Cours et travaux dirigés de modélisation UML (introduction)

Sylvie Damy

Master 1 Informatique I2A

Master DVL Master ITVL

- Finance
- Histoire
- Géographie
- Informatique
- Mathématiques
- Sciences Pour l'Ingénieur
- Français Langue Étrangère
- Administration Économique et Sociale
- Diplôme d'accès aux études universitaires

Master mention Informatique

Parcours Informatique Avancée et Applications (I2A)

CTU
Centre de Télé-enseignement Universitaire
http://ctu.univ-fcomte.fr

Filière Informatique

VVI7MPO

Modélisation et programmation orientées objet - UML

Mme DAMY - SYLVIE
sylvie.damy@univ-fcomte.fr

UNIVERSITÉ DE FRANCHE-COMTÉ
UBFC
UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

Cette brochure a été réalisée en LaTeX 2_ε.

Parcours pédagogique

Les différentes parties du cours de Modélisation et Programmation Orientées Objet sont à voir comme l'indique le parcours proposé ci-dessous. Une durée à passer sur chaque partie est proposée dans le calendrier présenté ci-dessus.

Sem.	Date	Partie	Chapitre(s)	Devoir	Correction
1	09 Oct	Java	Intro et 1ère partie chap. 2		
2	16 Oct	Java	2ème partie chap2 + 3 et 4		
3	23 Oct 26 Oct	Java	Chap 5 : Collections		
- - -	30 Oct	- - -	- - - - - - - *Vacances* - - - - - - -	- - - -	- - - -
4	06 Nov	Java	Chap 6 : Les entrées-sorties		
5	13 Nov	Java	Chap 7 : JDBC		
6	20 Nov	Java		Devoir 1	
7	27 Nov	UML	Intro UML		
8	04 Déc	UML	Diagramme de classes		
9	11 Déc	UML	Diagramme de cas d'utilisation		Devoir 1
10	18 Déc	UML	Diagramme de séquence		
- - -	25 Déc	- - -	- - - - - - - *Vacances* - - - - - - -	Devoir 2	- - - -
- - -	01 Janv	- - -	- - - - - - - *Vacances* - - - - - - -	- - - -	- - - -
11	08 Janv	UML		Devoir 3	Devoir 2
12	15 Janv	UML	Révision		Devoir 3
	22 Janv		Examen : 26/01		

Contents

Part I

Introduction

Chapter 1

Introduction à UML

Ce chapitre propose une présentation générale du langage de modélisation UML (Unified Modelling Language). L'objectif de cette partie du cours de MPOO n'est pas de vous faire découvrir complètement UML, mais plutôt de vous permettre d'utiliser des outils pour concevoir, implémenter et déployer des systèmes informatiques.

1.1 Modèle et langage de modélisation

Avant de parler de langage de modélisation il faut bien évidemment définir la notion de modèle ; notion présente dans toutes les disciplines scientifiques.

1.1.1 Modèle

Un **modèle** est une représentation d'une partie du monde réel qui a pour objectif de la comprendre et de la construire. Il ne s'agit pas du monde réel, mais d'une description limitée et orientée de ce monde : une **abstraction**. L'abstraction permet d'isoler les aspects importants du système de ceux qui ne le sont pas. Pour cela elle doit être réalisée avec une finalité clairement définie.
Il n'est pas possible de décrire de manière exhaustive le monde réel, ce qui d'ailleurs n'aurait pas vraiment d'intérêt. Un modèle n'est pas conçu pour représenter complètement le monde réel mais pour répondre à un usage. L'abstraction du monde réel rend le modèle plus facile à manipuler et permet de gérer la complexité. Il n'existe pas de modèle correct et universel mais seulement des modèles correspondant à un objectif donné. Citons dans un autre domaine, le statisticien G. Box "*Tous les modèles sont faux, certains sont utiles*".

Le modèle est un outil qui permet de communiquer, réfléchir et transformer les organisations et les systèmes qui les structurent. Ces principaux objectifs sont :

- de **tester un système** avant de le construire. La construction d'un modèle avant celle du système revient moins cher et permet de corriger les défauts de façon précoce. Dans des domaines tels que la physique notamment il est possible de simuler des systèmes avant leur construction.
- de **communiquer avec les clients**. Les architectes et les concepteurs informatiques conçoivent des modèles pour les présenter aux clients, en particulier des maquettes et des prototypes.

- **de visualiser**. Les modèles ont souvent une représentation graphique assez intuitive qui permet simplement de visualiser les parties, comportements, ... du système décrit.
- **de réduire la complexité**. Un système complet est souvent très complexe, le modèle en simplifiant le système et en ne prenant en compte que certains éléments permet de traiter la complexité.

1.1.1.1 Que contient un modèle ?

Un modèle décrit une partie du monde réel, en utilisant des concepts qui permettent de classifier la réalité observée. Par exemple, un processus de validation de stage sera décrit avec des événements (sujet de stage déposé), des activités (compléter les informations demandées, décrire le sujet de stage, demander la validation du stage) et des transitions (lorsque les informations obligatoires sont remplies alors la convention peut être lancée).

On décrit le monde réel en répondant aux questions : "***Quoi ? Qui ? Quand ? Où ? Comment*** ?" et en décrivant les contraintes qui définissent les limites du modèle. Un modèle contient ainsi :

- **le quoi :** ce sont des objets ou des concepts, dénommés par un nom ou groupe nominal.
- **le qui :** ce sont des tiers physique/moral ou rôles, dénommés par un nom se rapportant à cette entité ou ce rôle.
- **le quand :** les éléments temporels servent dans beaucoup de systèmes à dater les événements de manière absolue (20 février 2017) ou relative (3h après le début de la pousse). Il peut s'agir aussi d'intervalles de temps, et de durées. Certaines actions ne sont pertinentes dans un système que durant des périodes (dans le temps) données.
- **le où :** ce sont des éléments de localisation. Dans les systèmes d'informations prenant en compte des observations environnementales par exemple, la précision du lieu où l'observation a été faite est essentielle (localisation GPS, commune, ...) avec des niveau de précision plus ou moins fins.
- **le comment :** ce sont des actions qui décrivent ce que font les acteurs humains ou machine sur les objets ou leur représentation, ainsi que l'enchaînement de ces actions.

1.1.1.2 Les contraintes

Un modèle inclut ensuite des contraintes sur les liens entre les différents éléments qu'il contient. Par exemple, le concept "Stage" est lié à un et un seul "Etudiant", le montant minimal de la gratification mensuelle du stage est de 545€(valeur mise à jour chaque année), etc. Le langage de modélisation UML inclut un langage déclaratif pour ces contraintes : OCL.

1.1.1.3 Les concepts utilisés dans le modèle : le méta-modèle

Un modèle contient des concepts de base (qui, quoi, quand, où, comment, contraintes) mais également d'autres concepts ou un approfondissement de ces concepts permettant de préciser ce que le modèle représente de la réalité. Le choix du méta-modèle est essentiel pour aboutir à une représentation de qualité. Dans le domaine informatique, le paradigme objet est le méta-modèle incontournable qui s'est imposé pour la description et l'exécution

de modèles grâce à sa richesse et sa pertinence pour décrire le monde réel, et également grâce aux nombreux langages de programmation basés sur ce paradigme.

1.1.2 Langage de modélisation

Un **langage de modélisation** permet de décrire un système. Il peut être constitué de pseudo-codes, codes, diagrammes, textes, images, ... On appelle **notation** les éléments qui forment un tel langage. Il doit permettre :

- de représenter des concepts abstraits (graphiquement par exemple),
- de limiter les ambiguïtés en proposant un langage commun, au vocabulaire précis, indépendant des langages informatiques,
- de faciliter l'analyse.

La description de la signification des notations constitue la **sémantique du langage**. Il existe différents langages de modélisation qui dépendent du domaine d'utilisation tels que Unified Modeling Language (UML), Systems Modeling Language (Sysml) et Business Process Modeling Notation (BPMN). SysML est à l'ingénierie des systèmes complexes et/ou hétérogènes ce qu'UML est à l'informatique. SysML permet à des acteurs de corps de métiers différents de collaborer autour d'un modèle commun pour définir un système. BPMN est une notation graphique qui est utilisée pour représenter un processus métier en séparant les informations métier des informations techniques.

1.2 UML

UML ou *Unified Modeling Language* est, comme son nom l'indique un **langage de modélisation** qui permet de décrire les systèmes d'informations du point de vue conceptuel et physique. L'OMG ou *Objet Management Group* définit UML comme un langage graphique de visualisation, spécification, construction et documentation d'un système.
A l'origine d'UML se trouvent Grady Booch, Ivar Jacobson et James Rumbaugh. Chacun d'entre eux avait défini sa propre méthode (respectivement Booch, OOSE, et OMT) de modélisation objet. A l'initiative de Rational Software, ils ont mis en commun leurs travaux pour proposer la première version d'UML en 1995, comme le montre la figure 1.1 .

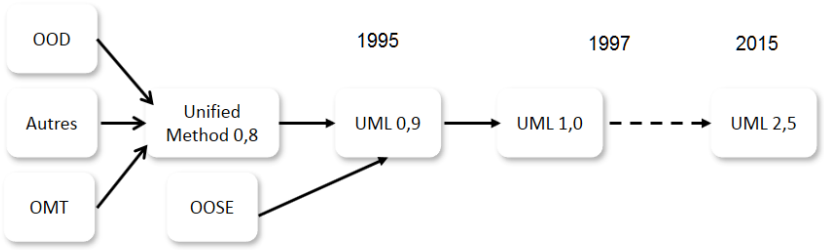

Figure 1.1 : Evolution des méthodes de modélisation objet vers UML

Les concepteurs de ce langage se sont fixés quatre objectifs :

- représenter des systèmes entiers à l'aide de concepts objet,
- établir une dépendance explicite entre les concepts et les éléments d'informations,

- prendre en compte les facteurs d'échelle liés aux systèmes complexes et critiques,
- proposer un langage de modélisation manipulable par des humains et des machines.

La version 1.1 a été soumise à l'OMG en septembre 1997. Ce qui a démarré comme l'unification de différentes méthodes de conception de logiciels est devenu un langage de modélisation unifié pouvant être utilisé pour l'ensemble des tâches de la conception des logiciels et des systèmes. UML est ainsi devenu un standard incontournable dans les domaines du génie logiciel et des bases de données.

L'OMG qui en a fait son langage de modélisation (UML a été utilisé comme langage de base pour les nouvelles normes de l'OMG : MOF, MDA, ...), a proposé en mars 2015 les spécifications de la version 2.5. La dernière version 2.5.1 est sortie en décembre 2017 (https://www.omg.org/spec/UML/2.5.1/).
UML permet de représenter un système en utilisant plusieurs vues complémentaires : les **diagrammes**. Un diagramme présente certaines parties du modèle selon une vue.

1.3 Les diagrammes

UML définit treize diagrammes au total, dont l'utilisation, ou non, est laissée à l'appréciation de chacun. Chaque diagramme représente une facette du monde réel.
Un **diagramme UML** est une représentation graphique, qui traite un aspect précis du modèle. Il permet de visualiser et de manipuler des éléments de modélisation.
Chaque diagramme a une structure (éléments de modélisation qui le composent) et véhicule une sémantique précise, comme l'illustre le schéma 1.2 .

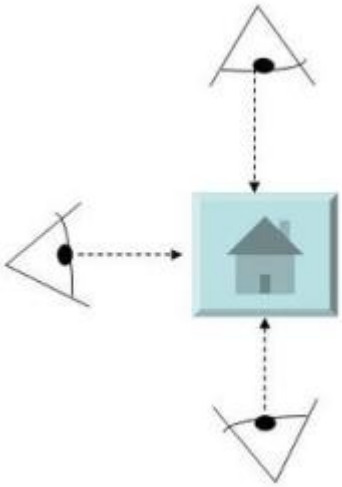

Figure 1.2 : Les diagrammes offrent différentes vues du modèle

Les diagrammes se regroupent en deux catégories de diagrammes **structurels** et **comportementaux** et permettent de représenter respectivement des vues **statiques et dynamiques** d'un système.

- **Diagrammes structurels ou statiques** :

  - **Diagramme de classes ou *Class Diagram* :** ce diagramme, le plus connu de tous les diagrammes UML, est considéré comme le plus important. Il représente l'architecture conceptuelle du système : il décrit les classes que le système utilise, ainsi que leurs relations.
  - **Diagramme de composants ou *Component Diagram* :** un composant est un élément encapsulé et réutilisable. Le diagramme de composants permet de décrire l'organisation du système.
  - **Diagramme d'objets ou *Object Diagram* :** il permet d'illustrer un diagramme de classes par des exemples. Il représente les objets et leurs liens.
  - **Diagramme de structures composites ou *Composite Structure Diagram* :** les structures composites permettent de montrer comment des objets fonctionnent ensemble ; elles captent certains détails que les diagrammes de classes et/ou de séquences n'arrivent pas toujours à montrer et mettent l'accent sur les liens entre les sous-ensembles qui collaborent.
  - **Diagramme de déploiement ou *Deployment Diagram* :** il présente la vue physique du système et doit permettre la livraison d'une application utilisable.
  - **Diagramme de paquetages ou *Package Diagram* :** il illustre les dépendances entre les différents paquetages définis lors de la modélisation.
- **Diagrammes comportementaux ou dynamiques :**
  - **Diagramme d'activités ou *Activity Diagram* :** il propose une vision des enchaînements des activités propres à une opération ou à un cas d'utilisation. Il permet de représenter graphiquement le comportement d'une méthode ou le déroulement d'un cas d'utilisation.
  - **Diagramme des cas d'utilisation ou *Use Case Diagram* :** il décrit ce que le système doit faire (exigences fonctionnelles).
  - **Diagramme d'états-transitions ou *State Machine Diagram* :** il décrit les états du système et ses transitions. Il permet de mettre l'accent sur certains comportements du système. Ce diagramme est appelé aussi diagramme de machines à états.
  - **Diagramme global d'interactions ou *Global interaction diagram* :** il permet de présenter une vue générale du fonctionnement des interactions décrites dans les autres diagrammes. Il regroupe en une seule vue les interactions qui réalisent une partie spécifique du système.
  - **Diagramme de séquences ou *Sequence Diagram* :** il décrit l'ordre dans lequel les interactions entre les différentes parties du système peuvent avoir lieu. Il représente les collaborations entre objets d'un point de vue temporel.
  - **Diagramme de collaboration ou *Collaboration Diagram* :** il présente les interactions entre objets notamment avec les messages échangés. C'est un diagramme proche du diagramme de séquences. Ce diagramme est appelé aussi diagramme de communication.
  - **Diagramme de temps ou *Timing Diagram* :** il permet de représenter des états et des interactions dans un contexte où le temps a une réelle importance. Ce diagramme est appelé aussi diagramme de chronométrage.

