est à dire la TRANSITION vers un autre état ;
- L'ensemble K des états du logiciel correspond donc à un certain nombre de combinaisons des valeurs des paramêtres d'état qui conduisent à des états de fonctionnement "stables" (c'est à dire qui se maintiennent dans la durée).
- L'ensemble E est formé par les événements susceptibles d'influer sur l'état du logiciel (commandes, signaux, exceptions, etc.) ;
- L'application Δ (k, ε) peut être définie par un tableau à deux entrées (état courant, événement reçu) dont les cases contiennent l'état futur résultant de l'application du traitement de l'événement reçu sur l'état courant du logiciel.

VIII.2.2.2.MODÉLISATION GRAPHIQUE :

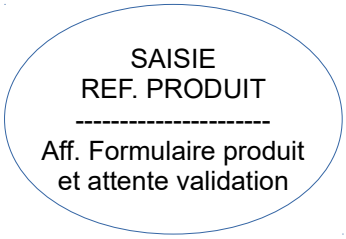
Les diagrammes d'états-transitions sont utilisés dans le cadre de très nombreuses méthodes ou langages de conception, avec un formalisme plus ou moins rigoureux et complexe. Cependant, nous pouvons dégager un certain nombre de règles communes :

VIII.2.2.2.1.REPRÉSENTATION DES ÉTATS :

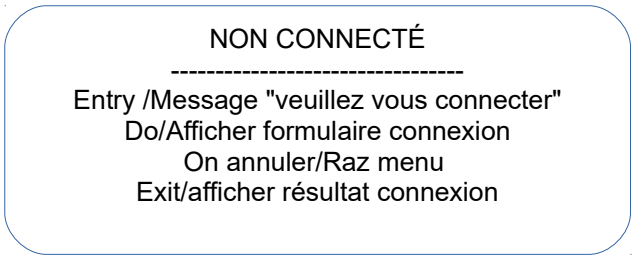
Les ÉTATS sont représentés par des surfaces pleines aux contours elliptiques ou en forme de rectangles "arrondis". A l'intérieur de ces formes figure au minimum l'identification de l'état :



Des indications sur les traitements supportés peuvent être ajoutées à l'identificateur de l'état :

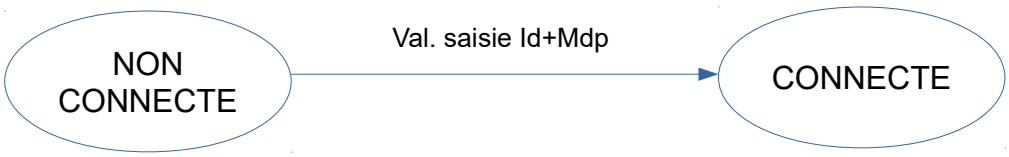


REMARQUE : Dans U.M.L, ces indications permettent de préciser les actions effectuées à l'entrée dans l'état (entry), à la sortie de l'état (exit), pendant que l'état est maintenu (do) ou quand un événement survient pendant que l'état est en cours (on). La syntaxe est la suivante :



VIII.2.2.2.2.REPRÉSENTATION DES TRANSITIONS :

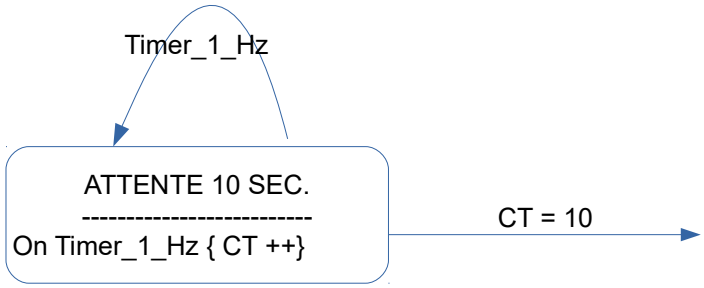
Les TRANSITIONS d'un état à un autre sont représentées par des arcs orientés (flèches droites ou courbes). Une légende identifiant l'événement déclencheur est associée à la transition (signal, commande, validation de saisie, etc.) :



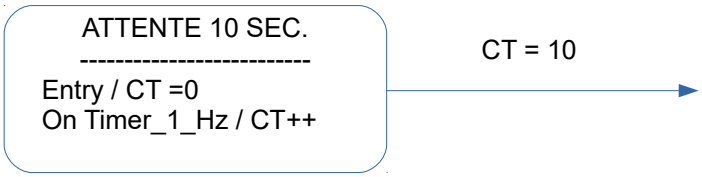
REMARQUE : une transition sans identificateur est souvent considérée comme se produisant automatiquement.

VIII.2.2.2.3.TRANSITIONS RÉFLEXIVES :

Une transition est dite réflexive lorsqu'elle aboutit au même état que son état de départ. Les transitions réflexives permettent de représenter des traitements itératifs pendant le maintien d'un état. Exemple :

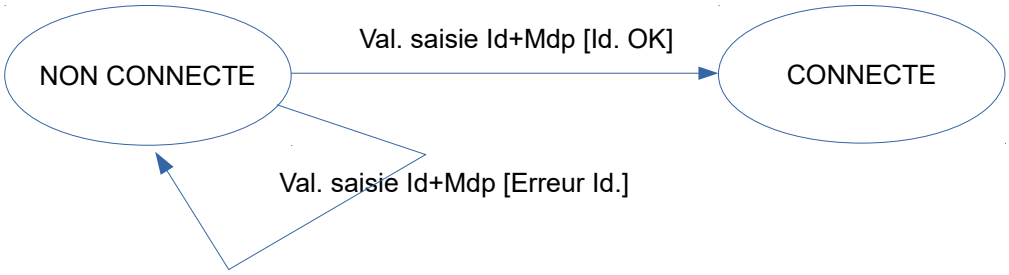


REMARQUE : notation U.M.L équivalente :

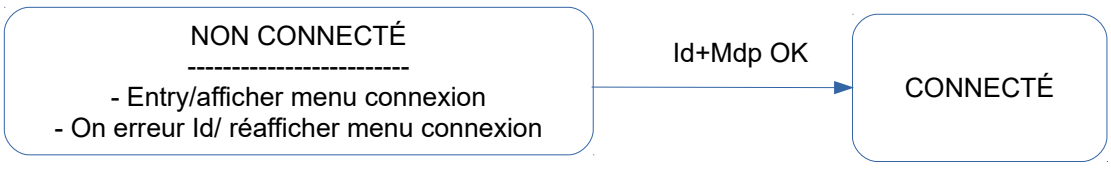


VIII.2.2.2.4.TRANSITIONS CONDITIONNELLES:

Il est possible de spécifier que les transitions partant d'un même état peuvent être déclenchées alternativement en fonction d'une condition (condition de GARDE) :



Équivalent U.M.L :



VIII.2.2.3.EXEMPLE:

VIII.2.2.3.1.AVERTISSEMENT :

Le graphique d'états qui figure dans cet exemple utilise une notation que nous qualifierons de NAÏVE, car elle ne respecte entièrement aucune norme connue. Elle a l'avantage de n'utiliser que des représentations graphiques simples, qui correspondent à celles qui ont été abordées au sous-chapitre précédent. Les notations textuelles qui accompagnent le diagramme permettent de compenser le manque de précision de ces notations.

VIII.2.2.3.2.PRÉSENTATION DU CAS :

L'exemple consiste à étudier le comportement du logiciel embarqué dans un lecteur de DVD. Le boîtier de ce lecteur est muni des commandes suivantes :

- On/Off (commutateur sous-tension/hors tension) ;
- Lecture (commutateur marche/arrêt lecture ;
- Pause (commutateur Mise en pause/Fin de pause) ;
- + (impulsion passage à la plage de lecture suivante) ;
- - (impulsion retour à la plage de lecture précédente) .

