Langage UML : développement de logiciel et modélisation visuelle

par **Patrick GIROUX**
Ingénieur consultant, EADS Defence and Security Systems
Maître de conférence associé, université de Rouen

Le langage UML (Unified Modeling Language) est un langage graphique de modélisation initialement conçu pour représenter, spécifier, concevoir et documenter les artefacts de systèmes logiciels. Adopté par l'Object Management Group (OMG) en tant que standard, il est devenu une référence incontournable dans le domaine du génie logiciel. Sa richesse et sa puissance

d'expression le rendent également éligible pour la modélisation de concepts et de processus « métier » (« business modeling ») et pour l'ingénierie de systèmes non logiciels. UML résulte de l'unification de techniques ayant fait leurs preuves pour l'analyse et la conception de grands logiciels et de systèmes complexes.

UML intègre neuf types de diagrammes destinés à la caractérisation du système modélisé et à la représentation des éléments qui le constituent :

*— les **diagrammes de cas d'utilisation** permettent de décrire les fonctionnalités du système et de représenter les différents types de sollicitations auxquelles il doit pouvoir répondre ;*

*— les **diagrammes de classes** sont destinés à décrire les propriétés structurelles des objets du monde réel, les concepts spécifiques du domaine considéré ou encore les notions abstraites que le système doit appréhender. Ce sont les diagrammes le plus fréquemment utilisés en modélisation orientée objet. En phase de conception du logiciel, ils sont exploités pour décrire l'architecture statique du système et les interdépendances entres ses constituants ;*

*— les **diagrammes d'objets** offrent un moyen de représenter les objets (c'est-à-dire les instances des classes figurant dans les diagrammes de classes) ainsi que leurs relations ;*

*— les **diagrammes de collaboration** permettent de formaliser les scénarios de mise en œuvre du système et de montrer comment les objets sont mis en jeu pour réaliser les cas d'utilisation. Ils décrivent les interactions entre les objets ;*

*— les **diagrammes de séquence**, comme les diagrammes de collaboration, décrivent les interactions entre objets. Ils mettent l'accent sur l'ordre chronologique dans lequel s'effectuent les échanges de messages entre objets ;*

*— les **diagrammes d'états** (ou diagrammes états-transitions) apportent le complément nécessaire à la formalisation des aspects dynamiques : ils répondent au besoin de modéliser les processus d'exécution et les comportements des objets en réaction aux stimuli auxquels ils sont soumis ;*

*— les **diagrammes d'activités** sont également dédiés à la représentation de l'exécution d'un processus : ils constituent une variante des diagrammes d'états ;*

*— les **diagrammes de composants** sont destinés à la description des éléments de configuration qui constituent le logiciel (binaires exécutables, bibliothèques, unités de compilation, etc.) et à la formalisation de leurs dépendances ;*

*— les **diagrammes de déploiement** permettent, enfin, de représenter l'implantation des différents programmes et composants logiciels sur l'architecture physique du système.*

À travers quelques exemples simples, le présent article décrit la syntaxe du langage UML pour chacun de ces diagrammes et tente de délimiter leurs champs d'application dans un processus de développement de logiciel.

1. Modélisation en génie logiciel

La modélisation est une activité technique qui s'inscrit dans de nombreux processus d'ingénierie. Son but est de fournir une représentation approchée du système ou du produit que l'on veut analyser, concevoir ou fabriquer. Cette représentation, appelée modèle, contribue à l'étude des caractéristiques techniques du système, des phénomènes relatifs à son fonctionnement ou encore de son architecture.

L'élaboration d'un modèle est souvent motivée par une complexité importante du système étudié. Du fait de cette complexité, il est difficile, voire impossible, de représenter le système globalement et exhaustivement avec un niveau de détail suffisant pour le comprendre, le définir ou le documenter. De façon générale, les modèles relèvent donc d'abstractions, c'est-à-dire qu'ils sont élaborés par rapport à l'ensemble restreint des propriétés que l'on veut étudier. Les caractéristiques secondaires sont volontairement occultées ou masquées afin de limiter la complexité. Un modèle peut intégrer différentes vues qui correspondent à des abstractions particulières et à des représentations distinctes, complémentaires et cohérentes. Dans certains cas, le développement d'un système peut même nécessiter l'élaboration de plusieurs modèles distincts portant sur des champs d'abstraction différents en fonction de l'avancement du processus.

Si le problème est simple, la solution peut résulter d'une intuition, d'une idée ou d'une simple réflexion intérieure. Si le problème est plus complexe, une tendance naturelle est d'utiliser un support écrit pour exprimer la solution et le cheminement intellectuel dont elle découle. Ce support peut alors servir pour raisonner et pour évaluer la solution envisagée puis pour la compléter, l'améliorer et la préciser jusqu'au niveau de détail voulu. La forma-

lisation écrite de la solution devient alors une référence qui va guider le processus de réalisation : l'objectif est de concrétiser la solution élaborée en s'appuyant sur l'expression qui en a été faite. Si le concepteur et le réalisateur sont des individus distincts qui interviennent au sein d'une même équipe, le support écrit qui formalise la solution devient également un vecteur de communication indispensable.

La complexité du processus d'ingénierie, la difficulté afférente au problème posé et les enjeux liés à la résolution de ce problème induisent donc des besoins plus ou moins forts de modélisation. La fabrication d'une cabane, la construction d'une maison particulière et la réalisation d'un grand ensemble immobilier n'engendrent pas les mêmes exigences en matière de modélisation et de formalisation. Si un simple croquis suffit dans le premier cas, un plan d'architecte est nécessaire dans le second alors qu'une maquette à échelle réduite sera parfois élaborée dans le troisième. Bien entendu, le génie logiciel n'échappe pas à cette règle et à chaque stade du processus de développement, la modélisation présente un intérêt particulier plus ou moins important en fonction de la complexité et de la taille du logiciel.

En phase d'analyse, l'intention première est de comprendre. Il s'agit de modéliser sans déformer la réalité en recherchant avant tout l'exactitude et la précision du modèle. L'analyse est une phase de découverte et le modèle doit formaliser les notions que l'analyste perçoit et cherche à caractériser afin de mieux les étudier. L'analyste est avant tout un observateur et le modèle qu'il élabore est une représentation de ce qu'il a observé. En phase de conception, le but est d'imaginer une solution susceptible de répondre à un problème connu et exprimé. La conception est une phase d'invention et le modèle décrit les éléments de la solution proposée. Il permet de confronter cette solution au problème initial. Le concepteur est un créateur et le modèle qu'il élabore est une représentation du système qu'il a imaginé. En phase de réalisation, l'objectif est de donner corps à la solution retenue à l'issue de la conception. La réalisation est une phase de concrétisation et le modèle doit expliciter les décisions techniques et les choix effectués lors de l'élaboration des programmes. Le modèle sert également à documenter le logiciel pour faciliter sa maintenance et sa réutilisation. Le réalisateur est un expert du langage de programmation et le modèle qu'il élabore porte sur la structure des programmes et sur leurs modalités de mise en œuvre.

On constate donc que, en règle générale, l'objectif de la modélisation est centré sur la compréhension et la communication. Pour atteindre cet objectif, l'activité de modélisation doit être conduite selon un certain nombre de principes :

— le modèle doit contribuer à la simplification du problème posé et il doit présenter la solution de façon synthétique : un modèle complexe est, *a priori*, un mauvais modèle ;

— le modèle doit être facile à comprendre (en totalité ou en partie) par un lecteur qui n'a pas participé à son élaboration. Il doit donc répondre à des critères de lisibilité et de modularité et il doit être présenté de façon pédagogique, par exemple, selon une approche descendante, en allant du général au détaillé ;

— le modèle doit pouvoir être facilement complété, modifié, adapté en fonction des progrès de l'étude et au gré des besoins engendrés par les demandes d'évolution et les choix techniques réalisés ;

— le modèle doit pouvoir illustrer la solution et contribuer à la documentation de celle-ci : il est souvent l'élément central d'un document d'analyse ou de conception.

Le besoin de communiquer et les orientations pratiques relevant de ces principes amènent naturellement à la nécessité de formaliser les concepts et les idées issus du processus d'ingénierie. Pour ce faire, la création d'un langage commun compréhensible par tous les acteurs du processus devient indispensable. Par ailleurs, l'utilisation d'un langage graphique et les approches de **modélisation « visuelle »** (*visual modeling*) présentent de nombreux avantages en ce qui concerne les principes cités plus haut : simplicité, universalité, concision, capacité d'expression, etc.

2. Origine et objectifs d'UML

Dans le domaine du génie logiciel, l'avènement des technologies objets a coïncidé avec le besoin impérieux de rationaliser le processus de développement. Ce besoin s'est révélé au travers des difficultés à maîtriser le développement des produits logiciels de taille et de complexité croissantes. Il s'en est suivi une prolifération de méthodes OO (orientées objets) censées apporter des réponses en terme de modélisation tant au niveau de l'analyse que de la conception (moins de 10 méthodes recensées en 1989 ; plus de 50 en 1994). Face à une telle diversité, les choix s'avèrent souvent difficiles pour les utilisateurs et les querelles d'experts ne font que compliquer la problématique de sélection. La disparité des méthodes est, bien sûr, un lourd handicap lorsqu'un projet nécessite la collaboration de plusieurs intervenants et l'introduction d'une nouvelle méthode dans l'organisation engendre des coûts d'appropriation souvent très importants. La standardisation apparaît donc comme l'unique moyen d'homogénéiser et de sécuriser les choix tout en apportant une garantie de pérennité.

À l'origine d'UML se trouvent les deux méthodes les plus utilisées dans le monde au début des années 1990 : OMT (Object Modeling Technique) de J. Rumbaugh [1] et OOD (Object-Oriented Design) de G. Booch [2].

Grady Booch rejoint la société américaine Rational dès le début des années 1990. Il travaille à l'élaboration de la méthode nommée OOD qui propose une notation graphique à base de « nuages » permettant de représenter les classes et les objets selon différentes visions (statique/dynamique, logique/physique). En 1991, James Rumbaugh, Mickaël Blaha et quelques autres ingénieurs du centre de recherche de General Electric aux États-Unis publient un livre intitulé *Object Oriented Modeling and Design* [1] dans lequel ils présentent les fondements de la méthode OMT. En France, Philippe Desfray et la société Softeam définissent la méthode classe-relation supportée par l'outil Objecteering : un premier livre est publié en 1992 [3]. De 1991 à 1994, toutes ces méthodes évoluent progressivement et gagnent en maturité au fil des expériences et des projets. De nouvelles versions de OOD et OMT-2 se font jour en 1993.

En 1994, J. Rumbaugh rejoint G. Booch chez Rational et, en octobre 1995, ils publient ensemble un *draft* version 0.8 de la méthode unifiée qui est largement diffusé et soumis à la critique des utilisateurs. Fin 1994, c'est au tour d'Ivar Jacobson, auteur de la méthode OOSE (Object Oriented Software Engineering), appelée aussi Objectory, et des techniques de modélisation par cas d'utilisation, de rejoindre Rational pour apporter sa contribution au travaux d'unification.

Un nouveau *draft* 0.9 paraît en juin 1996 immédiatement suivi de la version 0.91 en octobre 1996. Cette version marque un tournant dans le processus d'unification. La méthode unifiée devient UML. Les travaux se recentrent donc sur la définition d'un langage de modélisation graphique et textuel qui se veut universel. La normalisation du processus devient un objectif de second plan et la priorité est donnée à la formalisation du modèle, indépendamment du processus ayant conduit son élaboration. UML est destiné à permettre la présentation du résultat sous une forme compréhensible par le plus grand nombre sans présumer des moyens mis en œuvre pour le produire.

Dans ce premier temps, le processus d'unification repose essentiellement sur une consolidation de l'existant. L'optique prise est plutôt de reprendre les techniques qui existaient dans les méthodes OMT, Booch et OOSE et de les mettre en cohérence plutôt que d'en créer de nouvelles. Il s'agit donc bien de travaux d'unification conformément à ce que laisse présager le terme « Unified ».

En 1996, quelques grandes sociétés informatiques (DEC, HP, I-Logix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Texas Instrument et Unisys) rejoignent Rational sur le projet UML et fondent l'UML Partners Consortium pour travailler à sa définition.

À cette même époque, une *task force* « *Analysis & Design* » se met en place à l'OMG, autre consortium réunissant quelque six cents sociétés concernées par les technologies objets et supportant une mission de normalisation. Ce groupe de travail lance un appel à soumission auquel l'UML Partners Consortium répond en janvier 1997 avec la version 1.0. La version 1.1, qui prend en compte un certain nombre de remarques, est publiée en septembre 1997 et retenue par l'OMG dès le mois de novembre de cette même année. À partir de ce moment, UML ne cessera plus de se diffuser dans tous les domaines liés directement ou indirectement au logiciel et aux systèmes d'information.

Outre la satisfaction du besoin lié à la communication, la normalisation de la notation s'avère être un élément fondamental pour l'intégration des outils d'analyse et de conception objet dans l'entreprise. L'existence d'UML permet d'envisager la mise en place de passerelles permettant d'échanger les données de modélisation entre différents outils. Il devient donc possible de travailler dans des contextes multioutils où chaque industriel, voire chaque équipe, peut utiliser l'outil de son choix en fonction des critères qu'il juge pertinents : coût, stratégie société, ergonomie, adéquation au besoin technique, etc.

À partir de la version 1.3, les travaux menés à l'OMG s'orientent vers l'industrialisation du langage et, parallèlement, ils vont s'efforcer d'intégrer des concepts et des mécanismes destinés à faciliter et à généraliser son utilisation.

En 2003, la spécification de la version 2.0 est disponible sur le site Web de l'OMG.

3. Fondements et concepts de base

Selon le « paradigme objet », un système peut être vu comme un ensemble fini d'objets qui collaborent pour assurer une mission globale. Chaque objet intervient dans cette collaboration selon un modèle comportemental particulier que l'on spécifie par l'intermédiaire d'une **classe**. Le principe de classification amène à considérer que deux objets distincts régis par un même modèle comportemental appartiennent à la même classe. Dans le modèle objet, outre un moyen de définir les caractéristiques et le comportement propres à un sous-ensemble des objets du système, la classe est aussi une structure dédiée à la création (instanciation) des objets de ce sous-ensemble.

Exemple : pour dessiner une figure géométrique simple, il faut, en premier lieu, savoir à quelle catégorie elle appartient. Est-ce un carré ? Un cercle ? Un triangle ? Indépendamment de la taille, de la couleur et de l'épaisseur du trait, la forme générale et la méthode de dessin de la figure sont, bien entendu, fonctions de la classe à laquelle elle appartient. La définition précise de la classe des cercles devra donc permettre de reproduire le dessin d'un cercle autant de fois que nécessaire en appliquant chaque fois le même procédé et en particularisant l'instance par des attributs de taille et de couleur. Les différentes catégories de figures géométriques pourront être organisées selon une structure hiérarchique basée sur la classification. Ainsi, la classe des carrés sera considérée comme une sous-classe des rectangles qui sera elle-même vue comme une sorte de quadrilatère. On pourra procéder soit par *généralisation* (définir la classe de quadrilatères en généralisant la notion de rectangle, de losange, de trapèze, etc.), soit par *spécialisation* (définir la classe carré en spécialisant le rectangle). L'emploi du lien « est une sorte de » permettra de définir de nouvelles classes en procédant par analogie avec d'autres classes, les classes spécialisées héritant des propriétés des classes générales. Un modèle plus riche de la géométrie pourra inclure des notions de point, de plan, de repère, etc., et la réalisation d'un schéma géométrique pourra se ramener à l'invocation de la méthode « dessiner » sur un ensemble particulier d'instances des différentes sous-classes de figures géométriques.

