Grenoble INP – Ensimag
Deuxième année

Analyse et Conception Objet de Logiciels

C. Oriat
2023–2024

Table des matières

Introduction

Les projets logiciels sont célèbres pour leurs dépassements de budget, leurs cahiers des charges non respectés. De façon générale, le développement de logiciel est une activité complexe, qui est loin de se réduire à la programmation. Le développement de logiciels, en particulier lorsque les programmes ont une certaine taille, nécessite d'adopter une méthode de développement, qui permet d'assister une ou plusieurs étapes du cycle de vie de logiciel. Parmi les méthodes de développement, les approches objet, issues de la programmation objet, sont basées sur une modélisation du domaine d'application. Cette modélisation a pour but de faciliter la communication avec les utilisateurs, de réduire la complexité en proposant des vues à différents niveaux d'abstraction, de guider la construction du logiciel et de documenter les différents choix de conception. Le langage UML (Unified Modeling Language) est un langage de modélisation, qui permet d'élaborer des modèles objets indépendamment du langage de programmation, à l'aide de différents diagrammes.

Ce cours commence par une présentation du langage UML. Ensuite sont abordés les problèmes liés à l'analyse, la modélisation et la conception de logiciels. Ce cours est illustré par plusieurs exemples et études de cas.

Chapitre I

UML

1. Introduction

UML (Unified Modeling Language) est un langage, plus précisément une notation graphique, de modélisation à objets. UML permet de visualiser, spécifier, construire et documenter les différentes parties d'un système logiciel. Il s'agit d'un langage graphique, basé sur différents *diagrammes*. UML n'est pas une méthode, mais peut être employé dans tout le cycle de développement, indépendamment de la méthode.

Initialement, les buts des concepteurs d'UML étaient les suivants :

- représenter des systèmes entiers (pas uniquement logiciels) par des concepts objets ;
- lier explicitement des concepts et le code qui les implantent ;
- pouvoir modéliser des systèmes à différents niveaux de granularité, (pour permettre d'appréhender des systèmes complexes) ;
- créer un langage de modélisation utilisable à la fois par les humains et les machines.

a) Historique

Programmation objet

Les approches objets ont pour origine la programmation objet.

1967	Simula introduit la notion de classe pour implémenter des types abstraits
1976	SmallTalk introduit les principaux concepts de programmation objet (encapsulation, agrégation, héritage)
1983	C++
1986	Eiffel
1995	Java, Ada95
2000	C#

Méthodes de développement

Les méthodes objet sont apparues après les premiers langages objet, à la fin des années 1980, après les méthodes fonctionnelles et les méthodes « systémiques », qui permettent de modéliser à la fois les données et les traitements.

1970	Premières méthodes fonctionnelles
1980	Approches systémiques : modélisation des données et des traitements (Merise)
1990–1995	Apparition d'une cinquantaine de méthodes objet (Booch, OMT, OOA, OOD, HOOD, OOSE)

UML

Suite à l'apparition d'une grande quantité de méthodes objet au début des années 1990, Booch (auteur de la méthode Booch), Rumbaugh (auteur de la méthode OMT) et Jackobson (auteur de la méthode OOSE, et des cas d'utilisation) commencent à travailler sur la « méthode unifiée » (Unified Method).

En 1996 est créé un consortium de partenaires pour travailler sur la définition d'UML dans L'OMG. Parmi les participants, on trouve Rationale, IBM, HP, Microsoft, Oracle...

1995	Début du travail de Booch, Rumbaugh et Jackobson sur la méthode unifiée
1996	Création d'un consortium de partenaires pour travailler sur la définition d'UML dans l'OMG.
1997	UML 1.1 (normalisé par l'OMG)
1998	UML 1.2
1999	UML 1.3
2001	UML 1.4
2003	UML 1.5
2005	UML 2.0
2015	UML 2.5

OMG (Object Management Group)

L'OMG est une organisation internationale, comportant plus de 800 membres (informaticiens et utilisateurs), créée en 1989 pour promouvoir la théorie et la pratique de la technologie objet dans le développement de logiciel.

b) Concepts objet

L'idée principale de l'approche objet est de centraliser les données et les traitements associés dans une même unité, appelée objet.

Objet

Un objet est caractérisé par :
- une identité (ou un nom) ;

— un état, défini par un ensemble de valeurs d'attributs ;
— un comportement, défini par un ensemble de méthodes.

Classe

Une classe est une abstraction qui représente un ensemble d'objets de même nature (c'est-à-dire ayant les mêmes attributs et méthodes). On dit qu'un objet est une « instance » de sa classe.

Encapsulation

L'encapsulation est la possibilité de masquer certains détails de l'implantation. Ceci peut être réalisé en particulier par des constituants *privés* des objets.

Agrégation et composition

L'agrégation et la composition permettent de définir des objets composites, fabriqués à partir d'objets plus simples.

Extension

L'extension est la possibilité de définir une nouvelle classe, appelée classe *dérivée*, à partir d'une classe existante. On peut ajouter des attributs ou des méthodes. L'extension permet de définir des hiérarchies de classes.

Héritage

Une classe dérivée hérite des attributs et méthodes de la classe mère. L'héritage évite la duplication de constituants (attributs, méthodes) et encourage la réutilisation.

Concepts de programmation objet

Les concepts d'objet, de classe, de composition, d'extension... que nous venons de présenter sont communs aux approches objet et à la programmation objet.

Par contre, les notions de *redéfinition* et de *liaison dynamique* sont propres à la programmation objet.

Redéfinition de méthodes héritées

Dans une classe dérivée, certaines méthodes héritées peuvent être redéfinies (ou spécialisées).

Liaison dynamique

L'exécution d'une méthode dépend du type dynamique (type à l'exécution) d'un objet. L'intérêt de ce mécanisme est de pouvoir définir de façon uniforme des opérations qui s'appliquent à des objets de types différents, mais dérivés d'une classe commune.

c) Les différents diagrammes UML

La notation UML définit 14 diagrammes, qui permettent de modéliser différents aspects d'un système. On se concentre ici sur les 9 diagrammes de UML 1.5.

Besoins des utilisateurs

Les besoins des utilisateurs peuvent être décrits à l'aide de *diagrammes de cas d'utilisation*. Un diagramme de cas d'utilisation définit les différents acteurs (qui interagissent avec le système) et les différents cas d'utilisations (ou fonctionnalités) du système.

Aspects statiques

Les aspects statiques d'un système sont décrits à l'aide de deux sortes de diagrammes :

- Les diagrammes de classes définissent les différentes classes, ainsi que les relations entre ces classes.
- Les diagrammes d'objets définissent différentes instances de ces classes, ainsi que les liens entre ces instances.

Aspects dynamiques

Les aspects dynamiques d'un système sont décrits par quatre sortes de diagrammes :

- Les diagrammes de séquence permettent de représenter des interactions d'un point de vue chronologique. Il permettent d'une part de représenter les interactions entre le système et les acteurs, d'autre part de représenter les interactions entre objets à l'intérieur du système.
- Les diagrammes de collaboration permettent de représenter les interactions entre objets d'un point de vue spacial.
- Les diagrammes d'états-transitions sont des automates hiérarchiques qui permettent de décrire des comportements : comportement d'un acteur, d'un système ou d'un objet d'une certaine classe.
- Les diagrammes d'activités permettent de modéliser le comportement d'une méthode en représentant un enchaînement d'activités.

Aspects physiques

Les aspects physiques d'un système peuvent être décrits à l'aide de deux sortes de diagrammes :

- Les diagrammes de composants permettent de décrire les composants (ou modules) d'un système (fichiers sources, fichiers objets, librairies, exécutables...).
- Les diagrammes de déploiement décrivent la disposition physique des matériels, et la répartition des composants sur ces matériels.

2. Diagrammes des cas d'utilisation

Les cas d'utilisation permettent de décrire le système du point de vue de l'utilisateur. Ils permettent de définir les limites du système et les relations entre le système et son environnement.

Un cas d'utilisation est une manière spécifique d'utiliser le système. Un cas d'utilisation correspond à une fonctionnalité du système.

Un acteur représente une personne ou une chose qui interagit avec le système. Plus précisément, un acteur est un rôle joué par une personne ou une chose qui interagit avec le système. Une même personne physique peut donc correspondre à plusieurs acteurs si celle-ci peut jouer plusieurs rôles.

Relation entre un acteur et un cas d'utilisation

Un acteur qui interagit avec le système pour utiliser une fonctionnalité est représenté par une relation entre cet acteur et le cas d'utilisation qui représente cette fonctionnalité. On dit également que l'acteur *déclenche* le cas d'utilisation auquel il est lié.

La figure I. 1 montre la représentation d'un acteur, d'un cas d'utilisation et d'une relation entre l'acteur et le cas d'utilisation.

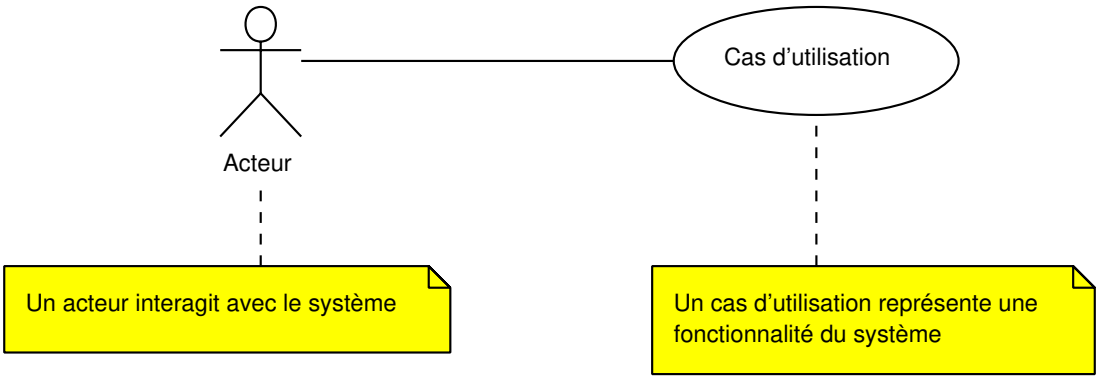

FIGURE I. 1 – Relation entre un acteur et un cas d'utilisation

Généralisation entre deux acteurs

Un acteur, appelé *acteur parent*, *généralise* un acteur, appelé *acteur enfant*, lorsque tout ce qui peut être réalisé par l'acteur parent peut être réalisé par l'acteur enfant. On dit également que l'acteur enfant spécialise l'acteur parent.

La figure I. 2 montre la représentation d'une généralisation entre deux acteurs.

FIGURE I. 2 – Généralisation entre deux acteurs

Inclusion entre deux cas d'utilisation

Un cas d'utilisation, appelé *cas source*, *inclut* un cas d'utilisation, appelé *cas destination*, si les comportements décrits par le cas source contiennent les comportements décrits par le cas destination (cf. figure I. 3). Les inclusions entre cas d'utilisation sont fréquemment utilisés, en particulier pour factoriser une fonctionnalité partagée par plusieurs cas d'utilisation.

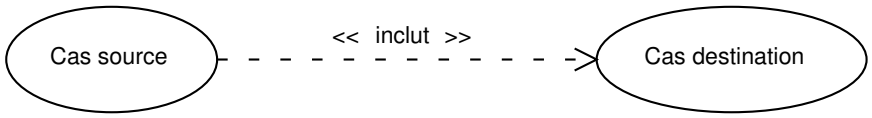

FIGURE I. 3 – Inclusion entre deux cas d'utilisation

Extension d'un cas d'utilisation

Un cas d'utilisation, appelé *cas source*, *étend* un cas d'utilisation, appelé *cas destination*, si les comportement décrits par le cas source étendent les comportement décrits par le cas destination (cf. figure I. 4). L'extension peut être soumise à une condition.

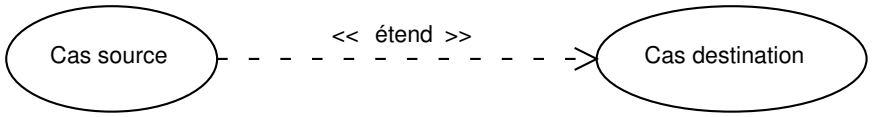

FIGURE I. 4 – Extension entre deux cas d'utilisation

Généralisation entre deux cas d'utilisation

Un cas d'utilisation, appelé *cas d'utilisation parent*, *généralise* un cas d'utilisation, appelé *cas d'utilisation enfant*, lorsque les comportements décrits par le cas d'utilisation parent spécialisent les comportements décrits par le cas d'utilisation enfant (cf. figure I. 5). Une généralisation indique que le cas d'utilisation enfant est une variation du cas d'utilisation parent. Les généralisations entre cas d'utilisation sont assez rarement utilisées.

Limites du système

Les limites du système peuvent être précisées à l'aide d'un rectangle (cf. figure I. 6). Les acteurs, en général, ne font pas partie du système (ils interagissent avec le système).

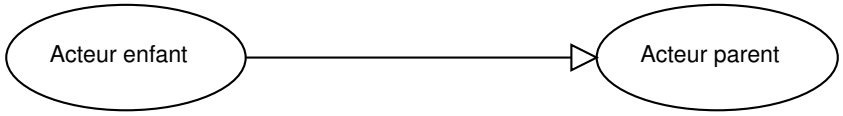

FIGURE I. 5 – Généralisation entre deux cas d'utilisation

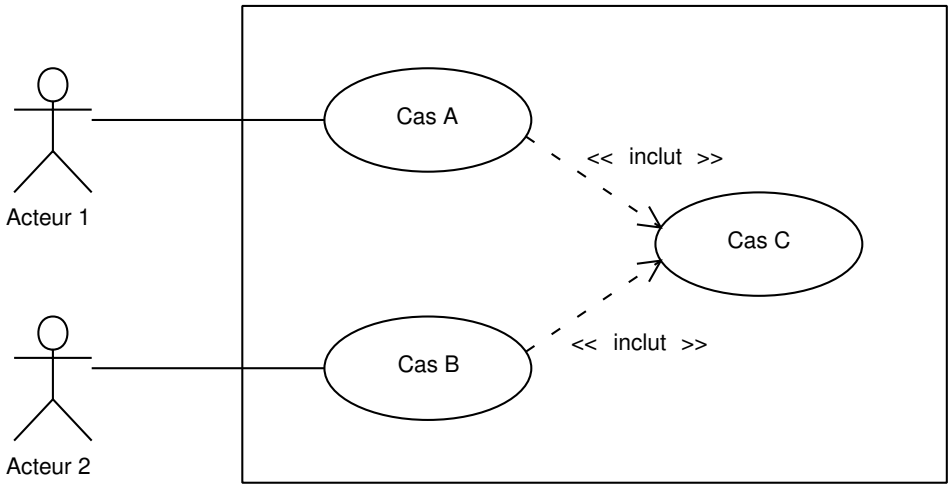

FIGURE I. 6 – Limites du système

Exemple 1. Système de virements bancaires

La figure 2. représente un diagramme de cas d'utilisation d'un système permettant d'effectuer des virements bancaires.

On a deux sortes d'acteurs : les clients locaux et les clients distants. Un client local peut effectuer un virement (cas d'utilisation *Virement*, qui inclut le cas d'utilisation *Identification*). Un client distant peut effectuer un virement par minitel (cas d'utilisation *Virement par minitel*, qui étend le cas d'utilisation *Virement*).

Exemple 2. Distributeur de billets

La figure I. 8 est un diagramme de cas d'utilisation pour un distributeur de billets de banque.

On a deux acteurs : le client, qui peut soit consulter le solde de son compte, soit retirer de l'argent au distributeur ; le technicien, qui peut allumer ou éteindre le distributeur et ravitailler le distributeur.

Une *note* (ou *commentaire*) indique que le technicien doit éteindre le distributeur avant de le ravitailler.

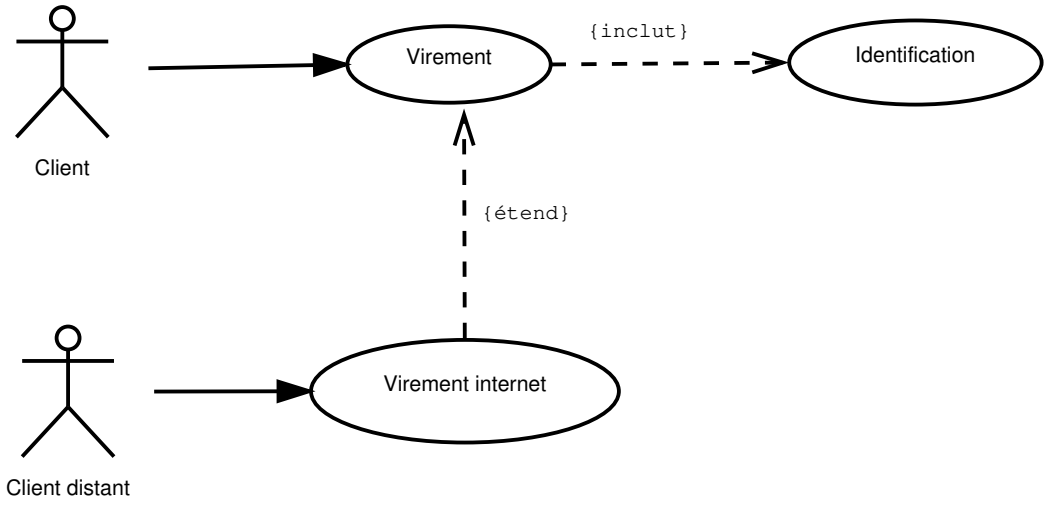

FIGURE I. 7 – Diagramme de cas d'utilisation pour un système de virements bancaires

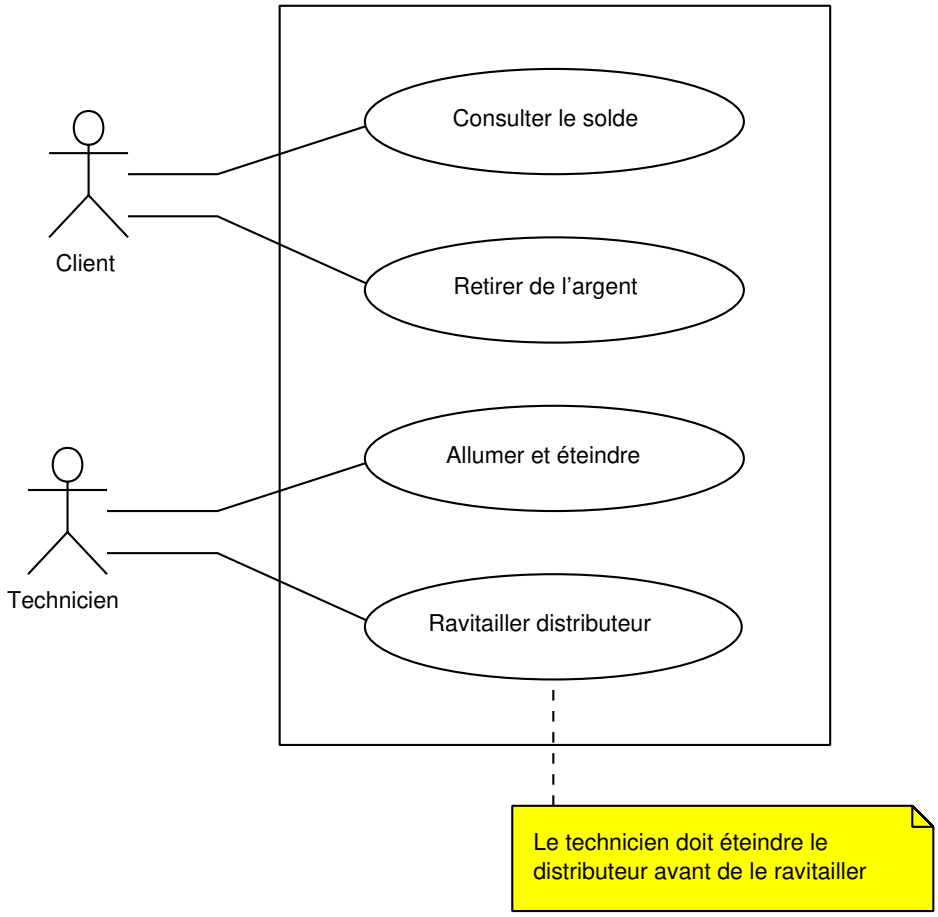

FIGURE I. 8 – Diagramme de cas d'utilisation pour un distributeur de billets

3. Diagrammes de classes et d'objets

a) Classe, objet et association

Classe

Une classe est une abstraction qui représente un ensemble d'objets de même nature. On dit qu'un objet est une « instance » de sa classe. Une classe est « instanciée » lorsqu'elle contient au moins un objet.

Une classe comporte un nom, des attributs et des opérations. La figure I. 9 représente deux classes : la classe `Compte` (pour des comptes bancaires) et la classe `Personne`. La classe `Compte` comporte deux attributs : `solde`, qui représente le montant disponible sur le compte, et `min`, qui représente le montant minimal que le compte peut contenir.

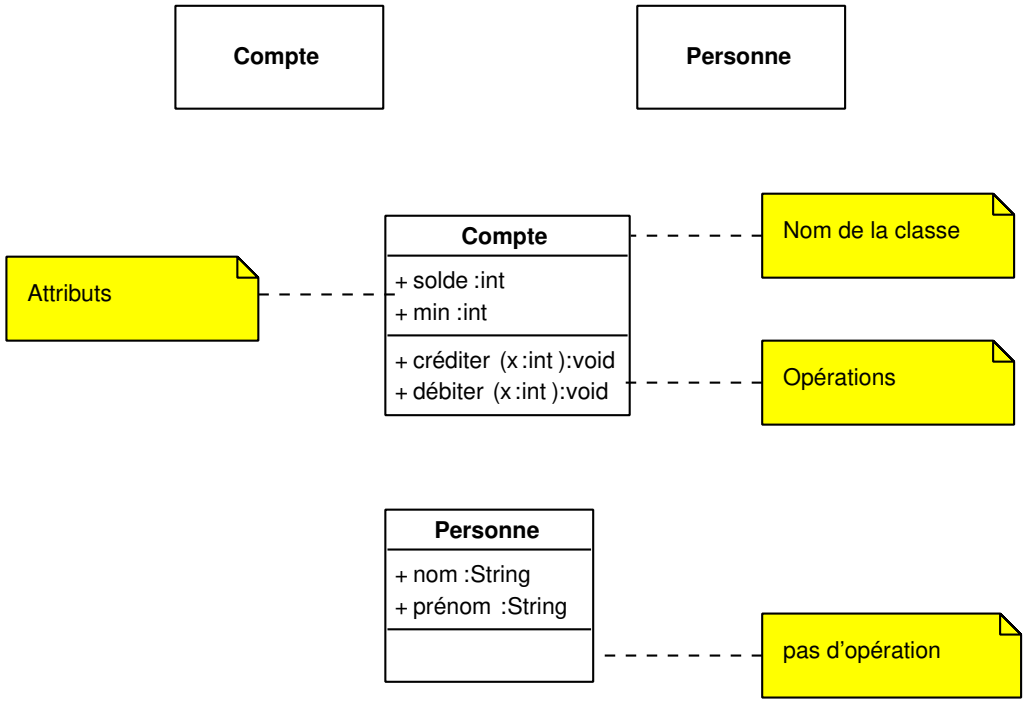

FIGURE I. 9 – Représentation des classes `Compte` et `Personne`

Un attribut est une propriété d'un objet de la classe. Il comporte un nom et éventuellement un type et une valeur initiale. Les types possibles ne sont pas spécifiés en UML.

On peut remarquer que la notation UML n'oblige pas le concepteur à indiquer tous les attributs et méthodes d'une classe. De même, il n'est pas obligatoire de toujours préciser le type des attributs et le profil des opérations.

Un attribut *dérivé* est un attribut dont la valeur peut être calculée à partir des attributs non dérivés.

Dans l'exemple représenté figure I. 10, l'âge d'une personne peut être déduit de l'année courante et de sa date de naissance. De même, la surface d'un rectangle peut être déduite de sa largeur et de sa longueur.

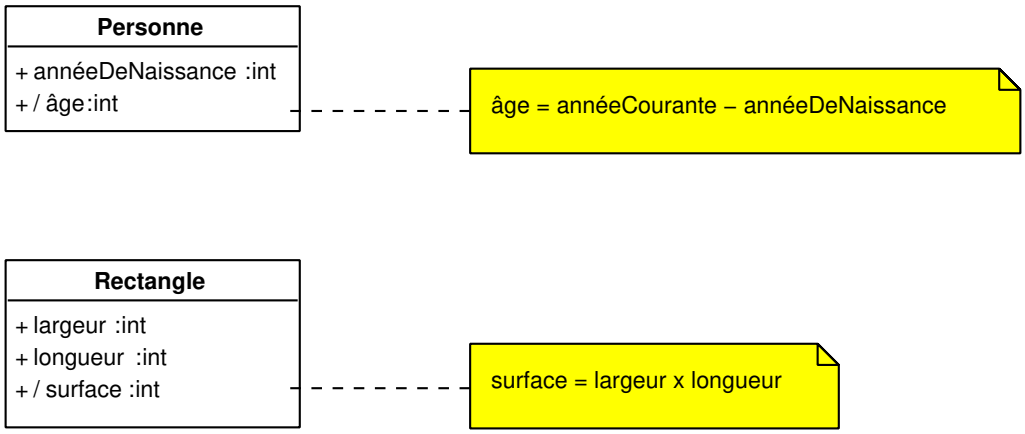

FIGURE I. 10 – Exemples d'attributs dérivés

Un attribut *de classe* est un attribut qui est partagé par toutes les instances de la classe.

Une opération est un service offert par les objets de la classe. En UML, on distingue *opération*, qui spécifie un service, et *méthode* qui implante l'opération. Les opérations peuvent comporter des paramètres (qui peuvent être typés et comporter une valeur initiale) et un résultat.

La syntaxe d'une opération est la suivante :

$$\text{Opération } (mode_1\ arg_1 : Type_1 = val_1, \ldots\ mode_n\ arg_n : Type_n = val_n) : TypeRésultat$$

où :
- le mode des paramètres peut être `in` (paramètre d'entrée), `out` (paramètre de sortie) ou `inout` (paramètre d'entrée sortie). Le mode par défaut est `in`.
- les paramètres, le type, et la valeur initiale sont optionnels.

Objet

Une classe est une abstraction qui représente un ensemble d'*objets*. Chaque objet est une « instance » d'une classe.

La figure I. 11 représente deux instances `C1` et `C2` de la classe `Compte`, deux instances `Pierre` et `Paul` de la classe `Personne`, ainsi que des instances anonymes de ces deux classes.

Niveaux de visibilité

Il y a quatre niveaux de visibilité en UML pour les attributs et les opérations :
- le niveau public (+), qui indique qu'un élément est visible pour tous les clients de la classe ;

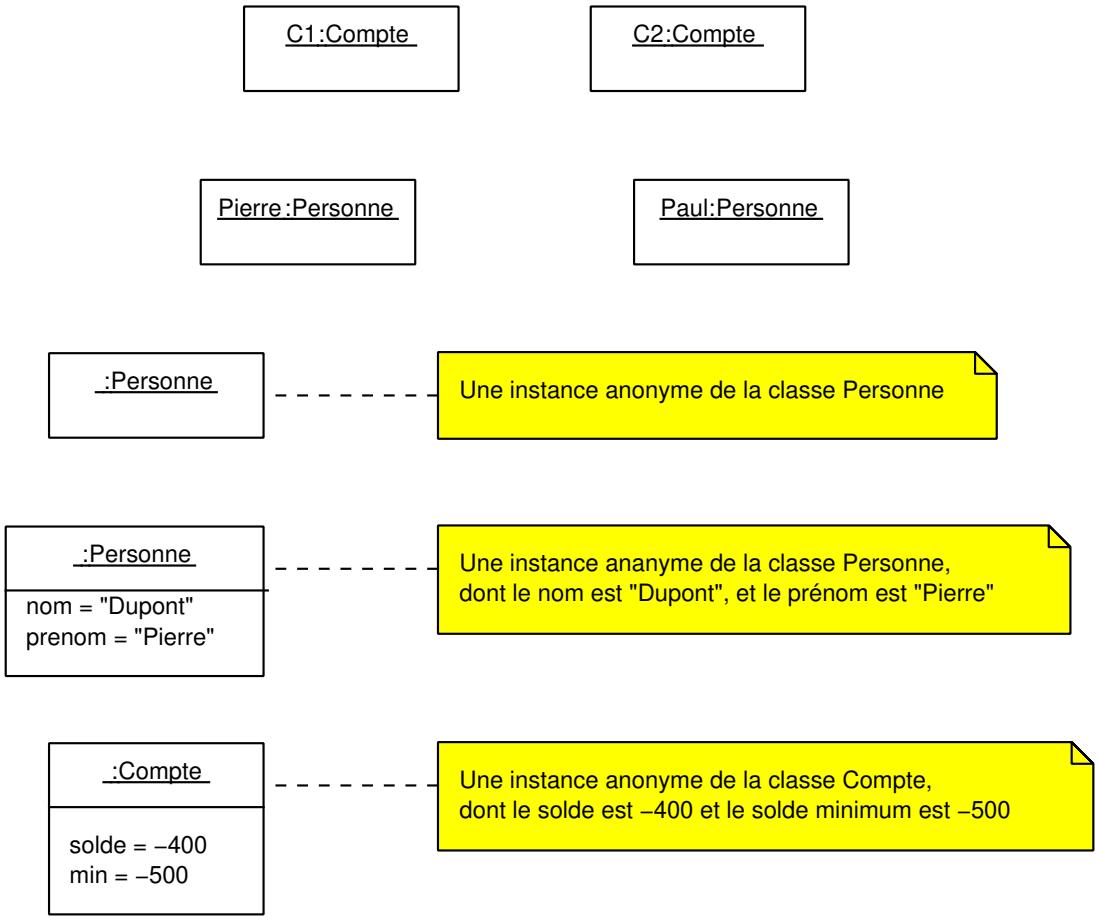

FIGURE I. 11 – Exemples d'instances des classes `Compte` et `Personne`

— le niveau protégé (#), qui indique qu'un élément est accessible uniquement pour les sous-classes de la classe où il apparaît ;
— le niveau paquetage (~), qui indique qu'un élément est visible uniquement pour les classes définies dans le même paquetage ;
— le niveau privé (-), qui indique que seule la classe où est défini cet élément peut y accéder.

