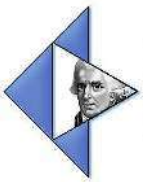

Université de Corse
Faculté des Sciences et Techniques

Méthode d'analyse orientée objet UML

Notes de cours

F. Bernardi, 2002

bernardi@univ-corse.fr

http://spe.univ-corse.fr/bernardiweb/cours.htm

Introduction au cours

Diagrammes statiques

Diagrammes dynamiques

Design Patterns

Bibliographie et liens Internet

Booch G., J. Rumbaugh et I. Jacobson[1], *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999

Booch G., J. Rumbaugh et I. Jacobson, *Le guide de l'utilisateur UML*, Eyrolles, 2000

Conallen J. , *Concevoir des applications Web avec UML*, Eyrolles, 2000

D'Souza D.F. et Wills A.C., *Objects, Components and Frameworks with UML*, Addison-Wesley, 1999

Gamma E., E. Helm, R. Johnson et J. Vlissides[2], *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

Graham, I., *Object-Oriented Methods, Principles and Practice*, Addison-Wesley, 2001

Larman C., *UML et les Design Patterns*, CampusPress, 2002

Larman C., *Applying UML and Patterns*, Prentice Hall, 2002

Muller P.A. et N. Gaertner, *Modélisation objet avec UML*, Eyrolles, 2000

Oestereich B., *Developing Software with UML*, Addison-Wesley, 2001

Stevens P. et R. Pooley, *Using UML, Software Engineering with Objects and Components*, Addison-Wesley, 2000

`http://www.uml.org`

`http://www.rational.com/uml/index.jsp`

`http://researchweb.watson.ibm.com/designpatterns/publications.htm`

`http://www.objectmentor.com/resources/index`

`http://uml.free.fr`

`http://www-iiuf.unifr.ch/~schweizp/oopintro/concept.html`

[1] Initiateurs du langage. Cet ouvrage est le plus complet sur UML, mais il est malheureusement assez difficile d'accès.

[2] Connus sous le nom de « The Gang of Four » ou « GoF ». Cet ouvrage est considéré comme la Bible du développeur objet.

Table des matières

CHAPITRE 1

Introduction au cours

URLs à consulter :

`http://www.commentcamarche.net/poo/poointro.php3`,

`http://dept-info.labri.u-bordeaux.fr/~griffaul/Enseignement/POO/Cours/cours/cours.html`.

1.1 Présentation d'UML (Unified Modeling Language)

☞ *Modèle:* simplification de la réalité permettant de mieux comprendre le système que l'on développe.

La construction de modèles (appelée modélisation) a quatre objectifs principaux :

- les modèles aident à visualiser un système tel qu'il est ou tel que nous voudrions qu'il soit ;
- les modèles permettent de préciser la structure ou le comportement d'un système ;
- les modèles fournissent un canevas qui guide la construction d'un système ;
- Les modèles permettent de documenter les décisions prises.

Les quatre principes de la modélisation :

1. Premier Principe : « le choix des modèles à créer a une très forte influence sur la manière d'aborder un problème et sur la nature de sa solution ».
2. Second Principe : « tous les modèles peuvent avoir différents niveaux de précision ».
3. Troisième Principe : « Les meilleurs modèles ne perdent pas le sens de la réalité ».
4. Quatrième principe : « il est préférable de décomposer un système important en un ensemble de petits systèmes presque indépendants ».

Dans le cadre de la conception orientée objet, un langage unifié pour la modélisation a été développé : UML (« Unified Modeling Language »). Il s'agit d'un langage graphique de modélisation objet permettant de spécifier, de construire, de visualiser et de décrire les détails d'un système logiciel. Il est issu de la fusion de plusieurs méthodes dont « Booch » et « OMT » et est adapté à la modélisation de tous types de systèmes. La modélisation d'un système s'effectue indépendamment de toute méthode ou de tout langage de programmation.

UML est un langage : il comprend un vocabulaire et un ensemble de règles centrés sur la représentation conceptuelle et physique d'un système logiciel. Ses domaines d'utilisation sont :

- visualisation d'un système ;
- spécification d'un système ;
- construction d'un système ;
- documentation d'un système.

1.2 Le modèle conceptuel d'UML

Le modèle conceptuel d'UML comprend les notions de base génériques du langage. Il définit trois sortes de « briques » de base :

- les éléments, qui sont les abstractions essentielles à un modèle ;
- les relations, qui constituent des liens entre ces éléments ;
- les diagrammes, qui regroupent des éléments et des liens au sein de divers ensembles.

1.2.1 Les éléments

Il existe quatre types d'éléments dans UML :

- les éléments structurels (classe, interface, collaboration,...) ;
- les éléments comportementaux (interaction, automate à états finis) ;
- les éléments de regroupement (package) ;
- les éléments d'annotation (note).

1.2.2 Les relations

Il existe quatre types de relations dans UML :

- la dépendance ;
- l'association ;
- la généralisation ;
- la réalisation.

1.2.3 Les diagrammes

☞ *Diagramme:* représentation graphique d'un ensemble d'éléments et de relations qui constituent un système.

UML définit neuf types de diagrammes divisés en deux catégories :

1. diagrammes statiques (appelés aussi diagrammes structurels) : diagrammes de classes, d'objets, de composants, de déploiements et de cas d'utilisation.
2. diagrammes dynamiques (appelés aussi diagrammes comportementaux) : diagrammes d'activités, de séquences, d'états-transitions et de collaborations.

CHAPITRE 2

Diagrammes de classes (statique)

☞ *Diagrammes de classe:* expriment de manière générale la structure statique d'un système, en termes de classes et de relations entre ces classes.

2.1 Les packages

☞ *Package:* mécanisme d'ordre général qui permet d'organiser les éléments en groupes.

