

# UML

## Unified Modeling Language

---

L'approche objet	1
Présentation d'UML	5
Vue des cas d'utilisation	9
Vue logique	15



*Votre avenir  
nous engage*

Champs

*Notes de cours de JC Rigal du 27/03/2009 pour APDSI*

*Le sage réfléchit avant d'agir*

Proverbe persan

# L'APPROCHE OBJET

## NOTION D'OBJET

L'approche objet désigne l'ensemble des méthodes et langages utilisés au cours du cycle de vie du logiciel qui reposent sur la manipulation des objets. L'OBJET est une modélisation<sup>1</sup> d'un objet du monde réel par une entité informatique. Il est toujours vu comme une entité encapsulée, parfois représenté par un module, dont la structure interne est masquée. Nous en aurons une vision:

- **Externe** définissant les actions qu'il sera possible de lui faire subir,
- **Interne** dans laquelle seule sa structure sera considérée.

Un logiciel est vu comme un ensemble d'objets qui coopèrent. Un autre concept qui est très souvent lié à l'approche objet est celui de CLASSE qui permet de regrouper les propriétés communes des objets. C'est un modèle structurel d'objet à partir duquel il est possible de fabriquer<sup>2</sup> autant d'objet qu'il sera nécessaire. Ce concept est associé à la notion de composant logiciel réutilisable facilitant la production des logiciels et diminuant l'effort de tests.

## LES GRANDS PRINCIPES DE GENIE LOGICIEL

L'approche objet permet de répondre aux préoccupations des analystes et des concepteurs en leur offrant une démarche homogène tout au long du cycle de développement. Quelles que soient les méthodes mises en oeuvre à chacune des étapes, l'objectif final à atteindre est de produire un logiciel qui corresponde à des besoins réels et qui soit facilement maintenable. C'est à dire qu'il doit être:

CORRECT	En répondant exactement aux besoins des utilisateurs
ROBUSTE	En réagissant de façon prévisible à toutes les situations
MAINTENABLE	En acceptant les modifications de spécification (simplification ou enrichissement par rapport aux besoins initiaux) et en permettant facilement de localiser et corriger les anomalies
REUTILISABLE	En étant apte à être réutilisé dans un autre contexte afin de diminuer les coûts de production et d'améliorer la fiabilité
PERFORMANT	Vis à vis du temps et de l'espace mémoire

L'approche objet va permettre de respecter ces objectifs car elle offre un ensemble de concepts reconnus, permettant de garantir une production de logiciels de qualité:

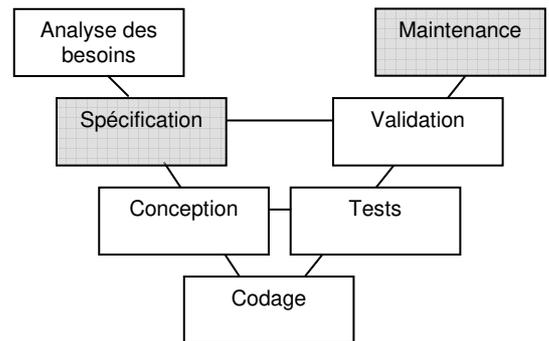
---

<sup>1</sup> On dit aussi abstraction

<sup>2</sup> Le terme instancier est souvent utilisé

## MOTIVATION DE L'APPROCHE OBJET

On constate que parmi les étapes du cycle de vie, celle correspondant à la spécification est la plus longue, environ 50 à 60% du temps développement. De plus, de récents rapports font état que 40 à 70% du budget informatique des sociétés est consacré à la maintenance des logiciels. A toutes les étapes du cycle de vie il est impératif de mettre en œuvre une démarche cohérente qui prenne en compte ces aspects importants.



Fort de cette motivation, l'approche objet est apparue progressivement afin de mettre entre les mains du programmeur une méthodologie<sup>3</sup> cohérente lui permettant:

- de décomposer l'application en modules les plus autonomes possibles et compréhensibles individuellement. Ceci favorise au maximum la réutilisation de parties de logiciel et diminue le temps passé en test
- de construire des modules dont la structure est masquée, pour que la conception ne soit plus basée sur des détails de bas niveau. ceci facilite le portage des applications tout en tenant compte des évolutions des OS et de l'architecture du matériel
- de définir des modules présentant un niveau d'abstraction suffisamment grand pour modéliser au mieux les situations du monde réel.

## UN PEU D'HISTOIRE

La plus importante contribution à l'approche objet provient très certainement des langages de programmation tels que Simula et Smalltalk, qui ont introduit à la fin des années 70, le concept d'objet par réaction à l'approche impérative, utilisée à l'époque dans des langages comme Fortran, Cobol, Algol et bien d'autres. Smalltalk qui a vu le jour dans les laboratoires de Rank Xerox (Paolo Alto), repose entièrement sur la notion d'objet et est intégré dans un environnement de développement interactif (menus déroulant, fenêtres,...). Il hérite de Simula 67 qui était, comme son nom l'indique, un langage de simulation introduisant les concepts d'encapsulation et d'héritage, permettant pour la première fois la modélisation des situations du monde réel. A partir des différentes versions de Smalltalk ont été définis un grand nombre de langages de programmation basés sur le concept d'objet (C++, Objective C, Object Pascal, Eiffel. et plus récemment Java).

A la fin des années 70 les méthodes de conception utilisées reposaient sur une approche structurée descendante. Ces méthodes, conçues à partir des travaux de Dijkstra, Yourdon, Constantine et bien d'autres, orientées traitements, étaient bien adaptées à la conception d'applications écrites avec les langages impératifs de l'époque. La taille et la complexité des logiciels évoluant, ces méthodes devinrent insuffisantes. C'est à la suite des travaux de Parnas, qui a introduit le concept de module, de Liskov et Guttag sur les types abstraits de donnée puis d'une équipe française dirigée par Galinier et Mathis sur les machines abstraites qu'a été formalisée, au début des années 80 la populaire méthode de Booch. Cette méthode peut être considérée comme une approche très pédagogique de la conception orientée objet. Elle a permis d'introduire la culture objet dans l'entreprise. De nombreuses autres méthodes ont été mises au point à partir des travaux de Booch parmi lesquelles Mach2, Hood, OOD, Mac-Adam ...

Plus tard, à la fin des années 80, héritant des travaux de Shlaer et Mellor sur le modèle sémantique de donnée, est apparue l'analyse orientée objet permettant enfin d'utiliser le concept d'objet pendant la phase de spécification.

<sup>3</sup> Ensemble de méthodes utilisées tous le long du cycle de vie

A partir des années 90 l'approche objet s'est révélée, être en informatique un concept fédérateur, non seulement pour les langages de programmation et les méthodes de conception et de spécification mais aussi pour: la conception d'interfaces utilisateur, les bases de données, les bases de connaissances. Dans le milieu de la décennie on comptait une cinquantaine de méthodes recouvrant en totalité ou partiellement les phases d'analyse (spécification et conception) du cycle de vie du logiciel. Trois méthodes ont eu un impact fort sur la communauté informatique:

1. La méthode de Grady Booch, dite Booch'93
2. La méthode de Jim Rumbaugh OMT-2 (Object Modeling Technique)
3. La méthode d'Ivar Jacobson OOSE (Object Oriented Software Engineering)

Les concepts et formalismes de ces trois méthodes ont été mis en commun dès 1995 pour former une méthode unifiée baptisée UML (Unified Modeling Language for Object-Oriented Development). La version 1.0 de la méthode a été formalisée par un consortium d'entreprises parmi lesquelles DEC, HP, IBM, Microsoft, Oracle et d'autres. La version 1.1 a été normalisée par l'OMG (Object Management Group) en 1997. La version UML 2.0 est prévue pour 2002

Booch, Rumbaugh et Jacobson sont souvent considérés comme les "pères fondateurs" d'UML. Parfois même, on les appelle les "3 Amigos".

## BIBLIOGRAPHIE

Les informations contenues dans ce document sont extraites des ouvrages suivant

Modélisation UML avec Rational Rose 2000  
de Terry Quatrani Eyrolles, Editions 2000

( Ouvrage de référence)

Modélisation Objet avec UML  
de Pierre-Alain Muller Eyrolles, Editions 2000

Journal of Object-Oriented Programming  
Vol 13, No 12 d'avril 2001



# PRESENTATION D'UML

## UML, C'EST QUOI

UML n'est pas une méthode. Elle formalise en fait, un ensemble de concepts et de diagrammes pouvant être utilisés durant toutes les phases d'analyse du cycle de vie du logiciel (définition des besoins, spécification, conception générale et conception détaillée).

Elle permet de représenter le système à analyser sous forme d'un ou plusieurs modèles permettant de se focaliser sur un aspect particulier du développement parmi lesquels:

1. Le modèle des cas d'utilisation qui décrit les besoins des utilisateurs,
2. Le modèle des classes qui définit l'architecture du système,
3. Le modèle des états qui se focalise sur le comportement dynamique,

Chaque modèle peut être étudié grâce à plusieurs diagrammes tels que:

1. Les diagrammes des cas d'utilisation
2. Les diagrammes de classes
3. Les diagrammes de séquence et de collaboration

UML n'est pas propriétaire, elle est accessible à toute entreprise qui peut librement en faire usage. En fonction du type d'application, des compétences ou des contraintes imposées à l'équipe de développement, le processus d'analyse peut-être différent et l'accent peut être mis sur tel ou tel autre modèle.

Certains fabricants offrent des ateliers, supportant UML, qui permettent l'élaboration des diagrammes et assurent la cohérence entre les modèles. Des générateurs de code Java, VB ou C++ permettent de passer aisément de l'analyse au codage. Certains intègrent même des outils de "reverse engineering" pour maintenir la cohérence entre le code et la conception.

Parmi les offres du marché les principaux sont:

- RATIONAL avec Rational Rose 2001 [www.rational.com](http://www.rational.com)
- SOFTEAM avec Objecteering UML Modeler 5.1 [www.softteam.com](http://www.softteam.com)
- MICROSOFT avec Visual Modeler pour VB [www.microsoft.com](http://www.microsoft.com)
- ... etc ...

Chaque atelier supportant UML propose un processus d'analyse. Un processus peut être assimilé à une méthode de développement logiciel par le fait qu'il propose une démarche et une chronologie dans l'analyse. Par exemple Rational utilise RUP (*Rational Unified Process*). D'autres entreprises proposent un processus de développement propriétaire basé sur UML, comme Communications & Systems avec *UML/CS SI Development Process*.

