Prolegomenes UML

Introduction

1 Objectifs de ce cours

En informatique, on s'efforce en permanence de résoudre le hiatus entre le *monde réel*, évolutif et ambigü, et le *monde informatique*, qui propose des langages codifiés, avec une sémantique unique.
Ce souci de modélisation ne provient pas d'un besoin effréné de dessiner des diagrammes pour l'amour de l'art, mais d'un besoin réel, face à un problème donné, de concevoir et de communiquer autour du projet, et éventuellement de programmer une application informatique autour de ce projet. Les outils de modélisation proposent des supports pour effectuer ces tâches nécessaires. Ils ne sont pas nés de l'imagination débridée d'un chercheur mégalomane, mais de la **pratique** de centaines de milliers d'individus qui ont souhaité, à un instant donné, travailler en groupe et parler des mêmes notions autour de leur projet commun.
Le but de ce cours est de faire un inventaire de la méthode de modélisation UML, de proposer un tour d'horizon des outils qu'elle propose, afin que le lecteur puisse acquérir les bases de ce langage de partage et l'utiliser dans sa pratique quotidienne de conception.
Les applications de la modélisation touchent des domaines divers et variés, et peuvent intervenir dans chaque discipline scientifique où des phénomènes complexes doivent être appréhendés de façon commune. Certains exemples de ce cours en témoignent. Nous avons volontairement évité de prendre des exemples uniquement issus du monde informatique, car notre sentiment profond est que la modélisation est une valeur universelle de la démarche scientifique.

2 Difficultés de la modélisation

Les difficultés de la modélisation proviennent de plusieurs types d'obstacles

- Les spécifications sont parfois imprécises, incomplètes, ou incohérentes
- La taille et la complexité des systèmes peuvent être importantes et surtout sont croissantes, car
  - les besoins et les fonctionnalités augmentent
  - la technologie évolue rapidement
  - les architectures se diversifient
  - il faut assurer l'interface avec le métier, et les domaines d'application évoluent
- Les applications sont destinées à évoluer en fonction de
  - l'évolution des besoins des utilisateurs
  - la réorientation de l'application
  - l'évolution de l'environnement technique (matériel et logiciel)
- La gestion des équipes peut également poser problème à cause de
  - la taille croissante des équipes
  - la spécialisation technique
  - la spécialisation du métier

3 Les méthodes de modélisation

Les méthodes de modélisation fournissent alors des guides structurants fondés sur les concepts suivants :la décomposition du travail, l'organisation des phases, les concepts fondateurs, des représentations semi-formelles. Tous ces principes fondateurs ont pour objectif de produire une démarche reproductible pour obtenir des résultats fiables.

3.1 La décomposition du travail

Elle repose sur la description de différentes ***phases*** : analyse, conception, codage, validation, etc.Elle touche à différents niveaux d'abstraction :

- le niveau conceptuel (description des besoins)
- le niveau logique (solution informatique abstraite)
- le niveau physique (solution informatique concrète)

3.2 L'organisation du travail

Elle consiste en la description du processus de développement, sous la forme de séparation en phases séquentielles et d'itération sur les phases

3.3 Les concepts fondateurs

Ils fondent l'approche du problème et l'expression de la solution. Par exemple, ce sont les notions de **classe , signal , état , fonction** , etc.

3.4 Les représentations semi-formelles

Elles sont des représentations partiellement codifiées basées sur les concepts fondateurs : diagrammes, formulaires, etc. Elles sont également le support de différentes activités : réflexion, spécification, communication, documentation, mémorisation (trace)
En résumé, une méthode d'analyse et de conception propose une démarche qui distingue les étapes du développement dans le cycle de vie du logiciel (modularité, réduction de la complexité, réutilisabilité éventuelle, abstraction), et elle s'appuie sur un formalisme de représentation qui facilite la communication, l'organisation et la vérification. Le langage de modélisation produit des documents (modèles) qui facilitent les retours

Présentation d'UML

L'acronyme UML signifie "Unified Modeling Language". C'est donc un langage de modélisation, qui véhicule en particulier les concepts des approches par objets : ***classe, instance, classification, etc.*** mais intégrant d'autres aspects : associations, fonctionnalités, événements, états, séquences, etc.
UML bénéficie des avantages procurés par l'approche objet :

- La simplicité
- La facilité pour coder et réutiliser
- Un modèle plus proche de la réalité
  - Description plus précise des combinaisons (données, opérations)
  - Décomposition basée sur une "classification naturelle"
  - Facile à comprendre et à maintenir
- La stabilité : de petites évolutions peuvent être prises en compte sans changements massifs

La portée d'UML s'explique par l'omniprésence de l'objet :

- Omniprésence technique de l'Objet dans les langages de programmation, les bases de données, les interfaces graphiques, etc. ainsi que dans les méthodes d'analyse et de conception.
- Universalité de l'Objet : la notion d'objet, plus proche du monde réel, est compréhensible par tous et facilite la communication entre tous les intervenants d'un projet.

1 Genèse d'UML

Au début des années 90, il existe une cinquantaine de méthodes objet , liées uniquement par un consensus autour d'idées communes (objets, classes, sous-systèmes, ...). Ce foisonnement est suivi de la recherche d'un *langage commun unique* utilisable par toute méthode de conception fondée sur l'objet, utilisable dans toutes les phases du cycle de vie des sytèmes modélisés, et compatible avec les techniques de réalisation actuelles.

La genèse d'UML

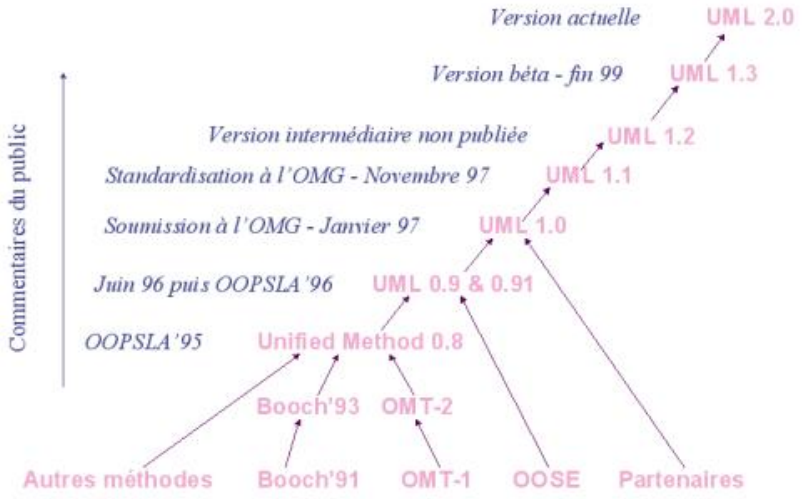


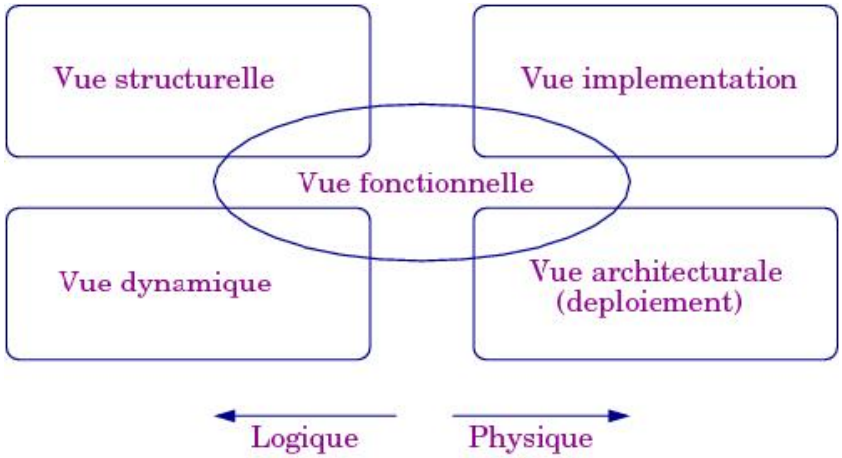
Figure 77 : La genèse d'UML

La figure précédente indique le processus de genèse de l'UML.

- OMG : l' **Object Management Group** est un consortium industriel à but non lucratif. L'une de ses activités essentielles est l'adoption de standards en matière de conception, execution et maintenance de logiciels.
- OOPSLA : l'acronyme signifie **Object-Oriented Programming, Systems, Languages and Applications** et désigne une conférence annuelle de portée internationale, se tenant depuis 1986. Cette conférence a joué un rôle fédérateur autour des standards proposés par l'OMG.
- OMT : **Object-Modeling Technique** est une méthode de modélisation développée au début des années 90 par Rumbaugh, Blaha, Premerlani, Eddy et Lorensen pour développer les sytèmes orientés objet.
- Booch : la **méthode de Booch** est une technique utilisée en conception de logiciels, développée par Grady Booch. On retrouve dans UML des éléments graphiques de la méthode de Booch. Les aspects méthodologiques de cette méthode ont été incorporés notamment dans la méthodologie RUP présentée dans ce cours.
- OOSE : l'acronyme signifie **Object-Oriented Software Engineering** et désigne une méthodologie et un langage de modélisation développpé par Ivar Jacobson en 1992. C'est la première méthode à introduire les cas d'utilisation.

2 Concepts généraux

On développe plusieurs points de vue sur le système



UML propose quatre modèles pour concrétiser ces points de vue :

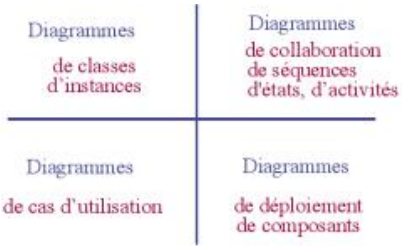
- Modèle structurel : définit les types d'objets et leurs relations
- Modèle Implémentation : description des composants et fichiers de bases de données ainsi que la projection sur le matériel
- Modèle d'utilisation : indique les fonctionnalités du système
- Modèle Dynamique : représente les stimuli des objets et leurs réponses

Chaque modèle est une représentation abstraite d 'une réalité, il fournit une image simplifiée du monde réel selon un certain point de vue. Il permet :

- de comprendre et visualiser (en réduisant la complexité)
- de communiquer (à partir d'un « langage » commun à travers un nombre restreint de concepts)
- de valider (contrôle de la cohérence, simuler, tester ...)

3 Diagrammes (représentations graphiques des modèles)

Ils sont le support de la réflexion et de l'analyse du modèle.



Les diagrammes permettent d'adopter une démarche uniforme sur le cycle de vie du système. Ils nécessitent de partager une même notation à toutes les étapes (Analyse, Conception, Implémentation)
Les diagrammes sont majoritairement des graphes

- Noeuds
- Arcs 
- Chaines de caractères : noms, étiquettes, mots clefs << interface >>
- Contraintes : Texte libre, langage de programmation du type OCL, etc.
- Notes 

Modèle fonctionnel : cas d'utilisation

1 Introduction

Les cas d'utilisation, ou « USE CASE » décrivent les fonctionnalités externes du système. Il s'agit de modèles descriptifs qui prennent le point de vue des utilisateurs. Ils précisent notamment :

- les interactions avec les acteurs extérieurs
- la manière d'utiliser le système

2 Diagramme de cas d'utilisation

Pour concevoir un diagramme de cas d'utilisation, on part de l'analyse des besoins, en faisant interagir deux concepts :

DÉFINITION : ACTEUR

Un acteur est une entité extérieure au système et interagissant avec celui-ci. Les acteurs peuvent être des acteurs humains ou acteurs « machine » (système extérieur communiquant avec le système étudié)

DÉFINITION : CAS D'UTILISATION

Un cas d'utilisation est une manière d'utiliser le système, ou suite d'événements notable du point de vue de l 'utilisateur

Les deux concepts fondateurs des diagrammes de cas d'utilisation

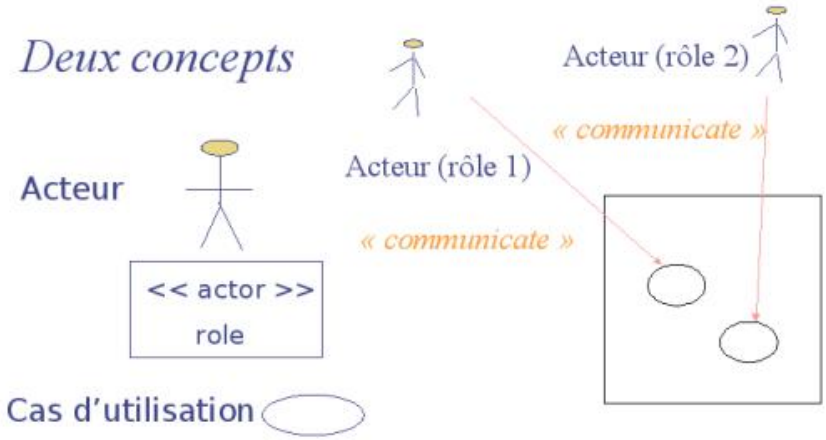


Figure 78 : Les deux concepts fondateurs des diagrammes de cas d'utilisation

Les cas d 'utilisation peuvent être liés par des relations :

- d'utilisation *« include »* (le cas origine contient obligatoirement l'autre)
- de raffinement *« extend »* (le cas origine peut être ajouté optionnellement)
- de généralisation/spécialisation *« generalizes »* ou *« specializes »* (le cas origine peut être réalisé de différentes façons décrites dans les cas spécialisés)

Les relations extend et include

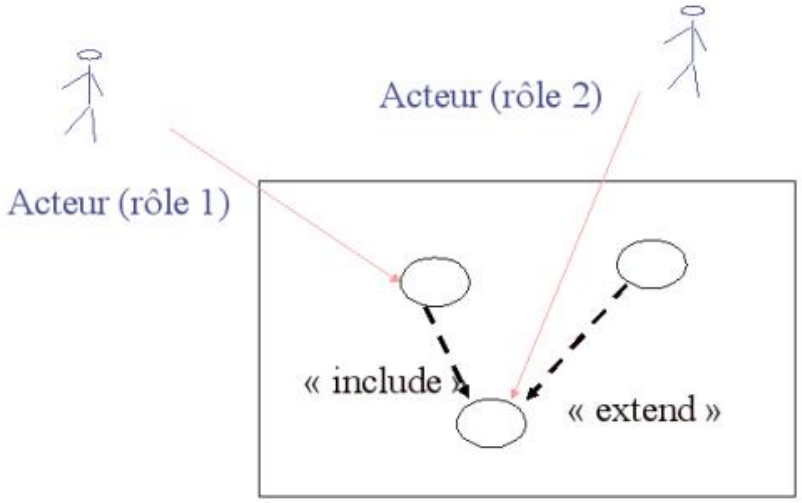


Figure 79 : Les relations extend et include

Le contexte statique est l'ensemble des acteurs et des relations qui apparaissent en dehors du système proporement dit dans le diagramme de cas d'utilisation. Doivent y être définis :

- les **acteurs** et les **rôles** qui leur sont associés (une même personne ou un même système extérieur peut bien entendu endosser plusieurs rôles)
- les **cardinalités**

DÉFINITION : CARDINALITÉ

La cardinalité d'un lien entre un acteur et le système détermine combien d'acteurs jouant le même rôle peuvent interagir en même temps sur le système.

Par exemple, sur le diagramme de la figure suivante, il ne peut y avoir au plus qu'un acteur jouant le rôle 1 interagissant avec le système à un instant donné.

Diagramme du contexte statique

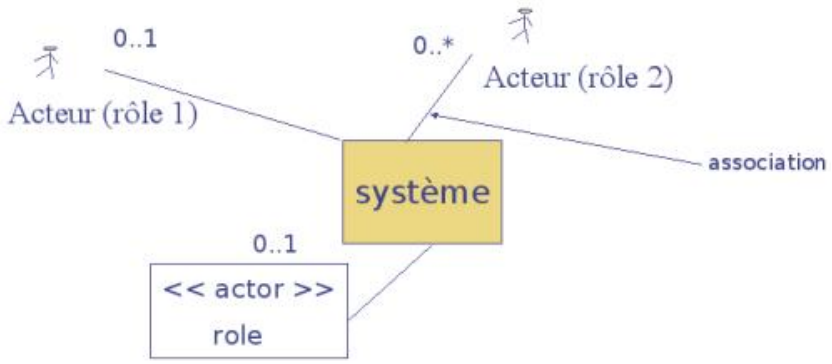


Figure 80 : Diagramme du contexte statique

3 Exemples de diagrammes de cas d'utilisation

Cas d'utilisation spécifiant deux acteurs

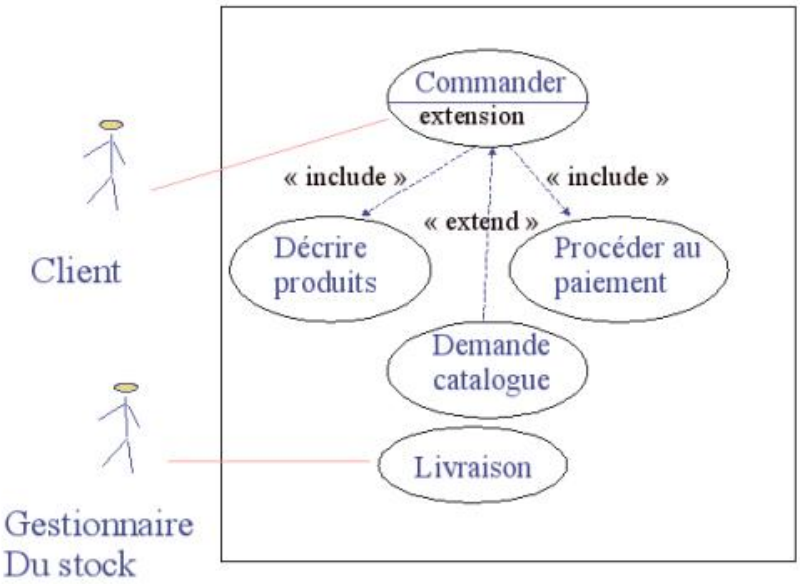


Figure 81 : Cas d'utilisation spécifiant deux acteurs

Cas d'utilisation avec une spécialisation

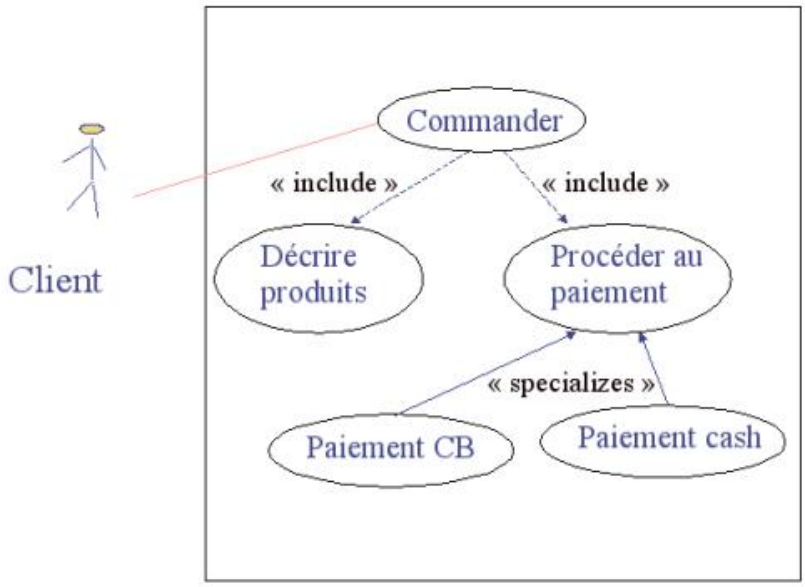


Figure 82 : Cas d'utilisation avec une spécialisation

4 Conclusion

Les diagrammes de cas d'utilisation permettent de

- Délimiter le système
  - ce qui est extérieur et qui communique avec le système
  - ce qui est interne au système
- Définir les fonctionnalités du système du point de vue des utilisateurs
- Donner une description cohérente de toutes les vues que l'on peut avoir du système

Ils s'accompagnent usuellement de descriptions complémentaires (textes, diagramme de séquences ou d'activités)
Exemple :

- Sommaire d'identification
  - Titre, résumé, acteurs, dates création maj, version, auteurs
- Description des enchaînements
  - Pré-conditions, scénario nominal, alternatives, exceptions,
  - post-conditions
- Besoins IHM
- Contraintes non fonctionnelles
  - Temps de réponse, concurrence, ressources machine, etc.