Si nous nous plaçons d'un point de vue fonctionnel, nous pouvons définir les différents ÉTATS que ce logiciel peut prendre par la liste suivante :

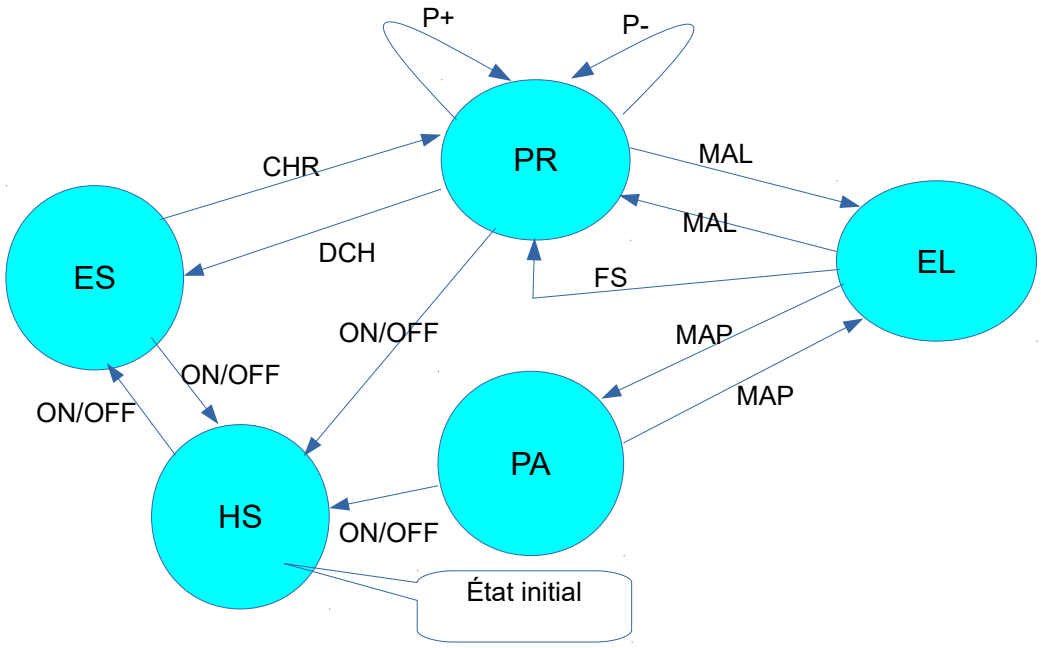
1. HS: En service (lecteur physique hors tension et déchargé);
2. ES: En service (lecteur physique sous tension mais déchargé);
3. PR: Prêt à lire (un DVD est en place, l'index de lecture est défini au début de la première plage) ;
4. EL: lecture en cours (le lecteur lit séquentiellement les enregistrements du DVD) ;
5. PA: lecteur en pause (la lecture est arrêtée, index et plage de lecture restent figés) ;

D'autre part, ce logiciel sera soumis aux commandes et signaux suivants :

- ON/OFF : mise sous-tension/hors tension ;
- MAL:Marche/Arrêt lecture;
- MAP : Mise en pause/Fin de pause ;
- CHR : Signal de chargement d'un DVD ;
- DCH : signal de déchargement d'un DVD ;
- P+ : passage à la plage de lecture suivante ;
- P- : retour à la plage de lecture précédente ;
- FS : signal fin de support de lecture détectée;

Le diagramme d'états/transitions de ce logiciel pourrait ressembler à ce qui suit :

VIII.2.2.3.3.DIAGRAMME D'ÉTATS-TRANSITIONS :



COMMENTAIRE :

- Ce diagramme permet de déterminer la modification de l'état du système consécutive à chacun action de l'utilisateur : il décrit donc le comportement du système étudié.
- Dans cette forme de représentation ne figurent que les transitions pour lesquelles la commande appliquée à l'état courant du système produit une modification du fonctionnement du lecteur. Les commandes P+ et P- ne produisent aucun changement d'état, mais en revanche, elles permettent de passer d'une plage de lecture à une autre.
- Dire qu'un logiciel se trouve dans un certain état ne signifie pas forcément que ce logiciel est inactif: dans l'exemple ci-dessus, lorsque le logiciel se trouve à l'état EL (En Lecture), il est tout sauf inactif, puisqu'il lit en permanence les données du D.V.D et les transforme en signaux audio. L'état EL correspond à une période d'activité répétitive que l'on peut représenter ici par le pseudo-code :
  - TANT QUE (je n'ai pas atteint la fin de l'enregistrement) FAIRE
  - Je lis le D.V.D
  - FIN FAIRE .

VIII.2.2.3.4.REPRÉSENTATION SOUS FORME DE TABLEAU :

Le diagramme ci-dessus peut utilement être complété par un tableau à deux entrées (transitions, états).

Les cases à fond blanc de ce tableau définissent une correspondance entre les doublés (transition, état courant) et l'ensemble des états futurs.
Les cases non renseignées correspondent à des transitions inopérante dans le contexte de l'état courant (exemple : la transition MAL est inopérante sur l'état ES).

États courants \ **Transitions**	**HS** (hors service)	**ES** (en service)	**PR** (prêt à lire)	**EL** (en cours de lecture)	**PA** (en pause)
ON/OFF	ES	HS	HS	interdit	HS
MAL (marche/arrêt lecture)	non sens	non sens	EL lancement lecture au début de la plage en cours.	PR arrêt lecture et retour au début de plage .	non sens
MAP (marche/arrêt pause)	non sens	non sens	non sens	PA arrêt lecture avec conservation de la plage et de l'index de lecture courants.	EL reprise lecture dans plage et à l'index courants.
CHR (chargement DVD)	non sens	PR	non sens	interdit	interdit
DCH (déchargement DVD)	non sens	non sens	ES	interdit	interdit
P+ (plage suivante)	non sens	non sens	PR avec plage courante = plage suivante	non sens	non sens
P- (plage précédente)	non sens	non sens	PR avec plage courantes = plage précédente	non sens	non sens
FS (fin de support atteinte)	non sens	non sens	non sens	PR avec l'index lecture fixé au début de première plage.	non sens

REMARQUES :

- Les cases marquées "non sens" signifient que la transition correspondante n'aurait pas de sens dans le contexte de l'état courant (exemple : la commande marche/arrêt lecture n'a pas de sens dans le contexte des état HS ou ES) ;
- Les cases marquées "interdit" signifie que la transition correspondante n'est pas impossible techniquement mais problématique dans le contexte de l'état courant (exemple : il est quelquefois possible de décharger un DVD en cours de lecture, mais cela risque d'endommager le lecteur ; Il faut donc soit interdire la commande, soit développer une fonction qui gère la transition : arrêt de la lecture avant de prendre en compte le déchargement).

VIII.3.LES DIAGRAMMES D'ACTIVITÉS :

VIII.3.1.INTRODUCTION:

Bien qu'appartenant à l'environnement d'U.M.L, les diagrammes d'activité peuvent parfaitement être utilisés dans d'autres cadres. En effet, le formalisme qui leur est attaché étant assez intuitif, ils demandent relativement peu d'apprentissage.

Dans le cadre du langage de conception U.M.L, les DIAGRAMMES D'ACTIVITÉS sont utilisés pour décrire le COMPORTEMENT d'un logiciel dans un contexte d'exécution déterminé. Ils permettent de faire apparaître les TRAITEMENTS liés aux comportements qu'ils décrivent, le cheminement du flot d'exécution ainsi que les relations entre les activités en les OBJETS qu'elles utilisent (le mot objet devant être compris au sens large).

Les diagrammes d'activité d'U.M.L peuvent être utilisés au cours de deux phases d'un projets logiciel:

- En phase de spécification du besoin, ils permettent de préciser le comportement que l'on peut associer à un CAS D'UTILISATION;
- En phase de conception, il permettent de représenter graphiquement le comportement d'une MÉTHODE DE CLASSE (U.M.L étant orienté objets).

VIII.3.2.NOTIONS D'ACTIVITÉ ET D'ACTIONS :

VIII.3.2.1.LA NOTION D'ACTION:

Une action est un bloc de traitement que l'on veut considérer comme ATOMIQUE. Ceci veut dire qu'au niveau d'analyse où on se trouve, on peut considérer une action comme une OPÉRATIONS ÉLÉMENTAIRE dont on peut faire abstraction des détails de réalisation.

Le développeur aura intérêt à choisir ces blocs de façon à ce qu'ils correspondent à des traitements bien identifiables, dont il est facile d'évaluer la complexité et la durée d'exécution. Par exemple:

- L'émission ou la réception d'un signal ou de données;
- L'attente d'un signal ou du déblocage d'une ressource;
- L'instanciation d'une classe;
- L'appel d'une fonction ou d'une méthode;
- L'exécution d'un algorithme mathématique;
- etc.

VIII.3.2.2.LA NOTION D'ACTIVITÉ:

VIII.3.2.2.1.DÉFINITION :

Une ACTIVITÉ représente le COMPORTEMENT d'un logiciel en cours d'exécution dans un contexte donné. Elle peut être NOMMÉE par un terme rappelant son utilité FONCTIONNELLE et CARACTÉRISÉE par la description algorithmique des traitements qui lui sont associés.

Une ACTIVITÉ peut être décomposée en un enchaînement de plusieurs ACTIONS suivant une logique algorithmique.

Cet enchaînement n'est pas forcément considéré comme séquentiel : un diagramme d'activité peut faire apparaître des bifurcations du FLOT D'EXÉCUTION (qui correspondent à des opportunités d'exécution en parallèle) ou des fusions de plusieurs flots en un seul. En revanche, il ne doit exister qu'un seul flot d'exécution "entrant" et un seul flot d'exécution "sortant".

