Première partie

UML

Chapitre 1

Langage de modélisation

1.1 Vocabulaire

Le sigle *UML* se lit en anglais. Il signifie *Unified Modeling Language* (en français, « langage de modélisation unifié »).

UML est un *langage de modélisation graphique*. Il est apparu dans le monde de la conception et du développement des systèmes informatiques, principalement dans le cadre de la « conception orientée objet ». Aujourd'hui, UML est naturellement utilisé dans les projets logiciels, mais on peut l'appliquer pour spécifier toutes sortes de systèmes sans se limiter au domaine informatique.
Dans ce livre, nous tirons profit d'UML, en tant que langage de modélisation, pour décrire un système spécifique, celui du psychisme humain.

Ce paragraphe d'introduction provoque probablement chez les lecteurs – lorsque ceux-ci ne sont familiers, ni avec les projets industriels, ni avec l'informatique – de nombreuses questions :

- Qu'est-ce qu'un projet, un produit, un système ?
- Qu'est-ce que l'analyse, la conception, et le développement ?
- Qu'est-ce que l'analyse et la conception orientées objet ?
- Qu'est-ce qu'un modèle ?
- Qu'est-ce que la modélisation?
- Qu'est-ce qu'un langage de modélisation ?
- Qu'est-ce qu'un langage de modélisation graphique ?
- ...

Notre premier but consiste ici à démontrer que la résonance technologique de ce vocabulaire n'est pas un obstacle, et qu'il suffit d'appréhender les principes de la modélisation pour ne pas rester emprisonné dans un environnement technique spécifique.
C'est pourquoi nous commençons par clarifier les termes de ce vocabulaire, afin de rendre limpide la manipulation d'UML et de rendre la lecture de ce livre agréable.

1.2 Projet, produit, système

Un *projet* désigne un ensemble de tâches accomplies le plus souvent par plusieurs personnes, ou plusieurs équipes, ou plusieurs corps de métier – dans une durée de temps limitée pouvant aller de quelques jours à plusieurs années – avec pour objectif de fabriquer un *produit*.
Des exemples de projets sont : la construction d'un immeuble, d'une route, d'une ligne de métro ... ou bien la conception d'un modèle d'automobile, la création d'un logiciel de courrier électronique, etc.

Lorsque le fonctionnement d'un produit implique la coordination du fonctionnement de divers éléments composant ce produit, on parle aussi de *système*.
Des exemple de systèmes fabriqués par l'homme – on emploie le terme d'*artefacts* pour désigner les produits de l'activité humaine – sont : un aéroport, un système de transports urbain comme un ensemble de lignes de bus, de métro, de tramway, ou bien un système de télécommunications ... Globalement, les produits informatiques sont tous des systèmes.

On utilise aussi le terme de système pour désigner des *systèmes naturels* existants dont on veut décrire le fonctionnement complexe : système planétaire, système nerveux, sanguin, hormonal, ou bien *système psychique que nous décrivons dans ce livre*.

1.3 Processus de développement des projets

Nous faisons dans ce paragraphe un détour succint par le *processus de développement* des projets, afin de décrire la place de *l'analyse des systèmes* dans la succession des tâches composant le *cycle de vie* d'un projet.
Un projet s'applique toujours à la production d'artefacts. Au départ d'un projet, il y a des *besoins*, exprimés par exemple sous la forme d'un *cahier*

des charges. L'objectif du projet est de fabriquer un produit répondant à ces besoins.

On distingue plusieurs phases dans le processus de développement d'un projet :

- la phase d'initialisation ;
- les phases d'analyse logique et de conception globale (on parle aussi d'élaboration) ;
- dans les systèmes complexes, la phase d'architecture est menée en même temps que l'élaboration ;
- la phase de conception détaillée ;
- la phase de développement (on parle aussi de construction) et de tests unitaires ;
- la validation, la livraison et la maintenance (on parle aussi de transition).

1.3.1 Initialisation

Très souvent, l'expression de la demande ne décrit pas forcément de façon suffisamment claire et précise les besoins des utilisateurs potentiels du système projeté.
Pour mener à bien un projet, il est nécessaire de comprendre au mieux et le plus complètement possible les besoins attendus du futur produit. Si l'on prend l'exemple – accessible à tous – d'un projet de construction de maison individuelle, l'architecte se préoccupe d'abord de comprendre les particularités d'usage que les futurs habitants feront de leur domicile, et quelle organisation des pièces leur conviendra le mieux.

On appelle *spécification fonctionnelle* la définition des exigences qui conduisent à la mise en place d'un projet pour créer un produit, puisque l'usage qui sera fait du futur produit se rapporte généralement à l'utilisation des *fonctions* fournies par le produit.
Par exemple, le sous-sol d'une maison peut avoir plusieurs fonctions : stationnement des automobiles, entretien de ces automobiles, salle de bricolage, salle de sport, salle de jeux pour les enfants, cellier, cave à vin, buanderie, débarras, etc.

Par ailleurs, il faut prendre en compte le plus tôt possible les contraintes s'appliquant à la fois au produit et au déroulement du projet. Dans l'exemple de la construction d'une maison, il faut gérer les contraintes de profil de

terrain ou d'orientation des pièces, les contraintes climatologiques, les contraintes administratives ou bien simplement de coût ..., la plupart des contraintes étant souvent liées entre elles.

1.3.2 Analyse logique

À partir de ce que devra faire un système (la spécification fonctionnelle), la phase d'*analyse logique* permet de spécifier clairement comment il sera organisé pour remplir les fonctions attendues : quels seront les principaux composants, quelles seront leurs responsabilités, et comment ils communiqueront entre eux.
Cette étape est nécessaire, essentielle, et difficile, car il n'y a aucune raison – aussi bien dans un système en projet que dans un système existant – d'établir une correspondance bi-univoque entre les fonctions observables ou bien invocables de l'extérieur, et l'organisation interne d'un système. Au contraire, certains modules internes d'un système participent le plus souvent à différentes fonctions, et il n'existe pas de module effectuant à lui seul toutes les tâches d'une seule fonction.

On parle aussi de *conception globale*, puisque l'on se préoccupe avant tout d'un point de vue de haut niveau sur l'ensemble du système et que l'on néglige volontairement les détails.
Certaines tâches de l'analyse logique sont aussi des tâches d'*architecture* : par exemple, dans le cas de la construction d'un aéroport, l'architecture définit l'organisation des différents modules et la communication entre eux : salles d'embarquement, d'arrivée, douanes, police, parkings, etc.

Mais l'analyse permet aussi de comprendre et de décrire la structure d'un système existant. C'est cette utilisation de l'analyse qui nous intéresse dans ce livre.
Nous complétons maintenant cette section en citant les autres phases du cycle de vie d'un projet, mais nous ne les détaillons pas puisqu'elles ne concernent pas l'objectif de ce livre.

1.3.3 Conception détaillée

Généralement, le résultat de la phase d'analyse se situe à un niveau d'abstraction trop élevé pour passer immédiatement à la construction d'un système.
La conception détaillée permet de préparer le développement. Elle aborde

le projet de manière concrète et définit de façon précise et exhaustive tous les détails de la construction.
Dans l'exemple de la construction d'une maison ou d'un immeuble, là où l'architecture et l'analyse ont pu rester vagues, il reste des détails à préciser pour les ouvriers des différents corps de métier : nature des matériaux constituant les cloisons, emplacement des gaines électriques, équipement des salles de bains, types de menuiseries, etc.

1.3.4 Développement

La phase de développement (construction) fabrique le produit ou les instances du produit : la maison individuelle, le logiciel, ou les automobiles.
Généralement, il faut gérer la succession de différentes tâches, par exemple l'intervention des maçons, puis des couvreurs, puis des électriciens , puis des plâtriers dans la construction d'une maison. Pour une automobile, il faut fabriquer d'un côté les moteurs, d'un autre les châssis, d'un autre les carrosseries, puis les intégrer, les habiller, les peindre ... et gérer les pièces produites par les sous-traitants.

En plus des tests unitaires qui vérifient la construction des modules, des phases de tests se déroulent à intervalles réguliers, afin de valider au fur et à mesure la construction du produit et d'éviter des dérives qui conduiraient à la fabrication d'un produit ne correspondant pas à la demande du client.

1.3.5 Livraison

Une fois un produit construit, il peut être validé, puis livré ou vendu. La maintenance permet de corriger les défauts éventuels après la livraison.

1.4 Modélisation

Il nous a paru nécessaire de faire ce bref détour par la gestion des projets industriels et informatiques – même si pour certains lecteurs ce ne sont que des banalités, alors que nous espérons que ce détour ne paraît pas trop aride pour d'autre lecteurs – afin d'arriver au cœur de nos

préoccupations dans ce livre, la notion de modèle.
La *modélisation* permet de comprendre des systèmes existants, ou bien de spécifier des systèmes en projet. Dans les deux cas, la modélisation permet d'affronter la complexité. Elle est l'outil essentiel des tâches d'analyse, de conception globale, et d'architecture.

1.4.1 Modèles

Un *modèle* est un point de vue simplifié de la réalité. C'est une *abstraction* d'un système ou d'une partie d'un système.
L'objectif de la modélisation n'est pas l'exhaustivité, mais la clarté du modèle. Pour cela, un modèle doit normalement se focaliser sur un point de vue d'un système, en simplifiant ou en négligeant la plupart des aspects concrets qui seraient inutiles par rapport au point de vue ciblé.
Selon les points de vue, il existe par conséquent plusieurs modèles différents capables de décrire un système donné.

Des modèles simples à comprendre peuvent suffire pour spécifier clairement la structure et le comportement d'un système existant, ou bien – dans un projet – d'une solution ou d'un prototype de solution. Par extension, les modèles constituent un excellent outil de communication entre les intervenants d'un projet ; ils permettent aussi d'organiser les développements et de documenter un produit.

Dans ce livre, nous construisons un modèle descriptif du système psychique humain, en appliquant le point de vue de la psychanalyse.

1.4.2 Démarche de modélisation

La démarche de modélisation a pour but de rechercher des abstractions et de faire apparaître les concepts essentiels pour comprendre un système.
Deux approches sont à considérer :

- un modèle peut servir à représenter la structure d'un système, c'est-à-dire à modéliser les éléments du système et leurs interactions internes ;
- un modèle peut aussi reproduire les fonctions d'un système, c'est-à-dire modéliser le comportement externe du système, sans nécessairement reproduire la structure interne.

Lorsque la démarche de modélisation s'appuie sur un langage commun, ce langage facilite les échanges entre utilisateurs au cours de la description de systèmes existants ou bien entre intervenants sur un même projet.

Le modèle élaboré dans ce livre a pour objectif d'analyser la structure des concepts de la psychanalyse, et de comprendre leurs interactions afin d'apprendre à les manipuler dans la pratique.
Ce modèle est construit avec le point de vue de la division signifiante introduit dans l'enseignement de Lacan. D'autres points de vue existent sur la psychanalyse. Nous ne les prenons pas en compte dans ce livre.

1.4.3 Analyse et conception orientées objet

L'*orientation objet* est un paradigme d'analyse, de conception, et de programmation utilisé dans le domaine de l'informatique. L'approche objet est apparue très tôt, à partir des langages de programmation, avec le langage *Simula 67* dès les années 1960. Elle s'est affirmée dans les années 1980 avec le langage *Smalltalk*. Elle est enfin devenue prépondérante dans les années 1990, par exemple avec les langages *C++* et *Java*.
Parallèlement, dès les années 1980, l'approche objet s'est généralisée dans les tâches d'analyse et de conception, car elle permet d'affronter la complexité croissante des grands systèmes, beaucoup plus facilement que ne l'autorise l'approche procédurale[1].
Les techniques orientées objet sont mieux adaptées à la modélisation que ne l'est la décomposition fonctionnelle[2], qui tend à produire des modèles plus fragiles aux évolutions et par conséquent plus difficiles à maintenir.

Un sytème – quel qu'il soit, mais cette approche devient surtout primordiale dans le cas des systèmes complexes – peut se modéliser comme une collection d'éléments capables d'interagir pour travailler ensemble. Ce sont les « objets ». L'organisation des objets, et le chemin de leurs interactions pour coopérer afin de réaliser le comportement du système, représentent la structure du système.

1. L'approche procédurale sépare les données et le code. Le code réalise des fonctions en travaillant sur des données qui lui sont passées en argument. Cette approche repose sur une connaisance globale de ce qui peut être fait dans un système. Lorsque l'on doit développer des évolutions, il est le plus souvent nécessaire de modifier le code à plusieurs endroits.

2. La décomposition fonctionnelle est une méthode d'analyse qui découpe hiérarchiquement les fonctions de haut niveau en fonctions plus élémentaires.

Objets

Il existe – principalement dans le domaine informatique – une abondante littérature spécialisée traitant de l'orientation objet. Les lecteurs intéressés peuvent s'y reporter à loisir.

Un objet est un élément concret faisant partie d'un système concret et jouant un rôle dans ce système.
Un objet est un élément de granularité moyenne, plus petit qu'un système, qu'un sous-système ou qu'un module, mais plus grand qu'une donnée élémentaire. La taille d'un objet doit correspondre à une granulatité facile à gérer et garantissant la cohésion du contenu de l'objet. Nous avons choisi trois mots-clés définissant – à un niveau suffisamment découplé de la technologie informatique – les objets et leurs relations :

- *l'encapsulation* : un objet détient des valeurs de données qui lui sont propres (les valeurs de ses *attributs*) et un comportement (les *implémentations*[3] de ses *opérations*). Avec l'encapsulation, il n'y a plus de données globales comme avec l'approche procédurale. Chaque objet détient les données sur lesquelles il travaille au lieu de les recevoir en argument. Un objet doit toujours masquer au monde extérieur, c'est-à-dire encapsuler, les valeurs de ses attributs ainsi que la façon dont il réalise ses opérations ;
- *les responsabilités* : dans un système, les objets travaillent ensemble pour fournir au monde extérieur le comportement global du système. Dans ce but collaboratif, les objets utilisent les liens qui les relient à d'autres objets internes au système pour échanger des messages afin de se demander des services. Lorsque un objet est capable de rendre des services à d'autres objets, il en prend la responsabilité. Il doit rendre publique pour les éventuels objets clients la liste des services (on parle de son *interface*) dont il est responsable ;
- *la délégation* : dans l'approche objet, tous les détails concernant un savoir-faire doivent être encapsulés localement dans des objets – à l'opposé de l'approche procédurale dans laquelle ces détails pouvaient être connus globalement ou modifiés à plusieurs niveaux. Par conséquent, un objet ayant besoin d'un service doit toujours délèguer ce travail à d'autres objets qui en prennent la responsabilité, et il ne doit jamais se préoccuper des détails de réalisation.