## OBJECTIFS DU DOCUMENT

Le processus de développement logiciel qui va être partiellement étudié dans la suite du document s'inspire librement de RUP et UML/CS. Son objectif est de présenter les principaux diagrammes UML à travers un processus simplifié permettant de formaliser simplement les **contraintes d'environnement** et les **contraintes fonctionnelles** du logiciel à développer. L'**architecture** du logiciel est mise en évidence par une décomposition de celui-ci en classes. Les documents ont été produits avec l'atelier Rational Rose 2000 Entreprise Edition.

## ETUDE DE CAS

Les exemples de ce document seront extraits d'un ensemble de diagrammes UML correspondant à une analyse d'un logiciel de gestion d'un terminal bancaire dont voici un extrait du cahier des charges



*Le terminal bancaire effectue des lectures régulières de cartes tant qu'il est en service. Le terminal accepte les cartes d'agence bancaires et les cartes de crédit. Une carte d'agence bancaire ne permet pas le retrait d'argent mais autorise simplement le client de la banque, à obtenir un relevé de compte. Elle ne possède pas de code confidentiel. Les cartes de crédit autorisent toutes les opérations. Une carte peut-être rejetée si elle émane d'une autre banque (carte d'agence bancaire uniquement) ou si elle est illisible. Si elle est acceptée, elle fait l'objet d'un contrôle de validité au niveau de la date de péremption. En cas de dépassement de validité, la carte est avalée. Une carte valide passe à la vérification du code confidentiel (codé dans la carte). Si l'utilisateur s'est trompé, le code confidentiel sera alors relu. Il a droit à 3 essais successifs pour fournir son code. Passée cette limite, la carte est avalée. Au cours des 3 essais l'utilisateur peut annuler l'opération. Dans ce cas, la carte lui sera rendue. Lorsque le code secret est correct il choisit éventuellement son type d'opération. Il peut effectuer soit une demande de relevé de compte (Franc ou Euro), soit un retrait d'espèces en indiquant la somme voulue. Les billets ne seront distribués que si l'état du compte le permet. L'état du compte est obtenu par interrogation du centre de traitement.*

*Une fois les opérations effectuées, l'utilisateur pourra récupérer sa carte. Le système passera alors en attente d'une nouvelle carte.*

## DEMARCHE GENERALE

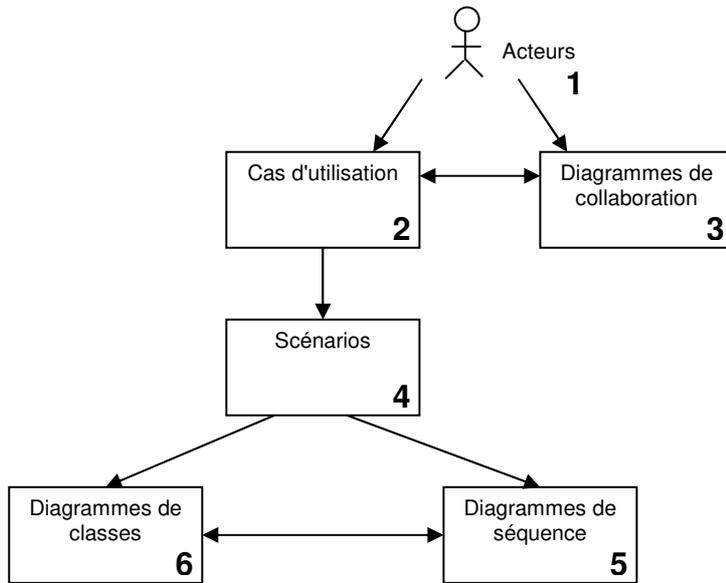
La démarche proposée est parfois appelée "Approche dirigée par les cas". Elle consiste en:

### Vue des cas d'utilisation (Use Case View)

1. Identification des **acteurs** actifs et passifs qui agissent sur le système à étudier
2. Définition et description des **cas d'utilisations** qui montreront les interactions entre les acteurs actifs externes et le système vu comme une boîte noire,
3. Construction de **diagrammes de collaboration** entre les acteurs passifs externe et le système afin de visualiser les échanges d'information,
4. Pour chaque cas d'utilisation seront décrits les principaux **scénarios** qui visualisent la chronologie des échanges entre les acteurs et le système,

### Vue logique (Logical View)

5. A chaque scénario sera associé un **diagramme de séquence** qui mettra en avant l'aspect temporel des interactions entre les objets de l'application qui participent au scénario,
6. En parallèle avec les diagrammes précédent, seront construits les **diagrammes de classes** qui visualisent l'architecture du logiciel, la hiérarchie et les associations entre classes



Démarche dirigée par les cas

Une autre démarche est possible mais ne sera pas étudiée dans ce document, c'est "L'approche dirigée par les classes" où sont d'abord identifiées les classes de l'application puis les cas d'utilisation

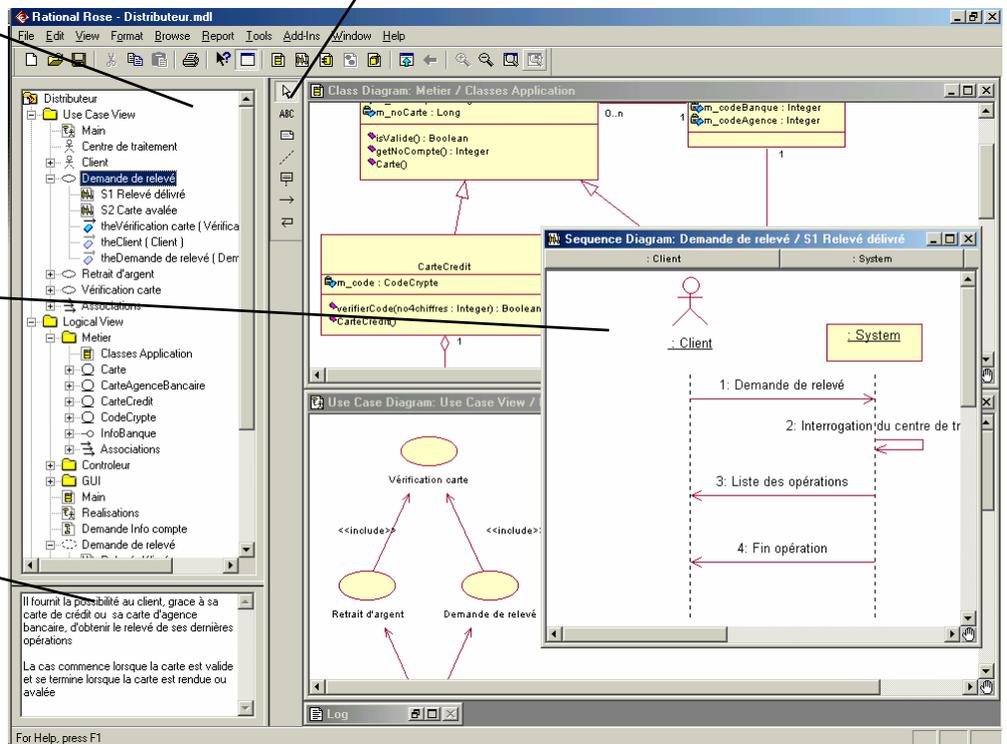
## L'ATELIER RATIONAL ROSE

**Navigateur**  
Il permet de sélectionner les entités du projet (acteur, cas, classe, diagramme, ...) dans les 2 dossiers importants (Use Case View et Logical View)

**Editeur graphique**  
Il permet de construire les diagrammes, une fenêtre pour chacun (Use Case Diagram, Sequence Diagram, Classes Diagram ...)

**Fenêtre de documentation**  
Elle permet de décrire les entités du projet (acteur, cas, classe, diagramme, ...) sous forme d'un texte libre

**Barre d'outils**  
Différente suivant le type de diagramme





# VUE DES CAS D'UTILISATION

## OBJECTIFS

La vue des cas d'utilisation ( *Use Case View*) permet d'identifier les acteurs qui agissent sur le système à étudier et regroupe un ensemble de documents qui définissent le comportement dynamique du système vis à vis de son environnement. Nous sommes en ce moment dans la première phase du cycle de vie du logiciel, la spécification. L'accent est mis sur les **contraintes externes** et une partie des **contraintes fonctionnelles**.

## LES ACTEURS

Les acteurs (*Actors*) ne font pas partie du système à étudier. Ce sont toute personne, système ou machine pouvant interagir avec le système. Un acteur peut par exemple donner et/ou recevoir des informations.

Ne seront pas pris en compte les personnes n'intervenant pas sur le système. Par exemple un emprunteur pour un système de gestion de bibliothèque n'est pas un acteur (sauf si c'est lui qui tape son numéro d'abonné).

Une même personne physique peut correspondre à plusieurs acteurs. Par exemple, vis à vis d'un système, une même personne peut être administrateur ou utilisateur. Dans ce cas il y aura 2 acteurs distincts.

Inversement si un professeur et un étudiant sont identifiés comme acteurs d'un système de gestion d'université, un assistant qui est à la fois professeur et étudiant ne doit pas être considéré comme un nouvel acteur car son rôle est déjà assumé par les 2 autres.

On distingue 2 types d'acteurs. Ceux-ci feront l'objet d'une étude différente:

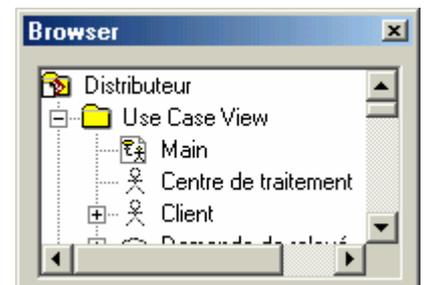
**Les acteurs actifs** sont des personnes, systèmes ou machines qui déclenchent une action sur le système à étudier

**Les acteurs passifs** sont des personnes, systèmes ou machines qui produisent ou consomment des données depuis ou vers le système sans déclencher un changement d'état du système à étudier

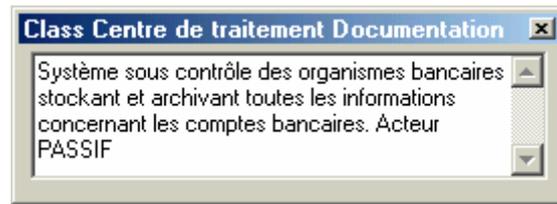
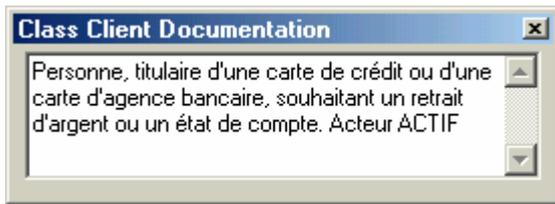
A partir du cahier des charges du terminal bancaire il est possible d'identifier 2 acteurs

**Le client** qui est un acteur actif. Il effectue des demandes (retrait, relevé de compte)

**Le centre de traitement** qui est un acteur passif car il ne fait qu'échanger des informations avec le terminal (no de compte, solde, état du compte...)