La figure I. 12 représente une classe pour les nombres complexes, qui comporte quatre opérations publiques (addition, soustraction, multiplication et division). Cette classe ne précise pas l'implantation d'un nombre complexe, ni les paramètres et le résultat des opérations.

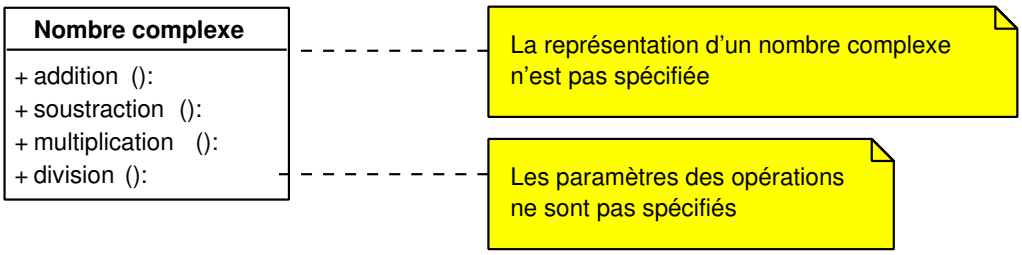

FIGURE I. 12 – Classe des nombres complexes ; implantation non précisée

La figure I. 13 implante les nombres complexes à l'aide du module et de l'argument, qui sont des attributs privés de la classe.

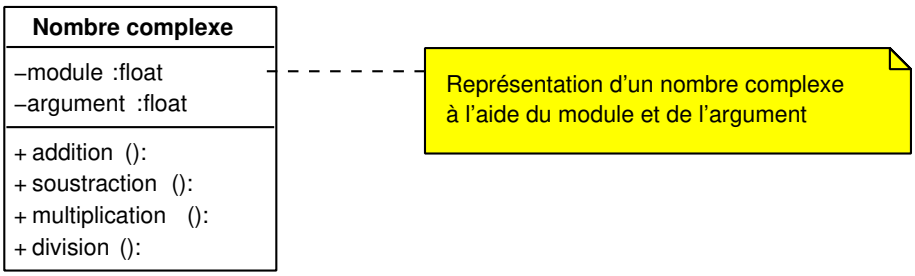

FIGURE I. 13 – Classe des nombres complexes ; implantation avec module et argument

La figure I. 14 implante les nombres complexes à l'aide de la partie réelle et de la partie imaginaire, qui sont des attributs privés de la classe.

Classe utilitaire

Une classe utilitaire permet de regrouper un ensemble de valeurs (valeurs d'attributs) et d'opérations. Une classe utilitaire ne peut pas être instanciée, et ne comporte que des attributs et des opérations *de classe*.

Association

Une association est, de façon générale, une relation n-aire entre n classes d'objets. Les relations les plus utilisées sont les relations binaires, entre deux classes.

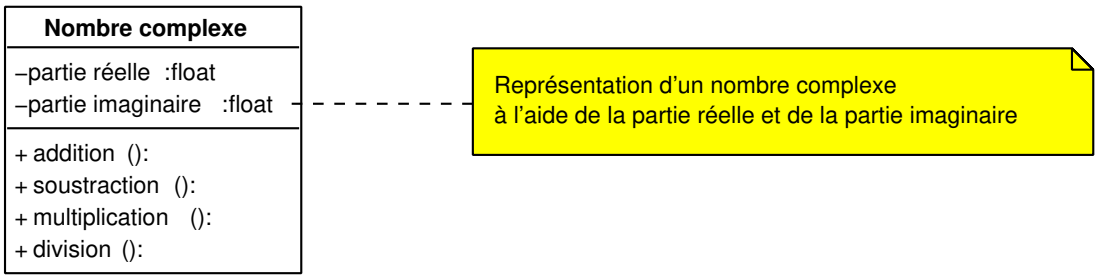

FIGURE I. 14 – Classe des nombres complexes ; implantation avec parties réelle et imaginaire

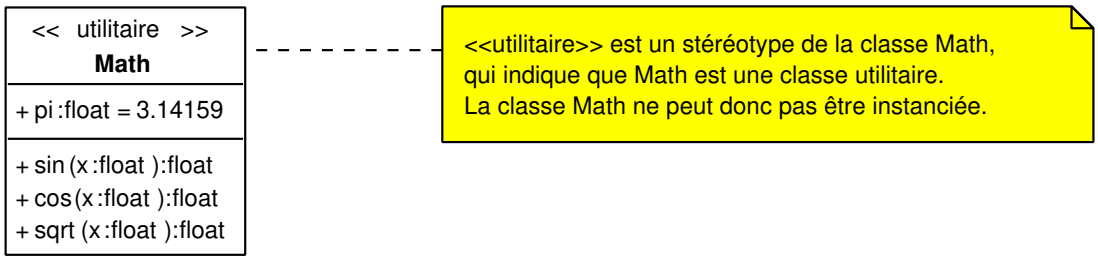

FIGURE I. 15 – Classe utilitaire `Math`

La figure I. 16 représente une relation R entre les classes A et B.

Remarque

Une association entre les classes A et B est une relation binaire, au sens mathématique, autrement dit un sous-ensemble de $A \times B$. Cela signifie qu'il existe au plus un lien entre un objet de la classe A et un objet de la classe B.

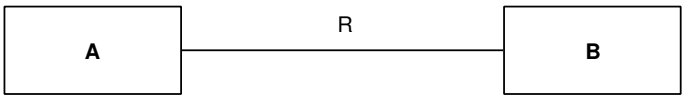

FIGURE I. 16 – Relation R entre les classes A et B

Exemple

On peut spécifier une relation entre la classe Compte et la classe Personne (*diagramme de classes* figure I. 17).

« `*` » et « `1..2` » sont des *multiplicités*, dont la signification est la suivante :
- « `*` » signifie qu'une personne peut ouvrir un nombre quelconque de comptes ;
- « `1..2` » signifie qu'un compte est ouvert pour une ou deux personnes.

La figure I. 18 est un *diagramme d'objets* associé au diagramme de classes représenté figure I. 17.

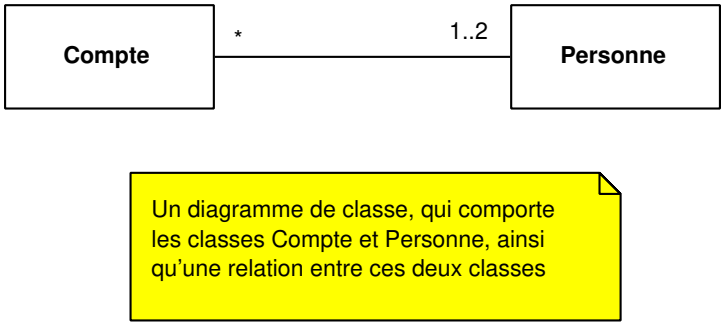

FIGURE I. 17 – Diagramme de classes

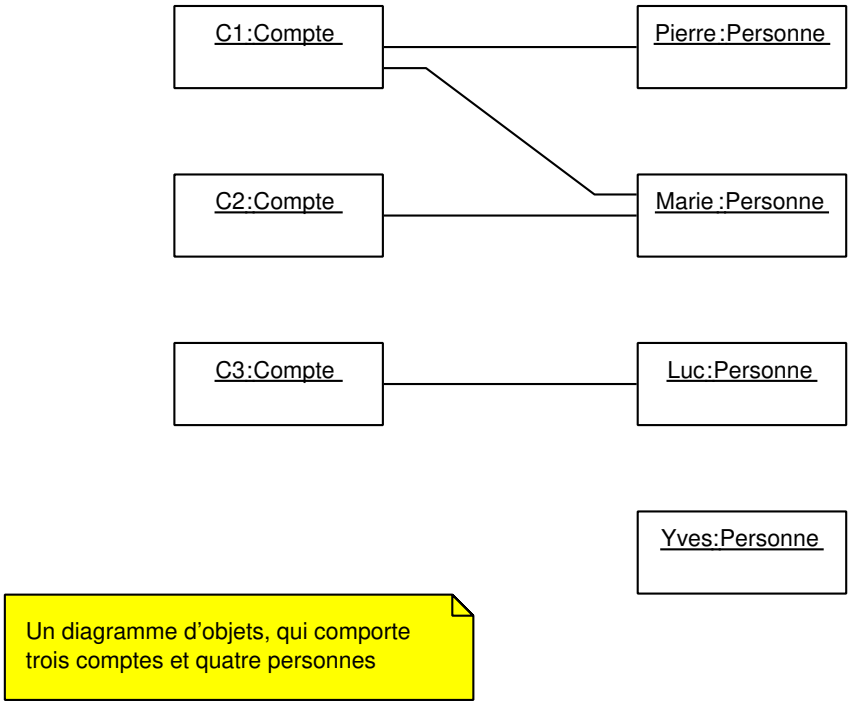

FIGURE I. 18 – Un diagramme d'objets correspondant au diagramme de classes

Ce diagramme d'objets spécifie que :
- Pierre et Marie ont un compte commun (`C1`) ;
- Marie a deux comptes (`C1` et `C2`) ;
- Luc a un compte (`C3`) ;
- Yves n'a pas de compte.

Navigabilité

La relation entre les classes `Personne` et `Compte` que nous avons définie permet :
- d'une part, à partir d'une personne, de déterminer l'ensemble des comptes qu'elle possède ;
- d'autre part, à partir d'un compte, de retrouver son ou ses propriétaires.

On dit que la relation est « navigable dans les deux sens ». Pour spécifier qu'un seul sens de navigation est autorisé, on peut orienter la relation. Par exemple, dans le diagramme de classes représenté figure I. 19, on peut à partir d'un objet de la classe A, accéder aux objets de la classe B qui sont en relation avec celui-ci, mais pas l'inverse.

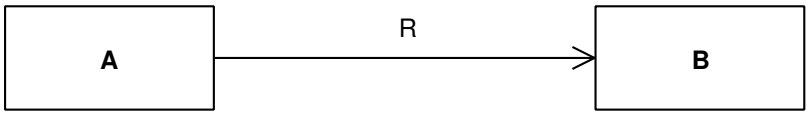

FIGURE I. 19 – Sens de navigation de la relation : de A vers B

Rôles

Les extrémités d'une association sont appelées des *rôles* et peuvent être nommées. Par exemple, dans le diagramme de classes représenté figure I. 20, on a une relation « travaille pour » entre des personnes et des entreprises. Dans cette relation, une personne joue le rôle d'un employé, et une entreprise joue le rôle d'un employeur.

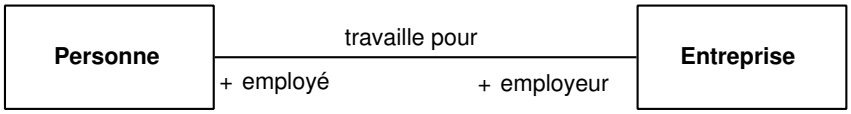

FIGURE I. 20 – Rôles *employé* et *employeur* associés à la relation *travaille pour*

Multiplicités

Une multiplicité est un sous-ensemble de $\mathbb{N}$. Les notations utilisées en UML pour décrire des multiplicités sont les suivantes :

Syntaxe	Sémantique
$*$	$\mathbb{N}$
m .. n	$\{x \in \mathbb{N} \ ; \ m \leq x \leq n\}$
m .. $*$	$\{x \in \mathbb{N} \ ; \ m \leq x\}$
m, n, p	$\{m, n, p\}$
M_1, M_2	$M_1 \cup M_2$

Soit une relation R entre deux classes A et B, avec les multiplicités M_A associée à A et M_B associée à B.

Les multiplicités M_A et M_B imposent les contraintes suivantes :

— pour chaque objet b de la classe B, le nombre d'objets de la classe A liés à b appartient à M_A, autrement dit :

$$\forall b \in B,\ \text{Card}\{(a,b) \in R\ ;\ a \in A\} \in M_A\,;$$

— pour chaque objet a de la classe A, le nombre d'objets de la classe B liés à a appartient à M_B, autrement dit :

$$\forall a \in A,\ \text{Card}\{(a,b) \in R\ ;\ b \in B\} \in M_B.$$

Exemple

Le diagramme de classes représenté figure I. 21 exprime qu'une personne travaille pour au plus une entreprise.

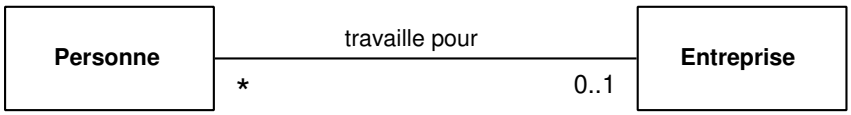

FIGURE I. 21 – Diagramme de classes (relation « travaille pour »)

Le diagramme d'objets représenté figure I. 22 est correct car il respecte cette contrainte.

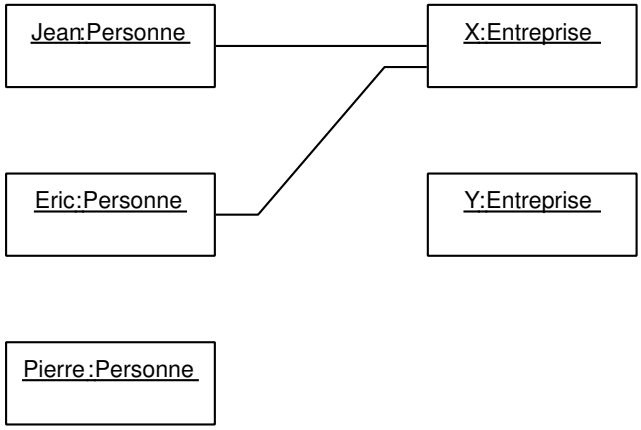

FIGURE I. 22 – Diagramme d'objet correct par rapport au diagramme de classes figure I. 21

Le diagramme d'objets représenté figure I. 23 n'est pas correct par rapport au diagramme de classes figure I. 21 car Eric ne peut pas travailler pour deux entreprises.

Cas particuliers d'associations

La figure I. 24 représente certains cas particuliers d'associations :

— application de la classe X vers la classe Y : à chaque objet de la classe X est associé un unique objet de la classe Y.

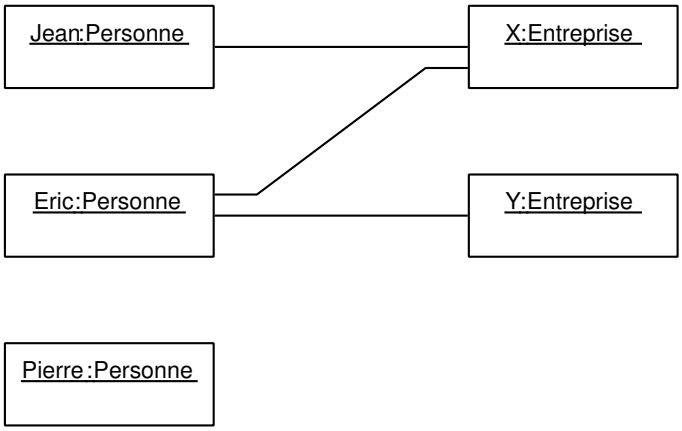

FIGURE I. 23 – Diagramme d'objet incorrect par rapport au diagramme de classes figure I. 21

— fonction partielle de la classe X vers la classe Y : à chaque objet de la classe X est associé au plus un objet de la classe Y.
— isomorphisme entre les classes X et Y : chaque objet de la classe X est en relation avec un unique objet de la classe Y, et chaque objet de la classe Y est en relation avec un unique objet de la classe X. Par conséquent, les ensembles d'objets correspondants aux classes X et Y sont isomorphes.

b) Classe-association

Une classe-association est une entité qui combine les propriétés d'une classe et les propriétés d'une association. Une classe-association R entre deux classes A et B permet d'associer à tout lien entre un objet de la classe A et un objet de la classe B un objet de la classe R.

Dans l'exemple présenté figure I. 25, `Travail` est une classe-association entre les classes `Personne` et `Entreprise`. `Travail` permet d'associer à tout lien entre une personne et une entreprise un objet de la classe `Travail`, qui comporte donc une fonction, un salaire, et peut utiliser l'opération `feuille_paie` (qui sert à produire des feuilles de paie).

Multiplicités

Comme les associations, les classes-associations peuvent spécifier des multiplicités à chacune de leurs extrémités.

Dans l'exemple figure I. 26, le nombre d'objets de la classe Y qui peuvent être associés à un objet de la classe X est spécifié par la multiplicité M_X ; le nombre d'objets de la classe X qui peuvent être associés à un objet de la classe Y est spécifié par la multiplicité M_Y.

De façon générale, la figure I. 27 montre comment ce schéma peut être simulé par deux relations. Intuitivement,
— chaque objet de la classe Z est associé à un objet de la classe X et un objet de la classe Y ;
— chaque objet de la classe X est associé à n objets de la classe Z, avec $n \in M_Y$;

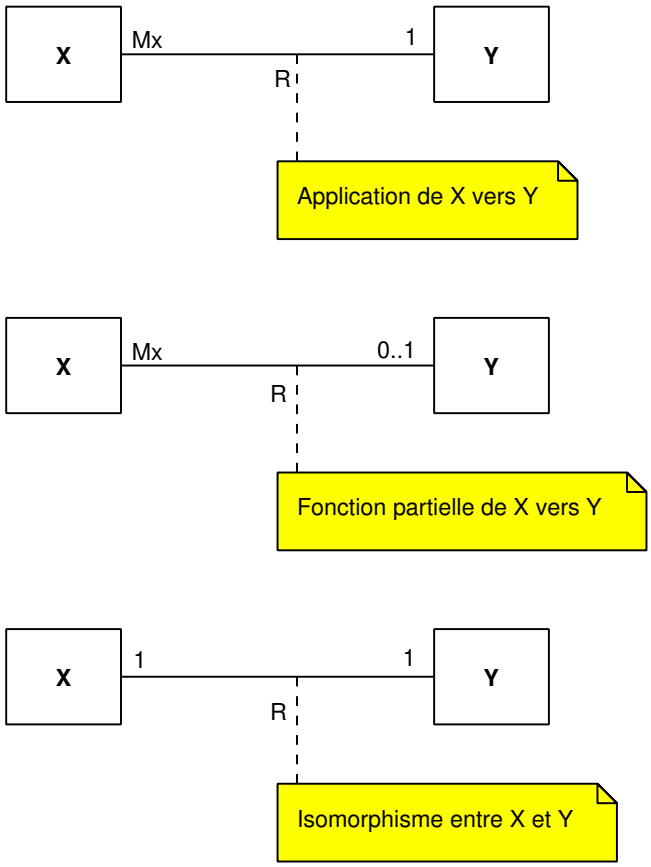

FIGURE I. 24 – Cas particuliers d'associations binaires

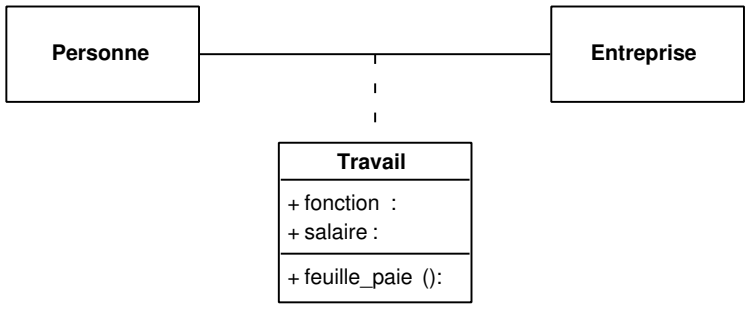

FIGURE I. 25 – Exemple de classe-association entre les classes `Personne` et `Entreprise`

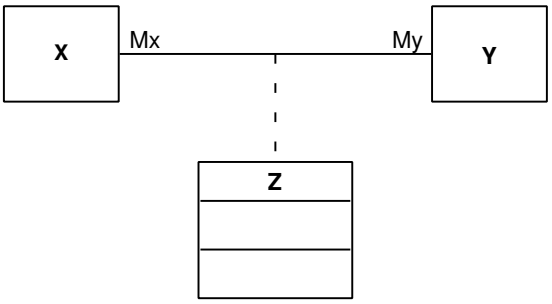

FIGURE I. 26 – Multiplicités d'une classe-association

— chaque objet de la classe Y est associé à n objets de la classe Z, avec $n \in M_X$.

FIGURE I. 27 – Simulation d'une classe-association

Exemple

Le diagramme de classes D_1 représenté figure I. 28 est simulé par le diagramme de classes D_2 représenté figure I. 29.

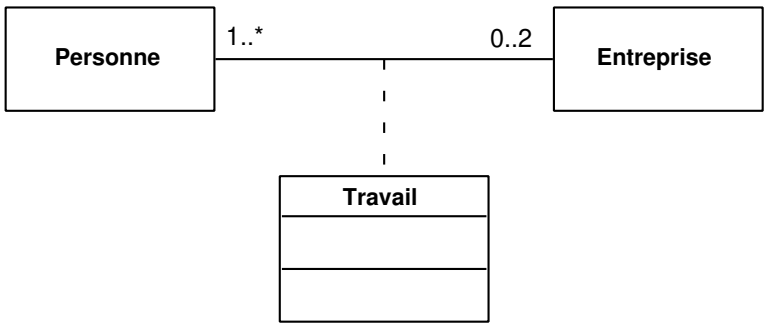

FIGURE I. 28 – Diagramme de classes D_1

La figure I. 30 montre un diagramme d'objets correspondant au diagramme de classe D_1. La figure I. 31 montre un diagramme d'objets correspondant au diagramme de classe D_2.

c) Agrégation et composition

Agrégation

Une agrégation est une association binaire particulière qui modélise une relation entre un « tout » et une « partie », autrement dit, une relation d'appartenance.

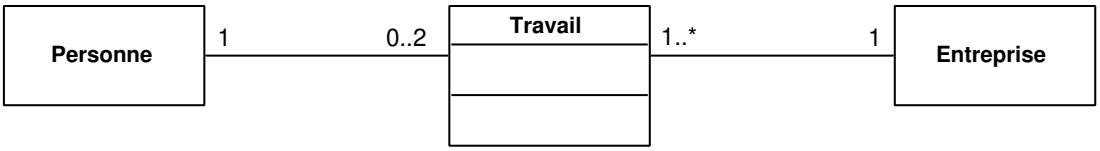

FIGURE I. 29 – Diagramme de classes D_2

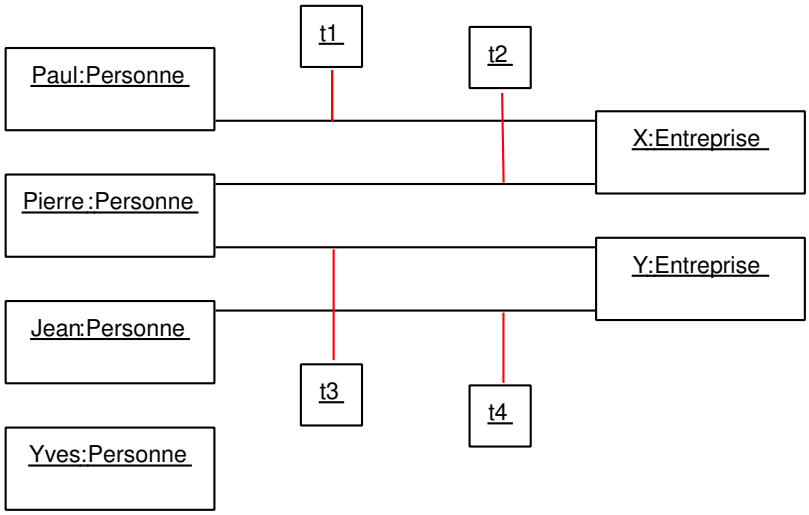

FIGURE I. 30 – Diagramme d'objets correspondant au diagramme de classes D_1

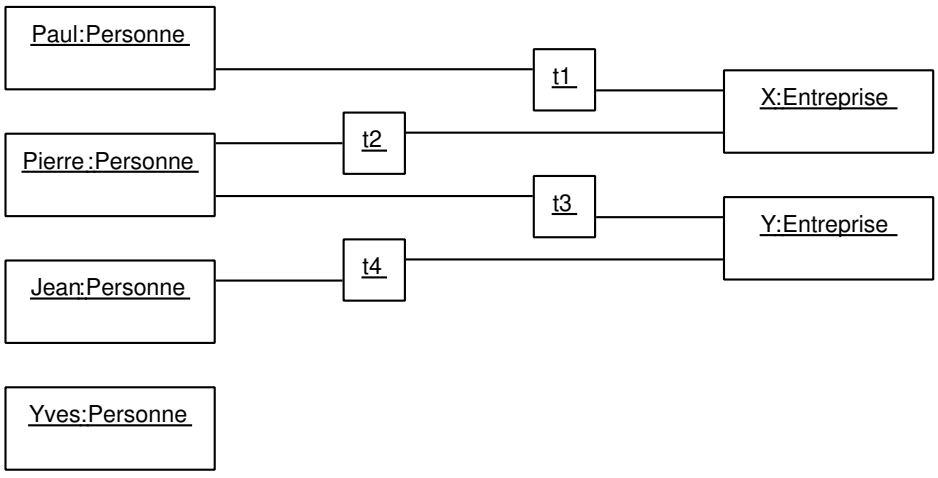

FIGURE I. 31 – Diagramme d'objets correspondant au diagramme de classes D_2

La figure I. 32 montre une agrégation entre les classes A et B : un objet de la classe B, appelé *agrégat*, contient un certain nombre d'objets de la classe A, appelés *composants*.

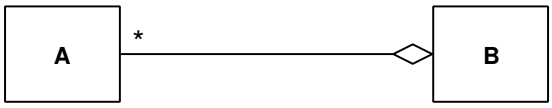

FIGURE I. 32 – Relation d'agrégation entre les classes A et B

Exemple : segment

La figure I. 33 montre la modélisation de points et de segments : un segment est formé de l'agrégation de deux points.

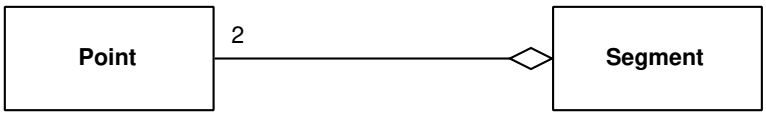

FIGURE I. 33 – Modélisation d'un segment

La figure I. 34 est un diagramme d'objets correspondant au diagramme de classes précédent, qui modélise trois points A, B et C, ainsi que deux segments AB et BC. On peut remarquer que le point B appartient à la fois au segment AB et au segment BC.

Dans une agrégation, certaines parties peuvent être partagée : par défaut, il n'y a pas de contrainte de multiplicité sur l'agrégat. Dans l'exemple, un point peut être partagé entre plusieurs segments.

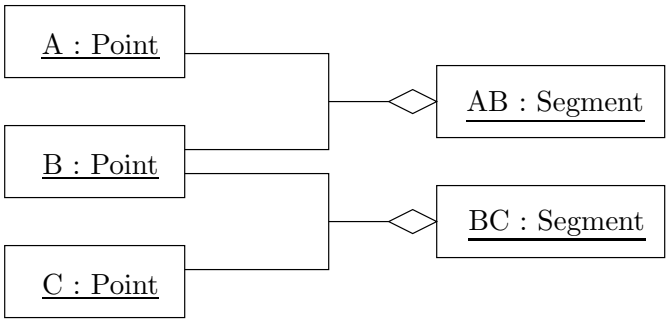

FIGURE I. 34 – Diagramme d'objets modélisant les segments AB et BC

La relation d'agrégation interdit les cycles dans les diagrammes d'objets. On peut par exemple définir une classe A en relation d'agrégation avec elle-même. Par contre, on ne peut pas définir de cycle dans un diagramme d'objets (cf.figure I. 35).