UML 2 a renommé et clarifié les diagrammes dont une présentation synthétique est donnée dans le tableau 1.1 .

Type de diagramme	**Modélisations possibles**	**1ère version**
Classe	Classes, types, interfaces et leurs relations	UML 1.x
Objet	Instances des classes définies dans le diagramme de classes	Informellement UML 1.x
Composant	Composants du système, interface, coopération des composants	UML 1.x mais nouvelle signification dans UML 2.0
Déploiement	Déploiement des composants sur les dispositifs matériels et communication entre tous ces éléments	UML 1.x
Paquetage	Regroupement de n'importe quels éléments UML (surtout les classes)	UML 2.0
Structure composite	Éléments internes d'une classe ou d'un composant	UML 2.0
Cas d'utilisation	Interactions entre le système et les utilisateurs ou d'autres systèmes	UML 1.x
Activité	Comportement d'une méthode ou d'un cas d'utilisation, ou processus métier	UML 1.x mais amélioré dans UML 2.0
Séquence	Interactions entre des objets lorsque l'ordre des interactions est important	UML 1.x
Collaboration	Manière dont les objets interagissent et connexions nécessaires à la prise en charge des interactions	Nouveau nom des diagrammes de collaboration de UML 1.x
Temps	Interactions entre des objets lorsque le minutage est un facteur important	UML 2.0 Appelé aussi diagramme de chronométrage
Global d'interaction	Réunit en une seule vue l'ensemble des interactions qui réalisent un élément particulier du système.	UML 2.0
Etats-transitions	Etats d'un objet et événements à l'origine des changements de ces états	UML 1.x

Table 1.1 : Les diagrammes UML et les versions

Les diagrammes de séquences coexistent avec les diagrammes de communication et de chronométrage pour modéliser de façon plus précise les interactions entre les différentes parties du système.

1.3.1 Les vues UML

Chaque type de diagramme joue un rôle particulier dans le modèle global. Et il existe plusieurs façons de décomposer les différents diagrammes en vues pour présenter une facette particulière du système.
Une **vue** est une manière de regarder des éléments de modélisation parfois issus de modèles différents.
La description d'un système avec UML se concrétise à travers 5 vues, présentées dans la figure 1.3 , qui permettent de décrire un système selon cinq préoccupations différentes :

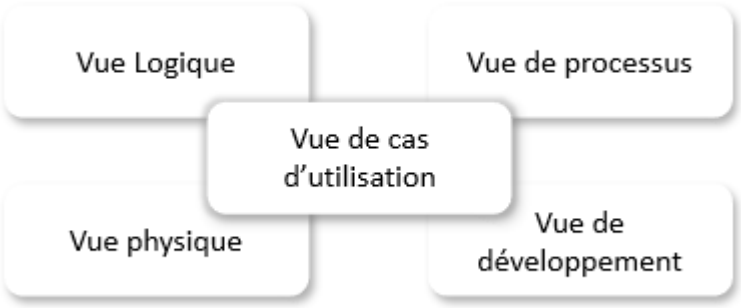

Figure 1.3 : Le modèle de vues 4+1 de Kruchten

1.3.2 La vue logique

Cette vue donne les définitions abstraites des parties d'un système et sert à modéliser les composantes du système et leurs interactions. Elle décrit les aspects statiques et dynamiques du système.
Les diagrammes que l'on trouvent dans cette vue sont :

- le diagramme de classes,
- le diagramme d'objets,
- le diagramme d'états-transitions,
- le diagramme global d'interactions.

1.3.3 La vue de processus

Cette vue décrit les processus du système. Elle contient en général le diagramme d'activités.

1.3.4 La vue de développement

Elle organise des parties du système en modules et composants. Elle contient en général :

- le diagramme de paquetages,
- le diagramme de composants.

1.3.5 La vue physique

Elle décrit comment le système présenté dans les trois vues précédentes est mis en oeuvre par un ensemble d'entités réelles. Elle contient généralement le diagramme de déploiement.

1.3.6 La vue de cas d'utilisation

Cette vue décrit la fonctionnalité du système du point de vue du monde extérieur. Cette vue forme la colle qui unifie les quatre vues précédentes, d'où le nom 4+1 donné au modèle de Kruchten. Cette vue contient en général :

- le diagramme de cas d'utilisation,
- le diagramme global d'interactions.

1.4 Etude d'UML dans ce cours

Ce cours n'a pas pour objectif de vous présenter tous les diagrammes UML, nous verrons simplement 3 diagrammes correspondant chacun à un des points de vue classique de la modélisation :

1. Fonctionnel : Diagramme de cas d'utilisation
2. Statique : Diagramme de classe
3. Dynamique : Diagramme de séquence

Chacun de ces diagrammes fera l'objet d'un chapitre de la partie 2 de ce cours.

Part II

UML : Les diagrammes

Chapter 2

Les diagrammes de classes

Les classes sont au coeur de tout système orienté objet. Il est donc naturel que le diagramme de classes soit le diagramme UML le plus connu, et le seul obligatoire lors d'une modélisation objet. La structure d'un **système** est composée d'un ensemble d'**objets** qui sont décrits par des **classes**.

Les **diagrammes de classes** présentent la structure statique du système en termes de classes et de relations entre ces classes.

Pour illustrer les notions décrites dans cette partie nous utiliserons l'exemple "`Formaperm`" présenté plus en détail à la fin de ce chapitre.

2.1 Description d'une classe

Une classe est représentée par un rectangle qui peut être composé de trois parties (cf. figure 2.1). Seule la partie supérieure est obligatoire. Le fait qu'une partie soit absente ne signifie pas qu'elle soit vide, mais plutôt que le diagramme est plus facile à comprendre sans cette partie.

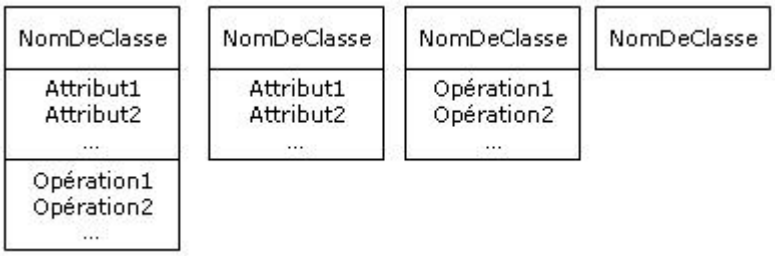

Figure 2.1 : Les différentes façons de représenter une classe en UML

- La partie supérieure donne le **nom de la classe**. Ce nom doit évoquer le concept décrit par la classe : `Personne`, par exemple. Il commence par une majuscule. Lorsqu'il s'agit dune classe abstraite (c'est à dire non instanciable), ce nom est écrit en italique.
- La partie médiane donne le **nom des attributs** pour lesquels peuvent être précisés :
  - leur type : int, string, ...,

  - leur visibilité :
    * # accès protégé : seuls les objets issus de sous-classes ont accès à cet attribut.
    * + accès public : tous les objets ont accès à cet attribut.
    * - accès privé : seul lobjet lui même a accès à cet attribut.
    * ~ accès paquetage : seul les objets issus de classes dans le même paquetage ont accès à l'attribut.
- La partie inférieure donne le **nom des opérations**. Celles-ci sont nommées et complétées par :
  - leurs paramètres sous la forme (nom : type) ; labsence de paramètre se note ().
  - leur visibilité, selon la même codification que pour les attributs.

La classe `Etudiant` est représentée en UML par la figure 2.2 . Un étudiant est ainsi formé de 3 attributs et on propose une opération d'affichage des informations correspondant à un étudiant.

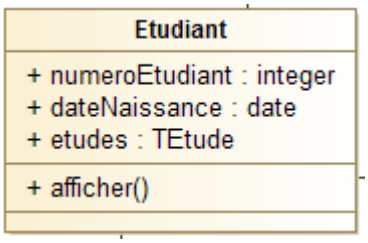

Figure 2.2 : La classe Etudiant

2.2 Visibilité

En utilisant la **visibilité**, une classe peut décider des opérations et des attributs qu'elle montre aux autres classes. Il existe quatre niveaux de visibilité : publique, protégée, de paquetage et privée.

2.2.1 Visibilité publique +

Cette visibilité qui permet l'accès le plus large, est dénotée par le signe `+` placé devant l'attribut ou l'opération concernés. L'ensemble des attributs ou des opérations déclarés public forment l'**interface de la classe**. Tous les objets ont accès à ces attributs ou opérations. Comme le montre la figure 2.3 , toutes les classes du modèle peuvent accéder à l'attribut public : `attribut1`.
Dans la classe `Etudiant` l'opération `afficher` est déclarée comme étant publique. Ce qui signifie que toutes les applications utilisant la classe `Etudiant` pourront utiliser cette méthode.

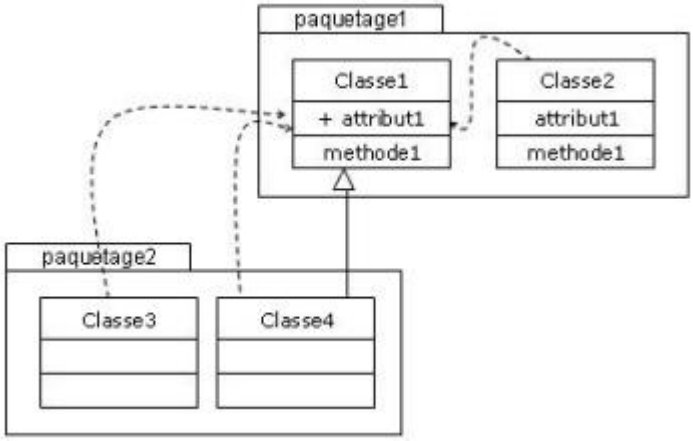

Figure 2.3 : Visibilité publique

2.2.2 Visibilité protégée

Cette visibilité permet un accès plus réduit pour les attributs ou opérations publics, mais plus importante que pour ceux déclarés privés. On dénote cette visibilité par le signe #. Les méthodes de la classe et celles des classes qui héritent de la classe peuvent accéder aux éléments déclarés protégés dans la classe. Cette visibilité est intéressante lorsque l'on veut que des classes spécialisées accèdent à un attribut ou une opération de la classe de base sans exposer cet élément à tout le système.
Comme le montre la figure 2.4 , toutes les classes du modèle qui héritent de la classe `Classe1`, peuvent accéder à l'attribut protégé : `attribut1`. C'est à dire les méthodes de la classe : `Classe4`.

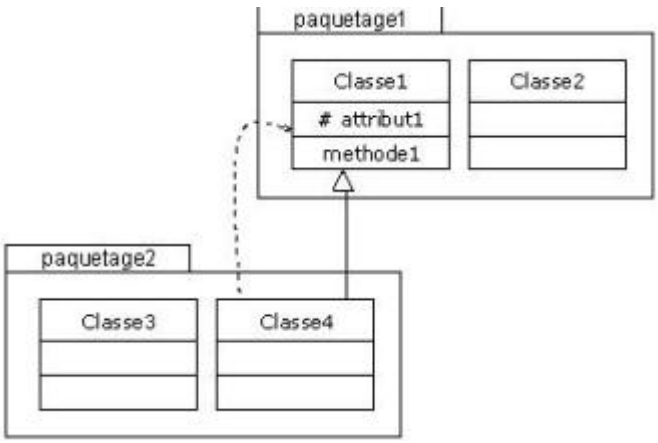

Figure 2.4 : Visibilité protégée

2.2.3 Visibilité de paquetage ~

Cette visibilité indiquée par le signe tilda ~ se situe entre les visibilités protégée et publique. Les paquetages déterminent les attributs ou les méthodes déclarés avec cette visibilité.

Dans la figure 2.5 , toutes les classes du paquetage 1 peuvent accéder à l'attribut : `attribut1`, par contre celles du paquetage2 ne le peuvent pas.

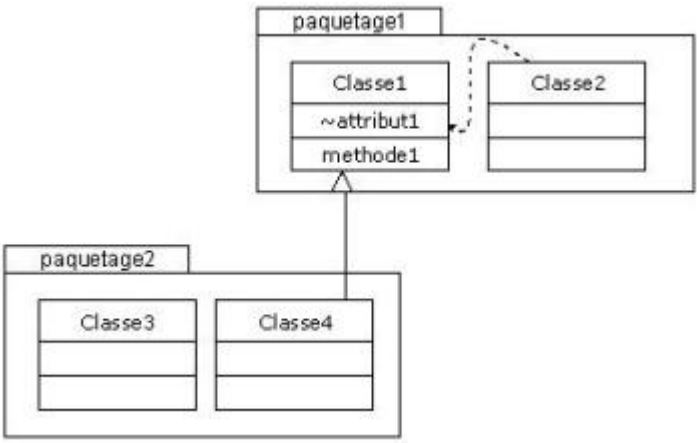

Figure 2.5 : Visibilité de paquetage

2.2.4 Visibilité privée -

Cette visibilité est la plus réduite, elle est indiquée par le signe - placé devant l'attribut ou la méthode. Dans la figure 2.6 , seules les méthodes de la classe `Classe1` peuvent accéder à l'attribut : `attribut1`.