Modèle structurel

1 Introduction

En UML, le modèle structurel ou statique est décrit à l'aide de deux sortes de diagrammes

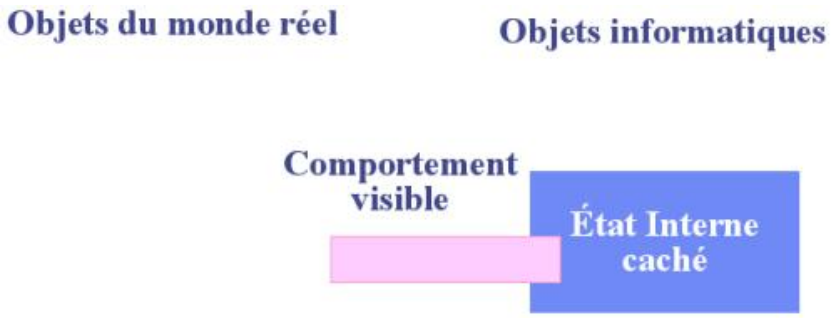
DÉFINITION : DIAGRAMME DE CLASSES

Un diagramme de classe est la description de tout ou d'une partie du système d'une manière abstraite, en termes de classes, de structure et d'associations;

DÉFINITION : DIAGRAMME D'OBJETS

Un diagramme d'objet est la description d'exemples de configuration de tout ou partie du système, en termes d'objets, de valeurs et de liens.

2 Notion d'objet

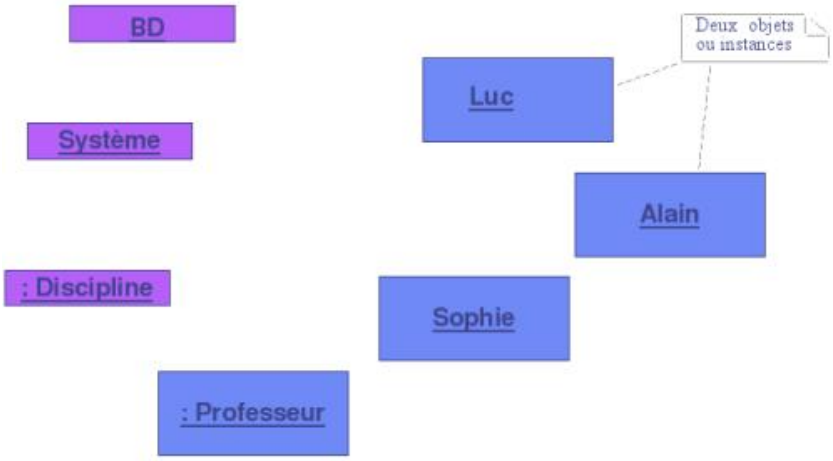


DÉFINITION : OBJET

Un objet se caractérise par :

- *son **état** : l'objet évolue au cours du temps*
- *son **comportement** : la description de ses actions et réactions*
- *son **identité** : l'essence de l'objet*

Le comportement influe sur l'état, l'état reflète les comportements passés



3 Première abstraction : la notion de classe

DÉFINITION : CLASSE

Une classe peut être vue comme

- *la description en intension d'un groupe d'objets ayant*
  - *même structure (même ensemble d'attributs),*
  - *même comportement (mêmes opérations),*
  - *une sémantique commune.*
- *la « génitrice » des objets ou instances*
- *le « conteneur » (extension) de toutes ses instances*

Exemple de lien entre classe et instance

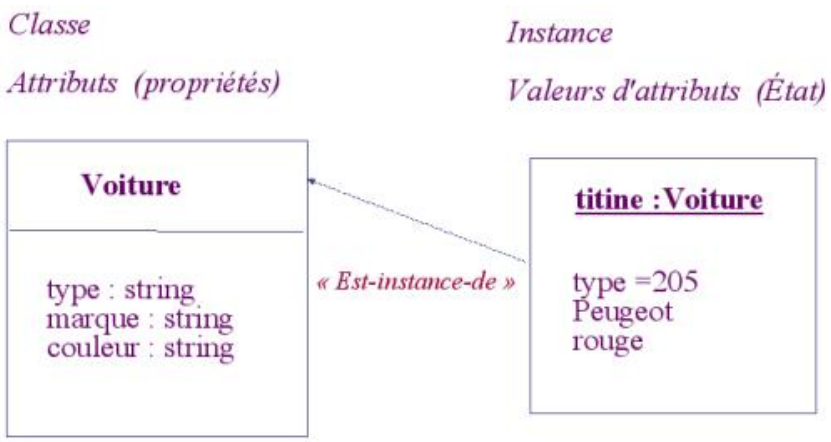


Figure 83 : Exemple de lien entre classe et instance

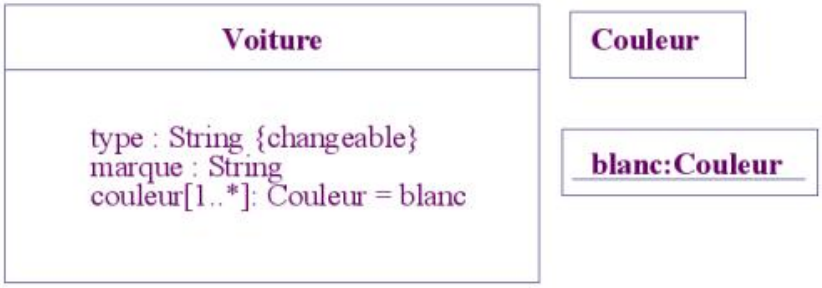
3.1 Attributs (propriétés)

DÉFINITION : ATTRIBUT

*Les **attributs** servent à décrire les propriétés portées par les classes. L'ensemble des valeurs des attributs d'une classe à un instant donné est une description de son état à cet instant.*

3.1.1 Attribut

[Visibilité] nom [[multiplicité]][: type][=valeur initiale][{ propriétés}]



3.1.2 Attribut de classe

DÉFINITION : ATTRIBUT DE CLASSE

Un attribut de classe indique une caractéristique partagée par toutes les instances. Sa présence révèle souvent une modélisation à approfondir.

Lien entre classe et instance avec attribut de classe

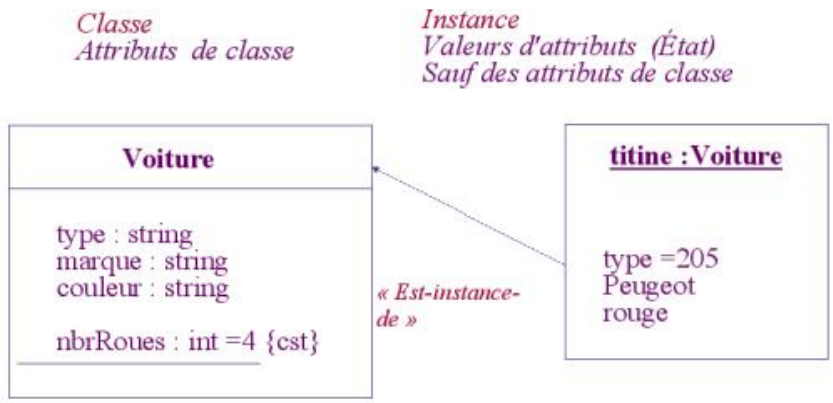


Figure 84 : Lien entre classe et instance avec attribut de classe

3.1.3 Attribut dérivé

DÉFINITION : ATTRIBUT DÉRIVÉ

Un attribut dérivé est un attribut qu'il est possible de calculer ou de déduire à partir des autres attributs et éventuellement d'éléments extérieurs à la classe. Il peut être révélateur d'une opération déguisée, et son stockage est optionnel.

Attribut dérivé

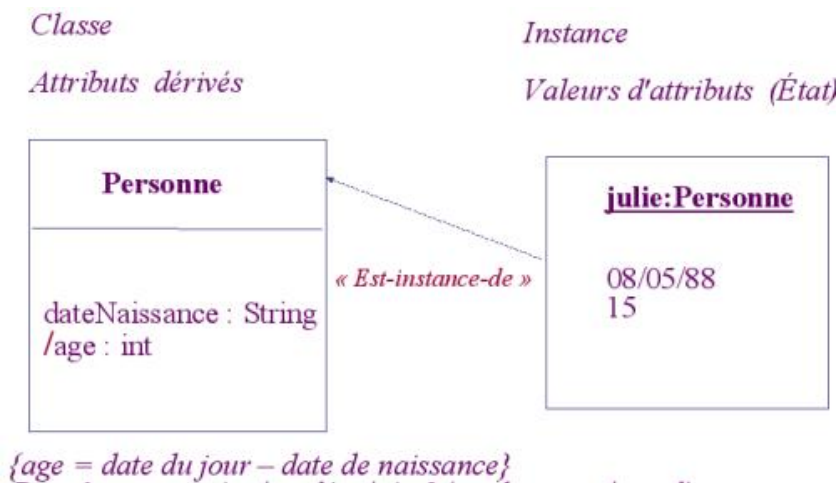


Figure 85 : Attribut dérivé

3.2 Opérations et méthodes

3.2.1 Opérations

DÉFINITION : OPÉRATION

*Les **opérations** déterminent le comportement de la classe. Elles sont instanciées par des méthodes dans les objets.*

Opérations

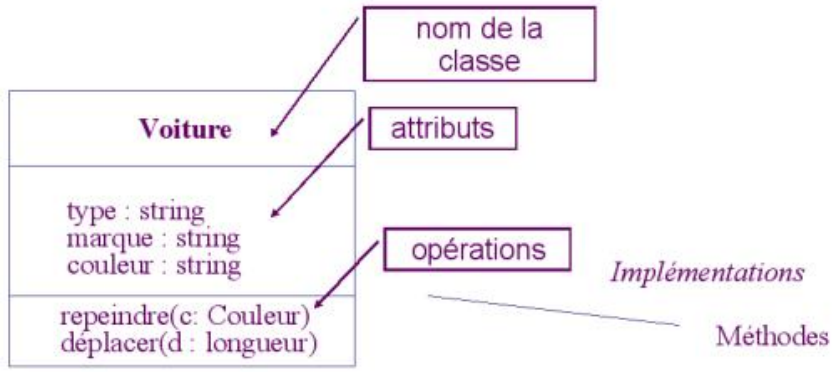


Figure 86 : Opérations

[Visibilité] nom [(paramètres)][:type retour][{ propriétés}]

Opérations

Figure 87 : Opérations

3.2.2 Opérations et méthodes de classe

DÉFINITION : OPÉRATION DE CLASSE

L'opération de classe est une opération qui ne s'applique pas à une instance, mais a la classe toute entiere.

Dans l'exemple ci-dessous, l'opération de classe apparaît en souligné.

Opération de classe

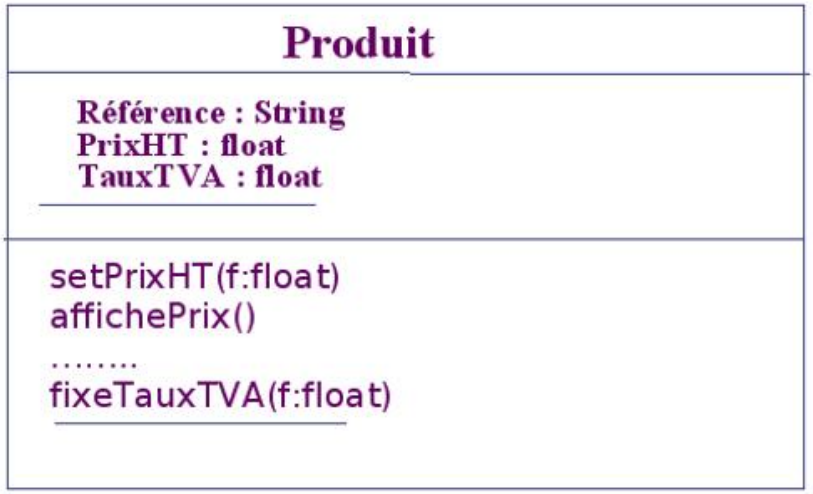


Figure 88 : Opération de classe

3.3 Classe, résumé

Un objet est instance (propre) d'une classe :

- il se conforme à la description que celle-ci fournit,
- il admet une valeur pour chaque attribut déclaré à son attention dans la classe,
- il est possible de lui appliquer toute opération définie à son attention dans la classe.

Tout objet admet une identité qui le distingue pleinement des autres objets : il peut être nommé et être référencé par un nom (mais son identité ne se limite pas à ça).

4 Les associations et liens

De la même façon que la classe représente un niveau d'abstraction par rapport à l' objet , on peut définir deux niveau d'abstraction pour désigner les relations entre les objets : L' association est une relation entre deux classes, tandis que le lien est une relation entre deux instances.

Relations entre classes (association) et objets (lien)

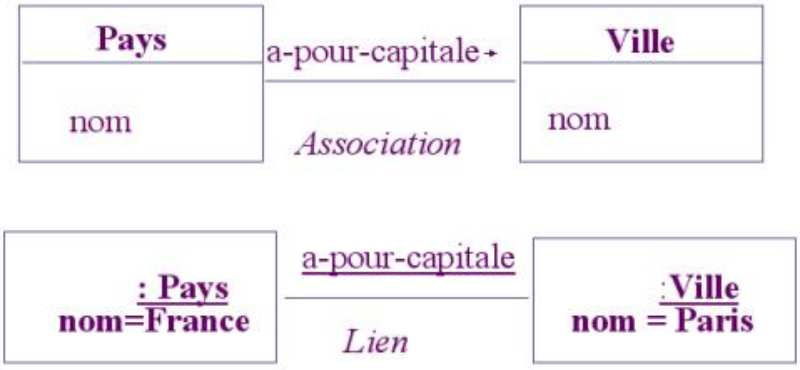


Figure 89 : Relations entre classes (association) et objets (lien)

4.1 Association

DÉFINITION : ASSOCIATION

Une association est l'abstraction d'un groupe de liens dont les caractéristiques sont communes

- *même type d'origine*
- *même type de destination*
- *même attributs*

Une association est en général binaire (degré = 2) mais dans certains cas le degré ternaire est pertinent

Exemple d'association ternaire

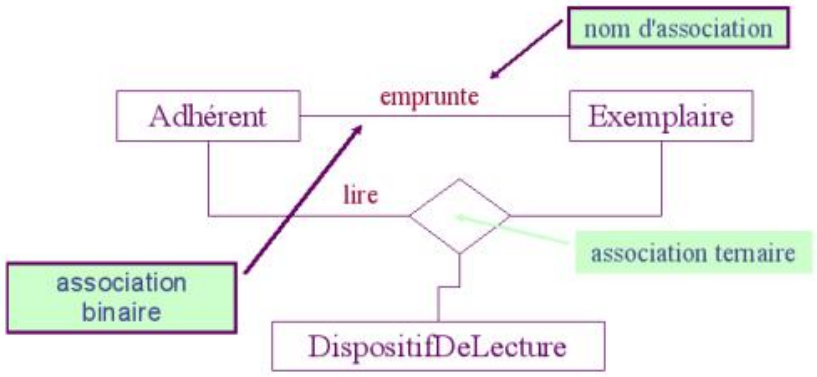


Figure 90 : Exemple d'association ternaire

4.2 Multiplicités et rôles dans une association

Les rôles et multiplicités d'une association

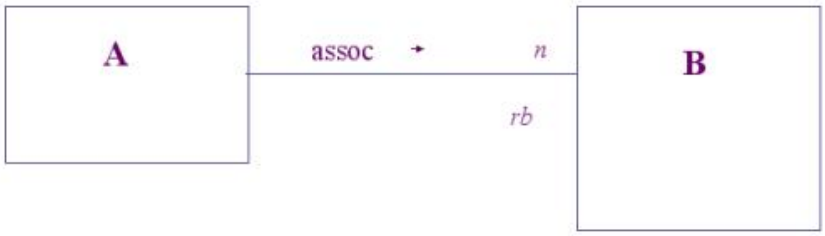


Figure 91 : Les rôles et multiplicités d'une association

Si A et B sont deux classes associées avec une multiplicité n et le rôle rb du côté de B :

- n instances de B peuvent être en relation avec une instance fixée de A
- une instance de B joue le rôle rb pour une instance de A dans le contexte de assoc