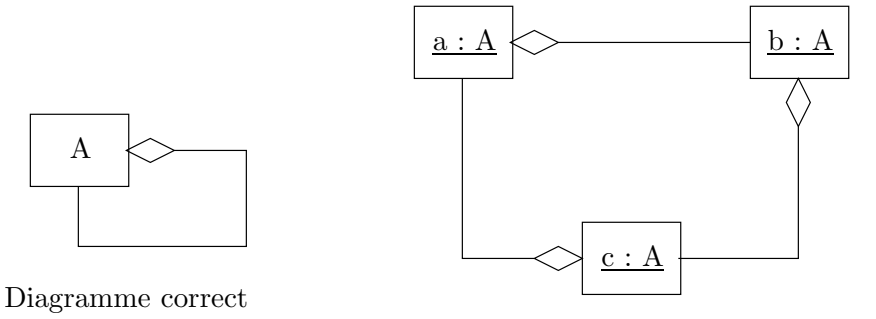

Diagramme correct

Diagramme incorrect : cycles interdits

FIGURE I. 35 – Cycles et agrégations

Composition

Une composition est une agrégation particulière, pour laquelle une partie ne peut être partagée entre plusieurs agrégats. Autrement dit, la multiplicité du coté de l'agrégat est 0 ou 1.

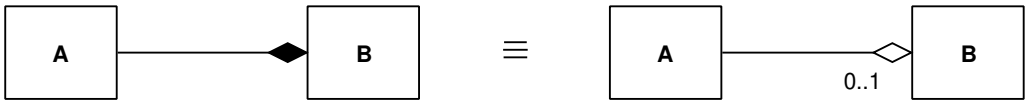

FIGURE I. 36 – Relation de composition entre les classes A et B

Exemple

La figure I. 37 est un diagramme de classes qui modélise des voitures. Une voiture comporte quatre roues. Une roue ne peut pas appartenir à deux voitures.

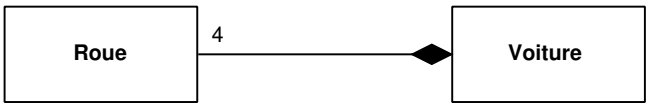

FIGURE I. 37 – Diagramme de classes

La figure I. 38 est un diagramme d'objets correspondant au diagramme de classes précédent, qui montre une voiture composée de quatre roues.

On peut également représenter les composants d'un objet à l'intérieur de l'objet lui-même. Cela est possible uniquement pour les compositions car deux objets ne peuvent pas partager un même composant. La figure I. 39 montre un diagramme d'objets équivalent au diagramme d'objets de la figure I. 38.

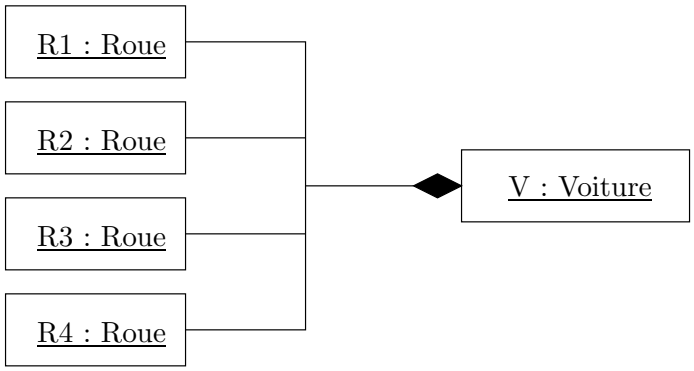

FIGURE I. 38 – Un diagramme d'objets correspondant au diagramme de classes figure I. 37

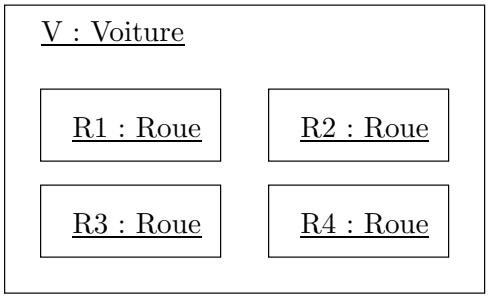

FIGURE I. 39 – Représentation équivalente à la figure I. 38

d) Association n-aire

Une association n-aire est une relation entre n classes. Mathématiquement, une association n-aire entre les classes A_1, A_2 … A_n est un sous-ensemble R de $A_1 \times A_2 \cdots \times A_n$, donc un ensemble de n-uplets de la forme $(a_1, a_2, \ldots a_n)$ avec $\forall i \in \{1, 2, \ldots n\},\ a_i \in A_i$.

Notation

La figure I. 40 montre une association ternaire R entre les classes X, Y et Z.

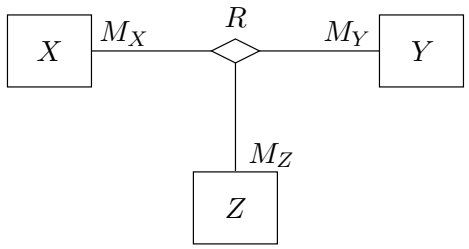

FIGURE I. 40 – Association ternaire entre les classes X, Y et Z

Multiplicités

Les multiplicités M_X, M_Y et M_Z sont associées respectivement aux classes X, Y et Z. La multiplicité associée à une classe pose une contrainte sur le nombre de n-uplets de la relation lorsque les $n-1$ autres valeurs sont fixées. Ces multiplicités posent les contraintes suivantes :

- $\forall y_0 \in Y,\ \forall z_0 \in Z,\ \text{Card}\,\{(x, y_0, z_0) \in R\ ;\ x \in X\} \in M_X$;
- $\forall x_0 \in X,\ \forall z_0 \in Z,\ \text{Card}\,\{(x_0, y, z_0) \in R\ ;\ y \in Y\} \in M_Y$;
- $\forall x_0 \in X,\ \forall y_0 \in Y,\ \text{Card}\,\{(x_0, y_0, z) \in R\ ;\ z \in Z\} \in M_Z$;

Exemple

La figure I. 41 montre une relation ternaire entre les classes `Cours`, `Enseignant` et `Classe`, qui modélise les cours effectués par des enseignants dans différentes classes. De plus, cette association ternaire est une classe-association : à chaque triplet *(cours, enseignant, classe)* est associée une localisation, qui comporte une salle et un horaire.

Les multiplicités posent les contraintes suivantes :

- un enseignant peut faire un même cours à plusieurs classes (multiplicité « `*` » associée à la classe `Classe`) ;
- dans une classe, un cours est effectué par un seul enseignant (multiplicité « `1` » associée à la classe `Enseignant`) ;
- un enseignant peut faire plusieurs cours dans une même classe (multiplicité « `*` » associée à la classe `Cours`). Un enseignant peut donc enseigner plusieurs matières.

e) Extension

Une classe Y *étend* ou *spécialise* une classe X si tout objet de la classe Y est, ou peut être considéré comme, un objet de la classe X. La notation UML est montrée figure I. 42.

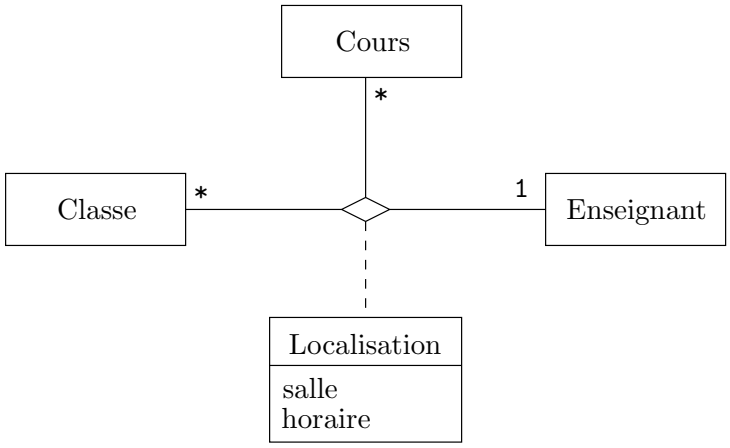

FIGURE I. 41 – Relation ternaire entre les classes `Cours`, `Enseignant` et `Classe`

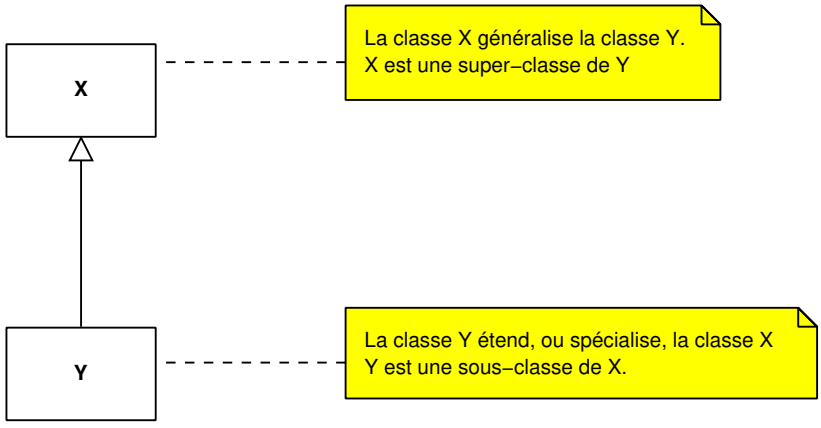

FIGURE I. 42 – La classe Y étend la classe X

Lorsqu'on réalise une extension, on a deux propriétés qui sont satisfaites : la propriété d'héritage et la propriété de substitution.

Propriété d'héritage

Les attributs et opérations définis dans X, qui ne sont pas privés, et qui ne sont pas *redéfinis* dans Y, sont *hérités* dans Y.

Propriété de substitution

La propriété de substitution exprime que tout objet de la classe Y *est* ou *peut être considéré comme* un objet de la clases X. Chaque fois qu'on a besoin d'une instance de X, on peut utiliser à la place une instance de Y.

Exemple

La figure I. 43 montre une classe `Point2D` qui comporte deux attributs `x` et `y` de type entier, représentant respectivement l'abscisse et l'ordonnée d'un point en deux dimensions. La méthode `distanceOrigine` calcule la distance à l'origine d'un point en deux dimensions.

La classe `Point2DColoré` (classe des points colorés) étend la classe `Point2D` : elle comporte un attribut `couleur`, qui représente la couleur d'un point coloré et hérite des attributs `x` et `y`, ainsi que de la méthode `distanceOrigine`.

La classe `Point3D` (classe des points en dimension trois) étend la classe `Point2D` : elle comporte un attribut `z`, qui représente la hauteur d'un point en dimension trois. La méthode `distanceOrigine` est *redéfinie* dans la classe `Point3D`.

Hiérarchies de classes

L'extension permet de gérer la complexité en organisant les classes sous forme de *hiérarchies de classes*.

La figure I. 44 montre une hiérarchie de classes pour différents types de véhicules : aériens, aquatiques et terrestres.

Par exemple, un hydravion est à la fois un avion et un bateau. La classe `Hydravion` étend donc les deux classes `Avion` et `Bateau`.

Héritage de relation

Lorsqu'une classe A_1 hérite d'une classe A, elle hérite de toutes les relations de A. Dans l'exemple figure I. 45, la relation entre les classes A et B est héritée par A_1 : on a donc également une relation entre A_1 et B.

Dans la figure I. 44, la relation entre `Char` et `Chenille` est héritée par `CharAmphibie`. En effet, un char a des chenilles, un char amphibie est un char, donc un char amphibie a des chenilles. De même, un hydravion a des ailes.

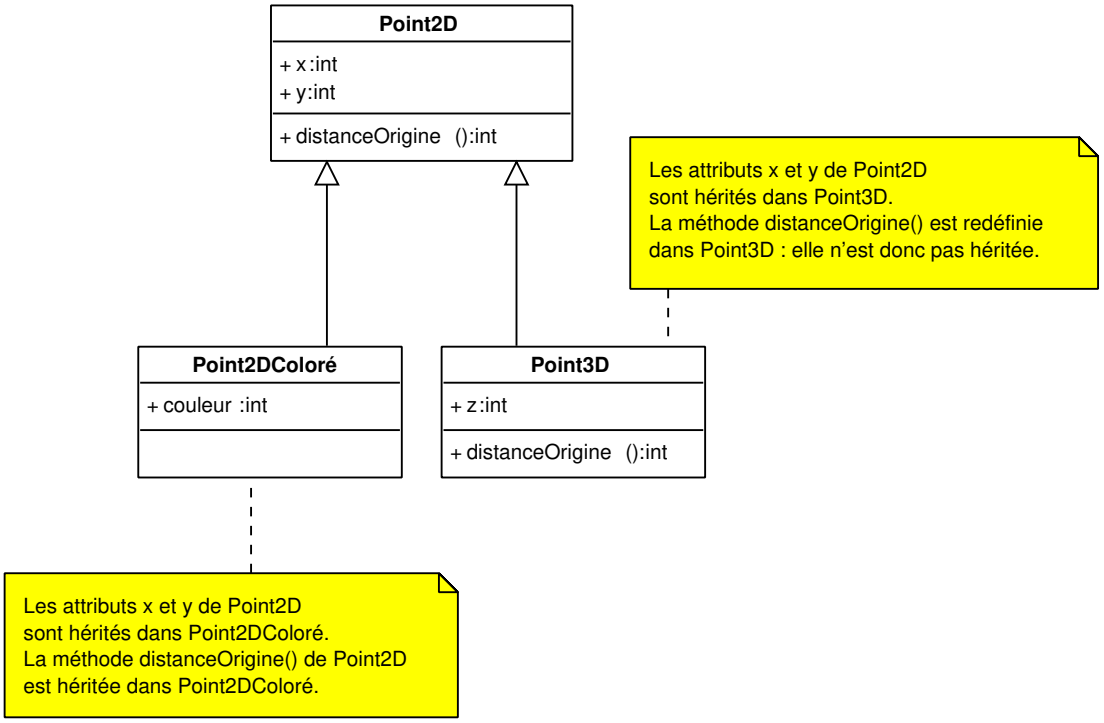

FIGURE I. 43 – Diagramme de classes pour des points

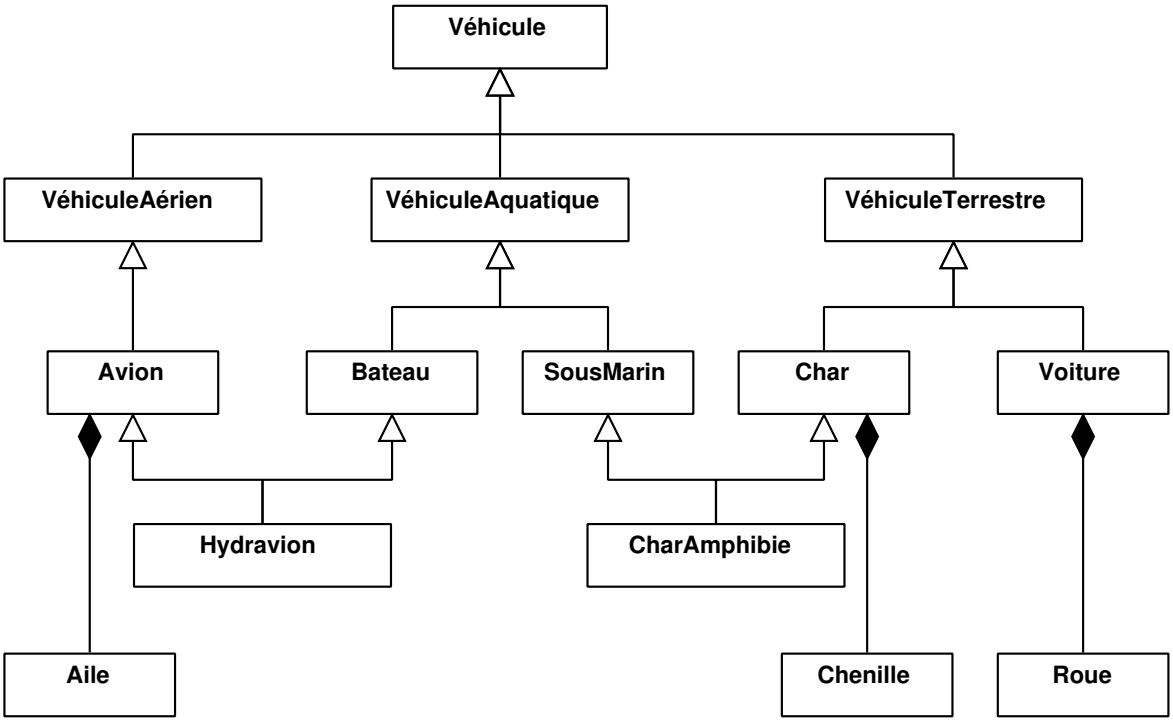

FIGURE I. 44 – Hiérarchie de classes pour différents types de véhicules

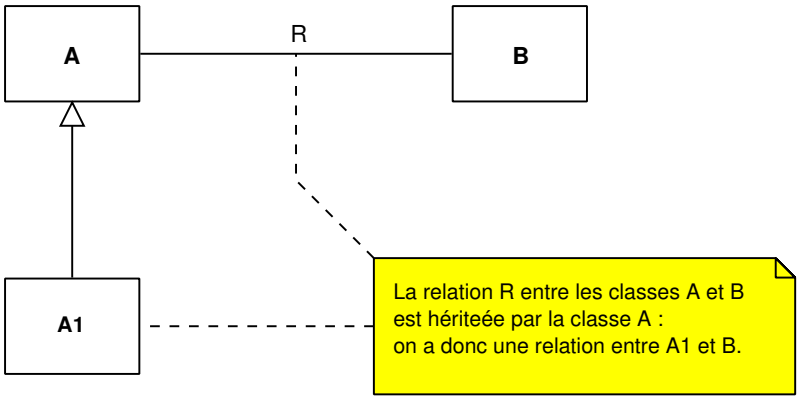

FIGURE I. 45 – La relation R est hérité par A_1

f) Classe abstraite

Une classe abstraite est une classe qui ne peut être instanciée, et qui peut contenir des opérations abstraites. Une opération abstraite est une opération sans implémentation, c'est-à-dire à laquelle ne correspond aucun code. Une classe abstraite peut également contenir des opérations concrètes (c'est-à-dire non abstraites).

Les classes abstraites servent notamment dans les hiérarchies de classes, où elles permettent de regrouper des attributs et opérations communes à plusieurs classes.

En programmation objet (par exemple en Java), une classe concrète doit implémenter les opérations abstraites dont elle hérite.

La figure I. 46 montre deux notations possibles pour la classe abstraite `Véhicule` : soit on met le nom de la classe en italiques, soit on utilise le stéréotype « abstrait ».

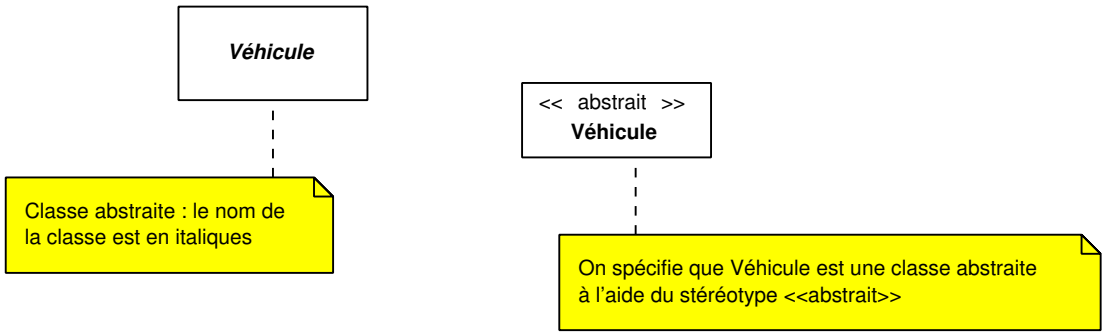

FIGURE I. 46 – Notations pour la classe abstraite `Véhicule`

Exemple

La figure I. 47 montre une hiérarchie de classes qui modélise différentes œuvres, en particulier des livres et des films.

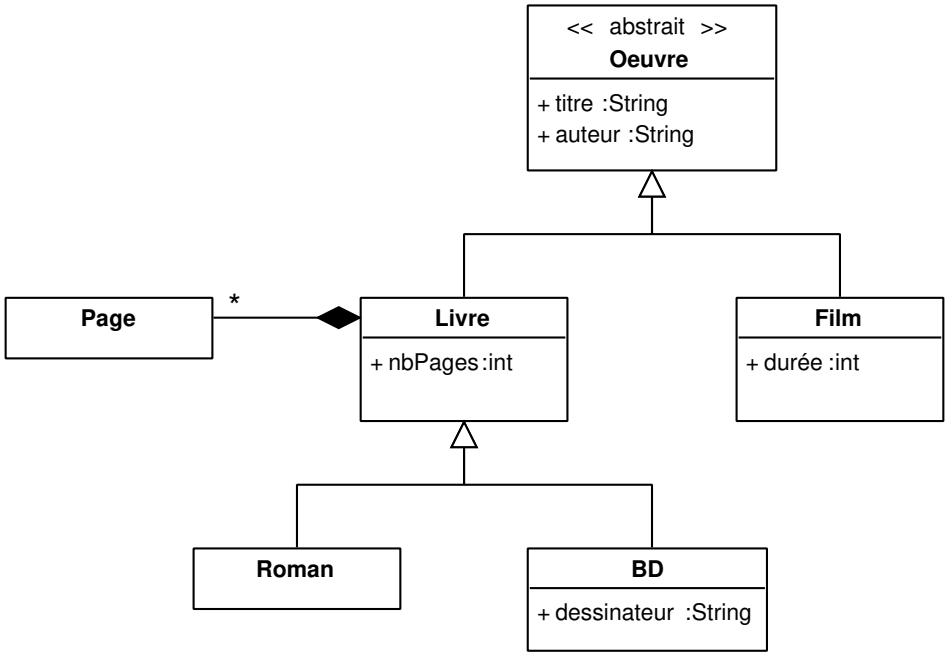

FIGURE I. 47 – Hiérarchie de classes pour des œuvres

On a une classe abstraite `Oeuvre` qui représente les différentes œuvres possibles. Les livres et les films sont des œuvres ; les romans et les bandes dessinées sont des livres.

La classe `Roman` hérite des attributs `titre` et `auteur` de la classe `Oeuvre`. `Livre` est une classe concrète, ce qui autorise la création de livres qui ne sont ni des romans, ni des bandes dessinées.

De plus, un livre est composé de pages, donc un roman et une bande dessinée sont également composés de pages.

g) Interface

Une interface spécifie un ensemble d'opérations qui constituent un service cohérent. Une interface contient uniquement des opérations abstraites, sans implémentation. Une interface est formellement équivalente à une classe abstraite qui ne contient que des opérations abstraites. Une classe *implémente* une interface lorsqu'elle fournit toutes les opérations de l'interface.

La figure I. 48 montre deux notations possibles pour l'interface `Liste`. Cette interface comporte une opération `élément(n:int):Object` qui renvoie le n-ième élément de la liste. La classe `Tableau` implémente `Liste`, donc fournit une méthode `élément(n:int):Object`.

Exemple : les collections Java

L'interface `Collection`, représentée figure I. 49, offre des services pour manipuler des collections d'objets, en particulier des ensembles (sous-interface `Set`) et des listes (sous-interface

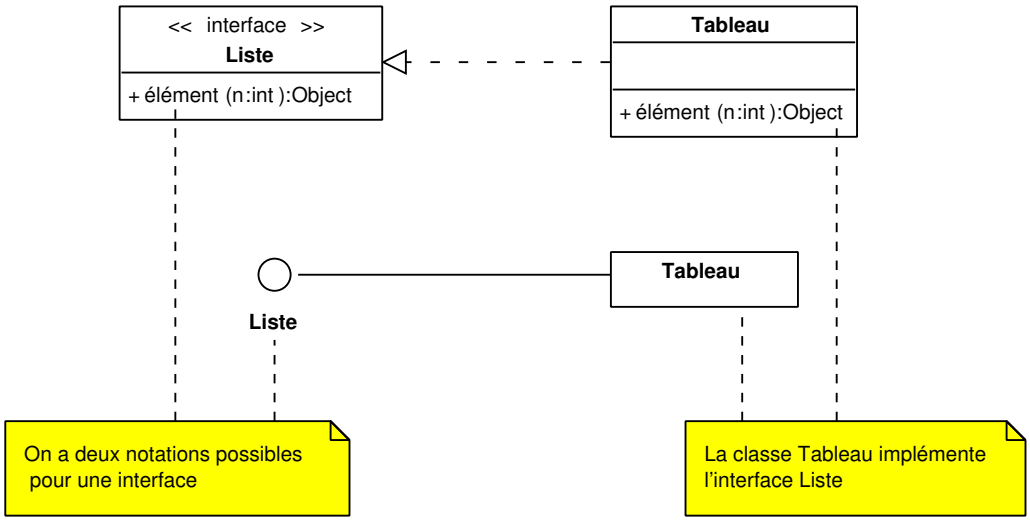

FIGURE I. 48 – La classe `Tableau` implémente l'interface `Liste`

`List`). L'interface `Collection` utilise l'interface `Iterator` (il s'agit du type retourné par la méthode `iterator()`).

La classe `HashSet` implémente l'interface `Set` et la classe `Vector` implémente l'interface `List`.

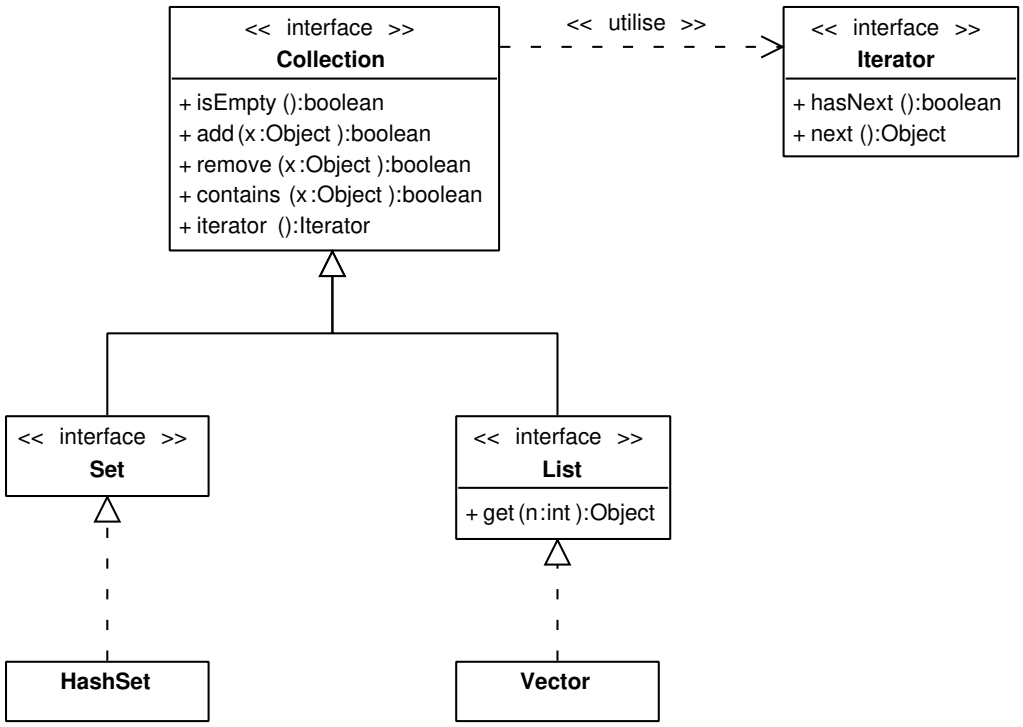

FIGURE I. 49 – Les collections Java

4. Diagrammes de séquence

Les diagrammes de séquence permettent de représenter les interactions entre objets d'un point de vue chronologique.

Les diagrammes de séquence permettent de documenter les cas d'utilisation, en représentant les interactions entre les acteurs et le système.

La figure I. 50 montre un diagramme de séquence décrivant une communication téléphonique entre le système (le système de télécommunications) et deux personnes (l'appelant et l'appelé).