Partant de ce principe, un objet peut donc toujours être considéré comme l'**instance** d'une classe et l'objectif fondamental d'une approche de modélisation orientée objet consiste en l'identification et la formalisation des classes d'objets utiles et nécessaires pour le fonctionnement du système. La difficulté majeure de la modélisation orientée objet réside précisément dans la recherche et la sélection des classes d'objets nécessaires au fonctionnement du système. Si le langage UML répond parfaitement au besoin de formalisation, il n'apporte, en revanche, aucune aide en ce qui concerne l'identification des classes et des objets. Ce travail difficile est de la seule responsabilité de l'ingénieur ou de l'analyste qui doit recenser les notions émergentes de l'étude pour les traduire conceptuellement. En résumé, l'utilisation d'UML ne garantit en rien la qualité du modèle.

De façon générale, les différents éléments qui permettent de caractériser les objets d'une classe sont appelés les **membres de classe** et se divisent en trois catégories :

— les *attributs*, qui décrivent les données représentatives d'un objet de la classe et qui permettent de lui associer des valeurs propres (variables d'instance) ;

— les *opérations* ou *méthodes*, qui décrivent les services offerts par chaque objet de la classe et qui peuvent être invoquées pour modifier l'état interne de l'objet ;

— les *relations* qui décrivent les liens avec les autres objets appartenant éventuellement à d'autres classes et qui traduisent donc les interdépendances entre les classes.

Exemple : dans le cas de la figure géométrique, le procédé de dessin sera modélisé par une méthode, les caractéristiques de taille et de couleur seront des attributs et on pourra mettre en relation un triangle et son cercle circonscrit par l'intermédiaire d'une association.

Une caractéristique essentielle d'un membre de classe concerne sa **visibilité**. Cette propriété permet de savoir si d'autres classes peuvent connaître et utiliser l'attribut, l'opération ou la relation. On distingue trois niveaux de visibilité :

— la visibilité d'un membre de classe est *publique* lorsque tous les objets du système peuvent y accéder (n'importe quel objet peut invoquer une opération publique, connaître ou modifier la valeur d'un attribut public et accéder aux instances mises en jeu par une relation publique) ;

— la visibilité d'un membre de classe est *privée* lorsque seul l'objet qui la définit y a accès (par exemple, une opération privée ne peut être invoquée que par l'objet lui-même) ;

— la visibilité d'un membre de classe est *protégée* lorsqu'il est accessible uniquement aux autres objets de la classe et aux objets de classes dérivées (c'est-à-dire les classes qui sont définies par spécialisation de la classe considérée).

Le principe d'encapsulation, largement répandu en génie logiciel, amène à proscrire l'utilisation de visibilité publique pour les attributs et les relations. Seul l'objet lui-même peut accéder et mettre à jour les valeurs caractéristiques de son état courant. Si cette règle est respectée, la connaissance et la mise à jour d'un attribut ou d'une relation définie par une classe d'objet ne peuvent se faire que par l'intermédiaire d'une opération publique de cette classe.

En UML, tous les éléments de modélisation qui peuvent être instanciés sont nommés, de façon générique, des **classificateurs**. La classe est le principal classificateur mais il en existe d'autres tels que le type, le composant, l'interface, etc. On distingue le concept de **type** et celui de classe : la classe est la réalisation d'un type particulier et en a donc toutes les caractéristiques mais elle correspond à une implémentation particulière. Le type est décrit par des caractéristiques comportementales qui déterminent une interface mais il peut exister plusieurs implémentations de cette interface. La classe est la réalisation de l'interface au travers d'une implémentation particulière.

Les membres d'un classificateur peuvent également être caractérisés par leur portée : soit ils concernent uniquement l'instance, ce qui signifie que chaque objet se caractérise par des valeurs

propres, soit ils concernent la classe toute entière, ce qui veut dire qu'ils ne possèdent qu'une valeur pour l'ensemble des instances. Par exemple, le dénombrement des instances d'une même classe pourra être réalisé par une opération de classe.

4. Modélisation statique

4.1 Diagramme de classes

En UML, la classe est représentée par un rectangle dans lequel figurent les labels et les éléments textuels qui décrivent ses caractéristiques. Le rectangle représentant la classe en UML peut être scindé en trois : le compartiment supérieur contient le nom de la classe, le compartiment du milieu contient la liste des attributs et le compartiment inférieur contient la liste des opérations. Seul le compartiment supérieur est obligatoire. Les attributs peuvent être typés et les signatures des opérations peuvent être également spécifiées dans une syntaxe proche de celle des langages de programmation. UML définit un certain nombre de types élémentaires (entier, réel, chaîne de caractères, booléen, etc.). La figure **1** présente différentes notations possibles pour une classe modélisant l'ensemble des coordonnées géographiques.

On peut distinguer plusieurs types de relations entre classes : l'association, l'agrégation, la spécialisation/généralisation, la dépendance.

4.1.1 Association

L'association traduit un lien structurel entre les objets des classes considérées. Dans la grande majorité des cas, une association permet de mettre en relation des objets appartenant à deux classes différentes mais elle peut aussi être utilisée pour lier entre eux les objets d'une même classe. Une association se caractérise par :

— un **nom** qui exprime l'idée du lien modélisé ;
— les **rôles** joués par les objets impliqués dans chacune des classes ;
— les **cardinalités** qui précisent le nombre d'objets impliqués dans chacune des classes ;
— l'orientation, ou **navigabilité**, qui permet de préciser les modalités de parcours du lien d'association.

Le modèle de la figure **2** indique qu'il existe une association entre la classe Coordonnee Geo et la classe Mobile. Cette association traduit un lien de *localisation instantanée* entre les objets appartenant à l'une et l'autre de ces deux classes. Ce lien doit nous permettre de connaître la *position* du mobile à un instant donné en accédant, *via* l'association, à la coordonnée géographique correspondante. Toute instance de la classe Mobile fait obligatoirement référence à une instance de la classe Coordonnee Geo tandis que l'on peut faire correspondre plusieurs instances de la classe Mobile

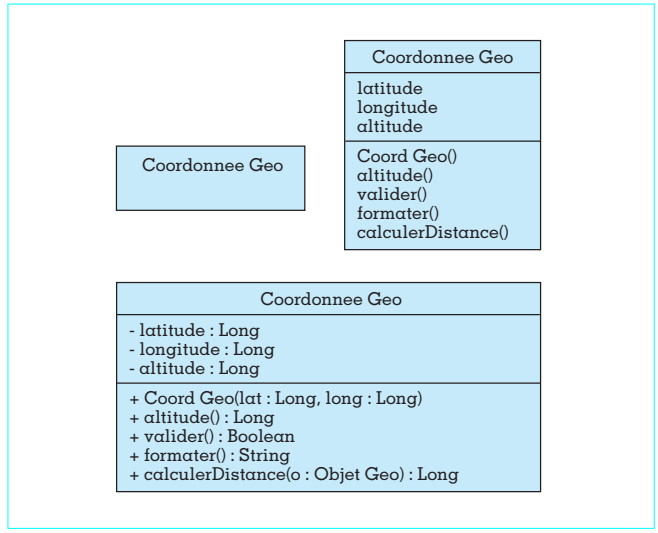

Figure 1 – Représentations de la classe Coordonnee Geo **en UML**

à une même instance de la classe Coordonnee Geo. Ces informations de cardinalité sont mentionnées sur le diagramme par l'intermédiaire du nombre minimum et du nombre maximum d'instances mises en jeu. Dans le cas présent, un mobile a une coordonnée géographique au plus et celle-ci peut ne pas être connue (notation 0..1). Par ailleurs, plusieurs mobiles peuvent être positionnés à la même coordonnée (notation *).

Nota : au contraire du modèle de données de Merise, les indications de cardinalité sont mentionnées sur le diagramme, à l'intersection du lien et de la classe correspondant aux instances concernées

Dans le cadre de ce même modèle, le lien d'association ne peut être parcouru que du mobile vers la coordonnée : chaque instance de Mobile gère donc une référence sur l'instance de Coordonnee Geo qui lui est associée mais il n'existe pas de lien direct d'une instance de Coordonnee Geo vers l'instance de Mobile. Cette indication est fournie par une flèche allant de la classe Mobile vers la classe Coordonnee Geo. De façon générale, on dit que les associations sont « navigables » et on exploite le sens de navigation pour s'assurer que l'on peut accéder à un objet en parcourant l'association qui le lie à un autre objet. Par défaut, l'association est navigable dans les deux sens et l'absence de flèche peut donc exprimer une relation mutuelle entre les deux classes.

Les attributs et les paramètres des opérations peuvent être typés par des types élémentaires de données. UML définit quelques types primitifs standards pour le métamodèle : booléen, entier, énumération, chaîne de caractères, etc., mais en règle générale, on utilise les types de base du domaine modélisé ou les types standards du langage de programmation en fonction du niveau de modélisation recherché.

Figure 2 – Association

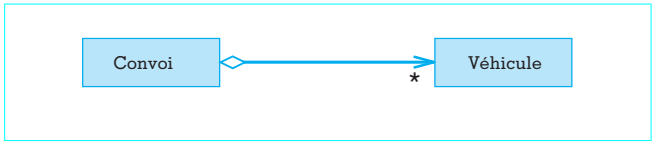

Figure 3 – Agrégation

En UML, un membre de classe de visibilité publique est signalé par un « + » précédant son nom, la visibilité privée est signalé par un « – » et la visibilité protégée est indiquée par le signe « # ». Le soulignement du nom désignant un membre de classe signifie que la portée de ce membre s'applique à l'ensemble de la classe. L'absence de soulignement signifie que la portée est limitée à l'instance.

4.1.2 Agrégation et composition

L'agrégation est une forme particulière d'association non symétrique qui exprime l'idée qu'un objet « fait partie » d'un autre objet. Sa sémantique détermine généralement un lien fort entre les classes concernées. L'une des classes est composée d'instances issues de l'autre classe et joue donc un rôle de conteneur.

Dans le modèle de la figure **3**, le losange situé à l'extrémité du lien côté Convoi indique qu'un convoi est une agrégation de plusieurs véhicules.

UML différencie également la **composition** de l'agrégation.

■ Une **agrégation** est dite *partageable* : elle permet de regrouper des éléments indépendants mais ceux-ci peuvent appartenir également à d'autres agrégats.

Exemple : un même joueur de football peut appartenir à plusieurs équipes (de club, de sélection, etc.). On modélisera donc l'équipe comme une agrégation de joueurs.

■ Une **composition** permet de traduire l'aspect composite d'un objet. Chaque composant correspond à une partie intégrante du tout et ne peut donc pas être partagée.

Exemple : un clavier est composé de touches et une touche ne peut appartenir qu'à un seul clavier.

La composition est parfois qualifiée d'agrégation par valeur, par opposition à l'agrégation par référence, ou agrégation partagée. Pour se rapporter à des opérateurs ensemblistes, on peut considérer que le lien d'agrégation détermine une appartenance et que le lien de composition correspond à une inclusion. Dans le cas de la composition, du fait du caractère non partageable, la cardinalité maximum relative au composite est toujours 1. Si, de plus, la cardinalité minimum est 1, alors la partie ne peut exister en l'absence du tout et tous les composants seront automatiquement détruits lors de la destruction du composite. En UML, on utilise un losange vide quand il s'agit d'une agrégation et un losange plein quand il s'agit d'une composition.

4.1.3 Spécialisation/généralisation

La **spécialisation/généralisation** se représente par une flèche allant de la classe la plus spécifique vers la classe la plus générale. La flèche est symbolisée par un triangle pour éviter qu'on la confonde avec la flèche de navigation d'une association.

Dans l'exemple de la figure **4** la classe *Objet Geo* est une classe *abstraite* (ce qui se traduit en UML par un nom de classe écrit en italique). Elle représente tout type d'objet localisé géographiquement et ne peut pas être directement instanciée car sa description est trop générale. Nous devons apporter des précisions en ce qui concerne les catégories possibles d'objets géographiques et la classe *Objet Geo* doit donc être dérivée (ou spécialisée) en différentes sous-classes qui définissent des propriétés spécifiques de l'objet en fonction de son type. Les membres (attributs, opérations et associations) définis au niveau de la classe générale sont hérités par les sous-classes. Ainsi, toutes les sous-classes de *Objet Geo* peuvent être représentées sur un fond de carte. La spécialisation peut s'accompagner d'une redéfinition des opérations de la classe générale. Dans ce même exemple, on peut supposer que les modalités de représentation d'un objet géographique sont différentes en fonction de son type. Les flèches de spécialisation peuvent être indépendantes les unes des autres ou interconnectées pour constituer une sorte de « râteau ».

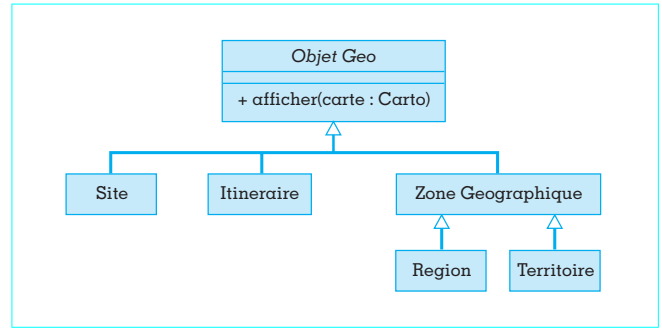

Figure 4 – Généralisation

4.1.4 Précisions de modélisation et notations complémentaires

Pour les associations, les spécifications de visibilité sont précisées par les mêmes signes que pour les attributs et méthodes (+, #, –) précédant le nom de rôle. De même, si la portée de l'association est du niveau de la classe, le nom du rôle concerné est souligné.

Une association entre deux classes traduit une multiplicité de liens entre des objets appartenant à l'une et l'autre de ces classes. C'est précisément le rôle des cardinalités que de préciser le nombre de liens à considérer. Au même titre que les objets sont des instances de classes, les liens sont, en quelque sorte, les instances de l'association. Chaque lien, en tant que représentant de l'association, peut avoir des caractéristiques qui lui sont propres et qui doivent pouvoir être modélisées. UML offre cette possibilité par l'intermédiaire de la **classe-association** qui permet de caractériser une association par des attributs comme on le fait pour une classe. Ces attributs peuvent donc prendre des valeurs particulières pour chaque lien défini entre deux objets. Une classe-association se représente de la même façon qu'une classe, c'est-à-dire par un rectangle, celui-ci étant rattaché à l'association par une ligne en pointillé.

La figure **5** présente un diagramme de classes UML qui modélise les objets géographiques. Ce modèle nous permet, en l'occurrence, de formaliser un certain nombre de définitions générales tout en spécifiant nos hypothèses de travail :

— un Site géographique se caractérise par une et une seule coordonnée géographique ;

— un Itineraire est défini comme une suite ordonnée de coordonnées géographiques correspondant aux différents *points de passage* qu'il détermine ;

— un Territoire est une zone géographique délimitée par un ensemble de coordonnées qui correspondent aux sommets d'un polygone ;

— une Region est également une zone géographique mais elle est délimitée par un cercle défini par une coordonnée géographique (le centre) et une distance maximum d'éloignement (le rayon), etc.