Exemple de rôles et multiplicités

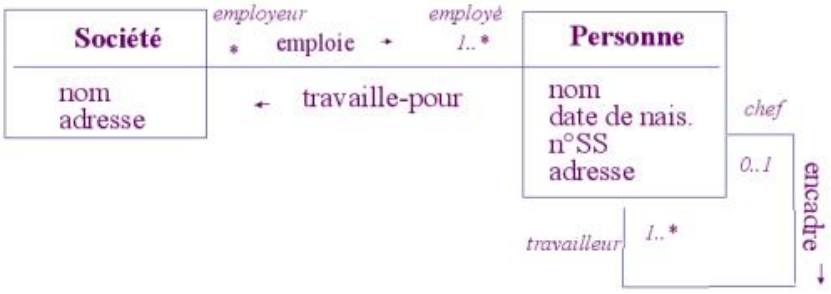


Figure 92 : Exemple de rôles et multiplicités

Les multiplicités

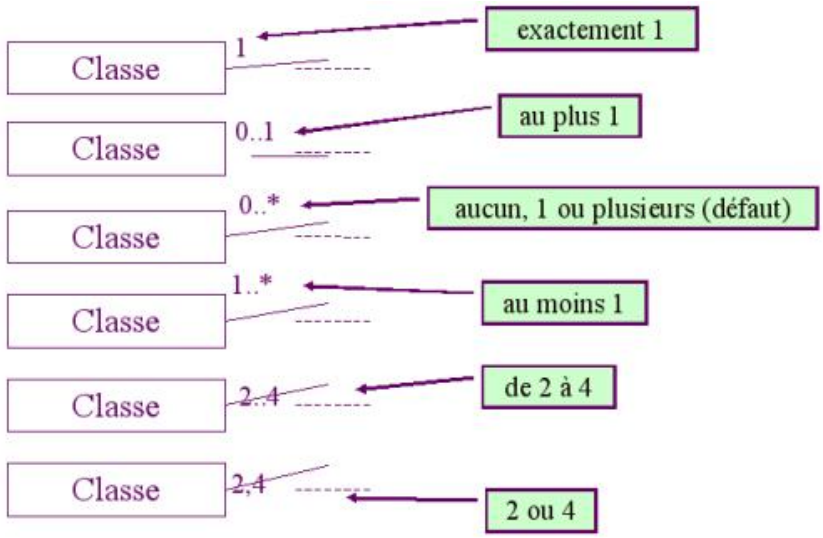


Figure 93 : Les multiplicités

Conseils pour la modélisation d'association, lorsque l'on doit donner des noms :

- lister les verbes candidats possibles
- ne pas "dériver vers la conception" (pointeurs ou attributs référentiels)

Attribut référentiel

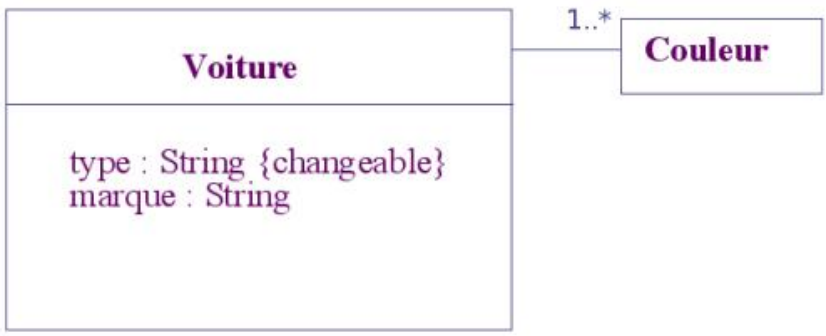


Figure 94 : Attribut référentiel

Un attribut à valeur multiple est souvent référentiel

4.3 Classe association

DÉFINITION : CLASSE ASSOCIATION

*Une **classe association** permet de donner des attributs à une association entre deux classes.*

Classe association

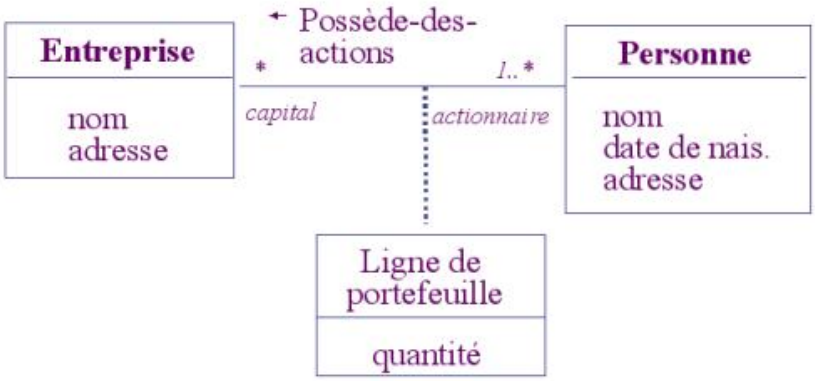


Figure 95 : Classe association

Classe association avec association vers une autre classe

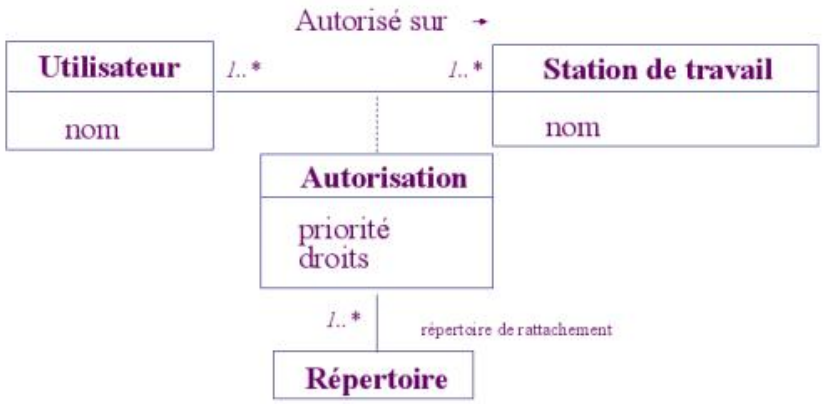


Figure 96 : Classe association avec association vers une autre classe

Une classe d'association permet de modéliser une association par une classe, donc de disposer d'attributs et d'opérations spécifiques.
Les liens d'une telle association sont alors des objets instances de cette classe. À ce titre, ils admettent une valeur pour tout attribut déclaré dans la classe d'association ; et on peut leur appliquer toute opération définie dans celle-ci.
En tant que classe, une classe d'association peut à son tour être associée à d'autres classes (voire à elle-même par une association réflexive).

4.4 Association qualifiée

DÉFINITION : ASSOCIATION QUALIFIÉE

*Une **association qualifiée** consiste en une association à laquelle on ajoute un identifiant caractérisant les relations entre les classes.*

Association qualifiée

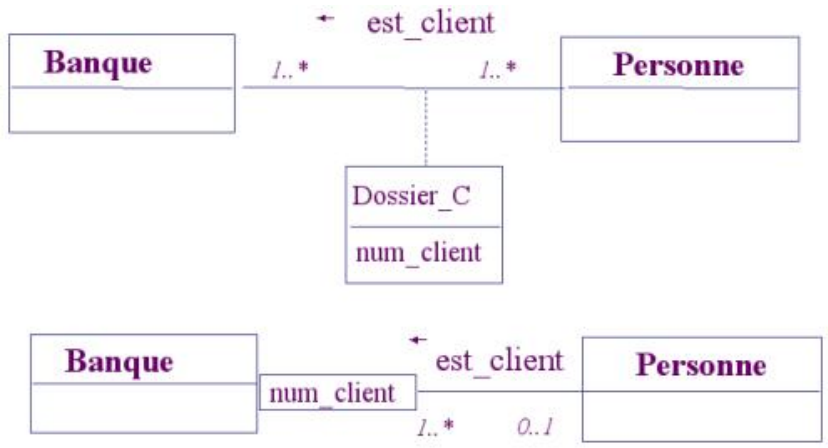


Figure 97 : Association qualifiée

5 D'autres types d'associations

Parmi les abstractions de relations qui n'entrent pas dans le cadre des associations vues ci-dessus, on trouve :

- Les associations particulières (compositions / agrégation)
- Les relations de Spécialisation / Généralisation

5.1 Agrégation et composition

5.1.1 Agrégation

DÉFINITION : AGRÉGATION

L'agrégation est une association dont la sémantique est celle du rapport collection/élément

Figure 98 : Agrégation

5.1.2 Composition

DÉFINITION : COMPOSITION

La composition est une association particulière tout /partie. L'existence du composant est assujettie à celle de l'objet

composite

Composition

Figure 99 : Composition

Autre exemple de composition

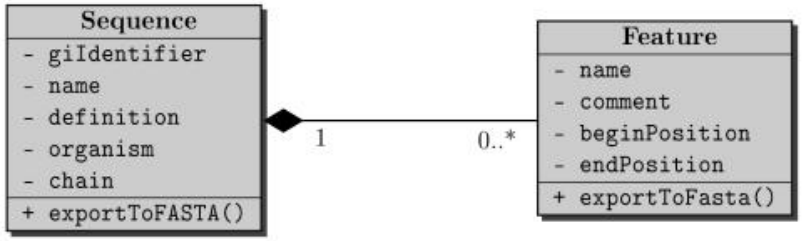


Figure 100 : Autre exemple de composition

Composition / Agrégation

- Contraintes
  - Exclusivité / Partage
  - Dépendance / Indépendance
- Propagation / Diffusion

5.2 Généralisation / Spécialisation

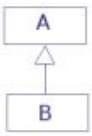
DÉFINITION : GÉNÉRALISATION / SPÉCIALISATION

La généralisation/spécialisation traduit les mécanismes d'inférences intellectuelles suivants :

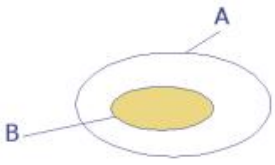
- *Soit on affine (spécialisation)*
- *Soit on abstrait (généralisation)*

Sa sémantique dépend du point de vue :

- Point de vue ensembliste
- Point de vue logique

Elle relie deux éléments de modèle (classes, cas d'utilisation, méthodes) 

L'extension de A (ses instances) contient l'extension de B



Conséquence : une instance de B possède les propriétés de A éventuellement sous une forme affinée

5.2.1 Généralisation / Spécialisation

Example de généralisation / spécialisation

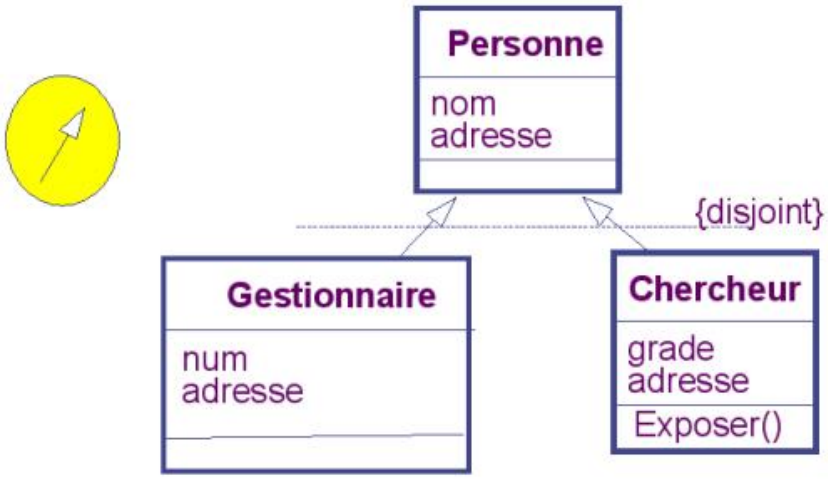


Figure 101 : Example de généralisation / spécialisation

Une sous-classe "hérite" des descriptions de sa super-classe :

- les déclarations d'attributs,
- les définitions d'opérations,
- les associations définies sur la super-classe,
- les contraintes (on en parle plus tard).

Une sous-classe peut redéfinir de façon plus spécialisée une partie ou la totalité de la description « héritée ».

5.2.2 Discriminant

DÉFINITION : DISCRIMINANT

Le discriminant exprime un critère de classification

Discriminant

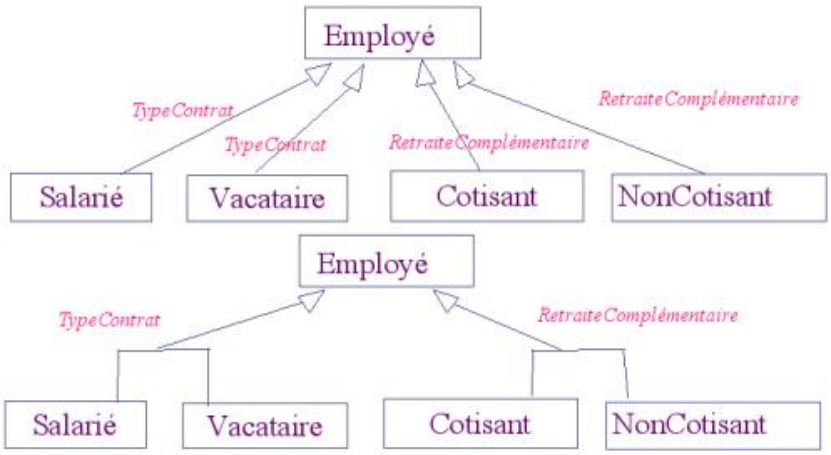


Figure 102 : Discriminant

5.2.3 Généralisation / Spécialisation multiple

Généralisation / Spécialisation multiples

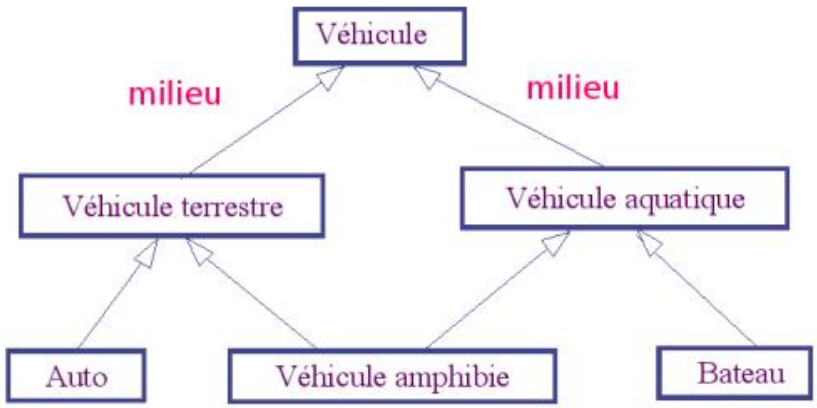


Figure 103 : Généralisation / Spécialisation multiples

5.2.4 Composition/Agrégation ou généralisation ?

Le choix entre ces deux notions découle directement du lien qui doit être mis en évidence

- Agrégation
  - lien entre instances
  - un arbre d'agrégation est composé d'objets qui sont parties d'un objet composite
- Généralisation
  - lien entre classes

6 Les contraintes

DÉFINITION : CONTRAINTE

Les contraintes sont des prédicats, pouvant porter sur plusieurs éléments du modèle statique, qui doivent être vérifiés à tout instant.

Les contraintes permettent de rendre compte de détails à un niveau de granularité très fin dans un diagramme de classe. Elles peuvent exprimer des conditions ou des restrictions. En UML, les contraintes sont exprimées sous forme textuelle, entre accolades et de préférence en OCL (Object Constraint Language).
Les contraintes sont héritées.

Contrainte de généralisation / spécialisation

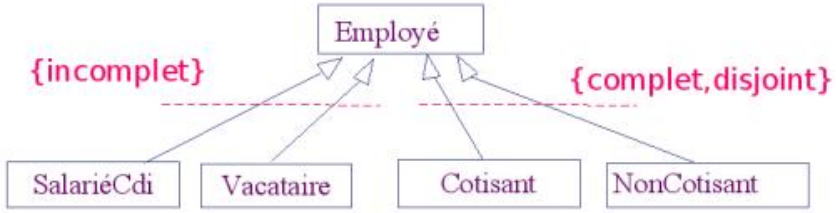


Figure 104 : Contrainte de généralisation / spécialisation

Contrainte de généralisation / spécialisation

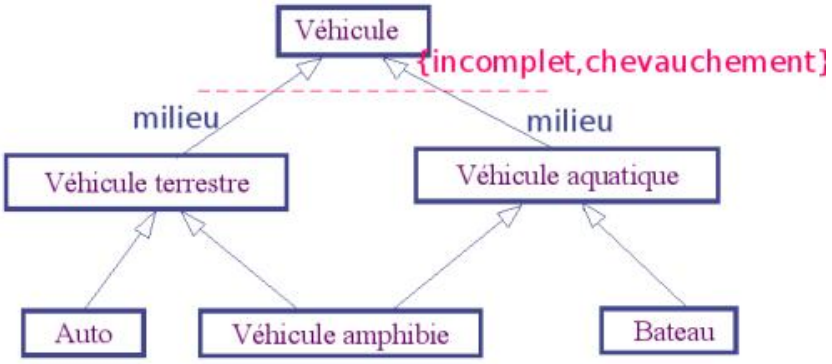


Figure 105 : Contrainte de généralisation / spécialisation

Contraintes sur associations

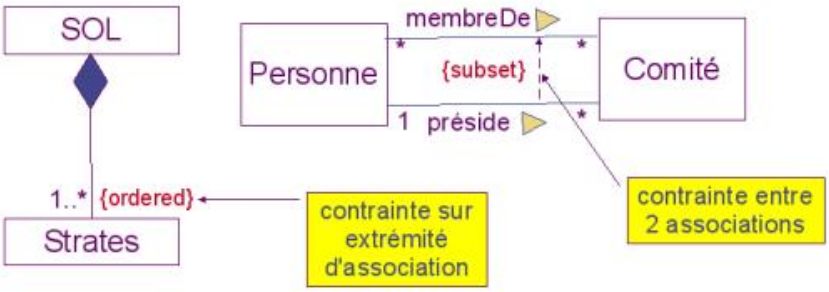


Figure 106 : Contraintes sur associations

Contraintes à différents niveaux

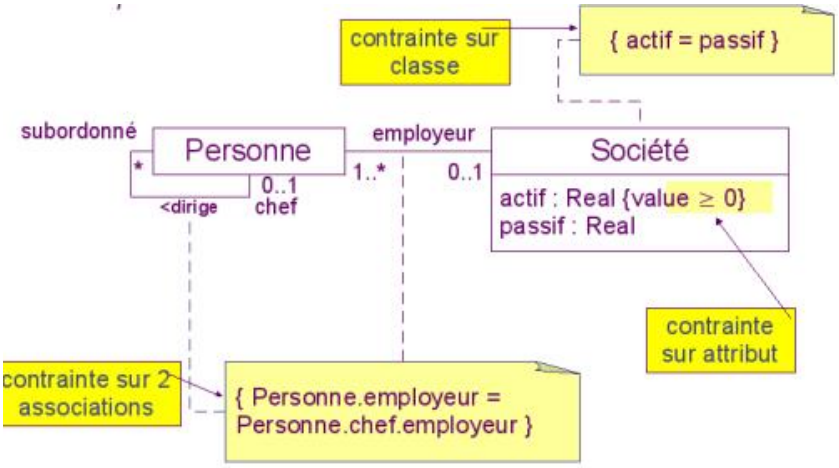


Figure 107 : Contraintes à différents niveaux

7 Quelques compléments de notation

7.1 Relation de dépendance

Sémantique de la relation de dépendance : « un changement dans la spécification de B peut affecter A » (appelle, crée, utilise, instance de, etc.)