3. Le terme *implémentation* est un anglicisme passé dans le langage courant des informaticiens. Il désigne le code qui réalise concrètement le comportement correspondant à une action ou bien à une succession d'actions.

Approche objet

D'une manière simple, pour comprendre l'approche objet, il suffit de se représenter un système comme un ensemble d'éléments (les objets) capables de coopérer, c'est-à-dire de communiquer entre eux et de se rendre des services les uns aux autres afin de fournir des fonctions globales aux utilisateurs externes du système.

Pour reformuler les trois mots-clés précédents, chaque objet masque son propre savoir-faire, il assume l'entière responsabilité des tâches qu'on lui demande d'accomplir, et, plutôt que de se mêler de tout, il sait déléguer à d'autres ce qu'il ne fait pas lui-même.

D'autres mots-clés importants sont nécessaires pour maîtriser la modélisation objet : la *généralisation*, les *dépendances*, les *messages*, les interfaces (déjà citées) ... Nous les décrivons dans le chapitre 2.

Classes

Les objets sont les éléments concrets qui travaillent dans un système. Par conséquent, l'identification et la sélection des objets est toujours une première tâche fructueuse dans l'analyse d'un système. Cependant, la construction d'un modèle demande de passer à un niveau plus abstrait. Lorsque des objets ont la même structure et effectuent le même travail, on dit qu'ils sont d'un même *type*. Dans un système donné, il peut à un certain moment y avoir un seul ou plusieurs objets d'un type donné.
Pour définir de façon générique un type d'objet, on utilise des abstractions appelées les *classes*.
On dit que les objets sont des *instances* de classes.

Par conséquent, dans un modèle, une classe représente un concept ou un élément du monde réel pouvant être instancié à un ou plusieurs exemplaires.
La première tâche d'une modélisation objet – celle qui correspond à l'analyse logique – consiste à identifier les objets du monde réel ou bien les concepts importants existant dans un système à analyser ou dans un système à construire.
Ensuite, il faut spécifier les classes décrivant ces objets ou ces concepts (leur structure et leur comportement) avec une granularité adéquate pour parvenir à une bonne description du système.

Relations

Bien entendu, un objet isolé n'aurait pas de sens puisqu'il ne pourrait pas travailler pour d'autres objets, ni faire travailler les autres. Pour qu'un système soit opérationnel, il est indispensable que les objets concrets aient des liens entre eux afin de jouer des rôles à l'intérieur du système.
L'*association* entre des classes décrit la connexion entre les objets instances de ces classes. Une association montre que des objets se connaissent (de manière uni-directionnelle ou bien réciproquement), c'est-à-dire qu'ils peuvent communiquer en s'envoyant des messages afin de travailler ensemble.
Les associations sont un type de *relations* entre classes. Il existe d'autres types de relations, que nous détaillons au chapitre 2.

La seconde étape de l'analyse logique dans une modélisation objet consiste à identifier les associations (et les relations dans un sens plus large) entre les classes. Cette étape est primordiale puisqu'elle détermine le fonctionnement du système. Elle n'est pas facile, puisqu'il ne faut pas oublier d'identifier des associations, mais il serait par contre périlleux d'en créer d'inutiles.

Diagrammes de classes

Les diagrammes apportent une *information topologique* nécessaire pour synthétiser l'organisation des éléments.
Un diagramme montrant les classes et leurs relations est appelé *diagramme de classes*. C'est l'outil essentiel permettant de décrire la structure d'un système.

Pour compléter la remarque du paragraphe 1.3.2 – qui établit une séparation majeure entre la vision fonctionnelle externe d'un système d'une part, et la vision interne du même système découpé en modules d'autre part – ce sont les diagrammes de classes qui décrivent l'organisation interne.
Cette étape fondamentale de l'analyse logique – *que l'on appelle « le passage à l'objet »* – permet de spécifier la structure d'un système.

1.4.4 Langage de modélisation graphique

Langage de modélisation

On parle de *langage de modélisation*, parce que les éléments du langage sont appropriés à la description ou bien à la spécification de la structure et du comportement de systèmes complexes, c'est-à-dire à la création de modèles.

Langage de modélisation graphique

UML est un langage écrit. L'écriture d'UML n'est pas basée sur les lettres de l'alphabet, mais sur le dessin de *pictogrammes*. Les modèles UML sont rédigés en associant entre eux des *idéogrammes* représentés par des pictogrammes.
Pour cette raison, UML est un *langage graphique* : écrire un diagramme UML est principalement une tâche de dessin associant des pictogrammes. Les éléments de notation du langage UML sont essentiellement des éléments graphiques. Bien sûr, des fragments de textes en langage naturel sont aussi utilisés dans les diagrammes UML, mais ce sont uniquement des valeurs d'attributs apportant des précisions sur les idéogrammes et ils ne constituent jamais le texte principal des modèles au sens UML.

La première partie de ce livre a pour objectif l'apprentissage des éléments de notation graphique d'UML, en insistant sur le dessin et la signification des éléments que nous utilisons dans la suite du livre pour atteindre notre objectif spécifique, celui d'employer le langage UML afin de modéliser la structure du psychisme humain avec le point de vue de la psychanalyse.

1.5 Aperçu d'UML

1.5.1 Repères temporels

La fin des années 1980 et le début des années 1990 ont vu apparaître de nombreuses méthodes de développement pour les systèmes informatiques. Ces méthodes avaient en commun d'intégrer l'orientation objet, mais elles avaient des approches légérement différentes et des notations foncièrement divergentes pour représenter des concepts voisins.

La naissance d'UML résulte de l'alliance entre trois créateurs, dans l'objectif d'unifier leurs trois méthodes qui présentaient chacune des avantages dominants dans une étape du domaine de la gestion de projets :

- Grady BOOCH : les modèles de conception ;
- James RUMBAUGH : l'analyse et le traitement des données ;
- Ivar JACOBSON : l'analyse des besoins et des fonctionnalités de haut niveau.

UML est le travail d'une équipe élargie à de nombreux autres auteurs dans le domaine de la méthodologie de développement de projets. Pour cette raison, UML intègre d'autres aspects provenant d'autres méthodes, ou bien essentiels pour la modélisation.

Les jalons temporels suivants marquent l'histoire d'UML :

- les travaux sur UML ont commencé en 1994 ;
- UML 1.1 est devenu un standard en 1997 ;
- la version UML 2.0 est apparue en 2004.

1.5.2 Objectifs d'UML

Dans le cadre spécifique de la création d'un langage commun à divers processus de développement de projets, UML a été conçu pour répondre aux objectifs suivants :

- comprendre des problèmes ;
- spécifier des modèles ;
- construire des solutions ;
- documenter des systèmes et des produits.

Par extension, dans une approche plus large, UML insiste sur la modélisation et satisfait aux exigences suivantes :

- définir un ensemble commun d'éléments de modélisation indépendant des domaines d'application, c'est-à-dire permettant de modéliser facilement toutes sortes de systèmes, mêmes non logiciels ;
- fournir aux utilisateurs un langage de modélisation fondamentalement basé sur l'approche objet ;
- fournir un langage graphique pour décrire des modèles en les dessinant ;

- fournir un langage permettant de manipuler, dans le cadre de différents diagrammes, les mêmes éléments de notation et la sémantique associée ;
- reposer sur un ensemble de concepts universels et être évolutif par extension de ces concepts de base.

En effet, au delà de l'objectif d'unification de méthodes, le terme « unifié » s'applique aussi à l'objectif de modéliser toutes sortes de systèmes sans se cantonner au domaine informatique ou technique.
C'est cette acception de l'unification qui nous intéresse dans ce livre.

1.5.3 Syntaxe d'UML

Conformément à la définition d'un langage de modélisation graphique (paragraphe 1.4.4), le vocabulaire d'UML est constitué d'un ensemble de pictogrammes représentant les éléments de base et les relations. Les éléments de base décrivent des choses ou des concepts. Les relations sont des éléments de notation décrivant comment divers éléments de base sont reliés entre eux.

Les éléments visuels contenant des collections d'éléments de base et de relations sont appelés des *diagrammes*.
Les diagrammes UML sont la traduction sous une forme graphique d'une partie d'un problème ou d'une solution. Inversement, un diagramme UML doit pouvoir se lire et s'expliquer en utilisant des phrases simples.
Le dessin de diagrammes revient à poser des concepts et à les relier entre eux. Par conséquent, on peut comparer la création de diagrammes à l'écriture de paragraphes ou de chapitres lors de la rédaction d'un document en langage naturel.
Les règles définissant comment écrire des éléments ou bien des relations, puis comment les associer entre eux pour créer des diagrammes, constituent la *syntaxe* du langage UML.

L'apprentissage d'UML repose sur l'acquisition de la sémantique associée aux différents pictogrammes composant la notation. Comme pour tous les langages, il faut débuter par les rudiments, apprendre les tournures de phrases en lisant des exemples, et s'entraîner en écrivant UML sous forme graphique.

1.5.4 Mise en œuvre d'UML

UML définit un modèle sémantique, mais ne prescrit pas de mise en œuvre.
Plus exactement, UML est un langage compréhensible aussi bien par les êtres humains (tableau, papier-crayon, transparents, documents écrits) que par les ordinateurs (logiciels de modélisation).

Logiciels de modélisation

Dans le cadre des projets industriels, le langage UML est le plus souvent mis en oeuvre par des outils logiciels qui fournissent des interfaces graphiques évoluées permettant de dessiner les diagrammes de manière standard, de conserver la cohérence entre les éléments du modèle apparaissant dans des diagrammes différents, puis de générer automatiquement une grande partie du code informatique à partir des modèles.
Les concepts MDE (*Model Driven Engineering*) et MDA (*Model Driven Architecture*) ont pour objectif la rationalisation de la production de logiciels.
Les plupart des fonctionnalités formelles ajoutées à la version UML 2 vont dans le sens de la production automatique de code informatique guidée par les modèles, tout en garantissant la correspondance entre les modèles et le code.

Analyse des systèmes

Néanmoins, les exigences industrielles n'ont pas condamné l'usage du mode papier-crayon. Lorsqu'il s'agit de créer des modèles pour comprendre des systèmes complexes, l'objectif principal n'est pas la génération de code informatique ni la productivité immédiate, mais l'acquisition et la formalisation d'un savoir.
Il faut alors dessiner, corriger, discuter, échanger, commenter des diagrammes, reformuler des concepts ... Pour toutes ces tâches de recherche, UML offre un avantage considérable, celui de fournir une représentation visuelle des concepts utilisés et de leurs liens.
Cette approche d'UML est celle qui nous a guidé dans l'écriture des diagrammes de ce livre.

Chapitre 2

Structure d'UML

Dans le chapitre 1, nous avons décrit le contexte d'apparition du langage UML. Ce second chapitre a maintenant deux objectifs.

Le premier objectif répond à une exigence d'ordre générique dans un livre qui utilise UML. Il consiste à présenter une vue d'ensemble des éléments d'écriture du langage UML – les éléments de base, les relations, et les diagrammes – ainsi que leur notation, c'est-à-dire les spécificités d'écriture associées aux divers éléments.
Cette présentation reste succinte. Elle ne vise pas l'exhaustivité et apporte simplement un panorama global des éléments fournis par UML pour créer des modèles et décrire des systèmes.

Le deuxième objectif répond à une exigence plus spécifique : procurer aux lecteurs de ce livre un guide de référence des éléments de la notation UML, que nous manipulons pour modéliser la structure des éléments du psychisme utilisés dans le domaine de la psychanalyse.
Cette référence a pour but de décrire de manière détaillée la sémantique et la notation des éléments UML utilisés dans ce livre, pour que la lecture des diagrammes de la partie III soit limpide, mais aussi pour que les lecteurs puissent à leur tour écrire aisément de nouveaux diagrammes afin d'enrichir et de faire évoluer le modèle UML de la psychanalyse.

Pour illustrer la modélisation UML, *nous avons choisi de construire un petit modèle en nous inspirant du cadre de la vie courante, celui de la boulangerie où nous achetons le pain chaque jour, les croissants du petit déjeuner du dimanche, et les gâteaux des jours de fête.* Notre approche technique du monde de la boulangerie est sans doute imparfaite, mais peu importe, l'essentiel étant de montrer les concepts et la notation UML.

2.1 Éléments structurels

2.1.1 Classes

Concept

Les *classes* décrivent les caractéristiques des éléments présents dans un système. À ce titre, elles jouent un rôle essentiel puisque ce sont elles qui définissent les concepts du domaine.
Dans un modèle objet bien construit – c'est-à-dire dans lequel les objets coopèrent pour réaliser les fonctions du système – les classes montrent les *responsabilités* des objets.

Une classe spécifie l'état (les *attributs*) et le comportement (les *opérations*) des objets qu'elle spécifie.
La granularité d'une classe doit fournir une bonne cohésion conceptuelle, c'est-à-dire être assez grande pour ne pas se limiter à des valeurs d'attributs ou à des opérations trop élémentaires, ni trop grande pour ne pas se disperser dans des responsabilités hétérogènes.