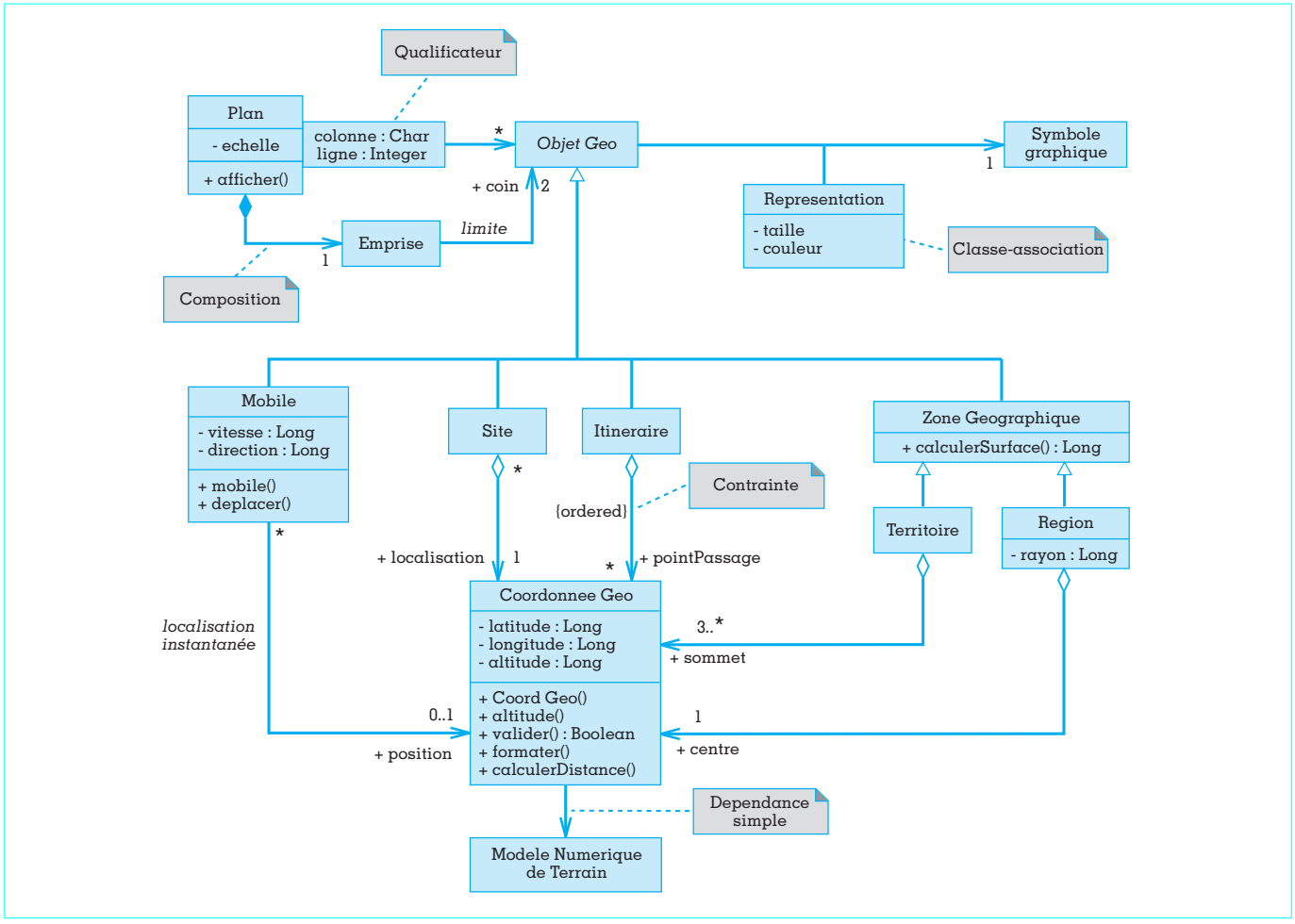

Figure 5 – Diagramme de classes

UML permet d'ajouter sur les diagrammes des **annotations** en langage naturel. Chacune d'entre elles est présentée dans un cadre spécifique qui peut éventuellement être relié par un trait en pointillé à l'élément concerné dans le diagramme. Dans le diagramme de la figure **5**, les annotations sont utilisées pour mettre en évidence certains éléments de notation graphique UML décrits ici mais, de façon plus générale, elles peuvent être exploitées pour documenter le diagramme ou pour apporter des précisions qui ne peuvent être exprimées formellement avec la syntaxe UML.

Les associations, les agrégations, les compositions et les généralisations sont des liens structurels entre classes qui apparaissent généralement dans une modélisation statique. Elles doivent être identifiées et formalisées dans les diagrammes de classes dès les premières étapes du projet car elles contribuent à la caractérisation des concepts utilisés. Lorsque l'on s'intéresse ensuite au fonctionnement dynamique du système modélisé et que l'on analyse les modalités de collaboration des objets, il arrive assez fréquemment que l'on rencontre d'autres types de relations de nature opérationnelle ou conjoncturelle. Par exemple, une méthode de la classe A peut prendre en paramètre une instance de la classe B, ce qui détermine un lien entre A et B. En UML, ces liens se traduisent par des **dépendances simples** représentées par des flèches en pointillé. Dans le diagramme de la figure **5**, on fait apparaître une dépendance entre la classe Coordonnee Geo et la classe Modele Numerique de Terrain (MNT) pour expliciter le fait que l'altitude terrain de chaque coordonnée géographique est fournie par un MNT. Ce lien permet de spécifier la solution technique mais n'influence pas vraiment les définitions respectives des deux concepts et n'a donc pas lieu d'être modélisé comme une association.

Une autre préoccupation du concepteur qui s'intéresse au fonctionnement dynamique du système concerne le parcours des liens d'association et la capacité à retrouver les instances liées les unes aux autres. Par exemple, lorsque l'on recherche un élément sur un plan, on se réfère généralement à un index qui fournit la position de l'objet dans une grille. En UML, on modélise ce mécanisme au moyen d'un **qualificateur** (en anglais *qualifier*) : on spécifie qu'il existe sur l'association un ou plusieurs attributs qui permettent de restreindre le domaine de recherche parmi les instances de la classe associée. Comme le montre la figure **5**, un qualificateur est représenté par un petit rectangle contenant le nom des attributs et positionné à l'extrémité du lien d'association du côté de la classe source. Ici, on traduit le fait qu'un plan permet, par l'intermédiaire d'un index, de retrouver les objets géographiques dans une grille.

Ce même diagramme de la figure **5** fait également apparaître une **contrainte** qui vient préciser que les coordonnées géographiques entrant dans la définition d'un itinéraire doivent être ordonnées. UML définit un certain nombre de contraintes normalisées

sur les associations et sur les spécialisations/généralisations mais il laisse aussi la possibilité à l'utilisateur de préciser des contraintes particulières soit de façon informelle (en langage naturel), soit de façon formelle [en utilisant, par exemple, le langage OCL (Object Constraint Language), voir § 10]. Dans le standard UML, une contrainte est formalisée par un mot-clef encadré par des accolades. Les contraintes peuvent permettre, par exemple, de préciser qu'une association est transitoire (mot-clef {transient}), qu'une spécialisation est totalement ou partiellement définie (mots-clefs {complete} et {incomplete}) ou encore que les sous-ensembles d'instances déduits d'une spécialisation sont disjoints ou non (mots-clefs {disjoint} et {overlapping}).

Les diagrammes de classes sont généralement utilisés tout au long du processus de développement d'un logiciel. Ils constituent, en quelque sorte, les diagrammes « pivots » qui définissent formellement et explicitement les concepts de base utilisés dans le reste du modèle. La plupart des autres diagrammes UML mettent en jeu des éléments dont la modélisation relève d'un diagramme de classe. Dans les phases préliminaires du développement, les diagrammes de classes permettent de modéliser des notions propres au domaine d'application dans lequel le logiciel doit s'insérer. Ils permettent de conduire la modélisation « métier » en offrant le moyen de décrire les objets du monde réel et les notions abstraites que l'utilisateur final perçoit ou manipule. En phase de conception, les diagrammes de classes permettent de décrire les objets qui constituent le système et qui vont collaborer pour assurer son fonctionnement. La syntaxe UML utilisée pour les diagrammes de classes est la même en analyse et en conception, ce qui permet d'évaluer la cohérence et la complétude des modèles issus de ces deux phases. Suivant les cas et les options méthodologiques, le modèle de conception est déduit du modèle d'analyse métier (le modèle « métier » est enrichi et complété pour intégrer les choix techniques d'architecture) ou il fait l'objet d'une autre modélisation avec identification de nouveaux objets de nature différente.

4.2 Diagramme d'objets

Comme nous l'avons déjà dit, le modèle de classe offre surtout une vision statique du système. L'exhaustivité et l'exactitude du modèle doivent également pouvoir être étudiées en fonction du comportement dynamique attendu. En premier lieu, il convient de vérifier que les classes modélisées couvrent la totalité du besoin et qu'elles permettent d'instancier les objets utiles au fonctionnement du système. Pour ce faire, le **diagramme d'objets** vient compléter la description de la structure statique en apportant le moyen d'exprimer l'organisation et les liens entre objets, c'est-à-dire entre des instances des classes modélisées grâce aux diagrammes de classes.

Est-il possible de construire un plan conforme au besoin de l'utilisateur en utilisant uniquement les instances des classes dérivées de *Objet Geo* ? Le modèle que nous avons élaboré décrit une structure générique mais il ne permet pas d'expliciter avec précision la représentation d'un plan donné. Si l'on veut, par exemple, modéliser un plan d'accès à un site S situé dans la région de Rouen, dans le parc industriel de Val de Reuil, en bordure de la route nationale 13 et de l'autoroute A13, il nous faut faire apparaître explicitement, une instance S de la classe Site, une instance Rouen de la classe Region, une instance zi_VDR de la classe Territoire et deux instances RN13 et A13 de la classe Itineraire. Le diagramme d'objets de la figure **6** constitue alors une réponse possible à ce besoin.

La représentation d'un objet est identique à celle d'une classe à ceci près que le nom de l'objet référencé dans la boîte est souligné et qu'il est suivi de « : » et du nom de sa classe d'appartenance. UML n'impose rien mais on peut également utiliser, comme dans la plupart des règles de programmation, des conventions de nommage pour différencier les classes et les objets (par exemple, la première lettre du nom en majuscule pour les classes et en minuscule pour les objets). Il n'est pas obligatoire de nommer un objet. Dans ce cas, le nom de la classe apparaît uniquement précédé par les « : » mais il est cependant souligné afin de bien différencier une instance de classe par rapport à la classe elle-même.

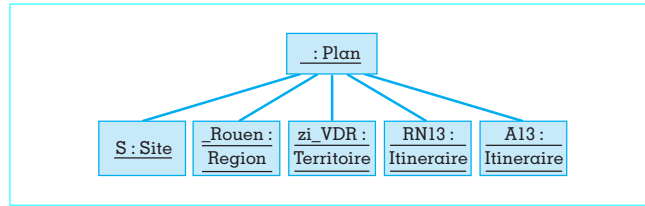

Figure 6 – Diagramme d'objets

Les attributs définis sur un objet peuvent prendre des valeurs particulières qui sont celles de l'instance considérée. Bien entendu, il n'est pas nécessaire de faire apparaître les cardinalités de relation entre objets puisque l'on représente explicitement les instances mises en jeu par la relation.

Les diagrammes d'objets sont utilisés pour traduire des structures particulières qui mettent en jeu plusieurs instances d'une même classe et/ou de différentes classes. Ils constituent l'un des moyens offerts par UML pour compléter et préciser les diagrammes de classes. Dans la plupart des cas, ils sont élaborés pour modéliser un ensemble d'objets qui collaborent au sein d'une structure globale. La formalisation de la collaboration entre les objets peut alors être modélisée en complétant le diagramme d'objets pour le transformer en un diagramme de collaboration tel que nous le décrirons au paragraphe 7.

5. Modélisation fonctionnelle

Si les diagrammes de classes et les diagrammes d'objets permettent de représenter explicitement la structure statique du système et de modéliser les concepts qui participent à sa définition, ils n'apportent pas de réponse acceptable en ce qui concerne l'expression du besoin fonctionnel à satisfaire. Quelles fonctions le système doit-il remplir ? Quels services doit-il rendre ? Quelle doit être sa mission au sein de l'organisation dans laquelle il prend place ? Autant de questions qui traduisent le problème posé selon le point de vue de l'utilisateur et qui sont donc essentielles pour exprimer le besoin et établir la spécification du système attendu. Pour y répondre, UML propose de modéliser les fonctionnalités du système au travers des **cas d'utilisation** ou *use cases*. Cette technique, développée et mise au point par Ivar Jacobson, constitue le complément indispensable des diagrammes de classes dans une approche cohérente de modélisation orientée objets.

5.1 Cas d'utilisation

Les cas d'utilisation décrivent les interactions du système par rapport à son environnement et plus particulièrement par rapport à ses utilisateurs. Chaque cas correspond à une façon particulière d'utiliser le système et à l'une de ses fonctionnalités. La modélisation des cas d'utilisation consiste à établir une classification des services que le système fournit : chaque cas d'utilisation se détermine par rapport à une catégorie de sollicitations possibles. De cette façon, le système peut être caractérisé par un ensemble de cas d'utilisation qui décrivent les différents types de sollicitations auxquelles il peut répondre.

Notons que l'on peut établir un lien étroit entre la définition d'un cas d'utilisation et celle de fonction de service en analyse de la

valeur : « *une fonction d'usage demandée à un produit ou réalisée par lui, afin de satisfaire une partie du besoin d'un utilisateur donné* (norme NF EN 1325-1) ». Un cas d'utilisation doit contribuer à la satisfaction du besoin opérationnel et apporter une valeur ajoutée au système modélisé. Il ne traduit pas une action ou une opération élémentaire réalisée par ce système mais bien une mission globale de celui-ci dont le résultat présente un réel intérêt ou constitue un apport significatif. Ce point fondamental constitue un critère majeur pour l'identification des cas d'utilisation et pour l'appréciation de leur granularité.

Les cas d'utilisation permettent donc de dire ce que le système permet de faire. Le modèle de la figure **5** représente un ensemble de notions relatives au domaine de la géographie sous la forme d'une structure cohérente de classes d'objets mais il ne précise pas quelle est l'utilité de cette structure et comment les objets coopèrent pour rendre un service global à un utilisateur. Les cas d'utilisation vont nous permettre de combler cette lacune.

5.2 Acteurs

En premier lieu, il s'agit précisément d'**identifier les utilisateurs potentiels** du système étudié et de définir leurs profils respectifs. Chaque entité située dans l'environnement du système et intervenant de façon plus ou moins directe sur son fonctionnement va être modélisée par un objet particulier appelé **acteur**. Bien que la notation UML prévoit sa représentation par un petit personnage, l'acteur n'est pas une personne physique et n'est pas obligatoirement humain : il représente une entité située à la périphérie du système et jouant un rôle déterminé. Un même individu ou une même entité physique peut être modélisé par plusieurs acteurs dans la mesure où il joue plusieurs rôles différents.

On peut classer les acteurs en quatre catégories :

— les acteurs principaux qui utilisent les fonctions essentielles représentatives de la mission du système ;

— les acteurs secondaires qui effectuent des tâches relatives à l'exploitation et à la bonne marche du système. Leur existence est directement liée à celle du système et à ses caractéristiques opérationnelles. On trouve dans cette catégorie les administrateurs, les superviseurs, les opérateurs de maintenance, etc. ;

— les autres systèmes avec lesquels l'objet modélisé doit interagir ;

— les matériels externes représentant des équipements, des dispositifs ou des ressources qui ne font pas partie du système mais interagissent avec lui dans le cadre de la réalisation de sa mission.

Conformément à l'approche orientée objet, les acteurs sont organisés en différentes classes et peuvent être situés les uns par rapport aux autres dans une hiérarchie de spécialisation/généralisation. On se contente généralement de décrire chaque classe d'acteurs par quelques mots en langage naturel mais l'on peut également la caractériser par des attributs, par des opérations correspondant aux tâches prises en charge et par des associations avec des classes ou avec d'autres acteurs. Dans le modèle UML, on peut donc dire que l'acteur est un objet comme les autres à ceci près qu'il ne fait pas partie du système bien qu'il interagisse avec lui. Une bonne modélisation des acteurs permet de bien appréhender l'environnement du système et de mieux comprendre par qui et comment il sera utilisé. La figure **7** représente un exemple de classification des acteurs.