7.2 Stéréotype

DÉFINITION : STÉRÉOTYPE

Un stéréotype est un label qui permet d'apporter une précision supplémentaire à un élément de notation (classe, relation, ...)

Stéréotype

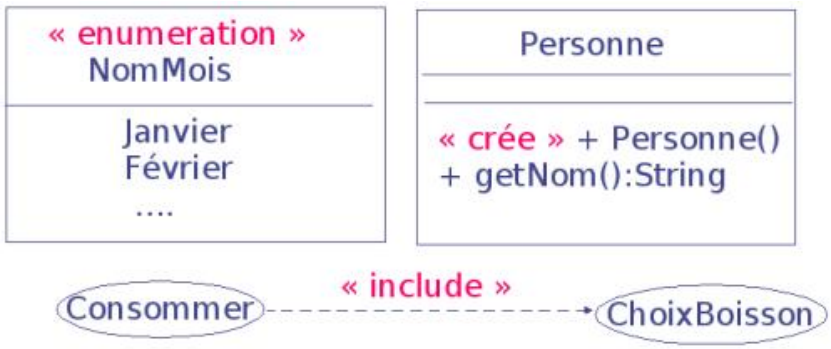


Figure 108 : Stéréotype

8 Classes et opérations abstraites

8.1 Classes abstraites

(notation italiques ou avec mot-clef {abstract})

DÉFINITION : CLASSE ABSTRAITE

Une classe abstraite est une classe non instanciable, c'est à dire qu'elle n'admet pas d'instances directes. Une classe abstraite est une description d'objets destinée à être « héritée » par des classes plus spécialisées.

Pour être utile, une classe abstraite doit admettre des classes descendantes concrètes. La factorisation optimale des propriétés communes à plusieurs classes par généralisation nécessite le plus souvent l'utilisation de classes abstraites.

8.2 Opérations abstraites

DÉFINITION : OPÉRATION ABSTRAITE

Une opération abstraite est une opération n'admettant pas d'implémentation : au niveau de la classe dans laquelle elle est déclarée, on ne peut pas dire comment la réaliser.

Les opérations abstraites sont particulièrement utiles pour mettre en oeuvre le polymorphisme. Toute classe concrète sous-classe d'une classe abstraite doit "concrétiser" toutes les opérations abstraites de cette dernière.

8.3 Exemple

Classes et opérations abstraites

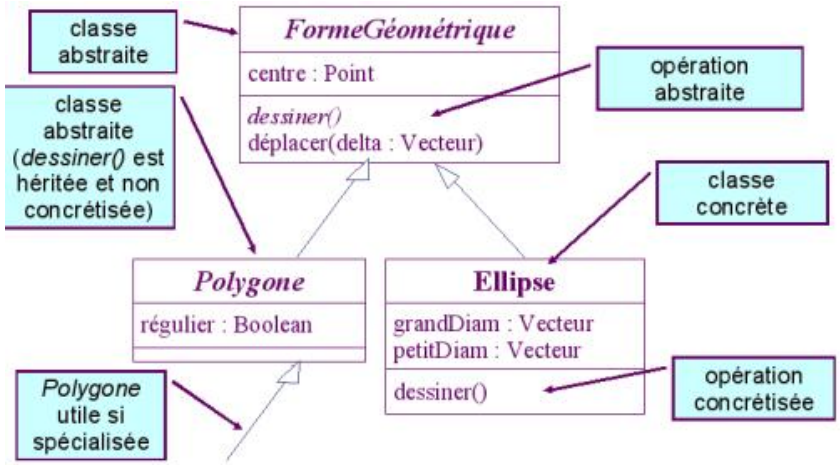


Figure 109 : Classes et opérations abstraites

9 Interfaces

DÉFINITION : INTERFACE

Une interface est une collection d'opérations utilisée pour spécifier un service offert par une classe. Une interface peut être vue comme une classe sans attributs et dont toutes les opérations sont abstraites.

Une interface est destinée à être "réalisée" par une classe (celle-ci en hérite toutes les descriptions et concrétise les opérations abstraites). Une interface peut en spécialiser une autre, et intervenir dans des associations avec d'autres interfaces et d'autres classes.

Interface

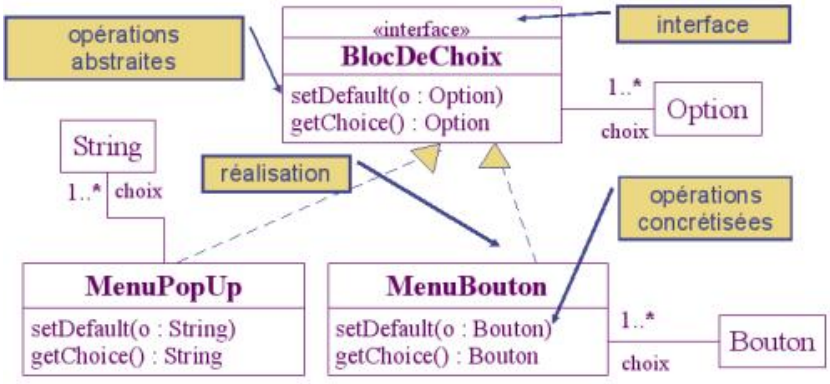


Figure 110 : Interface

Deux notations pour l'utilisation d'une interface

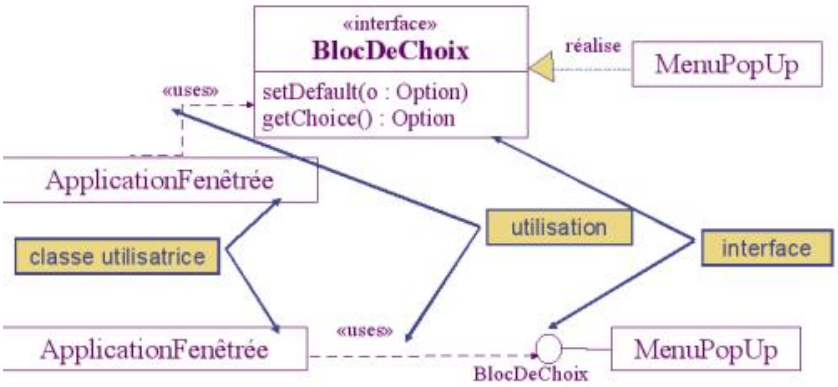


Figure 111 : Deux notations pour l'utilisation d'une interface

10 Conclusion

Heuristiques d'élaboration du modèle structurel

- Bien comprendre le problème
- Faire simple
- Bien choisir les noms
- Bien expliciter les associations
- Ne pas trop "généraliser"
- Relire
- Documenter

De nombreuses révisions sont nécessaires !

Modèle dynamique

1 Introduction

DÉFINITION : MODÈLE DYNAMIQUE

Un modèle dynamique décrit les interactions entre objets et les changements au cours du temps :

- *Le déroulement des actions, le contrôle*
- *Les états des objets et leurs interactions*
- *La survenue des événements*

Les différents types de diagrammes qui interviennent sont :

- les diagrammes de collaboration (ou de communication)
- les diagrammes de séquences
- les diagrammes états-transitions
- les diagrammes d'activités

1.1 La communication

Les systèmes informatiques peuvent être vus comme des sociétés d'objets travaillant en synergie pour réaliser les fonctions de l'application

Communication

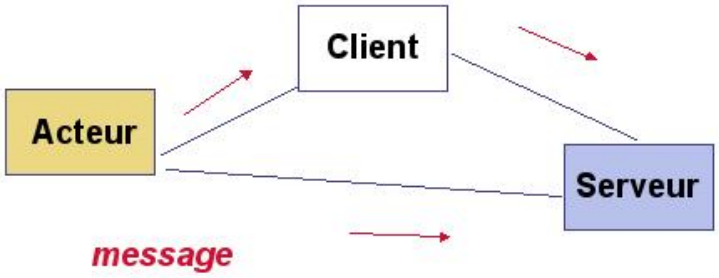


Figure 112 : Communication

1.2 Les messages

Les types de messages :

- constructeurs
- destructeurs
- appel de méthodes

Synchronisation :

Expressions
[condition]
*[condition d'itération] itération

2 Diagramme de collaboration

DÉFINITION : DIAGRAMME DE COLLABORATION

Un diagramme de collaboration exprime l'interaction entre objets pour la réalisation d'une fonctionnalité du système mais l'accent est mis sur la collaboration entre objets

Diagramme de collaboration

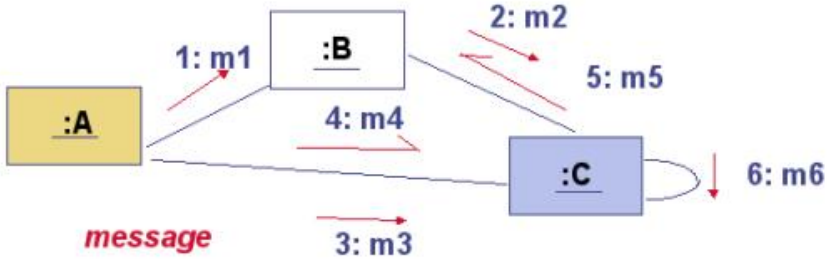


Figure 113 : Diagramme de collaboration

Exemple : On part du diagramme d'instance d'une roue de brouette, avec comme diagramme de classe un diagramme très simple avec une seule classe pièce :

Diagramme d'instance d'une roue de brouette

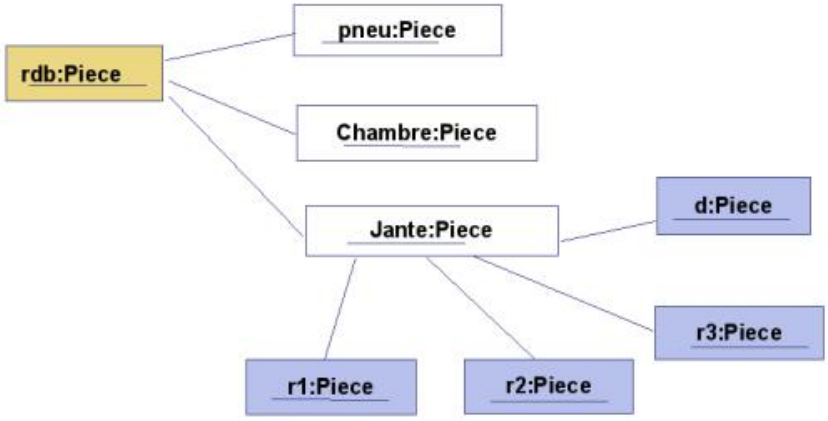


Figure 114 : Diagramme d'instance d'une roue de brouette

On en déduit le diagramme de collaboration pour le calcul du prix

Diagramme de collaboration pour le calcul du prix

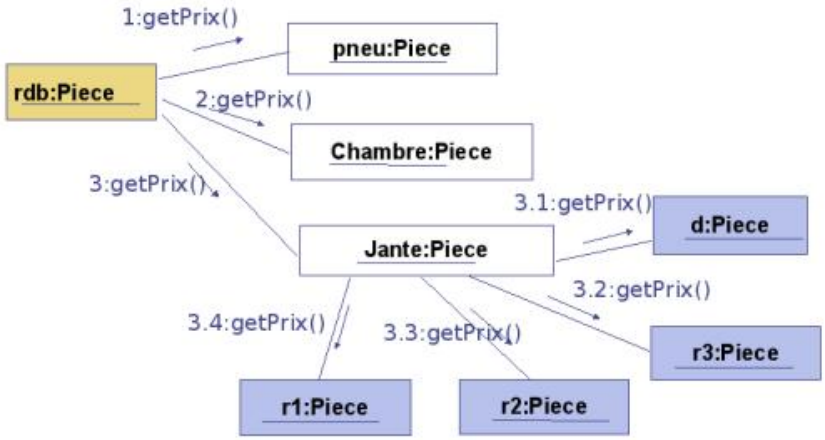


Figure 115 : Diagramme de collaboration pour le calcul du prix

3 Diagramme de séquences

DÉFINITION : DIAGRAMME DE SÉQUENCE

Un diagramme de séquence exprime également l'interaction entre objets pour la réalisation d'une fonctionnalité, mais l'accent est mis sur la chronologie des événements du système

Diagramme de séquence

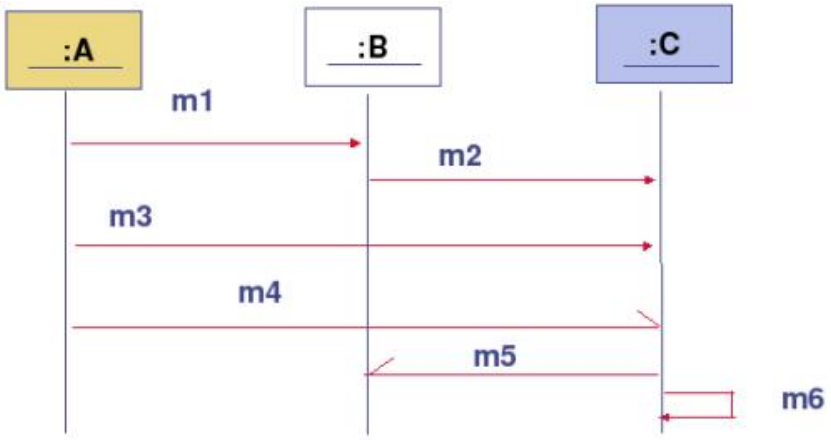


Figure 116 : Diagramme de séquence

3.1 La ligne de vie

DÉFINITION : LIGNE DE VIE

Il s'agit de la ligne figurant la "vie" d'un objet, entre sa création et sa destruction.

Ligne de vie

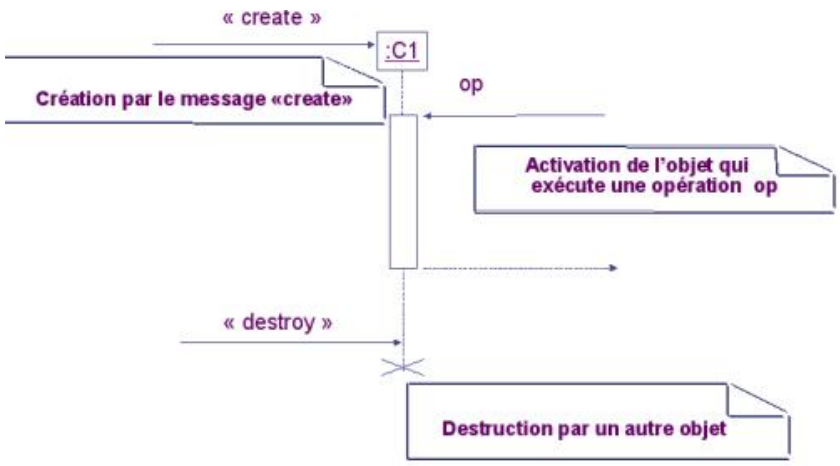


Figure 117 : Ligne de vie

Exemple de diagramme de séquence pour la roue de brouette

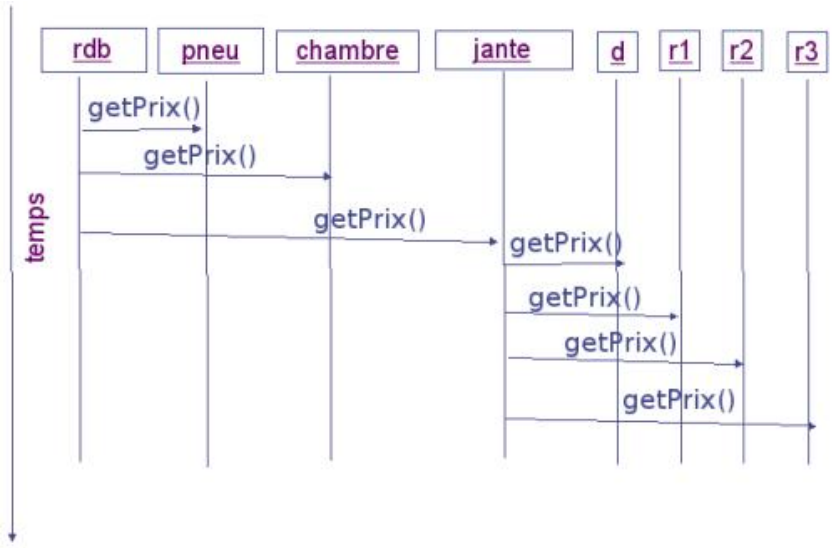


Figure 118 : Exemple de diagramme de séquence pour la roue de brouette

Exemple de diagramme de séquence pour la roue de brouette

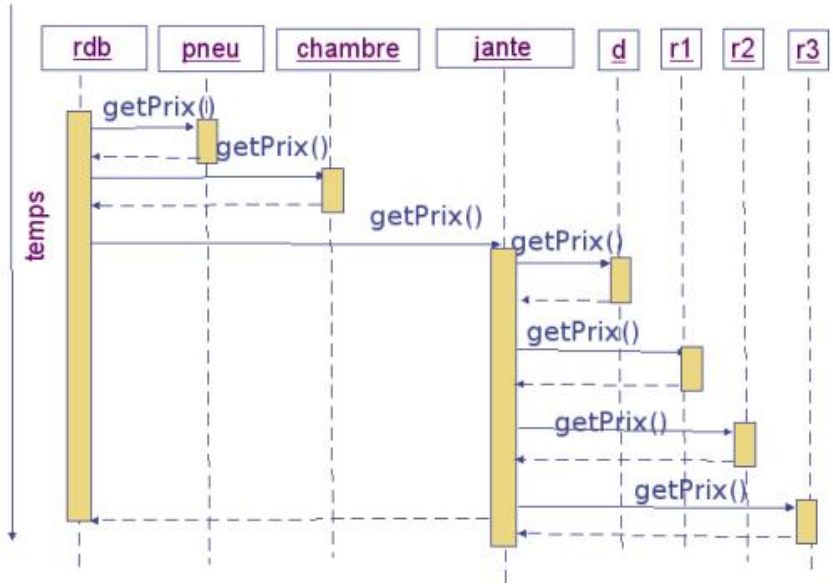


Figure 119 : Exemple de diagramme de séquence pour la roue de brouette

Exemple de diagramme de séquence pour le bananier

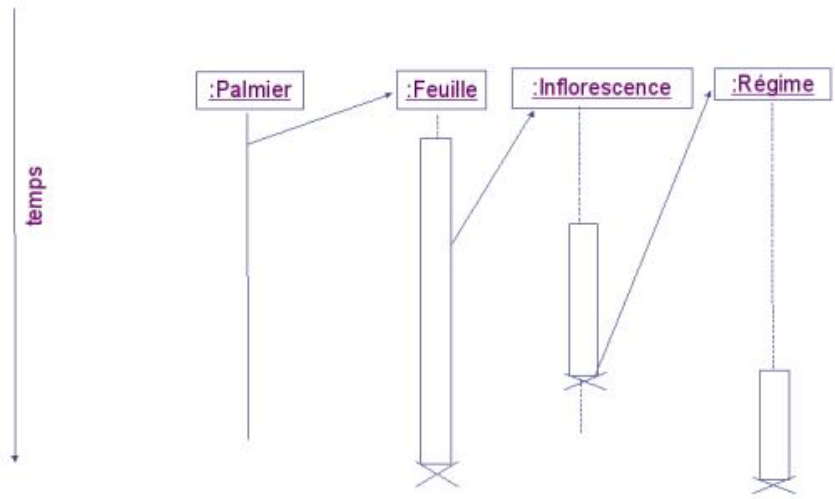


Figure 120 : Exemple de diagramme de séquence pour le bananier

4 Diagrammes d'états

4.1 État d'un objet

DÉFINITION : ÉTAT

C'est le moment de la vie d'un objet où

- *Il accomplit une action*
- *Il satisfait une condition*
- *Il attend un événement L'état est défini par la valeur instantanée des attributs et liens de l'objet*