VIII.3.2.2.2.ÉVÉNEMENT INITIATEUR D'UNE ACTIVITÉ:

L'exécution d'une activité est toujours conditionnée par la survenue d'un événement:

- Le lancement de l'application;
- La réception d'une requête d'utilisateur (Ex ; un "double clic" sur un lien hypertexte);
- Un signal interne ou externe (ex: arrivée à échéance d'un "timer", interruption logicielle ou matérielle, etc.) ;
- La réception de données, d'un compte-rendu, d'une exception, etc.;
- Ou toute autre cause susceptible de provoquer l'exécution d'un élément de logiciel.

VIII.3.2.2.3.TERMINAISON D'UNE ACTIVITÉ:

Une activité est terminée lorsque tous les flots d'exécution "parcourant" cette activité ont été arrêtés.

VIII.3.2.2.4.FORMALISME GRAPHIQUE :

A.INTRODUCTION:

Les diagrammes d'activité d'U.M.L sont des GRAPHES : ils sont donc composés de nœuds reliées entre eux par des arcs.

LES TROIS TYPES DE NŒUDS :

Il existe trois types de nœuds : les nœuds EXÉCUTABLES, les nœuds de CONTRÔLE et les nœuds OBJETS :

- Les nœuds exécutables encapsulent:
  - Soit des ACTIONS (NŒUDS D'ACTION) ;
  - Soit une partie d'un diagramme d'activité qui peut être considérée comme une sous-activité de l'activité englobante. Ce type de nœud est appelé NŒUD D'ACTIVITÉ STRUCTURÉ ;
- Les nœuds de contrôle sont des nœud d'activité abstrait (vides de traitements) dont le rôle est de spécifier l'aiguillage des flots de contrôle entre les différents nœuds d'une activité. Ils décrivent la logique algorithmique de l'activité. Un nœud de contrôle peut représenter :
  - Le début ou la fin d'une activité ;
  - Une bifurcation du flot de contrôle ou la fusion de deux flots ;
  - L'union de deux flots de contrôle (avec synchronisation) ;
  - L'exécution alternative de deux flots de contrôle en fonction de conditions ;
- Les nœuds OBJETS modélisent les OBJETS échangés entre les activités (dans le sens de "informations structurées"),

LES DEUX TYPES D'ARCS :

Les arcs représentent soit les FLUX DE CONTRÔLE, soit les FLUX D'OBJETS (ou flux de données) échangés entre les différents nœuds :

- Les flux de contrôle représentent les différents cheminements possibles de la trace d'exécution des processus entre les nœuds exécutables en fonction du contexte d'exécution. En cela, les diagrammes d'activité UML sont très proches dans leur principe des anciens ORGANIGRAMMES DE PROGRAMMATION ;
- Les flux d'objets modélisent les échanges d'informations entre les activités. La notion est très proche de celle de FLUX DE DONNÉES ;

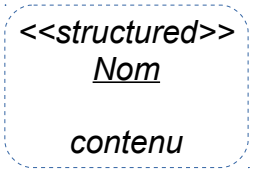
REMARQUES :

- Un nœud d'activé structuré ne peut avoir qu'un seul flot de contrôle entrant et un seul flot de contrôle sortant ;
- Un nœud d'activité structuré peut encapsuler d'autres nœuds d'activité structurés. La notion est donc récursive.

REPRÉSENTATION GRAPHIQUE :

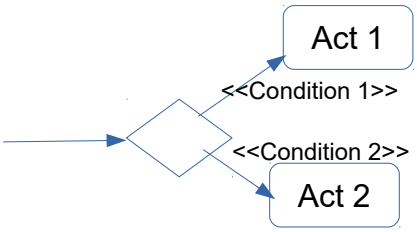
Nœud d'ACTION :

Nœud d'ACTIVITÉ STRUCTURÉ :

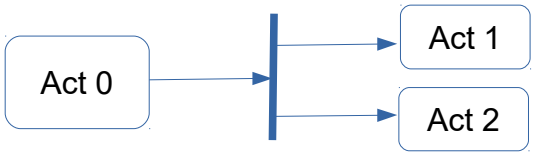


Nœud de CONTRÔLE :

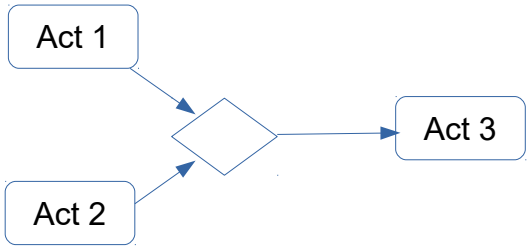
Exécution alternative :



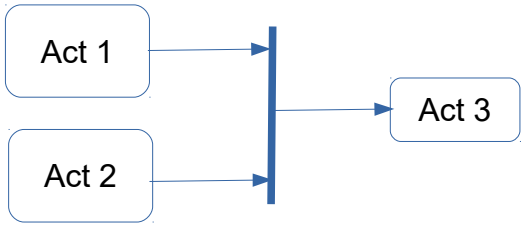
Bifurcation :



Fusion :



Union (avec synchronisation) :

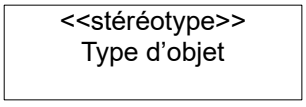


Début d'activité :

Remarque : une activité peut comprendre plusieurs nœuds "début"

Fin d'activité :

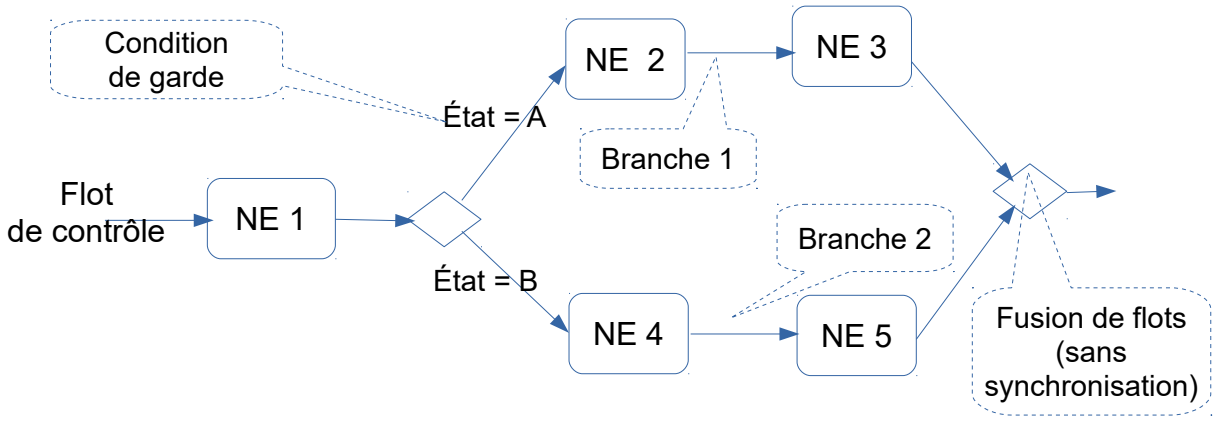
Nœud objet :



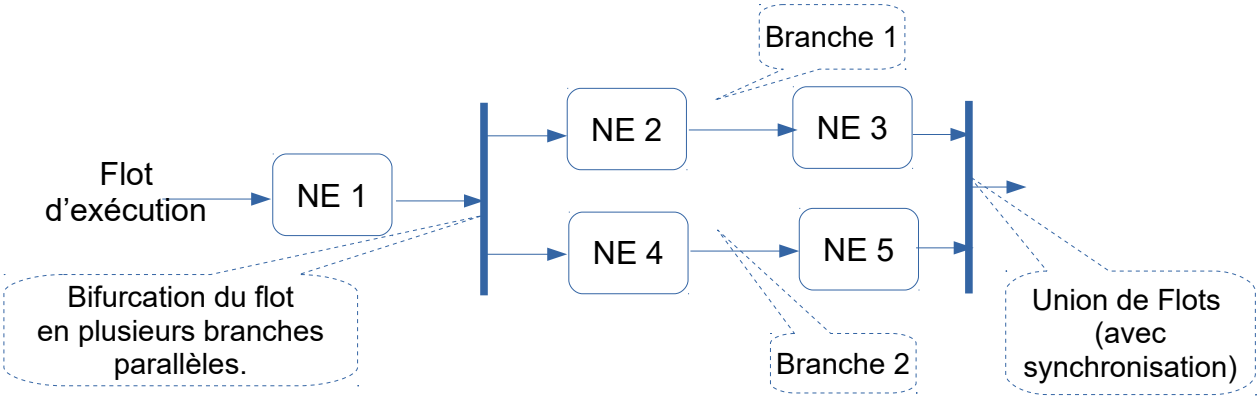
B.EXÉCUTION DE PLUSIEURS ACTIONS OU ACTIVITÉS STRUCTURÉES EN SÉQUENCE:

NOTA: Les flèches représentent le cheminement du FLOT DE CONTRÔLE entre les blocs

C.EXÉCUTION ALTERNATIVE EN FONCTION DE L'ÉTAT DU CONTEXTE D'EXÉCUTION:



D.EXÉCUTION EN PARALLÈLE DE PLUSIEURS BRANCHES DE TRAITEMENT:



REMARQUES:

- Pendant l'exécution d'une activité, le "flot d'exécution" reste dans cette activité jusqu'à ce que celle-ci soit terminée.