5.3 Diagramme de cas d'utilisation

L'intervention d'un acteur se traduit par un ou plusieurs cas d'utilisation. Chaque cas correspond à un ensemble de séquences d'actions réalisées par le système et conduisant à un résultat attendu par l'acteur considéré. Un cas d'utilisation spécifie donc une fonctionnalité que le système doit assurer pour répondre aux

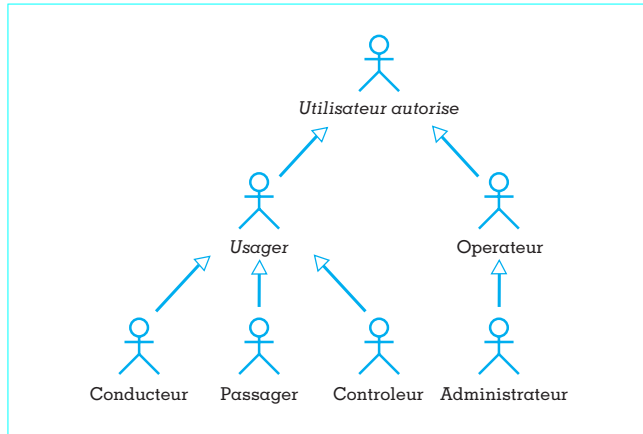

Figure 7 – Modélisation d'acteurs

attentes d'au moins un de ses utilisateurs. Cette spécification ne présume pas de la solution technique et des moyens mis en œuvre pour parvenir au résultat. La description des cas d'utilisation porte essentiellement sur le comportement global du système, sur la nature des résultats produits et sur la logique « métier » qui permet d'y parvenir. C'est donc, le plus souvent, en étudiant les besoins et les intentions des acteurs que l'on identifie les cas d'utilisation.

En UML, un cas d'utilisation est représenté par une ellipse connectée à un ou plusieurs symboles d'acteurs par des traits continus qui formalisent les liens de communication entre le système et les acteurs. Chacun de ces liens correspond sémantiquement à une association entre l'acteur et le cas d'utilisation. Tous les éléments descriptifs d'une association peuvent donc être utilisés pour préciser l'interaction entre l'acteur et le système : cardinalité pour spécifier le nombre d'acteurs mis en jeu, navigabilité pour indiquer le rôle de producteur ou de consommateur joué par l'acteur, etc. Le diagramme de la figure **8** représente les fonctionnalités d'un système sous la forme de cas d'utilisation : un *Usager* **doit pouvoir** Rechercher une destination **et** Calculer un itineraire ; un Controleur **doit pouvoir** Suivre les deplacements des vehicules ; **dans chacun de ces cas,** Visualiser (des objets geographiques) sur un fond de carte **est nécessaire ;** un Operateur **est chargé de** Numeriser les cartes **et** Editer les plans**,** etc.

La figure **8** fait apparaître des liens entre cas d'utilisation : le lien portant le mot-clef « include » formalise l'inclusion de la séquence d'actions Visualiser sur fond de carte dans les séquences d'actions des autres cas d'utilisation. Pour rechercher une destination, calculer une route ou suivre les déplacements d'un véhicule, il est nécessaire de visualiser les objets sur un fond cartographique. Les trois cas d'utilisation principaux intègrent tous la même suite d'actions dont la description peut donc être factorisée. Pour chacun de ces trois cas, l'affichage du fond cartographique avec superposition de représentations graphiques sera réalisé à un endroit donné de l'enchaînement prévu et fera l'objet d'une unique description sous la forme d'un **cas d'utilisation inclus**. En l'occurrence, ce dernier ne sera pas exécuté seul mais au sein d'un autre cas d'utilisation.

Il est également possible d'exprimer le fait qu'un cas d'utilisation incorpore, sous certaines conditions, une séquence d'actions optionnelle. Cette séquence fait alors l'objet d'un cas d'utilisation particulier qui étend le cas général. Le **cas d'utilisation étendu** est alors relié au cas d'utilisation de base par une flèche portant le mot-clef « extend ». Dans l'exemple de la figure **8**, on voit que le cas d'utilisation Suivre les deplacements en convoi est une extension de Suivre les deplacements des vehicules : on suppose donc que le suivi de véhicules peut être complété par une suite d'actions

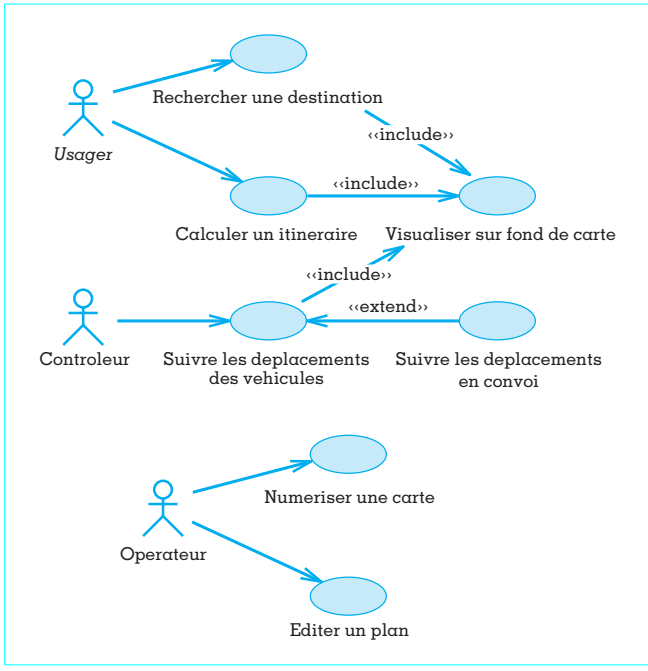

Figure 8 – Diagramme de cas d'utilisation

optionnelle au cours de laquelle des véhicules seront agrégés pour constituer le convoi à suivre. Une fois cet enchaînement terminé, le système reprendra le déroulement nominal du cas d'utilisation de base. Le **point d'extension**, formalisé par une étiquette dans la description du cas d'utilisation de base, pourra être explicité de façon à préciser qu'un regroupement de plusieurs véhicules est nécessaire dans le cas où l'on veut constituer un convoi.

Les cas d'utilisation peuvent être vus comme des classes de comportement dont les instances sont des scénarios d'utilisation du système. Dans notre exemple, Editer un plan est une classe et la suite d'actions réalisées par le concepteur pour élaborer un plan particulier correspond à un scénario d'utilisation possible. En conséquence, les cas d'utilisation peuvent être hiérarchisés par spécialisation/généralisation en considérant que certains comportements particuliers sont dérivés de comportements plus généraux. Comme le traduit la figure **9**, l'édition de plan pourra être réalisée librement sans support cartographique ou par annotation d'une carte préalablement numérisée. Ces deux classes de scénarios seront considérées comme des cas d'utilisation dérivés.

L'objectif de la modélisation étant de définir les fonctionnalités du système par un ensemble cohérent de cas d'utilisation, il faut donc s'assurer que le modèle apporte des éléments de réponse aux questions qui portent sur la définition du contexte dans lequel le système doit fonctionner, l'effet de ce fonctionnement, les services que l'on peut en attendre, la valeur ajoutée pour l'utilisateur, etc. L'objectif de la modélisation par cas d'utilisation n'est pas de décrire le fonctionnement interne du système mais plutôt de clarifier son utilité par rapport aux besoins des acteurs avec lesquels il interagit.

Le nombre et la granularité des cas d'utilisation fait aujourd'hui l'objet d'une controverse dans le monde des utilisateurs d'UML. Certains auteurs recommandent de ne modéliser que les cas d'utilisation les plus importants et de se limiter à une vingtaine de cas alors que d'autres prônent l'identification de multiples petits cas, d'utilisation. Il semble que le désaccord soit d'ordre méthodologique : dans un cas, on privilégie une approche *top-down* en dégageant les fonctionnalités majeures du système et en les détaillant

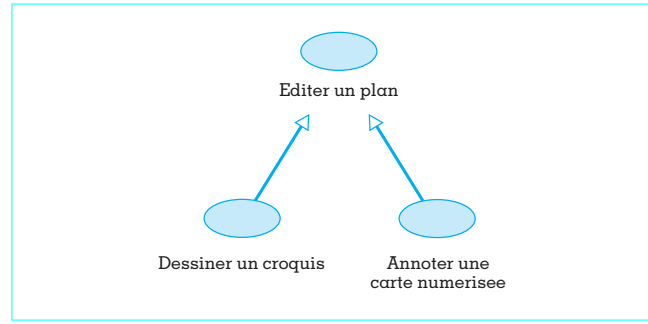

Figure 9 – Spécialisation/généralisation de cas d'utilisation

ensuite par la description de multiples scénarios ; dans l'autre cas, on assimile chaque cas d'utilisation à un unique scénario, ce qui amène à identifier un grand nombre de cas sans forcément chercher à en synthétiser la vision globale. Quelle que soit la stratégie retenue, la difficulté majeure lors d'une analyse par cas d'utilisation réside dans le choix du niveau de détail auquel l'on doit s'arrêter.

Il convient également de souligner que la modélisation par cas d'utilisation ne doit pas dériver vers une simple décomposition fonctionnelle. La tendance de nombreux utilisateurs est de reproduire avec les cas d'utilisation une démarche « traditionnelle » d'analyse fonctionnelle descendante pour aboutir à des « sous-cas d'utilisation » et, en final, à des transactions élémentaires entre les acteurs et le système. Outre le fait qu'elle ait montré dans le passé ses limites sur le plan méthodologique, cette façon de procéder risque bien de conduire à un modèle complexe qui n'aura que peut d'utilité pour la suite d'un développement orienté objet.

Comme nous l'avons vu, UML prévoit, par défaut, de représenter les acteurs avec un symbole graphique schématisant un humain. Pour éliminer toute ambiguïté, il peut être intéressant de représenter les acteurs matériels et les systèmes non humains en utilisant une notation plus appropriée que le petit personnage. UML offre la possibilité de représenter un acteur sous la même forme qu'une classe en ajoutant le mot-clef <<actor>> au-dessus de son nom. Il autorise également l'utilisation de symboles plus explicites dès lors qu'ils facilitent la lecture et la compréhension du modèle sans pour autant remettre en cause sa sémantique. Cette approche s'inscrit parfaitement dans une démarche de type *visual modeling* telle que la préconise UML et elle est particulièrement opportune dans le cadre d'une modélisation « métier » faisant intervenir des entités physiques.

En résumé, les diagrammes de cas d'utilisation UML permettent de représenter les acteurs du système modélisé, les cas d'utilisation de ce système ainsi que les interactions entre acteurs et cas d'utilisation. Un diagramme doit cependant être complété par une description textuelle de chaque cas d'utilisation en langage naturel ou en langage formel. La description peut également être enrichie par d'autres diagrammes UML qui illustrent des caractéristiques spécifiques du système par rapport au cas d'utilisation considéré. La notation UML prévoit, pour ce faire, des diagrammes d'états ou des diagrammes d'activités qui permettent de décrire des processus plus ou moins formellement mais des diagrammes de flux de données ou de flux de contrôle peuvent également être utilisés.

5.4 Intérêt méthodologique des cas d'utilisation

Au-delà de la notation UML, bon nombre de réflexions sont aujourd'hui menées autour du processus et des méthodes de développement de logiciel. Il est aujourd'hui communément admis que

le modèle de la « cascade » ou le « cycle en V » ne constituent pas des réponses totalement satisfaisantes en ce qui concerne le processus de développement de logiciel. Les méthodologies modernes doivent, entre autres choses, privilégier la **satisfaction du besoin** et la **maîtrise des risques**. De façon générale, les solutions aujourd'hui préconisées passent par la livraison précoce de versions préliminaires et par la capacité à intégrer le retour d'expérience dès les premières phases du projet. Dans le monde du génie logiciel, les auteurs recommandent donc aujourd'hui, de façon unanime, des pratiques de développement basées sur un **processus itératif et incrémental**.

L'un des problèmes majeurs de ce type de processus concerne la planification du développement : comment définir les incréments en liaison avec l'architecture de telle sorte que chaque version soit cohérente et qu'elle puisse être confrontée au besoin du client au travers d'une utilisation en situation réelle ? Une version de logiciel ne peut pas se définir exclusivement en terme de classes d'objets car les attentes du client/utilisateur s'expriment en terme de fonctionnalités et de services rendus et non pas en nombre de concepts pris en compte ou en quantité de classes compilées. Pour un utilisateur de logiciel, peu importe de disposer d'un ensemble de classes d'objets logiciels, ce qu'il souhaite c'est savoir les faire fonctionner ensemble pour répondre à son besoin. Dans ce cadre, les cas d'utilisation constituent un élément de réponse intéressant car ils offrent le moyen de définir des incréments qui traduisent à la fois des fonctionnalités et des **collaborations d'objets**. Chaque itération peut être consacrée à une ou plusieurs **réalisations de cas d'utilisation** et chaque nouvelle version satisfait donc de nouveaux besoins fonctionnels. Un cas d'utilisation peut donc être vu comme une unité de planification du projet.

En résumé, les cas d'utilisation permettent de :

— recentrer le développement sur le besoin ;

— délimiter le périmètre fonctionnel du système à développer ;

— améliorer le dialogue avec le donneur d'ordre ;

— planifier le développement selon un processus incrémental en définissant les versions successives par ajout progressif de cas d'utilisation ;

— préparer les tests de validation en formalisant les scénarios ;

— assurer la transition entre l'aspect fonctionnel du cahier des charges et l'aspect objet de la conception technique.

En ce qui concerne ce dernier point, l'analyse par cas d'utilisation apparaît comme le moyen le plus efficace pour recenser les objets nécessaires au fonctionnement du système. La réalisation d'un cas d'utilisation nécessite la collaboration d'un certain nombre d'objets qu'il faut pouvoir instancier à partir des différentes classes modélisées par ailleurs. Dans cette optique, un diagramme de classes « participantes » peut être associé à chaque cas d'utilisation. Cela permet de préparer la modélisation des interactions que nous aborderons au paragraphe 7.

6. Structuration de modèle

6.1 Notion de paquetage

Revenons à notre premier exemple de modèle UML dont le but était de décrire les notions relatives aux objets géographiques (site, itinéraire, territoire, région, etc.). Ce modèle fait apparaître la classe Coordonnee Geo qui décrit sommairement les caractéristiques des coordonnées géographiques. Cette caractérisation risque cependant de ne pas être suffisante dans un contexte plus général que celui de notre exemple. Pour améliorer la rigueur et la précision du modèle, il est nécessaire de prendre en considération certains concepts du « métier » de la géographie. Remarquons d'abord que la géographie repose elle-même sur une modélisation de la Terre dont la forme est approximée par une Ellipsoïde. Une coordonnée géographique se détermine donc par rapport à un système construit à partir d'une Ellipsoïde de référence et il nous faut donc considérer une nouvelle classe dans notre modèle. Les ellipsoïdes définis par les géographes (l'ellipsoïde de Clarke, ED50, WGS84, etc.) seront des instances de cette classe. Les coordonnées pourront également être exprimées suivant la projection UTM (Universal Transverse Mercator) et suivant différents formats : degrés décimaux, degrés minutes secondes ou UTM. Les calculs et projections cartographiques devront tenir compte du fait que la coordonnée se situe dans l'hémisphère nord ou dans l'hémisphère sud.

Toutes ces nouvelles notions devront prendre place dans notre modèle. D'un point de vue applicatif, il faudra peut-être envisager de nouvelles classes liées à des notions plus opérationnelles. Enfin, dans les phases de conception technique, il sera sans doute utile d'ajouter des classes d'IHM (interface homme-machine), des classes liées à la base de données ou toute autre classe permettant de satisfaire les exigences fonctionnelles du logiciel à développer. Toutes ces notions viendront donc enrichir le modèle et il en résultera un accroissement important de la complexité du modèle qui sera, certes, plus complet et plus proche de la réalité mais aussi beaucoup plus difficile à appréhender.

Pour éviter cet écueil, il est indispensable d'organiser les classes et de structurer le modèle pour le maîtriser dans sa globalité tout en préservant l'ensemble des détails qui font sa richesse. Cette démarche de structuration nécessite la création d'entités permettant de regrouper les classes par thèmes selon des critères plus ou moins formels pour constituer une synthèse du modèle. En UML, une telle entité prend le nom de **paquetage** (ou *package* en anglais). Pour le modèle que nous étudions en l'occurrence, on pourrait, par exemple, distribuer l'ensemble des classes identifiées dans deux paquetages différents :

— le paquetage geoLocalisation qui permettrait de regrouper toutes les classes présentées sur le diagramme de la figure **10** ;

— le paquetage objetGeo réunissant les autres classes que nous avons fait apparaître sur le diagramme de la figure **5**.