4.2 Événement

DÉFINITION : ÉVÉNEMENT

Il s'agit du stimulus (sans durée) envoyé à un objet par exemple :

- *une condition devient vraie*
- *appel d'une opération*
- *réception d'un signal*
- *fin d'une période de temps*

4.3 Diagrammes d'états

DÉFINITION : DIAGRAMME D'ÉTAT

Ils servent à représenter les états par lesquels passe un objet d'une classe donnée. Il s'agit de graphes dont les noeuds sont les états, et les arcs des transitions nommées par un événement. Une séquence d'événements représente un chemin dans le graphe.

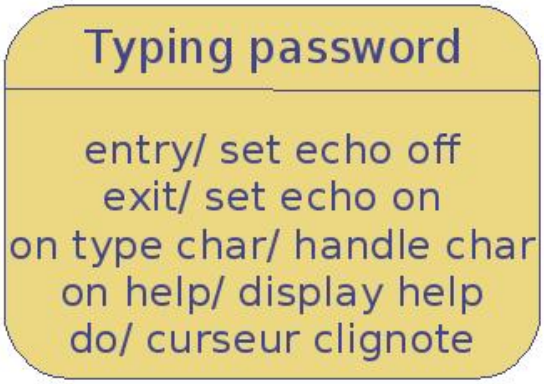
Les états et les événements sont duaux : un événement sépare deux états, un état sépare deux événements

4.4 Notation des états :

Initial / Final

Simple

Complexe



Les activités internes sont les actions qui se produisent au début, à la fin, lors d'événements, ou tout le temps

4.5 Notation des transitions

étiquette

- événement(paramètres)
- [condition]
- /action
- ^envoiMessage

4.6 Opération

Activité / Action

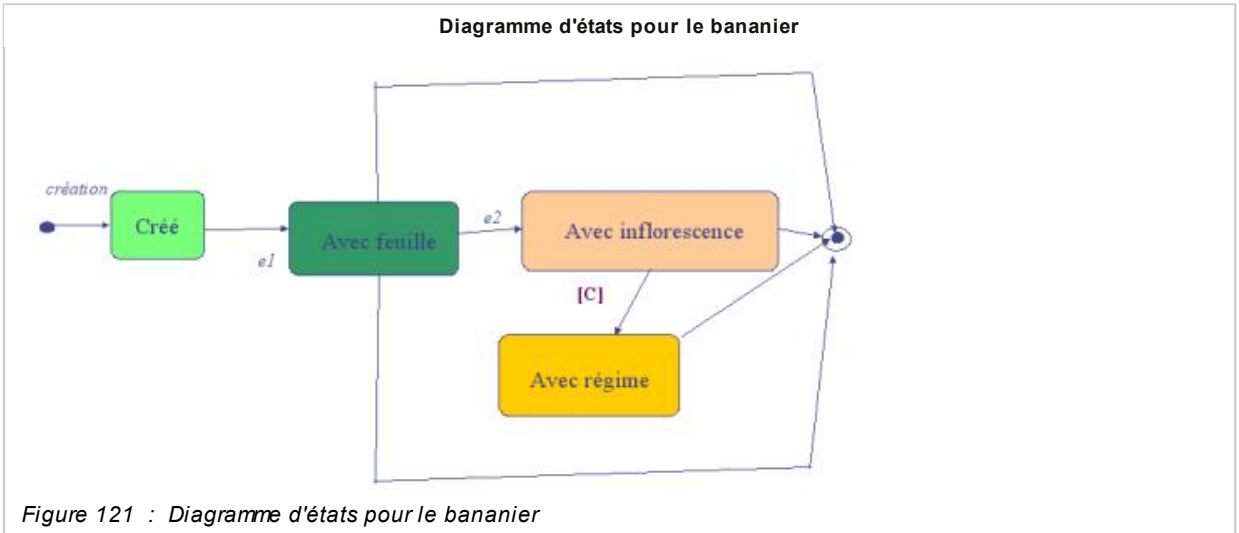


Figure 121 : Diagramme d'états pour le bananier

DÉFINITION : OPÉRATION

Elle peut être attachée à une transition ou à un état. Elle est exécutée en réponse à l'événement ou à l'état.

DÉFINITION : ACTION

C'est une opération instantanée, non interruptible, souvent utilisée pour faire des mises à jour de valeurs, attachée à une transition. Envoyer un événement est une action

DÉFINITION : ACTIVITÉ

C'est une opération qui prend du temps, interruptible par un événement, perpétuelle ou finie, nécessairement attachée à un état.

4.7 Sous-états

4.7.1 Transitions gardées

DÉFINITION : TRANSITION GARDÉE

Il s'agit des transitions internes à un état, entre ses différents sous-états.

Diagramme d'états d'un compte bancaire

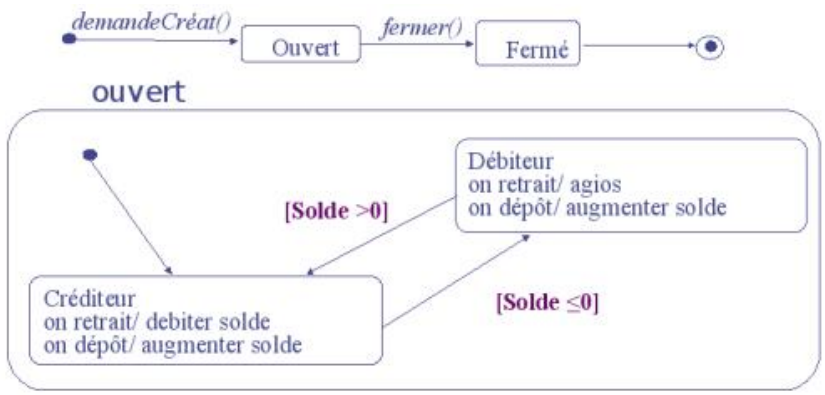


Figure 122 : Diagramme d'états d'un compte bancaire

4.7.2 Généralisation

La généralisation permet une meilleure structuration des diagrammes d'états Un objet dans un état du diagramme général doit être dans un des états du diagramme imbriqué (relation **ou** entre les états)

Généralisation

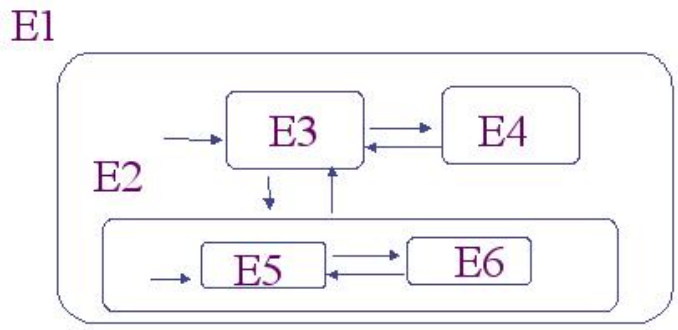


Figure 123 : Généralisation

4.7.3 Agrégation

Une classe "agrégat" aura un état défini par l'agrégation des états de ses composants. Agrégation concurrente (relation **et**)

Agrégation

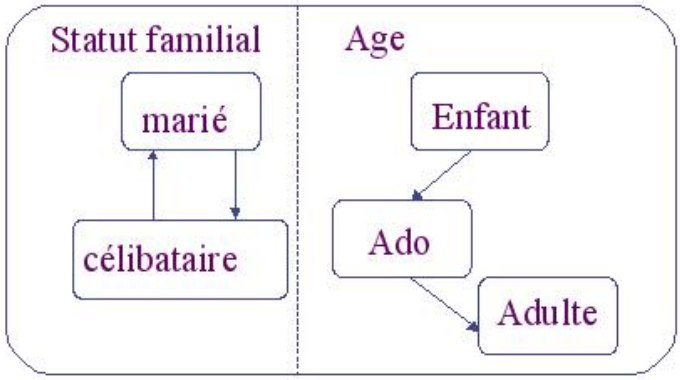


Figure 124 : Agrégation

5 Diagrammes d'activités

DÉFINITION : DIAGRAMME D'ACTIVITÉ

C'est une représentation du flot de contrôle

Flot de contrôle

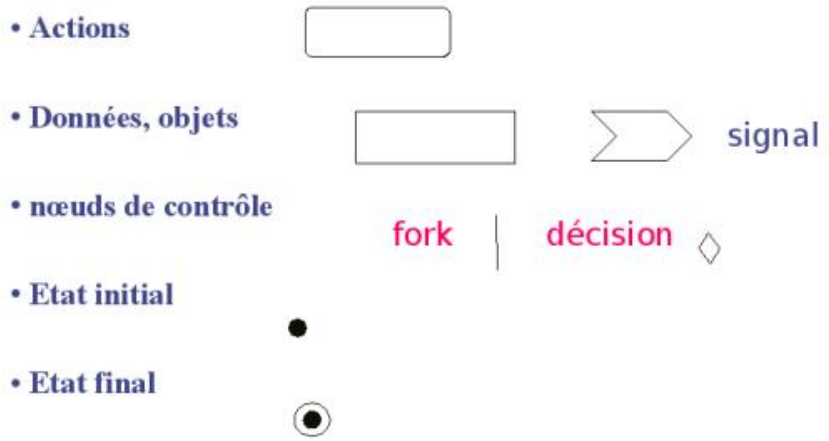


Figure 125 : Flot de contrôle

Diagramme d'activités d'un bon de commande

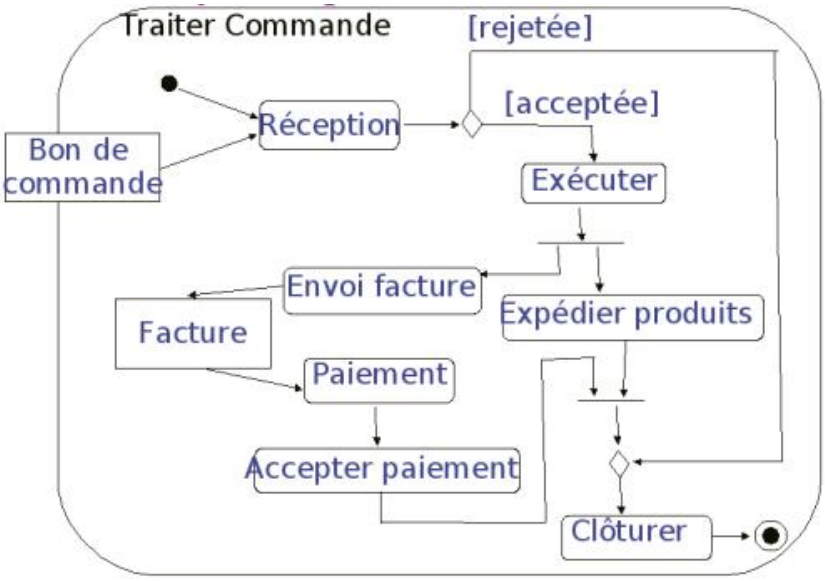


Figure 126 : Diagramme d'activités d'un bon de commande

6 Heuristiques d'élaboration du modèle dynamique

- Chemins suivis lors des envois de messages : diagrammes de séquences ou de collaboration
- Point de vue d'un objet
  - diagramme d'états : pour les objets pour lesquels il est significatif de montrer les changements d'états
  - diagramme d'activités : pour décrire un algorithme du point de vue d'un objet
- Contrôler la cohérence, structurer les diagrammes
- Accompagnement des diagrammes de cas d'utilisation
  - diag. de séquences
  - diag. d'activités (proches des workflows)

Modèle d'implémentation

1 Introduction

DÉFINITION : MODÈLE D'IMPLÉMENTATION

Un modèle d'implémentation est une série de diagramme indiquant comment le projet se concrétise au niveau des moyens physiques et informatiques.

2 Les packages

DÉFINITION : PACKAGE

Un package ou sous-système est un regroupement logique de classes, associations, contraintes... Un sous-système est généralement défini par les services qu'il rend. Les liens entre sous-systèmes sont des liens de dépendance.

Les « packages » servent à structurer une application. Ils sont utilisés dans certains langage de programmation objet (comme par exemple Java), ce qui assure une bonne « tracabilité » de l'analyse à l'implémentation.

Packages

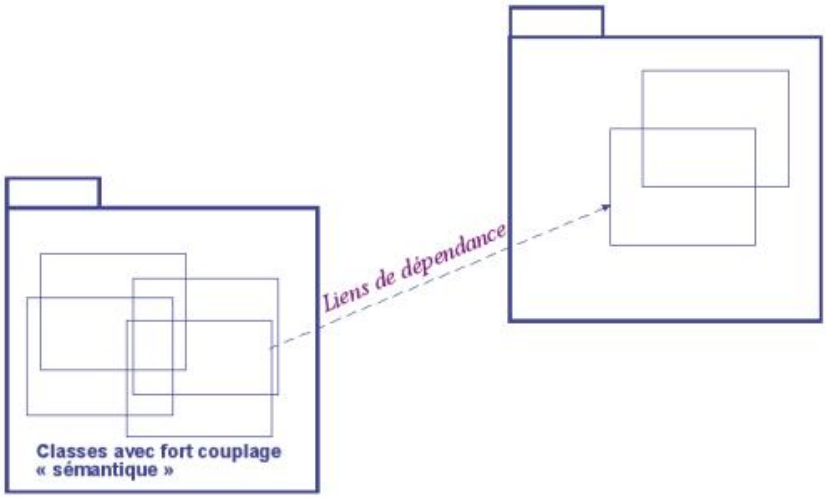


Figure 127 : Packages

3 Diagramme de composants

DÉFINITION : DIAGRAMME DE COMPOSANT

Un diagramme de composant met en évidence les dépendances entre composants logiciels (sources, binaires, exécutables, etc.)

Diagramme de composants

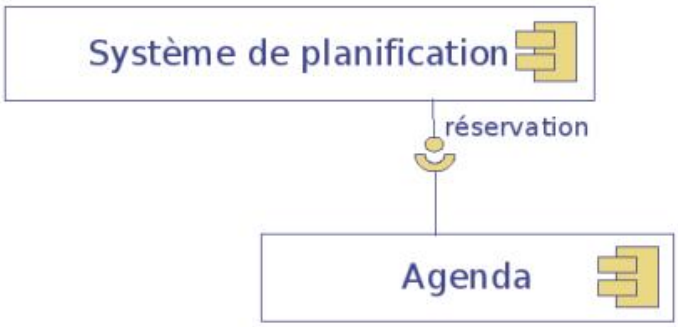


Figure 128 : Diagramme de composants

4 Diagramme de déploiement

DÉFINITION : DIAGRAMME DE DÉPLOIEMENT

Un diagramme de déploiement dépeint l'organisation matérielle des éléments de calcul et des composants logiciels

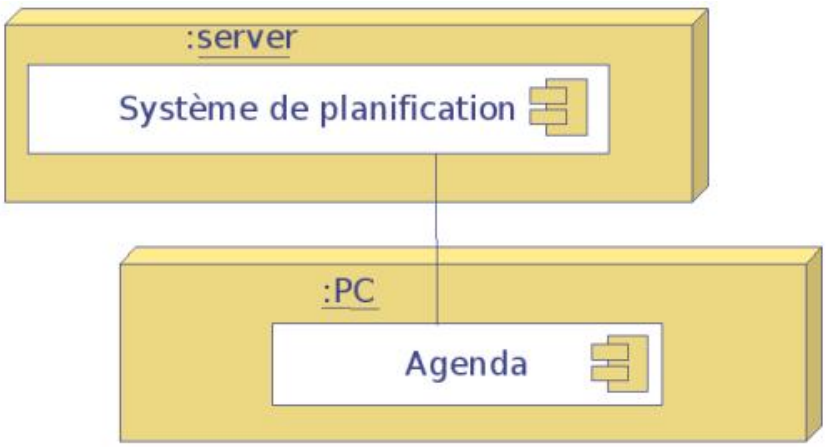


Figure 129 : Diagramme de déploiement

Projection vers les bases de données

1 Introduction

1.1 Adéquation aux BD « objet »

Persistance

- Classes vs Types / persistants vs éphémères
- Points d'entrée (racine de persistance)
- Regroupements

LDD, LMD, L Hôte

- LDD, LMD, L Hôte équivalents LPO
- Langage de requête « like » SQL
- Transactions

1.2 Adéquation aux BD « classiques »

On effectue une traduction des diagrammes :

- diagrammes structurels --> schémas
- diagrammes dynamiques --> requêtes et traitements applicatifs divers<

Mais il faut suivre quelques règles de « passage »

2 Règles de passage du modèle objet aux bases de données relationnelles

2.1 Classes

2.1.1 Cas simple

On traduit les attributs en colonnes, et on ajoute un identifiant.
Exemple

Classe :

Schema relationnel :

Attribut	Domaine	Non Null
Id_adresse	Identifiant	Oui
N°	Entier	Non
Rue	String(30)	Non
.....		