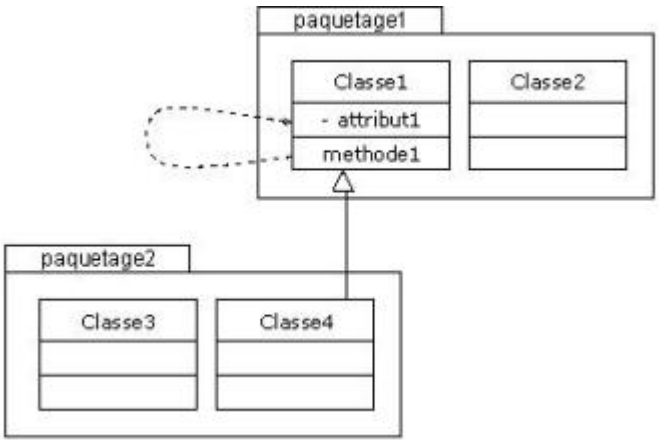

Figure 2.6 : Visibilité privée

2.3 Relations entre classes

Lors de l'analyse d'un système, il ressort que les classes sont reliées les unes aux autres, elles ne vivent pas isolées ; ces relations entre les classes sont les liens entre les objets instanciés et se codifient de différentes façons selon le degré de relation à exprimer.
Il existe différents niveaux de relations entre classes comme le montre la figure 2.7 :

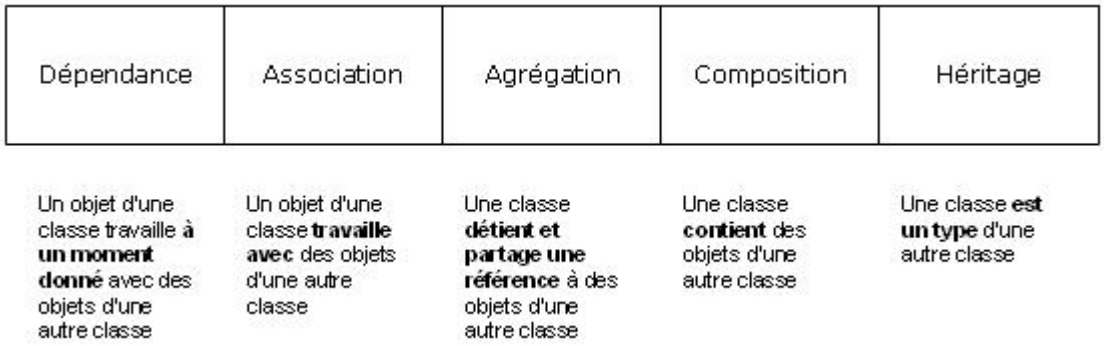

Figure 2.7 : Les relations entre classes

2.3.1 L'héritage

L'héritage décrit une relation entre une classe spécialisée ou sous-classe et une classe générale ou super-classe. L'héritage permet la factorisation des caractéristiques de plusieurs classes appelées sous-classes enfants dans une classe appelée super-classe. Cette mise en commun des attributs et des opérations est un mécanisme de réutilisation très plus important de lorientation objet.
Dans le langage UML, le symbole utilisé pour décrire une relation d'héritage (ou spécialisation) est une flèche avec un trait plein dont la pointe est un triangle fermé dont la pointe désigne la classe la plus générale comme le montre l'exemple de la figure 3.11.

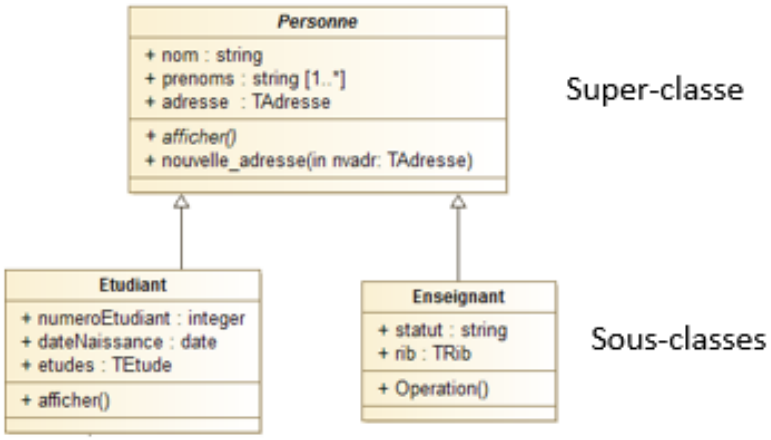

Figure 2.8 : Exemple d'héritages dans Formaperm

Dans l'exemple présenté nous définissons les classes `Etudiant` et `Enseignant` comme des sous-classes de la classe `Personne`. En plus des attributs `numeroEtudiant`, `dateNaissance` et ETUDES, un étudiant a les attributs `nom`, `prenoms` et `adresse`. Et il peut utiliser la méthode `nouvelle_adresse`.
Remarque : En UML la relation d'héritage n'est pas spécifique aux classes. On peut avoir des héritages entre paquetages ou acteurs dans les cas d'utilisation.

2.3.2 La dépendance

Les relations de dépendance sont utilisées lorsqu'il existe une relation sémantique entre plusieurs éléments qui n'est pas une relation structurelle.

Une **dépendance entre deux classes** stipule qu'une classe a besoin d'informations sur une autre classe pour pouvoir utiliser des objets de celle-ci.

Avec ce type de relation entre classes on autorise simplement une classe à utiliser des objets d'une autre classe, les objets des deux classes peuvent fonctionner ensemble. On utilise souvent ce type de relation pour les classes qui fournissent des fonctions utilitaires générales. Lorsque qu'une telle relation est réalisée par des liens entre objets, ceux-ci sont limités dans le temps, contrairement à des relations plus structurelles telles que les associations. Une dépendance est représentée par un trait discontinu allant de la classe dépendante vers la classe cible, se terminant par une flèche ouverte (cf. figure 2.9).

Figure 2.9 : Représentation UML de la relation de dépendance

Considérons l'exemple suivant dans la classe `Etudiant` on suppose que la méthode `afficher()` permet d'utiliser une méthode plus générale `imprimer`. Cette méthode permet au moment de l'impression de récupérer des informations sur l'imprimante en cours. La relation entre l'étudiant et l'imprimante est bien évidemment temporaire, elle ne dure que le temps de l'impression. Il s'agit d'une dépendance.

2.3.3 L'association

Une association est une relation structurelle entre deux classes ou plus.

Une **association** entre une classe A et une classe B signifie que la classe A contiendra une référence à un ou plusieurs objets de la classe B sous la forme d'un attribut.

Une association binaire est représentée par un trait plein comme le montre la figure 2.10

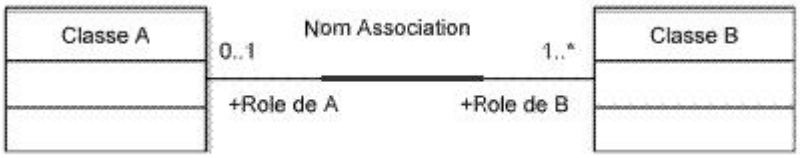

Figure 2.10 : Représentation UML d'une association

Une association n-aire (cf. figure 2.11) est représentée par un losange avec un trait allant vers chaque classe participante.

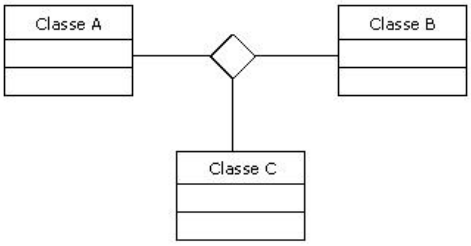

Figure 2.11 : Représentation UML d'une association ternaire

Une association compte au moins deux extrémités d'association ou terminaisons d'association (deux dans la figure 2.10 et trois dans la figure 2.11).

Chaque association peut comporter un certain nombre d'informations.

2.3.3.1 Nommage des associations

Les associations peuvent être nommées : le nom figure alors au milieu du trait qui symbolise l'association. Dans la figure 2.10 le nom de l'association est "`Nom Association`".

2.3.3.2 Rôle des terminaisons d'association

Une terminaison d'association peut être nommée, mais ceci est facultatif. Le nom est placé vers la terminaison dont il détermine le rôle. Dans la figure 2.10 , le nom de terminaison "`rôle de A`" est associé à la terminaison du coté de la classe `A`.

2.3.3.3 Visibilité

Tout comme les attributs, les terminaisons d'association possèdent une visibilité. Cette visibilité est notée à proximité de la terminaison ou, lorsque cela est possible, devant le nom de la terminaison.
Dans la figure 2.10 , le nom de terminaison "`rôle de A`" est précédé du symbole "+" pour préciser que la visibilité est de type public.

2.3.3.4 Multiplicité des associations

Chaque extrémité d'une association peut avoir une indication de multiplicité qui précise combien d'objets de la classe considérés peuvent être liés par l'association à un objet de l'autre classe.
Les multiplicités habituelles sont données dans la table 2.1 .

Notation	Multiplicité
1 ou 1..1	Un et un seul
0..1	Zéro ou un
N	N (entier naturel)
M..N	De M à N
* ou 0..*	De zéro à plusieurs
1 .. *	De un à plusieurs

Table 2.1 : Valeurs de multiplicité

Dans la figure 2.10 , la multiplicité `0..1`, placée vers la classe `A` signifie que tout objet de la classe `B` est lié à 0 ou 1 objet de la classe `A` avec l'association "`Nom Association`".

2.3.3.5 Navigabilité

La navigabilité indique s'il est possible de traverser une association. Par défaut les associations sont navigables dans les deux sens. Lorsque l'on représente la navigabilité sur une des extrémités, implicitement l'association ne peut être traversée que dans un sens.
La navigabilité est représentée par une flèche pointant l'extrémité vers laquelle la navigation est possible. D'après la figure 2.12 il sera possible de retrouver à partir d'un objet de la

Figure 2.12 : Navigabilité en UML

classe `A` les objets de la classe `B` qui lui "correspondent".
L'intérêt de réduire la navigabilité est de diminuer le couplage et les dépendances entre classes.
L'association de la figure 2.10 est navigable dans les deux sens.

2.3.3.6 Classe association

Dans certains cas il peut être utile d'associer à une association des attributs voire même des opérations. On peut alors représenter une telle association par une classe. On appelle ce type de classe : **classe-association**. Elle possède à la fois les caractéristiques d'une classe et d'une association et peut participer à d'autre relations dans le diagramme.
On représente une classe-association par un trait plein allant vers chaque classe participante et un trait pointillé vers la classe-association, comme le montre la figure 2.13 .

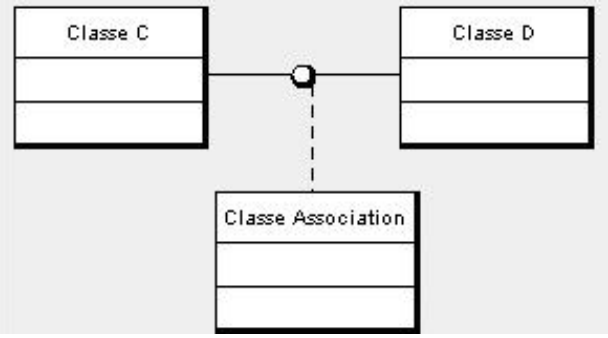

Figure 2.13 : Exemple de classe-Association

2.3.4 L'agrégation

L'agrégation représente une association non symétrique.

Une **agrégation** est une relation entre deux classes A et B spécifiant que les objets de la classe A sont composés d'objets de la classe B.

Les deux classes restent relativement indépendantes l'une de l'autre. Une des classes joue un rôle plus important que l'autre. L'agrégation permet d'exprimer des relations de type maître et esclaves. On représente une agrégation (cf. figure 2.14) comme une association en ajoutant un losange vide du coté de l'agrégat (maître).

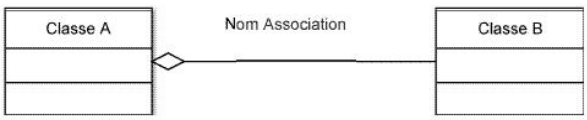

Figure 2.14 : Représentation de l'agrégation en UML

L'agrégation permet de modéliser une contrainte d'intégrité. Dans la figure 2.14 chaque objet de la classe `A` est composé d'un ou plusieurs objets de la classe `B`.

2.3.5 La composition

La composition est un cas particulier de l'agrégation avec un couplage plus important. La classe ayant un rôle prédominant est appelée **classe composite**.

Une **composition** est plus forte qu'une agrégation. Les objets de la classe B dépendent d'un objet de la classe A.

On représente une composition comme une association en ajoutant un losange plein du coté de la classe composite (cf. 2.15).
La composition implique la coïncidence des durées de vies des composants et du composite. La destruction du composite entraine la destruction des composants, et leur création ou modification sont de la responsabilité du composite.

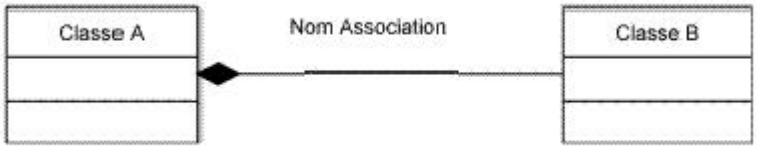

Figure 2.15 : Représentation de la composition en UML

Un composant n'est pas partageable. La multiplicité du coté du composite (classe A dans la figure 2.15) ne peut être que 0 ou 1.

2.4 L'exemple Formaperm

L'exemple `Formaperm` va permettre d'illustrer toutes les notions présentées dans cette partie du cours de MPOO. Nous donnons ici la description générale de cet exemple ainsi qu'une représentation sous la forme d'un diagramme de classes. Cet exemple a été inspiré d'un exemple proposé sur le site de cours de l'EPFL [EPFL].

2.4.1 Description générale

Considérons la gestion d'un institut de formation permanente. Il est nécessaire de gérer les cours de l'institut, ses enseignants, ses étudiants avec leurs inscriptions et leurs résultats aux différents cours.
On donne les règles suivantes :

- Un cours a un nom qui l'identifie, et un cycle. Chaque cours peut avoir, en pré-requis, 0 ou plusieurs cours du même cycle ou du cycle précédent. Un cours est réalisé par un seul enseignant. Le support d'un cours est composé de un ou plusieurs documents. Un cours possède un objectif. La suppression d'un cours entraine celle de son objectif.
- Un enseignant a un nom qui l'identifie et des prénoms. Il peut assurer un ou plusieurs cours. L'institut a besoin pour chaque enseignant de son nom, ses prénoms, son adresse, son numéro de téléphone, son statut et ses renseignements bancaires.
- Les étudiants s'inscrivent à plusieurs cours. Lors de la première inscription l'étudiant reçoit un numéro qu'il conserve durant toute sa formation. Chaque étudiant a un nom, des prénoms, un numéro (qui l'identifie), une adresse, des études antérieures (diplôme et année) et une date de naissance. De plus l'institut conserve pour chaque étudiant la liste des cours qu'il a validés, avec la note et l'année.
- Un document peut être utilisé dans plusieurs cours. Il est décrit par un titre. Un document n'a pas lieu d'exister s'il n'est pas utilisé dans un cours.
- Un objectif correspond à un et un seul cours. Il est composé d'un texte appelé contenu.