On peut donc dire, pratiquement, qu'un paquetage décrit un sous-ensemble du modèle et qu'il peut faire l'objet d'un diagramme indépendant offrant une vue détaillée de ce sous-ensemble. UML permet de représenter graphiquement les paquetages dans les diagrammes de classes. La notation retenue correspond à un symbole prenant la forme d'un dossier par analogie avec l'objet destiné communément à ranger des documents : les paquetages servent à « ranger » les classes. Les paquetages doivent être organisés et structurés en fonction de leurs interdépendances. Dans notre exemple, les objets géographiques qui sont modélisés par une classe contenue par le paquetage objetGeo font référence à la classe Coordonnee Geo qui appartient au paquetage geoLocalisation. Cela crée un lien de dépendance entre ces deux paquetages qui nous amène à dire que le paquetage objetGeo *importe* des classes du paquetage geoLocalisation. Le diagramme de la figure **11** donne une représentation, selon une vue externe, de ces deux paquetages et de leur dépendance.

On admet le plus souvent que toutes les classes d'un paquetage doivent pouvoir figurer sur un seul et même diagramme de classes. Si le nombre de classes contenues dans le paquetage ne permet pas de satisfaire ce critère, il convient d'envisager une nouvelle décomposition en paquetages de taille plus petite. Chaque paquetage peut donc faire l'objet d'un diagramme de classes où apparaissent toutes les classes qu'il regroupe ainsi que les caractéristiques de ces classes. UML autorise également des niveaux de décomposition hétérogènes incluant à la fois des paquetages et des classes. Des représentations de classes et de paquetages peuvent donc figurer sur un même diagramme de classes.

Pour simplifier le modèle ou, plus précisément, pour en masquer la complexité, il est également possible d'utiliser le principe d'encapsulation. Si on adopte le point de vue de l'utilisateur, certaines classes d'un paquetage sont plus importantes que d'autres.

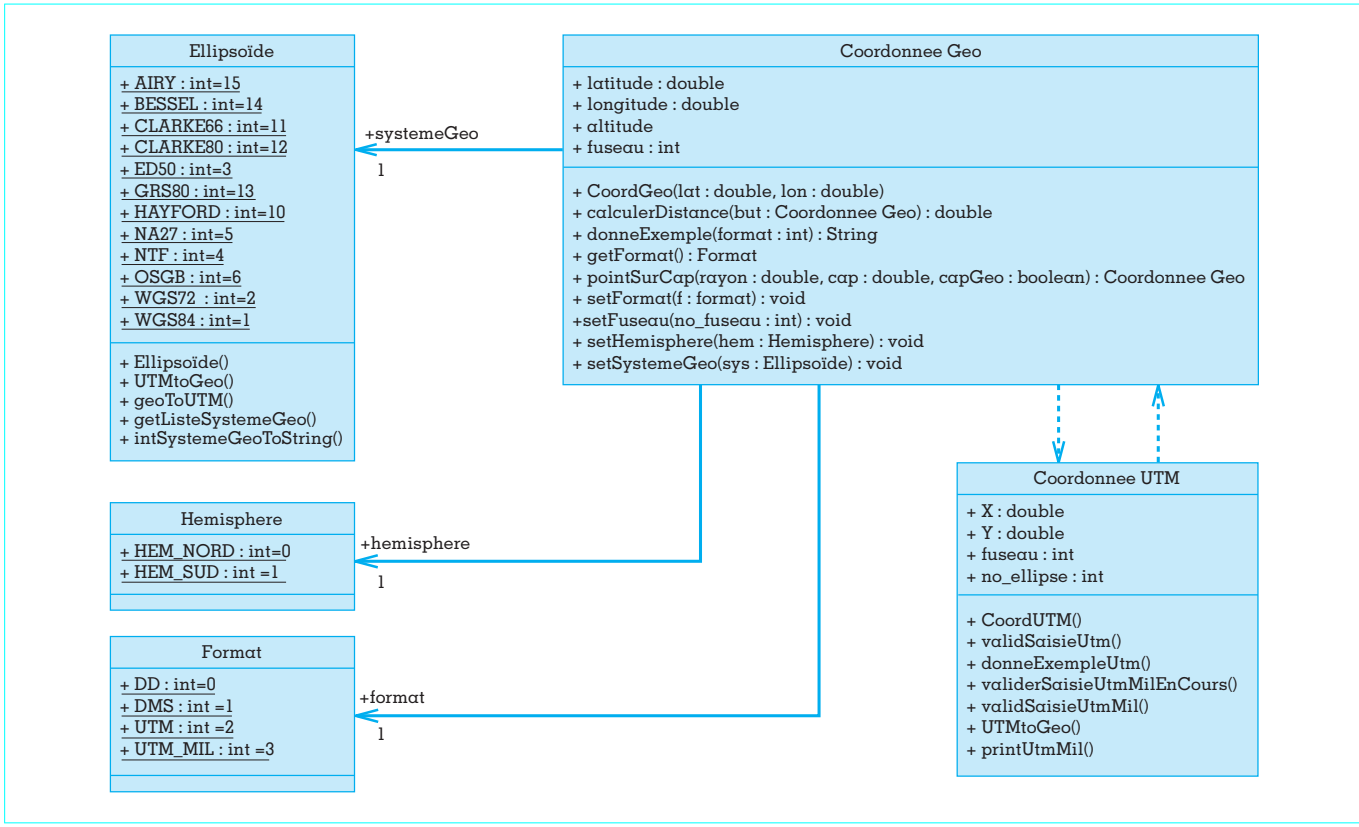

Figure 10 – Diagramme de classes pour le paquetage geoLocalisation

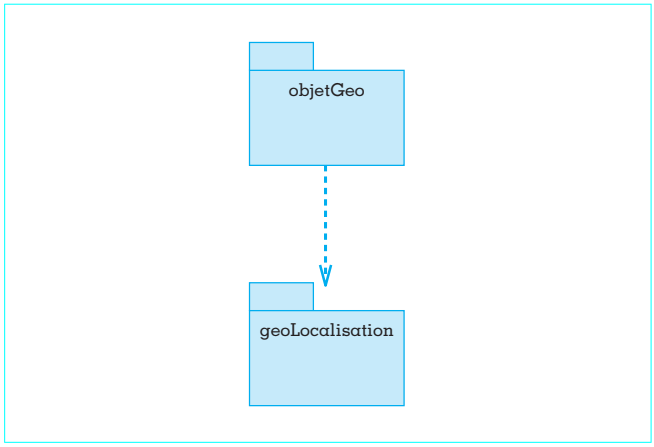

Figure 11 – Diagramme de classes : vue externe de paquetages

La connaissance de l'altitude du terrain en un lieu géographique donné nous est fournie par l'intermédiaire d'une méthode de la classe Coordonnee Geo. Pour accéder à ce service, il n'est pas nécessaire de savoir que l'on exploite un modèle numérique du terrain et la classe correspondant à ce mécanisme peut donc être masquée à l'utilisateur. La notion de modèle numérique de terrain est, en l'occurrence, une notion technique que certains lecteurs du modèle pourront ignorer sans que cela prête à conséquence. Le masquage de ce type de classe va donc contribuer à minimiser le nombre d'abstractions à intégrer dans la compréhension du modèle. La classe Coordonnee Geo sera considérée comme une classe d'*interface* (publique) et le Modele Numerique de Terrain sera en revanche modélisé par une classe d'*implémentation* (privée).

En UML, le paquetage correspond également à un espace de nommage spécifique, ce qui signifie que des classes de même nom peuvent cohabiter dans le même modèle à condition qu'elles appartiennent à des paquetages différents. Il convient donc, en cas d'ambiguïté, de préciser le paquetage d'appartenance d'une classe. UML propose, pour cela, de préciser le nom du paquetage préalablement au nom de la classe en utilisant le séparateur « :: » entre les deux identifiants. Par exemple, la classe Coordonnee Geo du paquetage geoLocalisation se notera en UML geoLocalisation: : Coordonnee Geo. Cette règle d'identification permet également de référencer, dans un diagramme de classes dédié à la modélisation d'un paquetage, des classes appartenant à d'autres paquetages. Dans ce cas, le nom du paquetage considéré est implicite pour les classes internes et les classes externes sont nommées en précisant explicitement le nom du paquetage.

6.2 Modélisation d'architecture

UML permet d'imbriquer les paquetages les uns dans les autres, c'est-à-dire d'agréger des paquetages pour constituer des paquetages de paquetages. Dans notre exemple, on pourra réunir au sein d'un paquetage geographie le paquetage objetGeo, le paquetage geoLocalisation et tous les paquetages qui contiennent des classes

se rapportant à la géographie. Cela nous amènera à désigner la classe Coordonnee Geo en précisant son chemin d'accès à travers l'arborescence d'inclusion des paquetages : systeme :: metier :: geographie :: Coordonnee Geo.

Sur le plan formel, UML ne fixe aucune contrainte de structuration relative à l'utilisation des paquetages. On peut, dans un même paquetage, trouver des classes et des sous-paquetages. L'expérience amène cependant à quelques recommandations. En premier lieu, il paraît préférable d'éviter les structures complexes basées sur des « emboîtements » de paquetages et des imbrications à plusieurs niveaux. Il est également recommandé de ne pas définir de dépendance circulaire entre paquetages : si un paquetage P1 dépend d'un paquetage P2, alors P2 ne doit pas dépendre, directement ou indirectement, de P1. Le respect de cette règle contribue à assurer la modularité du modèle et permet, lors de la programmation et de l'assemblage des paquetages, d'éviter un certain nombre de problèmes de compilation et d'édition de lien. Dans l'optique d'une modélisation de conception destinée à définir l'architecture statique d'un logiciel, il est presque impératif de respecter ces règles de structuration : pas d'utilisation mutuelle de paquetage, pas de cycle dans le graphe d'utilisation entre paquetages.

On peut donc construire une hiérarchie de paquetages et la représenter par une arborescence dans un diagramme statique correspondant à une vue externe du système modélisé. Chaque feuille de l'arbre, correspondant alors à un paquetage, peut faire l'objet d'une vue interne particulière qui détaille l'organisation des classes à l'intérieur du paquetage. Suivant ce principe, et en supposant que l'on veuille exploiter le modèle de la géographie pour construire un module logiciel destiné à afficher des objets géographiques dans une fenêtre, il faudra, par exemple, ajouter des classes d'objets concernant l'affichage et les symboles utilisés pour représenter les objets. Cela donnera lieu à la création de nouveaux paquetages affichageGeo et symbole, ce qui nous amènera, par exemple, à la structure proposée sur la figure **12**.

Tout comme la classe, le paquetage correspond à une abstraction particulière du système et sa définition est donc liée au point de vue très subjectif de l'individu qui réalise la modélisation. Il n'y a pas de règle ni même de méthode pour identifier les paquetages. L'existence du paquetage traduit la prise en compte d'un concept global dont le détail est fourni par les classes qu'il contient. Le paquetage peut donc être la représentation de toutes sortes de « choses » :

— une notion générale et ses spécialisations possibles issues d'une taxinomie quelconque et dont la traduction prend la forme d'une hiérarchie de classes se rapportant à un même domaine de connaissances (décomposition en domaines « métier ») ;

— un ensemble cohérent de classes dont les instances vont être mises en jeu pour la réalisation d'un service, d'une fonction ou d'un processus (décomposition fonctionnelle) ;

— un niveau d'abstraction : on peut avoir des paquetages de niveau « logique » dans lesquels on décrit des notions proches du besoin utilisateur et des paquetages de niveau « physique » dans lesquels on va trouver des notions liées à l'implémentation (découpage organique « en couches ») ;

— une bibliothèque de classes fournissant des services génériques (*framework*) ;

— un module, un lot de travaux ou un sous-système dont la réalisation relève d'une entité particulière dans l'organisation mise en place pour la maîtrise d'œuvre.

En ce qui concerne ce dernier point, il importe que le paquetage respecte les critères de modularité habituels :

— une forte cohésion des classes qui le constituent (toutes les classes se rapportent à l'abstraction centrale du paquetage) ;

— un faible couplage avec les autres paquetages (le paquetage peut être facilement isolé et référence peu de classes externes).

De façon générale, le paquetage est l'unité de structuration du modèle UML. Si le système modélisé supporte de nombreux cas d'utilisation, ceux-ci peuvent être répartis dans un ensemble de

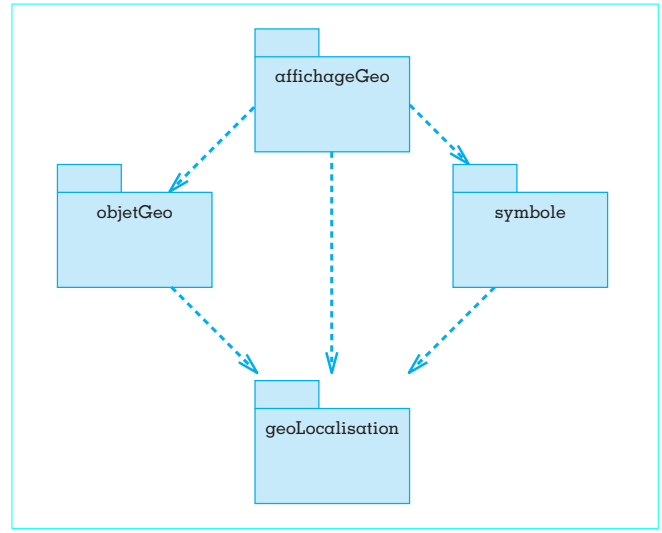

Figure 12 – Architecture statique et organisation en paquetages

paquetages indépendants. Chaque paquetage, considéré à ce niveau, regroupe donc un sous-ensemble des cas d'utilisation identifiés. Dans ce cas, il paraît opportun de faire correspondre un paquetage à un domaine fonctionnel particulier et la structure de décomposition en paquetage correspond alors à ce que l'on appelle généralement l'**architecture fonctionnelle** du système.

Les paquetages sont utilisés en UML pour définir l'arborescence du produit et permettent de traduire une décomposition du système en sous-systèmes. Chaque sous-système doit être aussi indépendant que possible des autres sous-systèmes et il peut être considéré comme un système à part entière à un niveau d'abstraction inférieur. Dans ce cadre, il inclut ses propres cas d'utilisation et ses propres classes d'objets. On peut remarquer qu'un objet modélisé comme partie intégrante du système au niveau d'abstraction le plus haut peut devenir un acteur du sous-système identifié au niveau d'abstraction inférieur.

7. Modélisation d'interactions entre objets

7.1 Scénario

À l'issue de la phase d'analyse de besoin, le modèle UML décrit des cas d'utilisation et un certain nombre de classes d'objets qui représentent ce que l'utilisateur perçoit. Chacun des cas d'utilisation peut être assimilé à l'intervention d'un acteur qui adresse au système une succession de stimuli afin de le piloter et d'activer ses fonctions. En UML, une telle séquence d'interactions entre les acteurs et le système est appelée un **scénario**. Pour un même cas d'utilisation, il peut exister différents scénarios, chacun se caractérisant par un début correspondant à son déclenchement, par une fin correspondant au résultat auquel il conduit et par des enchaînements successifs correspondant aux alternatives proposées et aux réactions ou décisions des intervenants. Un cas d'utilisation peut être considéré comme une classe de scénarios.

Dans un premier temps, la description d'un scénario peut être réalisée de façon informelle en langage naturel ou de façon structurée en utilisant des fiches organisées en rubriques. La figure **13** présente un exemple de fiche de description de cas d'utilisation.