Ces 2 acteurs sont créés dans le navigateur de l'atelier. Chacun est documenté dans la fenêtre de documentation:



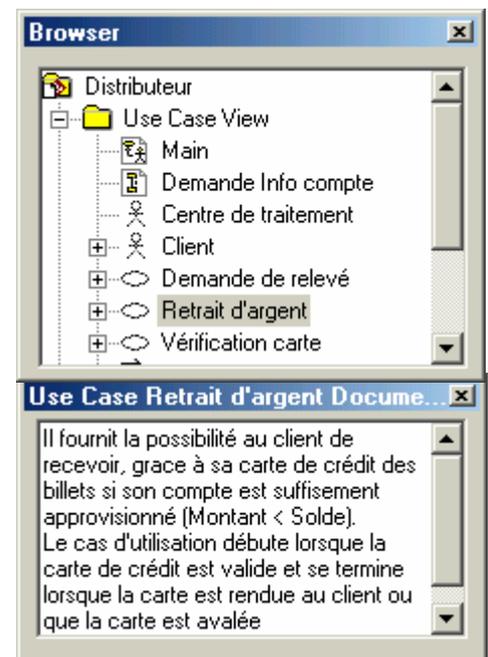
## LES CAS D'UTILISATION

Les cas d'utilisation correspondent à une formulation ou une re-formulation de l'expression des besoins, vu du côté des utilisateurs du logiciel à étudier. L'objectif est ici de définir les frontières du système.

Chaque cas d'utilisation représente un dialogue entre un acteur actif et le système, vu comme une boîte noire. L'ensemble des cas d'utilisation identifiés constitue toutes les façons dont le système peut être utilisé.

Les cas d'utilisation identifiables pour le système de gestion d'un terminal bancaire sont

Cas d'utilisation	Expression des besoins
<i>Retrait d'argent</i> <i>Demande de relevé</i>	...il peut effectuer soit une demande de relevé de compte (Franc ou Euro), soit un retrait d'espèces en indiquant la somme voulue ...
<i>Vérification carte</i>	... en cas de dépassement de validité ... une carte valide passe à la vérification du code confidentiel ..

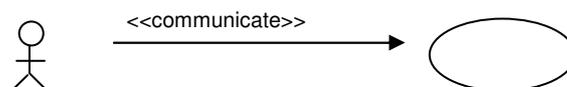


Chaque cas d'utilisation identifié doit être créé dans le navigateur de l'atelier et parfaitement documenté par une description de haut niveau. Cette description comprend les éléments suivant:

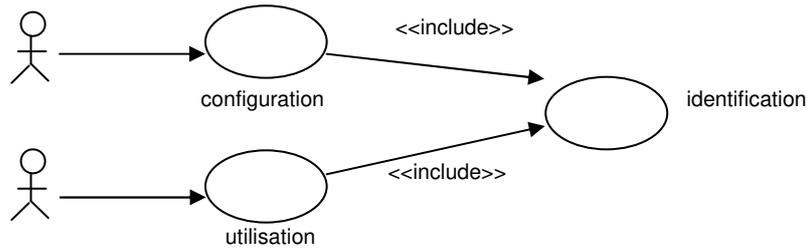
- le début du cas d'utilisation (quel événement le déclenche),
- la fin du cas d'utilisation (quel événement cause son arrêt),
- l'interaction entre le cas d'utilisation et les acteurs,
- les échanges d'informations
- la chronologie des informations
- les répétitions de comportement.

Lorsque tous les cas d'utilisation sont tous identifiés et documentés, il est nécessaire de les regrouper dans un diagramme de cas d'utilisation (☐ *Use Case Diagram*) qui est une vue graphique de tout ou partie des acteurs actifs du système, des cas d'utilisation et de leurs relations. Les relations peuvent être de plusieurs types (En UML on utilise le terme de *stéréotype*)

**<<communicate>>** qui représente un simple échange d'information entre un acteur et un cas. Son sens de navigation indique qui est à l'initiative de l'échange.

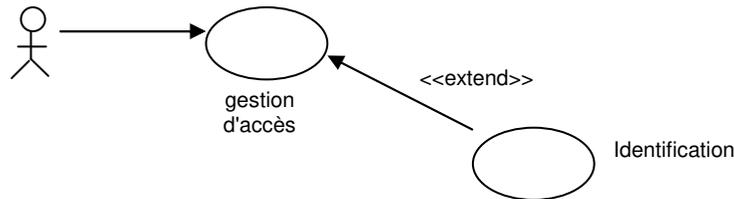


**<<include>>** qui matérialise des relations d'utilisation entre cas. Cette relation est utilisée lorsque des cas d'utilisations utilisent un cas commun. Par exemple la configuration et l'utilisation d'un système requièrent une identification de l'acteur.

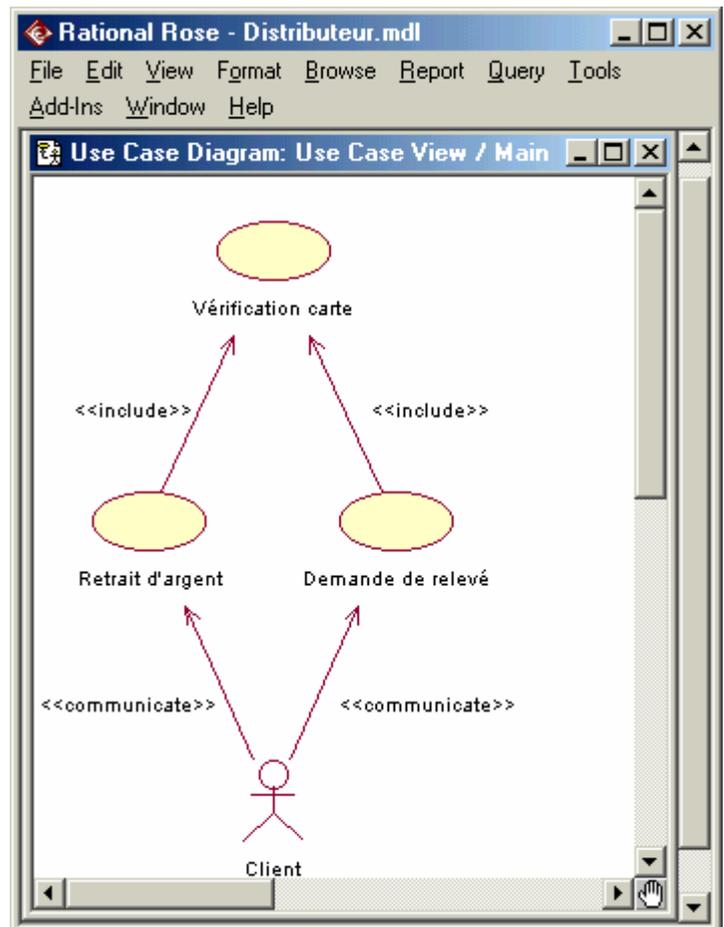
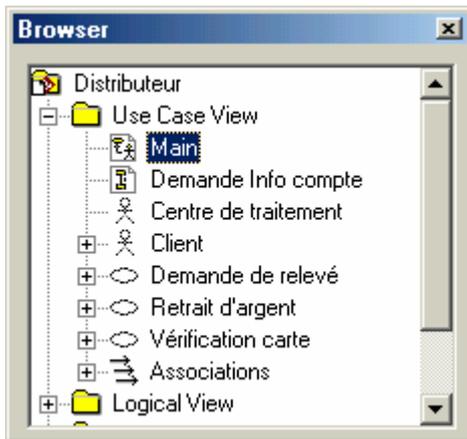


<<extend>>

qui matérialise aussi des relations d'utilisation entre cas. Cette relation exprime un comportement optionnel. Par exemple lorsqu'un utilisateur s'identifie et que son mot de passe a été erroné 3 fois, le cas d'utilisation peut être étendu d'un cas de gestion des accès au système pour permettre le blocage du compte.



Dans le cas de l'étude du terminal bancaire il s'agit d'insérer dans la vue des cas d'utilisation, un diagramme de cas d'utilisation que l'on nomme par exemple *Main*, dans lequel seront représentés les acteurs actifs, les cas d'utilisation et les relations. On notera que le cas "Vérification de carte" est lié au deux autres cas par une relation stéréotypée <<include>> puisque celui-ci est utilisé lors du "Retrait d'argent" et de la "Demande d'info compte".

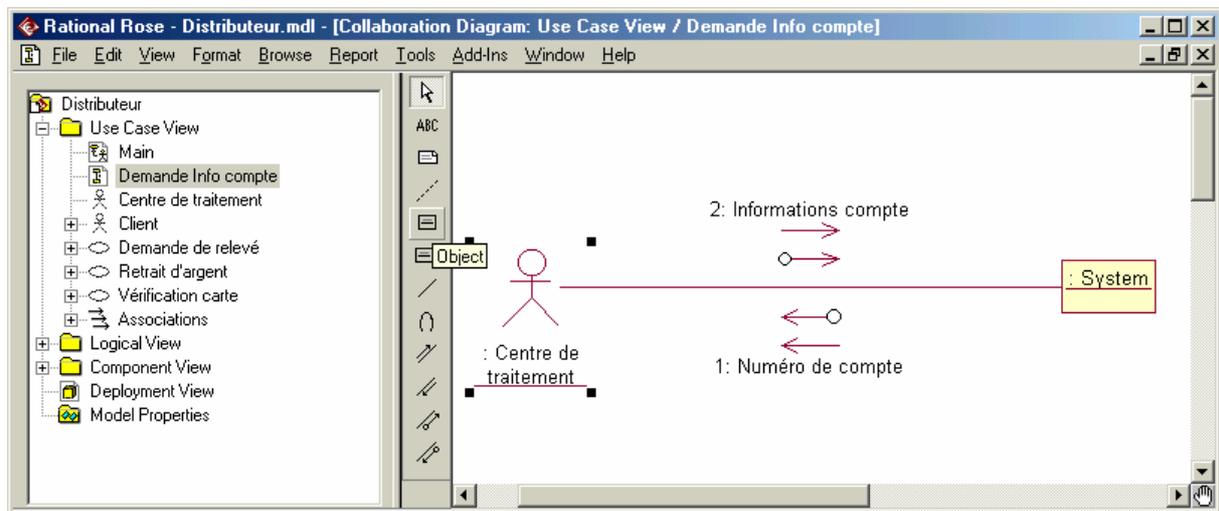


## LES DIAGRAMMES DE COLLABORATION

Afin de continuer à exprimer les besoins et d'affiner les frontières du système, il est nécessaire de déterminer les flots de données, statiques entre les acteurs passifs et le système. C'est le but des diagrammes de collaboration (□ *Collaboration Diagram*). Ces diagrammes mettent en avant les

données produites et consommées par le système à étudier.