Les packages permettent de définir des sous-systèmes formés d'éléments ayant entre eux une certaine logique. Leurs caractéristiques sont les suivantes :

- Ils regroupent des éléments de modélisation selon des critères purement logiques ;
- Ils permettent d'encapsuler des éléments de modélisation par l'intermédiaire d'interfaces ;
- Ils permettent de structurer un système en catégories ou sous-systèmes ;
- Ils servent de « briques » réutilisables dans la conception d'un logiciel.

Chaque package doit avoir un nom différent de celui des autres packages et peut être composé d'autres éléments, y compris d'autres packages. Les éléments contenus sont en fait « possédés » (au sens UML du terme), ce qui signifie que la destruction d'un package implique la destruction de tous ses éléments.

L'importation permet aux éléments d'un package d'accéder aux éléments d'un autre package. Cette relation est à sens unique et est représentée par une relation de dépendance associée à un stéréotype « import ».

☞ *Stéréotype:* permet de définir une utilisation particulière d'éléments de modélisation existants ou de modifier la signification d'un élément

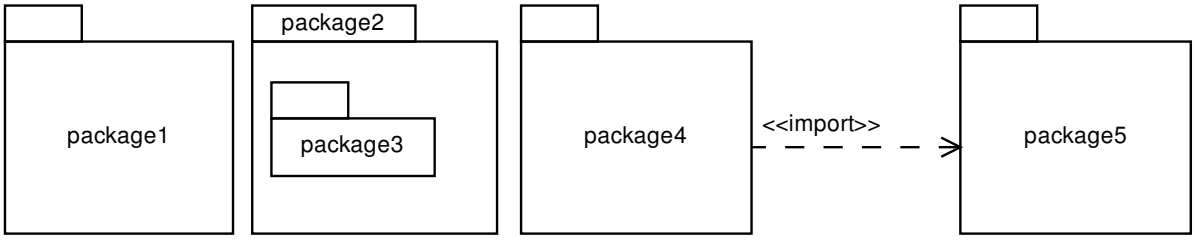

FIG. 2.1 – Représentation graphique d'un package

2.2 Les classes

☞ *Classe:* représentation d'un ensemble d'éléments partageant les mêmes attributs, les mêmes opérations, les mêmes relations et les mêmes sémantiques.

En programmation orientée objet, une classe définit une abstraction, un type abstrait qui permettra d'instancier des objets. Graphiquement, une classe décrite en UML peut être plus ou moins précise :

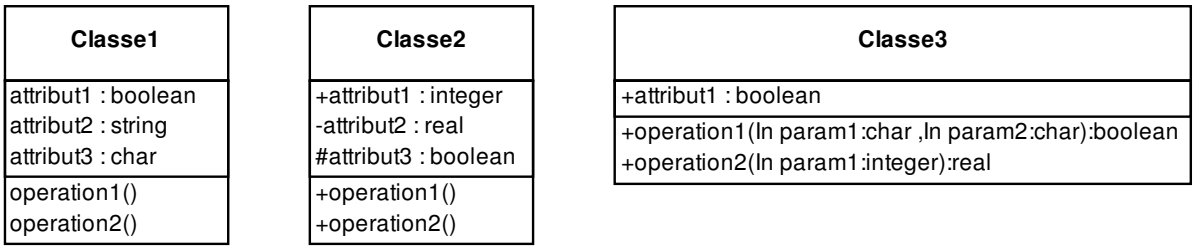

FIG. 2.2 – Représentation graphique d'une classe

La première classe (*classe1*) est dite à visibilité réduite, tandis que les deux autres (*classe2* et *classe3*) sont dites à visibilités détaillées.

☞ *Visibilité d'une caractéristique:* détermine si d'autres éléments peuvent l'utiliser.

Trois niveaux de visibilité sont possibles pour les attributs et les opérations :

– « + » : visibilité *public*. La caractéristique peut être utilisée par n'importe quelle instance ayant une visibilité sur les instances de la classe complète.
– « - » : visibilité *private* : La caractéristique ne peut être utilisée que par des instances de la classe elle-même.
– « # » : visibilité *protected* : La caractéristique ne peut être utilisée que par des instances de la classe elle-même ou bien par les descendants directs de cette classe.

2.3 Les classes abstraites

☞ *Classe abstraite:* classe ne pouvant pas être instanciée directement. Une telle classe sert de spécification pour des objets instances de ses sous-classes.

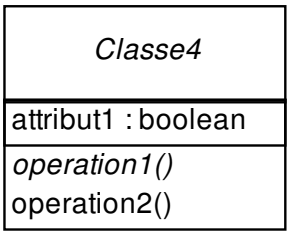

FIG. 2.3 – Représentation graphique d'une classe abstraite

2.4 Les interfaces

☞ *Interface:* décrit un contrat d'une classe ou d'un composant sans en imposer l'implémentation.

Une interface ne décrit aucune structure ni aucune implémentation. Elle ne peut donc pas contenir d'attributs, ni de méthodes fournies sous la forme d'une implémentation.

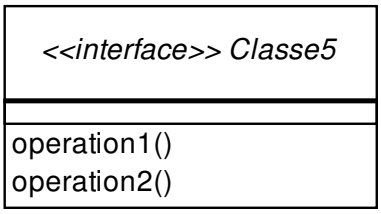

FIG. 2.4 – Représentation graphique d'une interface

2.5 Les relations entre classes

2.5.1 L'association

☞ *Relation d'association:* relation structurelle précisant que les objets d'un élément sont reliés aux objets d'un autre élément.

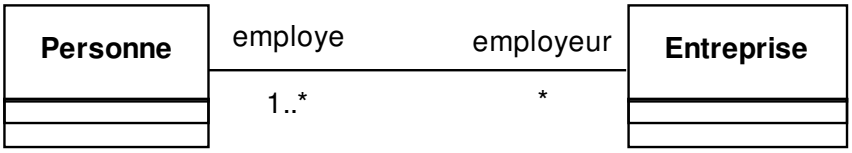

FIG. 2.5 – Représentation graphique d'une association

2.5.2 La dépendance

☞ *Relation de dépendance:* relation sémantique entre deux éléments selon laquelle un changement apporté à l'un peut affecter la sémantique de l'autre.