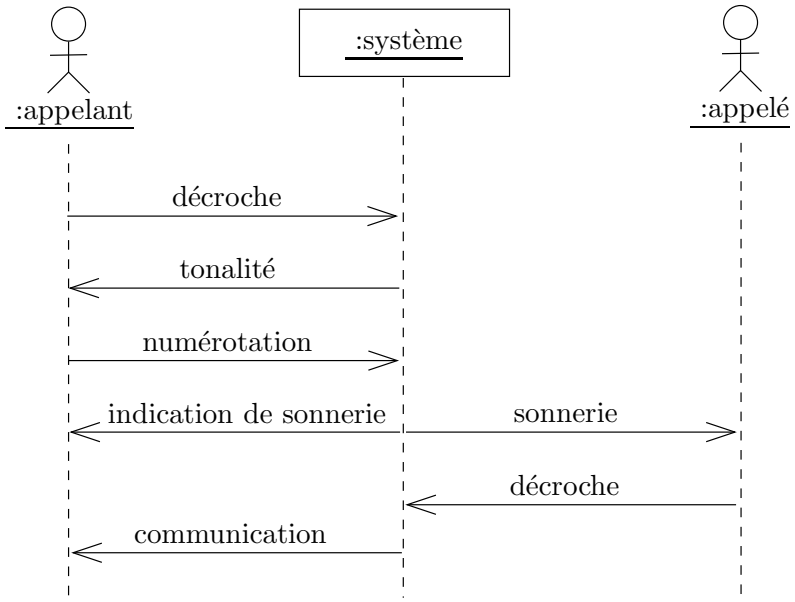

FIGURE I. 50 – Diagramme de séquence représentant une communication téléphonique

Les diagrammes de séquence permettent également de représenter de façon précise les interactions entre objets qui composent le système. Dans ce cas, les messages correspondent à des appels de procédure ou à des signaux.

Les différentes sortes de message

On a trois sortes de messages :

1. Message *synchrone* : correspond à un appel de procédure ou flot de contrôle imbriqué. La séquence imbriquée est entièrement faite avant que l'appelant ne continue : l'appelant est bloqué jusqu'à ce que l'appelé termine.
2. Message *asynchrone* : correspond à un message « à plat », non imbriqué. L'appelant n'est pas bloqué jusqu'à ce que l'appelé termine.
3. Retour de procédure. Les messages qui indiquent des retours de procédure peuvent ne pas être dessinés (ils sont alors implicites).

La notation UML correspondant à chaque sorte de message est indiquée figure I. 51.

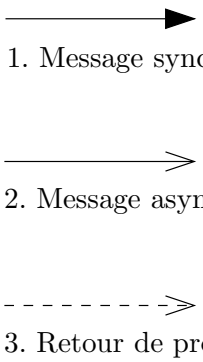

FIGURE I. 51 – Les différentes sortes de message

Période d'activation d'un objet

La période d'activation d'un objet est la période de temps pendant laquelle un objet effectue une action, soit directement, soit par l'intermédiaire d'un autre objet qui lui sert de sous-traitant, par exemple lors d'un appel de procédure.

La période d'activation d'un objet est indiquée par un rectangle sur sa ligne de vie (cf. figure I. 52).

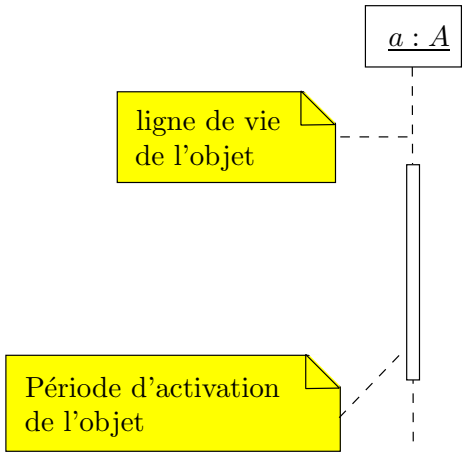

FIGURE I. 52 – Période d'activation d'un objet

La figure I. 53 montre la période d'activation de différents objets lors de deux appels de procédure imbriqués.

La figure I. 53 montre la période d'activation de différents objets lors de deux appels de procédure imbriqués.

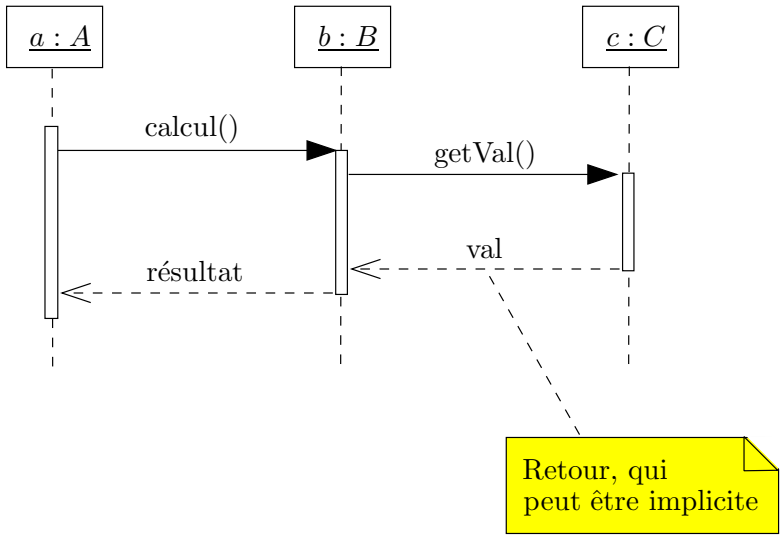

FIGURE I. 53 – Périodes d'activation lors d'appels de procédure

Création et destruction d'objets

On a deux messages particuliers qui permettent de créer et de détruire des objets (cf. figure I. 54).

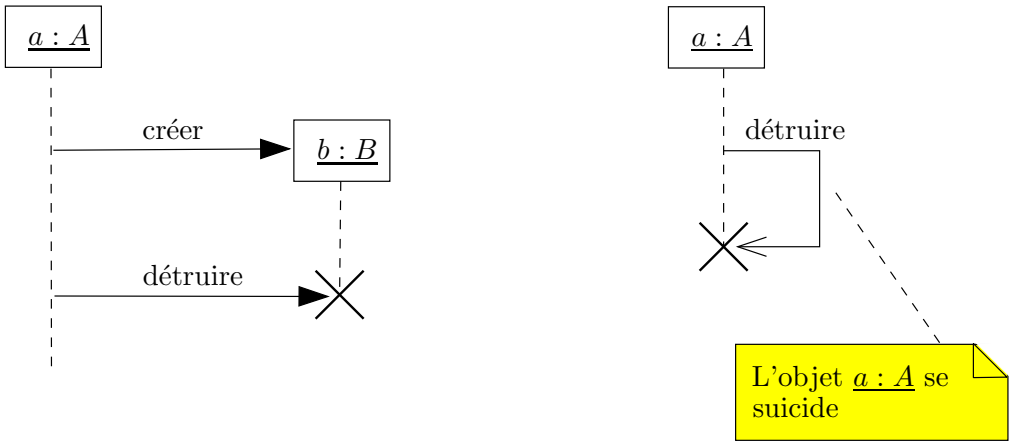

FIGURE I. 54 – Création et destruction d'objets

Conditions

Les messages peuvent être conditionnels : le message est envoyé uniquement si la condition est satisfaite. Dans le diagramme de séquence figure I. 55, si la condition X est satisfaite, l'objet a envoie le message m_1 à l'objet b, sinon il envoie le message m_2 à l'objet c.

La ligne de vie d'un objet peut être dédoublée pour indiquer des actions qui sont effectuées

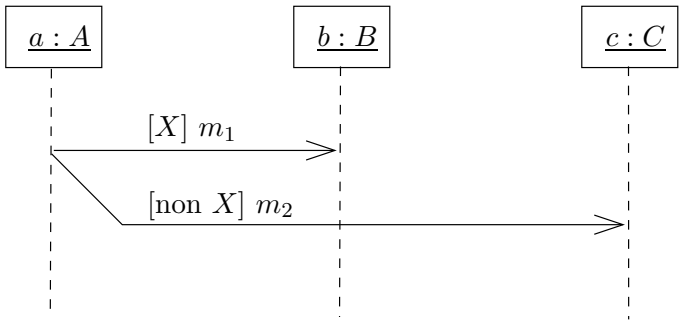

FIGURE I. 55 – Messages conditionnels

suite à un message conditionnel.

Dans le diagramme figure I. 56, si la condition X est satisfaite, alors l'objet a envoie le message m_1 à b, puis b envoie p_1 à c ; sinon a envoie le message m_2 à b, puis b envoie p_2 à c. Ensuite, dans les deux cas, a envoie le message m à b, puis b envoie p à c.

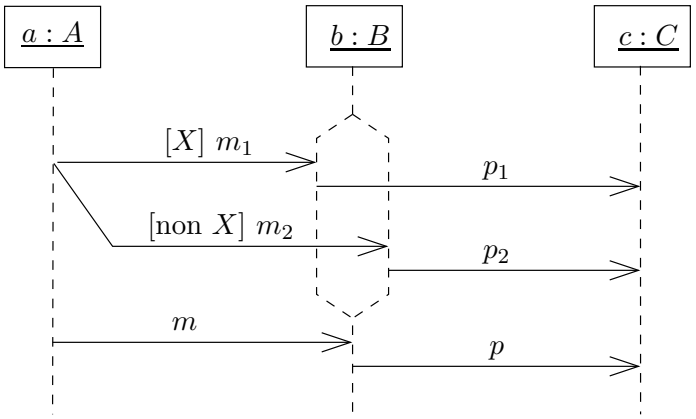

FIGURE I. 56 – Dédoublement d'une ligne de vie

Contraintes temporelles

On peut nommer l'instant d'émission d'un message, ainsi que l'instant de réception. Cela permet de poser des contraintes de temps sur l'envoi et la réception de messages.

Par convention, lorsque l'instant d'émission d'un message est x, l'instant de réception est x'.

La figure I. 57 pose les deux contraintes suivantes :
- il s'écoule moins d'une seconde entre l'envoi des messages m_1 et m_2 ;
- il s'écoule moins de deux secondes entre l'envoi et la réception du message m_3.

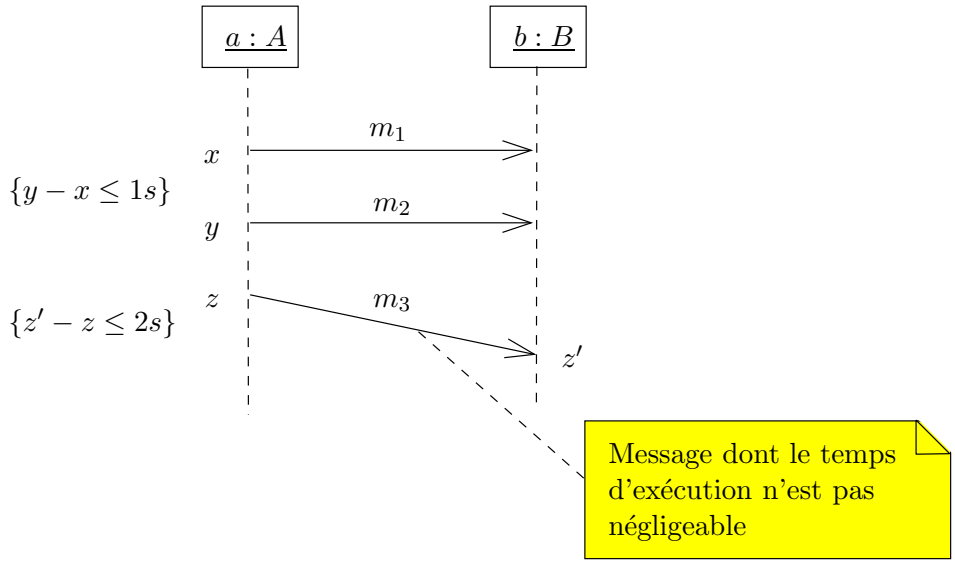

FIGURE I. 57 – Contraintes temporelles sur l'envoi et la réception des messages

Objet actif

Un objet *actif* est un objet qui a son propre flot d'exécution. Les processus et les threads sont des exemples d'objets actifs. Les objets sources d'événements, comme les boutons dans les interfaces graphiques, sont des objets actifs. Les minuteries, comme les « timers » Java, sont des objets actifs. Ces objets permettent d'exécuter une opération soit une seule fois, après un certain intervalle de temps, soit de façon répétée à intervalles de temps donnés.

Un objet actif peut activer, le temps d'une opération, un objet passif, qui peut alors activer d'autres objets passifs. Lorsque l'opération est terminée, l'objet passif redonne contrôle à l'objet qui l'a activé. Dans un environnement multi-tâches, plusieurs objets peuvent être actifs simultanément.

Un objet actif est noté en UML par un rectangle à bordure épaisse.

Dans l'exemple représenté figure I. 58, a et c sont des objets actifs, b est un objet passif. L'objet a active l'objet b par l'appel de procédure `erreur`, puis b envoie le message asynchrone `traitement` à c.

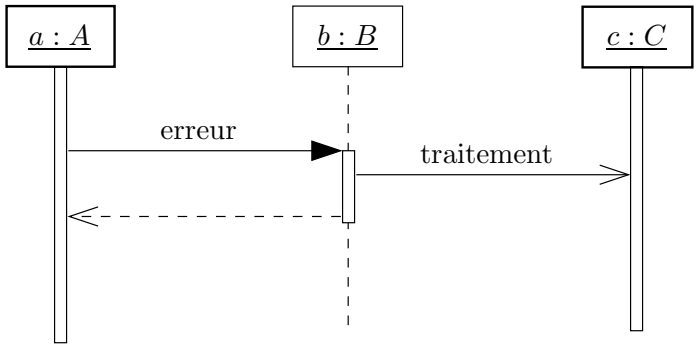

FIGURE I. 58 – Les objets a et c sont actifs

5. Diagrammes de collaboration

Un diagramme de collaboration représente les interactions entre des objets (et éventuellement des acteurs) d'un point de vue spatial. Par opposition aux diagrammes de séquence, les liens entre les différents objets sont explicitement représentés. Pour mettre en évidence la dimension temporelle, les messages envoyés par les différents objets peuvent être numérotés.

La figure I. 59 montre un diagramme de collaboration qui représente une communication téléphonique.

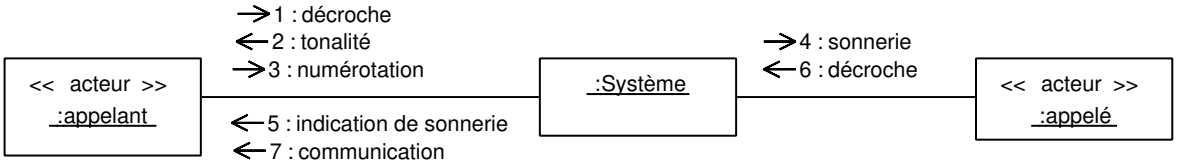

FIGURE I. 59 – Diagramme de collaboration représentant une communication télephonique

Dans cet exemple, les messages sont asynchrones.

La figure I. 60 montre un diagramme de collaboration qui représente l'affichage d'une figure composée de segments.

Pour afficher une figure, on affiche l'ensemble des segments dont la figure est composée. Dans ce diagramme, :Segment est un multi-objet, qui représente l'ensemble des segments dont la figure est composée. Le message « `2 *:afficher` » signifie qu'on envoie le message afficher à chaque instance de la classe `Segment` en relation avec cette figure. Les messages sont des appels et des retours de procédure.

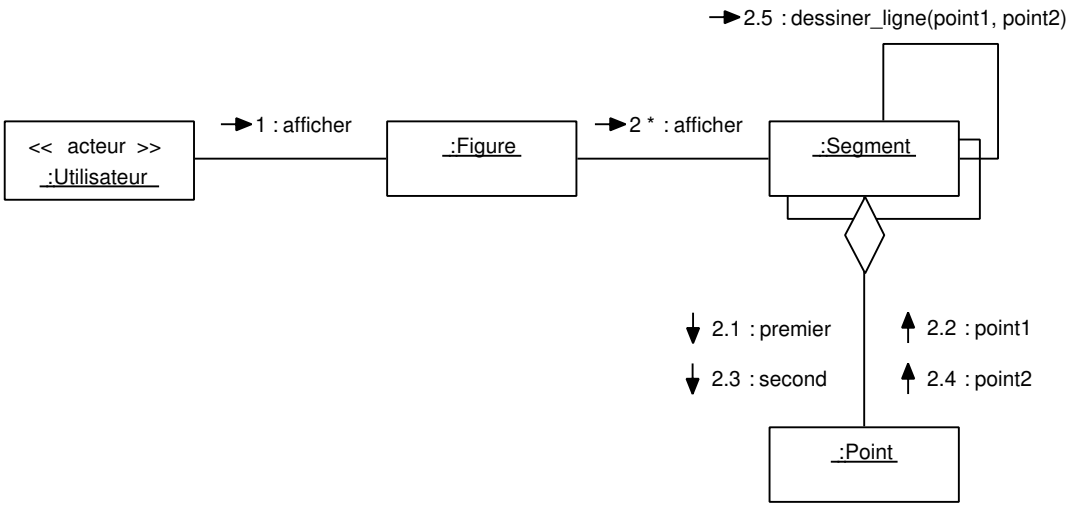

FIGURE I. 60 – Diagramme de collaboration représentant l'affichage d'une figure

6. Diagrammes d'états-transitions

Les diagrammes d'états-transitions permettent principalement de décrire le comportement des objets d'une classe. Il peuvent également décrire les aspects dynamiques d'un cas d'utilisation, d'un acteur, d'un système ou d'un sous-système.

Les diagrammes d'états-transitions d'UML sont inspirés des « Statecharts » de David Harel. Il s'agit d'automates hiérarchiques, qui peuvent être mis en parallèle.

a) État

L'état d'un objet correspond à l'ensemble des valeurs de ses attributs et à l'ensemble des liens qu'il entretient avec d'autres objets.

L'état d'un automate peut être vu comme une abstraction représentant un ensemble d'états de l'objet.

Un état est une condition ou une situation, dans la vie d'un objet, qui dure un certain temps pendant lequel cet objet satisfait une condition, effectue une activité, ou attend un événement.

Si on reprend l'exemple des personnes employées dans des entreprises, on peut considérer les états suivants pour une personne : en activité, à la retraite et sans emploi (cf. figures I. 61 et I. 62).

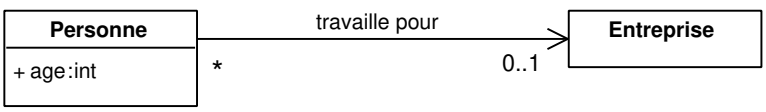

FIGURE I. 61 – Diagramme de classes qui représente l'emploi des personnes

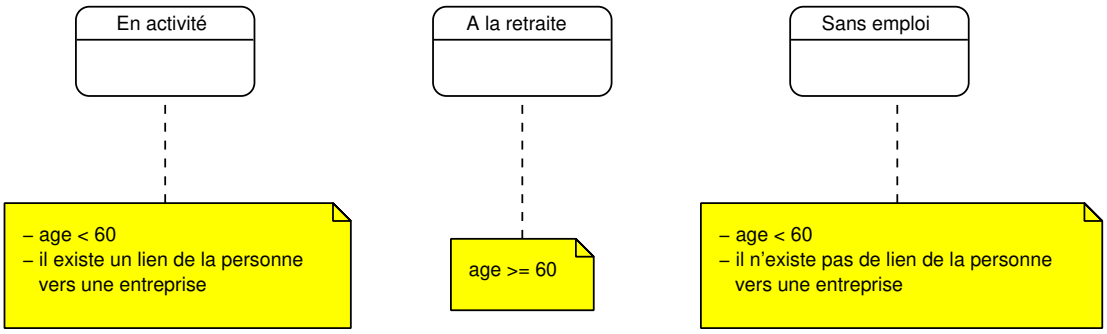

FIGURE I. 62 – Trois états possibles pour une personne

Ici, l'abstraction qui nous intéresse est l'emploi d'une personne. On pourrait s'intéresser à d'autres abstractions.

Etats initial et final

Un état initial est un pseudo-état qui permet de montrer l'état dans lequel un objet se trouve au moment de sa création. Un état final est un pseudo-état qui permet de montrer la fin du comportement d'un objet, en particulier le moment de sa destruction.

La figure I. 63 montre la notation UML pour les pseudo-états initiaux et terminaux.

FIGURE I. 63 – Pseudo-états initial et terminal

b) Transition

Les transitions permettent à un objet de changer d'état, en fonction des événements qu'il reçoit.

Une transition comporte un état source, un état destination, un événement, une condition (appelée garde) et une action.

Supposons que l'objet soit dans l'état A (figure I. 64). Si l'événement se produit et si la condition est vraie, alors l'objet effectue l'action et passe dans l'état B. Le passage d'un état à l'autre est considéré comme instantané (l'action doit donc également pouvoir être considérée comme instantanée).

Si aucune transition n'est étiquetée par l'événement reçu, alors rien ne se passe, et, en particulier, l'objet ne change pas d'état.

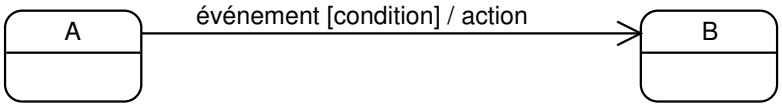

FIGURE I. 64 – Transition entre les états A et B

c) Événement

On a quatre sortes d'événements en UML.

- Un *événement d'appel* (≪ call event ≫) est un événement causé par l'appel d'une opération. Dans ce cas, l'événement est de la forme $op(x_1, x_2, \ldots x_n)$, où op est une opération de la classe.
- Un *événement modification* (≪ change event ≫) est un événement causé par le passage d'une condition de la valeur faux à la valeur vrai, suite à un changement de valeur d'un attribut ou d'un lien.
- Un *événement temporel* (≪ time event ≫) est un événement qui survient quand une temporisation arrive à expiration. Une temporisation peut être relative (délai), ou absolue (spécification de l'heure à laquelle une transition doit être effectuée).
- Un *événement signal* (≪ signal event ≫) est un stimulus asynchrone entre deux objets. Par exemple, un clic de souris est un signal.

Exemple 1

Dans la classe `Personne`, on considère les deux opérations `embauche` et `perteDEmploi` (cf. figure I. 65). La figure I. 66 est un diagramme d'états-transitions associé à la classe `Personne`.

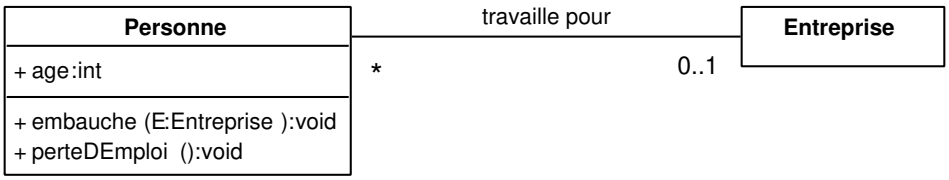

FIGURE I. 65 – Diagramme de classes qui représente l'emploi des personnes

Exemple 2

On considère une machine qui comporte deux boutons : on a un bouton pour mettre la machine sous tension (signal : on) et un bouton pour mettre la machine hors tension (signal : off). Un voyant indique si la machine est sous tension ou hors tension. Après une minute sans utilisation, la machine se met automatiquement hors tension.

Les figures I. 67, I. 68 et I. 69 montrent un diagramme de classes de la machine et des diagrammes d'états-transitions associés aux classes `Voyant` et `Machine`.

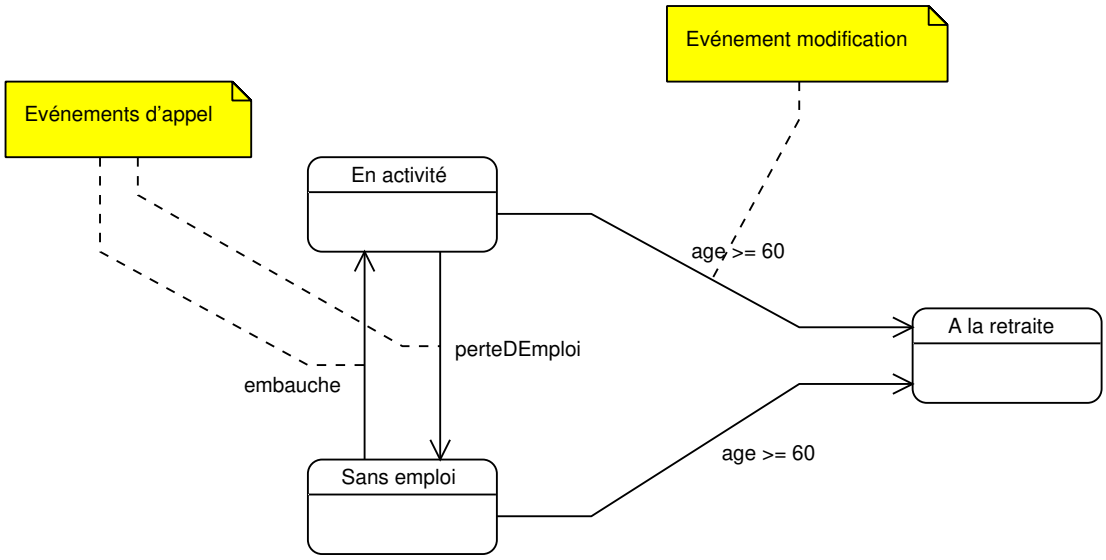

FIGURE I. 66 – Diagramme d'états-transitions associé à la classe `Personne`

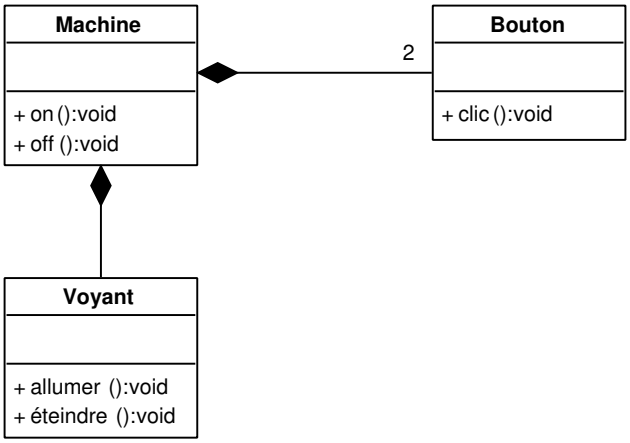

FIGURE I. 67 – Diagramme de classes de la machine

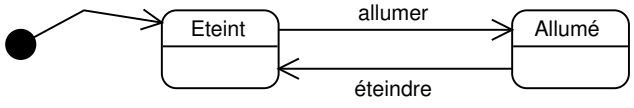

FIGURE I. 68 – Diagramme d'états-transitions associé à la classe `Voyant`

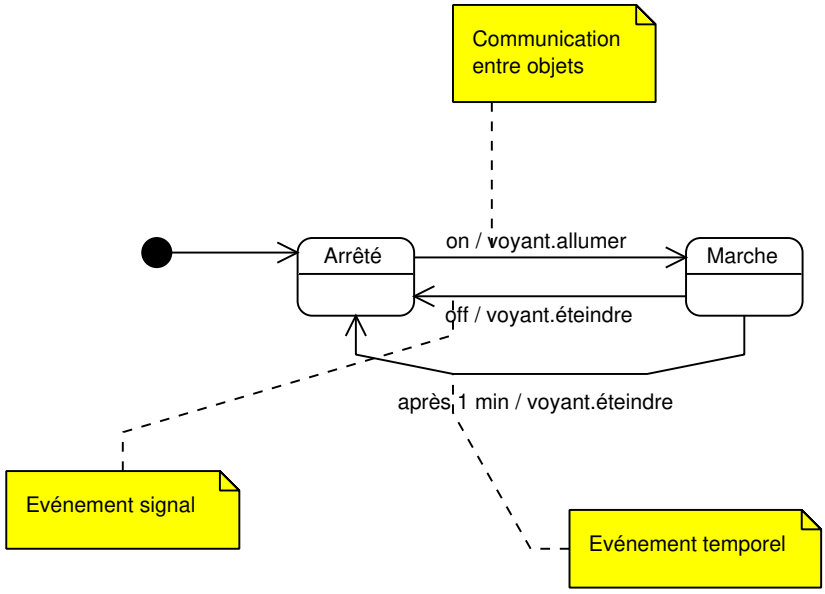

FIGURE I. 69 – Diagramme d'états-transitions associé à la classe `Machine`

Remarque

Les automates considérés en UML sont *a priori* non-déterministes. On peut donc avoir deux transitions étiquetées par le même événement qui partent du même état. Néanmoins, d'un point de vue méthodologique, il est souvent préférable d'utiliser des automates déterministes pour plus de clarté.

d) Garde

Une garde est une condition booléenne notée entre crochets. Une garde est évaluée lorsque l'événement se produit.

Supposons que plusieurs transitions partant du même état A soient déclenchées par le même événement (cf. figure I. 70). Pour que l'automate soit déterministe, il faut que les gardes c_1, c_2 et c_3 soient mutuellement exclusives.

Si aucune condition n'est vérifiée, alors rien ne se passe et l'objet ne change pas d'état.