Dans l'étude de cas un seul diagramme est nécessaire. C'est le diagramme de collaboration "Demande Info compte" qui est inséré dans la vue des cas d'utilisation.



Ce diagramme représente les données échangées avec le "Centre de traitement", le système étant considéré comme une boîte noire. Le système est un objet dont le nom est *: System*. Il est obtenu à partir de la barre d'outils centrale, *Toolbox*, ainsi que les données *Data Token* (O→) portées par les *Link Message* (→) visibles sur le diagramme ci-dessus.

Pour chaque vérification de solde ou demande de relevé, le système envoie le numéro de compte et reçoit en échange du Centre de traitement toutes les informations concernant le compte (solde, liste des opérations, informations d'état, ...).

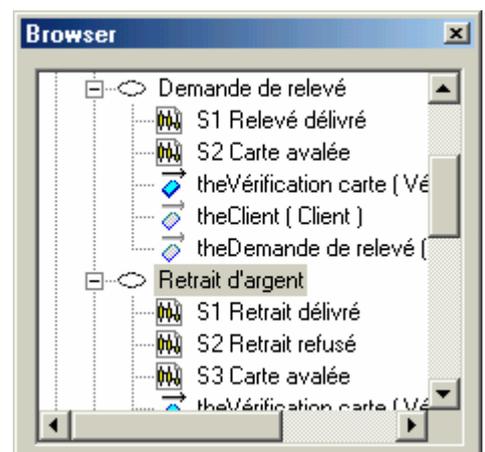
Les transactions et les situations anormales ne sont pas envisagées. Elles pourraient faire l'objet d'une étude sous forme d'un diagramme de séquence comme il est vu paragraphe suivant.

## LES SCENARIOS

Afin de terminer l'analyse externe du système il est nécessaire de déterminer les scénarios qui décrivent comment les cas d'utilisation sont réalisés. Chaque cas d'utilisation est vu comme un ensemble de scénarios. Chaque scénario est associé à un Diagramme de séquence (□ *Sequence Diagram*) simplifié où sont représentés les transactions entre un acteur actif et le système à étudier. Ces diagrammes mettent en avant l'aspect temporel des interactions.

Pour chaque cas d'utilisation on doit s'interroger sur les différents "cas de figure" dans les transactions. Par exemple, en cas de "Retrait d'argent" il faut envisager

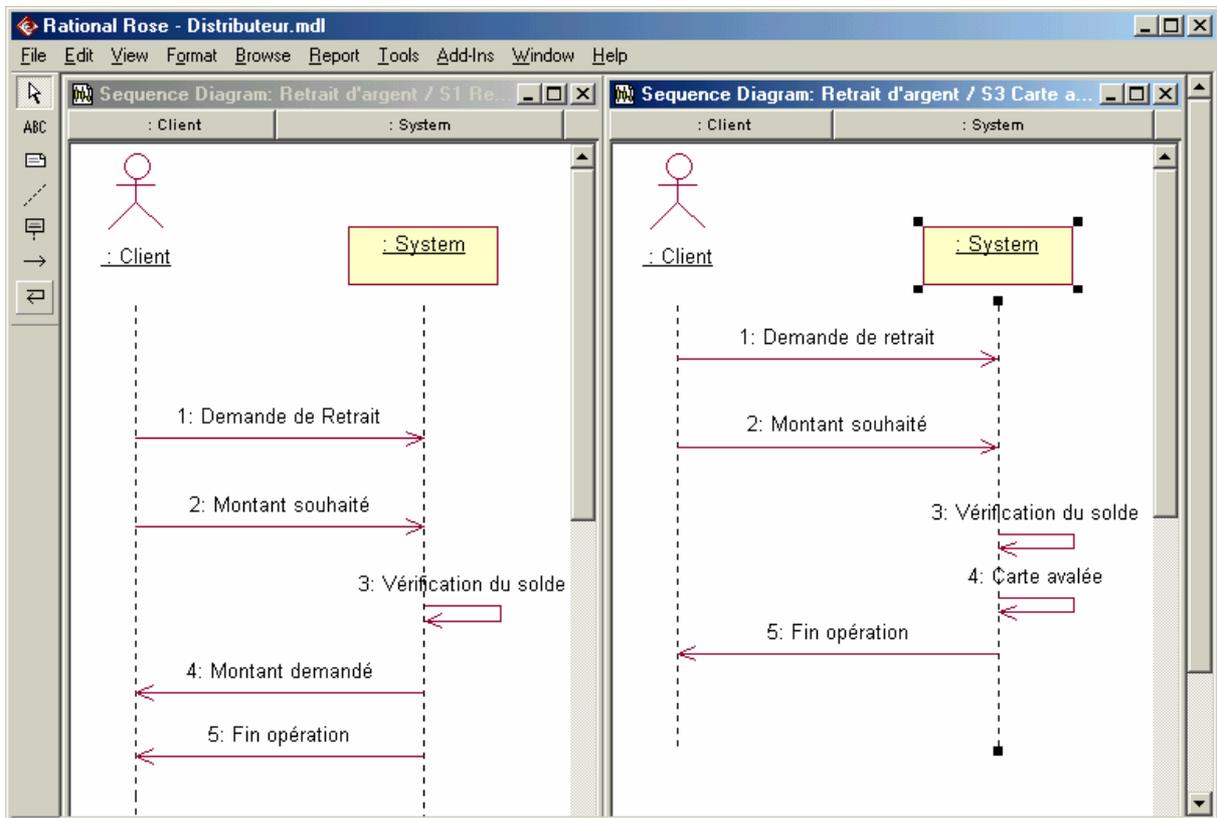
- S1. Le retrait est effectivement effectué
- S2. Le retrait est refusé car le solde est insuffisant
- S3. La carte est avalée car elle est périmée ou le compte est bloqué



Concrètement, dans le navigateur, à l'intérieur de chaque cas d'utilisation, il faut insérer autant de *Sequence Diagram* qu'il y a de scénarios possibles. Chaque diagramme représentera l'acteur actif concerné, le système à étudier sous forme d'un objet de nom *: System*, et des messages échangés

Les messages sont unidirectionnels, ils sont envoyés de l'acteur vers le système ou du système vers l'acteur. Ils correspondent à un signal explicite ou une demande de service. Les messages sont numérotés dans l'ordre chronologique d'envoi, le temps s'écoule de haut en bas.

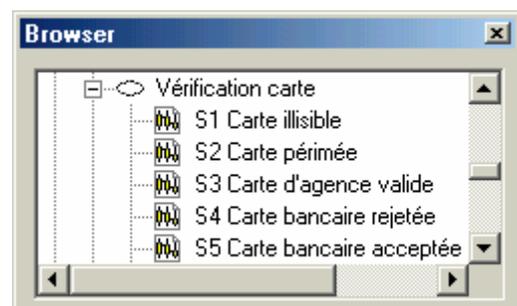
Les messages et l'objet système sont des entités sélectionnées depuis la barre d'outils *Toolbox*. L'objet système peut s'envoyer un message à lui-même pour matérialiser une activité de haut niveau qui s'exerce au sein du système (Exemple la vérification du solde, l'absorption de la carte, ...)



Du texte libre et des notes peuvent être insérés dans de tels diagrammes.

L'analyse du terminal bancaire fait apparaître:

- 2 scénarios pour le cas "Demande de relevé"
- 3 scénarios pour le cas "Demande d'argent"
- 5 scénarios pour le cas "Vérification de carte"





# VUE LOGIQUE

## OBJECTIF

Les documents produits au chapitre précédent ont permis d'affiner les besoins des utilisateurs. L'accent a été mis sur les contraintes externes du système à étudier ainsi que sur une partie des contraintes fonctionnelles. Ces contraintes fonctionnelles ont été limitées aux interactions entre les utilisateurs et le système.

Il est important maintenant de les enrichir en analysant la "dynamique interne" du système. La vue logique permettra d'une part, d'identifier d'autres **contraintes fonctionnelles** mais aussi d'élaborer une première **architecture du logiciel**. Celle-ci sera basée sur une décomposition en sous-systèmes (ou catégories), regroupant des classes liées par des associations.

Notons que la vue logique du système recouvre les phases de spécification et de conception générale du cycle de vie du logiciel. Tous les documents ainsi produits apparaîtront dans la vue logique ( *Logical View*) de l'atelier Rational Rose.

## IDENTIFICATION DES CLASSES

Il s'agit maintenant de décrire le comportement des objets et leurs interactions. Ces **objets** sont des entités concrètes ou conceptuelles qui modélisent des objets du monde réel, donc du système à étudier. Ainsi, le logiciel de gestion du terminal bancaire manipule par exemple des objets "cartes", un objet "lecteur de carte".

Comment identifie-t-on un objet? Un objet a un **état** qui correspond à l'une des multiples situations dans lesquelles il est susceptible de se trouver. Il est défini par un ensemble *d'attributs* qui ont une valeur propre. Ainsi par exemple l'objet lecteur de carte peut être dans l'état "libre", "en cours de lecture", "bloqué", ... Un objet a aussi un **comportement** constitué d'un ensemble *d'opérations* qui détermine sa dynamique, en réponse aux requêtes d'autres objets. Ainsi l'objet lecteur de carte peut avoir l'opération "Rendre la carte".

Une **classe** est la description d'un groupe d'objets qui partagent les mêmes attributs et les mêmes opérations. Cette notion est à prendre au sens large et s'applique aussi au cas où il n'existerait qu'un seul objet "du même modèle" dans le système à étudier (c'est le cas du lecteur de carte). Une classe est un moule à objets, tout objet est une **instance** d'une et une seule classe. La classe *Carte* regroupera par exemple les attributs (*code agence, numéro de compte, ...*) et les opérations (*Vérifier code, ...*) de tous les objets cartes qui seront insérées dans le terminal bancaire.