Notation

Une classe est représentée par un *rectangle* (figure 2.1). Celui-ci peut être divisé en *compartiments* (figure 2.2). Le premier compartiment – obligatoire – contient le nom de la classe. Le deuxième, s'il est présent, contient la description des attributs. Le troisième contient la description des opérations s'appliquant aux objets de la classe. Il est légal du point de vue de la spécification UML, quoique peu courant, d'adjoindre un quatrième compartiment pour énumérer les responsabilités assumées par les objets de la classe. Les règles d'écriture essentielles sont les suivantes :

- le nom d'une classe doit toujours être présent. Par convention, un nom de classe commence par une lettre majuscule ;
- les compartiments des attributs ou des opérations peuvent être laissés vides ou être absents. Cela ne veut pas dire qu'il n'y ait pas d'attributs ou d'opérations, mais seulement que, dans le modèle en cours d'écriture, cela n'a pas d'intérêt de les citer ;
- si l'on cite des opérations sans citer d'attributs, le second compartiment doit exister et rester vide.

Classes d'analyse

Les classes interviennent tout d'abord comme abstractions dans l'analyse logique d'un problème et dans la constitution des premiers modèles d'un domaine. À ce stade, les concepts représentés par les classes sont des éléments de réflexion avec les spécialistes du domaine (les experts-métier), ainsi que de compréhension pour les ingénieurs analystes ou les architectes de systèmes.

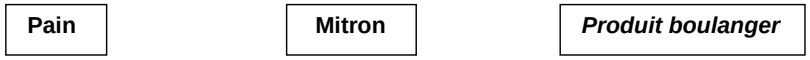

FIG. 2.1 –: *Classes d'analyse*

La figure 2.1 montre quelques exemples de classes d'analyse.
En général, on ne s'intéresse pas aux détails des classes, mais aux objets présents et à leurs interactions. Plutôt que de conserver les compartiments des attributs et des opérations en les laissant vides, il est plus élégant de représenter une classe comme un simple rectangle contenant uniquement le nom de la classe.
Sur la figure 2.1, les classes *Pain* et *Mitron* sont des classes concrètes : dans le système en fonctionnement, il existe des objets instances de ces classes. La classe *Produit Boulanger* est une classe abstraite : bien sûr il existe des produits boulangers dans le système, mais ce sont concrètement des pains, des baguettes, des chocolatines … Une classe abstraite représente un niveau de regroupement d'ensembles d'objets selon des critères génériques. Le nom d'une classe abstraite s'écrit en italique.

Dans ce livre, nous modélisons un système. Toutes les classes de nos diagrammes sont des classes d'analyse, représentées par un simple rectangle contenant un nom de classe.

Classes de conception

Au stade de la conception détaillée, les classes deviennent des sortes de patrons ou de moules décrivant précisément les détails des objets à instancier dans le système concret. Les classes de conception respectent un ensemble de règles formelles qui ont du sens pour les développeurs. Souvent ces classes sont directement utilisables par les outils de génération automatique de code informatique (voir paragraphe 1.5.4).

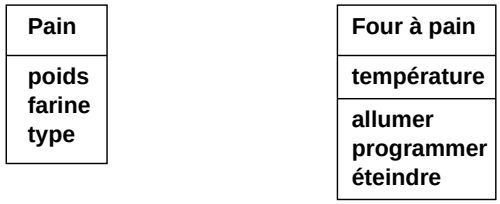

FIG. 2.2 –: *Classes de conception*

La figure 2.2 montre des exemples de classes de conception. Par convention, les noms d'attributs et d'opérations commencent par une lettre minuscule. On appelle *méthodes* les réalisations concrètes des opérations.

2.1.2 Objets

Concept

Un *objet* est une *instance* d'une classe. Les termes d'objet et d'instance sont synonymes en UML. Ils désignent un élément *concret* qui existe dans le système final. On dit aussi qu'un objet est *instancié* à partir d'une classe (le concept abstrait correspondant).
En UML, les objets sont utilisés dans les diagrammes d'objets, mais beaucoup plus souvent dans les diagrammes d'interaction (diagrammes de séquence ou de communication) pour représenter les rôles joués par les instances de classes dans les scénarios d'interaction.

Notation

La figure 2.3 montre différentes notations.

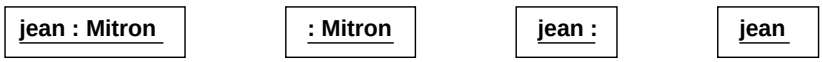

FIG. 2.3 –: *Objets*

Comme les classes, les objets sont représentés par un rectangle, comportant le plus souvent un seul compartiment.
Un nom d'objet commence par une lettre minuscule. Il est toujours souligné. Un nom d'objet peut être un nom *qualifié* (*nom d'instance : nom de*

classe) : ici, Jean est un mitron. Si le nom de classe est cité seul, il reste précédé du caractère « : ». On parle d'objet anonyme : un mitron dont on ne connaît pas le nom. Enfin, on peut – quoique cela soit rarement utile – désigner un objet « orphelin » sans citer la classe dont il est issu.

Dans notre modèle, nous utilisons les objets pour définir les rôles dans nos diagrammes de séquence. Puisque notre objectif se situe au niveau de l'analyse logique, nous avons pris quelques libertés avec le formalisme – lorsque cela ne nuit pas à la compréhension des diagrammes – en omettant de souligner les noms ou bien en omettant quelquefois le caractère « : ».

2.1.3 Classificateurs

Dans la spécification UML, le terme de *classificateur* désigne une abstraction plus générale que la classe. Les classificateurs représentent des entités encapsulant des caractéristiques propres. Ce sont par exemple des classes, mais aussi des interfaces, des composants, des noeuds, des acteurs, des cas d'utilisation ...

La notation d'un classificateur est le rectangle. Commes les classes en sont la représentation la plus courante, elles reprennent le rectangle comme notation. L'utilisation de mots-clés permet de stéréotyper les autres classificateurs.

2.1.4 Interfaces

Concept

Conceptuellement, une *interface* sert à définir et à publier vers le monde extérieur un ensemble spécifique d'interactions ou de services offerts par un objet. Lorsqu'une classe réalise une interface, on est certain que les objets instances de cette classe « savent accomplir » le comportement défini par l'interface.
On peut comparer une interface à un fragment de document de spécification définissant un type d'utilisation. Mais une interface peut également servir dans un simple but de classification des objets, en leur assignant un type sans associer nécessairement à ce type un comportement spécifique (voir les paragraphes 2.1.5 et 2.5.1).

Une interface peut être réalisée par plusieurs classes. Toutes ces classes garantissent alors un type d'utilisation commun, mais les implémenta-

tions qu'elles fournissent sont généralement différentes selon divers critères : performance, coût, sécurité, etc. De même, une classe donnée peut réaliser simultanément plusieurs interfaces.
La figure 2.4 montre qu'un boulanger et un mitron savent tous les deux faire du pain. Cela n'est pas restrictif, puisque beaucoup d'autres personnes – comme vous et moi – peuvent aussi savoir faire du pain (il suffit simplement de réaliser l'interface). De plus, un boulanger sait aussi gérer un magasin, ce que ne sait pas faire le mitron.
Du point de vue des types de classification, on voit que le boulanger et le mitron font partie de l'ensemble des gens qui savent faire du pain, alors que le boulanger fait aussi partie de l'ensemble des gestionnaires de magasin.

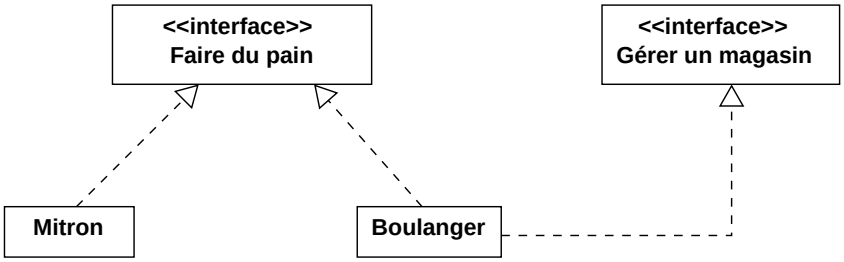

FIG. 2.4 –: *Interfaces*

Notation

Une interface est représentée par un rectangle de classe stéréotypé, ou bien par une icône. Nous montrons ici trois exemples de notation :

- la figure 2.4 montre une utilisation simple du stéréotype d'interface ;
- la figure 2.5 montre des interfaces détaillant la définition des services qu'elles spécifient ;
- la figure 2.6 montre l'utilisation d'une icône. Cette icône (un cercle complet) représente une interface « fournie », c'est-à-dire conforme au concept de la publication d'un ensemble de services. UML 2 définit la notion complémentaire d'interface « attendue », représentée par un demi-cercle, et déclarant un ensemble de services dont un objet a besoin et qu'il attend qu'on lui fournisse.

La relation entre une classe et une interface qu'elle réalise s'écrit différemment selon la notation représentant l'interface. Dans le cas du rec-

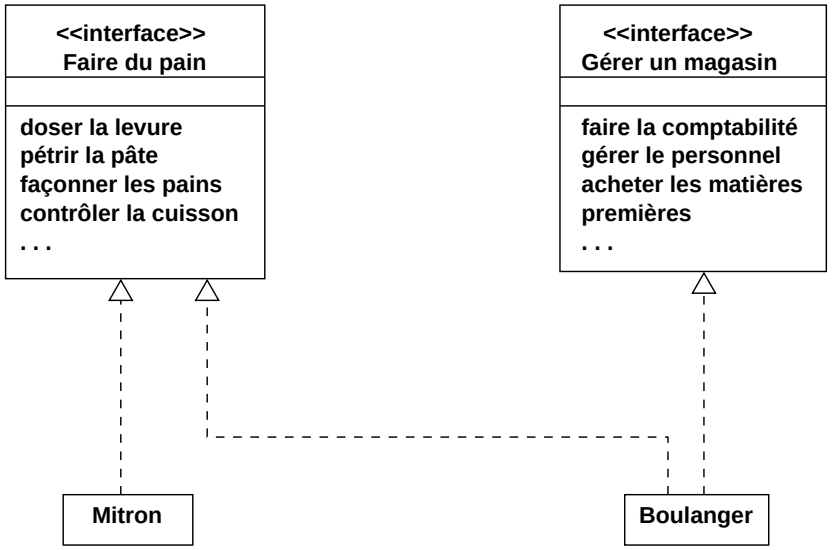

FIG. 2.5 –: *Interfaces documentées*

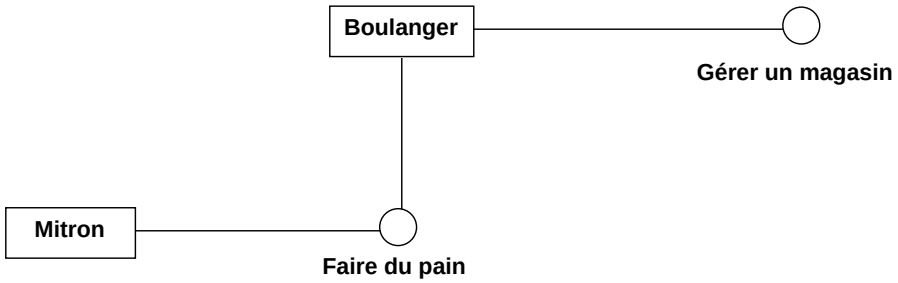

FIG. 2.6 –: *Icône d'interface « fournie »*

tangle stéréotypé, on emploie une relation de réalisation (paragraphe 2.6.9) ; dans le cas d'une icône, la relation est notée par un trait plein d'association reliant la classe et l'interface.

2.1.5 Types

Les *types* répondent à un besoin de classification des objets, en permettant de regrouper des objets ayant des caractéristiques communes. Une classe définit un type : tous les objets instances d'une même classe ont en commun le type défini par cette classe. De même, une interface définit un type : tous les objets instances de classes réalisant une certaine interface ont en commun le type défini par cette interface. Un objet a

aussi les types de toutes les classes mères (voir paragraphe 2.6.8) de la classe à partir de laquelle il est créé.
De plus, en modélisation UML, l'utilisation de stéréotypes (figure 2.13) permet de définir directement des types au stade de l'analyse logique.

En résumé, un objet a souvent plusieurs types. Selon le point de vue appliqué et selon les critères de classification choisis, un objet peut donc faire partie de différents ensembles regroupant des objets ayant un même type.

2.1.6 Composants

La notion de *composant* est fortement liée à la démarche de haut niveau qui consiste à construire des systèmes en assemblant divers éléments et en les reliant entre eux.
Dans un système, un composant est un module connu par les interfaces qu'il fournit (et aussi, en UML 2, par les interfaces qu'il requiert).
Les interfaces d'un composant représentent l'ébauche des chemins de connexion avec d'autres composants.

La figure 2.7 montre un exemple de composant : la caisse enregistreuse de notre boulangerie. Du point de vue de la notation, un composant est représenté par un rectangle avec deux onglets latéraux. Le nom d'un composant respecte les mêmes conventions qu'un nom de classe.