Remarque

Il ne faut pas confondre un événement de changement et une garde. Un événement de changement est un événement qui déclenche la transition lorsque la condition passe à vrai ; une garde est une condition booléenne évaluée lorsque l'événement se produit.

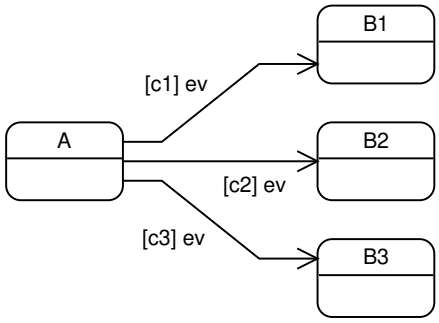

FIGURE I. 70 – Transitions gardées, étiquetées par le même événement

e) Action et activité

Une action consiste en la génération d'un signal ou l'invocation d'une opération. Une action est considérée comme *instantanée* (c'est-à-dire dont le temps d'exécution est négligeable) et *atomique* (c'est-à-dire non interruptible).

Une activité correspond à une opération qui prend un temps non négligeable et peut être interrompue.

Les actions sont généralement associées aux transitions, mais on peut également les associer aux états. On peut en particulier :

- spécifier une action à effectuer lorsqu'on entre dans un état : `entry/ action` ;
- spécifier une action à effectuer si un événement survient : `on événement/ action` (événement « interne ») ;
- spécifier une action à effectuer lorsqu'on sort d'un état : `exit/ action` ;
- spécifier une *activité* effectuée lorsqu'on est dans l'état : **`do/ activité`**. L'activité peut prendre un certain temps, et être interrompue.

Remarque

Le déclenchement d'un événement interne n'entraîne pas l'exécution des actions d'entrée et de sortie.

Dans l'exemple représenté figure I. 71, on suppose qu'on entre dans l'un des états `E1` ou `E2`, qu'on reçoit une suite d'événements e, puis qu'on sort de l'état.

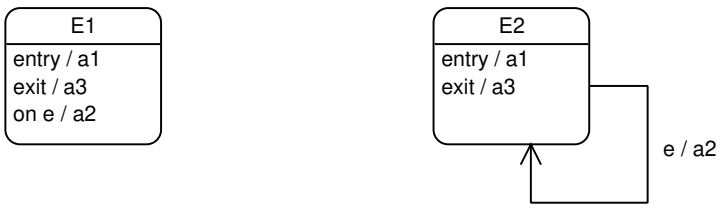

FIGURE I. 71 – Action associée à un événement interne et à un événement d'une transition

Dans l'état `E1`, les actions a_1 et a_3 sont effectuées une seule fois, lors de l'entrée et de la sortie

de l'état. Les séquences d'actions générées sont donc de la forme $a_1\, a_2^*\, a_3$. Dans l'état `E2`, les actions a_1 et a_3 sont effectuées à chaque réception de l'événement e. Les séquences d'actions générées sont donc de la forme $a_1\, (a_3\, a_2\, a_1)^*\, a_3$.

f) États composites

Un état *composite* est un état qui se décompose en plusieurs sous états. Les états composites permettent de structurer les automates pour les rendre plus lisibles.

Les états composites permettent en particulier de factoriser des transitions similaires qui partent de plusieurs états.

On reprend le diagramme d'états-transitions de la figure I. 66. On introduit l'état composite « `Age inférieur à 60` » pour les états « `En activité` » et « `Sans emploi` ». On peut ainsi factoriser les deux transitions « `age >= 60` » (cf. figure I. 72).

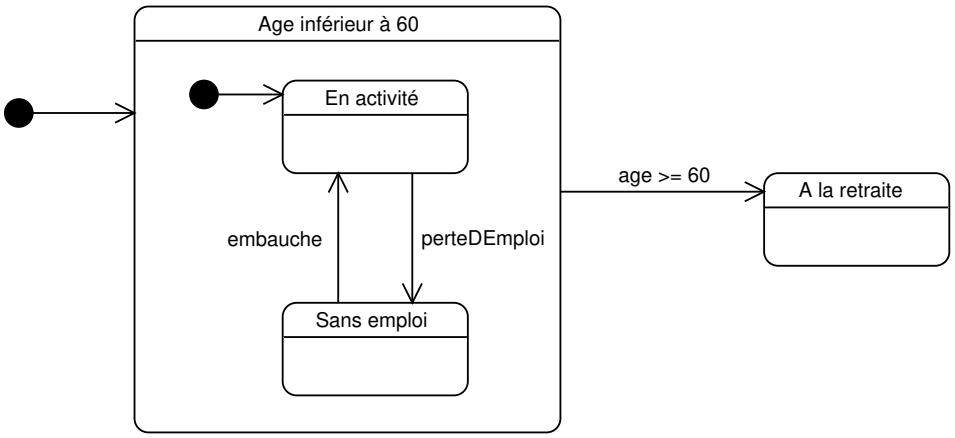

FIGURE I. 72 – Exemple d'état composite

Il peut exister des transitions entre différents niveaux de l'automate, mais il est préférable de limiter les transitions entre différents niveaux.

Lorsque deux transitions peuvent être effectuées, l'une sur un sous-état et l'autre sur l'état englobant, c'est la transition sur le sous-état (la plus « spécialisée ») qui est effectuée.

Dans l'exemple figure I. 73, si l'objet est dans l'état X et si l'événement e survient, alors l'objet passe dans l'état Y (et non dans l'état Z ou T).

g) Indicateurs d'historique

L'indicateur d'historique, noté « $\textcircled{H}$ », est un pseudo état qui permet de mémoriser le dernier état visité d'un automate pour y retourner ultérieurement.

La figure I. 74 est un diagramme d'états-transitions d'une machine à laver. Si on souhaite ouvrir la porte lorsque la machine est en marche, il faut interrompre le programme et la

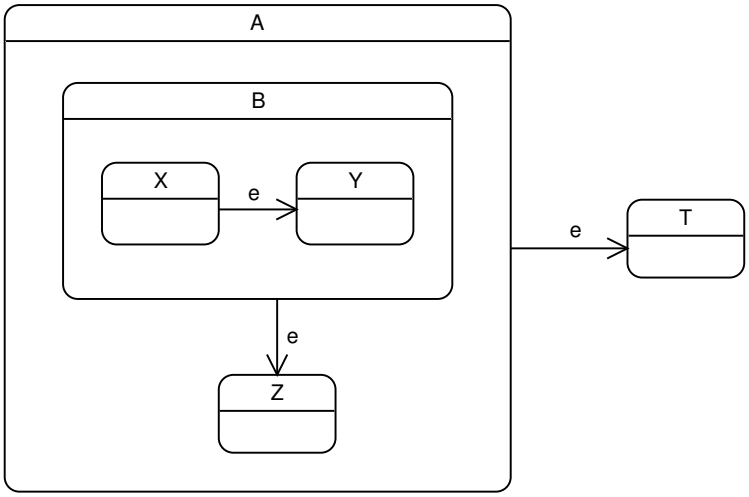

FIGURE I. 73 – L'objet passe de l'état X à l'état Y

vidanger. Une fois la porte refermée, la machine peut reprendre son cycle dans l'état où elle s'était arrêtée lors de l'interruption.

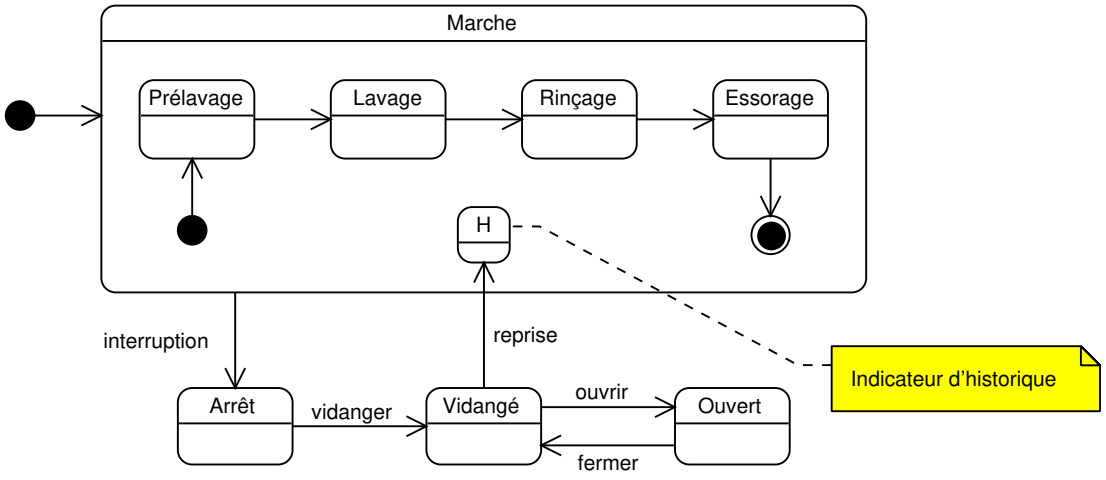

FIGURE I. 74 – Diagramme d'états-transitions d'une machine à laver

La transition `reprise` a pour cible l'indicateur d'historique. Lorsque la transition `reprise` est effectuée, l'automate reprend son exécution dans l'état où il se trouvait lorsque la transition `interruption` s'est produite, c'est-à-dire l'un des états `Prélavage`, `Lavage`, `Rinçage` ou `Essorage`.

L'indicateur d'historique à un niveau quelconque, noté « $\textcircled{H^*}$ », est un pseudo état qui permet de mémoriser le dernier état visité, à un niveau d'imbrication quelconque, pour y retourner ultérieurement.

Par exemple, dans la figure I. 75, lorsque la transition `reprise` est effectuée, l'automate

revient dans l'état où il était lors de la transition `interruption`, à un niveau quelconque, autrement dit, dans l'état A, B ou C.

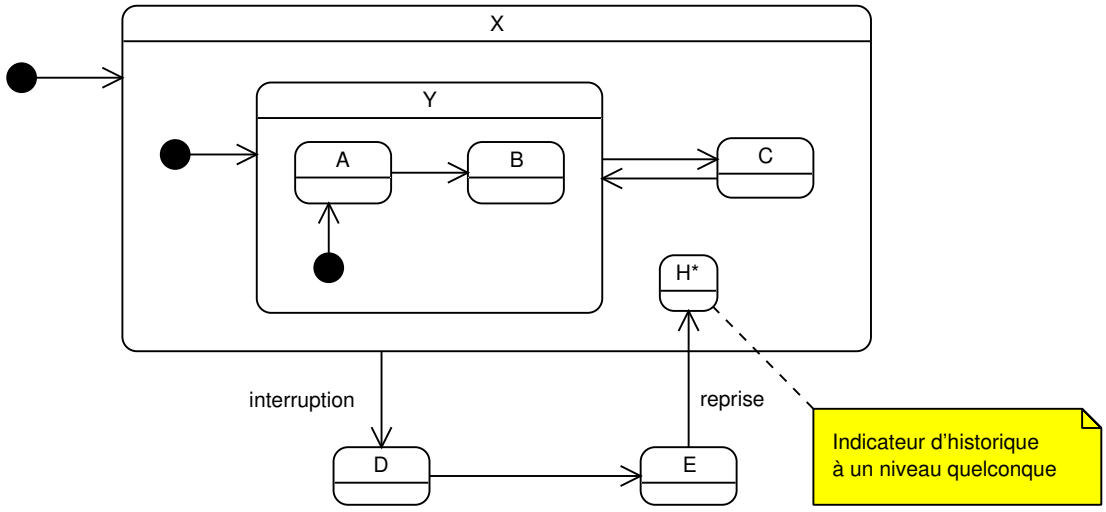

FIGURE I. 75 – Indicateur d'historique à niveau quelconque

h) Automates en parallèle

A‘ l'intérieur d'un état, plusieurs automates peuvent s'exécuter en parallèle. Chaque sous-automate a un état initial et un certain nombre d'états terminaux.

L'activité d'un tel état se termine lorsque tous les sous-automates parviennent à un état final.

Lorsqu'un événement se produit, toutes les transitions qui peuvent être effectuée sont effectuées.

La figure I. 76 montre un exemple d'état qui contient deux sous-automates s'exécutant en parallèle.

Si A_1 est dans l'état X et A_2 est dans l'état A, et si l'événement e_1 se produit, alors A_1 passe dans l'état Y et A_2 passe dans l'état B. Si A_1 est dans l'état Y et A_2 est dans l'état B, et si l'événement e_2 se produit, alors A_1 passe dans l'état X et A_2 reste dans l'état B.

L'automate de la figure I. 76 est équivalent à l'automate « aplati » représenté figure I. 77.

Un cas typique où on peut utiliser des automates en parallèle est lorqu'un objet est formé de la composition ou de l'agrégation d'autres objets, en particulier si le comportement de l'objet composite correspond à la mise en parallèle des comportements des composants.

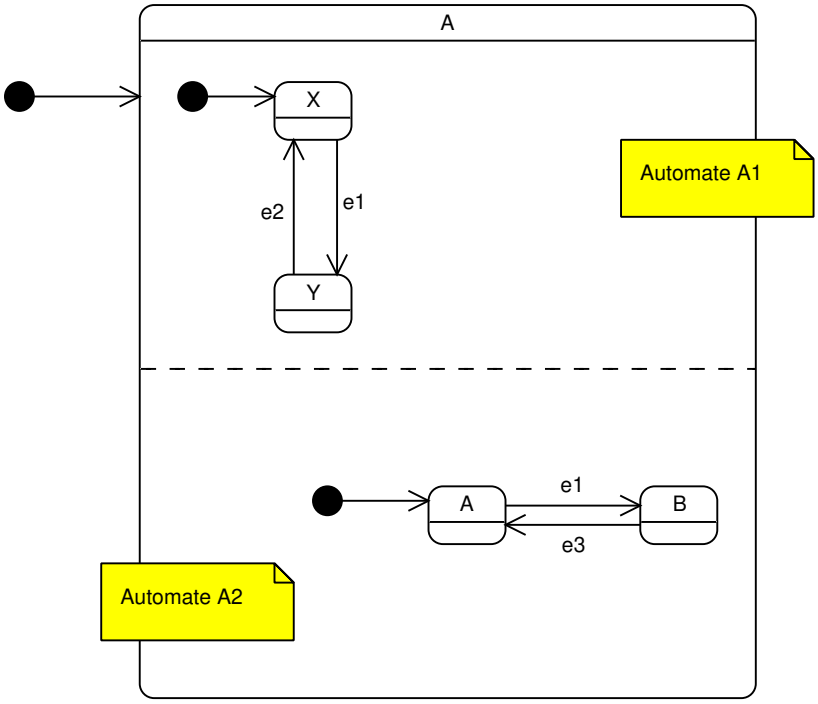

FIGURE I. 76 – L'état *A* contient deux automates qui s'exécutent en parallèle

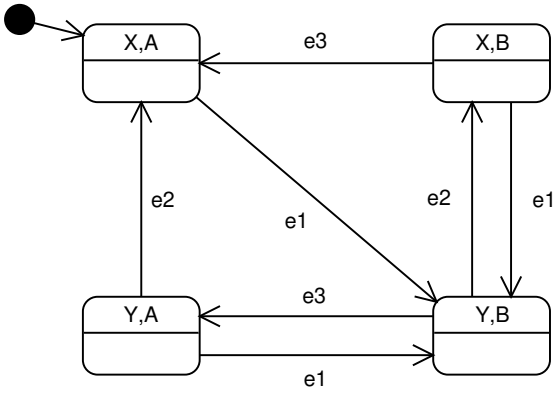

FIGURE I. 77 – Automate « aplati » équivalent

Les diagrammes d'activités permettent de modéliser le comportement d'une méthode ou le déroulement d'un cas d'utilisation, en représentant un enchaînement d'activités.

La figure I. 78 est un diagramme d'activités représentant la commande d'un produit.

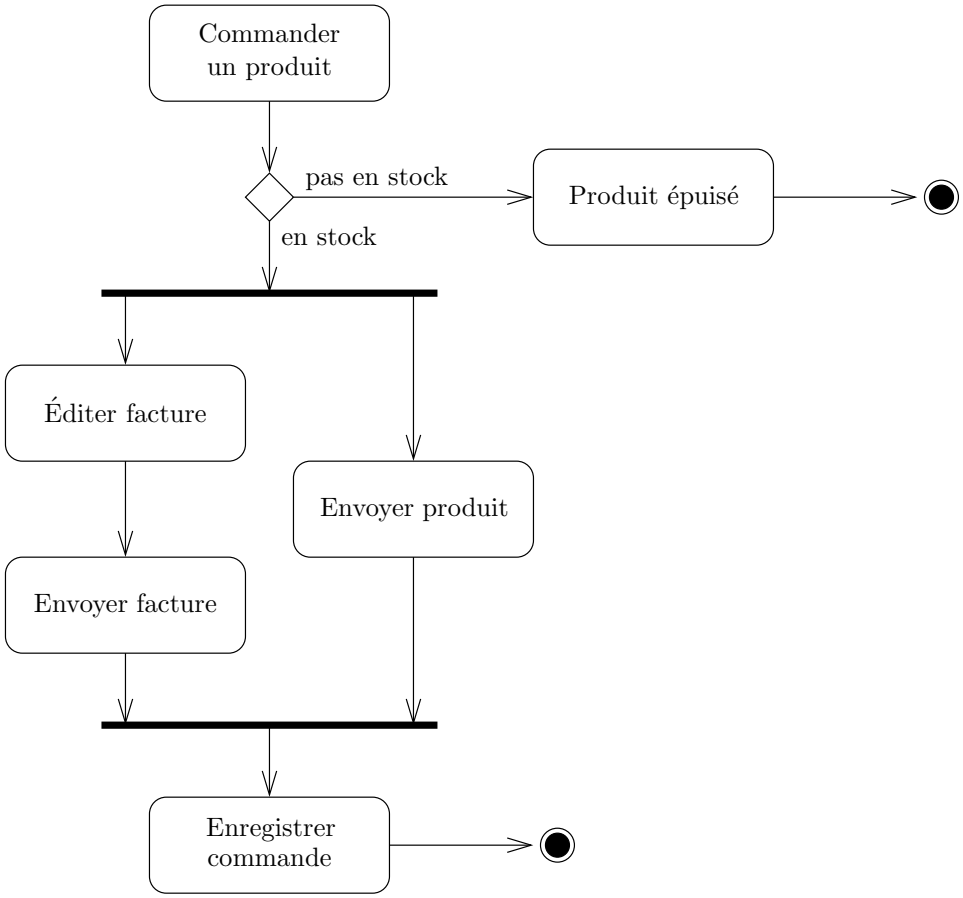

FIGURE I. 78 – Diagramme d'activités pour une commande de produits

7. Diagrammes de composants

Un diagramme de composants permet de décrire l'architecture physique d'une application en termes de modules : fichiers sources, librairies exécutables... etc.

La figure I. 79 est un diagramme de composants d'une application construite à partir de deux fichiers Java et d'une librairie mathématique.

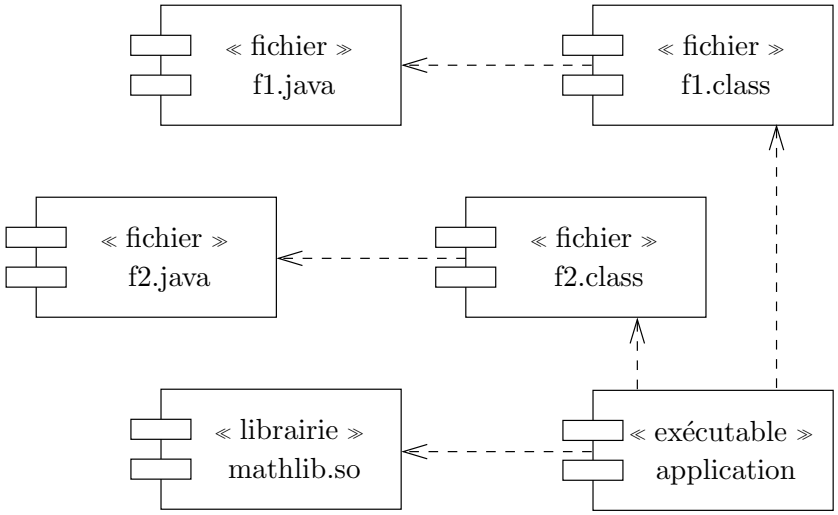

FIGURE I. 79 – Diagramme de composants d'une application

La figure I. 80 est un diagramme de composants d'un serveur Oracle utilisant une base de données qui stocke un catalogue de produits.

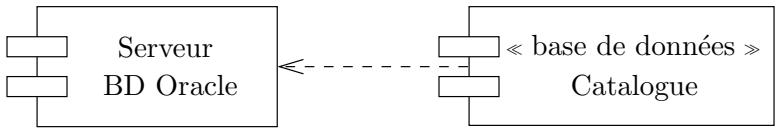

FIGURE I. 80 – Diagramme de composants d'une application

8. Diagrammes de déploiement

Le diagramme de déploiement décrit la disposition physique des matériels et la répartition des composants sur ces matériels.

La figure I. 81 est un diagramme de déploiement d'une application utilisant une base de donnée distante.

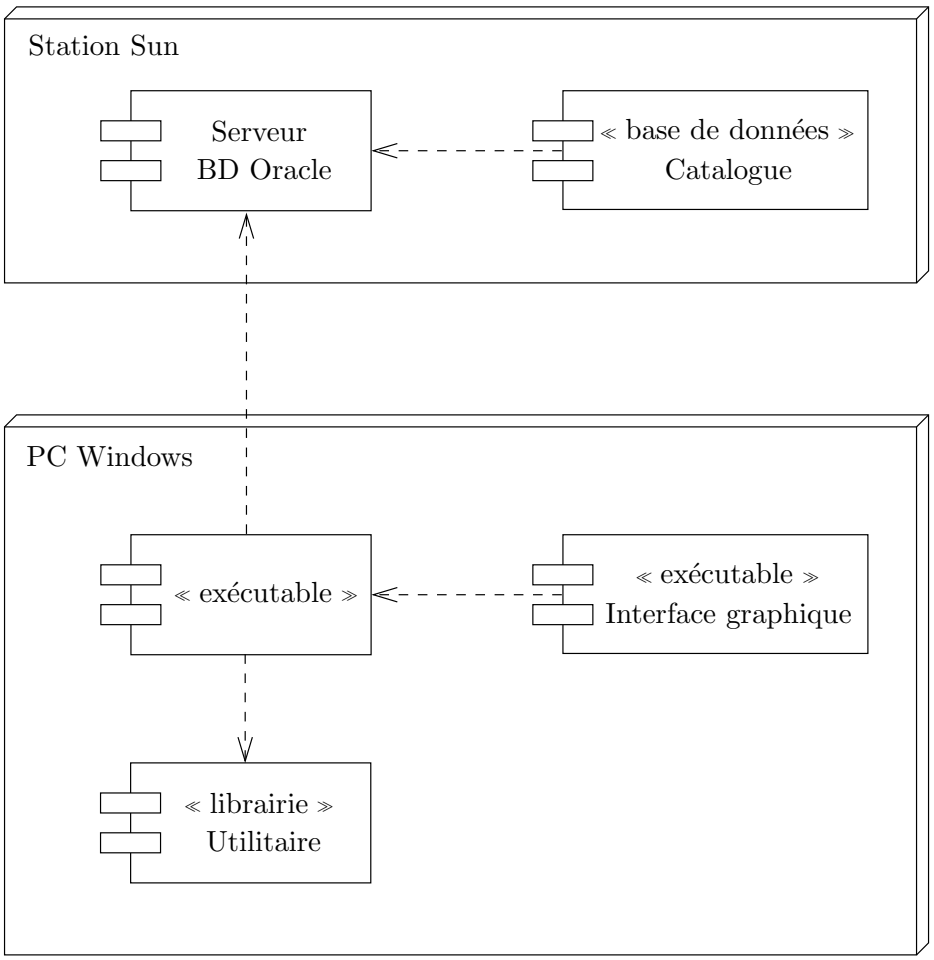

FIGURE I. 81 – Diagramme de déploiement d'une application

Chapitre II

Analyse

Les objectifs de l'analyse sont les suivants :
- comprendre les besoins du client ;
- effectuer une étude de faisabilité ;
- obtenir une bonne compréhension du domaine ;
- éliminer le maximum d'ambiguïtés du cahier des charges ;
- obtenir une première ébauche de la structure du système.

Dans ce chapitre, on s'intéresse d'abord à l'expression des besoins, qui a pour but de comprendre et reformuler les besoins des utilisateurs, puis à mettre en évidence les concepts significatifs qui interviennent dans le système.

1. Expression des besoins

a) Gestion des besoins

La gestion des besoins consiste à comprendre, exprimer et mémoriser les besoins du système sous une forme compréhensible par le client et l'équipe de développement. Les besoins ne sont pas figés une fois pour toutes au début du projet, mais sont amenés à évoluer, en particulier si on utilise un modèle de cycle de vie itératif.

Il est important de bien contrôler la gestion et l'évolution des besoins, car un tiers des problèmes rencontrés lors du développement de logiciels viennent de cette étape. On rencontre en particulier les problèmes suivants :
- les utilisateurs peuvent fournir des informations erronées ;
- les besoins du système peuvent être incomplets ;
- les besoins des utilisateurs évoluent d'une façon incontrôlée.

Ces problèmes ont des répercussions importantes sur la qualité du logiciel, le budget, les délais de livraison. Pour les éviter, il est important d'impliquer les utilisateurs dans l'étape d'expression des besoins.

b) Types de besoins

On distingue en général les besoins fonctionnels, qui concernent les fonctionnalités du logiciel, et les besoins non fonctionnels.

Parmi les besoins non fonctionnels, on trouve :
- la fiabilité (robustesse, possibilité de récupération après une panne) ;
- la facilité d'utilisation (ergonomie, aide, documentation) ;
- efficacité (temps de réponse) ;
- portabilité ;
- maintenabilité (facilité à corriger des erreurs, à faire des améliorations ou des adaptations du logiciel) ;
- effort de validation (tests, couverture des tests).

c) Analyse des besoins

L'analyse des besoins consiste à comprendre les besoins du client. Ces besoins sont, en général, décrits par un cahier des charges, qui décrit le système et son environnement.

On détermine les besoins fonctionnels et non fonctionnels par une analyse du cahier des charges. Il s'agit d'une analyse de texte. On peut par exemple déduire les aspects fonctionnels des verbes utilisés dans le cahier des charges. Les substantifs permettront de déduire les objets du système (cf. modélisation objet). Il faut tenir compte du fait que la langue naturelle est souvent imprécise ou ambiguë : certains mots ne sont pas pertinents, plusieurs mots peuvent avoir la même signification, plusieurs concepts peuvent correspondre au même mot. De plus, le cahier des charges comprend souvent une grande part d'implicite : il suppose que le lecteur a une bonne connaissance du domaine, ce qui n'est pas toujours le cas.

Le but de cette analyse de texte est de préciser le cahier des charges et de lever certaines ambiguïtés. En particulier, il peut être utile de rédiger un glossaire qui définit les principaux termes utilisés.

L'analyse du cahier des charges est en générale insuffisante pour comprendre entièrement les besoins. Le cahier des charges est en effet non seulement imprécis et ambigu, mais également incomplet, voire contradictoire. La compréhension des besoins nécessite alors d'utiliser d'autres méthodes qui consistent à communiquer avec les utilisateurs. Cette communication peut prendre différentes formes :
- on peut organiser des entrevues avec les utilisateurs, les techniciens, les gestionnaires ;
- on peut demander aux utilisateurs de remplir des questionnaires, en particulier lorsqu'il est nécessaire d'obtenir des données précises auprès d'un grand nombre de personnes. Cette méthode présente un inconvénient : les personnes interrogées sont souvent peu motivées pour répondre.
- on peut observer les activités des utilisateurs sur site, afin de mieux comprendre leurs besoins. Inconvénient : certains utilisateurs peuvent modifier leur méthode de travail lorsqu'ils se savent observés.

À l'issue de l'analyse des besoins, les documents suivants peuvent être produits :

- un cahier des charges, qui comporte :
  - une description de l'environnement du système : les machines, le réseau, les périphériques (imprimantes, capteurs...), l'environnement physique... etc.
  - le rôle du système ;
- un glossaire, qui définit les principaux termes utilisés dans le cahier des charges ;
- un manuel utilisateur ;
- un document de spécification globale, précisant les besoins fonctionnels et non fonctionnels du système.

d) Expression des besoins fonctionnels

Plusieurs diagrammes UML peuvent être utilisés pour exprimer les besoins fonctionnels du système :

- cas d'utilisation ;
- diagrammes de séquence ;
- diagrammes d'états-transitions.

Rédaction de cas d'utilisation

La rédaction de cas d'utilisation permet de reformuler les besoins fonctionnels du système. Les cas d'utilisation ont été introduits en 1986 par Ivar Jacobson.