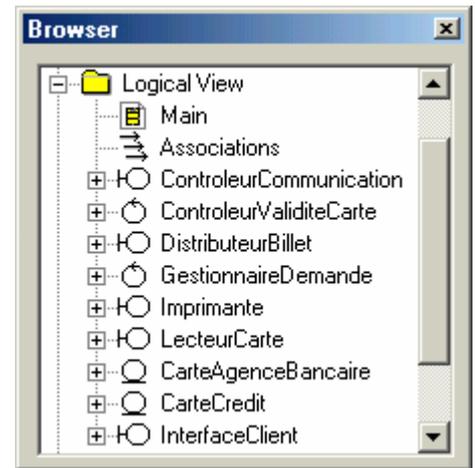
Les classes du système sont stéréotypées c'est à dire qu'elles peuvent appartenir à une des 3 principales catégories de classe



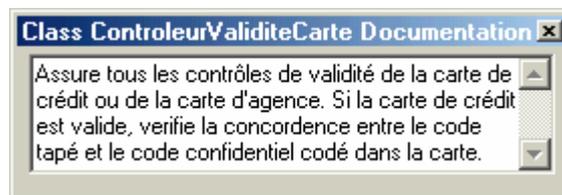
Les classes entités (*entity*) correspondent aux entités du monde réel. Elles effectuent des traitements internes au système. Elles sont indépendantes de la manière dont le système communique avec l'environnement. A partir du cahier des charges du système de gestion du terminal bancaire il est possible d'identifier rapidement les classes *Carte de crédit* et *Carte d'agence bancaire*.

Les classes interfaces (*boundary*) prennent en charge la communication entre le système et son environnement. Elles fournissent l'interface entre le système et les acteurs passifs ou actifs (*Interface client, Contrôleur communication*) ou entre le système et les organes physiques (*Imprimante, Lecteur carte, Distributeur billet*).

Les classes de contrôles (*control*) permettent l'exécution d'un cas d'utilisation et prennent en charge les flux d'événements. On les trouve souvent en relation avec les classes d'interface pour gérer les demandes utilisateur (*Gestionnaire demande*) ou pour assurer les contrôles importants (*Contrôleur validité carte*)



Les classes sont insérées, au fur et à mesure de leur identification dans la vue logique (*Logical View*) du navigateur. Au moment de leur création, les classes, quel que soit leur stéréotype, sont documentées. Cette documentation explicite l'objectif de la classe et sa fonctionnalité.

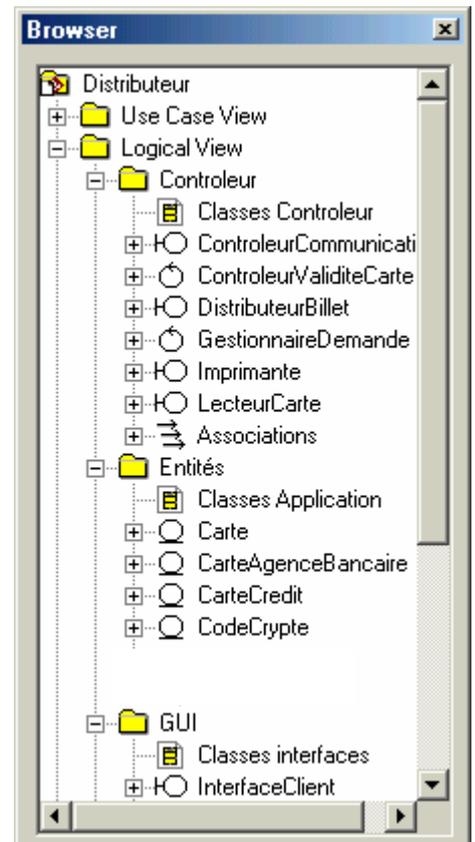


Lorsque toutes les classes sont identifiées, il est souvent nécessaire de les regrouper pour en faciliter la gestion, la maintenance et la réutilisation. Les **paquetages** (*packages*), regroupant une collection de classes, peuvent être insérés dans la vue logique (*Logical View*). Le choix des packages peut être effectué avec comme objectif de faire apparaître des sous-systèmes du système à étudier. Une autre approche possible consiste à répartir les classes en séparant:

- les classes assurant l'interface utilisateur. Celles-ci peuvent évoluer en cas de portage du système sur un autre environnement (Windows, UNIX, ...),
- les classes de stéréotype *entity* qui sont en général réutilisables,
- les "classes métier" qui correspondent à un savoir-faire propre à l'entreprise. Celles-ci sont en général réutilisées des précédents développements'
- les classes qui pilotent les organes physiques (ceux-ci peuvent évoluer pendant le développement du système).

L'analyse du système de gestion du terminal bancaire fait apparaître 3 packages (*Contrôleur, Entités, et GUI*).

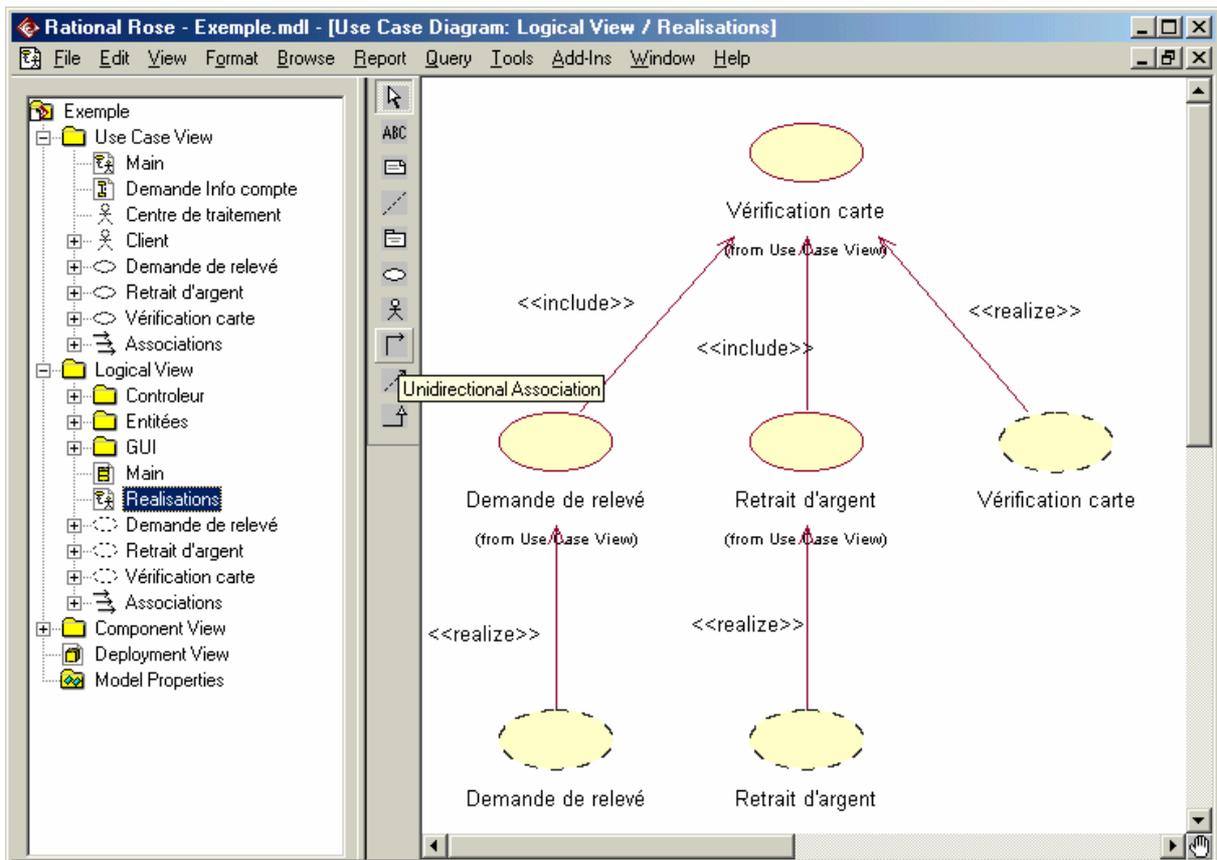
Lorsque tous les packages sont définis, les classes précédemment identifiées, sont déplacées dans les packages appropriés.



# REALISATION DES CAS D'UTILISATION

Cette étape est très spécifique à l'utilisation de l'atelier Rational Rose. Les cas d'utilisation ont été décrits dans la vue des cas d'utilisation. Les diagrammes de séquences qui vont être élaborés lors de la prochaine étape vont expliciter comment chaque cas d'utilisation est réalisé par le biais d'interactions entre les objets identifiés précédemment. Ces diagrammes appartiennent à la vue logique.

Le diagramme de cas d'utilisation (☐ *Use Case Diagram*) nommé "Réalisations" assure le lien entre les cas d'utilisation (☞ *Use Case View*) et leur réalisation (☞ *Logical View*).



Pour chaque cas d'utilisation il s'agit d'insérer, dans la vue logique, un cas d'utilisation de stéréotype `<<use-case realization>>` portant exactement le même nom. Le diagramme "Réalisations" de type *Use Case Diagram* est inséré lui aussi dans la vue logique. Les cas d'utilisation et leur réalisation sont placés dans le diagramme par un simple "Glisser-Poser".