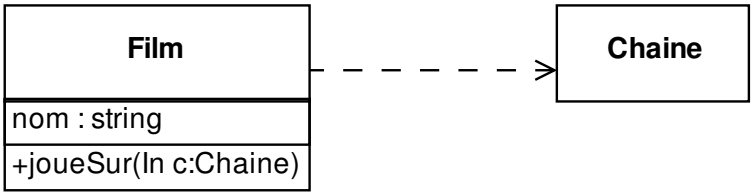

FIG. 2.6 – Représentation graphique d'une dépendance

2.5.3 La généralisation

☞ *Relation de généralisation:* relation entre un élément général et un élément dérivé de celui-ci, mais plus spécifique (désigné par sous-élément ou élément fils).

Le plus souvent, la relation de généralisation est utilisée pour représenter une relation d'héritage.

2.5.4 L'implémentation

☞ *Relation d'implémentation:* relation entre une classe et une interface spécifiant que la classe implémente les opérations définies par l'interface.

2.5.5 L'agrégation

☞ *Relation d'agrégation:* relation «tout-partie» dans laquelle une classe représente un élément plus grand (le «tout») composé d'éléments plus petits (les «parties»).

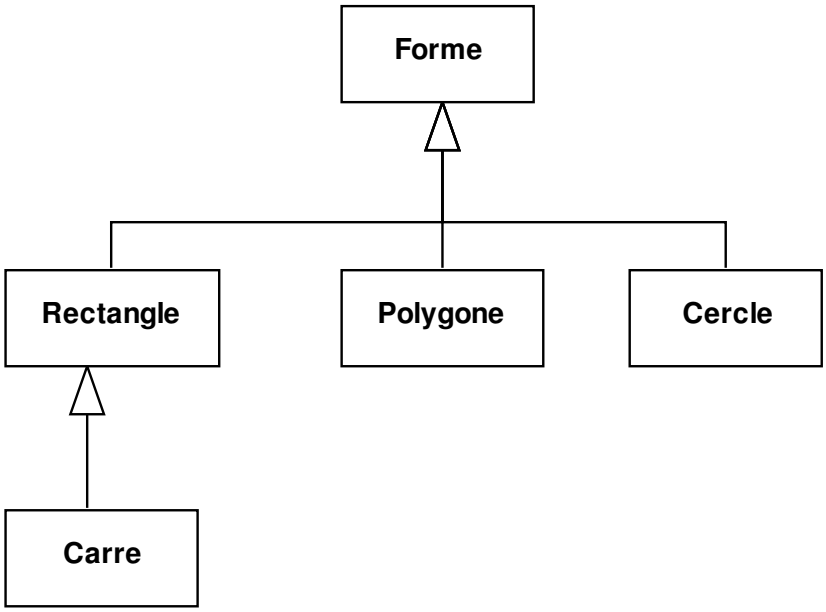

FIG. 2.7 – Représentation graphique d'une généralisation

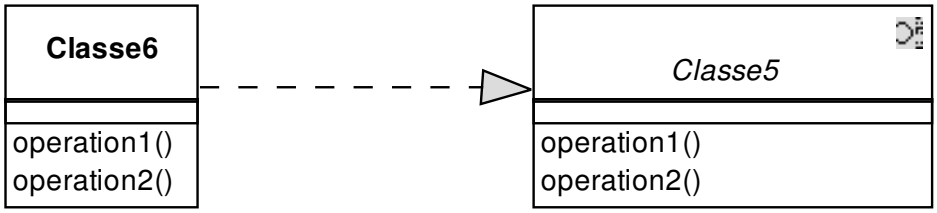

FIG. 2.8 – Représentation graphique d'une implémentation

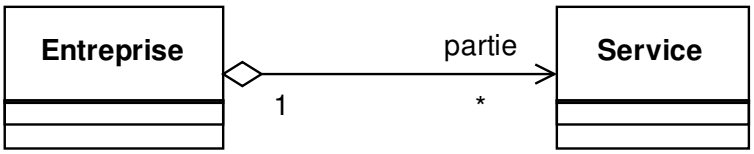

FIG. 2.9 – Représentation graphique d'une agrégation

La relation d'agrégation est souvent implémentée en utilisant des membres privés.

2.5.6 La composition

☞ *Relation de composition:* relation d'agrégation mettant en avant une notion de propriété forte et de coïncidence des cycles de vie. Les «parties» sont créées et détruites en même temps que le «tout».

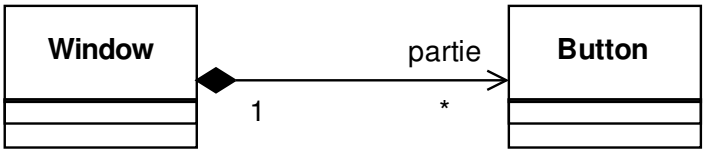

FIG. 2.10 – Représentation graphique d'une composition

CHAPITRE 3

Diagrammes d'objets (statique)

☞ *Diagrammes d'objets:* permet de représenter les relations existant entre les différentes instances des classes à un instant donné de la vie du système.

Dans les diagrammes d'objets, les instances peuvent être anonymes ou nommées.