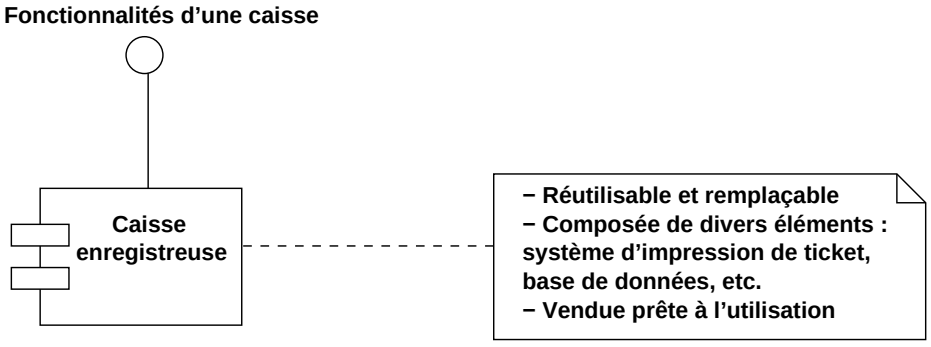

FIG. 2.7 –: *Composant*

Bien entendu, un composant s'engage à réaliser les interfaces fournies. Cependant, en comparaison avec d'autres éléments UML réalisant

des interfaces – par exemple les classes – les composants se différencient par des spécificités propres liées à la notion d'assemblage :

- *un composant est remplaçable.* On peut enlever un composant d'un système et le remplacer par un autre ayant les mêmes interfaces et les mêmes spécifications, sans que cela impacte les fonctionnalités globales du système. Mais le remplacement peut impacter les qualités non-fonctionnelles du système – cela constitue d'ailleurs le plus souvent une des motivations de l'échange – comme le coût de fonctionnement, les performances, la sécurité ... Par exemple, notre boulanger peut décider de remplacer sa caisse par un autre d'une autre marque pour de nombreuses raisons : l'ancienne est trop rustique, nouveau design, offre d'une maintenance gratuite ... Dans tous les cas, il suffit de débrancher l'ancienne caisse et de brancher la nouvelle à la place ;
- *un composant est réutilisable.* Une instance de composant peut s'intégrer dans différents systèmes et son fonctionnement n'est pas dépendant du système qui l'intègre. Par exemple, notre boulanger peut conserver son ancienne caisse en réserve en cas de panne de la nouvelle, la donner ou la vendre à un autre magasin (pas forcément une boulangerie), ou bien la réutiliser dans une succursale, etc.
- *un composant contient divers éléments éventuellement optionnels.* Le fonctionnement d'un composant est le résultat de la collaboration d'objets internes masqués par l'encapsulation. Surtout un composant se différencie d'un autre – ayant la même spécification – par des qualités de performance ou de coût, ou bien en offrant des options originales. Par exemple, la nouvelle caisse de notre boulangerie peut contenir une base de données mémorisant toutes sortes de paramètres historiques ;
- *un composant est fourni prêt à l'utilisation.* Excepté quelques détails de configuration, il doit être possible d'intégrer de façon transparente un nouveau composant dans un système ;
- *la taille d'un composant n'est pas un critère majeur.* Un composant peut être un élément léger, mais apporter un gain très important en étant remplaçable et réutilisable des millions de fois.

La note de la figure 2.7 est un résumé de ces spécificités propres aux composants. Dans notre boulangerie, on peut découvrir d'autres éléments modélisables en tant que composants : le four à pain, l'étalage. Un autre exemple de composant que nous manipulons fréquemment dans la vie courante est celui d'une ampoule électrique.

La notion de composant est un concept essentiel dans une approche architecturale. Par contre, pour la compréhension d'un système et pour la création d'un modèle d'analyse logique, les classes sont le plus souvent suffisantes.

2.1.7 Acteurs

Concept

Le concept d'*acteur* est utile dans une démarche d'analyse fonctionnelle. Un acteur représente un *élément externe* qui interagit avec un système, dans le but de le faire fonctionner et d'en tirer profit. Généralement il s'agit d'un utilisateur du système ou bien d'une entité qui coopère avec le système.
Un acteur ne représente pas une personne concrète, mais un *rôle* d'utilisation du système. Quand une même personne occupe différents rôles d'interaction avec un système, alors chacun de ces rôles définit un acteur différent dans le modèle. Par exemple, lorsque le mitron a fini son travail et qu'il achète une baguette de pain, il est dans un rôle de client comme tous les autres clients.

Un acteur peut être une personne ou bien un autre système. On différencie généralement les acteurs primaires qui interagissent avec un système pour en tirer profit, et les acteurs secondaires qui sont nécessaires au fonctionnement du système, mais n'en retirent pas forcément profit.
Les acteurs n'interagissent jamais directement entre eux.

Notation

FIG. 2.8 –: *Acteurs*

Un acteur est un stéréotype de classe : on le représente par l'icône standard du bonhomme en fil de fer (*stick-man*), ou bien par un rectangle de classe avec le stéréotype `<<acteur>>`. La figure 2.8 montre ces deux formes de notation.
Couramment, on utilise l'icône pour désigner des acteurs humains, et le

rectangle de classe pour représenter des systèmes, mais cela est seulement une convention.

Dans les diagrammes de cas d'utilisation, les acteurs sont connectés à des cas d'utilisation du système par des relations d'association (figure 2.9). De même, dans les diagrammes de séquence, les acteurs sont souvent utilisés pour matérialiser le point de départ – externe au système – des scénarios (figure 2.11).
Dans les deux cas, il s'agit de l'utilisation standard du concept d'acteur.

Dans nos diagrammes, nous avons choisi d'élargir le sens de l'icône du stick-man. En effet, notre modèle contient de nombreux diagrammes de classes. Afin de rendre « plus parlant l'aspect visuel » des diagrammes, nous avons décidé à plusieurs reprises d'utiliser l'icône du stick-man pour stéréotyper des classes représentant le concept de « sujet » – concept essentiel dans notre modèle – et non celui d'acteur au sens standard d'UML. Cette démarche n'est pas interdite. Toutefois, afin d'éviter des confusions, il est bien sûr nécessaire de préciser au préalable que l'on a choisi d'assigner une signification particulière à une icône UML standard.

2.1.8 Cas d'utilisation

Concept

De même que les acteurs, les *cas d'utilisation* sont des concepts d'analyse fonctionnelle. Ils décrivent les fonctionnalités de haut niveau d'un système ou d'un sous-système, en se plaçant du point de vue de l'utilisateur – c'est-à-dire ce qu'il « veut » faire (ou bien ce qu'il « peut » faire) avec le système – sans se préoccuper des mécanismes mis en jeu en interne (et par conséquent encapsulés) pour répondre à ses demandes.

Un cas d'utilisation constitue une session de travail : il a toujours un début impliquant une liste de *pré-conditions* devant être vérifiées pour commencer à travailler, ainsi qu'une fin décrite par un ensemble de *post-conditions* correspondant au travail accompli.
Un cas d'utilisation ne décrit aucune implémentation. Par contre, il est lui-même instancié par plusieurs scénarios décrivant des séquences d'activités. Il existe un scénario nominal (celui où tout se passe comme attendu) et plusieurs scénarios alternatifs ou d'exception. Dans le cas d'un scénario alternatif, certaines post-conditions sont vérifiées ; dans le cas d'un scénario d'exception, les post-conditions ne sont pas vérifiées.

Notation

Un cas d'utilisation est représenté par une ellipse en trait plein. Le nom du cas d'utilisation est disposé dans l'ellipse ou bien en dessous. La figure 2.9 montre un petit diagramme de cas d'utilisation décrivant les services qu'un client (en tant qu'acteur externe) s'attend à trouver dans une boulangerie (en tant que système).

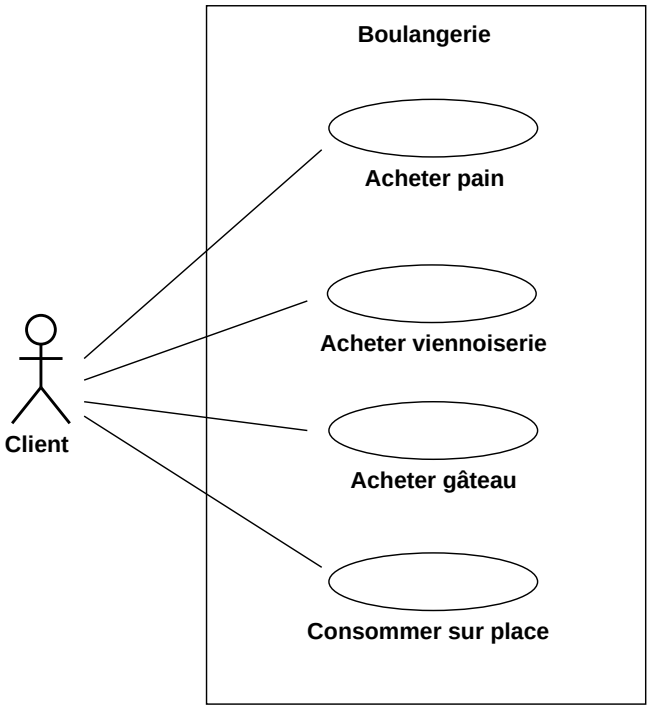

FIG. 2.9 –: *Diagramme de cas d'utilisation*

Le paragraphe 2.6.10 décrit les relations que l'on peut spécifier entre cas d'utilisation.

2.1.9 Collaborations

Les *collaborations* se placent au point charnière du processus de modélisation que nous avons appelé « le passage à l'objet » (paragraphe 1.4.3).

Lors une démarche de modélisation classique, il est courant de commencer par une analyse des besoins en représentant les interactions des utilisateurs avec le système. On obtient alors une représentation abstraite

d'un système sous la forme d'une boîte noire contenant un ensemble de cas d'utilisations.
Dans la modélisation objet, l'analyse fonctionnelle s'arrête à ce niveau. L'étape suivante se concentre sur la recherche de l'ensemble des éléments concrets coopérant entre eux pour réaliser chaque cas d'utilisation afin de fournir un certain comportement attendu.
C'est ce que l'on appelle une collaboration [*sous-entendu, entre objets*]. Une collaboration comporte un point de vue structurel (diagrammes de classes) montrant l'organisation des relations entre les objets, et un point de vue comportemental montrant les interactions entre les objets, ainsi que l'évolution au cours du temps de la société d'objets (diagrammes d'interaction).

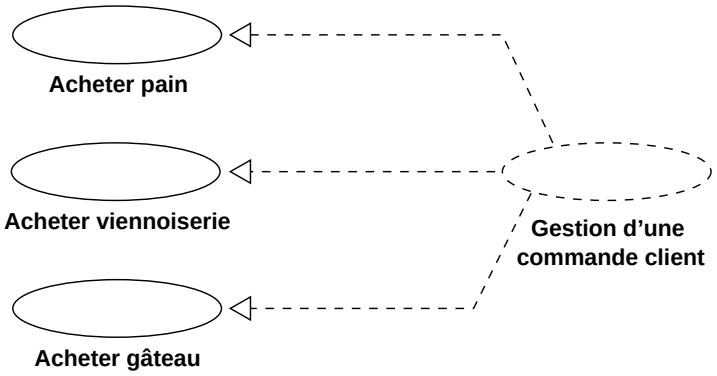

FIG. 2.10 –: *Collaboration*

Une collaboration se représente graphiquement par une ellipse dessinée en trait discontinu. Elle est reliée à des cas d'utilisation par des relations de réalisation (paragraphe 2.6.9).
La figure 2.10 montre une collaboration capable de réaliser différents scénarios de cas d'utilisation. La gestion d'une commande s'appuie probablement sur des interactions entre plusieurs instances de classes : la serveuse, l'étalage, le concept de commande, les produits … (voir figure 2.29, p. 51).

Nous n'utilisons pas les collaborations dans notre modèle.

2.1.10 Nœuds

Dans les projets industriels ou informatiques, un *nœud* représente une ressource physique concrète qui existe au moment de l'exploitation du

système. À l'intérieur d'un système informatique, les nœuds sont par exemple des ordinateurs reliés entre eux par divers types de réseau (réseau local, *Internet* ...) caractérisés par leur bande passante, leur fiabilité, leur coût, leur topologie, etc.
Graphiquement, un nœud est représenté par un parallépipède.

Nous n'utilisons pas les nœuds dans notre modèle.

2.2 Éléments comportementaux

Les éléments comportementaux décrivent la partie *dynamique* d'un système au cours du temps, c'est-à-dire comment les objets interagissent entre eux pour exécuter les tâches qui leur sont demandées, et par quels états passent ces objets en fonction des événements externes ou internes.

2.2.1 Messages

La notion de message est un point fort de l'orientation objet par rapport aux approches procédurales qui l'ont précédée.
Lorsqu'un objet envoie un message pour demander à un autre objet d'effectuer une action, c'est l'objet appelé qui décide lui-même de ce qu'il entend faire pour répondre à la demande reçue.

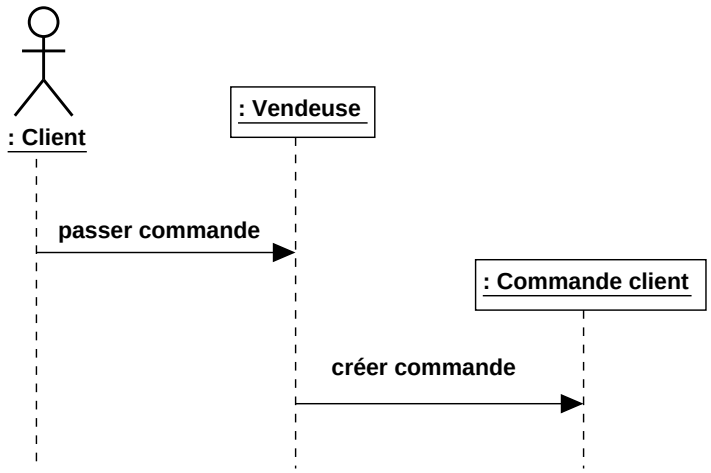

FIG. 2.11 –: *Messages*

La figure 2.11 est un fragment de diagramme de séquence. Bien sûr la vendeuse est à l'écoute des demandes du client, mais elle y répond à sa manière. De même, elle n'est pas censée savoir de quelle façon un objet de type `commande client` s'y prend pour mémoriser les détails des commandes.
Les messages décrivent la nature et le sens des informations que des objets échangent entre eux lorsqu'ils communiquent. Des messages peuvent être des requêtes, l'envoi de documents, de signaux, mais aussi des réponses à des messages de demande.
Graphiquement, un message est représenté par une ligne fléchée allant d'un objet vers un autre. Pour les phases de conception détaillée, il existe une syntaxe complète des types de flèches et des types de traits, afin de préciser formellement la nature des messages (appel synchrone, appel asynchrone, retour, etc.).