Titre :	*Édition d'un plan de travail régional*	**Version :**	*1.0*
Auteur :	*P. Giroux*	**Date :**	*02/09/03*
Objet :	*Saisie des sites d'étape dans une région et détermination des itinéraires de liaisons entre sites.*		
Acteur(s) :	*Opérateur*		
Précondition(s) :	*Un fond cartographique est disponible (les cartes ont été numérisées)*		

Enchaînement nominal :

Les acteurs	Le système
1. Démarre l'application	*2. Ouvre la fenêtre d'édition et un formulaire pour définir la zone de travail*
3. Définit la zone de travail	*4. Charge le fond cartographique*
Pour tous les sites de la région	
5. Désigne sur la carte un site d'étape	*6. Affiche un symbole graphique et ouvre un formulaire de description du site*
7. Complète le formulaire et le valide	*8. Enregistre le site dans la base de données*
	9. Détermine automatiquement des routes de jonction avec les autres sites en fonction de la cartographie et calcule les distances entre sites en fonction de l'échelle de la carte
11. Sort de l'application	*10. Enregistre les routes de jonction dans la base de données*
	12. Ferme la fenêtre d'édition

Exceptions :

Jonction impossible	*Le système ne parvient pas à déterminer la route de jonction avec le site désigné*
Base de données saturée	*Espace disque insuffisant pour enregistrer les données dans la base*

Séquences alternatives :

S'il existe déjà des sites définis pour la région	*4. Les sites existants sont lus en base de données et affichés avec la cartographie*

Figure 13 – Description d'un cas d'utilisation

À ce stade, le système est considéré comme un tout et les interactions le mettent en jeu globalement. L'étape suivante va consister à identifier les objets du système qui sont nécessaires à la réalisation d'un cas d'utilisation ainsi décrit. Pour chaque cas d'utilisation, on va donc recenser les principaux scénarios et on va utiliser, pour les formaliser, une nouvelle catégorie de diagrammes UML appelés, de façon générique, **diagrammes d'interactions**.

7.2 Diagrammes d'interactions

UML définit deux types de diagrammes d'interactions : le **diagramme de séquence** et le **diagramme de collaboration**. De façon générale, ces diagrammes s'appliquent aux objets (*a priori* les mêmes que ceux qui sont identifiés dans les modèles de structure statique) et ils permettent de montrer comment les objets interagissent et échangent des messages pour exécuter un scénario particulier. Ces deux types de diagrammes ont des sémantiques très proches et permettent, en pratique, de modéliser les mêmes réalités selon des points de vues différents. Le diagramme de séquence privilégie la formalisation des aspects liés à la chronologie des enchaînements, tandis que le diagramme de collaboration s'attache plus à la structure et aux liens qui s'établissent entre les objets mis en jeu.

Dans les deux types de diagrammes, chaque objet est représenté par un rectangle avec les mêmes conventions que pour les

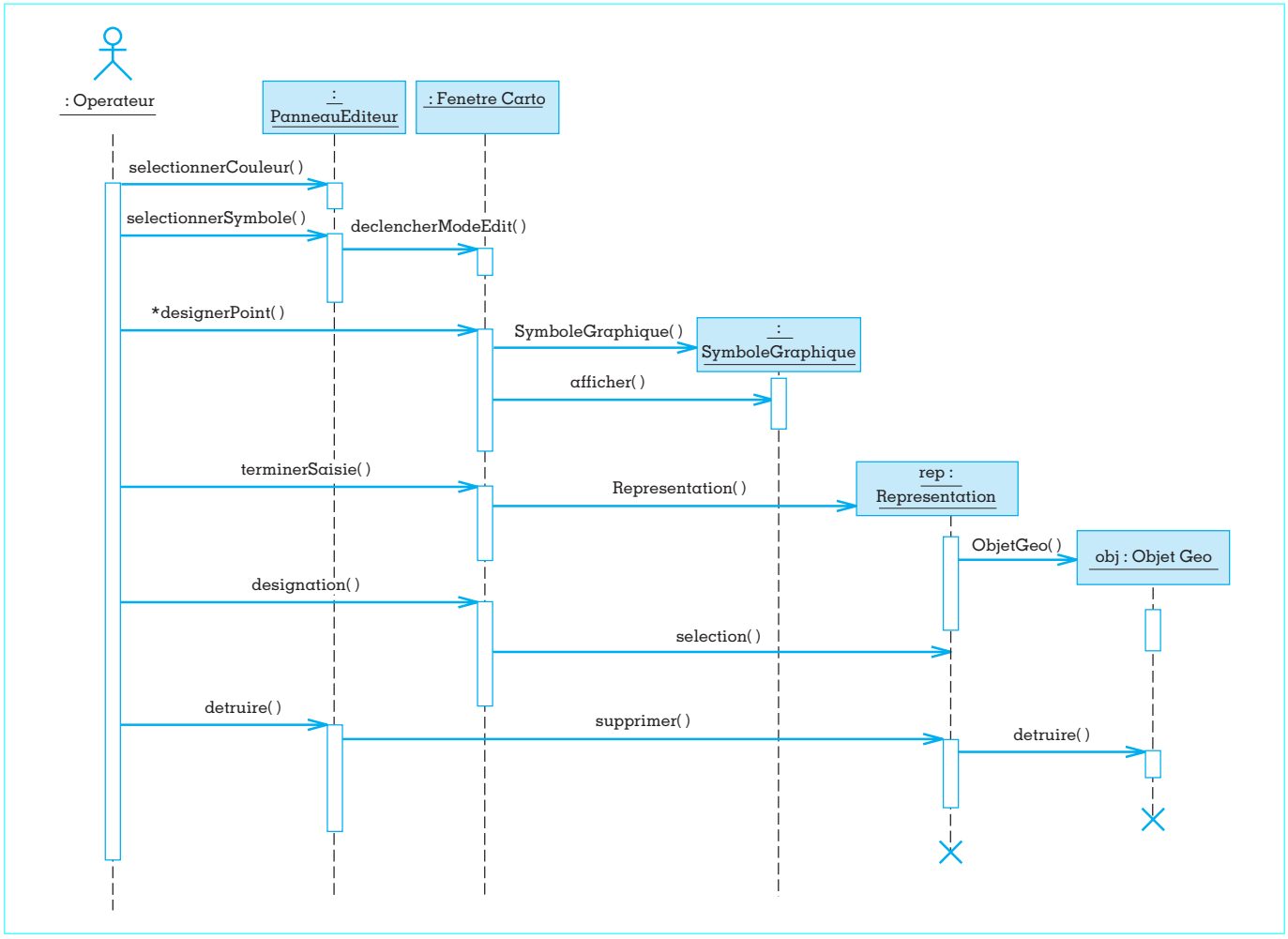

Figure 14 – Diagramme de séquence

diagrammes d'objets. Les acteurs sont représentés par des petits personnages (ou par des classes marquées <<Actor>>) comme dans les diagrammes de cas d'utilisation.

Pour chaque scénario, il faut donc établir une liste des objets participant aux enchaînements décrits. Ces objets sont, si possible, construits par instanciation des classes issues de l'étape précédente ou, à défaut, créés de toutes pièces, ce qui donne naissance à de nouvelles classes. Les diagrammes de classes sont alors complétés ou étendus et le modèle statique s'enrichit au fur et à mesure de l'analyse. La description des interactions entre objets conduit à l'identification des opérations mises en œuvre sur chaque objet et à l'enrichissement progressif des interfaces des classes.

7.2.1 Diagramme de séquence

Un diagramme de séquence se lit de haut en bas suivant une logique temporelle : le début du scénario est décrit en haut de page et la fin du scénario en bas de page. Les objets participants sont présentés en en-tête et sous chaque symbole rectangulaire, on trace un trait vertical symbolisant la **ligne de vie** de l'objet. Cette ligne permet de matérialiser l'existence de l'objet et le déroulement séquentiel des opérations qu'il exécute entre le moment de sa création et le moment de sa destruction. Pour modéliser les interactions entre acteurs et objets ou entre objets collaborants, on trace ensuite sur le même diagramme des flèches qui représentent les messages échangés pour déclencher les opérations nécessaires. Sur envoi d'un message, une méthode de l'objet destinataire est invoquée et une opération est exécutée. L'exécution de cette opération peut être représentée par un rectangle vertical superposé à la ligne de vie de l'objet qui la réalise.

Par défaut, l'appel est synchrone et le contrôle est implicitement restitué à l'appelant lorsque l'opération se termine. UML permet cependant, dans un diagramme de séquence, de modéliser, si besoin, des appels asynchrones (envois de messages symbolisés avec des flèches à une seule pointe et retours explicites), d'encadrer les opérations par des structures de contrôle (messages réflexifs, boucles de messages, etc.) et de spécifier des contraintes temporelles.

Sur l'exemple présenté sur la figure **14**, la séquence représente un scénario qui correspond à la création d'un objet graphique :

— l'acteur, c'est-à-dire en l'occurrence l'opérateur, déclenche des opérations sur les objets de l'interface homme-machine (panneau de l'éditeur et fenêtre cartographique) ;

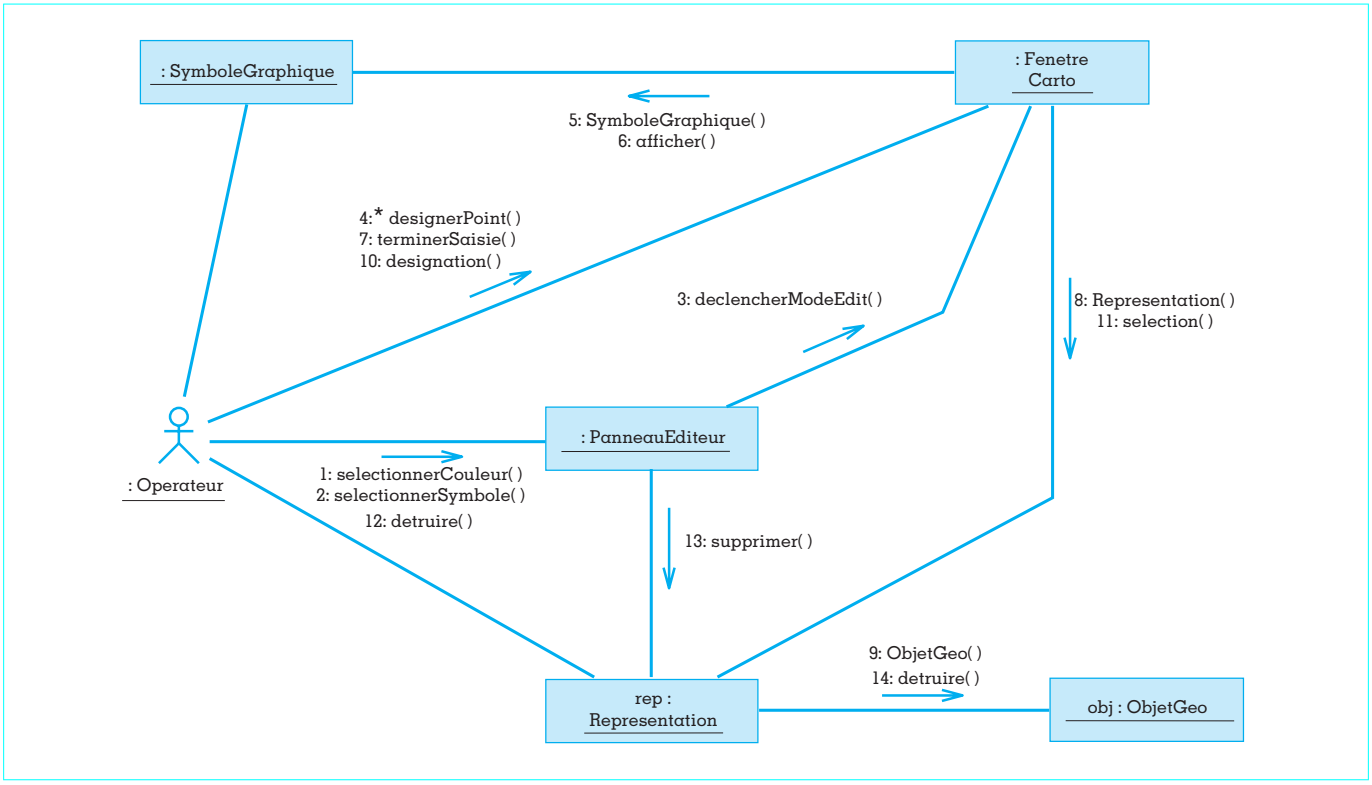

Figure 15 – Diagramme de collaboration

— les objets de l'interface répercutent les événements sur les objets définis par ailleurs dans les diagrammes de classes. On voit ici les lignes de vie des objets qui débutent lors de l'appel des constructeurs ;

— pour la destruction d'un objet, on fait apparaître une croix qui marque la fin de la ligne de vie.

Ce type de diagramme est particulièrement utile pour comprendre le fonctionnement dynamique de l'ensemble des objets et le rôle que chacun joue dans le processus global de mise en œuvre. Le diagramme de séquence peut également aider à la définition des interfaces d'objets, en particulier si l'élaboration du diagramme est assistée par un outil qui présente les opérations applicables à chaque objet lors de la spécification d'un message adressé à cet objet. Cela permet de compléter la définition de l'interface au fur et à mesure de la prise en compte et de la formalisation des scénarios.

Lors de la conception technique d'un logiciel, les diagrammes de séquence peuvent être également très utiles car ils permettent de formaliser un processus d'exécution avec la création et la destruction des objets. Ce type de représentation pourrait, par exemple, être très appréciable en phase de mise au point pour analyser l'exécution d'une séquence de code.

7.2.2 Diagramme de collaboration

Le diagramme de collaboration propose une représentation équivalente à celle des diagrammes de séquence. On vient plaquer les flux de messages entre objets sur le diagramme d'objets. Cela permet de vérifier que la structure du modèle de classe est satisfaisante et que les liens entre objets permettent de naviguer dans le modèle pour dérouler le scénario voulu. La séquence des opérations est représentée par un numéro d'ordre sur chaque flèche figurant un message (figure **15**).

L'un des principaux intérêts des cas d'utilisation et des diagrammes d'interactions est de permettre la modélisation d'un comportement global en s'attachant à la description du comportement de chacun des constituants mis en jeu. Cela est particulièrement appréciable :

— dans une approche d'analyse de système lorsque l'on veut représenter chacun des sous-systèmes pour décrire ses fonctionnalités, son rôle et ses interfaces avec les autres sous-systèmes ;

— lorsque l'on veut formaliser le fonctionnement d'un système à architecture distribuée faisant intervenir des « composants logiciels » mis en œuvre sur un réseau de machines reliées par un bus logiciel.

8. Modélisation dynamique

Les **diagrammes d'états** et les **diagrammes d'activités** permettent de décrire le comportement dynamique du système modélisé. Ils sont destinés à la représentation des processus qui régissent le fonctionnement du système et des objets qui contrôlent l'exécution des opérations.

8.1 Diagramme d'états

Les diagrammes d'états, ou diagrammes états-transitions, sont basés sur le concept de machine à états finis qui s'attache à la

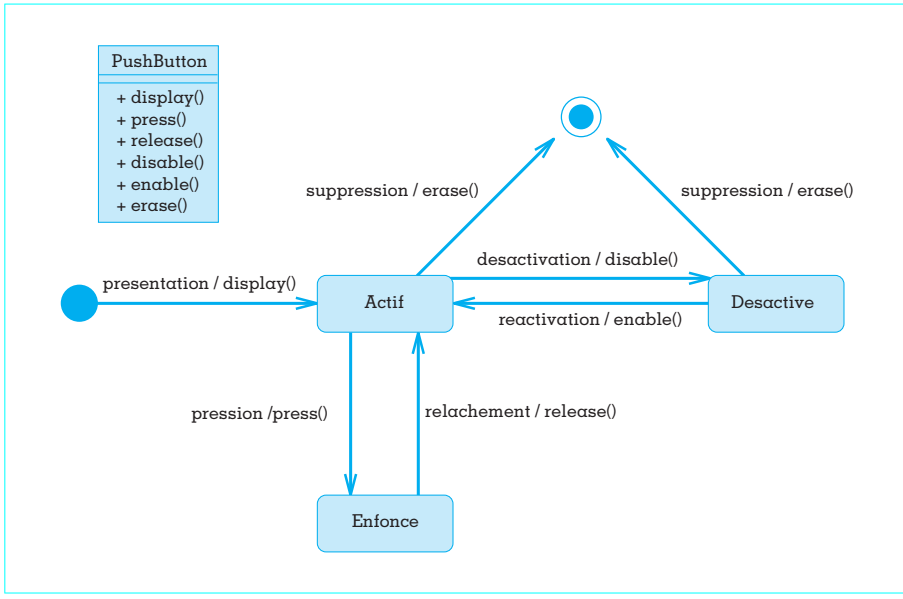

Figure 16 – Diagramme d'états

modélisation de processus complexes sous la forme d'automates. Ils offrent le moyen de formaliser le « cycle de vie » d'un système ou d'un objet, c'est-à-dire de décrire ses réactions en fonction de son état courant et des événements qui peuvent survenir durant sa mise en œuvre. Ces diagrammes sont généralement utilisés pour compléter la description des classes d'objets *actifs*. Ces classes, appelées classes actives en UML, se distinguent graphiquement des autres par un contour en trait gras dans les diagrammes de classes. Elles peuvent également être complétées par une description des événements et signaux auxquels sont soumis les objets. Dans ce cas, un troisième compartiment est ajouté sur la représentation en plus de ceux dédiés aux attributs et aux méthodes. C'est précisément pour expliciter les réactions des objets par rapport à ces événements et signaux que les diagrammes d'états sont utilisés.