On considère de plus que l'organisme ne gère qu'une seule filière, qui se déroule sur plusieurs cycles. Dans le modèle nous ne traitons que les informations concernant l'année en cours (inscription à un cours, ...).

2.4.2 Lecture et analyse de la description générale

L'objectif de cette lecture est de bien comprendre l'énoncé, et de délimiter le système qui nous intéresse, ici : `FormaPerm`. La première étape consiste à chercher les **acteurs** qui interagissent avec le système, à rechercher les fonctionnalités du système en décrivant les cas d'utilisation.
Enfin la description générale du système est analysée afin de faire ressortir les différentes **classes** ainsi que les **relations** existant entre ces classes.

2.4.3 Les classes

Nous allons développer de façon progressive le diagramme de classes. Tout d'abord repérons les différentes classes :

- `Cours`
- `Enseignant`
- `Etudiant`
- `Document`
- `Objectif`

Les classes `Enseignant` et `Etudiant` ont une partie commune, que nous représentons en définissant la classe `Personne`.
A partir de l'énoncé on définit les attributs suivants :

- Cours : nom : string, cycle : TCycle (1, 2 ou 3)
- Personne : nom : string, prénom : liste de String, adresse : TAdresse
- Enseignant : statut : String, rib : TRib
- Etudiant : numeroEtudiant : Integer, dateNaissance : date, etudes : Tetude
- Document : titre : String
- Objectif : contenu : String

Nous ne nous intéressons pas ici aux classes `TAdresse`, `TRib`, `TEtude` et `TCycle`. Ces classes pourront être développées plus loin pour l'implantation du modèle proposé. Mais ici le diagramme peut parfaitement être proposé sans que l'on donne plus d'informations sur ces classes.

2.4.4 Les relations

Définissons maintenant les relations entre ces classes.

2.4.4.1 Association réflexive

Chaque cours peut avoir 0 ou plusieurs cours du même cycle ou du cycle précédent en pré-requis.
La relation "`a pour pré requis`" décrite dans la figure 2.16 , est une association dite **réflexive** qui relie la classe `Cours` à elle même. Un cours peut être le pré-requis de 0 à plusieurs cours : cette information est donnée par la multiplicité `0..*` placée vers la terminaison "`est pré requis`". Nous considérerons dans la suite que cette association ne sera navigable que d'un cours vers ses pré-requis.

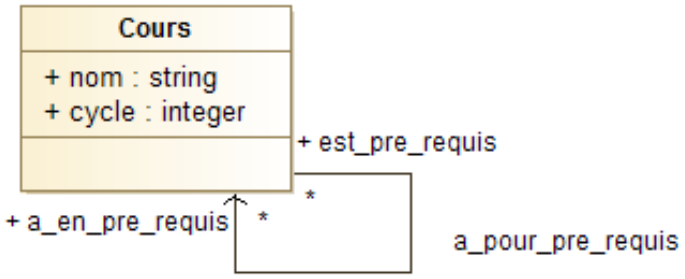

Figure 2.16 : Association "a pour pré requis"

2.4.4.2 Héritage

Un étudiant est une personne. Un enseignant est une personne.
Nous avons fait apparaître la classe `Personne` pour traiter les éléments communs aux étudiants et aux enseignants. Comme le montre la figure 2.17 , nous avons une relation d'héritage, entre les classes `Etudiant` et `Enseignant` et la classe `Personne`.
Par ailleurs la super-classe `Personne` est déclarée **abstraite** (en italique sur la figure).

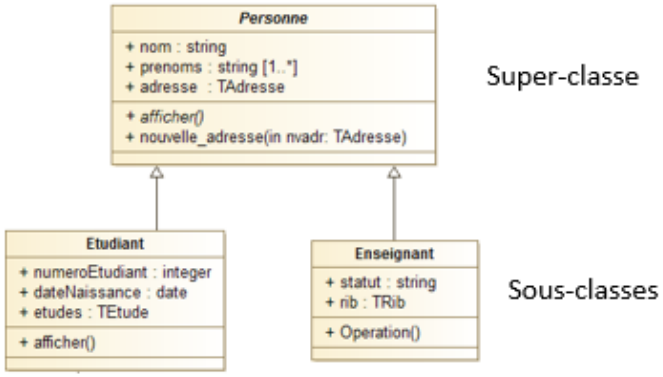

Figure 2.17 : Relation d'héritage

En effet, nous ne souhaitons pas instancier cette classe. Une personne ne sera définie qu'à travers un étudiant ou un enseignant. Dans cette classe on peut aussi remarquer que la méthode `Afficher` est déclarée abstraite. En effet l'affichage sera défini au niveau des étudiants ou des enseignants.

2.4.4.3 Associations

- Les étudiants s'inscrivent à plusieurs cours.
Nous supposons qu'un étudiant est inscrit au moins à un cours, et qu'un cours a un ou plusieurs étudiants. L'association `Est_inscrit` 2.18 est **navigable** dans les deux sens. On peut ainsi retrouver la liste des cours d'un étudiant et la liste des étudiants d'un cours.
- Un cours est réalisé par un seul enseignant. Un enseignant peut assurer un ou plusieurs cours.
L'association "`Realise`" 2.18 est navigable dans les deux sens.

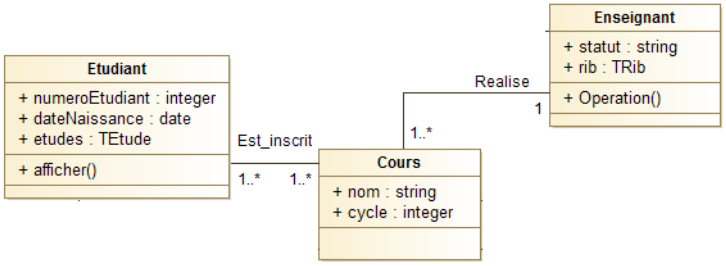

Figure 2.18 : Associations dans formaperm

2.4.4.4 Classe association

L'institut conserve pour chaque étudiant la liste des cours qu'il a validés, avec la note et l'année.
Un étudiant peut avoir validé zéro ou plusieurs cours, et un cours peut être validé par zéro à plusieurs étudiants. La note et l'année associées à une validation ne peuvent être stockées ni du côté d'un étudiant ni du côté d'un cours. Nous proposons donc ici la **classe-association** "`Validation`" présentée dans la figure 2.19 .

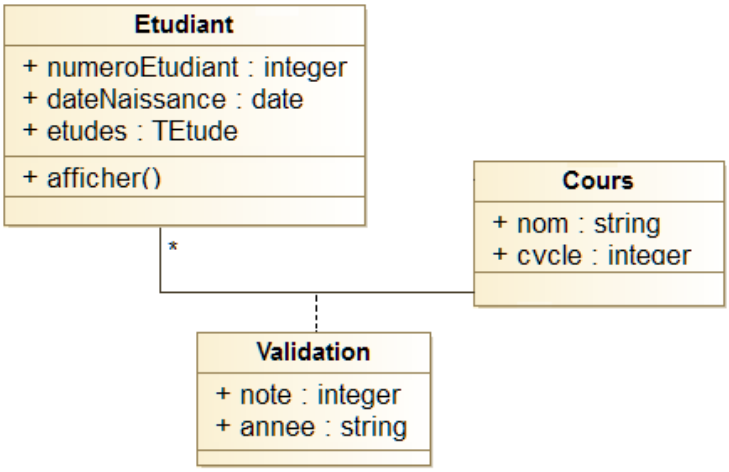

Figure 2.19 : Exemple de classe association

2.4.4.5 Agrégation

Le support d'un cours est composé de plusieurs documents. Nous représentons cette relation par l'**agrégation** "`utilise comme support`" (cf. figure 2.20).

2.4.4.6 Composition

Un cours ne peut posséder qu'un seul objectif. La suppression d'un cours entraine celle de son objectif. Nous avons ici une **composition** "`a pour objectif`" (cf. figure 2.20). La destruction du cours entrainera la destruction de l'objectif.

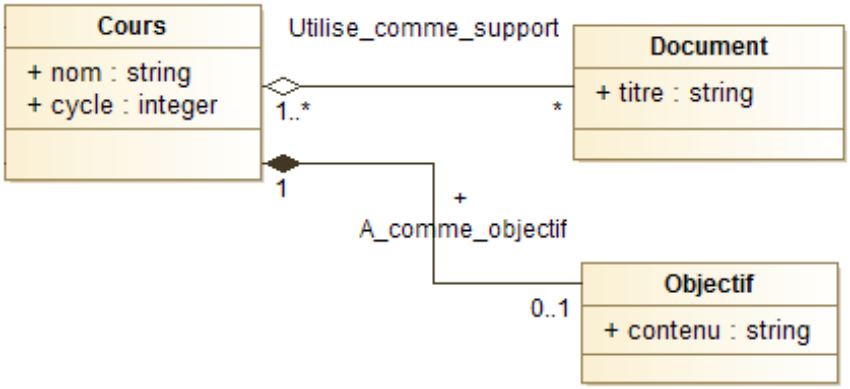

Figure 2.20 : Agrégation et composition dans Formaperm

2.4.5 Le diagramme de classes

Pour finir nous obtenons le diagramme de classes donné en figure 2.21

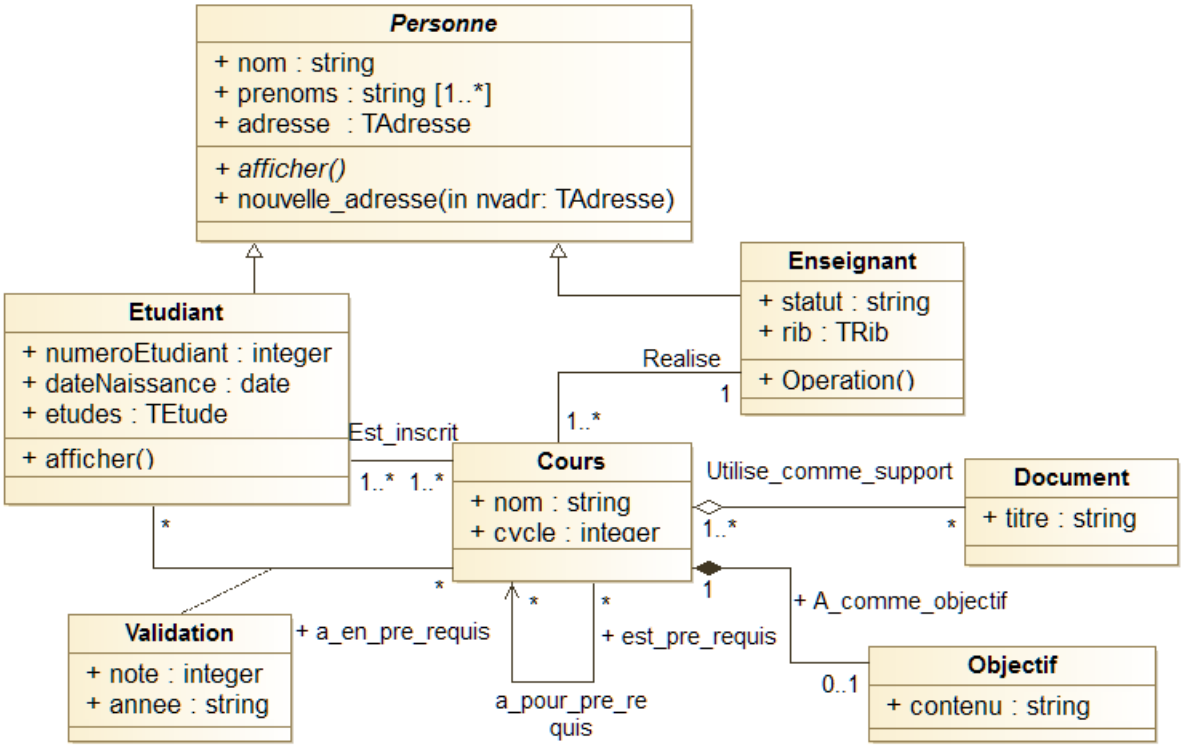

Figure 2.21 : Le diagramme de classe de Formaperm

Nous nous sommes plus intéressé à la partie données que traitement. Seules certaines fonctions ont été proposées. Celles-ci pourront être plus développées par la suite.

Chapter 3

Les diagrammes de cas d'utilisation

Le **diagramme de cas d'utilisation** ou **Use Case Diagram** présente les fonctionnalités du système. C'est une vue fonctionnelle du modèle UML qui décrit les fonctionnalités offertes par le système sans préciser comment ces fonctionnalités sont réalisées. Il capture le comportement d'un système, d'un sous-système, d'une classe ou d'un composant tel qu'un utilisateur extérieur le voit.

3.1 Les concepts de base

Un diagramme de cas d'utilisation représente les cas d'utilisation, les acteurs et les relations entre cas d'utilisation et acteurs.

3.1.1 Le cas d'utilisation

Un **cas d'utilisation** correspond à une situation dans laquelle le système répond à une attente des utilisateurs. Un cas d'utilisation permet d'exprimer un besoin des utilisateurs, il correspond à une vision orientée utilisateur de ce besoin et non informatique. On spécifie un cas d'utilisation par un intitulé.

Il est recommandé d'intituler les cas d'utilisation sous la forme `verbe + complément`. Le verbe de l'intitulé permet de spécifier la nature de la fonctionnalité offerte par l'application, tandis que le ou les compléments permettent de spécifier les données d'entrée ou de sortie de la fonctionnalité.
Un cas d'utilisation est représenté par une ellipse contenant son intitulé.