Nous utilisons les messages dans nos diagrammes de séquence.

2.2.2 États

Les *états* sont utilisés dans les diagrammes d'états-transitions afin de modéliser le cycle de vie d'un système, d'un sous-système, d'un objet ... La figure 2.30 montre les différents états décrivant le cycle de vie d'un objet « commande ».

Un état représente une phase pendant laquelle les attributs encapsulés par un élément restent conformes à certaines conditions. Dans un état, un élément peut effectuer une activité, ou bien être en attente de certains événements. Par exemple, lorsque notre objet se trouve dans l'état « choix produit », il est prêt à enregistrer un nouveau produit dès que le client se sera décidé en désignant un produit de son choix (éclair au chocolat, pain de campagne ...).
Le passage d'un état vers un autre état s'appelle une transition.

Graphiquement, un état se représente par un rectangle à bords arrondis, contenant le nom de l'état et éventuellement la description des actions, des activités ou des événements conceptuellement liés à cet état. Deux états spéciaux, l'état initial et l'état final, sont représentés par une notation spécifique. La figure 2.30 montre ces deux états.
La figure 2.30 montre aussi un *état composite*, c'est-à-dire un état comprenant des sous-états. Un état composite a généralement son propre état initial, mais rarement un état final, la transition de sortie se faisant direc-

tement vers un état extérieur.

Notre modèle ne contient pas de diagrammes d'états-transitions.

2.2.3 Transitions

Une *transition* représente le passage d'un état vers un autre. Elle est provoquée par un événement externe ou interne.

La figure 2.30 montre des transitions entre les différents états : par exemple, lorsque le client a terminé sa commande, il se dirige vers la caisse pour payer ; lorsqu'il a payé, il prend sa commande et s'en va.

2.3 Éléments de regroupement

2.3.1 Paquetages

Les *paquetages* répondent à un besoin de regroupement de concepts voisins ou d'objets participant naturellement à des objectifs communs.

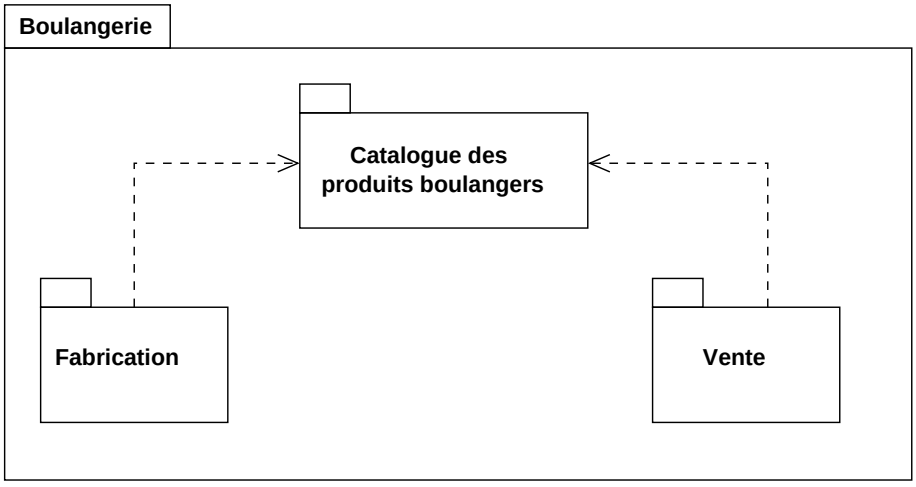

FIG. 2.12 –: *Paquetages et espaces de noms*

Graphiquement, un paquetage est représenté par un rectangle avec un onglet de classement. Le nom du paquetage est indiqué sur l'onglet ou bien dans le rectangle.

Un paquetage peut contenir d'autres paquetages, ce qui définit une organisation hiérarchique.
Il est important de noter que les paquetages sont des éléments de rangement et d'organisation au niveau d'un projet ou de plusieurs projets, mais qu'ils n'imposent absolument aucune directive concernant la réalisation d'un système.
Par contre, l'organisation des paquetages entre eux permet de faire apparaître et de minimiser autant que possible les relations de *dépendance* entre des concepts (paragraphe 2.6.7).

La figure 2.12 montre deux domaines d'activité indépendants : la fabrication et la vente, manipulant l'un et l'autre des produits rangés dans un catalogue commun. Clairement, chacun des trois paquetages que l'on en déduit possède sa cohérence interne. La spécification des tâches de fabrication ne dépend pas de celle des tâches de vente, ni inversement. Par contre, l'une et l'autre dépendent de la spécification des produits.

En permettant le rangement de notions d'après des critères de proximité conceptuelle, les paquetages procurent automatiquement des *espaces de noms*, c'est-à-dire des espaces où les mots ont une signification unique, celle du domaine correspondant[1].
Par exemple, dans le paquetage des produits boulangers, le terme de « ficelle » n'est pas ambigu : son sens est celui du domaine de la boulangerie. Au contraire, dans la vie courante, le sens des mots dépend souvent du contexte ou bien de précisions explicites. On s'en aperçoit déjà sans quitter le cadre de la boulangerie. En effet, à l'intérieur des paquetages « fabrication » et « vente », le terme de ficelle peut aussi désigner la ficelle fermant un sac de farine, ou bien celle d'une boîte à gâteaux, ou d'autres acceptions encore. Mais il faut préciser alors que ces sens proviennent de paquetages de sens externes et ne sont pas propres au domaine de la boulangerie.
On peut répéter la démonstration à propos de bien d'autres termes comme « croissant », ou bien « flûte » et « baguette » en les comparant par exemple avec le domaine de la musique d'orchestre.

Nous utilisons les paquetages pour décrire des dépendances entre des espaces de noms.

1. On trouvera un chapitre entier décrivant les espaces de noms – et leurs applications intéressantes – dans notre livre *XML Schema et XML Infoset pour les Services Web*, Éditions CÉPADUÈS, 2006.

2.4 Éléments d'annotation

2.4.1 Notes

UML est un langage graphique. Le texte n'a de sens que par rapport aux éléments UML qui le contiennent ou l'entourent. Autrement dit, le sens du texte est compris par un lecteur en que nom de classe, d'attribut, de message, d'association ... et non pas en tant que tel.
Lorsque l'on souhaite écrire du texte en langage naturel (par exemple des commentaires à propos d'un diagramme ou d'un élément), on doit placer ce texte dans une *note*, afin qu'il ne soit pas interprété par rapport aux éléments du diagramme.

Graphiquement, une note est représentée par un rectangle, dont l'angle supérieur droit est écorné, contenant du texte ou des images.
Une note s'appliquant globalement à un diagramme est placée dans le diagramme. Si elle s'applique à un élément UML spécifique, elle est rattachée à cet élément par un trait discontinu.

Les figures 2.7, 2.14, 2.25, 2.26 et 2.30 montrent des exemples de notes.

2.5 Éléments d'extension

2.5.1 Stéréotypes

Concept

Les *stéréotypes* sont des mots-clés définissant une information additionnelle de type s'appliquant à un concept désigné par un élément de notation UML standard.
Ils peuvent qualifier divers éléments de notation, comme les classes, les associations, les dépendances, les paquetages ...
Ils enrichissent la signification d'un concept de base, tout en limitant à un domaine métier donné le champ de signification supplémentaire. En d'autres termes, il permettent de qualifier des éléments en ajoutant une information métier spécifique (souvent habituelle et répétitive) valide uniquement dans un projet ou dans un ensemble de projets.

La figure 2.13 montre l'utilisation des stéréotypes dans un modèle d'analyse. Dans cet exemple, nous avons besoin (pour des utilisations diverses : étalage, étiquettage des prix ...) d'identifier différents types de pains dans un catalogue. Une solution consisterait à utiliser la généralisation (paragraphe 2.6.8), afin de spécifier une hiérarchie d'héritage ; à ce stade, cette solution n'est pas la bonne pour plusieurs raisons : manque d'extensibilité, complexité inutile ...
En phase d'analyse logique, les stéréotypes permettent d'ajouter simplement un type supplémentaire à des éléments UML, sans alourdir les diagrammes ni anticiper inutilement sur les solutions possibles au stade de la conception.

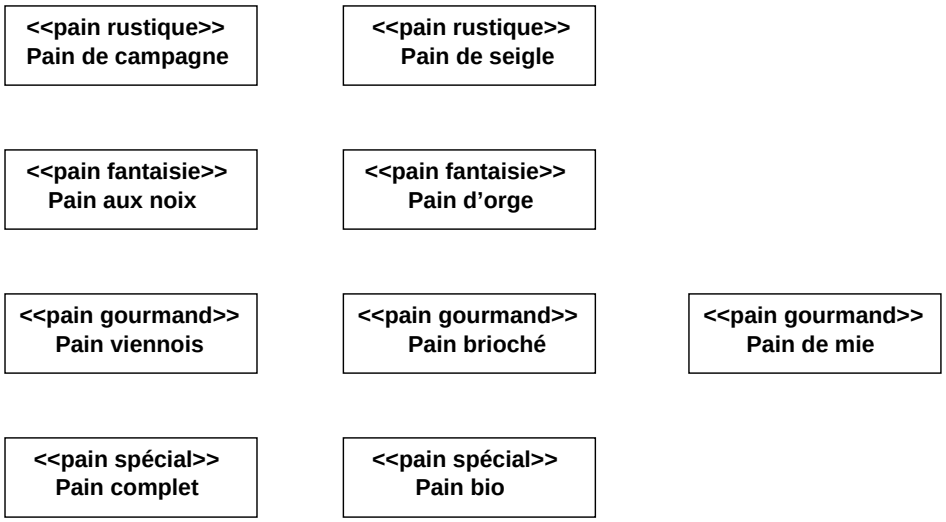

FIG. 2.13 –: *Stéréotypes*

Notation

Les *stéréotypes* sont représentés par une chaîne de caractères comprise entre guillemets, comme `<<pain rustique>>` sur la figure 2.13.
La spécification UML définit une liste de stéréotypes standard que l'on peut utiliser directement sans devoir documenter au préalable leur signification. Des exemples comme `<<interface>>` (paragraphe 2.1.4) et `<<acteur>>` (paragraphe 2.1.7) font partie de cette liste pré-définie.
La plupart du temps, puisque l'on modélise dans un domaine où les mots

ont déjà un sens précis, il suffit de les utiliser avec la syntaxe des stéréotypes ainsi que nous le montre la figure 2.13.

Nous utilisons abondamment les stéréotypes dans notre modèle, principalement afin de qualifier les classes et les relations d'association. Nous avons pour cela deux bonnes raisons : nous sommes dans un domaine spécifique – celui du psychisme – et nous travaillons sur un modèle d'analyse logique dans un objectif de compréhension.

2.5.2 Icônes

Un stéréotype fréquemment utilisé a souvent une *icône* associée. Cela permet d'enrichir le langage UML avec de nouveaux éléments de notation, puisque le simple dessin de l'icône représente alors l'élément de notation stéréotypé.
Il existe des icônes standard associées à des stéréotypes standard :

- la figure 2.6 montre l'icône représentant une interface fournie ;
- la figure 2.8 montre l'icône du bonhomme en fil de fer (*stick-man*) pour stéréotyper un acteur ;
- les icônes de la figure 2.14 sont elles aussi des icônes standard ;
- ...

Dans un domaine donné, pour améliorer la clarté des modèles, les responsables de la modélisation sont libres de définir des icônes spécifiques, de même que des stéréotypes spécifiques, ou bien de décider d'une utilisation spécifique des icônes disponibles.

Par exemple, dans le paragraphe 2.1.7, nous avons décidé de choisir l'icône du stick-man pour stéréotyper le concept de « sujet » dans les diagrammes de classes de notre modèle.

Icônes du modèle MVC

Le *pattern architectural MVC (Modèle, Vue, Contrôleur)* définit trois rôles que l'on trouve de manière répétitive dans l'analyse architecturale d'un grand nombre de systèmes[2].
Il joue un rôle important dans les phases initiales de l'analyse des systèmes. Il est très fréquemment décrit et très largement documenté.

2. Un *pattern* est un schéma qui se répète à l'identique, de telle manière que la connaissance d'un pattern permet d'en spécifier facilement de nouveaux.

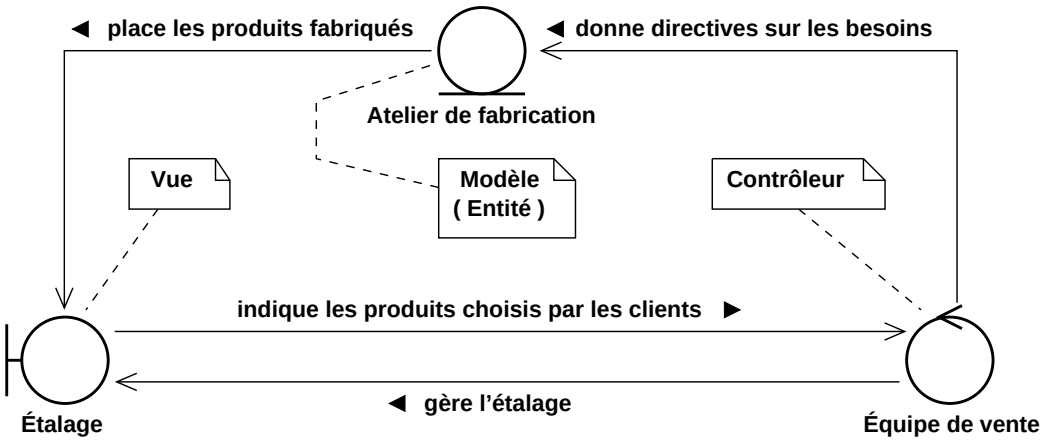

FIG. 2.14 –: *Icônes du modèle* MVC

Nous fournissons ici un bref panaroma du pattern MVC, afin de présenter les trois icônes associées que nous utilisons à de nombreuses reprises dans notre modèle. La figure 2.14 montre un exemple d'utilisation du pattern MVC. Les rôles assignés aux trois composants sont les suivants :

- *l'étalage représente la vue*. La vue est l'interface du système vers le monde extérieur : c'est en regardant l'étalage que le client fait ses choix. La vue peut être un composant passif, son rôle principal étant l'affichage et la réception des commandes. La vue est interchangeable selon l'air du temps ;
- *l'atelier de fabrication représente le modèle* (on dit aussi *entité*). Il détient l'essence du domaine métier (ici, le métier de la boulangerie). Le modèle est un élément pérenne. Il possède le savoir-faire et l'enrichit. Le modèle place lui-même dans l'étalage les produits fabriqués ;
- *l'équipe de vente représente le contrôleur*. Les classes du contrôleur sont actives : elles travaillent, manipulent et transmettent les informations, gèrent les échanges. Le contrôleur réagit selon les informations de la vue, et maintient l'état de la vue. Il transmet au modèle les informations à propos du monde extérieur, par exemple les besoins en fabrication.