# LES DIAGRAMMES DE SEQUENCE

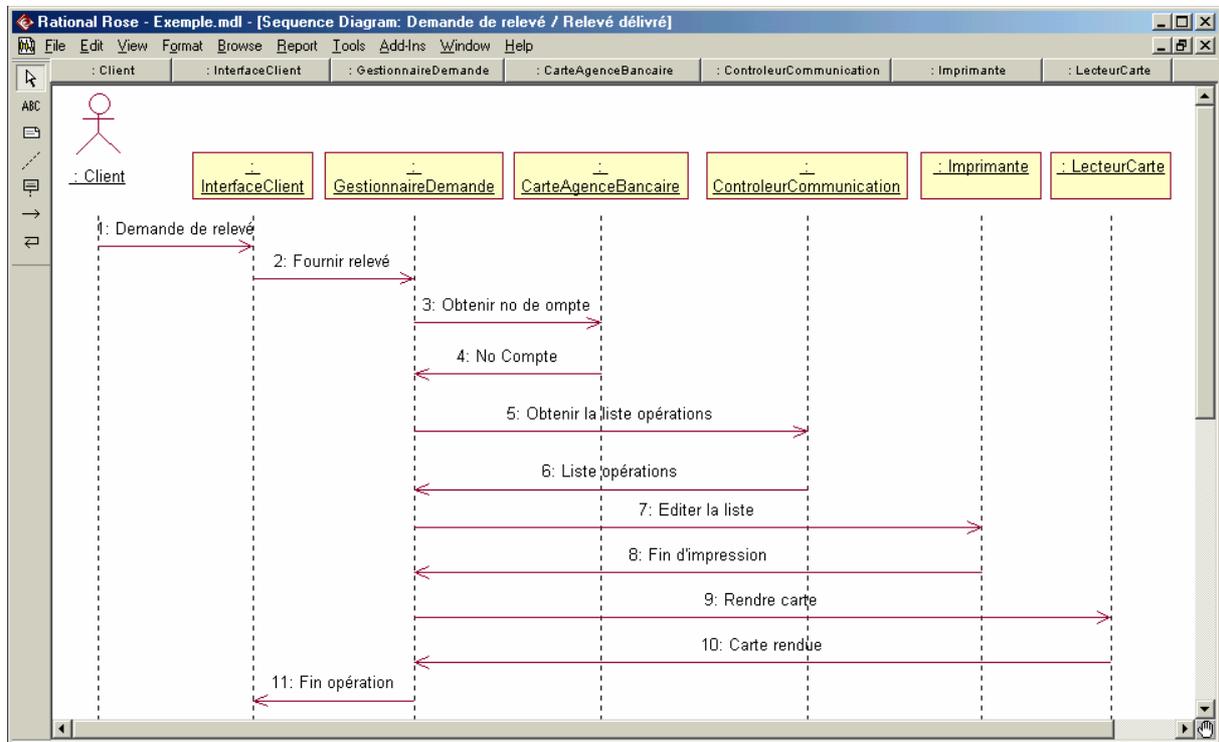
Les diagrammes de séquence (☐ *Sequence Diagram*) obligent l'analyste à réfléchir à la dynamique des objets donc au comportement interne du système à étudier. Ils vont permettre d'identifier de nouvelles **contraintes fonctionnelles**.

Ces diagrammes sont en fait, des extensions des scénarios qui ont été élaborés préalablement. Ils montrent les objets et les classes impliqués dans un scénario, ainsi que la succession des messages



échangés entre les objets pour réaliser la fonctionnalité souhaitée. Avec l'explorateur, un diagramme de séquence (□ *Sequence Diagram*) est créé, sous une réalisation de cas d'utilisation. Les objets concernés sont des instances des classes précédemment identifiées ou spécialement créées pour cette réalisation.

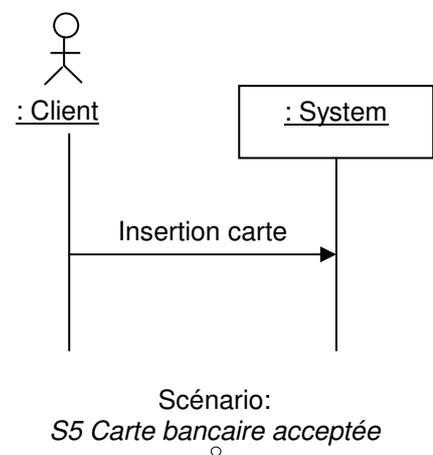
Comme pour les scénarios, les messages sont unidirectionnels, ils sont envoyés d'un objet vers un autre objet. Ils correspondent à un signal explicite ou une demande de service. Les messages sont numérotés dans l'ordre chronologique d'envoi, le temps s'écoule de haut en bas.

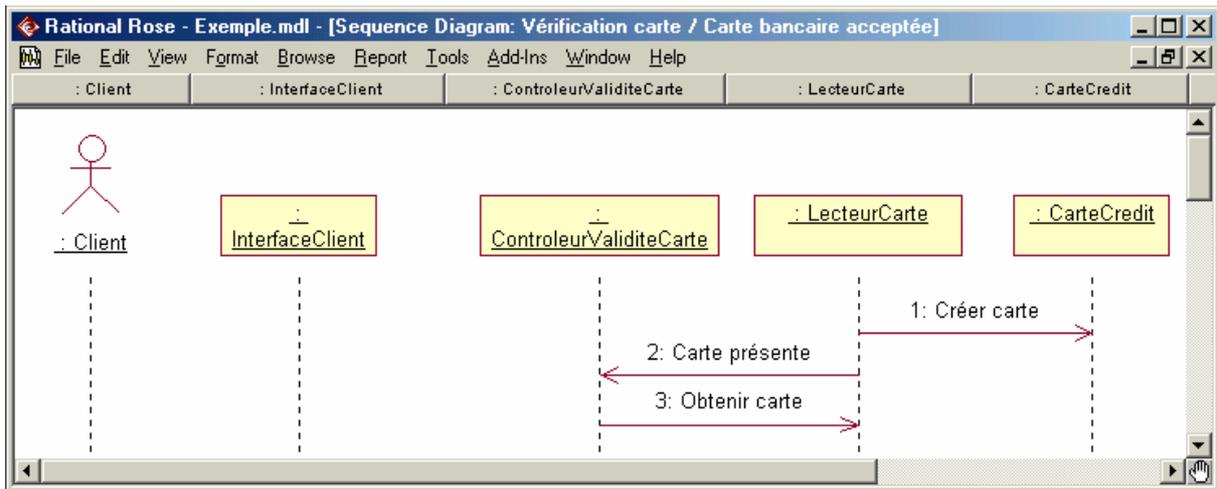


Les classes d'interface utilisateur (*InterfaceClient*) sont identifiées très tôt dans l'analyse. Par contre lors de la création d'un diagramme de séquence il est souvent nécessaire d'ajouter des classes de contrôle pour prendre en charge et centraliser les flots d'événement (*GestionnaireDemande*). Pour rendre le système à étudier indépendant des "philosophies" différentes d'interface utilisateur, il est toujours préférable de séparer les interactions vers les acteurs, de la "mécanique" qui en découle.

Un diagramme de séquence doit correspondre à un scénario. Souvent, les messages entre l'acteur et l'objet *:InterfaceClient* sont identiques à ceux décrit dans le scénario correspondant.

Une petite exception cependant: Les scénarios, affinant l'expression des besoins, explicitent les messages de haut niveau entre l'acteur et le système (exemple *Insertion carte*). Les diagrammes de séquence sont élaborés en fin de spécification, à une période de développement où l'architecture du logiciel commence à se dessiner. Les messages entre objets sont plus concrets. C'est la raison pour laquelle il n'y a pas toujours concordance, au sens stricte, entre un scénario et le diagramme de séquence associé. Au message "*Insertion carte*" entre l'acteur et le système dans le scénario "*S5 Carte bancaire acceptée*" correspond une séquence de 2 messages à l'initiative de l'objet *:LecteurCarte* (c'est le lecteur de carte qui détecte que le client a inséré sa carte) dans le diagramme de séquence ci-dessous.

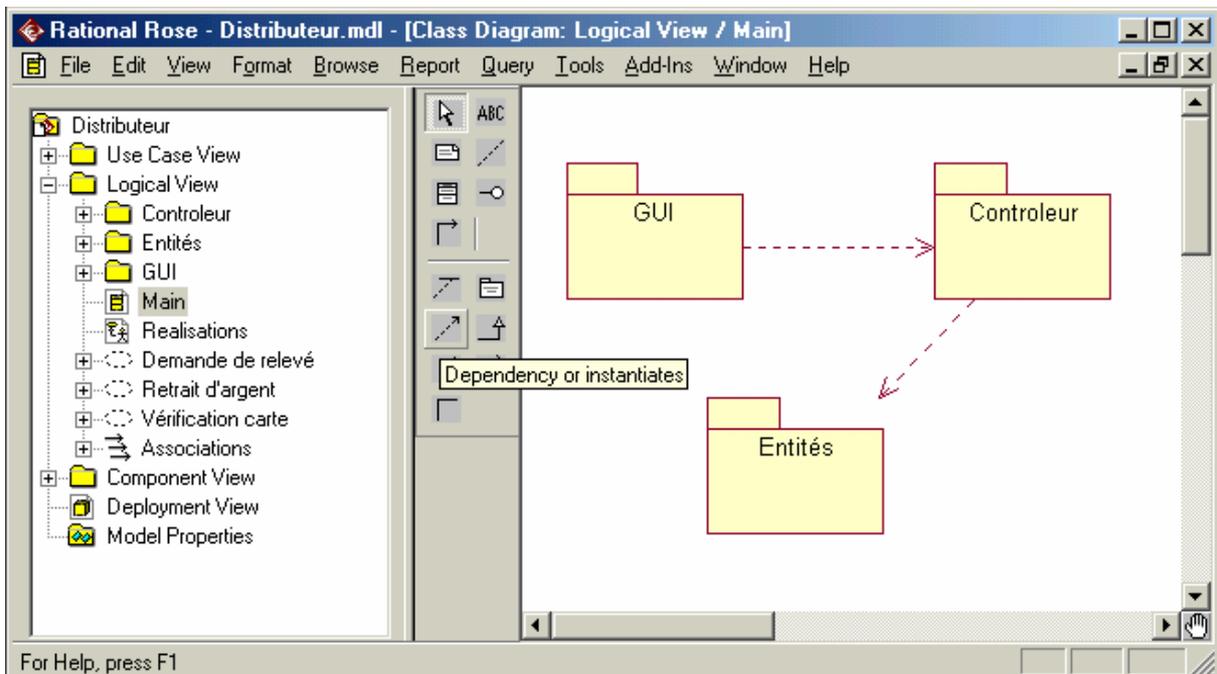




## LES DIAGRAMMES DE CLASSES

Les diagrammes de classe ( *Class Diagram*) mettent en évidence les relations qui existent entre les paquetages ( *packages*) ainsi qu'entre les classes d'un même paquetage. Ils vont permettre, de définir l'**architecture** du logiciel à étudier