- Un seul flot entre et sort d'une activité, mais ce flot peut se dédoubler à l'intérieur de l'activité (cas de l'exécution "en parallèle" de plusieurs branches).
- A ce niveau d'analyse, le fait que le flot d'exécution bifurque n'implique pas forcément que les branches créées par cette bifurcation doivent être traitées en parallèle: cela signifie uniquement que l'exécution en parallèle est POSSIBLE.

E.DÉBUT ET FIN D'ACTIVITÉ OU DE FLOT D'EXÉCUTION:

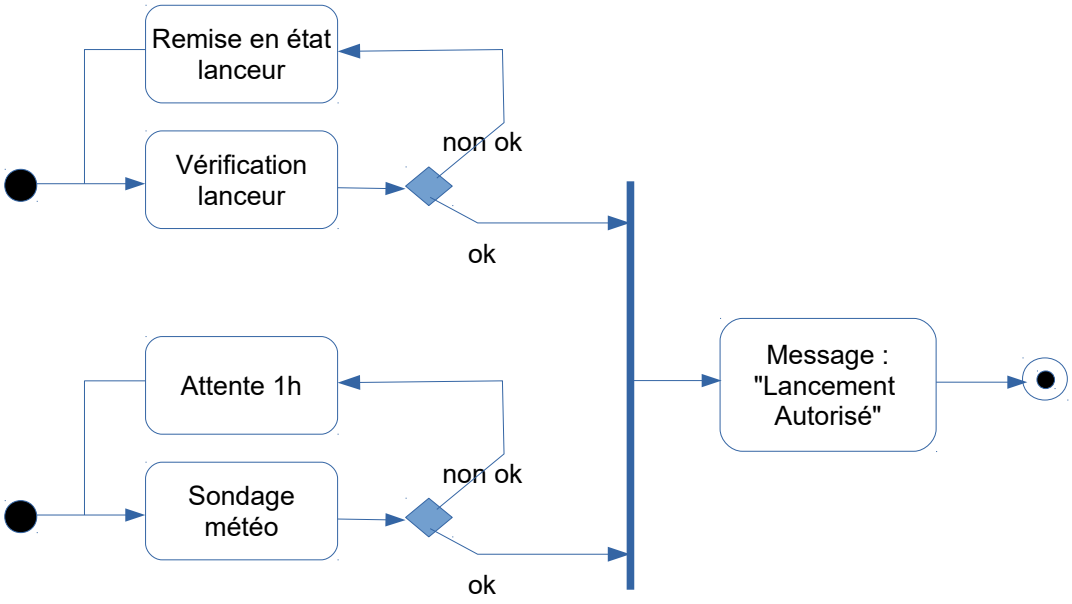
Nœud initial :

Un nœud initial est un nœud marquant le début d'un flot d'exécution attachés à une activité donnée. En effet, une activité pouvant être parcourue par plusieurs flots d'exécution, elle peut avoir plusieurs nœuds initiaux. Un nœud initial est toujours en relation avec l'événement qui déclenche le flot d'exécution à partir de ce nœud. Il est représenté par un petit cercle plein .

Nœud de fin d'activité :

Lorsqu'un flot d'exécution attaché à une activité atteint ce type de nœud, tous les flots d'exécution de l'activité englobante sont définitivement arrêtés. Un nœud de fin d'activité est représenté par un cercle vide contenant un petit cercle plein.

Exemple : lancement d'une fusée depuis un pas de tir :



REMARQUE : nœud de fin de flot d'exécution :

Un nœud de fin de flot est représenté par un cercle vide barré d'un X. Lorsqu'un flot d'exécution atteint ce type de nœud, ce flot est terminé. Les autres flots de l'activité englobante ne sont pas affectés.

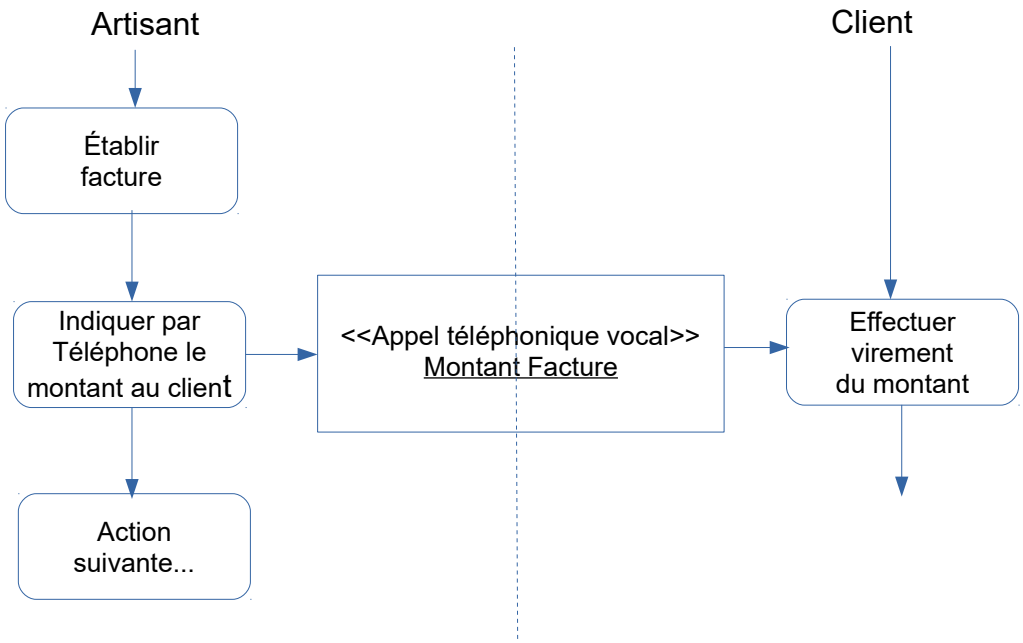
F.REPRÉSENTATION DES FLUX DE DONNÉES :

Les diagrammes d'activité d'UML permettent également de représenter graphiquement les FLOT (OU FLUX) DE DONNÉES entre les différentes activités. Pour cela, elle utilise des nœuds particuliers appelés NŒUDS D'OBJETS. Ces nœuds d'objet peuvent représenter :

- Des transferts de données immédiats entre deux actions ou activités ;
- Des tampons (buffers) de transfert acceptant les données de plusieurs nœuds ou fournissant des données à plusieurs nœuds suivant le mécanisme traditionnel de la "bufferisation" ;
- Des puits de données persistantes.

Ils sont représentés par des rectangles indiquant le type d'information transmise (ainsi que diverses autres informations).

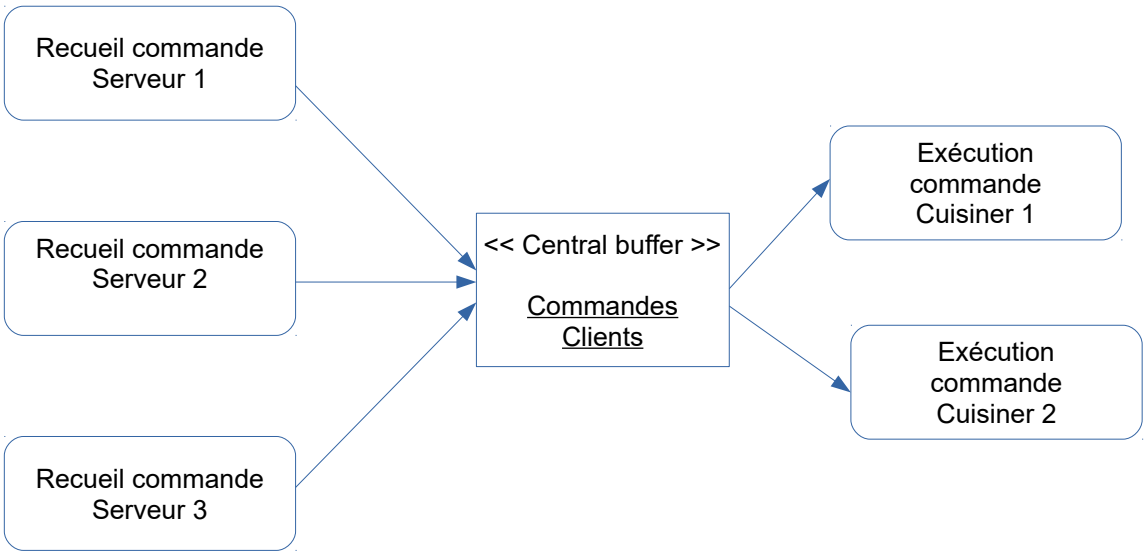
EXEMPLE 1 : Transfert immédiat (sans stockage intermédiaire ni bufferisation)



COMMENTAIRES :

- Un artisan établit une facture pour un client, puis communique à celui-ci (par téléphone) le montant de cette facture. Le client retourne alors le règlement ;
- La procédure exige une SYNCHRONISATION entre émetteur et récepteur. Il n'y a pas de bufferisation possible et l'information n'est pas persistante ;
- Le trait vertical détermine deux PARTITIONS : la partition ARTISAN et la partition CLIENT. Une partition regroupe en général les activités et actions initiées par une classe ou un acteur donné.