Chaque objet actif correspond à un **processus indépendant** (*process* ou *thread*) et plusieurs objets actifs peuvent cohabiter au sein d'un même système, entraînant ainsi des problèmes potentiels de parallélisme et de concurrence. Le modèle doit alors pouvoir expliciter les règles de cohabitation et les éventuelles modalités de synchronisation entre les différents processus. Les diagrammes d'interactions offrent alors une première solution de représentation et les signaux transmis d'un processus à l'autre se traduisent, dans ce cas, par des messages échangés entre objets actifs. Pour compléter et préciser le modèle, UML propose de décrire par un *diagramme d'états* le comportement de chaque objet impliqué. Au contraire des diagrammes d'interactions qui ne traitent que quelques scénarios représentatifs mettant en jeu une collaboration d'objets, les diagrammes d'états offrent un moyen de formaliser, avec précision et exhaustivité, le comportement spécifique de chaque objet.

Un **état** modélise une situation particulière dans laquelle l'objet peut se trouver au cours de son existence. Cette situation se caractérise par la satisfaction de certaines conditions, par une activité en cours de réalisation ou encore par l'attente d'un événement. En UML, un état est représenté par un rectangle aux coins arrondis dans lequel figure un label caractéristique. L'automate décrit les changements d'états de l'objet en fonction des sollicitations qu'il traite et des opérations qu'il réalise. Un changement d'état se traduit par une **transition** représentée par une flèche allant de l'état courant vers le nouvel état. Les transitions sont déclenchées par :

— des signaux provenant d'autres objets ;

— l'appel d'une opération de l'objet ;

— l'occurrence d'un événement temporel (échéance, durée écoulée, etc.) ;

— un changement quelconque se traduisant par la satisfaction d'une expression conditionnelle.

Dans l'exemple présenté sur la figure **16**, on a modélisé les états d'un bouton poussoir : les transitions correspondent aux interventions de l'opérateur et les états correspondent à la position du bouton. Cet automate spécifie une sorte de protocole associé à l'interface de la classe. Transposé à l'interface homme-machine d'un système informatique, il permettra de gérer l'apparence du bouton à l'écran. Si un objet n'est pas dans l'état correspondant à l'origine d'une transition, alors il n'est pas possible d'invoquer la méthode correspondant à cette même transition. En l'occurrence, il n'est pas possible, par exemple, de réactiver un bouton déjà actif, d'enfoncer un bouton inactif ou encore de relâcher un bouton qui n'est pas enfoncé.

Les *états initiaux* sont notés par un disque noir plein et les *états terminaux* par un disque noir plein à l'intérieur d'un cercle. Sur un état particulier, on peut spécifier le déclenchement d'une opération correspondant à une activité qui se déroule tant que l'objet est dans l'état considéré. Sur le rectangle représentant l'état, on mentionne alors le nom de l'opération précédé du mot-clef do. De la même façon, on peut spécifier le déclenchement d'une opération lorsque l'automate entre dans un état particulier (mot-clef entry et nom de l'opération) ou lorsqu'il en sort (mot-clef exit et nom de l'opération). Enfin, il est également possible de spécifier que, dans un état donné, une opération doit être déclenchée sur occurrence d'un événement à préciser. Dans ce cas, la mention portée sur le symbole représentant l'état sera de la forme : on <identificateur de l'événement> : <identificateur de l'opération>.

Les diagrammes d'états sont généralement utilisés pour décrire le comportement des objets d'une classe mais ils peuvent aussi permettre de représenter un processus complexe relatif à un groupe d'objets. Il est également possible d'imbriquer des automates les uns dans les autres pour décrire des comportements de façon plus ou moins détaillée. L'automate principal inclut alors des états généraux qui se décomposent en un certain nombre de

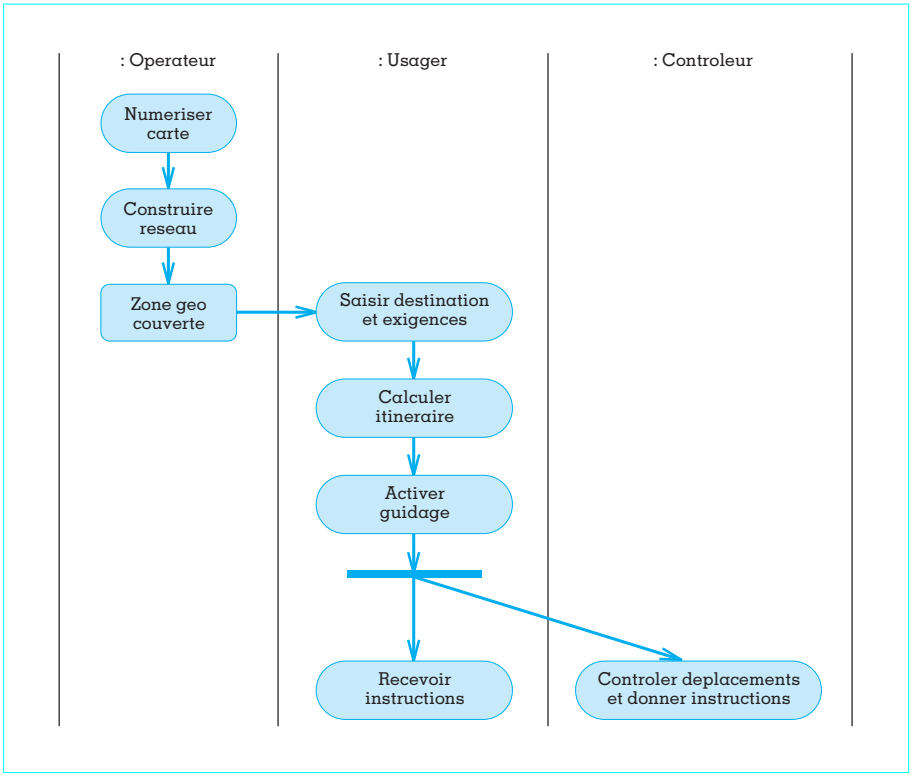

Figure 17 – Diagramme d'activités

sous-états à un niveau d'abstraction plus fin. Ce principe de structuration permet de hiérarchiser les états et de simplifier les diagrammes.

8.2 Diagramme d'activités

Les diagrammes d'activités permettent également de décrire la dynamique du système modélisé. On les utilise plutôt pour représenter la logique de traitement d'une méthode ou d'un cas d'utilisation. Ils représentent un déroulement d'étapes regroupées séquentiellement dans des branches parallèles de flot de contrôle. Il s'agit, en fait, d'une variante d'un automate où la priorité est donnée aux actions réalisées et non pas aux états résultants. On peut dire qu'il existe une sorte d'équivalence sémantique entre les diagrammes d'états et les diagrammes d'activités si l'on considère que la fin de la réalisation d'une activité correspond à un changement d'état entraînant le déclenchement d'une autre activité.

Les activités sont représentées par des formes géométriques composées de deux lignes horizontales jointes par des côtés arrondis. Ces formes encadrent les noms des activités. Chaque transition est représentée par une flèche qui peut :

— relier deux activités qui s'enchaînent séquentiellement ;

— relier une activité et une condition représentée par un losange quand l'enchaînement est régi par une alternative ;

— relier une activité à une barre de synchronisation lorsque plusieurs activités sont déclenchées simultanément donnant naissance à plusieurs flux de contrôle concurrents ou, au contraire, lorsque plusieurs flux concurrents sont unifiés.

Les activités peuvent être représentées dans des **couloirs** (*swimlines*) qui correspondent aux objets censés réaliser les opérations modélisées. UML permet également de spécifier des alternatives ou des contraintes de synchronisation entre activités. La figure **17** présente un exemple de diagramme d'activités.

9. Modélisation d'architecture

La structure de paquetages élaborée par raffinement progressif du modèle au travers des diagrammes de classes est représentative de l'**architecture statique** du système. Pour finaliser une conception de logiciel, il reste à définir les modalités de traduction du modèle dans le langage cible et la stratégie d'implémentation la mieux adaptée à l'environnement d'exécution. Si le langage choisi est un langage orienté objet, les règles de traduction du modèle UML sont presque implicites. Les choix d'implémentation portent donc le plus souvent sur une allocation des unités fonctionnelles du modèle (représentées par les classes et les paquetages) aux différents composants que l'on veut pouvoir déployer. Les décisions de conception doivent prendre en considération les exigences relatives à l'architecture physique du système. L'**architecture technique** ainsi élaborée peut être formalisée en utilisant les **diagrammes de composants**.

9.1 Diagramme de composants

Les diagrammes de composants permettent de représenter les éléments ou les articles de la configuration du logiciel et leurs liens de dépendance. Ils formalisent ainsi les solutions techniques choi-

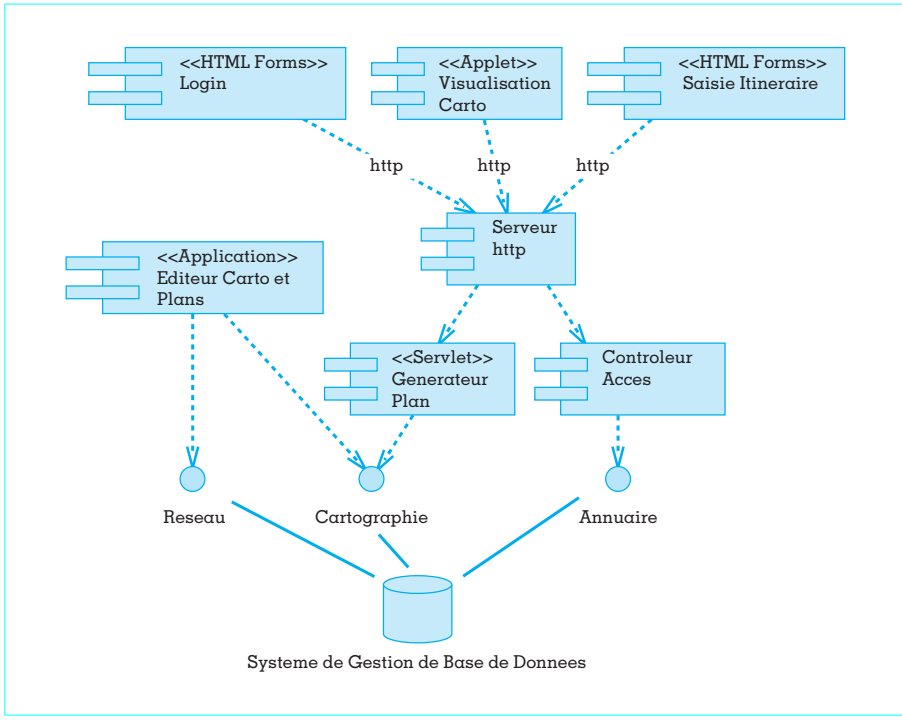

Figure 18 – Diagramme de composants

sies pour la réalisation et l'intégration du logiciel en faisant apparaître explicitement les programmes principaux, les bibliothèques, les modules, les « composants sur étagère » (COTS ou *components off the shelf*), etc., ainsi que la structure globale dans laquelle ils prennent place. Chaque composant traduit une réalisation particulière d'une ou de plusieurs entités préalablement modélisées dans des diagrammes de classes. La nature des composants est souvent très liée à la technologie choisie, à l'environnement d'exécution et au langage cible. Ainsi, on utilise cette représentation pour modéliser un *package* Ada, un composant CORBA, un active-X ou un Java Bean, une DLL ou une bibliothèque, etc.

Tout comme les classes et les cas d'utilisation, les composants peuvent être organisés en paquetages qui, dans ce cas, sont assimilés à des **sous-systèmes** logiciels. Chaque sous-système et chaque composant peut être représenté sur le diagramme de composants et caractérisé par une ou plusieurs **interfaces** qu'il met à la disposition des autres composants de l'architecture. UML permet de modéliser ces interfaces soit en utilisant une notation équivalente à celle de la classe avec la mention <<interface>> dans le compartiment supérieur, soit en utilisant un symbole spécifique qui prend la forme d'un petit cercle. Ce symbole peut figurer à la fois :

— sur les diagrammes de classes pour spécifier l'interface au travers des opérations offertes qui correspondent à des fonctionnalités ou des services mis en œuvre ;

— sur les diagrammes de composants pour matérialiser la connexion établie entre deux composants.

Le diagramme de la figure **18** représente en UML une architecture à base de composants en faisant apparaître explicitement les interdépendances entre composants au travers d'interfaces.

Pour vérifier la complétude de l'architecture technique ainsi définie et s'assurer que celle-ci permet bien de satisfaire les exigences fonctionnelles, il reste à démontrer que tous les composants requis ont bien été prévus et qu'ils offrent effectivement les services nécessaires. Des diagrammes d'interactions peuvent alors être utilisés pour décrire précisément les échanges qui interviennent entre sous-systèmes ou entre composants distribués à l'occasion d'un scénario de mise en œuvre opérationnel. Cette démarche et le modèle qui en résulte contribuent à la validation de l'architecture puisqu'ils permettent de s'assurer que tous les services requis sont alloués et que chacun des constituants mis en jeu est en mesure de jouer son rôle vis-à-vis des autres constituants en offrant une interface d'accès complète et cohérente.

9.2 Diagramme de déploiement

Le dernier type de diagrammes UML concerne le déploiement du système. Les **diagrammes de déploiement** montrent la disposition physique des différents matériels (les *nœuds*) qui composent le système et la répartition des programmes exécutables sur ces matériels. Ces diagrammes peuvent présenter un intérêt pour la modélisation d'architectures à objets distribués puisqu'ils permettent de formaliser la répartition des composants sur le réseau de calculateurs cibles.

Les diagrammes de déploiement permettent de représenter les nœuds de l'architecture physique sous forme de cubes reliés les uns aux autres par des traits simples qui correspondent aux liens de communication physique (bus matériel, réseau de télécommunication, etc.). Les nœuds peuvent être des processeurs sur lesquels s'exécutent les tâches identifiées dans l'architecture du logiciel) ou des ressources particulières qui correspondent aux éléments matériels mis en œuvre dans l'architecture (capteurs, équipements, périphériques, etc.). Les tâches exécutées sur un processeur sont listées sous la face frontale du cube représentant le processeur considéré. La figure **19** présente un exemple de diagramme de déploiement.

Pour plus de clarté, les ressources matérielles peuvent être représentées graphiquement par des symboles visuels qui restituent, de façon plus ou moins schématique, leurs apparences physiques.