2.1.2 Lorsqu'il y a un attribut unique

On ne rajoute pas d'identifiant, c'est l'attribut unique qui sert d'identifiant
Exemple

Classe :

Schema relationnel :

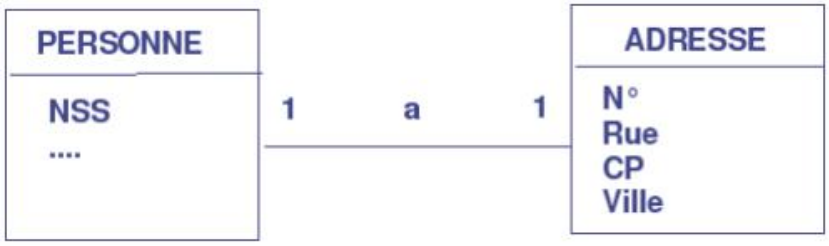
Attribut	Domaine	Non Null
Id_Pers	Identifiant	Oui
NSS	String(13)	Oui
Prénom	String(30)	Non
Date-nais	Date	Non

2.2 Associations

2.2.1 Cas d'une cardinalité 1 1

On suppose que les classes participant à l' association sont déjà traduites en relation ayant chacune leur clé primaire. On rajoute simplement dans la relation A une clé étrangère correspondant à la clé primaire de la relation B.

Association :



Schema relationnel :

Attribut	Domaine	Non Null
NSS	String(13) ID	Oui
Nom	String(35)	Oui
Id_adresse	Identifiant	Oui

2.2.2 Cas d'une cardinalité 1 *

Même chose, mais en prenant bien soin que la clé étrangère soit rajoutée dans la relation traduisant la classe est du côté "*". Dans l'exemple ci-dessous, on a bien un seul possesseur possible pour un compte donné, NSS est donc bien une clé étrangère.

Association :

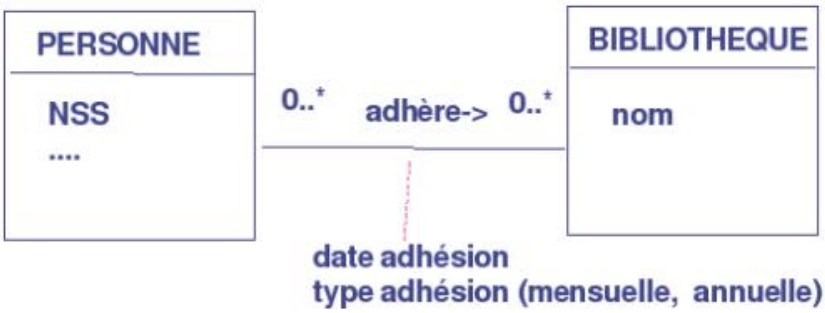
Schema relationnel :

Attribut	Domaine	Non Null
Id_compte	Identifiant	Oui
N°	String(35) Id	Oui
NSS	Identifiant	Oui

2.2.3 Cas d'une cardinalité * *

Dans ce cas, il faut rajouter une relation spécifique à l'association, dans laquelle figurent au moins deux colonnes correspondant aux clés primaires de deux relations traduites à partir des classes en association. La clé primaire de la relation association est alors le couple des clés des deux tables. On peut rajouter également des colonnes pour chaque attribut appartenant à la relation, comme la date et le type dans l'exemple ci-dessous.

Association :

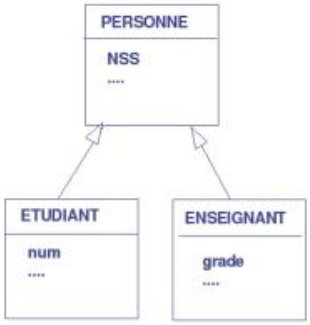


Schema relationnel :

Attribut	Domaine	Non Null
NSS	Id String(13)	Oui
Id_bibliothèque	Identifiant	Oui
Date_adhésion	Date	Oui
Type_adhésion	String(10)	Oui

2.3 Spécialisation

Spécialisation :



Plusieurs choix sont possibles, dépendamment de ce que l'on souhaite mettre en avant

2.3.1 Aplatir vers le haut

On rajoute dans la relation générale autant de colonnes qu'il faut pour tenir compte des attributs des différentes spécialisations. Ces champs sont alors vides pour certaines entrées. L'avantage, c'est que c'est plus simple à gérer lorsque la spécialisation admet des intersections (un objet appartient à plusieurs catégories en même temps). Le désavantage, c'est que cela prend plus de place, et que les traitements séparés des différentes spécialisations sont plus fastidieux.

2.3.2 Aplatir vers le bas

On rajoute dans chaque relation spécialisée tous les attributs communs de la classe générale. L'avantage est que les traitements séparés sont optimisés, mais l'inconvénient est que l'on risque une redondance des données lorsqu'il y a intersection.

2.3.3 Conserver les niveaux

C'est la solution intermédiaire, quand aucun des précédents choix ne s'impose. Les relations spécialisées ont alors pour clé étrangère l'identifiant de la relation générale. Cela peut complexifier les requêtes, mais colle au plus près à la modélisation.

Projection vers la programmation

1 Introduction

Toutes les indications portées sur le modèle ont une incidence sur la façon dont on va réaliser le projet. Le diagramme de classe est là pour donner une bonne indication sur la structure du code en langage objet qui va réaliser le projet.

Informations portées par une classe dans un diagramme

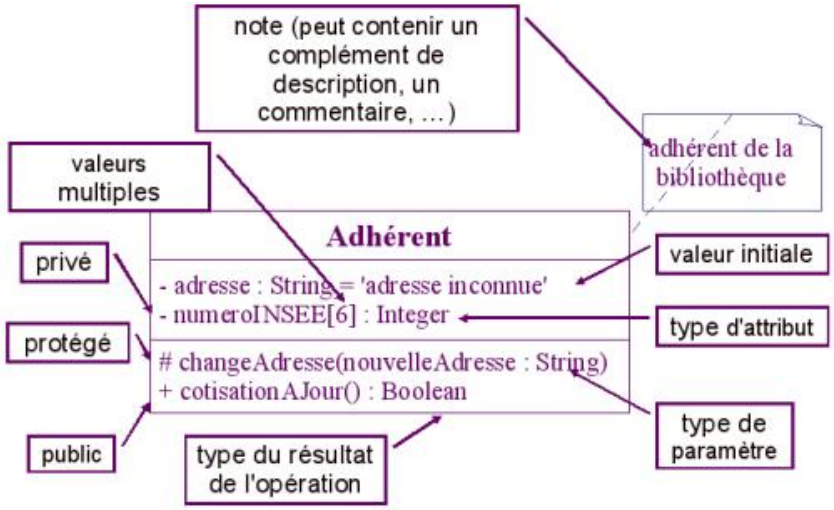


Figure 130 : Informations portées par une classe dans un diagramme

2 Classes

Elles sont destinées à proposer un service et réagir aux messages. Une classe du modèle deviendra une classe dans la programmation, avec les attributs qui deviennent des champs et les opérations qui deviennent des méthodes. Des méthodes particulières spécifiques à certains langages viennent se rajouter :

- Accesseurs L/E (convention get/set en Java)
- Constructeurs (Java, C++),
- Destructeurs (C++)

2.1 Exemple de la classe adhérent

Classe adhérent en Java

```
public class Adherent
{
 private String adr = ``adresse inconnue``;
 private int[] numeroInsee;

  public Adherent(){numeroInsee=new int[6];}
  public Adherent(String na, int[] ni){adr=na;
   numeroInsee=ni;}

  public String getAdr(){return adr;}
  protected void setAdr(String nouvelleAdresse)
                  {adr=nouvelleAdresse;}
  public int[] getNumeroInsee(){return numeroInsee;}
  protected void setNumeroInsee(int[] ni){numeroInsee=ni;}

  public boolean cotisationAjour(){.......}
}
```

Champs

Constructeurs

Accesseurs

Méthodes

Figure 131 : Classe adhérent en Java

Classe adhérent en C++ (extrait)

```
//..... Adherent.h .....................................................
class Adherent
{
 private:
    string adr;
    int* numeroInsee;
 public:
    Adherent();
    Adherent(string na, int* ni);
    virtual ~Adherent();
    virtual string getAdr() const;
    virtual void setAdr(string nouvelleAdresse);
    virtual int* getNumeroInsee()const;
    virtual void setNumeroInsee(int* ni);
    virtual boolean cotisationAjour()const;
};
```

Champs

Constructeurs

Destructeurs

Accesseurs

Méthodes

Figure 132 : Classe adhérent en C++ (extrait)

Classe adhérent en C++ (extrait)

```
//..... Adherent.cc
   .....
Adherent::Adherent()
        {adr = ``adresse inconnue``; numeroInsee=new
   int[6];}
Adherent::Adherent(string na, int* ni)
        {adr=na; numeroInsee=ni;}
Adherent::~Adherent(){delete[] numeroInsee;}
string Adherent:: getAdr()const{return adr;}
void Adherent::setAdr(string nouvelleAdresse)
         {adr=nouvelleAdresse;}
int* Adherent::getNumeroInsee()const{return
   numeroInsee;}
void Adherent::setNumeroInsee(int* ni)
   {numeroInsee=ni;}
```

Figure 133 : Classe adhérent en C++ (extrait)

2.2 Retour sur la notion d'encapsulation

Point de vue de l'objet

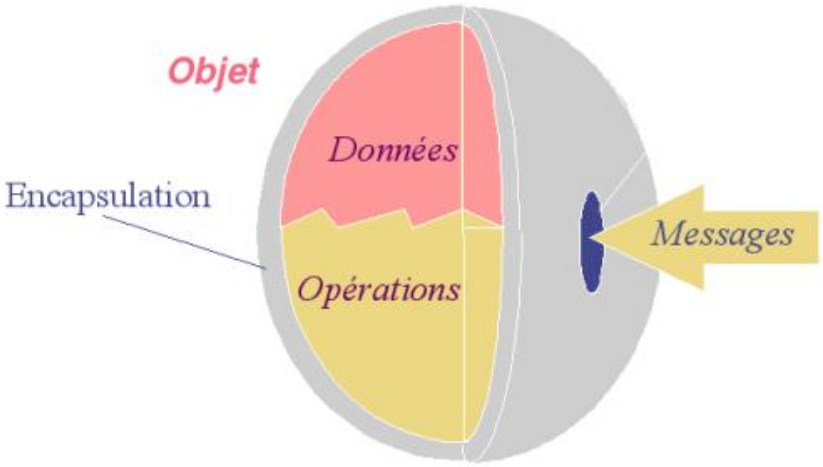


Figure 134 : Point de vue de l'objet

Utilité de l'encapsulation

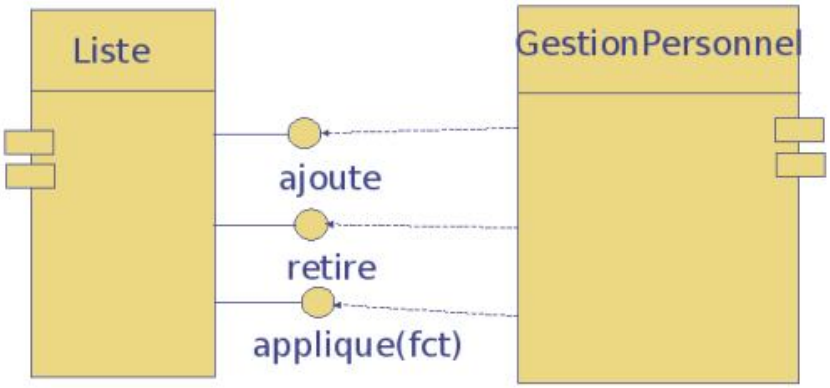


Figure 135 : Utilité de l'encapsulation

Dans cet exemple, on va confiner les modifs de Liste dans la classe Liste, le reste du code n'a pas à intervenir. L'encapsulation se fonde sur le consensus suivant : ce qui est relatif à l'implémentation est confiné dans le domaine privé, ce qui est relatif à la spécification est dans le domaine public. En pratique, chaque langage a ses particularismes.

- Smalltalk. attributs privés, méthodes publiques, ...
- Java. package, protected, ...
- C++. friends, protected, protection sur le lien d'héritage, ... Le choix du langage est alors représentatif de la philosophie que l'on souhaite adopter.

Exemple de protections en Java

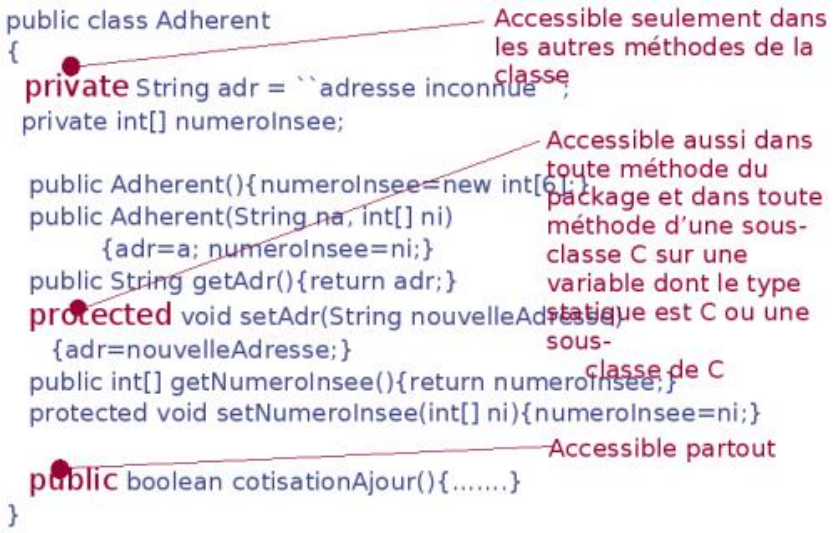


Figure 136 : Exemple de protections en Java

2.3 Variables et méthodes de classe

Variables et méthodes de classe

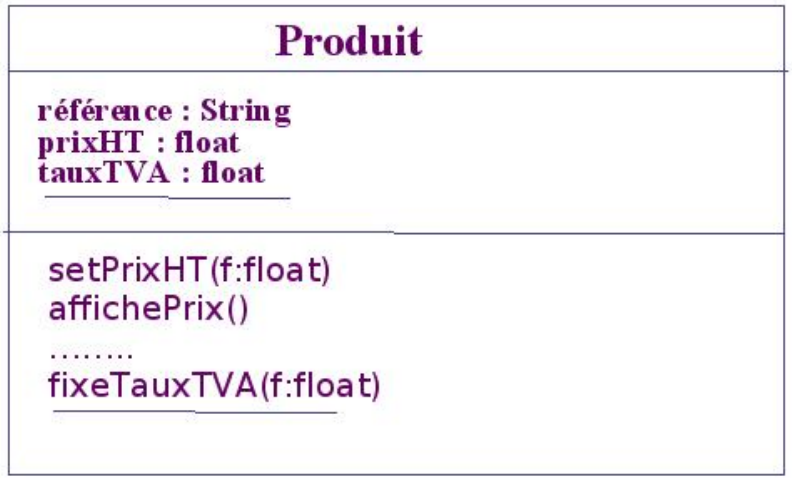


Figure 137 : Variables et méthodes de classe

En Java :

```
public class Produit
                {
                  private static float tauxTVA;
                  .....
                  public static void fixeTauxTVA(float f){.....}
                }
```

2.4 Classes abstraites, méthodes abstraites

En Java :

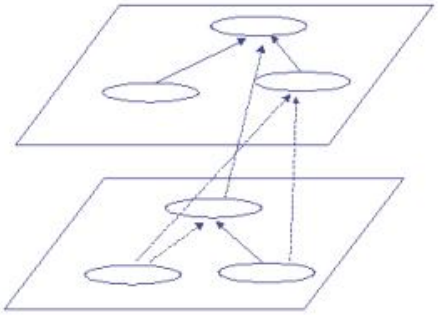
```
abstract public class Figure
    {
      abstract public void dessine();
      ........
    }
```

En C++ :

```
class Figure
    {
      public:
       virtual void dessine()const=0;
       ........
    }
```

2.5 Interfaces

Traduction des interfaces en Java : spécification de services (sans implémentation) Héritage par réalisation



Interfaces en Java

```
interface Ipolygon
{float perimeter(); ...}

interface Iquadrilateral ....
{static const int sideNb = 4; ....}

interface Isquare ....
{float getCote(); .....}

public class Square implements Isquare
{public float getCote(){....}  ....}

public class x
{....  public void meth(Isquare i) {...}  .....}
```


Figure 138 : Interfaces en Java

Interfaces en Java

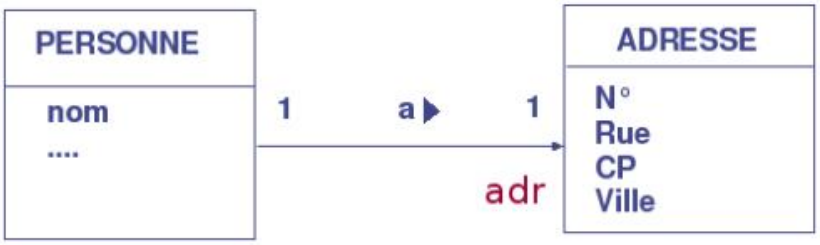
```
interface Ipolygon
{float perimeter(); ...}

interface Iquadrilateral ....
{static const int sideNb = 4; ....}

interface Isquare ....
{float getCote(); .....}

public class Square implements Isquare
{public float getCote(){....}  ....}

public class x
{....  public void meth(Isquare i) {...}  .....}
```


Figure 139 : Interfaces en Java

3 Associations

3.1 Navigabilité dans un seul sens

On utilise les noms de rôle pour déterminer les variables de classe référent aux classes associées.




```
class PERSONNE
{
                private :
                  string nom;
                  ADRESSE adr;
                  ....
              }
```

rien de spécial dans la classe ADRESSE

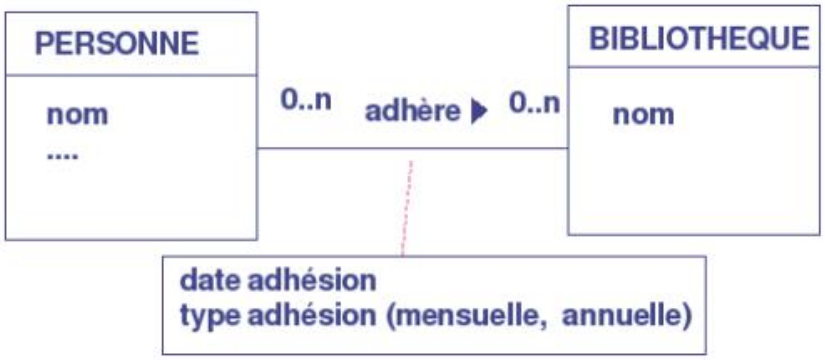
3.2 Navigabilité dans les deux sens

La classe personne est le conteneur, la classe compte la référence

```
class PERSONNE
                {private :
                    string nom;
                    list <COMPTE> comptes;....}
class COMPTE
                {private :
                    PERSONNE possesseur;....}
```

3.3 Classe d'association

On réifie la classe association en lui créant une classe à part entière.