La rédaction de cas d'utilisation permet de :

- comprendre et clarifier le cahier des charges, à travers une reformulation ;
- structurer les besoins.

Un cas d'utilisation décrit ce que le système doit faire, du point de vue des utilisateurs, sans décrire comment cela sera implémenté. Il s'agit donc d'un document de spécification, et non de conception.

Un cas d'utilisation :

- décrit une manière spécifique d'utiliser le système ;
- regroupe une famille de scénarios d'utilisation du système.

On utilise le langage naturel et on utilise la terminologie des utilisateurs. Les cas d'utilisations doivent être compréhensibles à la fois par les utilisateurs et les développeurs. Un cas d'utilisation regroupe une famille de scénarios d'utilisation suivant des critères fonctionnels. C'est une abstraction du dialogue entre les acteurs et le système. Les scénarios seront décrits ensuite par d'autres diagrammes (diagrammes d'interaction, diagrammes d'états-transitions).

Pour exprimer ces besoins fonctionnels, on commence par définir les acteurs.

Un acteur est quelqu'un ou quelque chose qui interagit avec le système (personne, environnement, autre système). Plus précisément, il s'agit du *rôle* joué par quelqu'un ou quelque chose qui interagit avec le système.

La détermination des acteurs permet de préciser les limites du système.

Lorsqu'il y a beaucoup d'acteurs, on les regroupe par catégories. On peut distinguer :
- les acteurs *principaux* : personnes qui utilisent les fonctions principales du système ;
- les acteurs *secondaires* : personnes qui effectuent des tâches administratives ou de maintenance ;
- le matériel externe : les dispositifs matériels périphériques ;
- les autres systèmes (avec lesquels le système interagit).

Chaque cas d'utilisation peut comprendre :
- une pré-condition (condition qui doit être satisfaite pour que la fonctionnalité puisse être utilisée) ;
- une description de la suite des interactions entre le système et les acteurs, en distinguant les différents scénarios possibles. On décrit en particulier la répétition des comportements et les échanges d'information.
- une post-condition (condition satisfaite une fois l'interaction terminée) ;
- des exceptions (cas exceptionnels, ou ce qui se passe si une pré-condition n'est pas satisfaite).

La description des cas d'utilisation est faite en langue naturelle. Par rapport au cahier des charges, cette description doit être plus précise, mieux structurée, et décrire précisément les interactions entre le système et l'utilisateur. Il s'agit donc d'une reformulation et d'une clarification du cahier des charges.

Pour être plus précis, cette description évitera par exemple l'utilisation de pronoms impersonnels (« on »), mais utilisera à la place « l'utilisateur » ou « le système ». Cette description évitera les synonymes, les termes flous ou non définis dans le glossaire.

Pour que la description soit mieux structurée que le cahier des charges, elle pourra comporter des énumérations à différents niveaux, par exemple :

1. Le système demande à l'utilisateur d'entrer son code secret.
   1.1. L'utilisateur entre son code secret et termine en appuyant sur « valider ».
   1.2. Le système vérifie que le code secret entré par l'utilisateur est correct.
      1.2.1. Si le code est correct...
      1.2.2. Si le code est incorrect...
2. ...

Les cas d'utilisation structurent les différentes fonctionnalités du logiciel. Cette structuration peut être utilisée dans l'ensemble du développement du logiciel, de la spécification aux tests. Ils permettent en particulier d'aider à la planification des versions successives, si on utilise un développement incrémental du logiciel. Par exemple, un incrément peut consister à réaliser (de la spécification aux tests) un ou plusieurs cas d'utilisation.

Diagrammes de séquence

Les diagrammes de séquence permettent d'illustrer les différents cas d'utilisations : chaque cas d'utilisation est accompagné de plusieurs diagrammes de séquence qui montrent différents scénarios de fonctionnement du système logiciel.

On utilise des « diagrammes de séquence système », c'est-à-dire des diagrammes de séquence qui représentent les interactions entre les acteurs et le système, sous forme d'actions et de réactions. Les actions internes au système ne sont, en général, pas représentées.

Le diagramme de séquence représenté figure II. 1 décrit un scénario d'utilisation du téléphone.

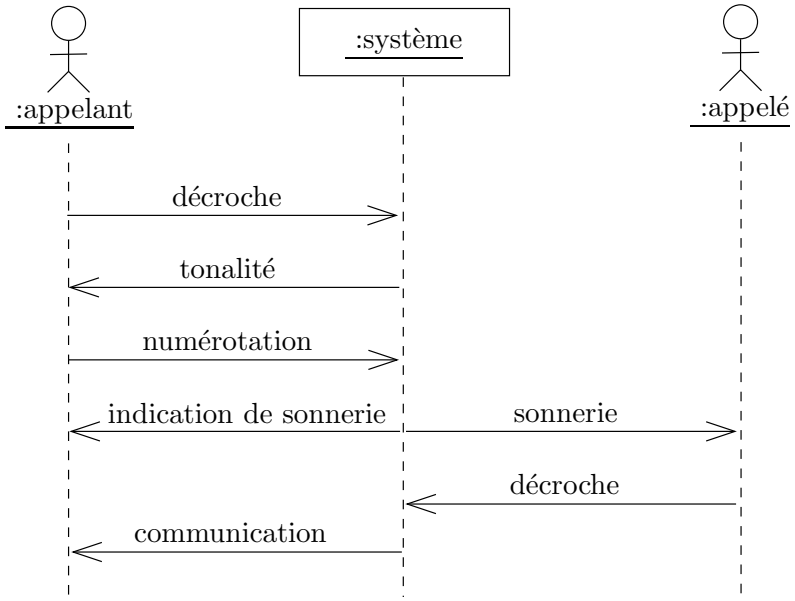

FIGURE II. 1 – Diagramme de séquence représentant une communication téléphonique

Ce scénario peut se dérouler sous un certain nombre d'hypothèses, ou pré-conditions :
- le téléphone de l'appelé est correctement branché ;
- la numérotation correspond à un numéro de téléphone correct ;
- le téléphone de l'appelant est correctement branché ;
- l'appelé n'est pas déjà en communication ;
- l'appelé décroche le téléphone ;
- ... etc.

Pour décrire de tels scénarios, il est nécessaire de définir les événements auxquels le système doit réagir et les événements que le système peut générer.

Les diagrammes de séquences doivent être compréhensibles par le client. Pour cela, comme pour les cas d'utilisation, on utilise la terminologie des utilisateurs.

Diagrammes d'états-transitions

Les diagrammes d'états-transitions permettent de spécifier le comportement général du système, en précisant :
- les états possibles du système ;
- tous les enchaînements possibles d'opérations.

On utilise souvent des « diagrammes d'états-transitions de protocole ». Il s'agit de diagrammes d'états-transitions dans lesquels on ne représente que les événements auxquels le système peut réagir, accompagnés de conditions. On ne représente pas les actions internes au système. Ces diagrammes permettent de définir l'ensemble des séquences d'événements intéressantes pour utiliser une fonctionnalité du système.

2. Diagramme de classes d'analyse

Au cours de l'analyse, on est amené à élaborer un *diagramme de classes d'analyse*. C'est un des modèles les plus importants à créer lors d'une analyse objet, qui constitue une source d'inspiration pour la conception du système.

a) Diagramme de classes d'analyse

On élabore un ou plusieurs diagrammes de classes d'analyse, qui représentent les classes significatives du système. Ces classes, appelées « classes conceptuelles », représentent des objets du monde réel, qui existent indépendamment du système.

Les classes conceptuelles ne sont pas destinées à représenter des objets purement logiciels, comme des classes Java, des fenêtres, une interface graphique, des tables d'une base de données... etc.

Ces diagrammes de classes sont constitués de classes dans lesquelles aucune opération n'est définie. Ils comportent plus précisément :
- les classes conceptuelles ;
- les attributs de ces classes ;
- les associations entre ces classes.

L'objectif est d'identifier les éléments du monde réel qui sont utiles pour le système, et de faire abstraction des détails inutiles.

b) Classes conceptuelles

Une classe conceptuelle comporte :
- un *nom* ;
- une *intention*, qui indique ce que représente cette classe, en incluant le rôle de ses attributs ;
- une *extension*, qui décrit l'ensemble des objets qui sont instances de la classe, à un instant donné.

Pour identifier les classes conceptuelles, une technique consiste à repérer les noms et les groupes nominaux dans le cahier des charges. Il faut ensuite effectuer une classification de ces noms : le cahier des charges peut comporter des synonymes (deux mots différents représentent le même concept) et des ambiguïtés (un même mot représente deux concepts distincts).

Il est souvent utile d'introduire des classes qui permettent de représenter des objets qui sont des descriptions ou des spécifications d'autres objets. Par exemple, si on manipule des

produits, on utilise une classe `Produit`. Il peut être utile de stocker des informations sur ces produits, comme le prix, la référence... etc., qui sont communes à plusieurs produits similaires. On introduit pour cela une classe `SpécificationProduit`. On a une relation entre les produits et leur spécification : à chaque produit est associé sa spécification et à une spécification est associée l'ensemble des produits ayant cette spécification.

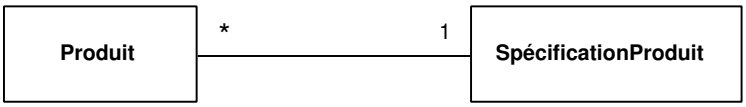

FIGURE II. 2 – Produit et spécification de produit

c) Relations entre les classes conceptuelles

Lors de l'identification des relations entre classes conceptuelles, il faut éviter d'avoir un nombre trop important de relations. Pour cela, on se concentre sur les liens qui devront être stockés un certain temps dans le système. D'autre part, on évite de faire apparaître les relations *dérivées*, c'est-à-dire qui peuvent se déduire d'autres relations.

Parmi les associations, il faut identifier les agrégations et les compositions. On introduit ces relations lorsqu'on veut faire apparaître qu'un élément « appartient » à, ou « est contenu » dans un autre élément. On a une composition lorsque le composant ne peut pas être partagé entre deux parties.

Une fois qu'on a identifié un certain nombre de classes, il est utile de construire des généralisations. Pour cela, on identifie les éléments (attributs, associations) qui sont communs à plusieurs classes ; si ces éléments correspondent à une abstraction cohérente, on peut définir une classe qui comporte ces éléments et qui généralise les classes de départ.

d) Attributs

Dans le diagramme de classes d'analyse, les attributs doivent être d'un type primitif (entier, réel, booléen, chaîne de caractères...). Lorsqu'on a besoin de stocker des valeurs de type non primitif, il est en général préférable d'introduire une classe et une association. Lors de l'implantation, par exemple en Java ou C++, ces associations seront codées par des attributs de type non primitif. On introduira en particulier des classes lorsque :

- la valeur à stocker est composée de plusieurs valeurs ;
- des opérations sont associées à cette valeur (par exemple une vérification que la valeur est correcte) ;
- il s'agit d'une abstraction représentant plusieurs autres types.

Par exemple, il peut être utile de stocker un montant sous la forme d'un objet d'une classe, en particulier si le paiement peut être effectué dans différentes monnaies, et que des conversions sont nécessaires.

Chapitre III

Conception

Dans ce chapitre, on s'intéresse à la conception de logiciels, en particulier à la conception architecturale, qui consiste à déterminer la structure générale du système, et à la conception détaillée.

1. Architecture logicielle

L'architecture logicielle décrit la structure générale du logiciel en constituants de haut niveau, ainsi que l'interaction entre ces éléments.

a) Description d'une architecture

Le modèle des $4+1$ vues de Kruchten représenté figure III. 1 permet de décrire l'architecture d'un système selon un ensemble de cinq vues.

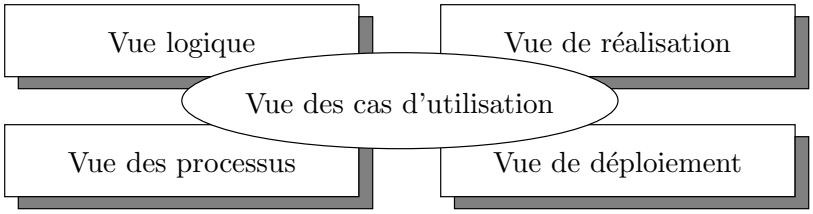

FIGURE III. 1 – Le modèle des $4+1$ vue de Philippe Kruchten

Vue logique

La vue logique de l'architecture décrit l'organisation du système en sous-systèmes, couches, paquetages, classes et interfaces. Un paquetage regroupe un ensemble de classes et d'interfaces ; un sous-système représente un sous-ensemble du système, fournissant un ensemble d'interfaces et d'opérations.

Vue de réalisation

La vue de réalisation concerne l'organisation des différents fichiers (exécutables, code source, documentation...) dans l'environnement de développement, ainsi que la gestion des versions et des configurations. Cette vue peut être en partie décrite à l'aide d'un diagramme de composants.

Vue des processus

La vue des processus représente la décomposition en différents flots d'exécution : processus, fils d'exécution (threads). Cette vue est importante dans les environnements multi-tâches.

Vue de déploiement

La vue de déploiement décrit les différentes ressources matérielles et l'implantation du logiciel sur ces ressources. Cette vue concerne les liens réseau entre les machines, les performances du système, la tolérance aux fautes et aux pannes. Cette vue peut être décrite à l'aide d'un diagramme de déploiement.

Vue des cas d'utilisation

Cette vue, décrite par un diagramme de cas d'utilisation, sert à motiver et justifier les différents choix architecturaux.

Paquetages UML

Les paquetages UML permettent de regrouper un ensemble d'éléments de modélisation, en particulier des classes et des interfaces.

On peut utiliser les paquetages pour décrire la vue logique de l'architecture.

La figure III. 2 montre la notation UML pour un paquetage P.

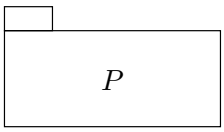

Figure III. 2 – Notation UML pour un paquetage P

b) Principes de conception architecturale

Pour réaliser une architecture logique, découpée en paquetages, on peut appliquer certains principes de conception architecturale.

Couplage

Le couplage est une mesure du degré selon lequel un paquetage est lié à d'autres paquetages. Des paquetages fortement couplés ne sont pas indiqués, car ils seront sensibles à beaucoup de

modifications. De plus, ils sont plus difficiles à comprendre isolément, et difficiles à réutiliser. Un principe de conception architecturale consiste donc à concevoir des paquetages faiblement couplés.

Cohésion

La cohésion mesure les liens, la cohérence entre les différents services proposés par le paquetage. Il est préférable de réaliser des paquetages ayant une forte cohésion, car ils sont plus faciles à comprendre et à réutiliser.

Variations

Au cours du développement d'un logiciel, certaines parties sont rapidement stables, et d'autres au contraire subissent de nombreuses modifications. Le principe de *protection des variations* consiste à éviter qu'un paquetage utilisé par de nombreux autres paquetages ne subisse trop de variations. En d'autres termes, un paquetage utilisé par de nombreux autres paquetages doit être rapidement stable.

Un exemple de « mauvais » paquetage

Le paquetage Java `java.util` est un exemple de « mauvais » paquetage. La documentation de ce paquetage indique :

> *Le paquetage java.util contient les collections, le modèle d'événements, des utilitaires de date et heure, d'internationalisation et des classes utilitaires diverses comme un « string tokenizer » [analyseur lexical], un générateur aléatoire et un tableau de bits.*

On remarque que la cohésion de ce paquetage n'est pas très forte : il propose un certain nombre d'utilitaires n'ayant aucun rapport entre eux.

Dans le reste de ce paragraphe, nous examinons deux exemples d'architecture logique : l'architecture en couches et l'architecture « modèle – vue – contrôleur ».

c) Architecture en couches

Le logiciel est organisé en couches. Une couche regroupe un ensemble de classes et propose un ensemble cohérent de services à travers une interface.

Les couches sont ordonnées : les couches de haut niveau peuvent accéder à des couches de plus bas niveau, mais pas l'inverse. Dans une architecture *étanche*, une couche de niveau n ne peut accéder qu'aux couches de niveau $n-1$.

La figure III. 3 montre l'architecture en couches d'un système comportant une interface graphique et communiquant avec une base de données.

Le contenu de chaque couche est le suivant :

— Couche A : présentation (interface graphique).

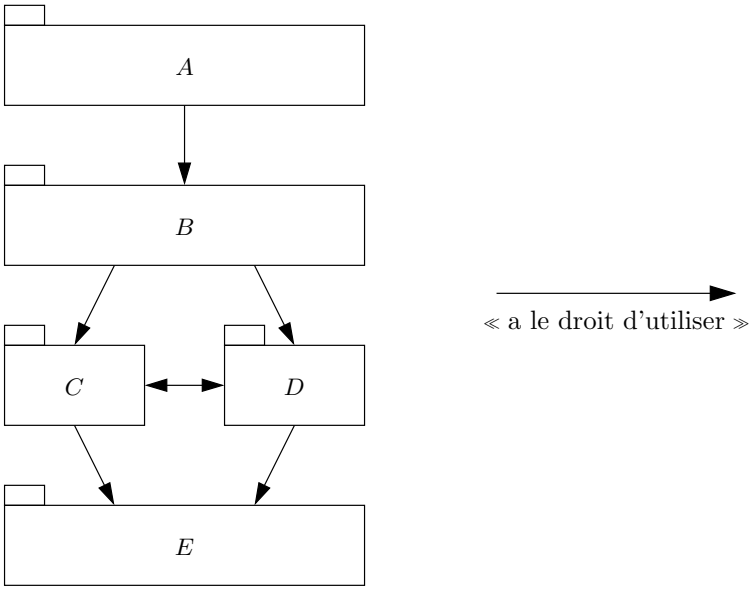

FIGURE III. 3 – Architecture en couches d'une application

— Couche B : couche ≪ application ≫, qui contient les classes qui implémentent les principales fonctionnalités du système, et qui réalise la médiation entre l'interface graphique et les couches C et D.
— Couches C et D : deux couches ≪ domaine ≫, contenant les principales classes du domaine d'application (objets ≪ métier ≫), conçues afin d'être indépendantes de l'interface graphique et de l'infrastructure.
— Couche E : couche ≪ infrastructure ≫ comportant des services techniques de bas niveau (par exemple des classes permettant de communiquer avec une base de données).

Avantages d'une architecture en couches

Les avantages d'une architecture en couche sont les suivants :

Maintenance
Le système est plus facilement modifiable. Une modification d'une couche n'affecte pas les couches de niveau inférieur. Une modification d'une couche qui ne modifie pas l'interface publique n'affecte pas les couches de niveau supérieur ou égal.

Réutilisation
Des éléments de chaque couche peuvent être réutilisés. Par exemple, les couches ≪ domaine ≫ peuvent être communes à plusieurs applications.

Portabilité
On confine les éléments qui dépendent du système d'exploitation aux basses couches. On réécrit les couches basses pour chaque système d'exploitation, et les couches hautes sont portables.

Inconvénients d'une architecture en couches

Si on a un grand nombre de couches, et si en plus ces couches sont étanches, l'appel de fonctions de bas niveau est moins efficace, puisqu'il faut traverser toutes les couches pour parvenir à ces fonctions. Il faut donc trouver un compromis entre une bonne encapsulation et une bonne efficacité.

d) Architecture Modèle – Vue – Contrôleur

Les origines

L'architecture « Modèle – Vue – Contrôleur », ou architecture MVC, a été initialement définie pour les interfaces graphiques utilisateurs. Historiquement, cette architecture a été introduite dans le langage SmallTalk en 1980 et son principe fondateur est de découper le logiciel (ou un morceau du logiciel) en trois parties :

— le « modèle » : correspond aux données métier de l'application (par exemple les données d'une station météo comme la température, etc) ;
— la « vue » : correspond à une représentation du modèle (ou d'une partie du modèle). L'intérêt en est qu'un même modèle peut être représenté par des vues multiples (par exemple une température peut être représentée à la fois par une étiquette et par une barre de progression).
— le « contrôleur » : traite les entrées de l'utilisateur et met à jour en conséquence le modèle.

Cette vision classique du MVC se traduit par les flux de traitement/information suivants :

Entrées	$\rightarrow$	Traitement	$\rightarrow$	Sorties
Contrôleur	$\rightarrow$	Modèle	$\rightarrow$	Vue

La vue est donc mise à jour indirectement par le modèle. À l'issue d'une modification, le modèle produit un évènement de notification à la vue qui s'adapte en conséquence (on dit que la vue observe le modèle).

Quant au contrôleur, il n'interagit pas directement avec la vue et se contente de mettre à jour le modèle. En effet, la vue est mise à jour automatiquement à l'issue d'évènements de notification produits par le modèle.

MVC et frameworks de développement

Plusieurs frameworks modernes de développement (comme Php/Zend ou JEE/Struts) sont fondés sur une vision différente de la séparation Modèle / Vue / Contrôle. En effet, dans ces frameworks le contrôleur est au cœur de l'application. Les flux entre les différentes parties de l'application sont souvent traduits par :

Vue $\longleftrightarrow$ Contrôleur $\longleftrightarrow$ Modèle

Dans ce type d'architectures, la vue renvoie les entrées de l'utilisateur au contrôleur qui effectue les traitements correspondant en agissant sur le modèle. Le contrôle produit ensuite les informations nécessaires à l'interface pour sa mise à jour éventuelle. Dans cette architecture, généralement connue par MVC1, les échanges entre Vue et Modèle sont exclus. Toutefois, d'un point de vue conceptuel, il peut s'avérer judicieux de combiner les deux visions du MVC.

MVC en tant que modèle conceptuel

Les deux visions, présentées ci-dessus, sont issues de problèmatiques d'implémentation. D'un point de vue conceptuel, MVC est un moyen de séparer le traitement des entrées, des sorties et des fonctionnalités principales du logiciel.

Modèle

La partie « modèle » comporte les classes principales correspondant aux différentes fonctionnalités de l'applications : données et traitements. Cette partie, indépendante des parties « vue » et « contrôleur », effectue des actions en réponse aux demandes de l'utilisateur (par l'intermédiaire de la partie « contrôleur »), et peut informer la partie « vue » des changements d'états du modèle pour permettre sa mise à jour.

Vue

La partie « vue » comporte les classes relatives à l'interface graphique (ce que l'utilisateur voit). Cette partie est informée des changements d'états du modèle et est mise à jour lors de ces changements d'états.

Contrôleur

La partie « contrôleur » récupère les actions de l'utilisateur (clics sur les boutons et appuis sur les touches) et associe à ces événements des actions qui modifient le modèle.

Interactions

Les principales interactions entre ces trois parties sont les suivantes (cf. figure III. 4) :

- le contrôleur, en fonction des événements qu'il reçoit de l'utilisateur, effectue des modifications du modèle ;
- ces modifications du modèle sont ensuite transmises à l'interface graphique ;
- le modèle peut notifier l'interface d'éventuels changements en vue que celle-ci soit mise à jour automatiquement sans passer par un contrôleur.

Les avantages d'une architecture MVC sont les suivants :

Vues multiples
: On peut facilement gérer et afficher plusieurs vues du même modèle.

Portabilité
: Si on mélange le code de l'application avec le code de l'interface, le portage sur d'autres plates-formes est plus difficile. Le portage de l'interface sur d'autres plates-formes doit être possible sans modifier le code du noyau de l'application.

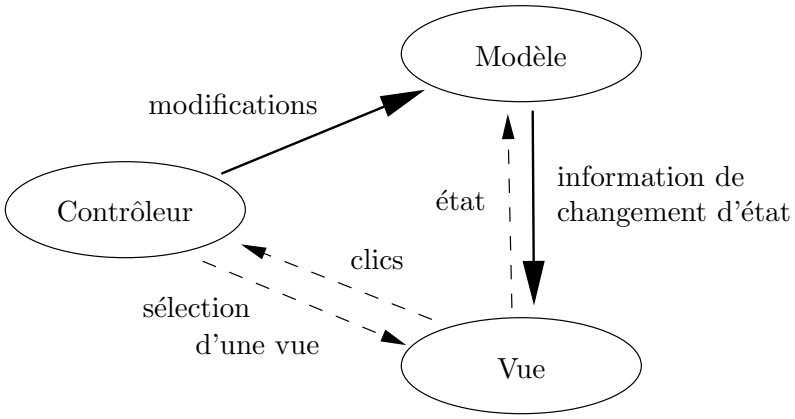

FIGURE III. 4 – Architecture MVC

Évolution

L'ajout de nouvelles fonctionnalités relatives à l'interface graphique, comme l'ajout de lignes dans un menu ou l'ajout de boutons, peut se faire plus facilement. Ces modifications sont également possibles lorsque le logiciel s'exécute : de nombreux logiciels permettent la personnalisation de leur interface à l'exécution.

2. Conception objet

Le but de cette étape est de réaliser la conception détaillée du logiciel. On se concentre dans ce paragraphe sur une conception « objet ». Un travail important de cette étape consiste à réaliser un diagramme de classes de conception. Il s'agit d'un diagramme de classes logicielles, inspiré du diagramme de classes d'analyse. Par rapport au diagramme de classes d'analyse, des classes, des attributs et des associations peuvent être ajoutés, modifiés, voire supprimés. Le diagramme de classes logicielles peut ensuite être traduit dans un langage de programmation objet.

a) Affectation des responsabilités

Une activité de cette étape consiste à affecter les responsabilités aux objets. Les responsabilités sont de deux types : connaissance et comportement.

Les connaissances peuvent être :
- la connaissance de données encapsulées ;
- la connaissance d'objets connexes ;
- la connaissance d'éléments qui peuvent être dérivés ou calculés.

Les comportements peuvent être :
- la réalisation d'une action, comme effectuer un calcul ou créer un objet ;
- le déclenchement d'une action d'un autre objet ;
- la coordination des activités d'autres objets.

Le but de ce travail est de permettre de préciser le contenu des classes, en détaillant les méthodes de chaque classe et en déterminant comment les objets interagissent pour réaliser certaines actions.

b) Principes de conception

Pour réaliser une bonne conception, on applique certains principes de conception.

Principe de l'expert

Une tâche est effectuée par un objet qui possède, ou a accès à, l'information nécessaire pour effectuer cette tâche.

Principe du créateur

On affecte à la classe B la responsabilité de créer des instances de la classe A s'il existe une relation entre A et B (typiquement, si B est une agrégation d'objets de A, ou si B contient des objets de A). L'idée est de trouver un créateur qui est connecté à l'objet créé.

Principe de faible couplage

Le couplage est une mesure du degré selon lequel un élément est relié à d'autres éléments. Une classe faiblement couplée s'appuie sur peu d'autres classes. Une classe fortement couplée a besoin de connaître un grand nombre d'autres classes. Les classes à couplage fort ne sont pas souhaitables car :

- elles sont plus difficiles à comprendre ;
- elles sont plus difficiles à réutiliser, car leur emploi demande la présence de nombreuses autres classes ;
- elles sont plus sensibles aux variations, en cas de modification d'une des classes auxquelles celle-ci est liée.

Le principe de faible couplage consiste à minimiser les dépendances afin d'obtenir un faible couplage entre les classes.

Principe de forte cohésion

La cohésion est une mesure des liens entre les tâches effectuées par une classe. Une classe a une faible cohésion si elle effectue des tâches qui ont peu de liens entre elles, et dont certaines auraient dû être affectées à d'autres classes. Il est préférable d'avoir des classes fortement cohésives car elles sont plus faciles à comprendre, à maintenir, à réutiliser. Elles sont moins affectées par une modification.

Principe du contrôleur

Le principe du contrôleur consiste à affecter la responsabilité du traitement des événements systèmes (c'est-à-dire des événements générés par un acteur externe) dans une ou plusieurs « classes de contrôle » (des « contrôleurs »). Par exemple, si on a une interface graphique, les événements reçus par l'interface sont délégués à un contrôleur. Cela permet de séparer la

logique de l'application de l'interface : la logique de l'application ne doit pas être gérée par la couche interface.

c) Utilisation des diagrammes UML

On peut utiliser différents diagrammes UML pour effectuer la conception.

Diagramme des classes logicielles

Le diagrammes des classes logicielles est inspiré du diagramme de classes d'analyse. Des classes peuvent être ajoutées, des liens ajoutés ou modifiés. On définit pour chaque classe les opérations qu'elle contient. On peut également donner des diagrammes d'objets montrant des configurations typiques pouvant être construites.

Diagrammes de séquence ou de collaboration

Les diagrammes de séquence ou de collaboration illustrent la façon dont les objets collaborent pour réaliser une fonctionnalité. Ces diagrammes permettent d'illustrer et de motiver les choix d'affectation de responsabilités aux objets.

Diagrammes d'états-transitions

Les diagrammes d'états-transitions permettent de spécifier le comportement des objets d'une classe. Ces diagrammes peuvent être considérés comme généralisant les diagrammes de séquence et de collaboration car ils doivent décrire tous les comportements, et non seulement quelques comportements possibles.