Dans notre modèle, nous tirons profit des trois icônes du pattern MVC pour stéréotyper les catégories R.S.I. du modèle de Lacan.
L'icône d'entité représente le Réel, ce qui est stable. L'icône du contrôleur repré-

sente le Symbolique, ce qui travaille. Enfin, l'icône de la vue représente l'Imaginaire, ce qui est volatile.

2.5.3 Contraintes

Les *contraintes* expriment des conditions devant être vérifiées dans le modèle, ou bien apportent des informations formelles.
Elles s'écrivent placées entre accolades.

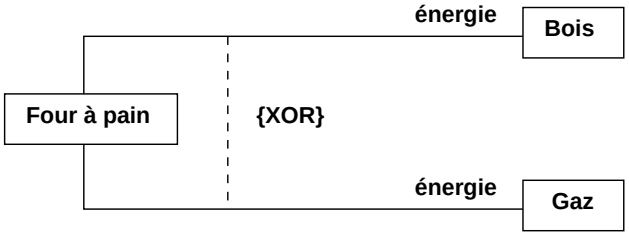

FIG. 2.15 –: *Contrainte du « ou exclusif »*

La figure 2.15 spécifie que le four à pain peut être un four à bois ou bien un four à gaz, mais l'un ou l'autre, et pas les deux en même temps. La contrainte `{XOR}` (ou exclusif) s'applique à deux associations dont une seule peut être valide dans le modèle. Les deux associations sont reliées par une ligne discontinue à côté de laquelle est placée la contrainte.

Sur la figure 2.26 (paragraphe 2.6.8), la contrainte `{incomplet}` indique que l'arbre d'héritage montré sur le diagramme n'est pas exhaustif : dans la réalité il existe sûrement d'autres éléments enfants que l'on ignore ici, soit parce qu'on ne les connaît pas tous, soit parce que l'on ne veut pas alourdir le diagramme.
Par ailleurs, UML spécifie de nombreuses contraintes standard.

Dans notre modèle nous utilisons `{incomplet}` *, de même que des contraintes appliquées à des couples d'associations. Dans ce dernier cas, nous complétons la ligne discontinue par des flèches dans les deux sens.*

2.5.4 Propriétés étiquetées

Ce sont des associations `nom=valeur` placées entre accolades, et qualifiant un nom d'élément, par exemple: `{version=0.9}` ou `{langage=java}`.

2.6 Relations

Les relations spécifient comment des éléments UML sont reliés entre eux. Il existe quatre types de relations :

- les associations ;
- les dépendances ;
- les généralisations ;
- les réalisations.

2.6.1 Association

Les *associations* permettent les interactions entre objets, grâce à la circulation des messages dans un système – c'est-à-dire en spécifiant quels objets [instances de quelles classes] sont reliés à quels autres objets [instances de quelles autres classes].
On parle d'associations dans un diagramme de classes, et de *liens* dans un diagramme d'objets ou de communication. Les messages circulent sur les liens.

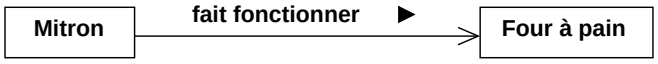

FIG. 2.16 –: *Association*

Graphiquement, une association est représentée par un trait plein reliant deux classes.
La figure 2.16 montre que le mitron actionne le four à pain. Cette association est nommée et navigable (paragraphe 2.6.2).

Une association exprime un lien durable entre des objets : les objets de cessent pas de se connaître et peuvent en permanence s'échanger des messages. Si un lien est épisodique – par exemple lorsqu'un un objet en crée un autre (il faut bien qu'à ce moment là il y ait un lien entre eux), puis ensuite ils vivent chacun leur vie de leur côté – il n'y a pas association mais dépendance (paragraphe 2.6.7).

2.6.2 Décorations des associations

Nom d'association

Une association peut avoir un nom afin de préciser la nature de la relation qu'elle décrit.
Le sens de compréhension d'un *nom d'association* n'étant pas forcément le même que le sens naturel de lecture, il est d'usage de placer, à côté du nom d'association, un triangle plein en forme de flèche de direction, indiquant dans quel sens – c'est-à-dire de quelle classe vers quelle autre classe – fonctionne la relation.

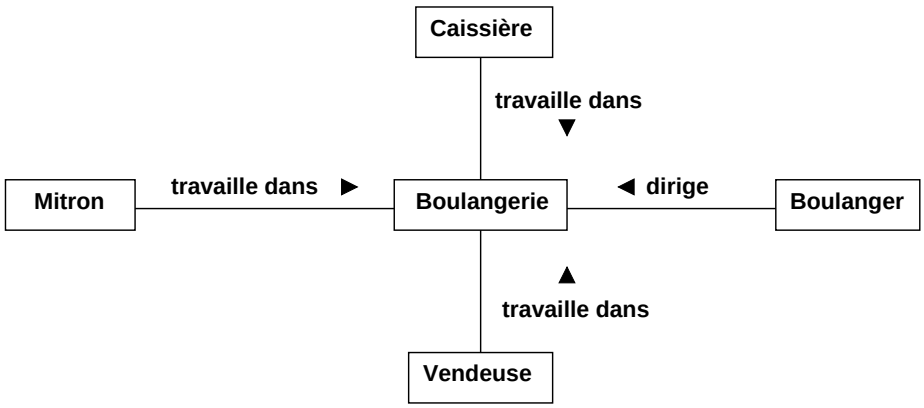

FIG. 2.17 –: *Noms d'association et sens de lecture*

La figure 2.17 montre quatre personnes travaillant dans une boulangerie, ainsi que le type d'association (complété par le sens de lecture) de chacun avec son lieu de travail.

Dans notre modèle, toutes les associations placées dans les diagrammes de classes possèdent un nom et un sens de lecture.

Navigabilité

La présence d'une association entre classes n'implique pas que les objets situés des deux côtés de l'association se connaissent réciproquement. Lorsque la possibilité d'action ne fonctionne que dans un sens, on le précise en dessinant une flèche en bout d'association. Cette décoration

est appelée *navigabilité* : elle indique dans quel sens naviguent les messages sur le lien entre objets. Lorsqu'une association est navigable dans les deux sens, on ne met pas de flèche ni d'un côté ni de l'autre, ou bien – mais plus rarement – on peut aussi placer des flèches des deux côtés de l'association.
La figure 2.18 montre deux associations navigables.

Dans notre modèle, la navigabilité est indiquée sur la plupart des associations.

Multiplicité

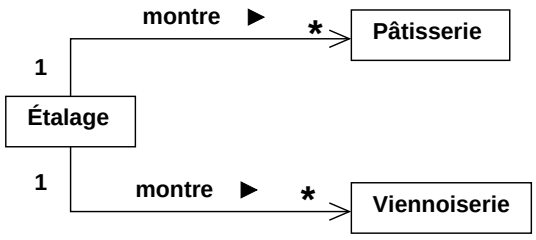

FIG. 2.18 –: *Multiplicité*

La figure 2.18 montre la *multiplicité*, c'est-à-dire le nombre d'objets prenant part à l'association de chaque côté. La multiplicité peut être un nombre exact, un intervalle entre deux nombres, ou bien le caractère « * » indiquant un nombre indéfini.

Dans notre modèle, nous n'indiquons la multiplicité que lorsqu'il nous a paru indispensable de la préciser.

D'autres décorations – que nous n'utilisons pas dans notre modèle – peuvent enrichir des associations : les noms de rôles et la visibilité.

2.6.3 Négation

En UML, lorsque des objets instances de deux classes ne se connaissent pas, il n'y a simplement pas d'association reliant les classes correspondantes : l'inexistence d'une association signifie en même temps sa négation. Cette approche est apparemment suffisante dans les domaines de l'ingénierie.

Dans notre modèle – à la suite de Lacan – le concept de négation ne peut pas s'exprimer par la simple inexistence, pour deux raisons :

- *l'absence ne signifie pas la négation : le négatif est du même ordre que le positif ; simplement, il est porteur d'un attribut spécifique exprimant la négation ;*
- *pour que quelque chose puisse être nié, il faut qu'il existe au préalable la possibilité d'une représentation symbolique. C'est à cette représentation que s'applique la négation.*

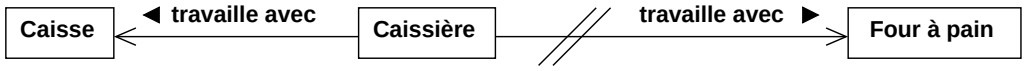

FIG. 2.19 –: *Négation*

La figure 2.19 montre comment nous exprimons le concept de négation d'une manière convenable, c'est-à-dire en appliquant la négation à une représentation.
Pour nier une association, d'abord nous la dessinons normalement, puis nous la barrons par deux traits obliques. Plus généralement, nous appliquons cette expression de la négation à tous les types de relations.

Cette notation n'est pas une notation UML standard.

2.6.4 Classe d'association

Une *classe d'association* permet d'ajouter des attributs et des opérations à l'association elle-même.

Nous savons que le contenu d'un contrat de travail n'est propre, ni à l'employé, ni à l'entreprise, mais qualifie l'association entre l'un et l'autre. L'exemple de la figure 2.20 montre une classe d'association permettant d'exprimer le contrat de travail en tant que classe à part entière du modèle, en lui adjoignant le concept d'être aussi une association entre deux autres classes du modèle.
Une classe d'association est reliée à l'association qu'elle représente par une ligne discontinue.

Nous utilisons plusieurs classes d'association dans notre modèle.

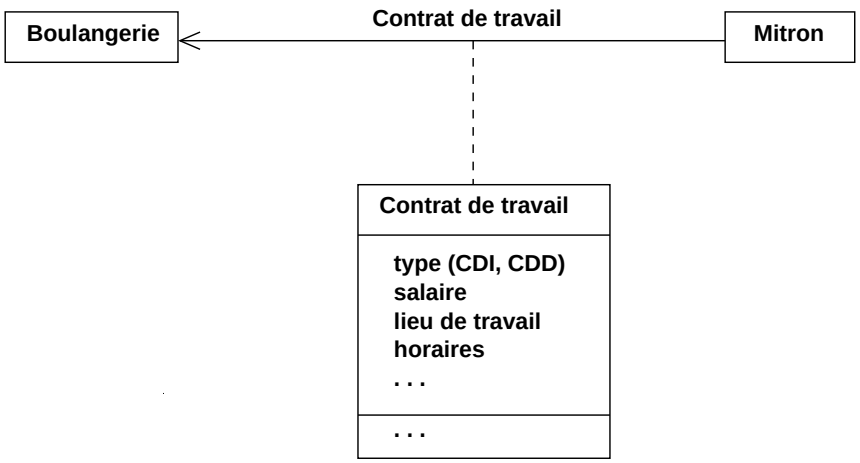

FIG. 2.20 –: *Classe d'association*

2.6.5 Agrégation

Une *agrégation* est une association dans laquelle les objets de l'une des classes jouent un rôle de conteneur par rapport aux objets d'autres classes. L'association spécifiée par une agrégation n'est *a priori* valide que pendant une période partielle, plus courte que la durée de vie des objets des deux côtés.
L'agrégation se représente comme une association, en ajoutant un losange vide du côté du conteneur.

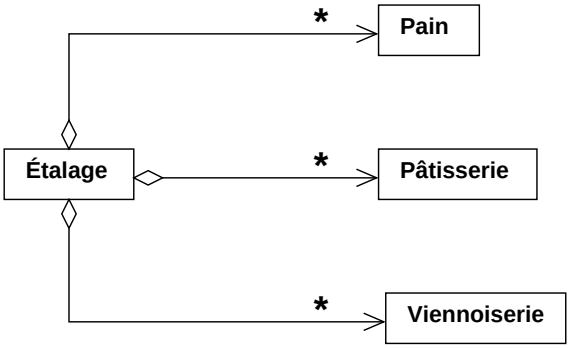

FIG. 2.21 –: *Agrégation*

La figure 2.21 montre un exemple d'agrégation : les produits boulangers sont disposés pendant un certain temps dans un étalage. Toutefois, ce n'est pas le fait d'être dans l'étalage qui en fait des produits boulan-

gers : ils existent en tant que tels avant qu'on les y place, et ce sont encore des produits boulangers quand on les en enlève.

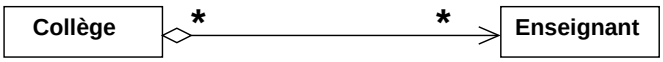

FIG. 2.22 –: *Autre exemple d'agrégation*

La figure 2.22 montre un autre exemple d'agrégation. Un collège est en effet un agrégat d'enseignants : l'équipe des professeurs. Cependant, chaque enseignant était peut-être déjà professeur avant d'être en poste dans un collège donné ; il reste professeur s'il change de collège ; et il peut aussi enseigner dans différents collèges à la fois.
C'est cet ensemble de caractéristiques qu'exprime l'agrégation.