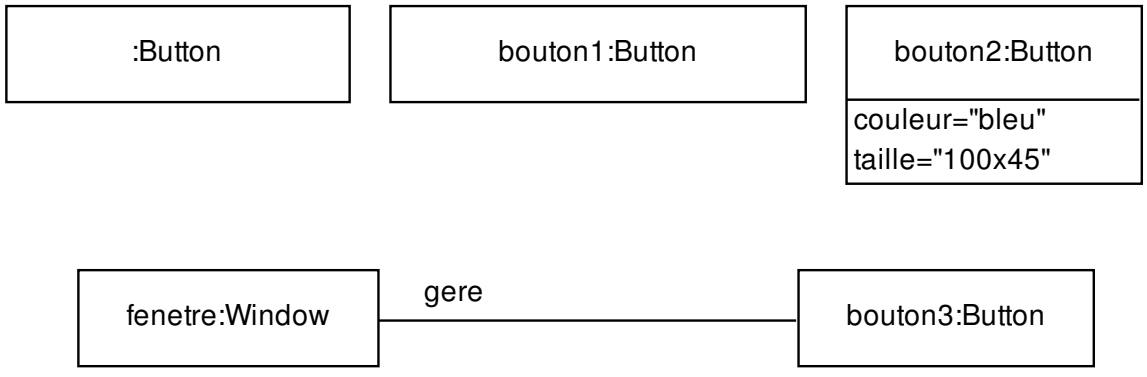

FIG. 3.1 – Un diagramme d'objets

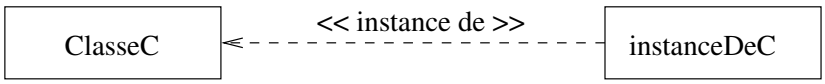

FIG. 3.2 – Autre représentation utilisant un stéréotype

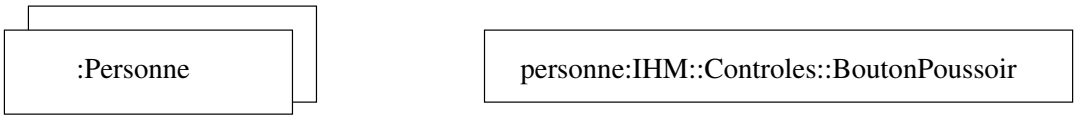

FIG. 3.3 – Groupe d'instances et package

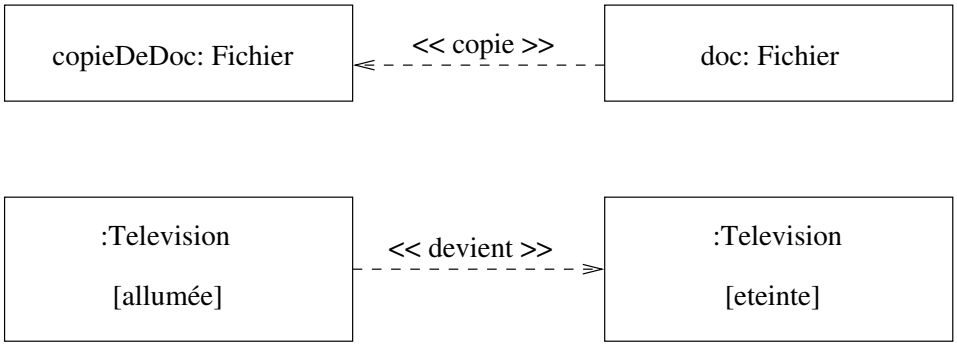

FIG. 3.4 – Création d'un objet à partir des valeurs d'un autre objet

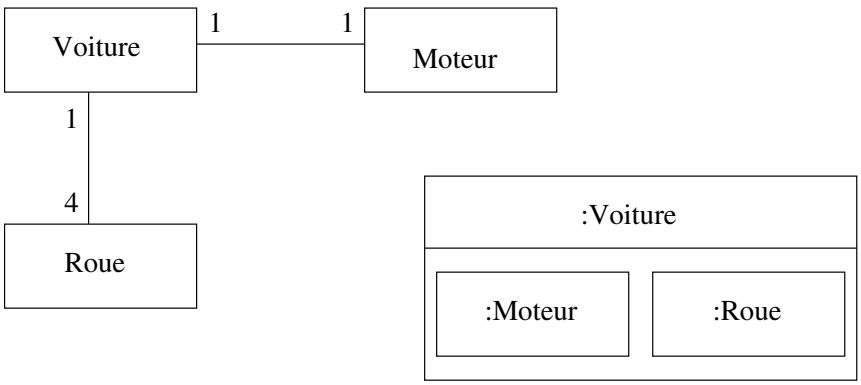

FIG. 3.5 – Objets composites

CHAPITRE 4

Diagrammes de composants (statique)

☞ *Composant:* élément physique qui représente une partie implémentée d'un système. Il peut être du code, un script, un fichier de commandes, etc. Ils présentent un ensemble d'interfaces.

☞ *Interfaces d'un composant:* élément définissant le comportement offert à d'autres composants.

☞ *Diagramme de composants:* permet de décrire les composants et leurs dépendances dans leur environnement d'implémentation.

Les composants peuvent être organisés en sous-sytèmes de packages permettant de masquer la compléxité, par l'encapsulation des détails d'implémentation.

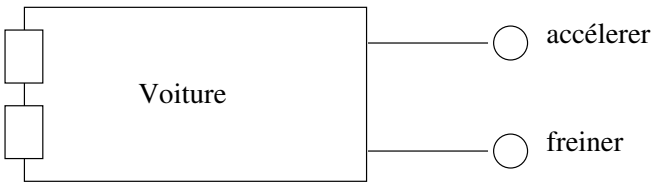

FIG. 4.1 – Représentation d'un composant

☞ *Nœud:* Ressource matérielle du système étudié.

Les instances des composants résident dans des instances de nœuds :

Différents stéréotypes de composants :

<<document>> : un document quelconque ;

<<executable>> : un programme qui peut s'exécuter sur un nœud ;

<<fichier>> : un document contenant du code source ou des données ;

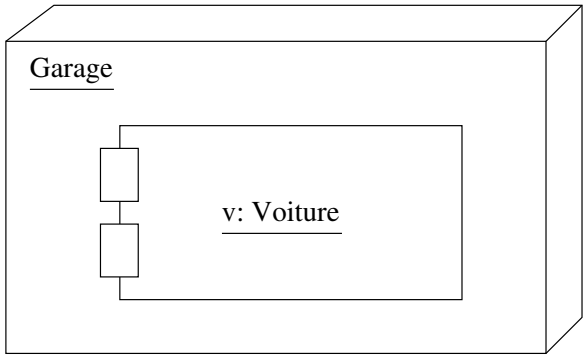

FIG. 4.2 – Instances d'un composant à l'intérieur d'un nœud

<<bibliothèque>> : une bibliothèque statique ou dynamique ;

<<table>> : une table d'une base de données relationnelles.