Diagrammes d'activités

On peut enfin utiliser des diagrammes d'activités pour décrire le comportement des opérations d'une classe.

d) Cohérence entre les différents diagrammes

Il est important de s'assurer de la cohérence entre les différents diagrammes élaborés au niveau de la conception. En effet, la mise en oeuvre est *a priori* très proche de ces modèles. Cela suppose d'effectuer certaines vérifications.

Les vérifications contextuelles consistent à vérifier que les différents éléments de modélisation (objets, attributs, événements, opérations, associations, rôles... etc.) sont correctement déclarés et utilisés.

Exemple 1

Si on dispose de diagrammes d'objets, il faut vérifier que chaque diagramme d'objets est cohérent avec le diagramme de classes associé :

- correction des attributs de chaque objet ;
- correction des liens entre les objets ;
- respect des multiplicités.

Exemple 2

On considère une classe A, à laquelle est associé un automate comportant une transition étiquetée par $\ll m_1 \ / \ o.m_2 \gg$ (cf. figure III. 5).

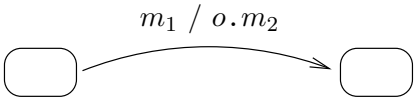

FIGURE III. 5 – Une transition de l'automate

Il faut vérifier que :

- m_1 est une méthode de la classe A ;
- o est un objet, d'une certaine classe B, accessible depuis un objet de la classe A (attribut de A de type B, ou lien entre les deux classes A et B, ayant pour nom de rôle o) ;
- m_2 est une méthode de la classe B.

Exemple 3

Dans un diagramme de séquence, on peut mettre en correspondance les séquences d'événements reçues par un objet avec une suite de transitions de l'automate associé à sa classe.

Chapitre IV

Patrons de conception

Concevoir un logiciel est une activité difficile, écrire un programme facile à maintenir, extensible et réutilisable est encore plus difficile. Le développement orienté objet peut faciliter l'extension et la réutilisation, à condition que les développeurs soient expérimentés.

L'objectif des patrons de conception (≪ design patterns ≫ en anglais) est de recueillir l'expérience et l'expertise des programmeurs, afin de la transmettre à d'autres programmeurs qui pourront la réutiliser.

Les patrons de conception ont été introduits en 1977 par Christopher Alexander, architecte en bâtiment. Alexander a proposé un catalogue de problèmes classiques en construction (bâtiments, villes), avec des solutions classiques. Les patrons de conception orientés objet reprennent cette idée de fournir un catalogue de solutions classiques à des problèmes de conception objet. Ils ont été introduits par Beck et Cunningham en 1987, et la thèse de Erich Gamma, soutenue en 1991 porte sur ce sujet.

Référence : *Design Patterns. Elements of Reusable Object Oriented Software*. Gamma, Helm, Johnson et Vlissides. Addison Wesley, 1995.

1. Notion de patron

Un patron de conception est la description d'un problème récurrent, dans un certain contexte, accompagné d'une description des différents éléments d'une solution à ce problème. Il s'agit de décrire une solution suffisamment générale et flexible à un problème qu'on rencontre souvent.

Un patron de conception comporte différents éléments :
- le nom, qui doit permettre de reconnaître le patron et indiquer son utilisation ;
- le problème, qui doit décrire l'objectif du patron ;
- le contexte, qui décrit les circonstances d'utilisation du patron ;
- la solution, qui décrit le schéma de conception résolvant le problème ;
- les conséquences, qui décrivent les avantages et inconvénients de la solution proposée.

On a différentes sortes de patrons de conception, qui peuvent être utilisés dans différentes

étapes du cycle de vie.

a) Analyse et définition des besoins

Lors de cette étape, le principal problème est la communication entre les utilisateurs et les informaticiens d'une part et l'évaluation de la complexité d'autre part. Les patrons d'analyse servent à aider à résoudre ce genre de problèmes.

b) Analyse et conception

Lors de cette étape, les problèmes sont la définition d'une architecture adéquate, la résolution de sous-problèmes et la communication entre les développeurs. Les patrons architecturaux et les patrons de conception permettent d'aider à résoudre ce genre de problèmes.

Les patrons de conception peuvent être classifiés en différentes sortes :

1. les patrons « de création », qui concernent la création d'objets ;
2. les patrons « structurels », qui concernent la structure des objets et les relations entre ces objets ;
3. les patrons « comportementaux », qui sont relatifs au comportement des objets.

c) Mise en oeuvre

Lors de cette étape, l'objectif est de produire un code correct et facile à maintenir. On utilise des patrons de programmation (ou *idiomes*), qui sont des solutions spécifiques à un langage de programmation.

Exemples d'idiomes :
- implémentation, en C, de tableaux de taille variable, réalloués dynamiquement si la taille dépasse les bornes ;
- implémentation de l'héritage multiple en Java.

2. Étude de quelques patrons de conception

Dans ce paragraphe, nous étudions en détail quelques patrons de conception.

a) Singleton

Le Singleton est un patron de conception de création.

But

L'objectif de ce patron est d'assurer qu'une classe a une instance unique, et d'en donner un accès « global ».

Motivation

Ce patron peut servir dans beaucoup de circonstances, en particulier lorsque la classe correspond à un objet unique dans le monde réel.

Exemples

— système de fichiers ;
— gestionnaire de fenêtres ;
— système de comptabilité pour une entreprise ;
— ... etc.

Code Java

```
class Singleton {

   /* Attribut privé contenant l'instance unique de Singleton. */
   final private static Singleton instanceUnique =
      new Singleton() ;

   /* Méthode qui renvoie l'instance unique de Singleton. */
   static Singleton instance() {
      return instanceUnique ;
   }

   /* On déclare le constructeur privé afin d'interdire
      l'instanciation de cette classe depuis une autre classe. */
   private Singleton() { }
}
```

b) Méthode Fabrique

La Méthode Fabrique est un patron de création.

But

L'objectif de ce patron est de permettre la création d'objets sans que l'on sache la classe exacte de ces objets.

Principe

On cherche à créer des objets d'une sous-classe de la classe `Produit`. On écrit une interface `Fabrique` contient une méthode `créer` qui renvoie un objet de type `Produit`. La méthode `créer` est la méthode fabrique.

On crée ensuite des instance de `ProduitConcret`, sous classe de `Produit` en implémentant l'interface `Fabrique` avec une classe `FabriqueConcrète` (cf. figure IV. 1).

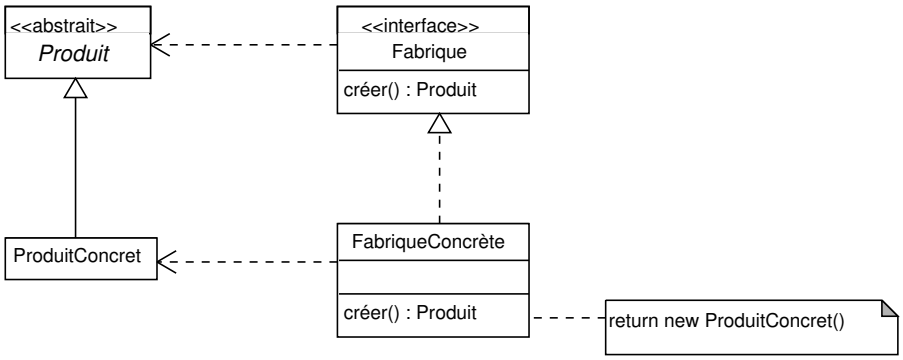

FIGURE IV. 1 – Diagramme de classes du patron Méthode Fabrique

Utilisation

Ce patron est souvent utilisé dans les « *framework* »(ensemble de classes réutilisables) lorsque du code réutilisable a besoin de créer des objets de sous-classes définies par une application qui utilise ce framework.

c) Fabrique Abstraite

La Fabrique Abstraite est un patron de création.

But

L'objectif de ce patron est de créer une famille d'objets qui dépendent les uns des autres, sans que l'utilisateur de cette famille d'objets ne connaisse la classe exacte de chaque objet.

Principe

L'utilisateur a accès uniquement à différentes classes abstraites (une par produit), par exemple `ProduitAbstraitX` et `ProduitAbstraitY` et à une classe qui permet de créer des instances de ces produits : `Fabrique1` ou `Fabrique2` (cf. figure IV. 2). Ces instances sont créées à l'aide des méthodes `créerX` et `créerY`.

Si l'utilisateur utilise la classe `Fabrique1` pour créer les objets, alors il obtient des instances de `ProduitX1` et `ProduitY1`. S'il utilise la classe `Fabrique2`, alors il obtient des instances de `ProduitX2` et `ProduitY2`.

Avantages de ce patron

Les avantages de ce patron sont les suivants :

— Le patron facilite l'utilisation cohérente des différents produits : si le client utilise toujours la même classe, par exemple `Fabrique1`, pour créer des objets, il est sûr de toujours utiliser des produits de la même famille.

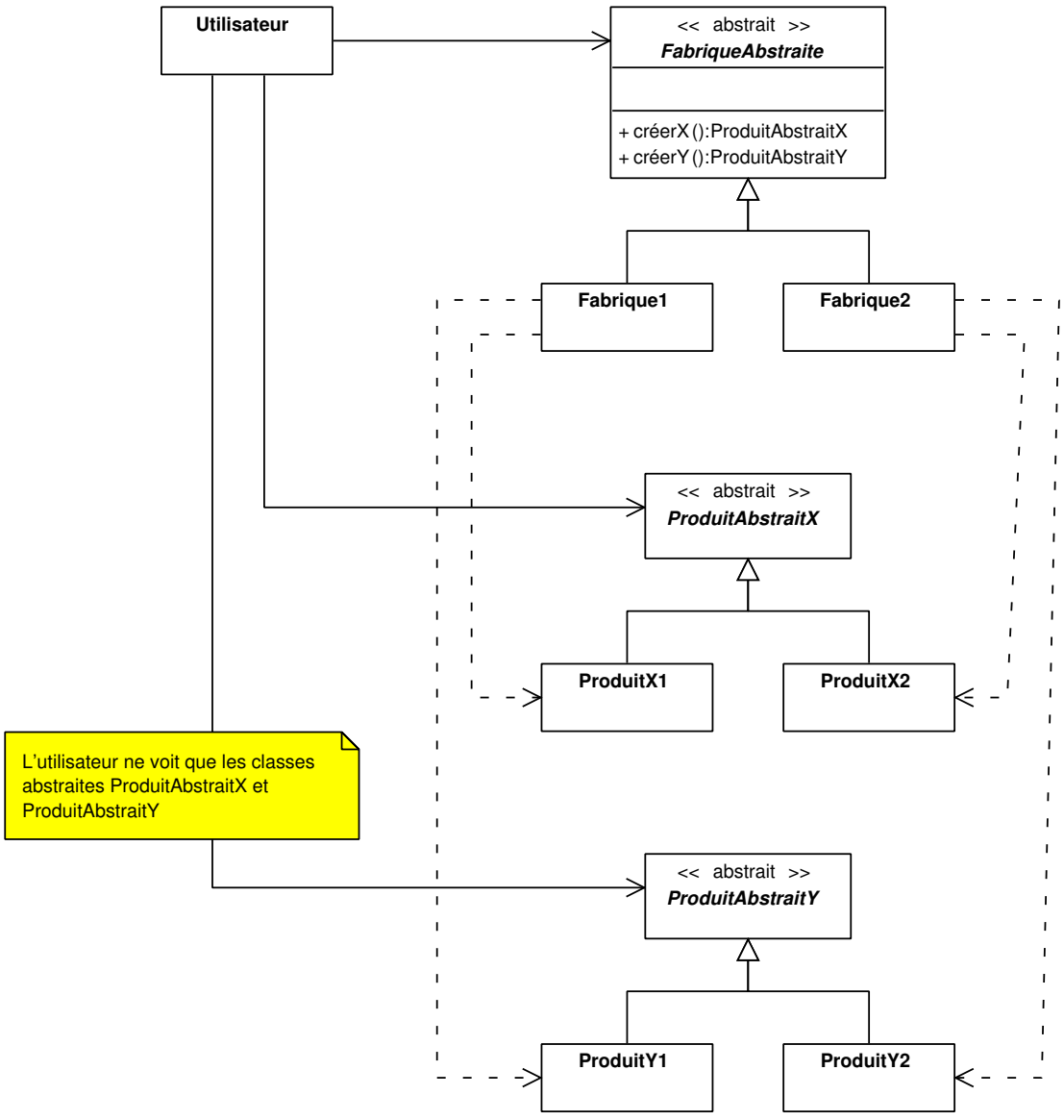

FIGURE IV. 2 – Diagramme de classes du patron Fabrique Abstraite

— L'utilisateur peut facilement changer de famille de produits, puisqu'il suffit de changer l'instantiation de la fabrique.
— On peut assez facilement ajouter une nouvelle famille de produits, en ajoutant une nouvelle sous-classe pour chaque produit abstrait.

Inconvénient de ce patron

Il peut être difficile d'ajouter de nouveaux produits puisqu'il faut modifier toutes les classes qui dérivent de `FabriqueAbstraite`.

Remarque

On peut implémenter les sous-classes de `FabriqueAbstraite` en utilisant le patron Singleton.

d) Objet composite

Objet composite est un patron de conception structurel.

But

L'objectif de ce patron est de créer des objets simples ou composés, avec des méthodes de traitement uniformes, pour lesquelles le client n'a pas à savoir s'il applique un certain traitement à un objet simple ou composé.

Solution

On utilise une classe abstraite `Composite` contenant une (ou plusieurs) méthodes abstraites de traitement (cf. figure IV. 3).

La figure IV. 4 montre un exemple d'application du patron objet composite pour modéliser des figures géométriques.

e) Adaptateur

Le patron Adaptateur est un patron structurel. On utilise un adaptateur lorsque, pour implémenter une interface « cible », on souhaite réutiliser une classe « source » qui ne respecte pas exactement cette interface. La solution qui consiste à modifier la classe source n'est pas satisfaisante lorsque cette classe est réutilisée à d'autres endroits.

Solution par héritage

Une solution consiste à réaliser une sous-classe `Adaptateur` de `Source` qui implémente l'interface `Cible` (cf. figure IV. 5).

Cette solution a les inconvénients suivants :
— on ne peut pas adapter des sous-classes de `Source` ;
— on peut redéfinir des méthodes de `Source`.

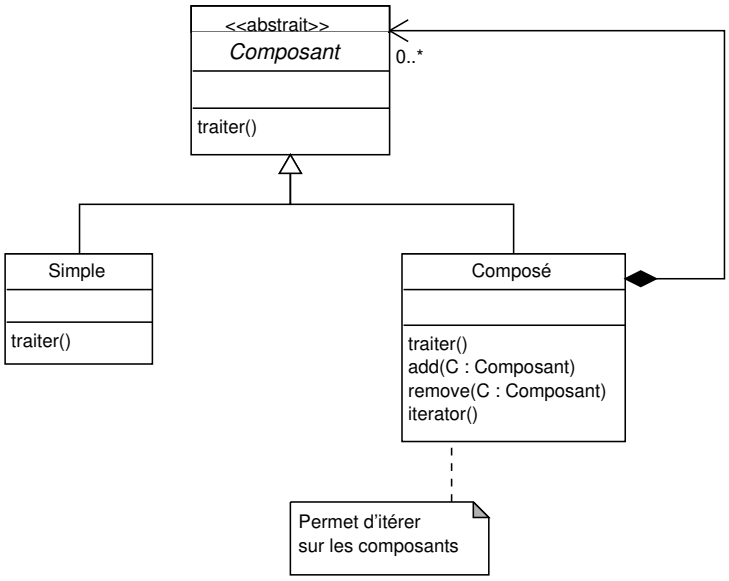

FIGURE IV. 3 – Diagramme de classes du patron Composite

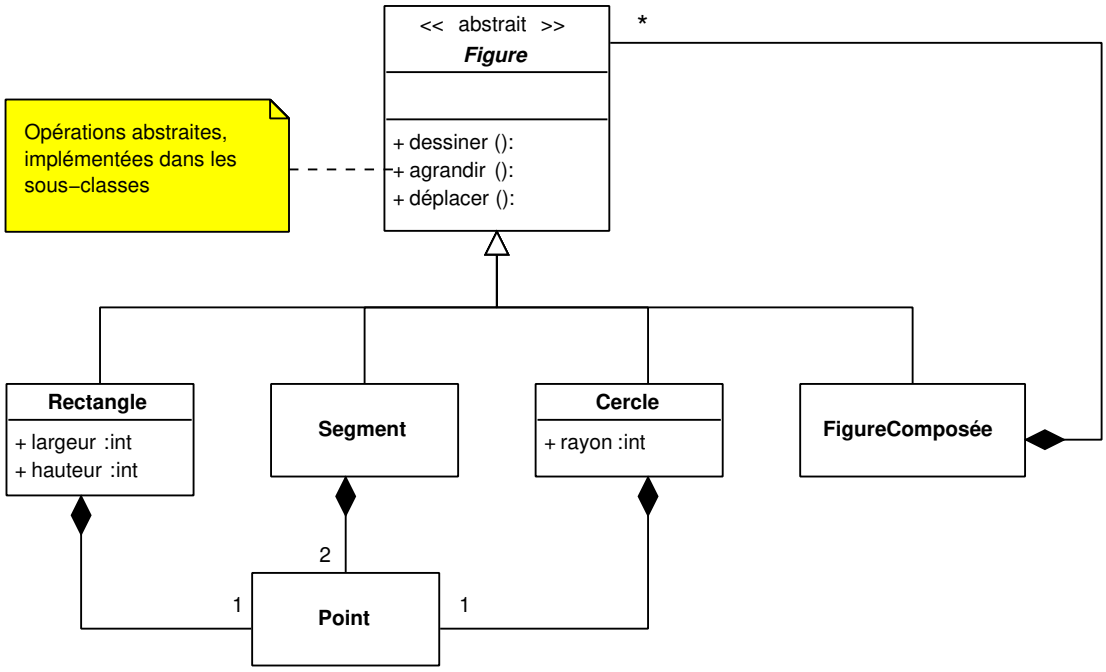

FIGURE IV. 4 – Figures géométriques

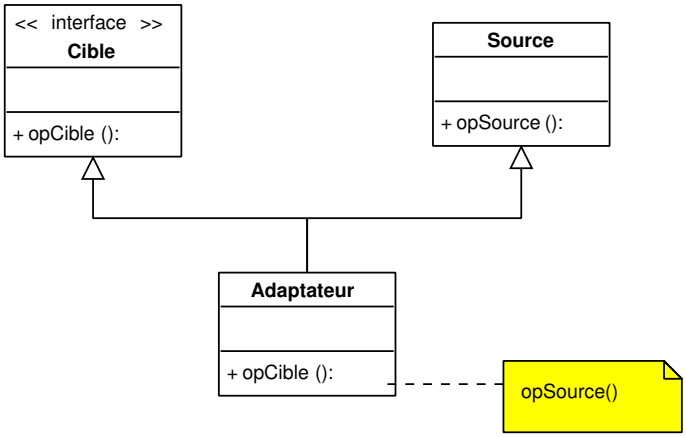

FIGURE IV. 5 – Patron Adaptateur, par héritage

Solution par délégation

Le principe de la délégation consiste à créer un lien entre l'adaptateur et la source et à ce que l'adaptateur délègue le travail à effectuer à la classe source. La source joue le rôle du délégué.

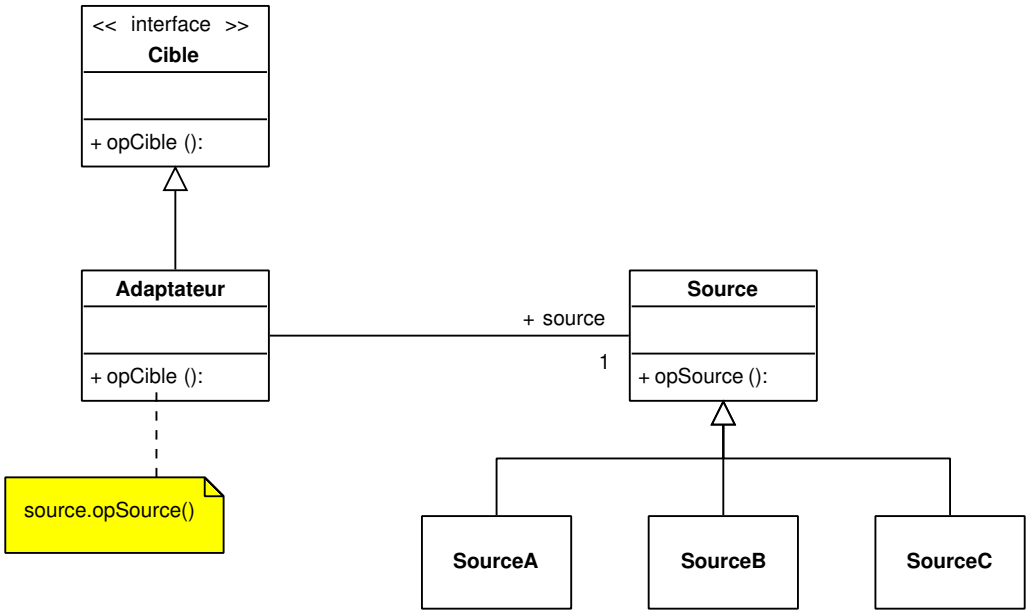

FIGURE IV. 6 – Patron Adaptateur, par délégation

Remarque : cette solution ressemble à l'implémentation de l'héritage multiple.

Cette solution a les avantages suivants :
- on peut adapter des sous-classes de `Source` : `SourceA`, `SourceB`, `SourceC` ;
- cette solution ne permet pas la redéfinition des méthodes de `Source`.

f) Patrons Stratégie, Commande et État

Les patrons de conception Stratégie, Commande et État sont des patrons comportementaux assez similaires qui consistent à :

— associer à certains traitements (ou méthodes) des objets ;
— définir une hiérarchie de classes pour effectuer ces traitements de façon uniforme ;
— découpler les données et les traitements : la hiérarchie de classes des traitements est indépendante de celle des objets qui vont utiliser ces opérations.

Patron Stratégie

L'objectif de ce patron est de définir une famille d'algorithmes encapsulés dans des objets, afin que ces algorithmes soient interchangeables dynamiquement.

Motivation

Pour résoudre un problème, il existe souvent plusieurs algorithmes ; dans certains cas il peut être utile de choisir à l'exécution quel algorithme utiliser, par exemple selon des critères de temps de calcul ou de place mémoire.

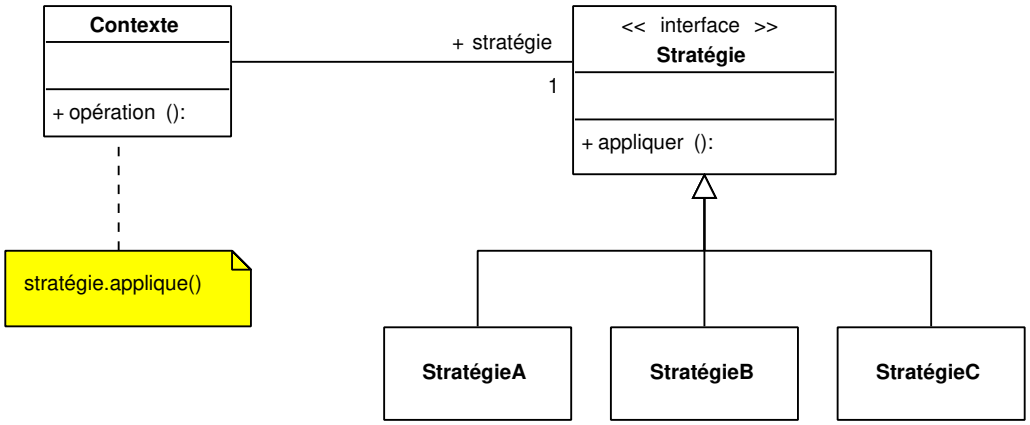

FIGURE IV. 7 – Patron Stratégie

Les avantages de ce patron sont les suivants :

— en découplant les algorithmes des données, la classe `Contexte` peut avoir des sous-classes indépendantes des stratégies ;
— on peut changer de stratégie dynamiquement.

Ce patron a les inconvénients suivants :

— on a un surcoût en place mémoire, car il faut créer des objets `Stratégie` ;
— on a un surcoût en temps d'exécution à cause de l'indirection `stratégie.applique()`.

Patron Commande

L'objectif de ce patron est d'encapsuler des « commandes » dans des objets.

Motivation

Dans des applications qui comportent une interface graphique, on peut associer des commandes à des boutons, ou à des lignes dans des menus ; revenir en arrière d'une ou plusieurs commandes (≪ annuler ≫) ou repartir en avant (≪ rétablir ≫). Pour cela, on doit associer à des actions des objets que l'on peut stocker, passer en paramètre... etc.

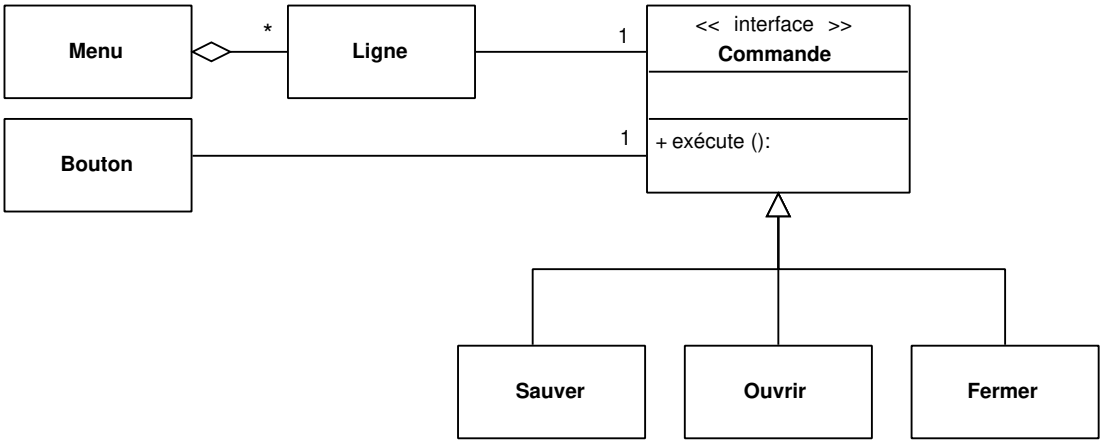

FIGURE IV. 8 – Patron Commande

Les avantages de ce patron sont les suivants :

- on peut modifier une association `Ligne` — `Commande` à l'exécution, afin de paramétrer le logiciel à l'exécution ;
- on peut effectuer la même action par différents moyens (en passant par un menu, ou en appuyant sur un bouton... etc.) ;
- on peut définir des ≪ macro commandes ≫, composées de séquences de commandes (cf. figure IV. 9) ;
- on peut revenir en arrière (annuler, rétablir). Cela nécessite de stocker l'historique des commandes, et un état interne associé à chaque commande effectuée (cf. figure IV. 10).

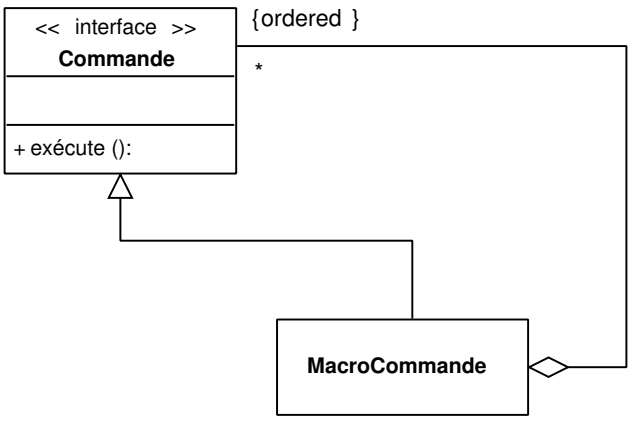

FIGURE IV. 9 – Macro commandes

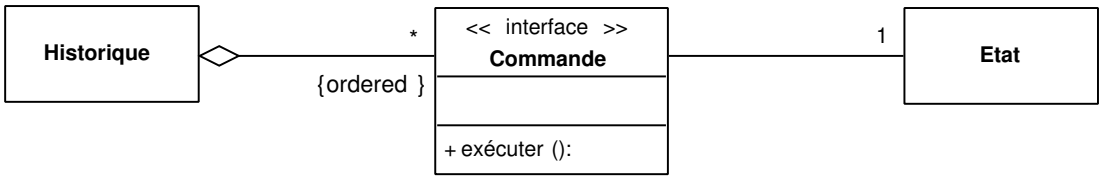

FIGURE IV. 10 – Historique de commandes

Patron État

Le patron État permet de réaliser des objets dont le comportement change lorsque leur état interne est modifié.