Exemple : Le cas d'utilisation `imprime liste étudiant d'un cours` spécifie que l'application `Formaperm` permet à certains utilisateurs d'imprimer la liste des étudiants inscrits dans un cours. Il peut être représenter comme le montre la figure **??**. Les cas d'utilisation permettent d'exprimer les besoins en se centrant sur les utilisateurs. On part du principe qu'un système est construit pour ses utilisateurs. Pour réduire la complexité de la détermination des besoins d'un système, on détermine les cas d'utilisation pour chaque

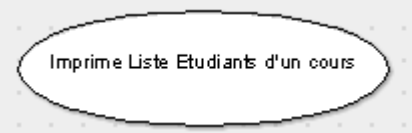

Figure 3.1 : Représentation graphique d'un cas d'utilisation

type d'utilisateur.
Le formalisme d'expression d'un cas d'utilisation est basé sur le langage naturel, ainsi les cas d'utilisation décrivent le futur système de façon compréhensible par tous et surtout par les futurs utilisateurs. Les termes employés, qui correspondent à ceux des utilisateurs, permettent d'exprimer facilement les besoins et évitent ainsi des dérives vers des solutions inadaptées.

3.1.2 L'acteur

Un **acteur** représente une entité appartenant à l'environnement du système qui interagit directement avec lui. C'est un utilisateur externe qui communique avec le système. Un acteur peut être une personne, un équipement, un autre système ; il interagit simplement avec le système.
Un acteur est identifié par un nom et est représenté par un *bonhomme bâton* ou sous la forme d'un classeur stéréotypé. Le nom donné à l'acteur doit être compris par le client et les concepteurs du système.

Exemple : Dans l'application `Formaperm` nous définissons les acteurs `Enseignant`, `Etudiant`, `Scolarité` (cf. fig 3.1.2). Nous définissons un héritage de Enseignant vers

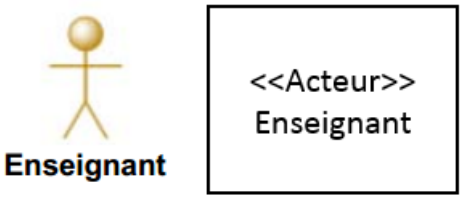

Figure 3.2 : Représentations graphiques d'un acteur

Scolarité (cf. fig 3.3), ce qui permettra à toute personne de la scolarité de disposer des mêmes fonctionnalités que les enseignants. Ils auront en plus certaines fonctionnalités spécifique à la scolarité telle que réaliser l'inscription administrative d'un étudiant.

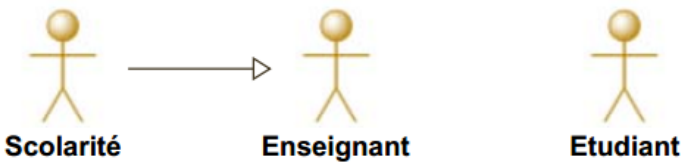

Figure 3.3 : Les différents acteurs de FormaPerm

3.1.3 Le système

Un système représente une application dans le modèle UML. Il est identifié par un nom et regroupe un ensemble de cas d'utilisation qui correspondent aux fonctionnalités offertes par l'application à son environnement. L'environnement est spécifié sous forme d'acteurs liés aux cas d'utilisation.
Il est représenté par un cadre avec un nom, ses acteurs sont à l'extérieur et ses cas d'utilisation à l'intérieur.
Exemple : Nous présentons l'exemple d'un diagramme de cas d'utilisation de la gestion des étudiants (cf. figure 3.4). On a l'acteur :*Scolarité* et cinq cas d'utilisation : *"inscrire un étudiant"*, *"Retirer un étudiant"*, *"Inscrire étudiant à un cours"*, *"Saisir une note"* et *"Modifier étudiant"*.

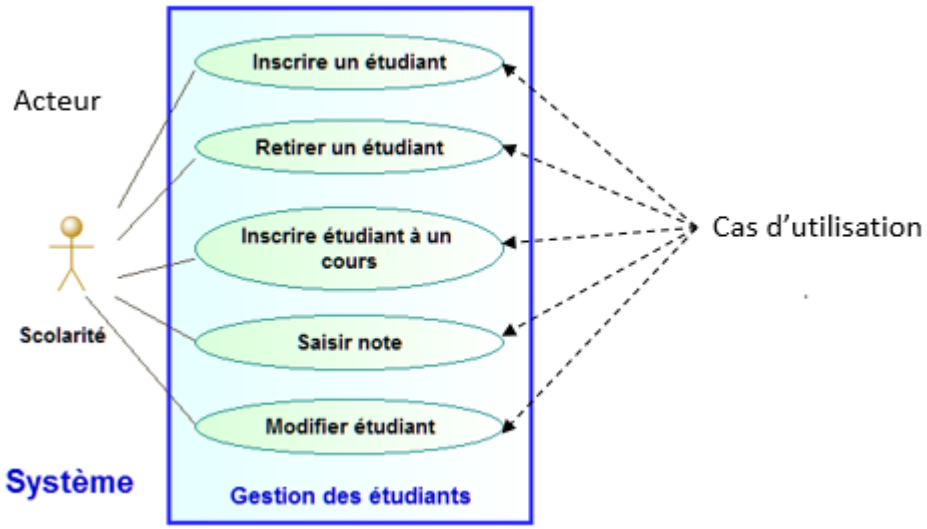

Figure 3.4 : Diagramme de cas d'utilisation

3.2 Les relations entre cas d'utilisation

On distingue 3 types de relations entre cas d'utilisation : la généralisation, l'inclusion et l'extension.

3.2.1 Relation de généralisation/spécialisation

Comme pour les classes, il est possible de spécialiser un cas d'utilisation. Le sous-cas d'utilisation hérite du cas d'utilisation ou sur-cas d'utilisation. Dans un diagramme de cas d'utilisation cette relation est représentée par une flèche comme dans le cas des diagrammes de classes.
On peut même définir des cas d'utilisation abstraits ; dans ce cas le libellé du cas d'utilisation est écrit en italiques ou accompagné du stéréotype $<<$ `abstract` $>>$.

3.2.2 Relation d'inclusion

Une relation d'inclusion entre cas d'utilisation signifie que l'instance du cas d'utilisation source comprend le comportement décrit par le cas d'utilisation destination. Cette relation est utilisée lorsqu'un ensemble d'actions peut être utilisé dans plusieurs cas d'utilisation et que l'on ne souhaite pas répéter cet ensemble.
Un tel ensemble est alors décrit dans un cas d'utilisation séparé et est lié au cas d'utilisation qui l'utilise par une flèche pointillée munie du stéréotype $<<$ `include` $>>$.

3.2.3 Relation d'extension

Une relation d'extension entre cas d'utilisation signifie que le cas d'utilisation source peut étendre le comportement du cas d'utilisation destination. Cette extension est optionnelle, contrairement à l'inclusion. Cette relation est utilisée lorsqu'un cas d'utilisation est similaire à un autre cas d'utilisation à l'exception d'une petite variation. Celle variation est alors décrite dans un cas d'utilisation à part et les deux cas d'utilisation sont ensuite liés par une relation d'extension. Dans un diagramme de cas d'utilisation une telle relation est représentée par une flèche pointillée munie du stéréotype $<<$ `extend` $>>$.
Ces deux relations sont représentées de la façon suivante :

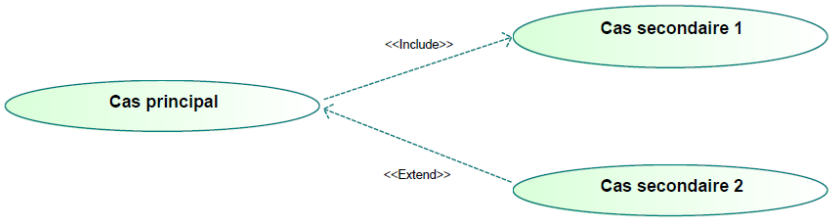

Figure 3.5 : Relations d'inclusion et d'extension

Le cas principal doit utiliser le cas secondaire 1 et le cas secondaire 2 peut être utilisé par le cas principal.

3.3 Recommandations pour l'écriture des cas d'utilisation

L'élément essentiel de la rédaction d'un cas d'utilisation se situe dans la description des étapes du scénario. Le style d'écriture doit être clair, précis et concis, sans ambiguïté.
La représentation textuelle d'un cas d'utilisation est utile pour discuter avec le client car elle est intuitive et concise. Cependant elle n'est pas suffisante pour l'équipe de développement. UML ne préconise pas de format particulier pour la présentation textuelle d'un cas d'utilisation.
Les recommandations essentielles présentées ci-dessous, sont présentées dans les ouvrages de Alistair Cockburn ([1], [2]) qui font référence. Ce dernier préconise une décomposition en trois grandes parties :

- Définition des interactions en cas de succès
- Cas particuliers et leurs contraintes diverses
- Présentation du cas par des illustrations

3.3.1 Douze recommandations d'écriture de A. Cockburn

La description des différentes étapes sert à la fois à l'utilisateur et au développeur. Le langage naturel doit être un minimum "guidé" afin que tout le monde puisse s'y retrouver. En effet le développeur en déduit un système déterministe qui doit satisfaire l'utilisateur final.

1. : Partir du sommet (les grandes fonctions), et se maintenir le plus possible au niveau objectif utilisateur
2. : Centrer son attention sur le cas nominal (un scénario typique de succès)
3. : Préciser toujours les parties prenantes et leurs intérêts
4. : Utiliser un verbe au présent de l'indicatif à chaque étape
5. : Utiliser la voie active pour décrire les sous-objectifs en cours de satisfaction
6. : Le sujet doit être clairement localisable (en début de phrase généralement)
7. : Rester concis et pertinent (éviter les longs documents)
8. : Éviter les si, et placer les comportements alternatifs dans les extensions
9. : Signaler les sous-cas d'utilisation (représentés par la relation d'inclusion)
10. : Identifier le bon objectif
11. : Signaler la portée
12. : Laisser de côté l'interface utilisateur

3.3.2 Représentation textuelle de cas d'utilisation

La représentation textuelle des cas d'utilisation, présentée ci-dessous, est couramment utilisée. Elle permet de donner une description plus détaillée du comportement d'un cas d'utilisation avec un format de présentation textuelle à la fois souple et riche. Elle contient les éléments suivants :

- Le nom du cas d'utilisation
- Description
- L'acteur primaire
- Le système concerné par le cas d'utilisation
- L'ensemble des acteurs intervenants dans le cas d'utilisation
- Le niveau du cas d'utilisation :
  - objectif de l'acteur principal
  - sous-fonction
- Les conditions d'exécution du cas d'utilisation
- Les opérations du scénario principal
- Les extensions

3.4 Exemple : Formaperm

Nous développons ici les cas d'utilisation de FormaPerm.

3.4.1 La scolarité

Reprenons le cas de la scolarité. Nous avons décrit 5 cas d'utilisation sans aucun détail. Cependant nous pouvons préciser que pour réaliser chacune des opérations décrites dans le diagramme de cas d'utilisation 3.4 il est nécessaire que la personne de la scolarité qui réalise l'opération soit connectée. Ce qui nous donne le diagramme suivant 3.6 . La relation d'inclusion est utilisée pour préciser que chaque opération nécessite au préalable une connexion de la part de l'utilisateur.

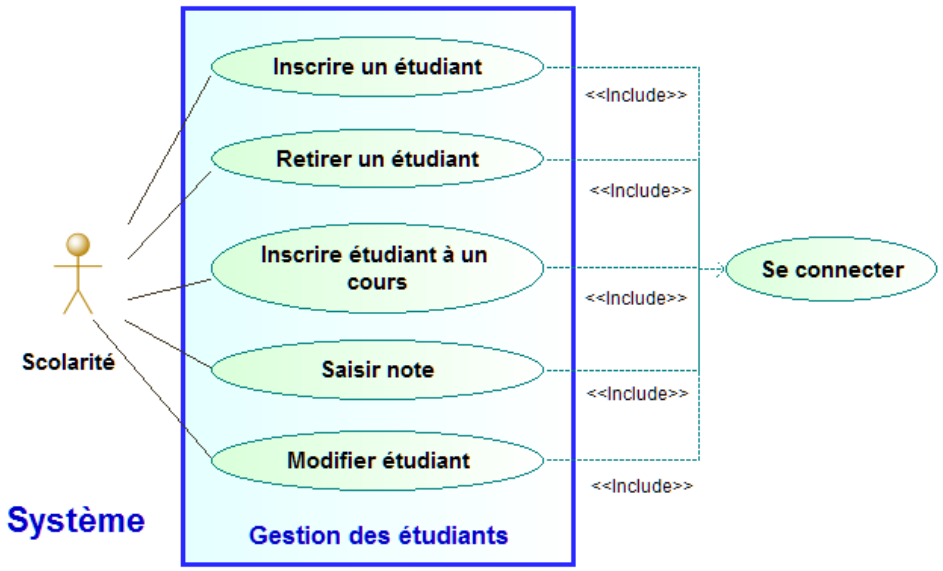

Figure 3.6 : Connexion de la scolarité

3.4.2 L'enseignant

Un enseignant doit pouvoir créer un cours, ajouter des documents à un cours, modifier la description d'un cours, consulter la liste des étudiants d'un cours, consulter la liste de ses cours, choisir un de ses cours, ... Lorsqu'il consulte la liste des étudiants il peut imprimer deux types de listes : une liste d'appel et une liste avec les notes des étudiants.

3.4.2.1 Premier DCU Enseignant

Proposons un premier cas d'utilisation avec les grandes fonctions pour l'enseignant. L'enseignant peut créer un cours, ajouter des documents à un cours, modifier la description d'un cours, consulter la liste des étudiants d'un cours, consulter la liste de ses cours, choisir un de ses cours. Nous obtenons le premier DCU 3.7 .

3.4.2.2 Deuxième DCU Enseignant

Nous allons maintenant ajouter dans notre DCU le fait que l'enseignant doit se connecter pour pouvoir utiliser chacune de ces fonctions (3.8).