La majeure partie des relations existantes entre composants est constituée par des contraintes de compilations et d'édition de liens.

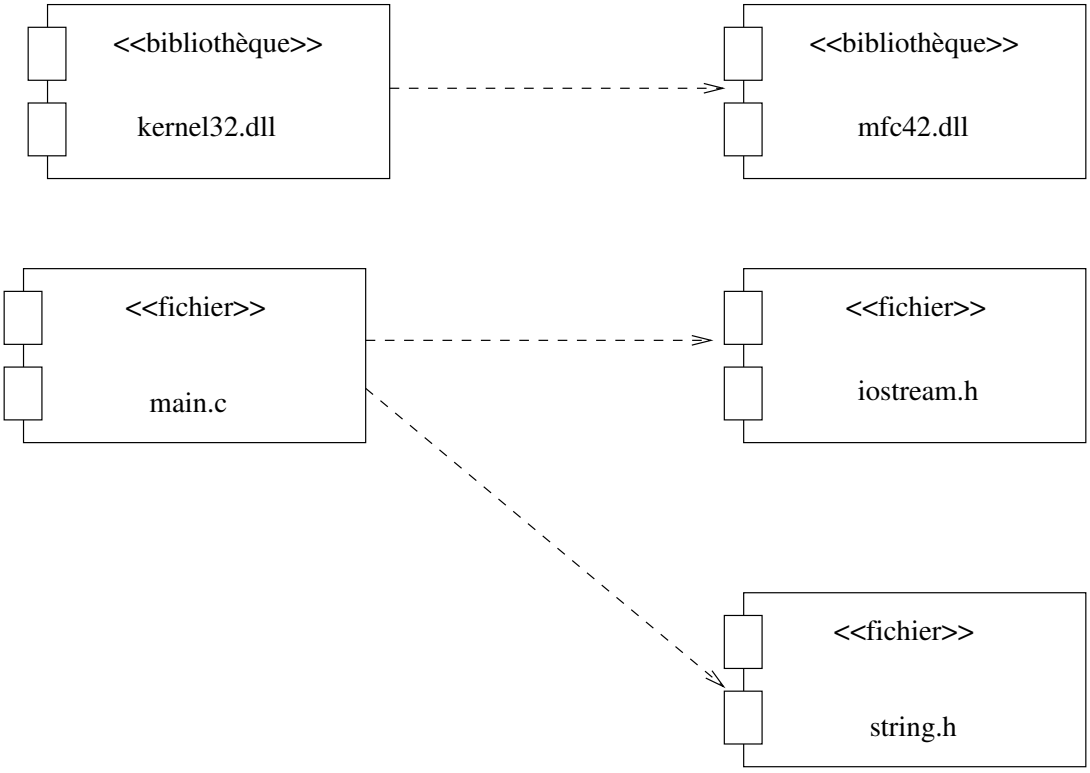

FIG. 4.3 – Dépendances entre composants

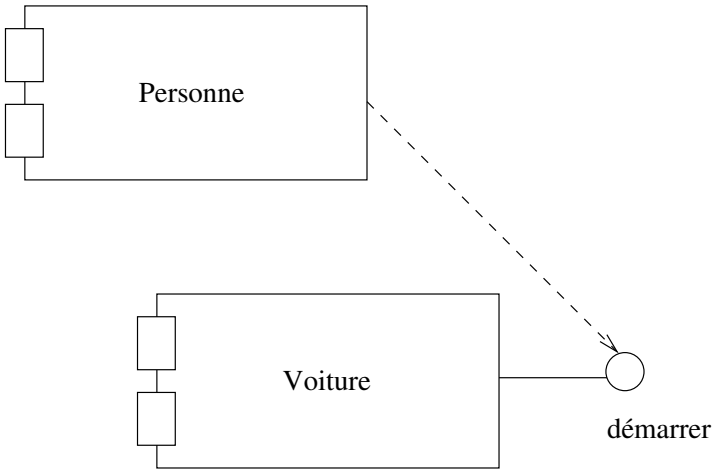

FIG. 4.4 – Dépendances entre composants par l'intermédiaire d'une interface

CHAPITRE 5

Diagrammes de déploiements (statique)

☞ *Diagrammes de déploiement:* permet de montrer la disposition physique des matériels qui composent le système, ainsi que la répartition des composants sur ces matériels représentés par des nœuds.

Ce type de diagramme est utilisé principalement pour la modélisation de trois types de systèmes : les systèmes embarqués, les systèmes client/serveur et les systèmes totalement répartis. Deux notations sont possibles pour montrer qu'un composant réside sur un nœud :

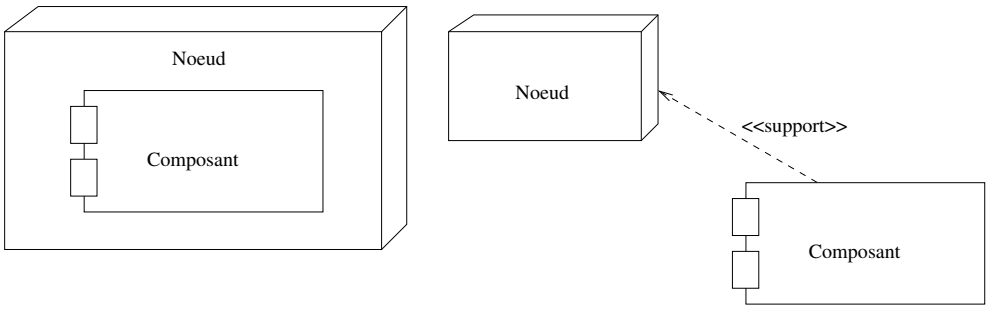

FIG. 5.1 – Représentation d'un composant à l'intérieur d'un nœud

La migration et la copie d'un composant d'un nœud vers un autre sont effectuées respectivement en utilisant les stérétotypes <<devient>> et <<copie>>.