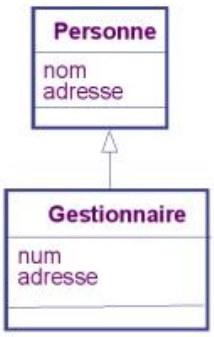

```
class PERSONNE {private : string nom;....}
            class BIBLIOTHEQUE {private : ....}
            class ADHESION
            {private :
```

```
PERSONNE adhérent;
BIBLIOTHEQUE biblio;
DATE dateadhésion;
String typeAdhesion; ....}
```

4 Généralisation / Spécialisation... et héritage

4.1 Cas simple

L'héritage traduit la généralisation/spécialisation



En C++ :

```
class Personne{};
class Gestionnaire : virtual public Personne
```

En Java :

```
public class Personne{}
public class Gestionnaire extends Personne {}
```

Malheureusement, l'implémentation de la relation de spécialisation parl'héritage pose quelques problèmes ....

- adéquation des deux relations
- représentation et traitement des conflits
- spécialisation des attributs
- représentation de la surcharge et du masquage
- utilisation de l'héritage à des fins d'implémentation ?

4.2 Spécialisation multiple

Spécialisation multiple

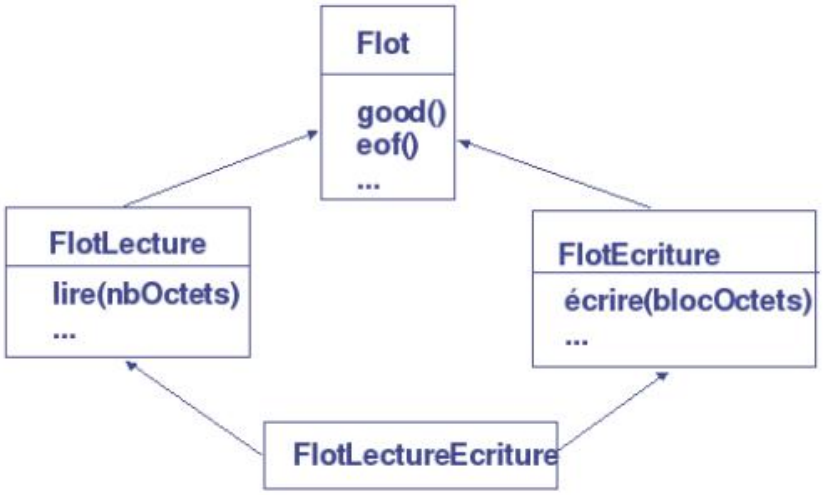


Figure 140 : Spécialisation multiple

La spécialisation multiple est idéalement rendue par un mécanisme d'héritage multiple. Celui-ci est possible en C++, impossible entre classes Java, seulement entre interfaces.

Héritage multiple entre interfaces

```
interface Ipolygon
{float perimeter(); ...}

interface Iquadrilateral extends Ipolygon
{static const int sideNb = 4; ....}

interface Isquare extends Irectangle,Irhombus
{float getCote(); .....}

public class Square implements Isquare
{public float getCote(){....} ....}

public class x
{....  public void meth(Isquare i) {...}  .....}
```

héritage multiple entre interfaces

spécialisation

implémentation

Figure 141 : Héritage multiple entre interfaces

Passage à l'implémentation : plus d'héritage multiple

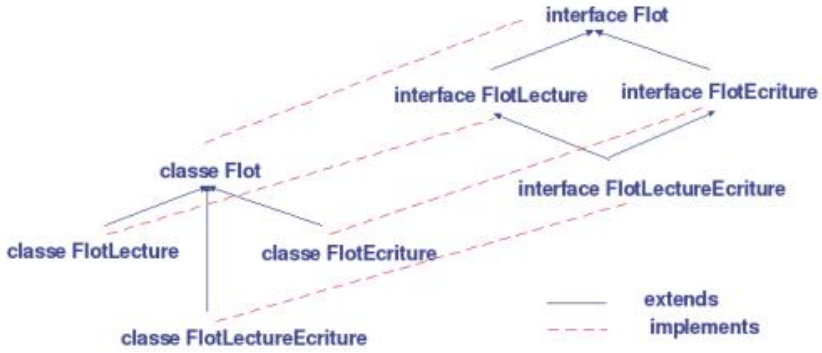


Figure 142 : Passage à l'implémentation : plus d'héritage multiple

Les méthodes de FlotLecture et FlotEcriture doivent être réécrites dans FlotLectureEcriture

4.3 Gestion des conflits

Conflit portant sur le nom des propriétés

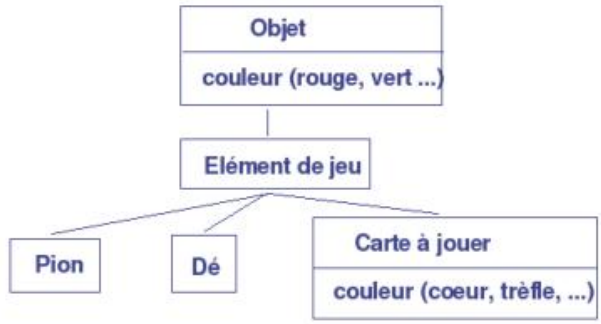


Figure 143 : Conflit portant sur le nom des propriétés

Il y a clairement deux propriétés

Conflit portant sur les valeurs ou le domaine d'une propriété

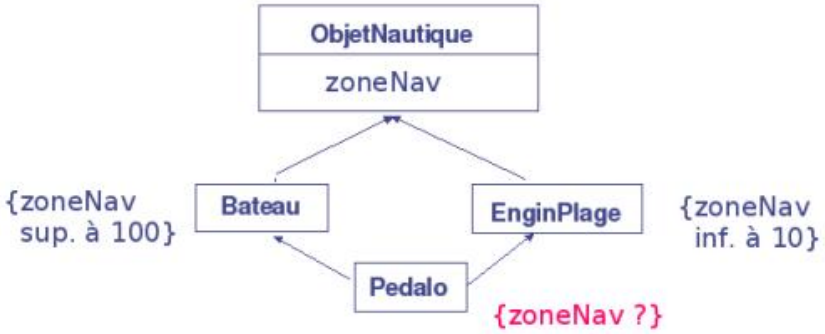


Figure 144 : Conflit portant sur les valeurs ou le domaine d'une propriété

Le pédalo n'a qu'une propriété zoneNav, mais quel sera son domaine ?

Représentation et traitement d'un conflit de nom

C++	Smalltalk	Java
Désignation explicite Objet ::Couleur CarteAJouer::Couleur	Donner deux noms différents	Pas d'accès possible à couleur d'objet autre qu'avec **super** (dans les méthodes) mieux vaut donner deux noms différents

4.4 Spécialisation des attributs

Spécialisation des attributs

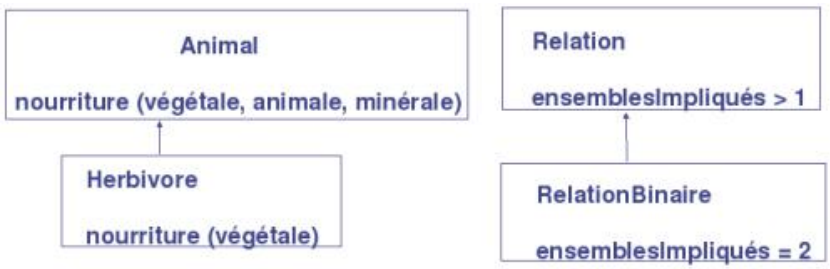


Figure 145 : Spécialisation des attributs

Il n'y a aucune représentation de ce modèle, ni en C++, ni en Smalltalk, ni en Java Seule issue : vérification des contraintes dans les méthodes (surtout au niveau des accesseurs) ...

4.5 Surcharge statique et redéfinition

DÉFINITION : SURCHARGE STATIQUE

C'est la coexistence de méthodes de même nom dans des classes différentes. L' appel est déterminé par le type de la variable (statique)

DÉFINITION : REDÉFINITION

C'est la coexistence de méthodes de même nom dans des classes comparables. L'appel déterminé par le type de l'objet (dynamique)

Cas de surchage statique

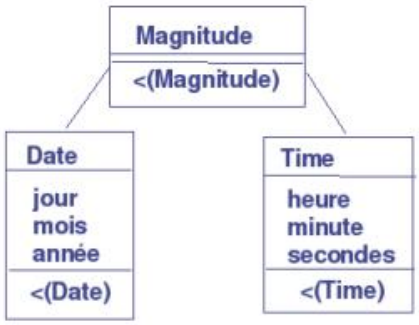


Figure 146 : Cas de surchage statique

En Java ou C++ : on ne peut pas représenter directement une spécialisation des paramètres

Cas de redéfinition

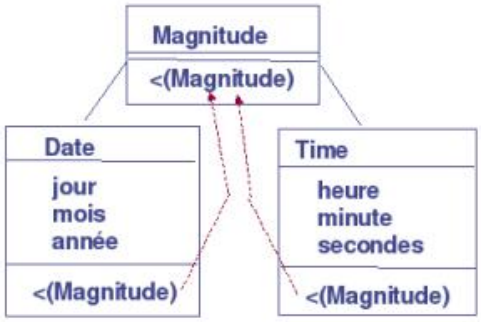


Figure 147 : Cas de redéfinition

En Java, on peut utiliser une redéfinition seulement si les signatures sont strictement identiques. En C++, la réfinition est possible si les types des paramètres sont identiques, et le type de retour peut être spécialisé
En conclusion, à chacun sa surcharge

Surcharge

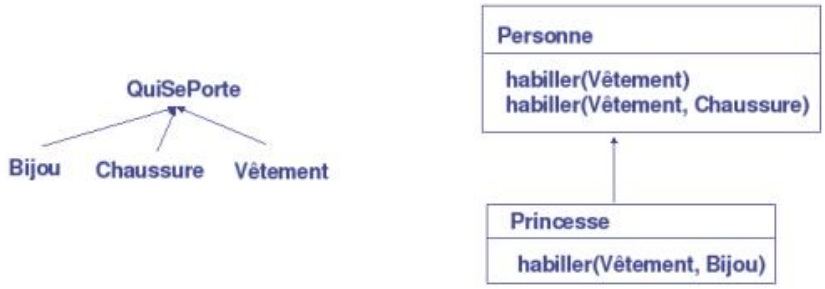
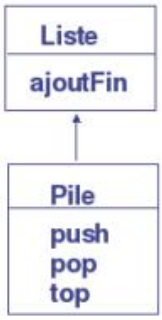


Figure 148 : Surcharge

En Java, c'est correctement interprété comme de la surcharge statique
En C++ : habiller(Vêtement, Bijou) cache les deux autres

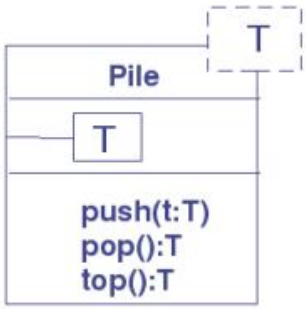
4.6 Utilisation de l'héritage pour des raisons d'implémentation ?

Il faut toujours se poser la question de la pertinence d'introduire cet héritage. Il doit être indépendant des choix d'implémentation des structures de données.



A éviter dans tous les cas ... même si les livres en sont pleins.

5 Généricité paramétrique



En C++ : on utilise un template

```
template<typename T>
        class Pile
        {... push(T element) ...}

        Pile<int> p;
```

En Java : la généricité est simulée

```
class Pile
{.... push(Object element) ....}

Pile p = new Pile();
```

Problèmes :

- contrôle des éléments insérés
- cast des éléments retirés

6 A partir du modèle dynamique

6.1 Diagrammes de séquence

Diagramme de séquences à traduire

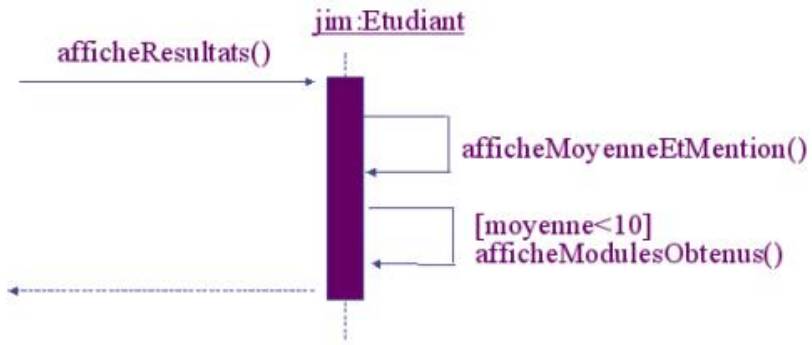


Figure 149 : Diagramme de séquences à traduire

```
public class Etudiant
   {
      public void afficheResultats()
      {
          afficheMoyenneEtMention();
          if (getMoyenne()<10)
              afficheModulesObtenus();
      }
   }
```

6.2 Diagrammes de collaboration

Diagramme de collaboration à traduire

Figure 150 : Diagramme de collaboration à traduire

```
public class Promotion
    {
       private Vector listeEtudiants;
       .....
       public float moyenneGenerale()
       {
           double mg=0;
           for (int i=0; i<listeEtudiants.size(); i++
               mg+=((Etudiant)(listeEtudiants.get(i))).moyenne();
```

```
        return mg/listeEtudiants.size();
    }
}
```

Méthodologie (RUP)

1 Introduction

Une méthode ou un processus de développement réunit :

- acteurs nécessaires (qui)
- grands types d'activités (comment)
- artefacts (quoi)
- organisation du travail (quand)

1.1 Historique

- Méthodes cartésiennes : Jackson, SADT, Yourdon
- Méthodes systémiques : Merise, Axial, Information Engineering
- Méthodes objet : OOD, HOOD, OMT, OOSE, OOA/OOD, unifiées dans le RUP (Rational Unified Process)

1.2 Concepts généraux

- Conceptualiser : obtenir un énoncé du problème (utilisateurs)
- Analyser : spécifier le problème
- Concevoir : une solution informatique
- Implémenter : réaliser la solution informatique

Étapes	Résultats
Planification	Schéma directeur
Analyse des besoins	Modèle des besoins
Spécification formelle ou analyse	Modèle conceptuel
Spécification technique ou conception	Modèle logique
Implémentation	Modèle physique
Intégration et Tests	Rapport de cohérence logique
Validation du système	Rapport de conformité
Maintenance et évolution	Documentation et trace

1.3 Cycles de développement

- en cascade
- en V
- en spirale
- tridimensionnel

1.3.1 Modèle de cycle en cascade

Modèle de cycle en cascade

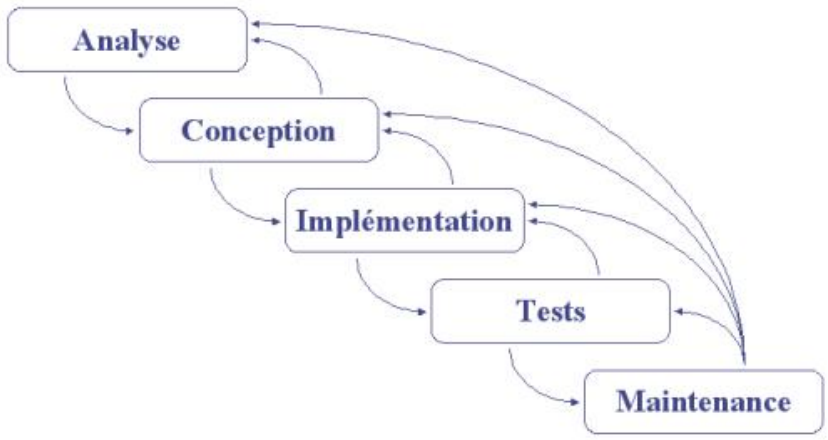


Figure 157 : Modèle de cycle en cascade

- Organisation séquentielle des phases du cycle de vie
- Une phase est structurée en un ensemble d'activités pouvant s'exécuter en parallèle par plusieurs personnes
- Le passage d'une phase à la suivante ne se fait que lorsque les sorties de la première ont été fournies

Inconvénient :

- Retours sur les phases précédentes difficiles (rupture entre les phases)
- Visualisation et validation tardive

Avantage : Organisation simple et directe

1.3.2 Modèle de cycle en V

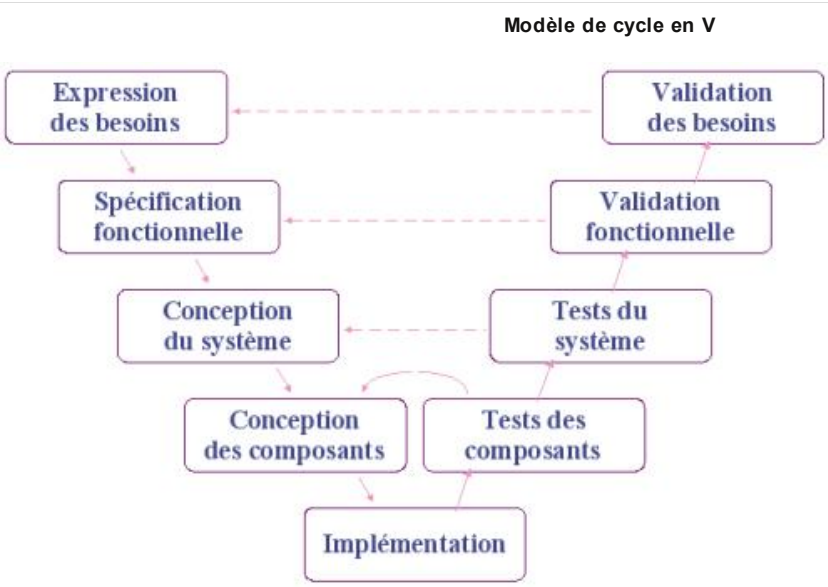


Figure 158 : Modèle de cycle en V

- Approche descendante dans les phases précédant l'implémentation : on décompose le système au fur et à mesure qu'on le construit
- Approche ascendante dans les phases suivantes : on recompose le système en testant les parties
- Hiérarchie de tests : les différents tests provoquent un retour d'information directement sur la phase permettant de corriger les erreurs.