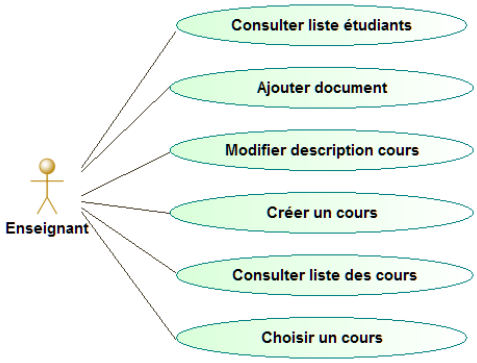

Figure 3.7 : Premier DCU Enseignant

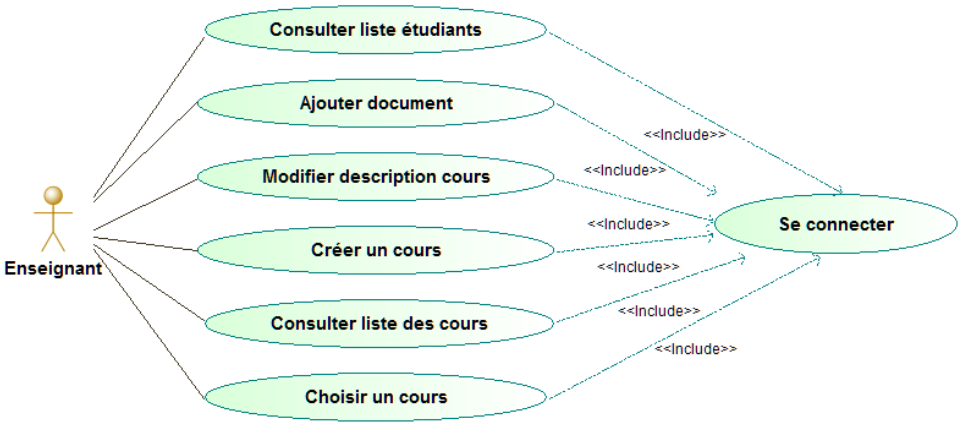

Figure 3.8 : DCU Enseignant avec la connexion

3.4.2.3 Troisième DCU Enseignant

Nous nous intéressons ici à la fonction `"Ajouter document"` qui suppose dans certains cas de télécharger sur la plateforme le document.

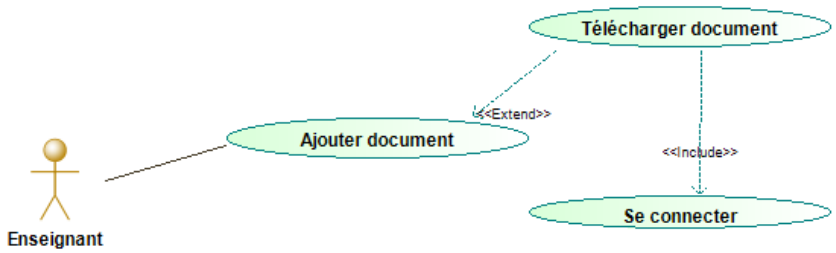

Figure 3.9 : Téléchargement de document

3.4.2.4 Quatrième DCU Enseignant

Intéressons nous maintenant à l'impression des listes d'appel et de notes. Ces deux opérations ne sont pas obligatoires, dans certains cas, lors de la consultation de la liste d'étudiant l'enseignant pourra imprimer une liste. On utilise donc ici une relation

d'extension. Par ailleurs nous considérons ici que les deux opérations d'impression héritent d'une mêm fonction d'impression de liste (cf. 3.10).

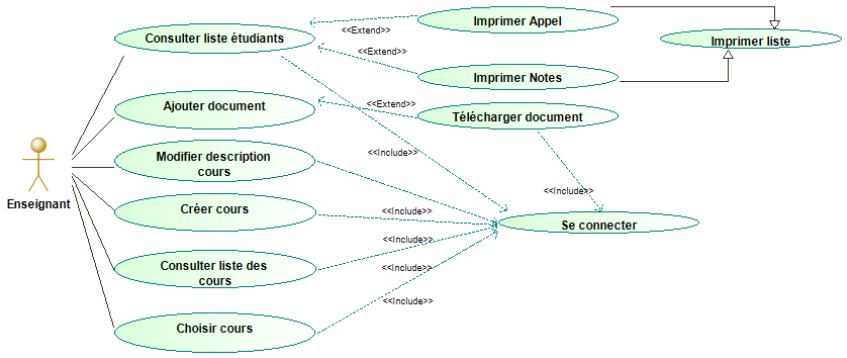

Figure 3.10 : DCU Enseignant : Impression des listes

3.4.2.5 Enfin ...

En reprenant le diagramme on peut aussi voir que certaines fonctions supposent le choix d'un cours. On arrive ainsi au diagramme complet 3.11 .

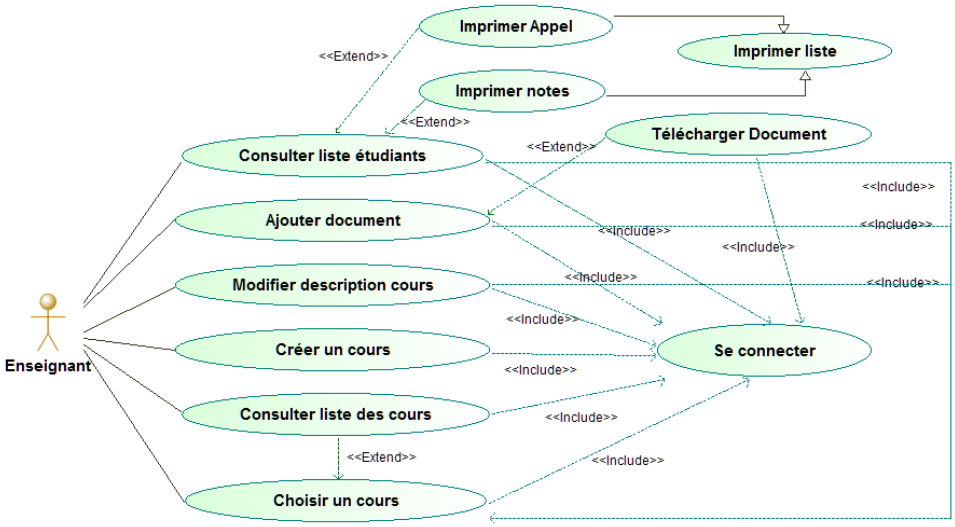

Figure 3.11 : DCU Enseignant complet

3.4.2.6 Représentation textuelle du cas d'utilisation "*Consulter liste étudiants*"

Reprenons le cas d'utilisation *"Consulter liste étudiants"*, nous obtenons le tableau 3.1 .

Cas d'utilisation	**Consultation liste**
Description	L'utilisateur consulte une liste d'étudiants. Il doit d'abord se connecter et ensuite choisir le cours. Enfin il peut si il le souhaite imprimer une liste.
Acteur primaire	Enseignant
Système	Gestion formaPerm
Intervenants	Enseignant, Base de données de login Système
Objectif acteur principal	Consulter une liste
Sous-fonction	Choisir cours
Conditions d'exécution	être connecté
Extensions	
1 2	Imprimer liste d'appel Imprimer liste de notes

Table 3.1 : Représentation textuelle du cas d'utilisation "Consulter liste"

Chapter 4

Les diagrammes de séquence

L'aspect comportemental d'une application orientée objet est défini par la façon dont interagissent les objets qui la composent. L'application réalise ses traitements grâce aux objets qui la compose et leurs interactions. Ces interactions correspondent à des échanges de messages. Le diagramme de séquence permet de représenter les interactions entre objets.

4.1 Introduction

Le **diagramme de séquence** est un diagramme d'interaction qui capture l'ordre des interactions entre les différentes parties du système.

Dans une application, chaque objet peut envoyer et recevoir des messages des autres objets qui composent l'application. En UML, les objets qui participent à une interaction, s'échangent des messages entre eux.

Un diagramme de séquences permet de décrire comment les **éléments du système** interagissent entre eux et avec les **acteurs** selon un point de vue temporel. Les objets au coeur d'un système interagissent entre eux en échangeant des **messages**. Les acteurs interagissent avec le système au moyen d'interfaces homme-machine ou IHM.
Pour spécifier de façon complète une interaction plusieurs diagrammes UML doivent être définis :

- Cas d'utilisation
- Classes
- Séquences

Les diagrammes de séquences mettent l'accent sur l'expression des interactions avec la chronologie des envois de messages. Ils peuvent servir à illustrer un cas d'utilisation en mettant l'accent sur la chronologie des opérations en interaction avec les objets. Ils font partie des diagrammes les plus importants d'UML en terme de **vue dynamique du système**.
En général on propose pour décrire un système plusieurs diagrammes de séquences. Chaque diagramme de séquence détaille une fonction du système en décrivant les interactions entre différents participants de façon séquentielle.

4.2 Concepts de base

4.2.1 Les participants

Un diagramme de séquence est composé d'un ensemble de **participants** : les parties du système qui interagissent les unes avec les autres pendant la séquence.
Nous distinguons différents types de participants : les objets qui sont une **instance d'une classe** et des objets sans classe dits **objets non typés** qui sont utilisés pour simplement demander la réalisation d'opérations. Les objets non typés sont en général les acteurs du système. Parmi les objets qui participent à une interaction certains peuvent ne pas avoir d'identifiant (objet utilisé une seule fois) ; ils sont appelés **objets anonymes**.
Les participants d'un diagramme de séquences sont nommés généralement de la façon suivante :

```
[nom_du_role] : [nom_du_type]
```

Au moins un des deux noms doit être défini et les ':' sont obligatoires.

Une **ligne de vie** représente un participant à une interaction que ce soit un objet ou un acteur. Cette ligne de vie est représentée par une ligne verticale pointillée et les noms des participants sont spécifiés par des rectangles contenant le nom du participant comme le montre l'exemple de la figure 4.1 .

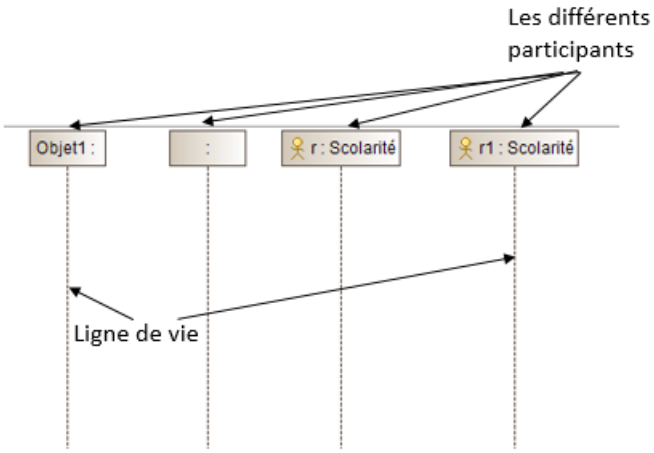

Figure 4.1 : Lignes de vie et participants

Une **zone d'activation** ou **barre d'activité** montre qu'un participant est actif. La ligne de vie, ligne en pointillé, est remplacée par une barre, comme le montre la figure 4.2 .

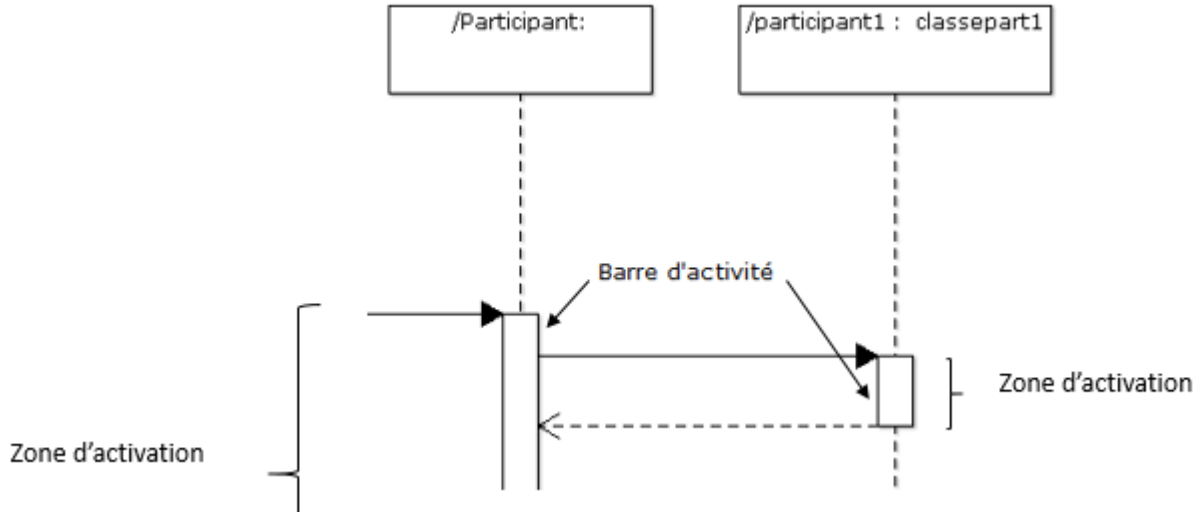

Figure 4.2 : Barre d'activité

4.2.2 Le temps

Le **temps** est un facteur essentiel du diagramme de séquences, en effet il permet de décrire l'ordre dans lequel les interactions ont lieu.
Il s'écoule de haut en bas selon l'axe vertical du diagramme et la disposition des participants sur l'axe horizontal n'a pas d'effet. Le placement des interactions du haut vers le bas dans le diagramme indique l'ordre dans lequel elles se réalisent. Le diagramme décrit l'ordre dans lequel les interactions ont lieu mais ne donne aucune information sur la durée de ces interactions.

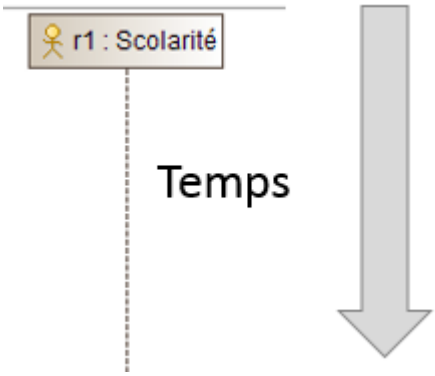

Figure 4.3 : Le temps dans un diagramme de séquence

4.2.3 Les messages

Un message définit une communication particulière entre des participants. Plusieurs types de messages existent dont les suivants :

- l'envoi d'un signal,
- la demande de réalisation d'une opération,

– la création ou la destruction d'un participant.