Un diagramme de déploiement permet également de représenter les relations entre différents nœuds.

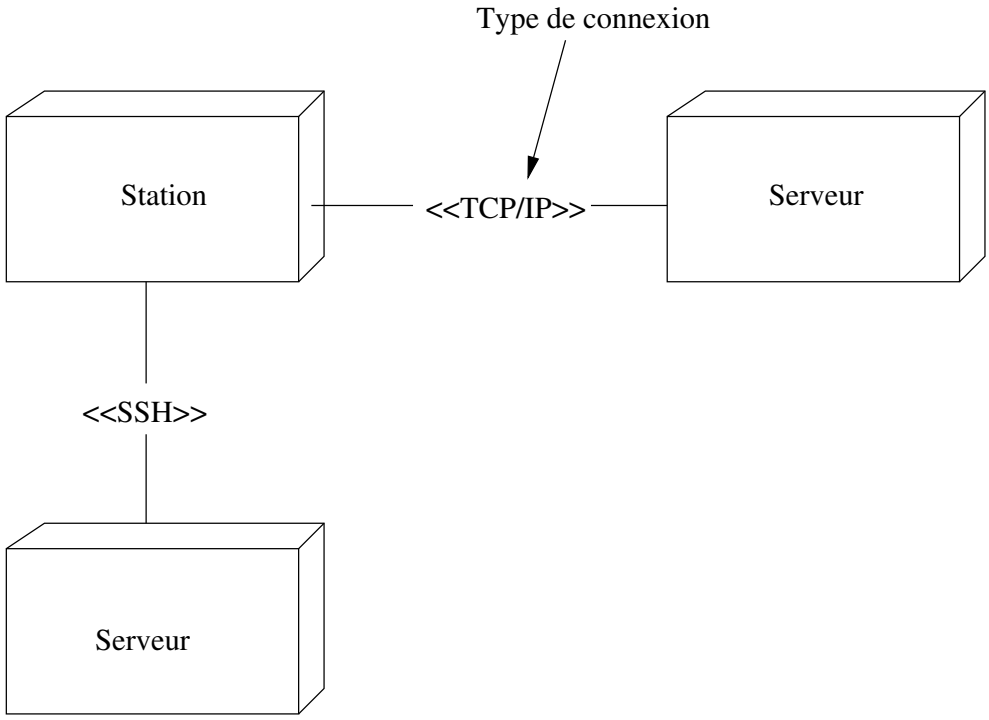

FIG. 5.2 – Représentation de relations entre nœuds

CHAPITRE 6

Diagrammes de cas d'utilisation (statique)

☞ *Diagramme de cas d'utilisation:* représentent les cas d'utilisation du système, les acteurs et les relations existant entre eux.

Les diagrammes de cas d'utilisation décrivent sous la forme d'actions et de réactions, le comportement d'un système du point de vue d'un utilisateur. Ils permettent de définir les limites du système et les relations entre le système et l'environnement. Ce type de diagrammes interviennent tout au long du cycle de développement, depuis le cahier des charges jusqu'à la fin de la réalisation.

☞ *acteur:* représente un rôle joué par une personne ou une chose qui interagit avec un système.

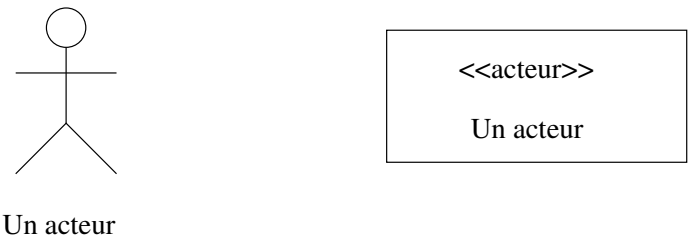

FIG. 6.1 – Représentation d'un acteur

Un acteur peut également participer à des relations de généralisation/spécialisation :

☞ *cas d'utilisation:* est une modélisation d'une fonctionnalité ou d'une classe.

Les cas d'utilisation se déterminent en observant et en précisant, acteur par acteur, les séquences d'interaction du point de vue de l'utilisateur. Ils se décrivent en termes d'informations échangées et d'étapes dans la manière d'utiliser le système. Un cas d'utilisation regroupe une famille de scénarios d'utilisation selon un critère fonctionnel. Ils décrivent des interactions potentielles, sans entrer dans les détails de l'implémentation.

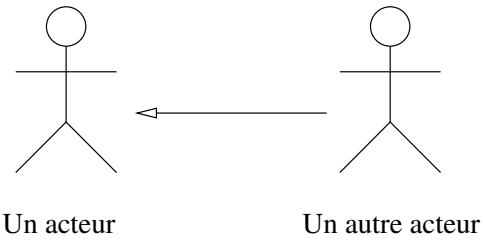

FIG. 6.2 – Généralisation entre acteurs

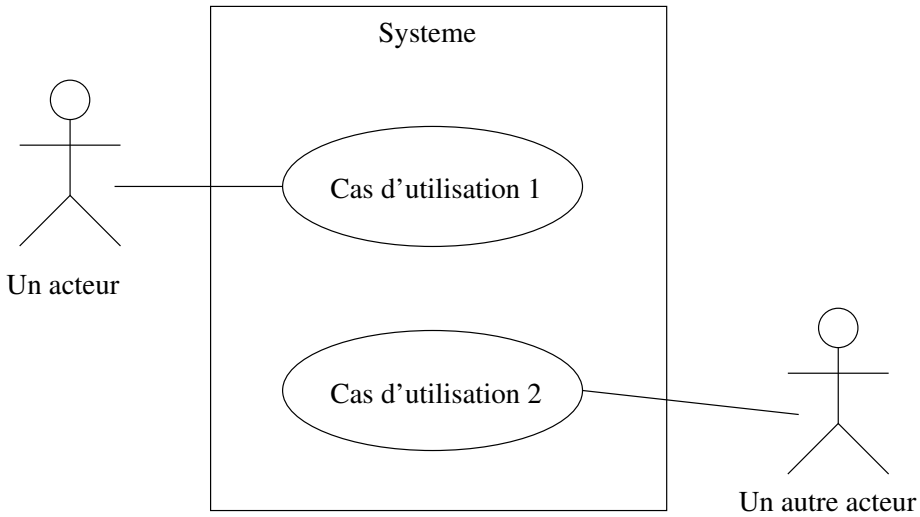

FIG. 6.3 – Représentation d'un cas d'utilisation

Il existe trois types de relations entre cas d'utilisation :