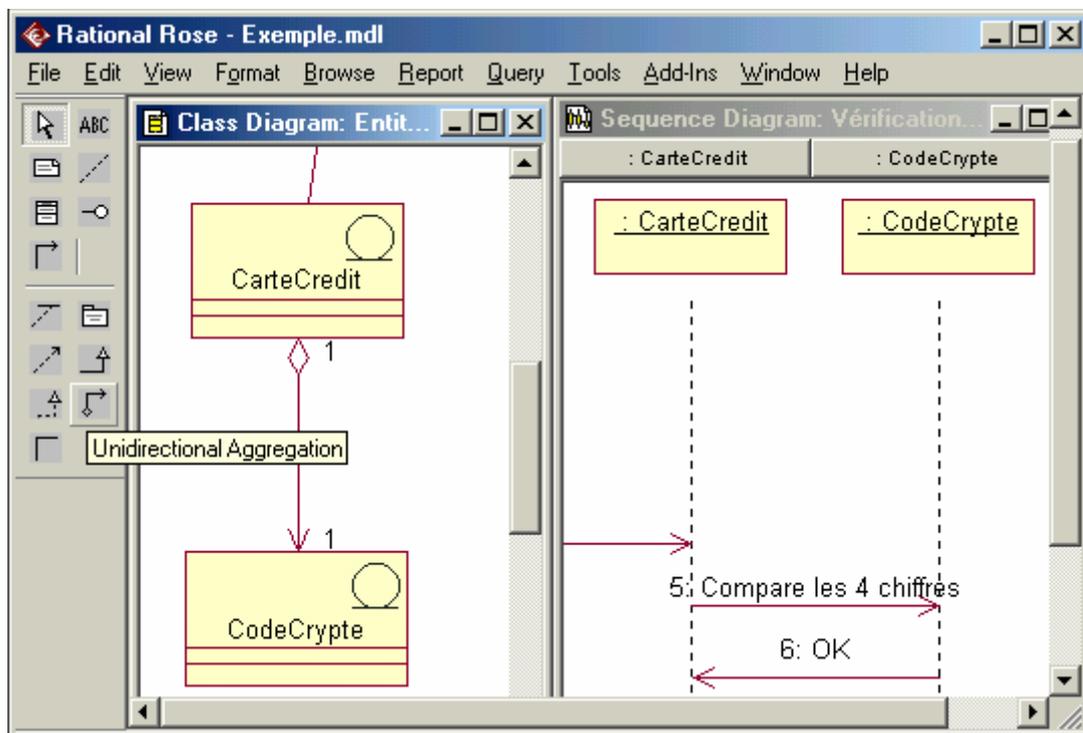
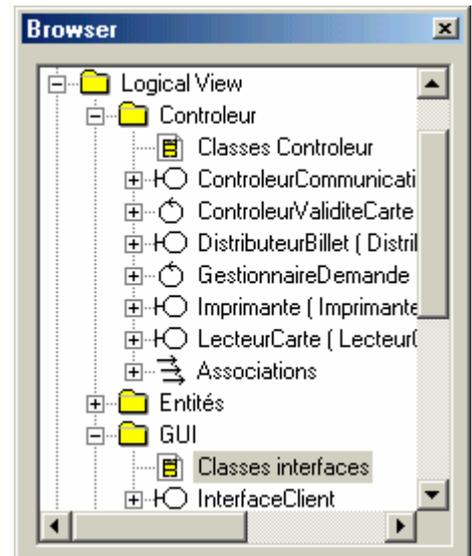
Le diagramme de classe *Main* illustre les relations de dépendance entre les 3 paquetages GUI, Contrôleur et Entités. Une dépendance  $A \rightarrow B$ , d'un paquetage A vers un paquetage B, sous-entend qu'un ou plusieurs objets d'une ou plusieurs classes de A communiquent avec un ou plusieurs objets d'une ou plusieurs classes de B. A est le client et B est le fournisseur. Une relation est déduite de l'étude des diagrammes de séquence. Par exemple, dans le diagramme "Demande de relevé / Relevé délivré", la classe *InterfaceClient* ( GUI) communique avec la classe *GestionnaireDemande* ( Contrôleur) qui elle-même communique avec la classe *Carte* ( Entités).



Chaque paquetage doit maintenant contenir un diagramme de classe ( *Class Diagram* ) décrivant les relations qui existent entre ses propres classes internes. Ces relations découlent aussi de l'étude des diagrammes de séquence. Un échange de message entre 2 objets indique que ceux-ci communiquent, donc qu'il existe une relation d'association ou d'agrégation entre leur classe respective.

Une **relation d'association** entre classe précise qu'il existe une communication entre les objets des classes concernées. C'est un lien bidirectionnel. Par exemple il existe un lien d'association entre la classe *ControleurValiditeCarte* et *LecteurCarte* puisque dans le diagramme "Vérification carte / Carte bancaire acceptée" leurs objets respectifs échangent des messages.

Une **relation d'agrégation** est un cas particulier d'association où des classes sont liées par une relation du type "est une partie de"; d'où une notion classe mère / classe fille. Par exemple, une carte de crédit est composée d'un objet code crypté

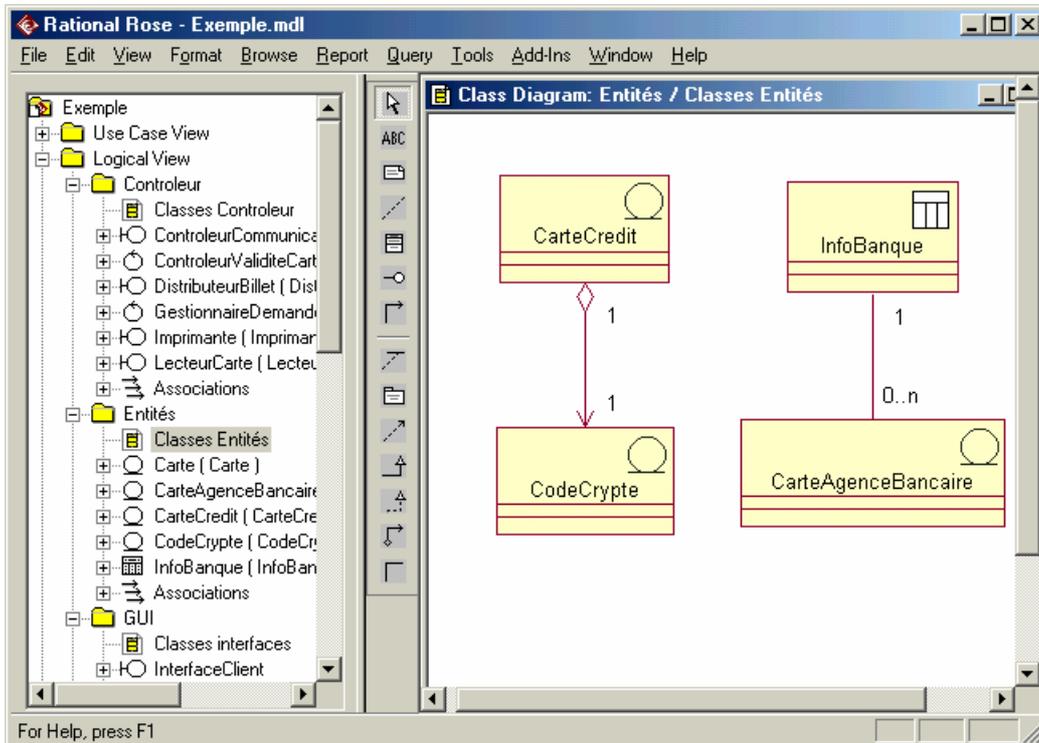


Les relations d'association et d'agrégation peuvent être décorées de deux indicateurs de **cardinalité** précisant le nombre d'objets qui participent à la relation. Les indicateurs ci-dessus indiquent qu'une carte de crédit dispose d'un seul code crypté (1 en bas) et qu'un code crypté est associé à une et une seule carte (1 en haut). Les indicateurs les plus courants sont:

- 1            Un et un seul
- 0..\*        Zéro ou plus
- 1..\*        Un ou plus
- 0..1        Zéro ou un
- 3..5        Intervalle donné (3, 4 ou 5)

Les diagrammes de classe sont construits à partir des classes précédemment identifiées lors la création des diagrammes de séquence. Il est parfois nécessaire de créer des classes supplémentaires afin de fournir des services particuliers à une ou plusieurs classes. Par exemple, la classe

*CarteAgenceBancaire* devra connaître les informations relatives à l'agence pour assurer son comportement (contrôle de validité). Cette classe est particulière, elle n'a que des attributs. Son stéréotype sera <<table>>. Le diagramme de classe associé au paquetage "Entités" est présenté ci-dessous.



Il nécessaire maintenant de définir la structure et le comportement de chaque classe identifiée. La structure correspond à un ensemble d'attributs, le comportement à un ensemble d'opérations.

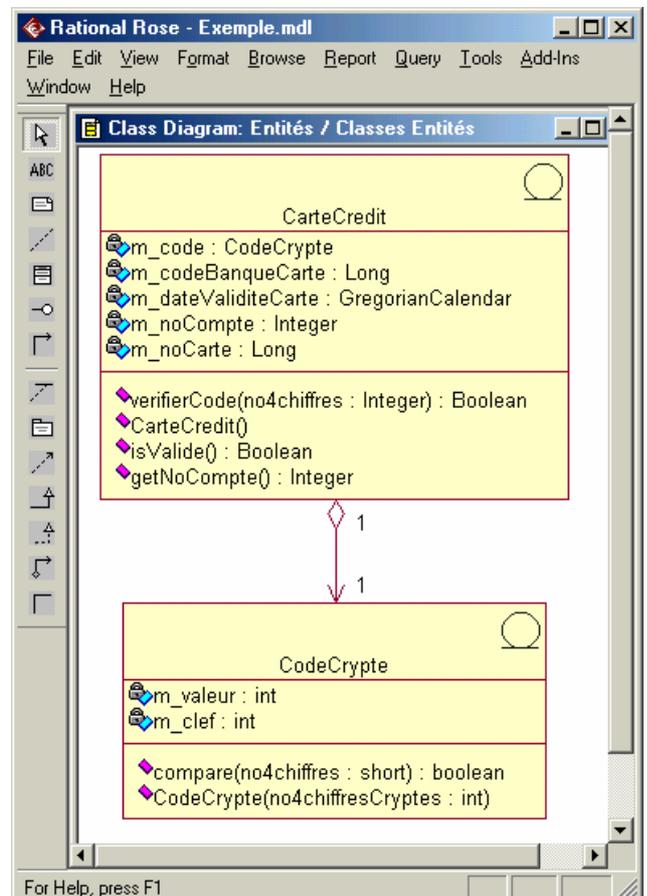
Les **attributs** sont des données qui décrivent la structure des objets de la classe. Ils sont identifiés lors de l'examen de l'énoncé du problème:

*question* Qu'est-ce qui caractérise toutes les "cartes de crédit" ?