Avantage :

- Décomposition du système en sous-systèmes
- Hiérarchie de tests et retours facilités
- Vérification ascendante

Inconvénient : Validation en fin de cycle (erreurs d'analyse coûteuses)

1.3.3 Modèle de cycle en spirale

Modèle de cycle en spirale

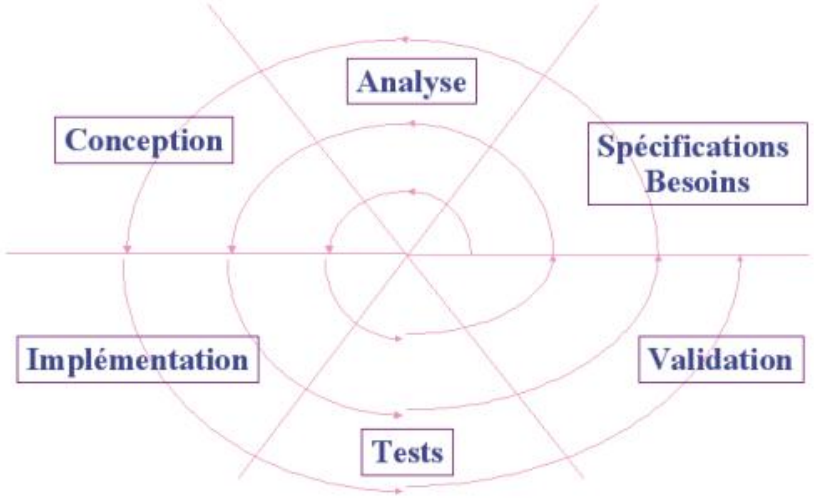


Figure 159 : Modèle de cycle en spirale

- Modèle itératif : On passe par toutes les phases du cycle de vie plusieurs fois
- Modèle incrémental : On améliore à chaque passage
- Un passage peut aussi bien permettre d'évaluer un nombre réduit de fonctionnalités ou l'organisation générale du système de façon non détaillée

Avantage :

- Réalisation de plusieurs prototypes (versions) avant la réalisation du système réel (définitif)
- Validation progressive et précoce
- Souplesse dans le choix des prototypes

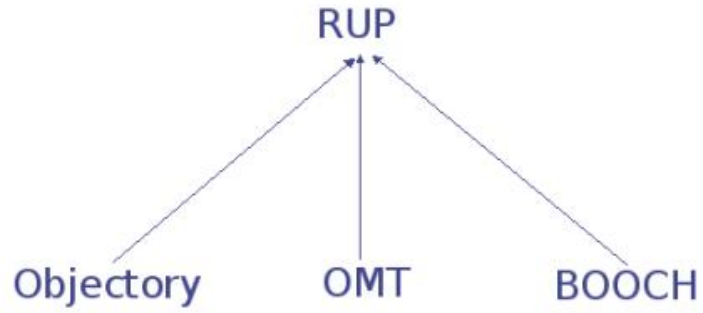
Inconvénient :

- Mise en oeuvre parfois coûteuse
- Possibilité de divergence, nombre de prototypes difficile à déterminer

Le RUP est une réponse à ces critiques

2 Rational Unified Process

2.1 Origine et principes



Mots-clefs :

- développement itératif
- développement incrémental
- pilotage par les cas d'utilisation
- centré sur l'architecture
- configurable

RUP décrit :

1) Les grandes activités

2) La notion d'architecture logicielle

3) L'organisation itérative des activités

Bibliographie

- « The RUP, an Introduction » P. Kruchten, Addison-Wesley 2000
- « The unified Software Developement Process » I. Jacobson, G. Booch, J. Rumbaugh, Addison-Wesley 1999
- « Modélisation Objet avec UML » P.-A. Muller, N. Gaertner, Eyrolles, 2002

2.2 Les grandes activités

Le RUP distingue 9 activités, à chacune correspond :

- des artefacts
- des métiers
- des outils (cf. site de Rational)

2.2.1 Activité 1. GESTION DE PROJET

- Plannification
- Allocation des tâches et des responsabilités
- Allocation des ressources
- Etude de faisabilité et des risques

Artéfact : calendrier des tâches
Métier : chef de projet

2.2.2 Activité 2. MODELISATION DE L'ORGANISATION

Modélisation

- de la structure
- du fonctionnement de l'organisation où le système sera déployé

Artéfact : cas d'utilisation de l'organisation (avec scenarii)
Métier : concepteur d'organisation

2.2.3 Activité 3. ANALYSE DES BESOINS

Détermination des besoins :

- fonctionnels (ce que l'on attend du système)
- non fonctionnels (fiabilité, temps de réponse, environnement distribué, etc.)

Artéfacts :

- cas d'utilisation du système à construire (avec scenarii)
- documents descriptifs
- conception de l'interface utilisateur

Métier : analyste

2.2.4 Activité 4. ANALYSE ET CONCEPTION

Évoluer depuis la spécification des besoins jusqu'à une solution informatique
analyse~besoins fonctionnels
conception~intègre aussi les besoins non fonctionnels
Artéfacts :

- diagrammes de classes, paquetages, sous-systèmes
- diagrammes de collaboration, d'états
- diagrammes de composants

Métier : architecte, concepteur

2.2.5 Activité 5. IMPLEMENTATION

Transcription dans un langage de programmation ou de base de données Utilisation de composants existants
Artéfact : code
Métier : implémenteur, développeur

2.2.6 Activité 6. TEST

Estimer

- si les besoins sont satisfaits
- s'il y a des erreurs/défauts à corriger

Renforcer et stabiliser l'architecture
Artéfact : modèles de test, scripts
Métier : concepteur de test, testeur
On distingue différents niveaux de tests

- unitaires (test d'une classe, d'un module isolément)
- intégration (plusieurs modules ensembles)
- validation (les fonctionnalités du système sont assurées)
- recette (souvent contractualisés, avec le client)

Ainsi que différents types de tests

- Benchmark (sur un ensemble de données type)
- Configuration/installation
- Charge
- Fiabilité, stress
- Performance

2.2.7 Activité 7. DEPLOIEMENT

Distribuer le logiciel dans son environnement opérationnel : installation, test, formation des utilisateurs, migration des données
Artéfact : diagrammes de déploiement
Métiers : formateur, graphiste, rédacteur de documentation, testeur, implémenteur (scripts d'installation)

2.2.8 Activité 8. MAINTENANCE ET EVOLUTION

Gérer pendant l'avancement du projet l'évolution :

- des besoins des utilisateurs,
- du niveau des développeurs,
- de la technologie, etc.

Artéfact : plan de modification
Métiers : tous les métiers !

2.2.9 Activité 9. ENVIRONNEMENT

Activité de support du développement :

- sélection des outils de travail,
- administration système et réseau,
- administration BD,
- formation de l'équipe de travail, etc.

Artéfact : documentation sur les outils, documentation de l'installation
Métiers : administrateur Système et Réseaux, formateur, administrateur Bases de Données

2.3 L'architecture logicielle

2.3.1 Objectif

- Analogie avec l'architecture dans le domaine du bâtiment

- Désigne un ensemble de descriptions de haut niveau (les « plans de construction »)

La vue de l'architecte est générale et sert à :

- contrôler l'intégrité du système
- identifier les éléments réutilisables
- baser le partage du travail

2.3.2 Plans de construction

Plan de construction

Figure 160 : Plan de construction

Orientation des modèles par les cas d'utilisation

2.3.3 Une définition de la notion d'architecture

« vue limitée du système permettant de comprendre :

- ce qu'il fait
- comment il fonctionne
- comment travailler sur une seule partie
- comment l'étendre
- comment réutiliser certaines parties »

Seules les grandes lignes de chaque diagramme font partie de l'architecture

2.4 L'organisation itérative des activités

Pour répondre aux problèmes connus du développement en cascade :

- découverte tardive des défauts
- intégration difficile des modifications
- contrôle temps et coûts délicat

2.4.1 Cycle de base

Cycle de base du processur itératif dans RUP

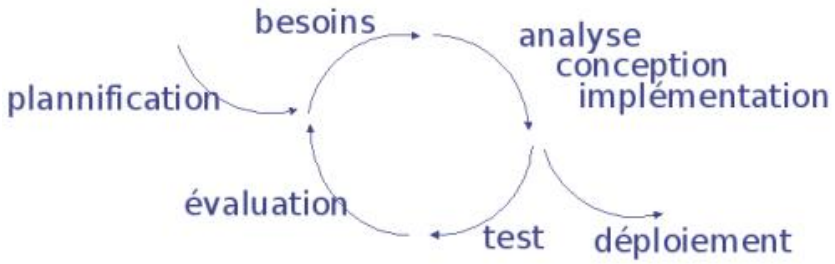


Figure 161 : Cycle de base du processur itératif dans RUP

2.4.2 Contrôle de la convergence : instauration de 4 PHASES

Chaque phase développe tout ou partie d'un ou plusieurs cycles Des points de contrôle entre les phases permettent de vérifier l'avancement

Les 4 phases du processus itératif dans RUP

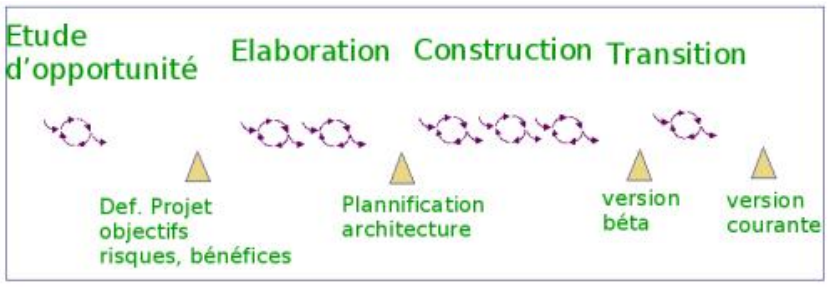


Figure 162 : Les 4 phases du processus itératif dans RUP

A chaque itération

- la plannification est remise à jour et étendue
- les modèles sont approfondis
- un prototype est développé ou augmenté

- on teste (on re-teste parfois ...) l'existant

2.4.3 Bénéfices

- résultats concrets précoces et réguliers
- problèmes et évolutions sont intégrés au fur et à mesure
- meilleure compréhension par les utilisateurs de ce qu'ils peuvent comprendre -> ils précisent mieux leur besoin
- la faisabilité est objectivement mesurée, on a des points de mesure
- tous les métiers sont en permanence sollicités, les problèmes remontent plus vite

Patrons de conception

1 Introduction

DÉFINITION : PATRON DE CONCEPTION

Un patron de conception est une solution de conception générique pour résoudre un problème de conception récurrent. Il représente la micro-architecture d'une application. On pourrait le comparer avec les plans de construction détaillés d'un élément de bâtiment

2 Un exemple : le patron « objets composite »

Nous désirons représenter des objets qui se décrivent par une hiérarchie d'objets, avec les deux particularités :

- la hiérarchie est une hiérarchie d'agrégation
- tous les objets jusqu'au plus haut niveau présentent un même comportement (on peut leur appliquer un même ensemble de méthodes)

Exemple 1
Dans un éditeur de dessin, une figure géométrique est simple ou se compose d'autres figures.Toutes les figures peuvent être dessinées, déplacées, effacées, agrandies, etc.
Exemple 2
Dans un système d'exploitation, les fichiers peuvent être ordinaires, ou bien des liens, ou encore des répertoires qui contiennent eux-mêmes d'autres fichiers.Tout fichier peut être déplacé, détruit, renommé, etc.

Solution générique [E. Gamma et al. « design patterns », 94] version dite « sécurité »

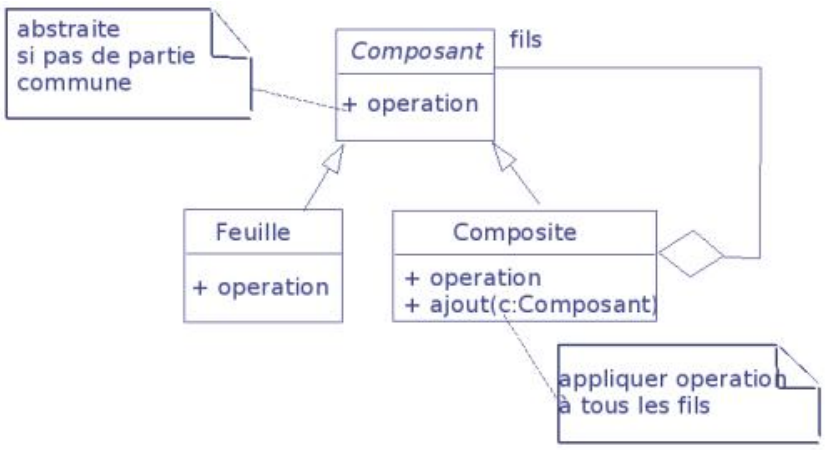


Figure 151 : Solution générique [E. Gamma et al. « design patterns », 94] version dite « sécurité »

Application à l'exemple des figures géométriques

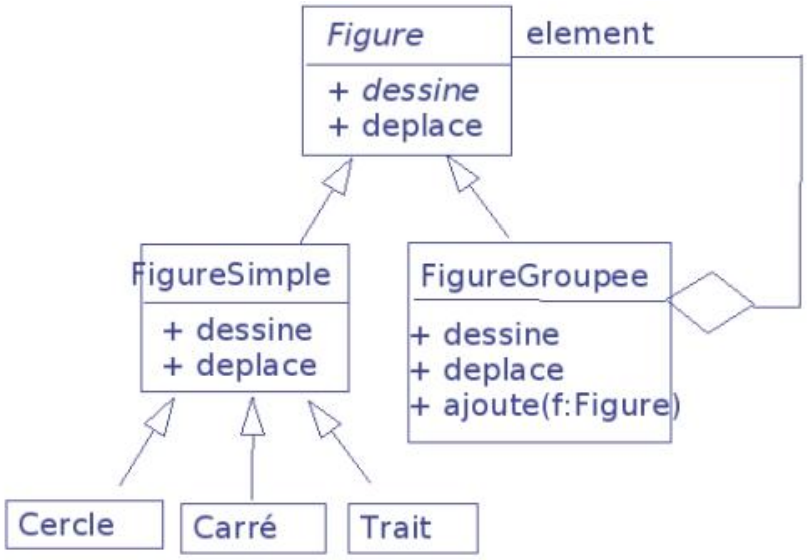


Figure 152 : Application à l'exemple des figures géométriques

3 Aperçu des principaux patrons de conception

Il existe plusieurs types de patrons de conception. Les 23 plus classiques sont appelés patrons "GoF", pour "Gang of Four", d'après les quatre personnes ayant initié le concept. Ils sont classifiés en trois grandes catégories :

- Création
  - Fabrique abstraite (Abstract Factory)
  - Monteur (Builder)
  - Fabrique (Factory Method)
  - Prototype (Prototype)
  - Singleton (Singleton)
- Structure
  - Adaptateur (Adapter)
  - Pont (Bridge)
  - Objet composite (Composite)
  - Décorateur (Decorator)
  - Façade (Facade)
  - Poids-mouche ou poids-plume (Flyweight)
  - Proxy (Proxy)
- Comportement
  - Chaîne de responsabilité (Chain of responsibility)
  - Commande (Command)
  - Interpréteur (Interpreter)

- Itérateur (Iterator)
- Médiateur (Mediator)
- Memento (Memento)
- Observateur (Observer)
- État (State)
- Stratégie (Strategy)
- Patron de méthode (Template Method)
- Visiteur (Visitor)

Tous ces patrons sont détaillés dans l'ouvrage des quatre concepteurs : Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (trad. Jean-Marie Lasvergères), *Design Patterns - Catalogue de modèles de conceptions réutilisables* (1999). Ils sont également souvent décrits sur le web, et implicitement ou explicitement utilisés dans de nombreux logiciels.

4 Le patron MVC (Modèle-Vue-Contrôleur)

Le patron Modèle-Vue-Contrôleur (MVC) est une combinaison des patrons Observateur, Stratégie et Composite. Il se compose de trois parties :

- un **modèle** décrivant les données
- une **vue** décrivant l' interface avec l'utilisateur
- un **contrôleur** indiquant comment sont gérés les événements et leurs enchaînements

Ces trois parties interagissent soit directement, soit indirectement, par le biais d'un observateur. Elles peuvent être implémentées dans des langages différents, par exemple lorsqu'il s'agit d'une application web, le modèle peut être implémenté en php et mysql, la vue en html, css et php, et le contrôleur en php seul.
Ce patron est l'un des plus répandu lorsque l'on a à concevoir une application manipulant des données en interaction avec un utilisateur. Il intervient souvent dans une analyse préliminaire, mais doit bien entendu être adapté et affiné au cas particulier qui est en train d'être traité.

5 Conclusion

Les patrons de conceptions sont des outils qui ont fait leurs preuves, mais ne doivent pas être considérés comme **la** solution indiscutable au problème que l'on se pose. En effet, ils représentent en quelque sorte des stéréotypes de réponses, qui demandent en général à être adaptés, ou combinés pour apporter une réponse acceptable à la demande de modélisation. La stratégie à employer serait alors la suivante :

- Se demander si certaines parties de la modélisation peuvent être traitées grâce à un patron de conception
- Intégrer ces patrons au modèle
- Ajuster le modèle, quitte à modifier légèrement les patrons introduits

Comme ce sont des recettes éprouvées, les patrons de conception apportent une certaine garantie du bien-fondé de la modélisation. En contrepartie, ils peuvent manquer de souplesse, et nécessitent l'intervention du cerveau humain.