2.6.6 Composition

La *composition* exprime une liaison plus forte que l'agrégation. En particulier, les durées de vie du conteneur et des composants sont obligatoirement conjointes du début à la fin.
La composition se représente comme l'agrégation, mais avec un losange plein.

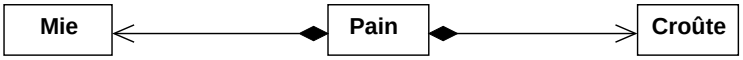

FIG. 2.23 –: *Composition*

La figure 2.23 montre un exemple de composition : un pain est composé de mie et de croûte. Des instances de mie ou de croûte font partie d'un certain pain, jamais d'un autre pain ; elles ont été crées en même temps que le pain au moment de la fabrication ; c'est le fait de faire partie d'un pain qui en fait de la mie et de la croûte.

2.6.7 Dépendance

Il y a *dépendance* lorsqu'un élément – l'élément dépendant (classe, interface en tant que type de haut niveau, composant, paquetage) – peut

être impacté par des changements éventuels dans la structure ou le comportement d'un autre élément auquel il est lié épisodiquement.

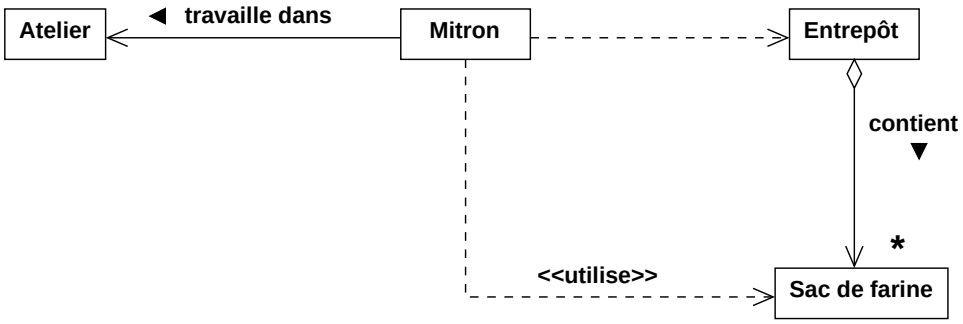

FIG. 2.24 –: *Dépendance*

La dépendance est représentée graphiquement par un trait interrompu et une flèche ouverte dirigée de l'élément dépendant vers celui dont il dépend.
La figure 2.24 montre deux dépendances. Le mitron, employé dans un atelier de boulangerie, est dépendant de la structure des sacs de farine qu'il utilise, parce qu'il doit les transporter, les ouvrir ... mais ce n'est pas l'essentiel de son métier. De même, pour faire correctement son propre travail, il dépend de l'approvisionnement en sacs de farine dans un entrepôt.
Il est souvent utile de stéréotyper une dépendance pour documenter sa nature : création, utilisation ...
La dépendance s'applique aussi entre paquetages (paragraphe 2.3.1) : si un élément d'un paquetage A dépend d'au moins un élément du paquetage B, alors le paquetage A est dépendant du paquetage B.

2.6.8 Généralisation

La *généralisation* permet de regrouper dans un type générique (le type de base) des caractéristiques communes à un ensemble d'éléments plus spécifiques (les types dérivés).
Les types dérivés héritent de l'ensemble des caractéristiques du type de base, et peuvent les spécialiser ou bien les compléter par des caractéristiques propres.

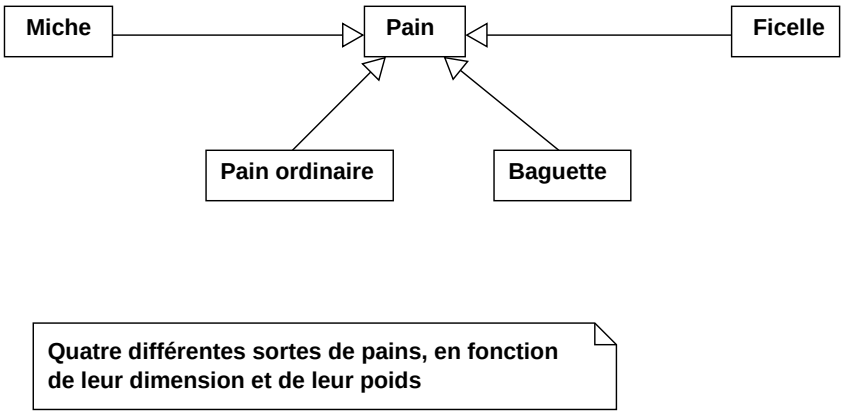

FIG. 2.25 –: *Généralisation*

Graphiquement, la généralisation est représentée par un trait plein avec une flèche fermée dirigée vers la classe de base.
La figure 2.25 montre un exemple de généralisation. Nous l'employons dans le langage courant, mais il est également valide pour les spécifications techniques et légales appliquées par les boulangers dans leurs processus de fabrication.
Cet exemple illustre clairement les deux règles de base de la généralisation :

- *la relation « est-un »*. On sait que chacune des sortes de pain est simplement un pain : la notion de pain regroupe un grand nombre de caractéristiques communes (composition, fabrication, aspect ...). Par contre, on emploie les noms spécifiques dans le langage courant lorsque l'on achète son pain, ou bien, d'un point de vue technique, lorsque l'on veut préciser le poids ou la forme ;
- *le principe de substitution*[3] : une généralisation est conceptuellement correcte si un type dérivé peut effectivement jouer le même rôle qu'un type de base en se substituant à lui. Bien évidemment, on peut employer une miche, un pain ordinaire, une baguette ou une ficelle pour un usage générique de pain. De même, dans la conversation, on laisse ouverte la porte de la substitution quand on emploie le terme de pain.

On peut dessiner plusieurs relations de généralisation allant vers un même type de base (figure 2.25), ou bien utiliser une notation « en rateau » (figure 2.26), souvent plus lisible.

3. *Liskov Substitution Principle (LSP)* énoncé en 1987 par Barbara Liskov.

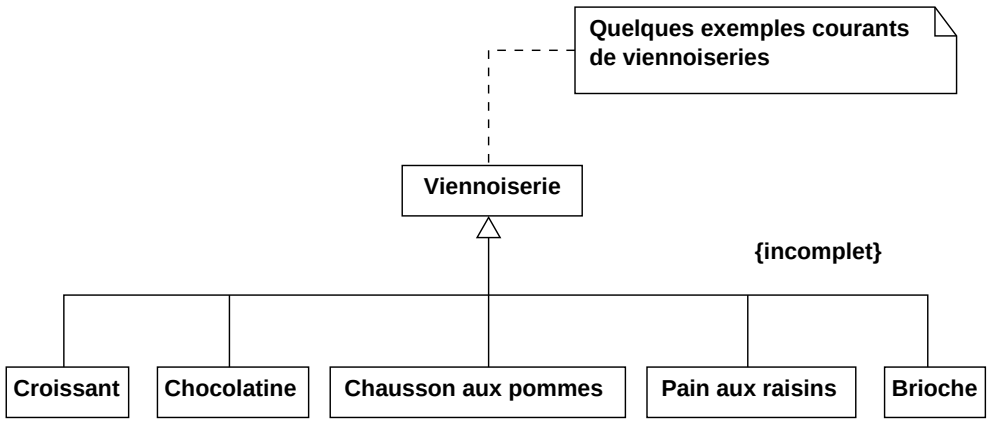

FIG. 2.26 –: *Généralisation – notation « en rateau »*

La figure 2.26 montre une généralisation ouverte (contrainte `{incomplet}`) : on ne prétend pas avoir énuméré ici toutes les sortes de viennoiseries.

2.6.9 Réalisation

La *réalisation* est une relation reliant un type concret à un type purement abstrait dont il assure la réalisation.
Elle est représentée par un trait interrompu se terminant par une flèche de généralisation, et allant de l'élément concret vers l'élément abstrait qu'il réalise.

Les figures 2.4 et 2.5 (paragraphe 2.1.4) montrent des réalisations reliant des types concrets (boulanger, mitron) à des spécifications d'interfaces décrivant leurs savoir-faire.
La figure 2.10 (paragraphe 2.1.9) montre une collaboration réalisant plusieurs cas d'utilisation.

2.6.10 Relations entre cas d'utilisation

Généralisation

Le concept de généralisation (paragraphe 2.6.8) s'applique facilement aux cas d'utilisation.
La figure 2.27 montre trois spécialisations de la fonction générique d'achat d'un produit dans une boulangerie.

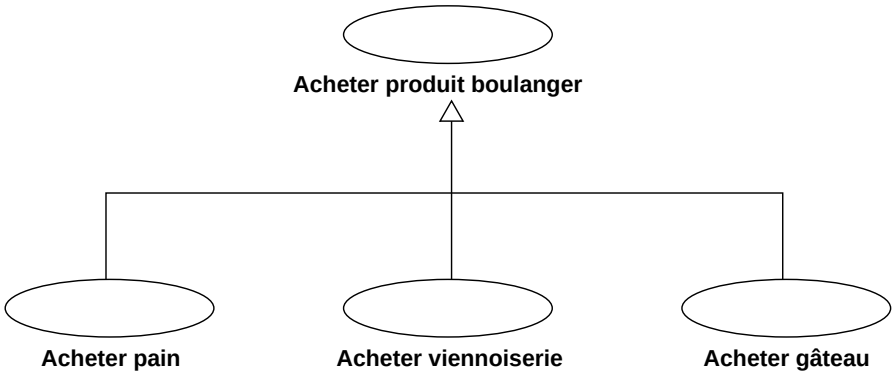

FIG. 2.27 –: *Généralisation entre cas d'utilisation*

Dépendances

Des cas d'utilisation peuvent aussi invoquer d'autres cas d'utilisation. La figure 2.28 montre les deux types de dépendances que l'on peut alors modéliser.

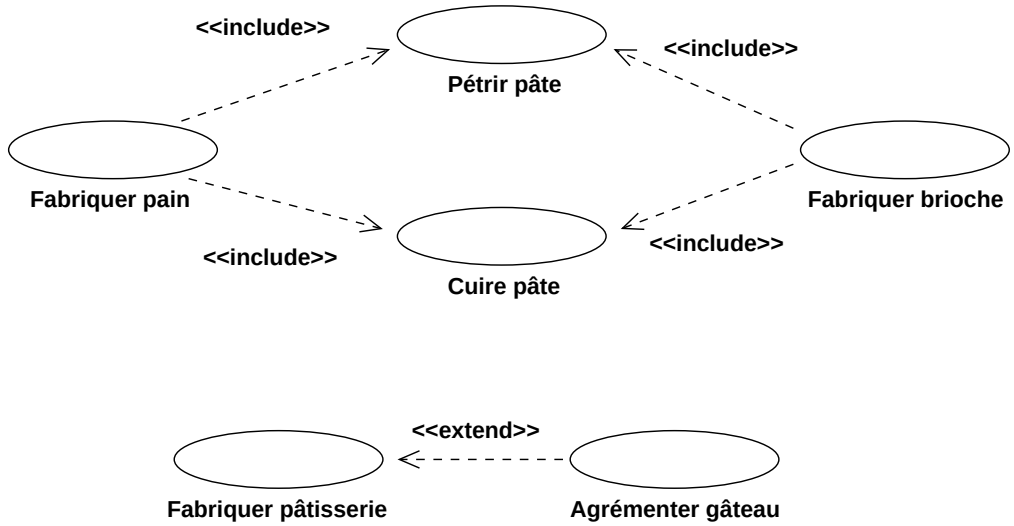

FIG. 2.28 –: *Relations d'inclusion et d'extension*

Le premier cas est celui d'un sorte de fonction de service appelée – obligatoirement – par d'autres fonctions principales pendant leur déroulement. Cela permet de factoriser la description de fonctionnalités, mais au final elles sont simplement incluses dans l'ensemble des fonctionnalités de chaque fonction appelante.

Cette relation se représente par une dépendance portant le stéréotype <<include>>, et allant du cas d'utilisation appelant vers le cas d'utilisation appelé.

Le second cas est celui d'une fonction optionnelle, pouvant ou non prendre place dans le déroulement d'une fonction principale. Cette fonction dépend en quelque sorte de conditions d'extension décidées au niveau de la fonction principale.
Cette relation se représente par une dépendance portant le stéréotype <<extend>>, et allant du cas d'utilisation appelé vers le cas d'utilisation appelant.

2.7 Diagrammes UML

Les diagrammes apportent une information topologique décrivant l'organisation et l'interaction des éléments.
Il existe treize diagrammes standard en UML 2 (il y en avait neuf en UML 1).

Dans notre modèle, nous utilisons de manière largement majoritaire les diagrammes de classes représentant la structure.
Nous utilisons aussi quelques diagrammes de séquence, diagrammes de cas d'utilisation et diagrammes de paquetages.

2.7.1 Diagramme de cas d'utilisation

Un *diagramme de cas d'utilisation* spécifie les fonctions d'un système du point de vue des utilisateurs. Ce diagramme est doublement nécessaire : au début d'un projet pour comprendre la nature du produit à construire, puis durant le développement et lors des sessions de validation pour mettre au point les jeux de tests et les dérouler.
La figure 2.9 montre un exemple simple de diagramme de cas d'utilisation. Le système est généralement représenté par un rectangle. Les acteurs font fonctionner le système en activant les cas d'utilisation.

2.7.2 Diagramme de classes

Un *diagramme de classes* montre la structure statique d'un système en termes de classes et de relations. En principe, dans un projet, ce dia-

gramme ne devrait pas être produit en premier. Par contre, il joue ensuite un rôle crucial dès l'étape de passage à l'objet et dans les étapes suivantes.

En analyse logique de systèmes, ce diagramme joue un rôle fondamental, parce qu'il montre la structure des concepts et la façon dont ils sont reliés entre eux.
La plupart des figures de ce chapitre montrant des classes et des relations peuvent être considérées comme des exemples de fragments de diagrammes de classes.