– La relation *de généralisation* : le cas d'utilisation enfant est une spécialisation du cas d'utilisation parent.
– La relation *d'inclusion* : le cas d'utilisation source comprend également le comportement de son cas d'utilisation destination. Cette relation a un caractère obligatoire (à la différence de la généralisation) et permet ainsi de décomposer des comportements partageables entre plusieurs cas d'utilisation différents.
– Le relation *d'extension* : le cas d'utilisation source ajoute son comportement au cas d'utilisation destination. L'extension peut-être soumise à condition.Cette relation permet de modéliser des variantes de comportement d'un cas d'utilisation.

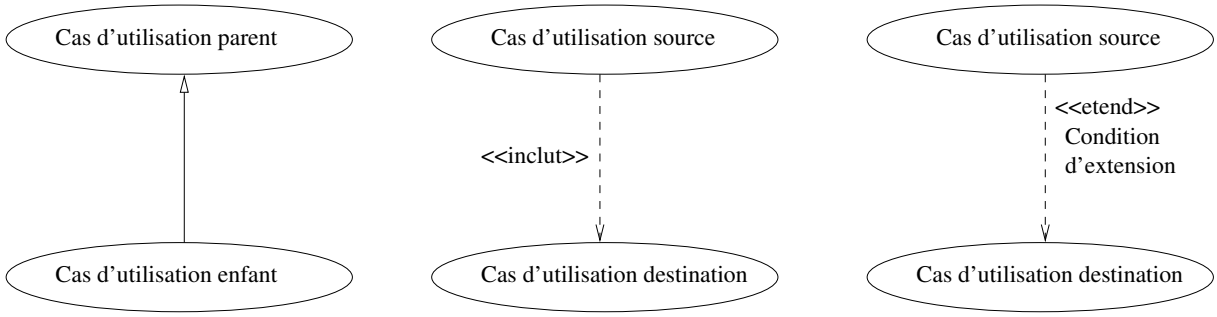

FIG. 6.4 – Les trois relations entre cas d'utilisation

CHAPITRE 7

Diagrammes de séquence (dynamique)

☞ *Diagramme de séquence:* montre des interactions entre objets selon un point de vue temporel. Ce type de diagramme sert à modéliser les aspects dynamiques des systèmes temps réels et des scénarios complexes mettant en œuvre peu d'objets.

Dans ce type de diagrammes, l'accent est mis sur la chronologie des envois de messages (cf. plus bas). La représentation se concentre sur l'expression des interactions et non pas sur l'état ou le contexte des objets. Ce type de diagramme est usuellement utilisé pour illustrer les diagrammes de cas d'utilisation.

☞ *Interactions:* modélisent un comportement dynamique entre objets. Elles se traduisent par l'envoi de messages entre objets. Un diagramme de séquence représente une interaction entre objets, en insistant sur la chronologie des envois de messages.

Dans un diagramme de séquence, les objets sont associés à une ligne de vie. La dimension verticale de celle-ci représente l'écoulement du temps (du haut vers le bas). Notons que la disposition des objets sur l'axe horizontal n'est pas importante dans ce type de diagrammes.

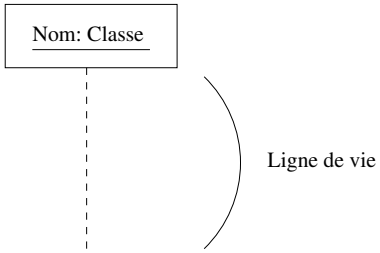

FIG. 7.1 – Un objet et sa ligne de vie

☞ *Message:* représentation d'une communication au cours de laquelle des informations sont echangées.

Les messages sont représentés par des flèches et leur ordre est donné par leurs positions sur la ligne de vie. Ils représentent toute forme de communication entre objets : appels de procédures, signaux, interruptions matérielles...

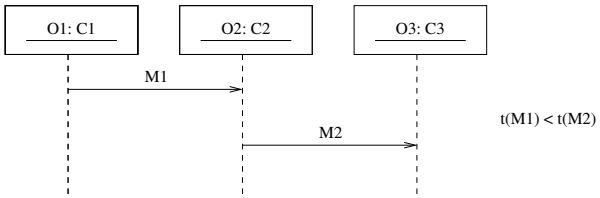

FIG. 7.2 – Agencement de messages

☞ *Période d'activation:* correspond au temps pendant lequel un objet effectue une action, soit directement, soit par l'intermédiaire d'un autre objet.