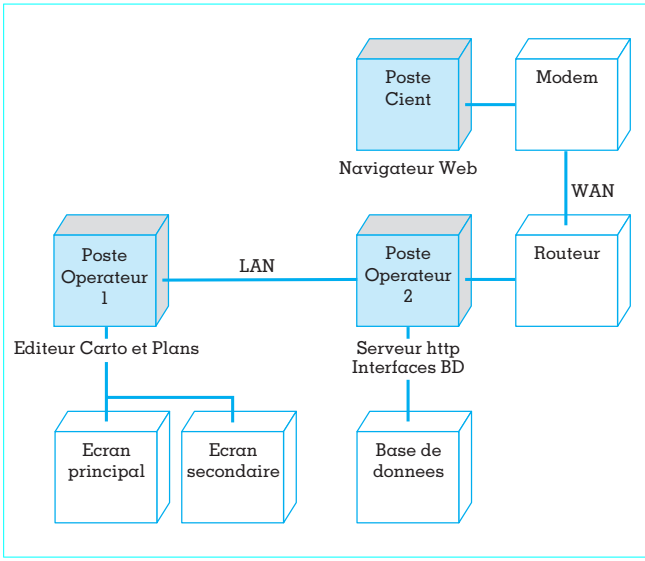

Figure 19 – Diagramme de déploiement

10. Extensibilité et ouverture

Les pages qui précèdent ne décrivent que quelques-uns des concepts figurant dans le **métamodèle UML** et l'article s'attache avant tout à introduire les notions fondamentales du langage en présentant le formalisme graphique sur lequel il repose et en illustrant celui-ci par quelques exemples simples des différents types de diagrammes existants. Notre propos n'est pas ici de retranscrire intégralement la norme publiée à ce jour, celle-ci étant beaucoup trop riche pour être résumée en quelques dizaines de pages. UML unifie et englobe différentes techniques de modélisation qui ont fait leurs preuves dans le domaine informatique et qui permettent de représenter, de façon claire et précise, la plupart des artefacts du développement de logiciel. Cependant, malgré la puissance du langage, la capacité d'expression d'UML reste bien sûr limitée et la notation n'est pas toujours adaptée à la formalisation précise et non ambiguë de phénomènes particuliers. Aussi les concepteurs d'UML ont-ils prévu des mécanismes permettant d'étendre le langage pour en augmenter la portée ou pour traiter des aspects spécifiques d'un domaine donné.

Pour ce faire, UML introduit la notion de **stéréotype** qui permet d'adjoindre au modèle de base des concepts complémentaires pouvant présenter un intérêt dans un contexte particulier. Un stéréotype peut être considéré comme une extension du langage : il élargit le vocabulaire pour traduire une réalité ou un concept nouveau. Il ajoute un nouvel élément de modélisation dans le métamodèle UML en dérivant l'un des éléments standards. Le symbole utilisé pour représenter ce nouvel élément peut être créé de toutes pièces et intégré à la notation UML mais la norme permet d'utiliser, plus simplement, le symbole de l'élément dérivé en le désignant par un marqueur encadré par des guillemets (par exemple << mon stereotype >>).

La figure **20** montre les différentes représentations possibles d'un stéréotype de classe dédié à la modélisation d'une fenêtre d'interface homme-machine.

Ce mécanisme va donc permettre de compléter le métamodèle UML en fonction du besoin engendré par les spécificités du sujet abordé ou par la mise en œuvre de techniques particulières. La norme UML n'est donc pas restrictive en ce qui concerne le formalisme et elle autorise les utilisateurs à « personnaliser » la notation. On peut cependant noter que le modèle de référence contient déjà un certain nombre stéréotypes « standards » dont nous avons déjà rencontrés quelques exemples précédemment. Par exemple, les acteurs, les interfaces, les types, les processus ou encore les signaux sont définis comme des stéréotypes de classes pour lesquels UML réserve les mots-clefs <<actor>>, <<interface>>, <<type>>, <<process>>, <<signal>>. Les dépendances peuvent être stéréotypées <<import>>, <<friend>>, <<bind>>, etc., tandis que les fichiers (<<file>>), les bibliothèques (<<library>>), les tables de bases de données (<<table>>) sont des stéréotypes de composants.

UML permet également d'attacher à certains éléments du modèle des indications ou des caractéristiques spécifiques qui doivent influencer l'utilisation ou le traitement qui leur seront appliqués. Ces propriétés, appelées **étiquettes** ou *tag values*, prennent des valeurs particulières en fonction des éléments de modélisation. Elles figurent sur les diagrammes entre accolades et sont juxtaposées aux éléments auxquels elles s'appliquent. Comme pour les stéréotypes, UML propose un certain nombre d'étiquettes standards prédéfinies : persistance permet de préciser si les objets d'une classe sont transitoires ou persistants, documentation permet d'ajouter un commentaire ou une description textuelle, etc.

Les étiquettes peuvent être considérées comme des attributs qui viennent compléter la caractérisation d'un élément du métamodèle. Si l'on veut préciser, pour chaque classe, par exemple, le langage cible, l'état d'avancement de son codage ou encore le nom du responsable de son implémentation, on peut le faire grâce à une étiquette. Les étiquettes gardent un côté formel puisqu'elles permettent d'aller jusqu'à la génération automatique de code ou de documentation. Elles sont intégrées dans le métamodèle de référence utilisé et peuvent être outillées, c'est-à-dire reportées au niveau du référentiel de l'atelier de modélisation UML pour que celui-ci sache les traiter de façon adaptée.

UML permet enfin d'ajouter des **contraintes** sur tous les éléments du modèle et en particulier sur les relations comme nous l'avons vu au paragraphe 4. Pour formaliser ces contraintes, l'OMG a retenu un langage développé par IBM et associé à UML : Object Constraint Language (OCL). OCL est un langage formel et il peut être interprété par des outils pour vérifier la cohérence du modèle et l'intégrité des objets. Il permet, entre autres, de décrire des invariants de classes, des pré- et postconditions pour les opérations, des instructions de navigation dans le modèle, des règles de gestion sous forme d'expressions booléennes, etc. Les contraintes OCL contribuent à préciser et à expliciter les éléments de modélisation de manière normalisée. Une expression OCL peut être insérée dans une annotation de diagramme. Notons également que OCL est largement utilisé dans la documentation de référence UML pour décrire les contraintes applicables au métamodèle.

Depuis quelques années, l'OMG a mis en place un groupe de travail nommé « Domain Task Forces » dont la mission est de proposer des modèles et des architectures adaptés à des domaines d'application ciblés. Certains domaines métier tels que la comptabilité, la finance, le commerce électronique ou encore les assurances se caractérisent par des objets et des processus qui leur sont propres. Ceux-ci peuvent être modélisés de façon générique et les modèles résultant doivent pouvoir être, eux aussi, normalisés. C'est le pari des industriels qui contribuent à ce groupe de travail dans le but d'améliorer l'interopérabilité des applications. L'ambition de ce projet va même jusqu'à la définition d'architectures précâblées et réutilisables. Les premiers résultats de ces travaux prennent la forme de documents normatifs publiés par l'OMG sous le nom de « UML Profiles »

Les profils UML regroupent et organisent des extensions du métamodèle définies sous forme de stéréotypes, d'étiquettes et de contraintes et s'appliquant plus spécifiquement à un domaine d'application donné. L'OMG propose déjà aujourd'hui des exemples de profils dédiés aux applications en temps réel, aux bases de données, à CORBA ou encore au langage C++. Tous les éléments

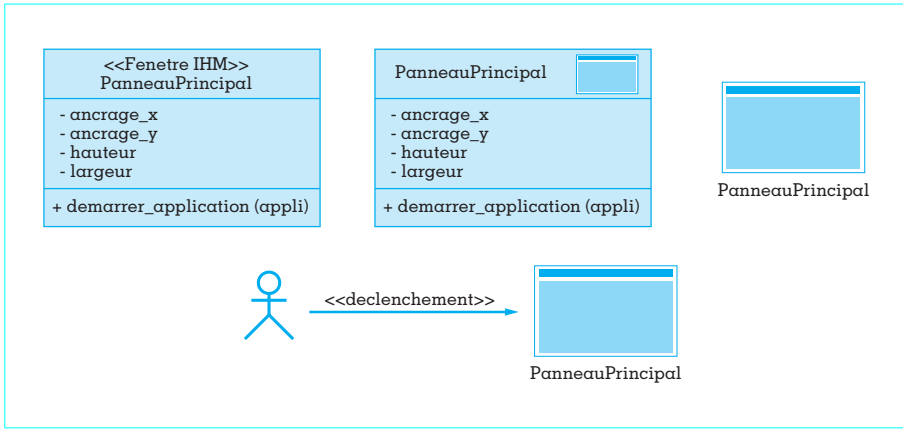

Figure 20 – Stéréotypes

d'UML ne sont pas pertinents pour un domaine donné et les profils ont donc aussi pour objet de restreindre et de simplifier la notation pour la rendre plus efficace. Les profils fournissent également les règles à suivre pour assurer la validation des modèles et pour les exploiter par transformations successives jusqu'à la production de code suivant des *design patterns*.

L'autre grand chantier de l'OMG a pour nom MDA (Model Driven Architecture). Sa motivation part du constat que seuls les modèles métier peuvent permettre de capitaliser les développements. L'évolution incessante des plates-formes et des technologies de développement entraîne inexorablement la remise en cause permanente des architectures techniques. L'approche de modélisation préconisée par MDA consiste à définir un modèle métier totalement indépendant de toute architecture technique : le PIM (Platform Independent Model). Ce modèle est ensuite instrumenté et il subit un certain nombre de transformations dictées par l'approche MDA pour prendre en compte les spécificités de la plate-forme cible. Il en résulte un nouveau modèle, le PSM (Platform Specific Model), qui permet de générer automatiquement le code des composants à distribuer. Ainsi, par exemple, le passage d'une architecture CORBA à une architecture .NET se résume à l'écriture de quelques règles de transformation et à la régénération du code à partir du PIM.

11. Outils

Dans les précédents paragraphes et au travers des exemples de diagrammes présentés, nous avons pu mettre en évidence tous les avantages de la modélisation visuelle, en particulier sur le plan de la communication. Un moyen d'expression tel qu'UML devient indispensable dès lors qu'il s'agit de développer de grands logiciels en équipe et en utilisant des technologies orientées objets. La maîtrise de cette technique donne au concepteur la faculté de pouvoir « dessiner » la solution qu'il préconise et de se faire comprendre rapidement sans avoir recours à de longs discours qui peuvent être source de malentendus ou d'incompréhensions. Dans les équipes ayant adopté UML, les diagrammes constituent un formidable support d'échange et de dialogue entre les différents intervenants du projet. Face à un problème de conception, les solutions techniques et les bases de leur programmation sont ébauchées en quelques coups de crayon et peuvent ainsi être discutées avant adoption par tous les protagonistes.

Mais les diagrammes doivent aussi pouvoir vivre et évoluer en fonction des besoins nouveaux qui apparaissent durant le projet. L'intégration d'une nouvelle fonctionnalité, la prise en compte d'un cas particulier initialement négligé, l'ajout d'un détail oublié, l'optimisation des programmes dans le but de gagner en efficacité sont autant de raisons qui peuvent nécessiter une reprise des diagrammes. Pour assurer la mise à jour des diagrammes et enrichir, préciser ou corriger le modèle au fil du projet, des moyens d'édition évolués sont indispensables. L'**édition graphique des diagrammes** et la **gestion de cohérence globale du modèle** sont donc les deux premières fonctionnalités requises pour un **atelier de modélisation en UML**.

En phase d'analyse ou de conception, un outil de modélisation visuel doit donc, avant tout :

— permettre l'élaboration rapide des différents types de diagrammes proposés par UML ;

— gérer, en tant qu'entités propres, les éléments de modélisation représentés sur les diagrammes ;

— contrôler la cohérence globale du modèle en maintenant à jour les liens entre le modèle et les diagrammes, c'est-à-dire entre les éléments de sémantique référencés de façon unique et leurs diverses représentations graphiques ;

— offrir les moyens de compléter les diagrammes par l'expression plus ou moins formelle de contraintes et par une documentation en langage naturel, le tout étant géré en liaison directe avec les éléments de modélisation concernés.

À ce niveau, les critères d'appréciation d'un tel outil portent essentiellement sur son ergonomie, sur sa conformité par rapport aux versions les plus récentes de la norme UML et sur le support des neuf types de diagrammes présentés dans cet article. La disponibilité de fonctions de consultation permettant de rechercher, de trier les éléments du modèle et de naviguer dans la structure de paquetages peut devenir essentielle si la taille du modèle et le nombre d'éléments qu'il intègre deviennent importants. L'outil peut également prendre en charge ou faciliter la **production** et la **publication d'une documentation technique** où figurent tout ou partie du modèle et des diagrammes. Bien que l'intérêt réel de cette dernière fonctionnalité ne soit pas avéré, elle peut être utile si le cadre contractuel du projet prévoit la livraison d'une documentation selon un formalisme imposé. Dans ce cas, l'outil peut décharger le concepteur de la mise en forme des documents.

Si le modèle va jusqu'à la représentation de la structure des programmes et qu'il intègre des éléments pouvant être traduits automatiquement dans la syntaxe du langage cible, la **génération de code** peut permettre d'envisager des gains de productivité appréciables. Certains programmeurs qui pratiquent la modélisation UML considèrent même que cette fonctionnalité est indispensable puisque le modèle représente des éléments qui correspondent aux unités syntaxiques du langage de programmation. Il serait donc vraiment dommage de ne pas l'exploiter pour produire automatiquement le code.

De nombreux outils permettent aujourd'hui, à partir des classes modélisées en UML, de générer des squelettes de programmes en C++, Java, Visual Basic, Smalltalk, Ada ou d'autres langages moins répandus. Certains sont aussi capables de générer du code SQL pour la définition d'un schéma de base de données, des descriptions d'interfaces en IDL CORBA ou encore des métadonnées XML. Quelques ateliers dédiés au développement de logiciel supportant des contraintes temps réel proposent également la génération de code exécutable à partir des diagrammes d'états.

Les outils les plus puissants assurent une **génération incrémentale** qui permet de mettre à jour le code produit chaque fois que le modèle évolue. Quelques-uns proposent même des couplages avec des environnements de développement intégrés (IDE), ce qui permet au programmeur d'intervenir simultanément sur les diagrammes UML et sur le code source des programmes avec un maximum de confort et d'efficacité. Enfin, les outils les plus sophistiqués proposent des fonctions de rétroconception qui analysent le code source des programmes existants pour reconstruire automatiquement un modèle et les diagrammes UML associés. L'utilisation de tels outils contribue à minimiser la frontière entre les activités de modélisation et de programmation et UML peut presque, dans certains cas et pour certains aspects, se substituer au langage de programmation.

Les meilleurs outils donnent de multiples possibilités en ce qui concerne les méthodes de développement mises en œuvre dans un projet. Dans le cas d'un projet de grande taille, ils peuvent permettre d'élaborer le modèle en équipe. Ils assurent alors le **contrôle du partage**, la **gestion des versions**, voire même la fusion des différentes parties du modèle qui ont été développées en parallèle.

Presque tous les éditeurs qui commercialisent ou distribuent des outils de modélisation visuelle ont aujourd'hui choisi d'adhérer au projet UML et il existe un très grand nombre de produits disponibles au catalogue des ateliers supportant UML. Pour faciliter l'interopérabilité des différents outils du marché et permettre ainsi à la communauté des utilisateurs d'UML de coopérer quels que soient les produits qu'ils ont choisis, l'OMG a décidé de développer et de promouvoir une norme de métamodèle et un format dédié aux échanges de modèles entre outils. Le MOF (Meta Object Facility) est né d'une généralisation du métamodèle UML, celui-ci n'étant plus aujourd'hui qu'un élément parmi d'autres. Le MOF se consacre donc à la normalisation des métamodèles et à l'interopérabilité d'outils supportant des techniques qui peuvent éventuellement venir compléter UML. XMI (XML Metadata Interchange) est un format d'échange d'informations de modélisation. Il a pour vocation de remplacer les formats propriétaires des outils UML et de permettre ainsi la coopération d'équipes travaillant avec des outils provenant de différents fournisseurs.