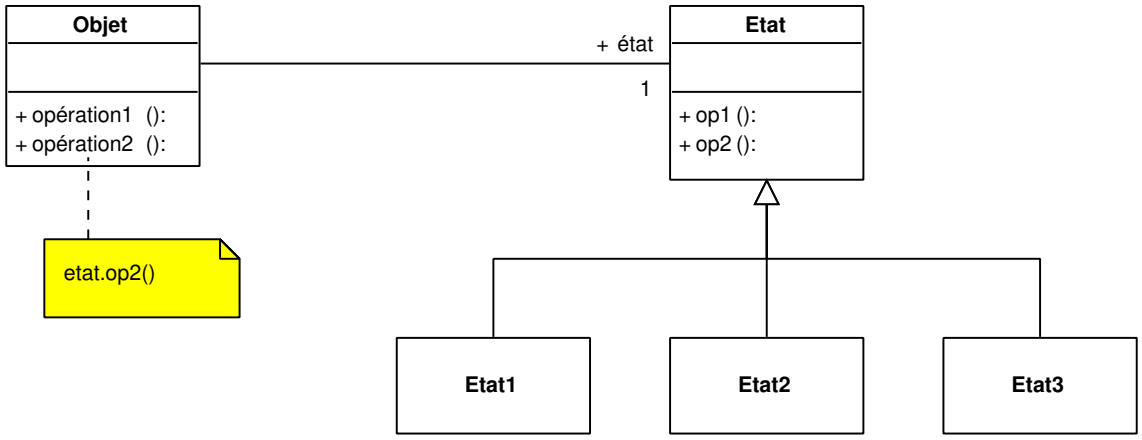

FIGURE IV. 11 – Patron État

Différences entre les patrons Stratégie, Commande et État

Les trois patrons ont une structure assez similaire, les différences entre ces patrons sont liées à ce que l'on cherche à implémenter.

- Pour le patron Stratégie, le but est d'implémenter une même opération de plusieurs façons différentes. On peut considérer qu'on a une seule spécification de l'opération.
- Pour le patron Commande, on peut vouloir associer une même commande à différents objets, et définir des séquences de commandes (macros) ou des historiques de commandes.
- Pour le patron État, plusieurs opérations peuvent être associées à un état.

g) Patron Observateur

L'Observateur est un patron comportemental, dont le but est de définir une dépendance entre un objet (appelé sujet) et un ensemble d'objets (appelés observateurs) de sorte que lorsque le sujet change d'état, tous les observateurs qui en dépendent soient informés et mis à jour automatiquement.

Par exemple, une application peut comporter plusieurs représentations graphiques pour un même objet (cf. figure IV. 12).

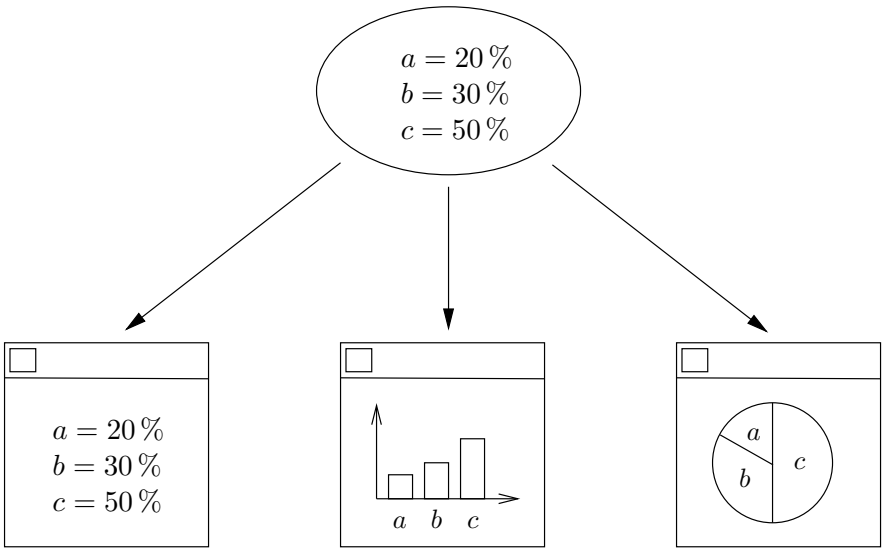

FIGURE IV. 12 – Plusieurs représentation graphiques pour un même objet

Le sujet est en relation avec un ensemble d'observateurs. Lorsque le sujet change d'état, tous les observateurs sont informés.

On définit une classe abstraite `Sujet`, dont héritent toutes les classes pouvant servir de sujet, autrement dit, pouvant être mises en relation avec un ensemble d'observateurs.

On définit une interface `Observateur` qui contient une méthode `miseAJour`.

Un sujet est en relation avec un ensemble d'observateurs. On dispose de méthodes pour attacher ou détacher un observateur à un sujet, autrement dit, ajouter ou enlever un observateur de l'ensemble des observateurs de ce sujet.

La méthode informe appelle `miseAJour` sur chaque observateur.

L'interface `Observateur` est implémentée par un certain nombre d'observateurs concrets, qui contiennent le code correspondant à l'affichage et la mise à jour de cet affichage.

Remarques sur la navigabilité des relations :
- `Sujet` doit connaître `Observateur` pour l'informer (mais `Observateur` n'a pas besoin de pouvoir accéder à `Sujet`) ;
- `ObservateurConcret` doit connaître `SujetConcret` pour effectuer la mise à jour de l'affichage : il doit en effet connaître l'état du sujet (mais `SujetConcret` n'a pas besoin de connaître `ObservateurConcret` car `Sujet` connaît `Observateur`).

Rapport avec l'architecture Modèle – Vue – Contrôleur : le patron Observateur est un moyen d'implémenter (en partie) une architecture MVC. Les classes qui héritent de `Sujet` sont des classes du modèle, et les classes qui implémentent `Observateur` sont des classes de la vue.

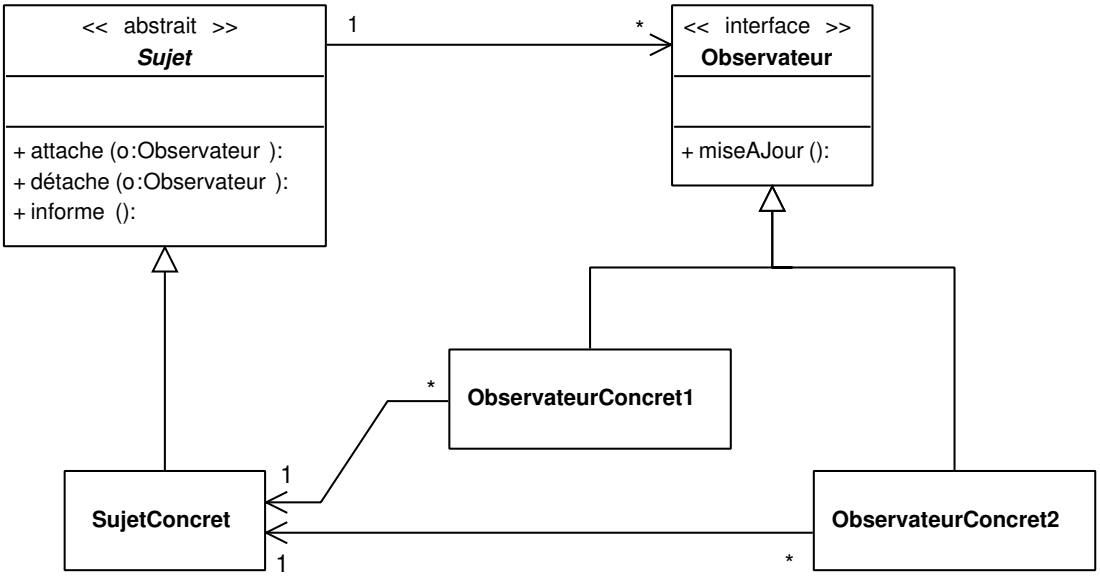

FIGURE IV. 13 – Patron Observateur

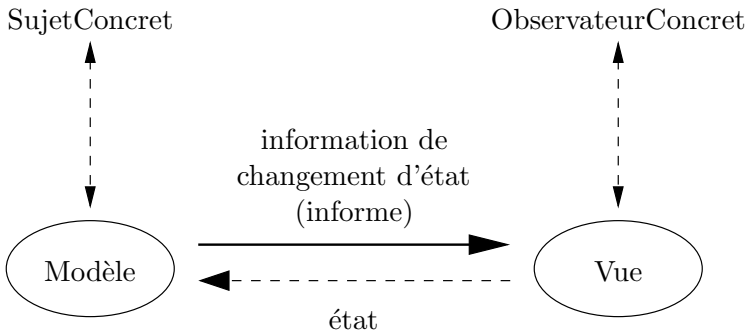

FIGURE IV. 14 – Implémentation d'une architecture MVC à l'aide du patron Observateur

h) Interprète

L'Interprète est un patron de conception comportemental. Il a pour but, étant donné un langage, de définir une représentation pour sa grammaire abstraite (autrement dit, une structure d'arbre abstrait), ainsi qu'un interprète utilisant cette représentation.

On suppose qu'on a une grammaire abstraite, avec des règles de la forme suivante :

$$\begin{array}{lcl} A & \to & \mathrm{Noeud}_1(A_{1,1}, \ldots, A_{1,n_1}) \\ & & \mid \ \mathrm{Noeud}_2(A_{2,1}, \ldots, A_{2,n_2}) \\ & & \mid \ \ldots \\ & & \mid \ \mathrm{Noeud}_k(A_{k,1}, \ldots, A_{k,n_k}) \end{array}$$

On associe à chaque non terminal une classe abstraite, et à chaque noeud une classe concrète qui hérite de cette classe abstraite.

Pour chaque classe abstraite A, on écrit une méthode abstraite `interp(c:Contexte)`.

La classe `Contexte` correspond à des paramètres nécessaires à l'interprétation, qui peut varier d'une classe abstraite à l'autre.

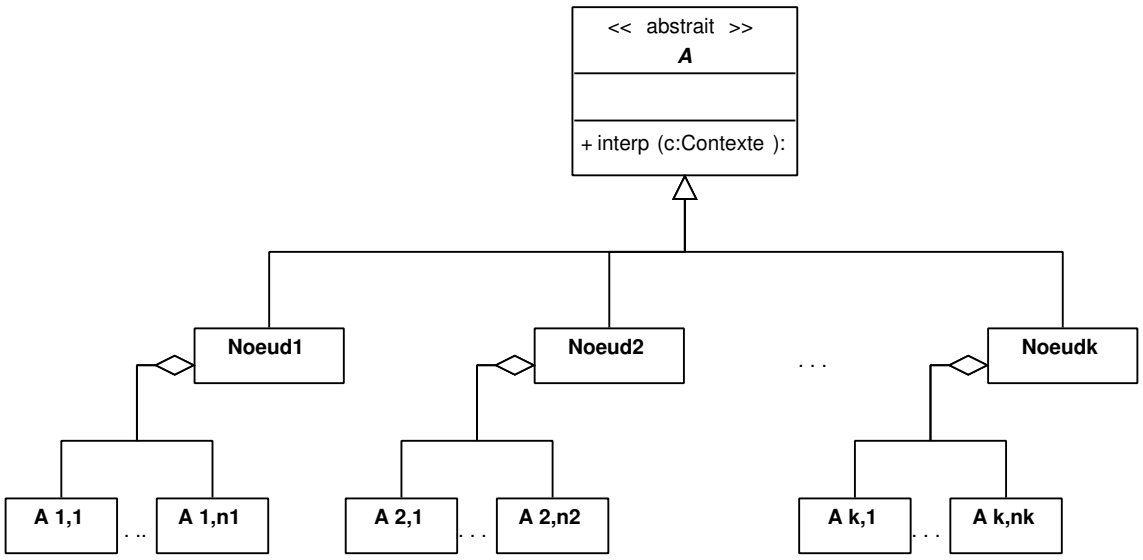

FIGURE IV. 15 – Patron Interprète

On peut appliquer ce patron de conception chaque fois qu'on doit définir des opérations ou des calculs dirigés par la syntaxe, en particulier en compilation :

- évaluation (pour écrire un interprète, au sens propre) ;
- vérification de type ;
- génération de code ;
- ... etc.

On peut par exemple appliquer le patron Interprète pour écrire un compilateur pour le langage Deca (cf. Projet Génie Logiciel).

Le patron Interprète a les avantages suivants :
- on peut facilement modifier et étendre la grammaire. Par exemple, on peut facilement ajouter de nouvelles expressions en définissant de nouvelles classes.
- l'implémentation de la grammaire est simple, et peut être réalisée automatiquement à l'aide d'outils de génération.

Le patron Interprète a certains inconvénients :
- lorsque la grammaire est complexe, on a une multiplication des classes.
- il devient alors délicat d'ajouter de nouvelles opérations, car celles-ci doivent être ajoutées dans toutes les classes de la hiérarchie.

i) Visiteur

Le Visiteur est un patron de conception comportemental.

L'objectif du patron Visiteur est de représenter une opération définie en fonction de la structure d'un objet. Le Visiteur permet alors de définir de nouvelles opérations sans modifier les classes qui définissent la structure des objets auxquelles elles s'appliquent.

On cherche à ne pas répartir les différents cas dans les classes qui définissent la structure des objets, mais au contraire à regrouper tout ce qui concerne une opération dans une seule classe.

Ne pas modifier les classes définissant la structure des objets présente les intérêts suivants :
- cela favorise la réutilisation (on récupère facilement, par exemple, une structure d'arbre abstrait pour un langage) ;
- cela permet le développement parallèle de code par plusieurs équipes. Par exemple, une équipe peut implémenter la vérification de types et une autre la génération de code sans travailler sur les mêmes classes ;
- cela permet de partager la structure des objets par plusieurs applications, qui peuvent alors coopérer plus facilement.

Principe

On considère une hiérarchie de classes définissant la structure d'objets. On suppose qu'on a une classe abstraite `Elément`, racine de cette hiérarchie (cf. figure IV. 16).

On définit une interface `Visiteur`, qui contient une méthode **`void`** `visite(ElémentX e)` pour chaque sous-classe concrète `ElémentX` de `Elément`.

```
/**
 * Interface Visiteur, qui permet de définir des opérations qui
 * s'appliquent sur des éléments.
 */
interface Visiteur {
   void visite(ElémentA e) ;
```

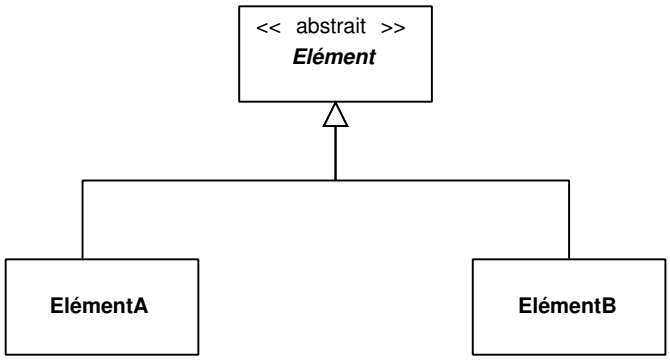

FIGURE IV. 16 – Hiérarchie Elément pour le patron Visiteur

```
    void visite(ElémentB e) ;
}
```

Dans chaque classe de la hiérarchie de classes définissant la structure des objets, on définit une méthode

```
    void applique(Visiteur v) ;
```

Cette méthode est abstraite dans les classes abstraites. Dans les classes concrètes, elle appelle la méthode `visite` sur le visiteur `v`, et passe **this** en paramètre.

```
/**
 * Hiérarchie d'éléments.
 */

abstract class Elément {
    abstract void applique(Visiteur v) ;
}

class ElémentA extends Elément {
    void applique(Visiteur v) {
        v.visite(this) ; // Appel de la méthode visite(ElémentA)
    }
}

class ElémentB extends Elément {
    void applique(Visiteur v) {
        v.visite(this) ; // Appel de la méthode visite(ElémentB)
    }
}
```

Remarque

La définition de la méthode `applique` (c'est-à-dire `v.visite(this)`) est différente dans chaque classe concrète car la méthode `visite` appelée a une signature différente pour chaque classe : `visite(ElémentA)` ou `visite(ElémentB)`. On ne peut donc pas définir la méthode `applique` une seule fois au niveau de `Elément`.

On appelle « visiteur » un objet d'une classe qui implémente l'interface `Visiteur`. Un visiteur définit une opération s'appliquant sur tout objet de type `Elément`.
La méthode `applique(Visiteur v)` représente l'application de l'opération définie par le visiteur à l'objet.

On peut alors définir une opération qui s'applique sur `Elément` à l'aide d'une classe qui implémente `Visiteur`.

```
/**
 * Opération qui s'applique à un élément.
 */
class Opération implements Visiteur {

   public void visite(ElémentA e) {
      // Ce que l'opération fait sur un objet de type ElémentA
      . . .
   }

   public void visite(ElémentB e) {
      // Ce que l'opération fait sur un objet de type ElémentB
      . . .
   }

   // Les attributs de cette classe peuvent servir de paramètres
   // d'entrée ou de résultat pour l'opération.
}
```

Un appel de l'opération se fait de la façon suivante :

```
Elément e = new ElémentA() ; // Un élément
Visiteur v = new Opération() ; // Une opération
e.applique(v) ; // L'opération v est appliquée sur l'élément e
```

L'appel « `e.applique(v)` » appelle « `v.visite(e)` », qui effectue le code correspondant à `ElémentA`, car `visite` a pour signature `visite(ElémentA e)`.

Si on souhaite programmer une opération s'appliquant sur un objet de type `Elément` sans utiliser le patron Visiteur, on a deux possibilités :

1. Répartir tout le code de l'opération dans les différentes classes de la hiérarchie. Cela correspond à une application du patron Interprète.
2. Écrire une classe `Opération` qui teste les différents cas à l'aide de **instanceof** (il s'agit

de programmation « classique », « non objet »).

```
class Opération {
   static void op(Elément e) {
      if (e instanceof ElémentA) {
        ElémentA aA = (ElémentA) e ;
        // Ce que l'opération doit faire sur ElémentA
         ...
      } else if (e instanceof ElémentB) {
         ElémentB eB = (ElémentB) e ;
         // Ce que l'opération doit faire sur ElémentB
         ...
      }
   }
}
```

Cette opération s'utilise de la façon suivante :

```
Elément e = new ElémentA() ; // Un élément
Opération.op(e) ; // Appel de l'opération op sur l'élément e
```

Le patron de conception Visiteur a les avantages suivants sur la programmation « classique » :
- le patron oblige à traiter tous les cas, et c'est vérifié à la compilation ;
- il s'agit d'une programmation « purement objet » ;
- le patron évite l'utilisation de **instanceof**, de devoir déclarer une variable initialisée à l'aide d'une conversion, et évite également ainsi le risque d'erreur de conversion à l'exécution ;
- le patron évite d'effectuer n tests pour trouver le code à exécuter (en particulier lorsque `Elément` a de nombreuses sous-classes) ;
- le patron permet l'ajout de nouvelles opérations sans modifier la hiérarchie de classes, et l'ajout de nouvelles classes (en modifiant l'interface).

Le patron présente les inconvénients suivants :
- ce patron est lourd à mettre en œuvre : il faut prévoir une méthode `applique` par classe.
- le traitement de méthodes comportant des paramètres et un résultat est également lourd ;
- on a un surcoût à chaque indirection `applique` → `visite` ;
- le code est peu lisible lorsqu'on ne connaît pas le patron.

Variante générique du patron Visiteur

Il existe une variante du patron qui consiste à utiliser une interface Visiteur générique, paramétrée par le type de retour `T` de la méthode visite.

```
interface Visiteur<T> {
   T visite(ElementA e);
   T visite(ElementB e);
}
```

On utilise ensuite des méthodes `applique` également génériques, paramétrées par le type de retour `T`.

```
abstract class Element {
   abstract <T> T applique(Visiteur<T> v);
}

class ElementA extends Element {
   <T> T applique(Visiteur<T> v) {
       return v.visite(this);
   }
}
```

L'avantage de cette variante est qu'on peut avoir des types de retour différents suivant les visiteurs utilisés. Cela permet en particulier d'éviter d'utiliser un attribut supplémentaire pour coder la valeur de retour.

j) Simulation de l'héritage multiple

Il y a *héritage multiple* lorsqu'une classe hérite de plusieurs classes. Les langages Java et Ada95 ne permettent pas de faire de l'héritage multiple, alors que C++ le permet.

Difficultés liées à l'héritage multiple

L'héritage multiple présente certaines difficultés, ce qui explique son interdiction dans certains langages :

- lorsqu'un attribut est hérité par deux chemins différents (cf. figure IV. 17), doit-on avoir une seule valeur d'attribut ou cette valeur doit-elle être dupliquée ?
- lorsque deux méthodes de même nom sont héritées, quelle méthode doit-on choisir ?

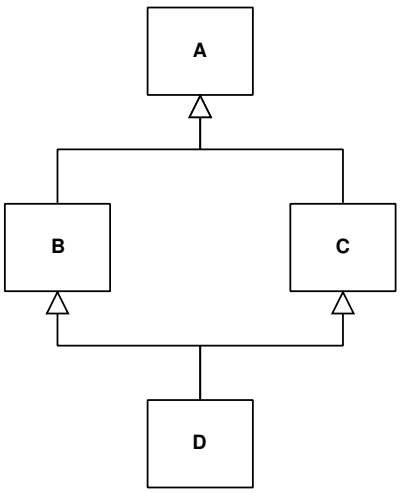

FIGURE IV. 17 – Classe A héritée dans D par deux chemins

- les programmeurs ont des difficultés à utiliser l'héritage multiple à bon escient ;

— l'heritage multiple est plus compliqué à compiler, et il implique également un surcoût des programmes à l'exécution.

Méthode héritées par deux chemins différents

Si les classes B et C comportent deux méthodes qui ont la même signature, il y a une ambiguïté sur cette méthode au niveau de la classe D. La résolution de l'ambiguïté peut être réalisée de plusieurs façons différentes :

— on peut effectuer un choix « arbitraire » de l'une des deux méthodes (par exemple, choisir la première ayant été compilée) ;
— on peut obliger le renommage de l'une des deux méthodes dans la classe D (c'est par exemple le choix effectué dans le langage Eiffel) ;
— lors d'un appel de cette méthode, on peut obliger l'utilisateur à préciser la méthode appelée (c'est le choix effectué dans le langage C++).

Le choix effectué dans le langage Eiffel a les avantages suivants :

— il oblige le programmeur à prendre conscience des ambiguïtés au moment de la création de la classe (donc assez tôt) ;
— il permet d'écrire du code plus simple, puisque la désambiguïsation est réalisée une seule fois.

Simulation de l'héritage multiple en Java

Utilisation de la délégation

Pour simuler l'héritage multiple en Java, on peut remplacer l'un des deux héritages par une délégation (cf. figure IV. 18).

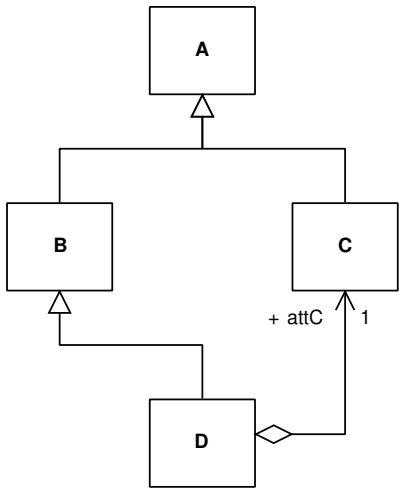

Figure IV. 18 – Héritage multiple, simulé par délégation

```
class D extends B {

   // Attribut permettant la délégation.
   C attC ;
```

```
    // Pour chaque méthode m de C qui doit être héritée dans D,
    // on réécrit un appel (utilisation de la délégation).
    void m(...) {
        attC.m(...) ;
    }
}
```

Supposons que les classes B et C comportent deux méthodes de même nom. On a trois possibilités :

- si on ne réécrit pas la méthode de C dans la classe D, c'est la méthode de B qui sera choisie ;
- si on réécrit la méthode de C dans D, c'est la méthode de C qui sera choisie ;
- si on redéfinit la méthode dans D, les méthodes de B et de C seront cachées par cette redéfinition.

Avec ce codage, on peut appeler les méthodes de B et les méthodes de C sur un objet de type D. On a donc simulé l'*héritage*.

Exemple

Soient `methA`, `methB`, `methC` et `methD` des méthodes définies respectivement dans les classes A, B, C et D.

On peut écrire le code Java suivant :

```
D d = new D(...) ;
d.methA(...) ;  // Héritage "normal" de A
d.methB(...) ;  // Héritage "normal" de B
d.methC(...) ;  // Héritage d'une méthode de C
                // (simulé par délégation)
d.methD(...) ;  // Appel d'une méthode de D
```

Néanmoins, un objet de type C n'est pas un objet de type D. On n'a donc pas la propriété de *substitution* (ou *sous-typage*).

En particulier, on ne peut pas écrire :

```
C d2 = new D(...) ;  // Erreur de type à la compilation
```

A fortiori, on n'a pas de liaison dynamique, et on ne peut pas effectuer d'appel de la forme `d2.methD(...)`. Pour cette raison, on introduit une interface, qui permet de simuler en partie le sous-typage.

Utilisation d'interfaces

On ajoute une interface `CInt` qui contient (le profil de) toutes les méthodes de C, y compris les méthodes héritées de A. Les classes C et D implémentent l'interface `CInt` (cf. figure IV. 19).

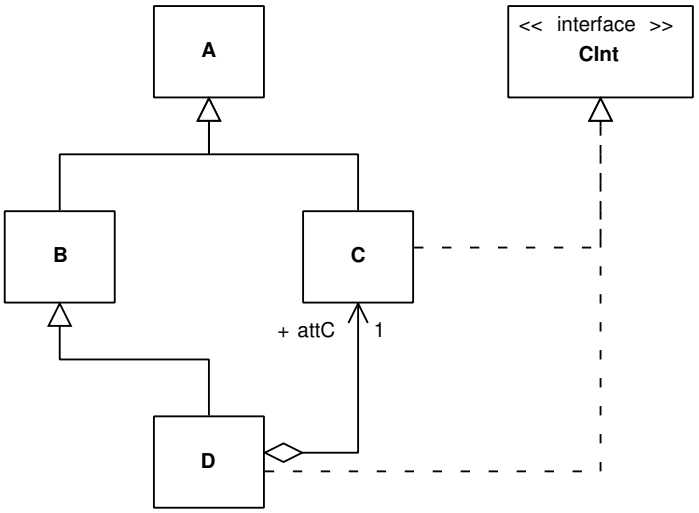

FIGURE IV. 19 – Héritage multiple, simulé par délégation et interface

Comme précédemment, la délégation simule l'héritage. De plus, on peut utiliser `CInt` pour faire du sous-typage :

```
CInt d2 = new D(...) ; // ok
```

On peut appliquer les méthodes de C (et de A) à `d2` :

```
d2.methA(...) ;
d2.methC(...) ;
```

L'objet `d2` n'est pas de type C, par contre c'est un objet de type `CInt`. Si la méthode `methC` est déclarée dans C, et redéfinie dans D, on a bien le mécanisme de liaison dynamique :

```
CInt c1 = new C(...) ;
c1.methC(...) ; // Appel de methC dans C.

CInt c2 = new D(...) ;
c2.methC(...) ; // Liaison dynamique : appel de methC dans D.
```

Cette approche présente cependant certains inconvénients :

- Il faut recopier dans `CInt` tous les profils des méthodes de C, y compris les méthodes héritées. Le programme est donc difficile à maintenir, par exemple si on modifie une classe héritée dans C, comme A.
- Les attributs de classes hérités à la fois par B et C, comme ceux de la classe A, sont dupliqués :
  - on récupère un attribut par l'intermédiaire de la classe B, héritée dans D ;
  - on récupère un attribut par le délégué `attC`.

Bibliographie

M.-C. Gaudel, B. Marre, F. Schlienger, G. Bernot. *Précis de génie logiciel.* Masson 1996.

J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

P.-A. Muller, N. Gaertner. *Modélisation objet avec UML. Deuxième édition.* Eyrolles, 2000.

J. Rumbaugh, I. Jacobson, G. Booch. *Unified Modeling Language Reference Manuel.* Addison Wesley, 1999.

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

Rémy Fannader, Hervé Leroux. *UML, Principes de modélisation.* Dunod, 1999.

C. Larman. *UML et les Design Patterns.* Campus Press, 2002.

Robert C. Martin. *Agile Software Development. Principles, Patterns and Practices.* Prentice Hall 2002.

Robert C. Martin. *UML for Java Programmers.* Prentice Hall 2003.

Sinan Si Alhir. *Introduction à UML.* O'Reilly, 2004.

F. Buschman, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture. A System of Patterns.* Wiley, 1996.

Unified Modeling Language Specification (2.0). OMG, 2005.

Site sur UML : `http://www.uml.org`