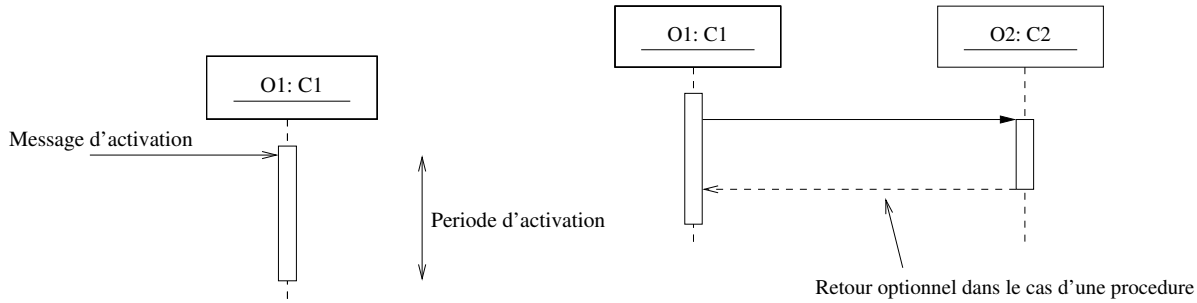

FIG. 7.3 – Activations d'un objet de manière simple et par l'intermédiaire d'un autre objet

Si l'on considère le second schéma de la Figure 7.3, O1 active O2. La période d'activation de O1 recouvre celle de O2. Dans le cas d'un appel de procédure, O1 est bloqué jusqu'à ce que O2 lui redonne la main.

Notons que la période d'activation d'un objet n'a aucun rapport avec sa création ou sa destruction. Un objet peut rêtre actif plusieurs fois au cours de son existence.

Il existe deux catégories d'envoi de messages : Les flots de contrôle à plat et les flot de contrôle emboités.

☞ *Flots de contrôle à plat:* simple progression vers la prochaine étape d'une séquence.

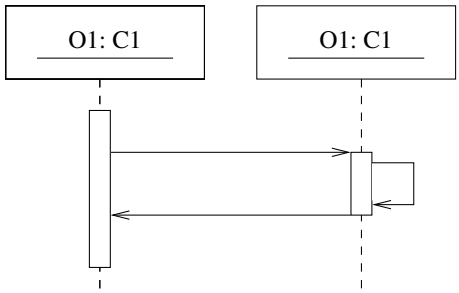

FIG. 7.4 – Un flot de contrôle à plat

☞ *Flots de contrôle emboité:* la séquence emboitée doit se terminer pour que la séquence englobante reprenne le contrôle. Ce type de flot est utilisé pour représenter des appels de procédures avec un retour de valeur optionnel.

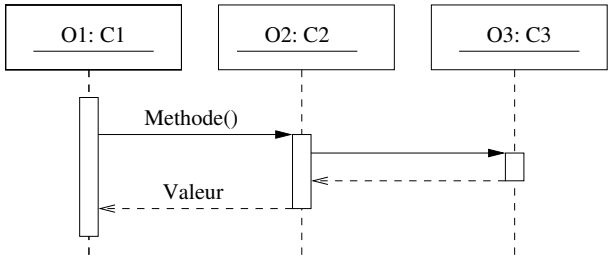

FIG. 7.5 – Un flot de contrôle emboité

Les diagrammes suivants présentent quelques constructions communes : récursivité, réflexion, création et destruction,...

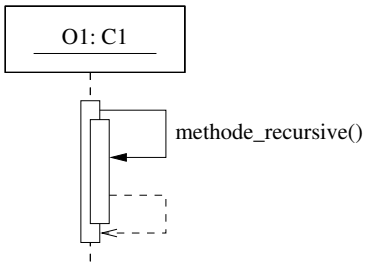

FIG. 7.6 – Représentation de la récursivité

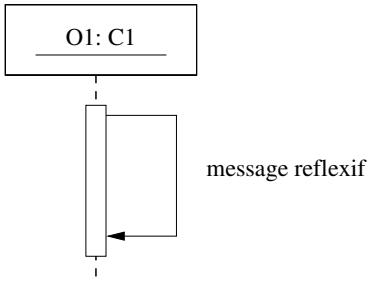

FIG. 7.7 – Représentation de la refléxion

Dans le cas d'un objet composite, cette construction peut indiquer un point d'entrée dans une activité de plus bas niveau qui s'exerce au sein de l'objet :

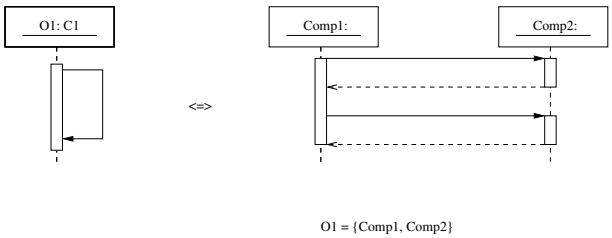

FIG. 7.8 – Masquage d'informations par utilisation de la réflexion

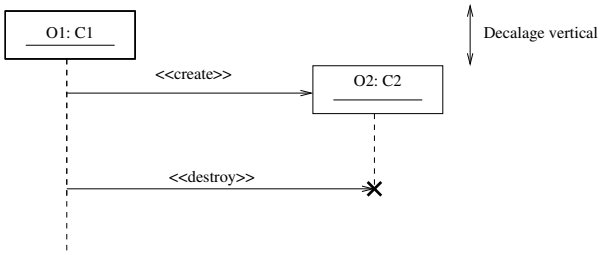

FIG. 7.9 – Création et destruction d'objets

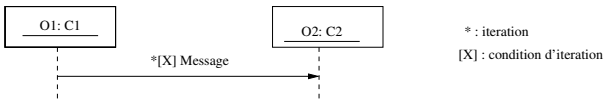

FIG. 7.10 – La boucle « while »

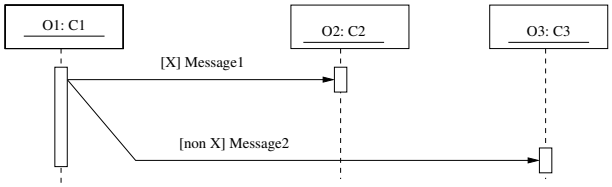

FIG. 7.11 – Le branchement conditionnel « if...then...else... »

CHAPITRE 8

Diagrammes de collaborations (dynamique)

CHAPITRE 9

Diagrammes d'états-transitions (dynamique)

CHAPITRE 10

Diagrammes d'activités (dynamique)

CHAPITRE 11

Le langage de contraintes OCL

CHAPITRE 12

Design Patterns

Liste des figures