La plus petite partie d'une interaction est **l'événement**. Il représente la brique de base des messages, tout point d'une interaction où quelque chose se produit, comme le montre la figure 4.4 . Les messages d'un diagramme de séquences sont représentés par une flèche

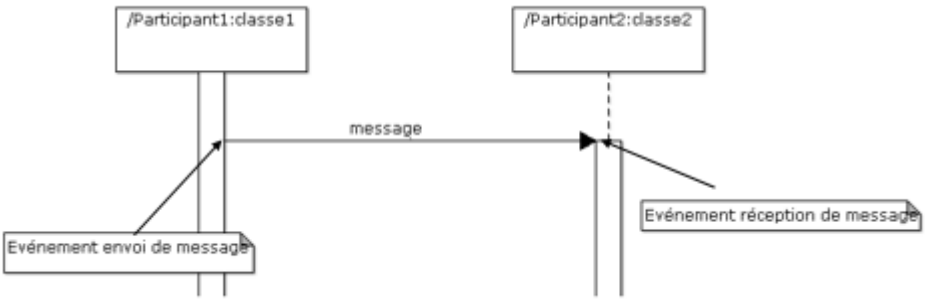

Figure 4.4 : Message et événement

partant du participant émetteur, dans la figure *'Participant1'*, et arrivant vers le participant destinataire, dans la figure *'Participant2'*.
Il est possible de numéroter les messages séquentiellement ; le premier message étant numéroté 1. Si un nouveau message est envoyé alors que le traitement associé au message précédent n'est pas terminé, on peut utiliser une numérotation composée comme le montre la figure 4.5 . On parle alors de messages imbriqués. Il existe différents types d'envois de

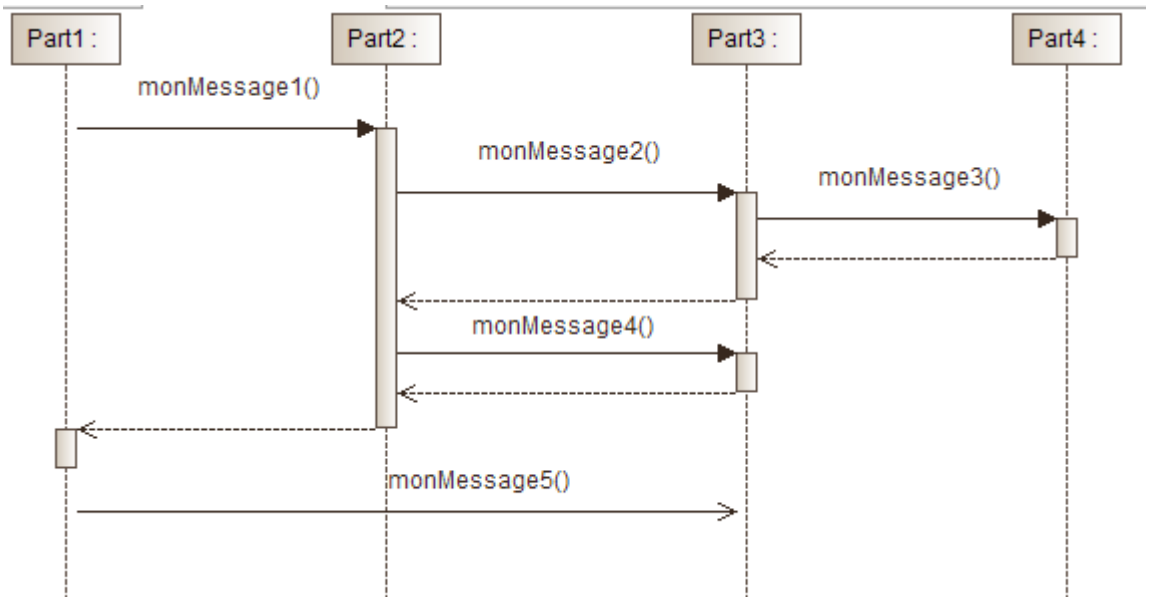

Figure 4.5 : Diagramme de séquence avec plusieurs messages

message : synchrone, asynchrone et de retours, qui sont représentés graphiquement en UML comme dans le schéma 4.6 . L'invocation d'une opération est le type de message le plus utilisé en programmation objet. L'invocation peut être asynchrone ou synchrone.

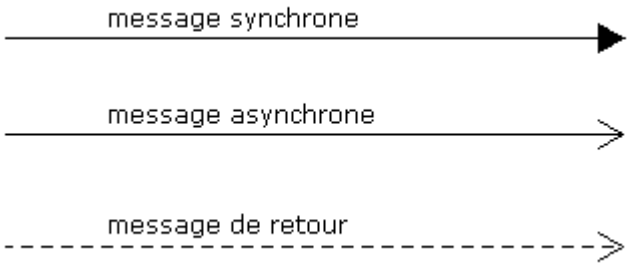

Figure 4.6 : Les différents types d'envoi de message

4.2.3.1 Signature de messages

Un message est décrit par une signature qui a la forme suivante :

```
variable = nom_message(arguments) : classe_retour
```

Il peut contenir un nombre quelconque d'arguments, séparés par des virgules. Chaque argument est présenté sous la forme `<nom>:<classe>`. On peut avoir ainsi des messages de différentes formes comme ceux présentés dans le tableau 4.1 .

Exemple de signature	Explication
`MonMessage()`	Nous avons à faire à un message qui se nomme `MonMessage` sans autre information.
`MonMessage(arg1 : classe1, arg2 : classe2)`	Le nom du message est `MonMessage` et il a 2 arguments `arg1` et `arg2` respectivement objet des classes `classe1` et `classe2`
`MonMessage() : ClasseRetour`	Le nom du message est `MonMessage`, il n'a pas d'argument et renvoie un objet de classe `classeRetour`
`res = MonMessage() : ClasseRetour`	Le nom du message est `MonMessage`, il n'a pas d'argument et renvoie un objet de classe `classeRetour` affecté à la la variable `res`

Table 4.1 : Exemple de signatures de message

4.2.3.2 Messages synchrones

Lors de l'invocation d'un message synchrone, l'émetteur reste en attente le temps que dure l'invocation de l'opération, avant de continuer son activité. Il attend une réponse du récepteur. Graphiquement, un message synchrone se représente par une flèche en traits pleins et à l'extrémité pleine partant de la ligne de vie de l'objet expéditeur et allant vers celle de l'objet destinataire. Ce message est suivi d'une réponse qui se représente par une flèche en pointillé (cf. figure 4.7).

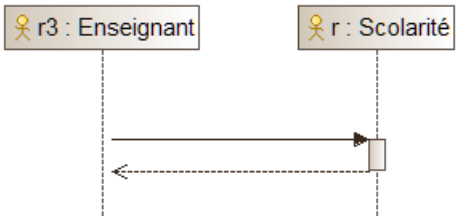

Figure 4.7 : Message synchrone

4.2.3.3 Messages asynchrones

Dans ce cas l'émetteur n'attend pas de retour du destinataire pour continuer son activité. Il peut ainsi invoquer d'autres messages avant un retour du message asynchrone. On rencontre ce type de message lorsque les différents participants du système peuvent fonctionner en parallèle (systèmes multithreads).

4.2.3.4 Messages de création et de destruction de participant

Ces messages permettent de gérer le cycle de vie des objets qui participent à une interaction. Les objets peuvent exister au début de l'interaction ou être créés pendant l'interaction par d'autres objets. La création d'un objet est représentée par un message spécifique qui donne lieu au début de la ligne de vie de l'objet créé comme le montre la figure 4.8 .
Un objet participant à une interaction peut aussi détruire un autre objet. L'objet détruit ne peut plus alors recevoir de messages. Le message de destruction d'objet est représenté par une croix (Figure 4.8).

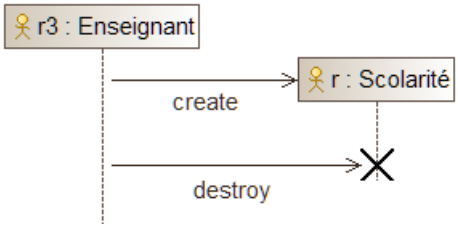

Figure 4.8 : Messages création et destruction

4.2.4 Les fragments d'interaction

Dans les versions précédentes à UML 1.4 il n'était pas possible de composer les interactions. Cependant, pour décrire des interactions complexes et réutiliser certaines interactions, cela posait problème. Il n'existait ainsi, aucun moyen standard pour représenter des boucles ou des flux alternatifs ; ce qui aboutissait à des diagrammes de séquences de taille et de complexité trop importantes. Dans UML 2.0 le concept de **fragment d'interaction** ou **fragment de séquence** est introduit. Il représente des compositions d'interactions et permet de décrire des diagrammes de séquence de manière compacte.
Un fragment d'interaction est défini par un opérateur et des opérandes il est représenté par un rectangle qui comprend une partie du diagramme de séquences (cf. figure 4.9). Dans

le coin supérieur gauche on a une étiquette qui donne le nom de l'opérateur.

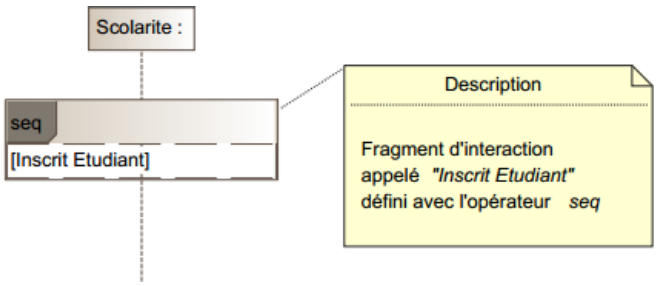

Figure 4.9 : Exemple de fragment d'interaction

Il existe 13 opérateurs définis dans la notation UML2.0. La composition de ces fragments permet de reconstituer simplement la complexité d'un système en décomposant une interaction complexe en fragments suffisamment simples pour être compris. Un fragment se présente sous la même forme qu'une interaction, c'est à dire sous la forme d'un rectangle dont le coin supérieur gauche contient un pentagone.

On a les opérateurs suivants :

- de choix et de boucle : `alternative, option, break` et `loop`
- contrôle de l'envoi de messages en parallèle : `parallel` et `critical region`
- contrôle de l'envoi de messages : `ignore, consider, assertion` et `negative`
- qui fixent l'envoi de messages : `weak sequencing` et `strict sequencing`
- qui appelle une séquence d'interaction : `ref`

4.2.4.1 Opérateur Alternative - `alt`

L'opérateur `"alt"` permet de spécifier qu'un ensemble d'interactions ne sera exécuté que sous certaines conditions. Il est similaire à la forme algorithmique `"Si ... Alors ... Sinon ..."`. Une seule des deux branches (`Alors, Sinon`) sera exécutée lors de la réalisation d'un scénario.
Considérons l'exemple suivant (cf. figure 4.10): un utilisateur essaie de se connecter à un serveur. Il a droit à 2 essais. Si au premier essai il donne le bon mot de passe il peut travailler sur le serveur, dans le cas contraire le système lui redemande un mot de passe. Si celui-ci est à nouveau incorrect, le serveur refuse la demande de connexion.

4.2.4.2 Opérateur Option - `opt`

L'opérateur `"opt"` permet de spécifier, comme pour l'opérateur `"alt"`, qu'un ensemble d'interactions ne sera exécuté que sous certaines conditions. Mais ici il n'y a pas d'alternative (`Sinon`).

4.2.4.3 Opérateur Break - `break`

L'opérateur break permet de représenter un fragment correspondant à des scenarii exceptionnels ou de rupture. Le scénario de rupture est exécuté si la condition de garde est satisfaite.

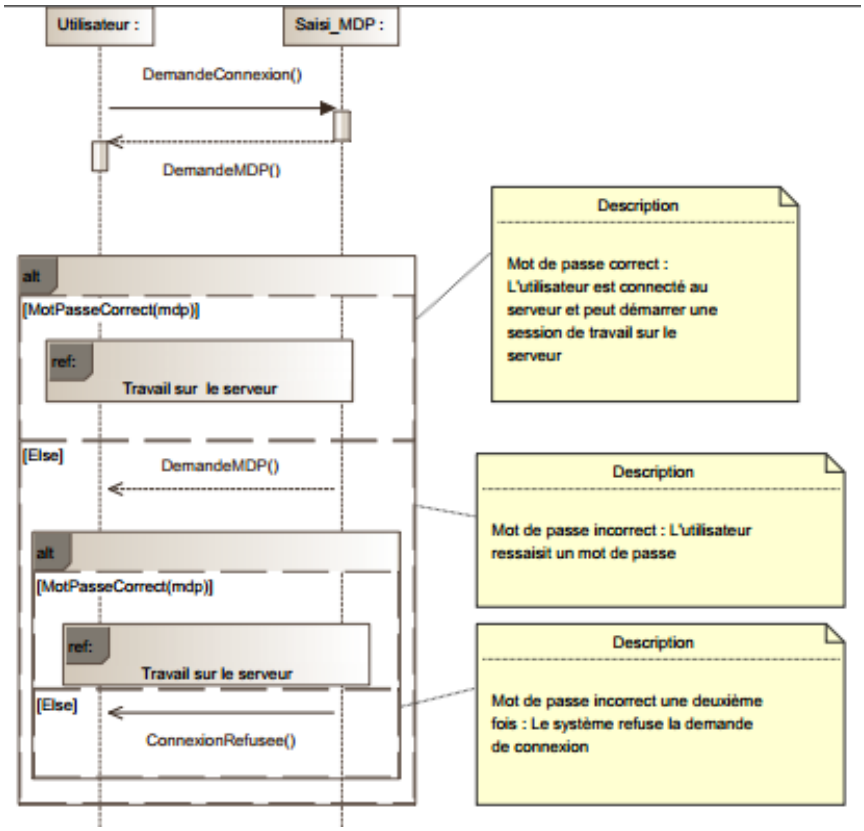

Figure 4.10 : Fragment combiné "alt"

4.2.4.4 Opérateur Loop - `loop`

L'opérateur de boucle : Loop, permet de décrire un ensemble d'interactions qui sera exécuté en boucle. L'ensemble d'interactions sera exécuté tant que la garde associée est vraie.

1. expression logique : touche_tapée $\neq$ `esc`
2. indication du nombre d'itérations : *minimum* et *maximum*

Elle est notée entre crochets dans la partie garde du fragment `loop`. On peut ainsi avoir les gardes suivantes :

- [4] : l'ensemble d'interactions est exécuté 4 fois ;
- $[3, 7]$: l'ensemble d'interactions est exécuté au moins 3 fois et au maximum 7 fois.

4.2.4.5 Opérateur de traitements parallèles - `par`

Cet opérateur permet de décrire plusieurs (au moins deux) ensembles d'interactions qui seront exécutés en parallèle. Dans le fragment d'interaction par ces différents ensembles sont séparés par des pointillés comme le montre la figure 4.11 . Dans l'exemple les traitements *"Traitement1"* et *"Traitement2"* pourront être exécutés en parallèle.