2.7.3 Diagramme d'objets

Un *diagramme d'objet* décrit une situation particulière d'un système. Il montre la structure statique à un moment donné, c'est-à-dire les objets et leurs liens.

2.7.4 Diagramme de paquetages

Ce diagramme est nouveau en UML 2. Il montre les paquetages (paragraphe 2.3.1) et leurs dépendances (paragraphe 2.6.7).

2.7.5 Diagramme de structure composite

Ce diagramme est nouveau en UML 2. Il montre la structure interne d'une classe composite, et définit les connexions avec le monde extérieur sous la forme de ports de communication spécifiés par des interfaces.
Il peut être comparé à la partie statique d'une collaboration.

2.7.6 Diagramme de séquence

Un *diagramme de séquence* fournit une représentation temporelle des interactions entre des objets (on parle aussi de rôles à la place d'objets).
La formalisation des messages échangés est une étape importante du passage à l'objet, puisqu'elle permet de spécifier les responsabilités et les interfaces des objets – c'est-à-dire ce que d'autres objets peuvent leur demander et quels types de réponse ils s'engagent à apporter aux demandes qu'on leur addresse.

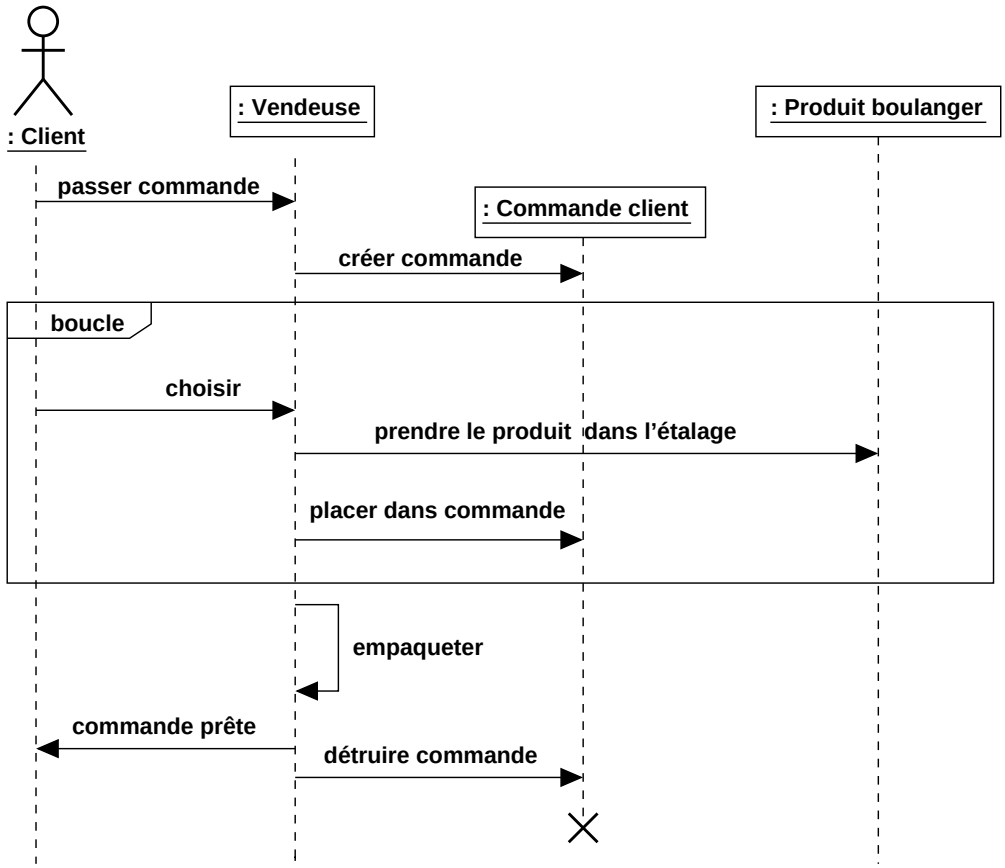

FIG. 2.29 –: *Diagramme de séquence*

La figure 2.29 montre les objets (rôles) prenant part à une commande d'un ou plusieurs produits dans une boulangerie, ainsi que la séquence chronologique de leurs interactions.
Le temps se déroule verticalement du haut vers le bas du diagramme. Une ligne discontinue est associée à chaque objet, c'est sa *ligne de vie*. Un message part horizontalement de la ligne de vie de l'objet émetteur et rejoint la ligne de vie de l'objet récepteur.
Un objet peut s'adresser un message à lui-même : cela correspond à une activité qu'il accomplit à de ce stade de sa ligne de vie.
Les noms des objets existant au début de la séquence de temps sont disposés côte à côte en haut du diagramme.
Lorsqu'un objet est créé, sa ligne de vie et son nom commencent au moment de sa création. Lorsqu'un objet est détruit pendant la séquence de temps, sa ligne de vie s'arrête à ce moment là. Parfois, la durée de

vie complète d'un objet est inférieure à la durée d'un diagramme de séquence : sur l'exemple de la figure 2.29, l'objet `commande client` est un objet temporaire destiné à mémoriser (en mémoire, sur papier ...) les articles commandés par un client, pendant que ce client est servi. Un nouvel objet est créé pour un nouveau client. Il est détruit quand le client quitte le magasin avec sa commande.

Le diagramme de la figure 2.29 contient un fragment de diagramme nommé `boucle` montrant une séquence d'opérations répétitives.

2.7.7 Diagramme de communication

Un *diagramme de communication* apporte la même information qu'un diagramme de séquence, mais en la présentant sous forme spatiale. Il correspond au diagramme de collaboration en UML 1.
Les diagrammes de séquence et de communication sont aussi appelés *diagrammes d'interaction*. Ils sont strictement équivalents du point de vue sémantique.

2.7.8 Diagramme de *timing*

Ce diagramme est nouveau en UML 2. C'est une sorte de diagramme de séquence représenté de manière plus formelle, dans lequel les contraintes temporelles sont explicitement mesurées en unités de temps.
Dans un diagramme de *timing*, le temps s'écoule horizontalement le long d'une échelle chronologique graduée.

2.7.9 Diagramme d'états-transitions

Un *diagramme d'états-transitions* est utilisé pour décrire des automates, c'est-à-dire chaque fois que l'on peut modéliser un comportement (scénario d'un cas d'utilisation, cycle de vie d'un objet, d'un système, processus métier ...) sous la forme d'une succession d'états (paragraphe 2.2.2) et de transitions (paragraphe 2.2.3).

La figure 2.30 montre le cycle de vie de l'objet `commande client` utilisé au paragraphe 2.7.6, entre le moment de sa création et celui de sa destruction.

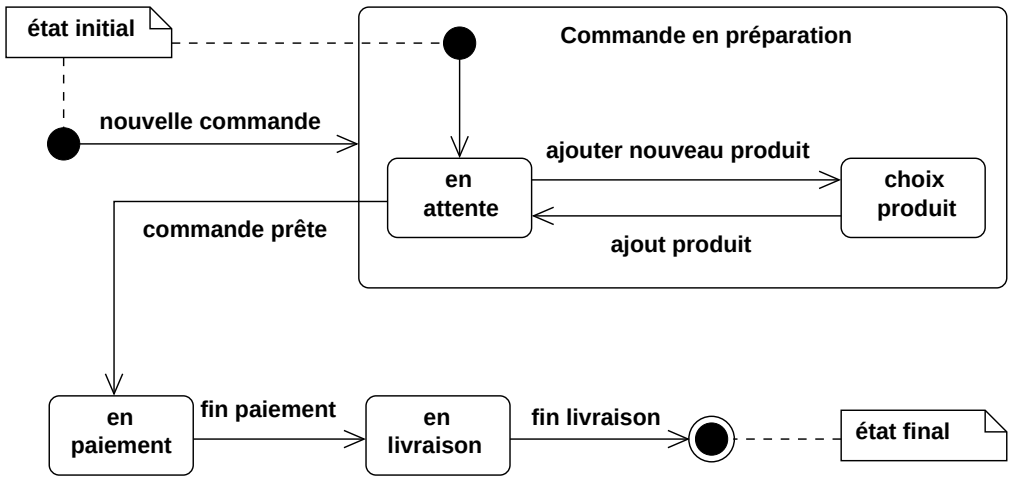

FIG. 2.30 –: *Diagramme d'états-transitions*

L'étape de préparation de la commande est elle-même un état composite comportant son propre état initial.

2.7.10 Diagramme d'activités

Un *diagramme d'activités* décrit un processus métier – par exemple un scénario de cas d'utilisation, ou bien le déroulement des traitements dans une opération invoquée sur un objet – sous la forme d'un organigramme. Ce diagramme apporte une information essentiellement procédurale sur la succession des activités et sur les flots d'information transmis.

Les diagrammes d'activités sont utilisés aussi bien en phase initiale pour comprendre des cas d'utilisation, qu'en phase de conception détaillée pour permettre à des experts-métier de préciser des processus complexes.

Le diagramme de la figure 2.31 décrit – à un niveau simple de vulgarisation – le processus de fabrication du pain.
Du point de vue de la notation, une activité est représentée par un rectangle terminé de chaque côté par des demi-cercles. L'exemple 2.31 montre aussi un losange de branchement conditionnel. Les états de début et de fin sont représentés comme dans les diagrammes d'états-transitions.

Un diagramme d'activité peut être divisé en travées (en anglais, *swimlanes*, par analogie avec les lignes d'eau dans une piscine) lorsque l'on

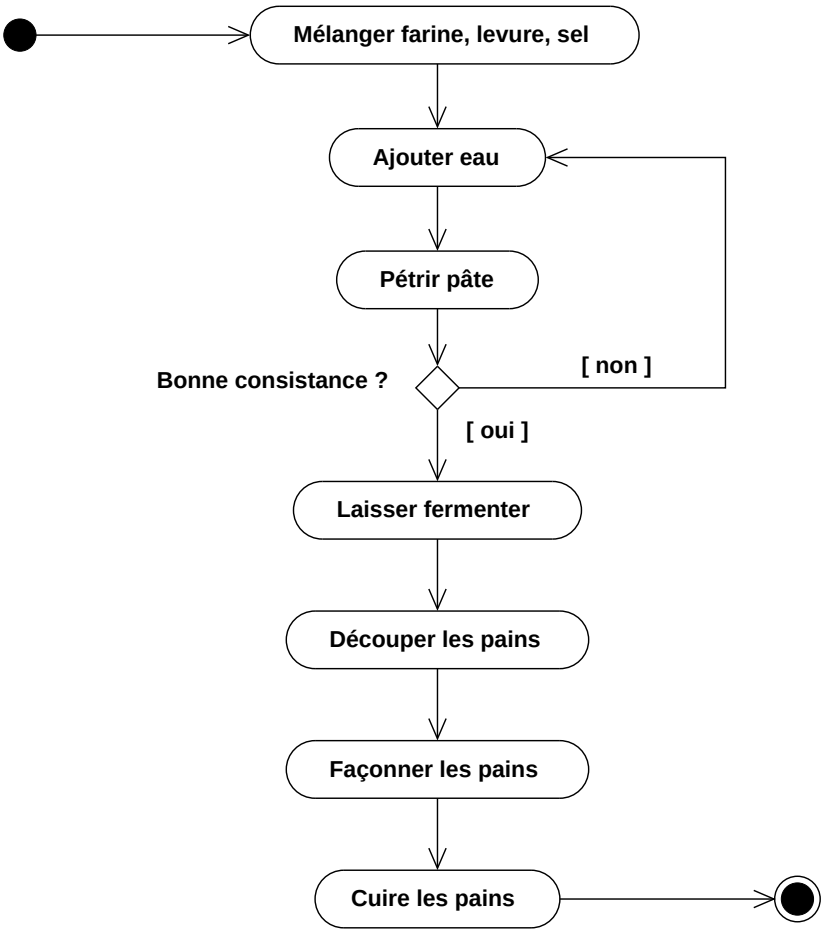

FIG. 2.31 –: *Diagramme d'activités*

veut représenter en parallèle les tâches concurrentes de différents intervenants, ainsi que les flots de données échangés et les contraintes entre les actions des uns et des autres.

2.7.11 Diagramme de vue d'ensemble des interactions

Ce diagramme est nouveau en UML 2. C'est un diagramme d'activités dans lequel les nœuds sont des diagrammes d'interaction – en général, des fragments de diagrammes de séquence.
Ce diagramme fournit une sorte de synthèse entre une approche procédurale à un haut niveau, et une approche objet au niveau plus détaillé des interactions.

2.7.12 Diagramme de composants

Un *diagramme de composants* est un diagramme architectural.
Il décrit le découpage d'un système en composants spécifiés par leurs interfaces, leur organisation, et leurs relations comme pour les classes dans un diagramme de classes.

2.7.13 Diagramme de déploiement

Un *diagramme de déploiement* est propre aux systèmes informatiques. Il définit l'installation d'un système, ainsi que la configuration des ressources matérielles installées dans les nœuds (paragraphe 2.1.10).

2.8 Résumé de la notation UML

Certains éléments UML présentés dans cette partie I sont très largement utilisés en partie III. D'autres ne le sont pas du tout.
Pour lire la suite de ce livre, il faut porter une attention particulière aux éléments suivants :

- classes (2.1.1) ;
- associations (2.6.1, 2.6.2) ;
- interfaces et types (2.1.4, 2.1.5) ;
- stéréotypes et icônes (2.5.1, 2.5.2) ;
- agrégation et composition (2.6.5, 2.6.6) ;
- généralisation (2.6.8) ;
- négation (2.6.3) ;
- notes (2.4.1) ;
- classes d'association (2.6.4) ;
- dépendance (2.6.7) ;
- réalisation (2.6.9) ;
- messages (2.2.1) ;
- contraintes (2.5.3) ;
- acteurs et cas d'utilisation (2.1.7, 2.1.8, 2.6.10) ;
- paquetages (2.3.1).