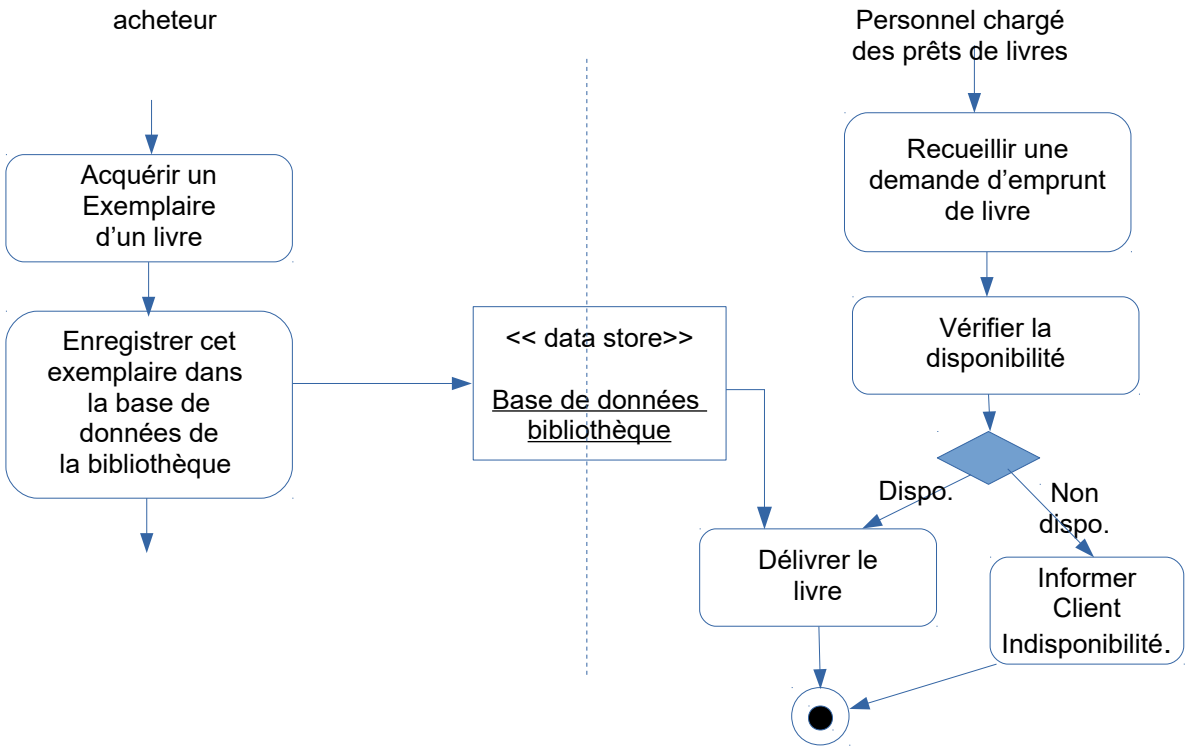
EXEMPLE 2: Bufferisation.



COMMENTAIRES :

- Un restaurant dispose de 3 personnels de salle qui recueillent les commandes des clients. Les bordereaux de commande sont épinglés sur un tableau dans la cuisine. Chacun des deux cuisiniers vient récupérer ces bordereaux en fonction de sa propre activité ;
- Le tableau des bordereaux fonctionne comme un BUFFER qui conserve l'information jusqu'à ce que quelqu'un la récupère. L'information (le bordereau) disparaît alors du tableau ;
- Les échanges entre serveurs et cuisiniers sont ASYNCHRONES.

EXEMPLE 3 : Données persistantes .



COMMENTAIRE :

- Dans une bibliothèque, la personne chargée des achats acquiert des livres et les enregistre dans la base de données (ces informations sont donc persistantes) ;
- Le personnel d'accueil recueille les demandes de prêt du public, va vérifier la disponibilité des livres demandés en interrogeant la base de données, puis délivre le livre ou informe le client de son indisponibilité ;
- Les échanges entre acheteurs et personnels chargés des emprunts sont ASYNCHRONES.

VIII.4.LES DIAGRAMMES DE FLUX DE DONNÉES SA/RT:

VIII.4.1.GÉNÉRALITÉS:

Les diagrammes de flux de données sont utilisés sous diverses formes par de nombreuses méthodes de développement, tant en analyse fonctionnelle qu'en analyse organique.

Lorsqu'ils sont utilisés en analyse organique, leur démarche commune consiste en une modélisation graphique des activités du système étudié, les principaux éléments graphiques constitutifs de cette modélisation étant:

- Des «bulles» ou des «boites» représentant des fonction, des activités ou des traitements;
- Des flèches représentant des flux de données ou de commandes;
- D'autres symbole graphiques tels que des sources, des récepteurs ou des "puits" de données ou des mécanismes de synchronisation.

Les diagrammes de flux sont utilisés en particulier par la méthode Hartey-Pirbhaï (SA-RT), dont l'emploi principal est la conception d'applications à contraintes "temps réel". Le langage de conception U.M.L en fait également usage, notamment dans ses "diagrammes d'activités".

VIII.4.2.SYMBOLIQUE DES DIAGRAMMES DE FLUX:

VIII.4.2.1.AVERTISSEMENT:

La description des différentes entités graphiques employées dans les diagrammes de flux correspond à celle qui est préconisée par SA/RT, avec des "enrichissements" empruntés à d'autres méthodes (symbolisme S.D, DARTS, RTSA) pour la représentation des données, des commandes et de la synchronisation.

VIII.4.2.2.LES SYMBOLES D'ACTIVITÉS:

Une ACTIVITÉ (ou "processus") est un ensemble de traitements, séquentiels ou non, concourant à un même but. Une activité est, en général, représentée graphiquement par une bulle ou une boîte à angles arrondis. L'identifiant d'une activité doit, en théorie, se résumer à un verbe (ou un groupe verbal) d'action et un groupe complément d'objet direct.

La méthode SA-RT distingue deux sortes d'activités :

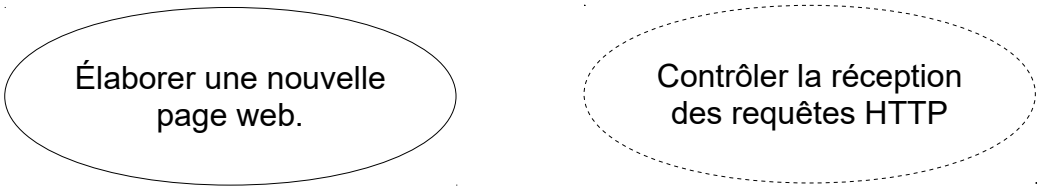
- Les processus "Fonctionnels" ou "de transformation" qui supportent les traitements induits par les exigences fonctionnelles exprimées dans les documents de spécification des besoins ;

- Les processus "de contrôle", qui supportent la logique de pilotage des processus fonctionnels ;

Les processus(ou activités) de contrôle gèrent les événements externes et vont, en fonction de ces événements, piloter les processus fonctionnels (Autoriser, Inhiber, Déclencher). En retour, les processus fonctionnels fournissent aux processus de contrôle tous les ÉTATS nécessaires aux prises de décision.

Les activités de contrôle sont différenciées graphiquement des activités fonctionnelles par le fait que les contours de leur "boîte" sont tracés en pointillés :

Exemples:



Activité fonctionnelle

Activité de contrôle

Concrètement, une activité correspond à la notion logicielle de TÂCHE. Dans un diagramme de flux, et en l'absence de spécifications contraires, les différentes activités décrites peuvent s'exécuter SIMULTANÉMENT ou non.