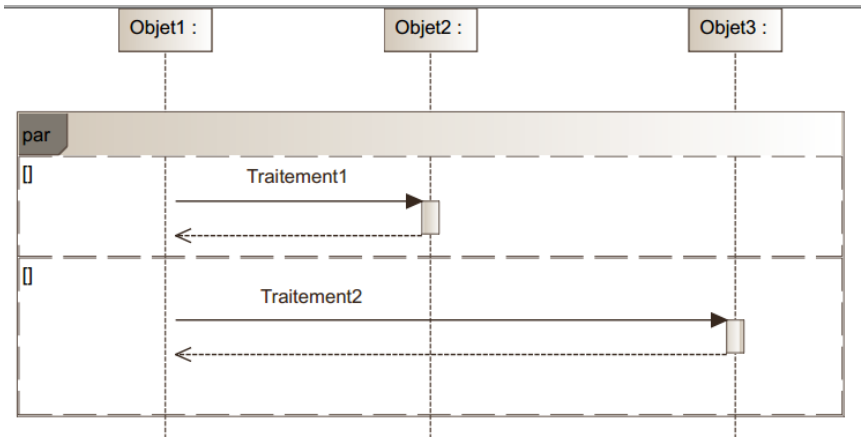

Figure 4.11 : Fragment combiné par

4.2.4.6 Opérateur critical - `critical`

Il permet d'exprimer qu'un ensemble d'interactions ne peut pas être interrompu. Il dénote le caractère critique des traitements réalisés. On considère que l'ensemble d'interactions est *atomique*. Dans l'exemple 4.12 , les 3 interactions seront exécutées sans aucune interruption.

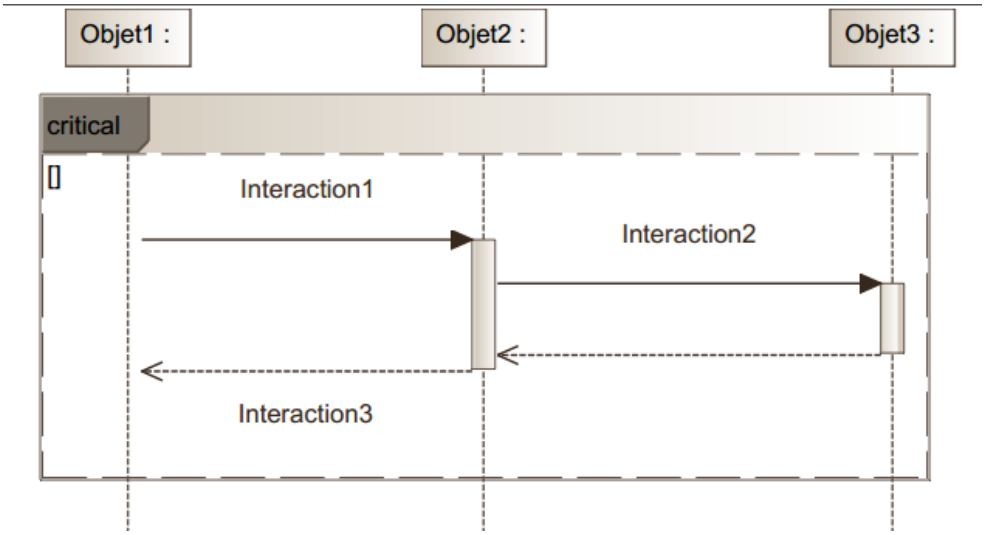

Figure 4.12 : Fragment combiné critical

4.2.4.7 Opérateur ignore - `ignore`

Cet opérateur indique qu'il existe des **messages** qui sont *facultatifs* dans le fragment combiné. Les messages peuvent être qualifiés d'insignifiants : intuitivement, ce sont des interactions que l'on ne prend pas en compte. On peut aussi interpréter l'opérateur `"ignore"` comme désignant des interactions pouvant intervenir à tout moment dans le flot des interactions du diagramme de séquence.

4.2.4.8 Opérateur consider - `consider`

Au contraire, l'opérateur `"consider"` (considérer) désigne les interactions à prendre en compte dans la séquence. Il spécifie une liste des messages que le fragment décrit. D'autres messages peuvent se produire dans le système en cours d'exécution, mais ils ne sont pas significatifs quant aux objectifs de cette description.

4.2.4.9 Opérateur assertion - `assert`

Ce fragment indique que l'ensemble d'interactions est une assertion, c'est à dire l'unique séquence possible. Ce type de fragment combiné est souvent utilisé avec les opérateurs *ignore* et *consider*.

4.2.4.10 Opérateur negative - `neg`

Cet opérateur indique qu'un ensemble d'interactions est invalide. Lorsque l'on exécute cet ensemble d'interaction une erreur est déclenchée.

4.2.4.11 Opérateur weak sequencing - `weak` et strict sequencing - `strict`

Les opérateurs strict et weak permettent de représenter une série d'interactions dont certaines s'opèrent sur des objets indépendants : L'opérateur strict est utilisé quand l'ordre d'exécution des opérations doit être strictement respecté. L'opérateur weak est utilisé quand l'ordre d'exécution des opérations n'a pas d'importance. Considérons l'exemple 4.13 qui spécifie que les messages mess1, mess2, mess3, mess4, mess5 et mess6 doivent être exécutés dans cet ordre puisqu'ils font partie du fragment d'interaction strict.

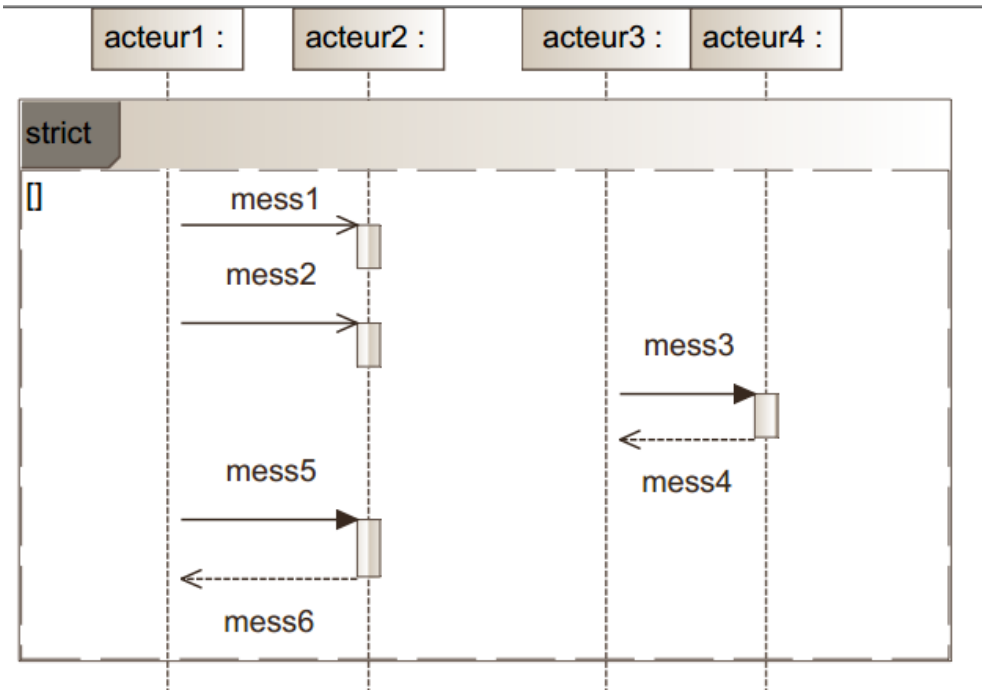

Figure 4.13 : Utilisation d'un fragment "strict"

4.2.4.12 Opérateur d'utilisation d'interaction - `ref` ou `seq`

L'opérateur *ref* permet d'appeler une séquence d'interactions décrite par ailleurs et constituant ainsi une sorte de sous-diagramme de séquence, comme le montre la figure 4.9 . On trouve ici deux notations pour cet opérateur `seq` ou `ref`.

4.3 Exemple : Formaperm

Considérons ici la gestion des notes pour un cours donné. Une fois les copies corrigées, l'enseignant envoie à la scolarité un mail avec le nom du cours et l'ensemble des notes.
Pour saisir les notes, la personne de la scolarité doit se connecter (nous ne détaillerons pas ici cette opération), elle peut alors choisir le cours pour lequel elle souhaite réaliser la saisie. Le système lui renvoie alors la liste des étudiants concernés.
La saisie des notes peut alors commencer, pour chaque étudiant de la liste, la scolarité saisit une note. Lors de la saisie des notes, un étudiant peut ne pas avoir de note, dans ce cas la scolarité ajoute le nom de l'étudiant à sa liste de problème.
Lorsque la saisie est finie, la scolarité envoie un message à l'enseignant pour lui demander de valider les notes et traiter les problèmes détectés. L'enseignant transmet la validation à la scolarité. La scolarité rend alors publiques les notes, le système envoie alors un message à l'enseignant, à chaque étudiant du cours et à la scolarité.

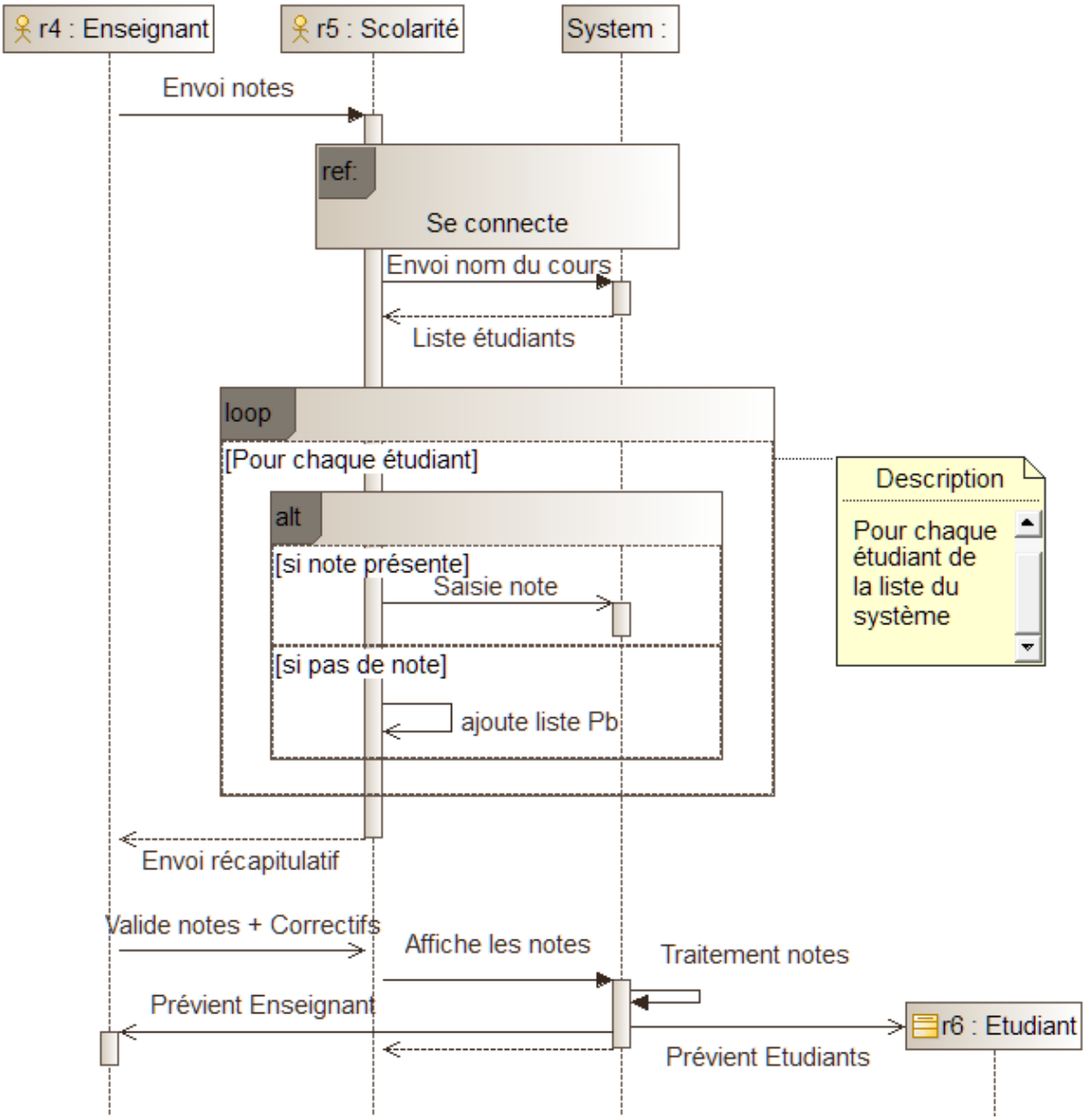

Figure 4.14 : DDS : Saisie des notes

Part III

Annexes

Bibliography

Les références données dans cette page correspondent à des supports qui ont été utilisés lors de la conception de ce cours.
Il n'y a aucun besoin de consulter ces documents pour atteindre les objectifs du module. Par contre ces supports peuvent être intéressants pour les étudiants qui souhaitent aller plus loin.

- Alistair COCKBURN *Writing Effective Use Cases*. 1999
https://www.researchgate.net/publication/31757301_Writing_Effective_Use_Cases_A_Cockburn

- Alistair Cockburn *Rédiger des cas d'utilisation efficaces*. Eyrolles, Collection : Technologies objet, 2001

- Pierre Alain MULLER Nathalie GAERTNER *Modélisation Objet avec UML*. .Eyrolles 2002.
Présentation du langage UML. Cette présentation simple et très bien illustrée d'UML s'arrête à la version 1.3 du standard UML.

- Russ MILES Kim HAMILTON *Introduction à UML 2.* O'Reilly 2006.
Une introduction pragmatique à UML.

- OMG *OMG Unified Modeling Language TM (OMG UML), Version 2.5 (Spécifications de UML version 2.5.)*. http://www.omg.org/spec/UML/2.5/PDF/

- Christian SOUTOU *UML2 pour les bases de données.* Eyrolles 2007.
Une introduction à UML pour la conception de bases de données relationnelles.