*réponse* Elles possèdent toutes une date de validité, un numéro, une référence de la banque émettrice, le numéro de compte du propriétaire, le code confidentiel crypté.

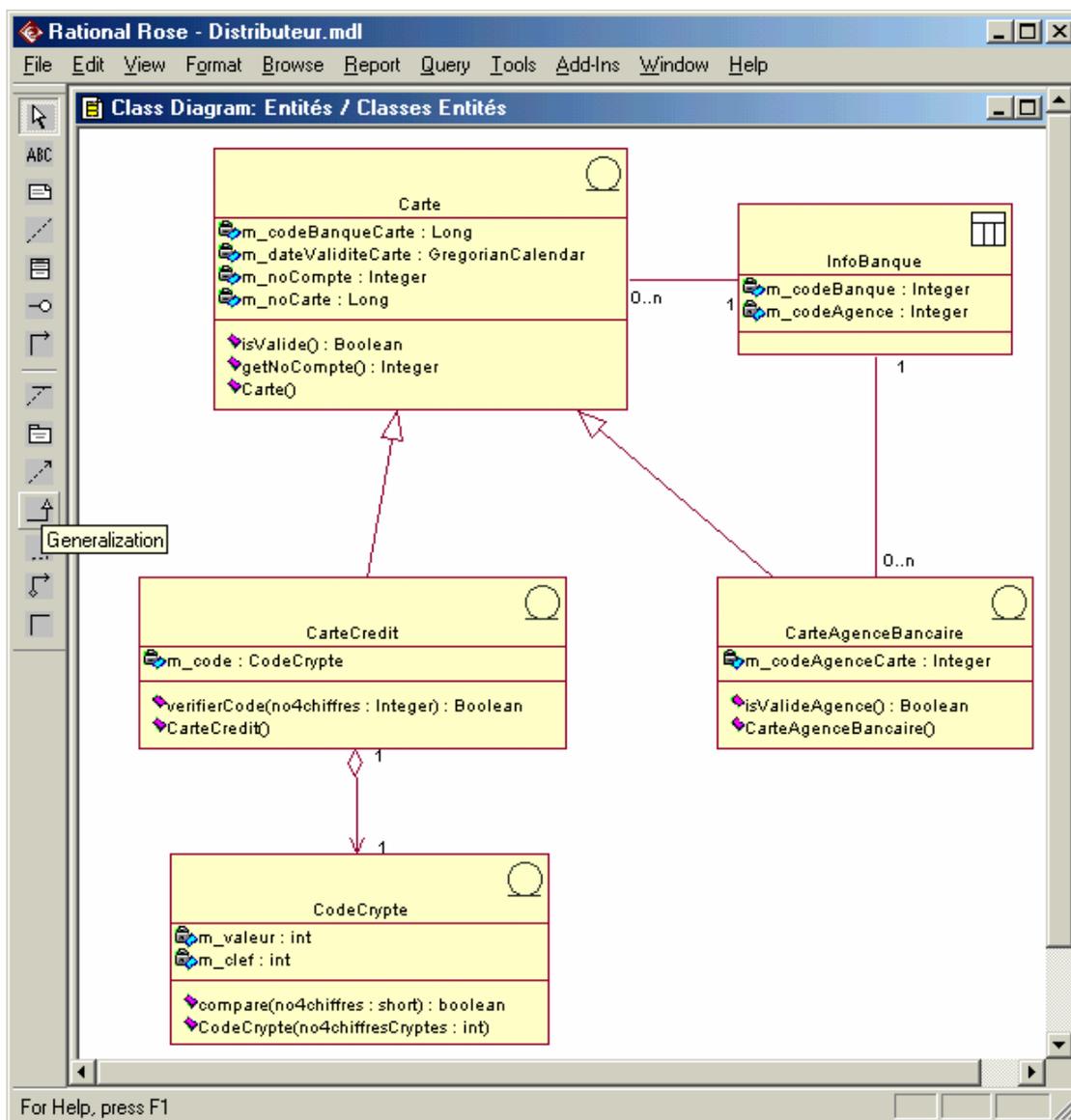
Comme toutes variables, les attributs sont typés. On utilise une notation indépendante des langages `m_noCompte : Integer`.

Les **opérations** sont des méthodes ou fonctions qui décrivent le comportement des objets de la classe. Les messages qui apparaissent dans les diagrammes de séquence correspondent tous à une opération de la classe réceptrice à l'exception des classes d'interface qui se substituent à l'interface utilisateur. Le nom des opérations est choisi en tenant compte de la classe réceptrice. Par exemple l'objet `CarteCredit` envoie le



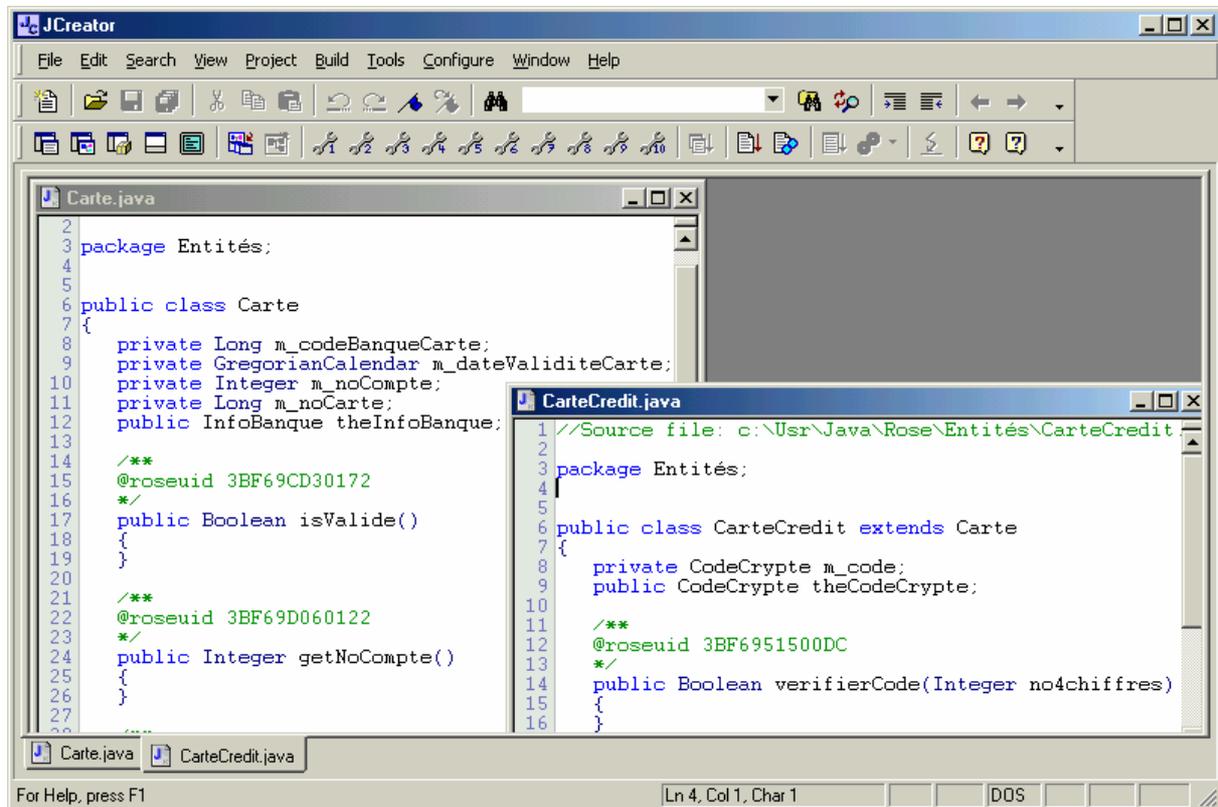
message "Compare les 4 chiffres" à l'objet `:CodeCrypte`. Il en découle que la classe `CodeCrypté` contiendra une opération nommée `compare()`. Il est possible d'affiner la signature de cette méthode en spécifiant l'argument déduit du libellé du message et un type de retour déduit du message qui suit. A cette transaction est associée l'opération `compare(short no4chiffres) : Boolean`.

Il est important maintenant de définir les éventuelles **relations d'héritage** entre les classes identifiées et de factoriser les structures et les comportements communs. Les relations d'héritage ne sont pas identifiées comme les autres relations. Elles découlent d'une analyse des attributs et des opérations. Une relation d'héritage indique qu'une classe partage la structure (attributs) et le comportement (opérations) d'une autre classe. La relation est du type "est-une" ou "est-une-sortre-de". Une classe hérite de la structure et du comportement de sa super-classe. Ainsi une carte de crédit et une carte d'agence bancaire ont toutes les deux un numéro, une date de validité, une référence à une banque. Elle contient le numéro de compte de son propriétaire. Leur contrôle de validité est identique. Il est donc possible de généraliser leurs opérations et attributs communs dans une classe `Carte`. Une "carte de crédit" est une "carte" qui a en plus un code confidentiel, une "carte d'agence bancaire" est une "carte" qui a en plus un code d'agence. La classe `Carte` sera la **super-classe** des classes `CarteCredit` et `CarteAgenceBancaire`. Les classes `CarteCredit` et `CarteAgenceBancaire` sont des **sous-classes** de la classe `Carte`. Elles **héritent** de ses attributs et de ses opérations. Le diagramme complet associé au paquetage `Entités` est présenté ci-dessous



Lorsque tous les diagrammes de classe sont élaborés, l'architecture du logiciel est connue. Le système à étudier est découpé en paquetages regroupant des classes liées par des relations d'association, composition ou héritage. Les attributs et les opérations sont connus. Par défaut, les attributs sont privés et les opérations publiques, mais leur visibilité est modifiable. Il est possible maintenant, paquetage par paquetage de générer automatiquement un squelette de source Java.

Les paquetages sont des *packages* Java, les super-classes sont des classes publiques. Les attributs sont des *attributs* Java privés. Les opérations sont des *méthodes* publiques



Les liens d'agrégation correspondent, dans la classe mère à des objets publics, instance de la classe fille. Lorsque la cardinalité est multiple un tableau d'objets est implémenté.

```
public class CarteCredit extends Carte
{
    public CodeCrypte theCodeCrypte;
```

Il en est de même pour les liens d'association

```
public class Carte
{
    public InfoBanque theInfoBanque;
```

La classe *InfoBanque* stéréotypée <<table>> est implémentée comme une *interface* Java publique

```
public interface InfoBanque
{
    private Integer m_codeBanque;
    private Integer m_codeAgence;
}
```