VIII.4.2.3.REPRÉSENTATION DES DONNES RÉMANENTES:

Une donnée est dite "RÉMANENTE" lorsqu'elle "survit" à l'activité qui l'a créée et reste donc disponible pour une autre utilisation. C'est le cas:

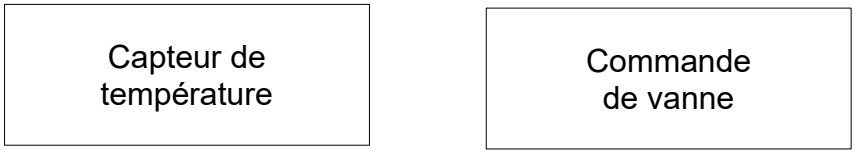
- Des données enregistrées sur un support permanent de type disque, bande magnétique, CD-ROM, etc.;
- Des données enregistrée dans une partie de la mémoire vive PARTAGEABLE entre différents PROCESSUS LOGICIELS ("SHEARED MEMORY" des systèmes d'exploitation UNIX, LINUX, WINDOWS, etc..);
- Des données déclarées STATIQUES. Une donnée STATIQUE a la particularité de ne pas être ré-allouée après la fin d'exécution de la procédure qui l'a crée;

Une donnée RÉMANENTE est représentée par deux lignes parallèles. On appelle ce symbole «réservoir de données» ou "puit de données". Son identifiant est un groupe nominal:

Mesures brutes

VIII.4.2.4.REPRÉSENTATION DES ÉLÉMENTS DE LA PÉRIPHÉRIE DU SYSTÈME:

On représente les producteurs ou les consommateurs de données appartenant à la périphérie du système sous formes de rectangles:



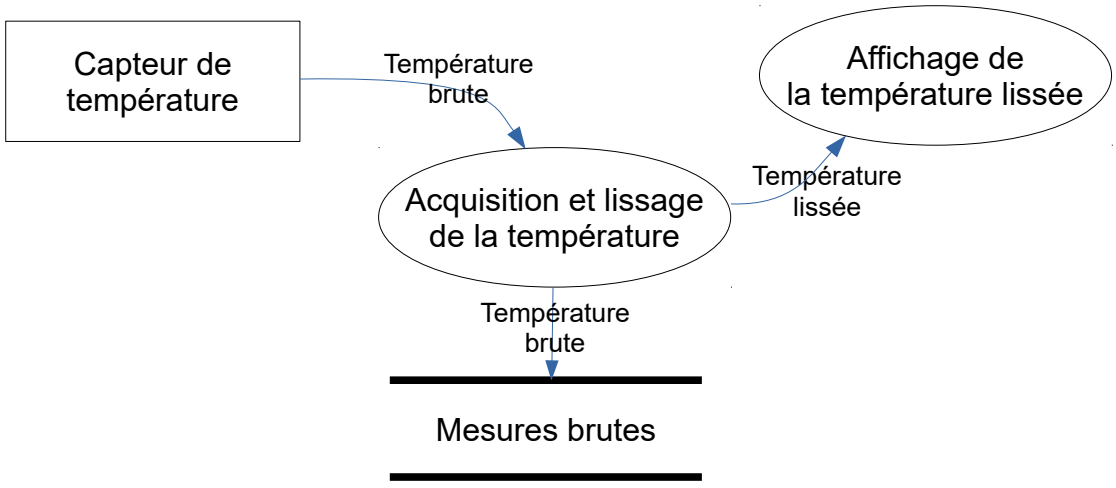
VIII.4.2.5.REPRÉSENTATION DES FLUX:

VIII.4.2.5.1.REMARQUE:

Le terme "FLOT" est souvent employé en alternative au terme "flux".

VIII.4.2.5.2.LES FLUX DE DONNÉES:

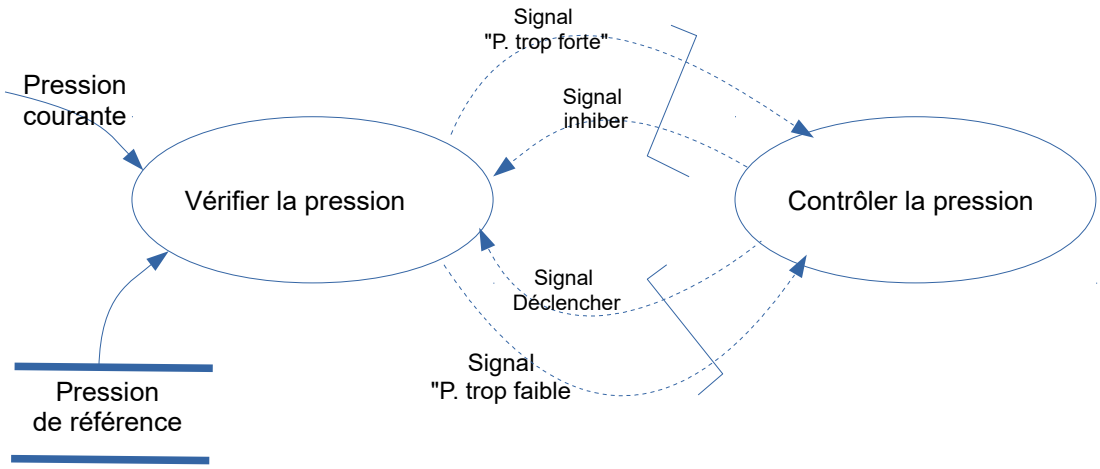
A l'inverse d'une donnée statique, un Flux de Données représente une «donnée en mouvement», c'est à dire un messages échangé entre deux activité d'un système, une activité et un «réservoir de données», ou encore une activité et un élément de la périphérie. Une telle donnée est représentée par une flèche pleine continue. Son identifiant est également un groupe verbal:



VIII.4.2.5.3.LES FLUX DE CONTRÔLE:

Un FLUX DE CONTRÔLE transporte les événements ou informations qui conditionnent directement ou indirectement l'exécution des activités qui le reçoivent. Un FLUX de contrôle est représenté par une flèche en pointillée avec un identifiant :

Exemple :



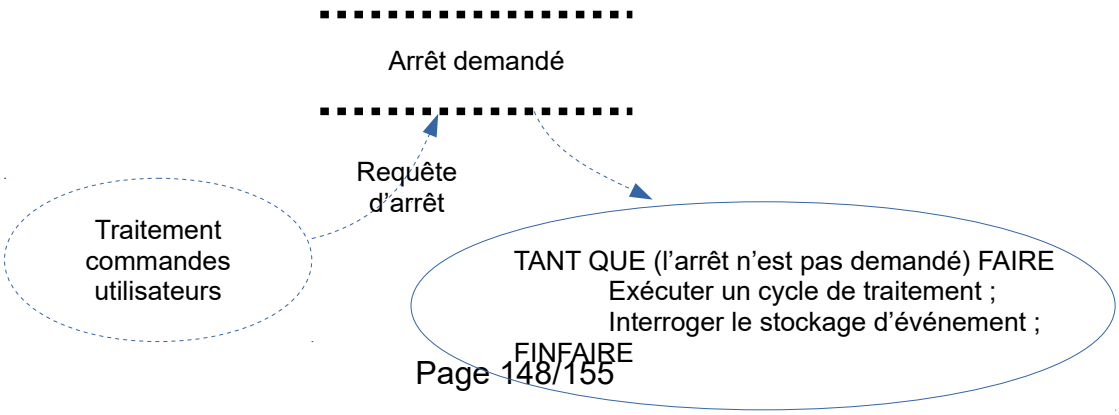
VIII.4.2.5.4.LES ÉVÉNEMENTS:

Un ÉVÉNEMENT est un flux de contrôle particulier qui occasionne une interruption prioritaire sur le récepteur (de type logiciel ou matériel, signal posix, etc...). Un événement ne peut véhiculer qu'une donnée très simple (en général, une valeur scalaire ou une adresse). Cette donnée doit être disponible en même temps que l'événement, sans que le récepteur soit obliger d'accéder à un réservoir de données globales.

VIII.4.2.5.5.LE STOCKAGE D'ÉVÉNEMENTS:

La symbolique SA/RT comprend des objets "stockage d'événements" dont la fonction est analogue à celle des "puits de données": ils permettent la mémorisation d'un ou plusieurs flux d'événements. Ceux-ci restent à la disposition des différentes activités pour une utilisation ultérieure. Le stockage d'événements est représenté par deux lignes parallèles en pointillé:

Exemple: dans l'exemple suivant, la requête d'arrêt des traitements n'est pas exploitée directement par le processus de traitement, mais stockée. De ce fait, elle et n'est prise en compte qu'une fois que le cycle de traitement en cours est terminé, ce qui permet d'éviter d'interrompre un cycle sans l'achever:



VIII.4.2.5.6.REMARQUES:

A.REMARQUE N°1:

Lorsqu'une activité reçoit un flux de contrôle ou la notification d'un événement, cette réception va occasionner la modification de l'exécution de cette activité. En effet;

- La réception d'un événement va occasionner le transfert automatique par le système d'exploitation du flux d'exécution de l'activité cible vers un "contrôleur d'événement" qui se chargera d'aiguiller l'exécution vers la "fonction de callback" correspondante;
- La réception d'un flux de contrôle aura un effet semblable à la différence que l'interprétation du "message de contrôle" pourra être différée dans le temps, mais elle aura également pour effet de modifier le flux d'exécution.

B.REMARQUE N°2:

Lorsque deux activités sont reliées directement par un flux de données pur, cela implique forcément une synchronisation entre ces activités: en effet, les données émises sont sensé être acquises par le récepteur sans délais autre que le temps de propagation du message, et sans passer par un réservoir de données qui permettrait de conserver ces données jusqu'à ce que le récepteur soit disponible pour les acquérir. Ceci n'est possible que dans deux cas:

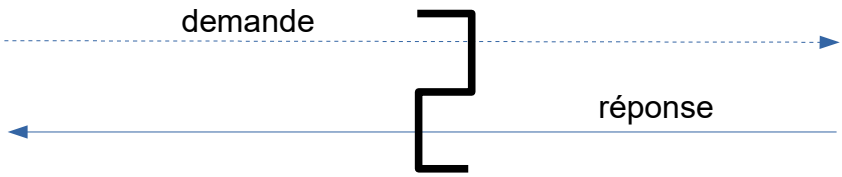
- Soit le récepteur reste EN ATTENTE des données;
- Soit le transfert est réalisé grâce à un APPEL PROCÉDURAL entre émetteur et récepteur. Dans ce dernier cas, on ne peut parler de parallélisme des activités.

Dans tout autre cas, l'échange de données entre activités parallèles nécessite soit le passage par un réservoir de données, soit l'utilisation en préalable du transfert de données d'un ÉVÉNEMENT permettant de synchroniser le récepteur.

VIII.4.2.6.SYMBOLISMES PARTICULIERS:

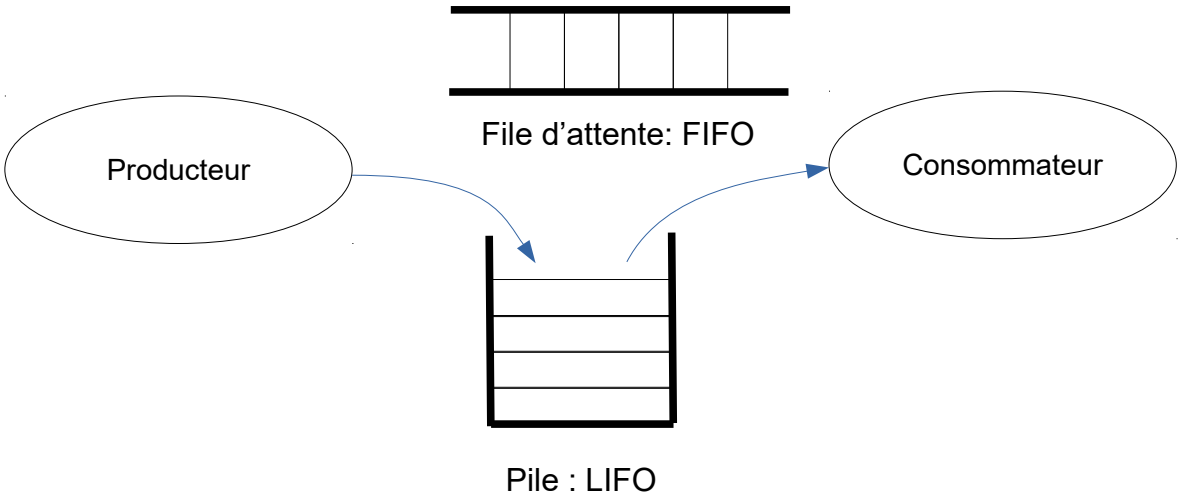
VIII.4.2.6.1.FLUX CONSÉCUTIFS:

Lorsque deux flux représentent un couple «demande-réponse», on peut le signaler par le symbolisme suivant (DARTS):



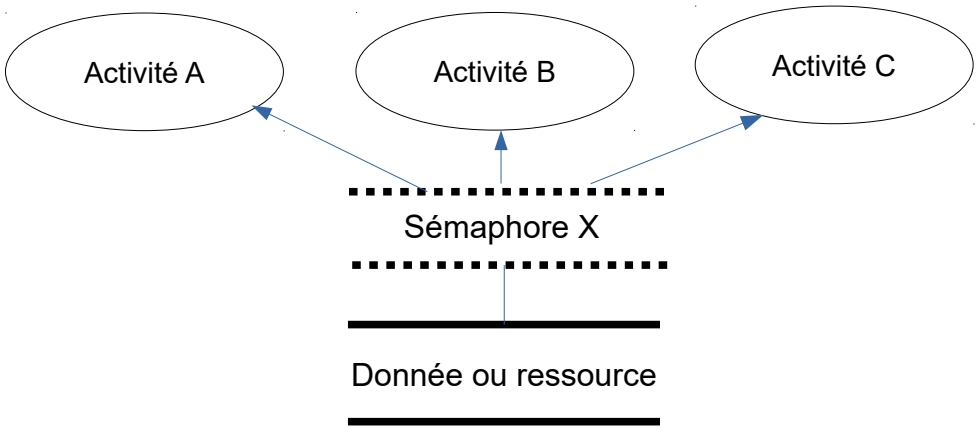
VIII.4.2.6.2.FILES D'ATTENTE ET PILES:

Le symbolisme DARTS permet également de figurer les files d'attente ou les piles entre processus:



VIII.4.2.6.3.LES SÉMAPHORES:

Le symbolisme SART représente les sémaphores sous l'aspect de deux lignes parallèles pointillées:



VIII.4.3.EXEMPLE D'ANALYSE DESCENDANTE PAR LES DIAGRAMMES DE FLUX:

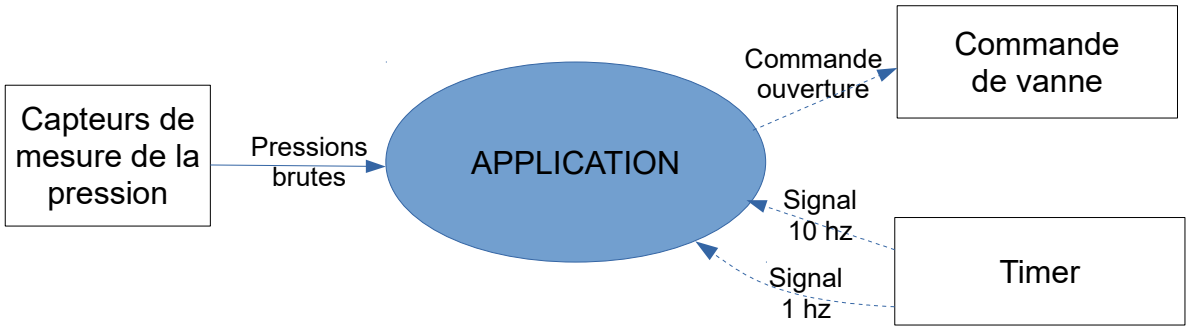
Une analyse par les diagrammes de flux peut être menée par raffinages successifs. La démarche est explicitée par l'exemple ci-après:

VIII.4.3.1.APPLICATION A ANALYSER:

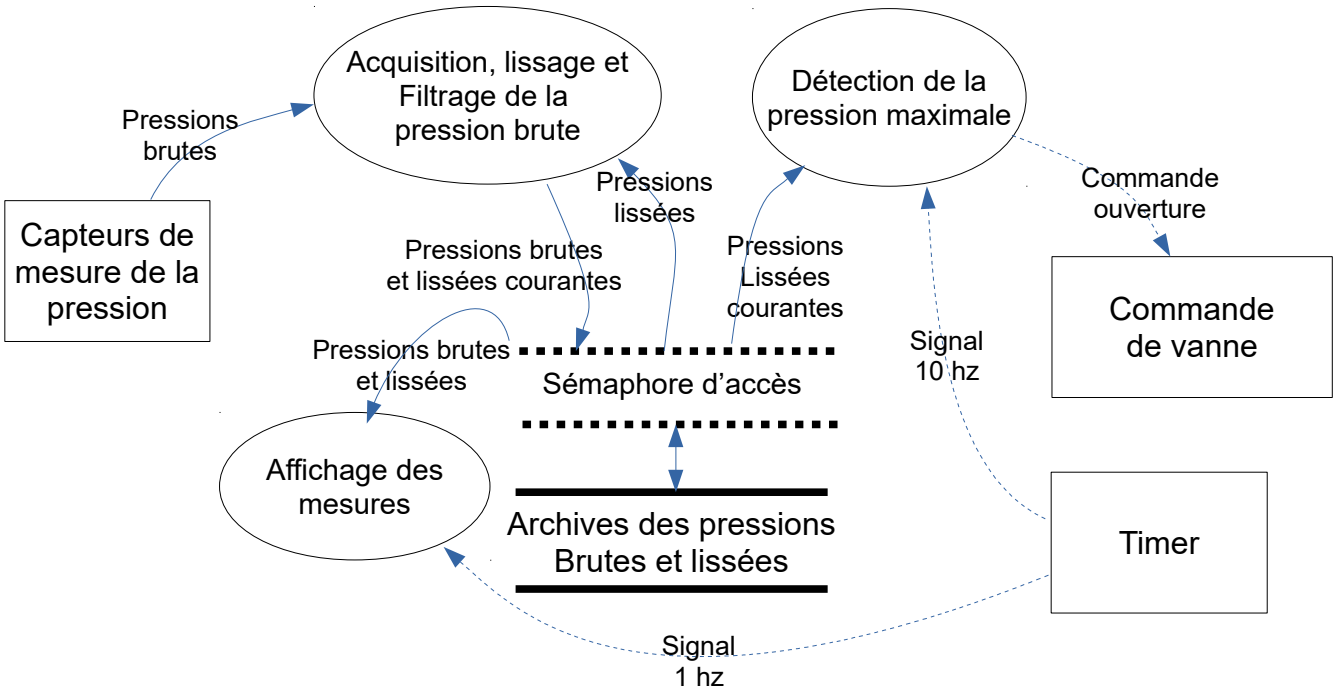
L'application permet de surveiller et de sécuriser un réservoir sous pression. Les exigences associées sont:

- Éviter toute surpression dans le réservoir par ouverture d'une vanne d'évacuation dès que la pression lissée (débarrassée des bruts de mesures) atteint la pression maximale de service;
- Surveiller les variations de pression par l'affichage des pressions instantanées et de l'historique des pressions brutes et lissées.

VIII.4.3.2.DIAGRAMME D'ENVIRONNEMENT:



VIII.4.3.3.RAFFINAGE (PREMIER NIVEAU):



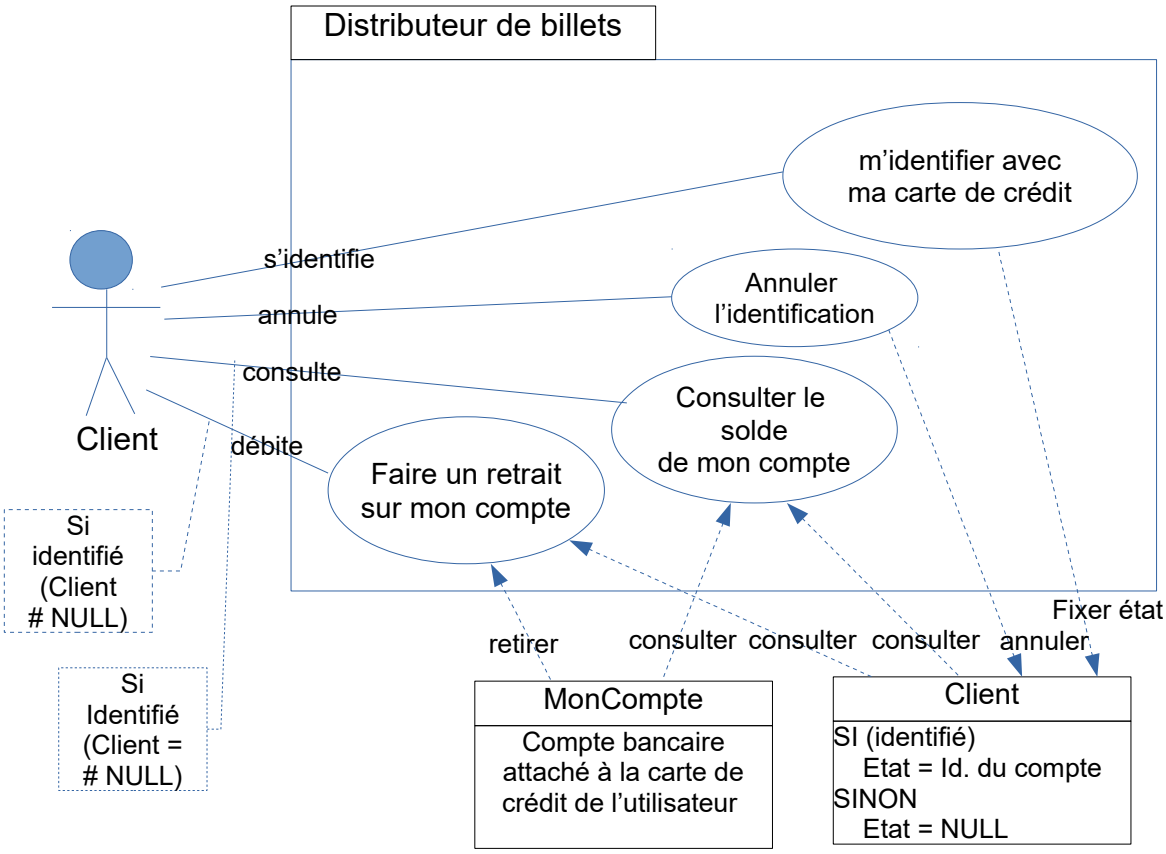
VIII.5.MÉTHODE POUR IDENTIFIER LES ACTIVITÉS D'UNE APPLICATION :

Les activités logicielles d'une application peuvent assez facilement se déduire des EXIGENCES FONCTIONNELLES du client (ou, dans le cadre des méthodes agiles, des CAS D'UTILISATION ou des SCENARII CLIENTS).

A cette occasion, les données communes à plusieurs activités peuvent également être identifiées.

EXEMPLE :

Soit de diagramme de cas d'utilisation suivant :



Ce diagramme permet d'identifier quatre activités:

- Identification_Client ;
- Annulation_Id ;
- Consultation_compte ;
- Retrait_sur_compte.

Accessoirement, il permet également d'identifier les données communes suivantes:

- **MonCompte**, qui est le compte bancaire de l'utilisateur.

- **Utilisateur** qui est un indicateur d'état renseignant sur le statut de l'utilisateur (identifié ou non identifié).

REMARQUES

- *Dans le cas présenté, nous pouvons considérer que ces activités sont des TRANSACTIONS et que, de ce fait, elles ne peuvent être interrompues dans leur déroulement, même par la touche "annulation" ;*
- *Cette démarche n'implique pas que les besoins soient exprimés sous la forme de CAS D'UTILISATION. Elle peut être menée de la même manière si la spécification de besoin est exprimée sous la forme d'EXIGENCES, de CAS D'UTILISATION ou de SCENARII CLIENTS.*

VIII.6.CARACTÉRISATION DES ÉCHANGES D'INFORMATIONS ENTRE ACTIVITÉS :

Nous pouvons distinguer trois manières différentes de transmettre des informations d'une activité à une autre : les modes SYNCHRONES, BUFFERISÉS et ASYNCHRONES :

VIII.6.1.LA TRANSMISSION SYNCHRONE :

La caractéristique de la transmission synchrone est qu'elle exige la synchronisation des activités émettrices et réceptrices : l'émetteur émet des informations qui doivent être immédiatement récupérées par le récepteur. Ceci implique :

- Soit que l'activité réceptrice est en attente du flux d'informations de l'émetteur (par exemple, le récepteur est à l'état "lecture bloquante" sur le port de réception du message) ;
- Soit que l'activité émettrice effectue un appel procédural vers l'activité réceptrice (avec ou sans passage d'arguments). Nous pouvons ranger dans cette catégorie les appels de procédures à distance.

REMARQUE: Dans la notation des diagrammes d'activité UML, ce type de transmission est représenté par un nœud d'objet non stéréotypé. Dans la notation SA-RT/DARTS, il est représenté par un flux reliant directement les deux activités.

VIII.6.2.LA TRANSMISSION BUFFERISÉE :

Dans ce type de transmission, les informations de l'émetteur sont stockées dans un TAMPON (ou buffer) fonctionnant comme une file d'attente. De ce fait, le récepteur n'a plus besoin de se synchroniser avec l'émetteur car il peut récupérer les messages contenus dans le buffer "à son rythme", par des lectures non bloquantes.

REMARQUES:

- Le tampon peut être lu par plusieurs récepteurs, mais lorsqu'une information a été lue, elle disparaît de la file d'attente : un message donné ne peut donc être transmis qu'à un seul récepteur ;
- Dans la notation des diagrammes d'activité UML, ce type de transmission est représenté par un nœud d'objet stéréotypé <<central buffer>>. Dans la notation SA-RT/DARTS, le tampon est représenté par une file d'attente.

VIII.6.3.LA TRANSMISSION ASYNCHRONE :

Dans ce type de transmission l'information émise par l'émetteur est stockée dans un tampon qui assure la persistance des données : le tampon ne fonctionne pas comme une file d'attente mais comme un "puits de données persistantes" :

- Les messages correspondant à un émetteur donné sont stockées dans un emplacement réservé à cet émetteur, un nouveau message reçu écrasant l'ancien message ;
- Lorsqu'un récepteur lit un message, celui-ci est conservé dans le tampon.

REMARQUE: Dans la notation des diagrammes d'activité UML, ce type de transmission est représenté par un nœud d'objet stéréotypé <<datastore>>. Dans la notation SA-RT/DARTS, il est représenté par un "puits de données persistantes" : un flux d'informations va de l'émetteur au tampon, un autre flux totalement asynchrone du premier va du tampon au récepteur.