L ’approche Objet

d ’abord …

Une méthodologie de conception

par
Mohammed Chraibi
Consultant JAVA/J2EE
chraibi@gmail.com

Problématique

Taille et complexité des logiciels :

- **Complexité fonctionnelle :**

  Exemples :

  1/Le S.I.A. : mémoriser et stocker l'information : mais en plus traiter de façon sophistiquée pour l'aide à la décision (***Entrepôt de données***).

  2/ Logiciels développés séparément et avec des démarches différentes et appelés à être interfacés pour les besoins de l'Entreprise.

- **Evolutions technologiques permanentes**
- **Complexité architecturale : Client/serveur, Intranet, Corba (Common Object Request Broker Architecture), Systèmes distribués…**

Solutions :

- **Découpage du processus de développement :**
  - phase analyse : aspects ;
  - phase réalisation : aspects technologiques et architecturaux.
- **Découpage du système en sous systèmes : diminution de la complexité ; répartition du travail et réutilisation .**
- **Utilisation d'une technologie de haut niveau : découpage naturel du système .**

L’approche objet

Notion d’objet

Un objet est défini à la fois par des **informations** : données ou attributs ou variables d’instances ; et des **comportements** : traitements ou méthodes ou opérations.

Exemple :

nom
capital UV
diplôme
VérifierNom
MajUV
ChangerDiplôme

OBJET Etudiant

Notion de classe et d'instance

Lorsque des objets ont les mêmes attributs et comportements : ils sont regroupés dans une famille appelée : **Classe**.

Les objets appartenant à celle-ci sont les **instances** de cette classe.

L'**instanciation** est la création d'un objet d'une classe.

Nom : *Dupont* Capital UV : *capital*$_1$ Diplôme : *maîtise de Sciences Eco.* VérifierNom MajUV ChangerDiplôme	Nom : *Durant* Capital UV : *capital*$_2$ Diplôme : *licence de Socio.* VérifierNom MajUV ChangerDiplôme

Deux instances d'une même classe peuvent avoir des attributs avec des valeurs différentes et partagent les mêmes méthodes.

Les messages

La manipulation des objets passe par des envois de **messages**.

Lorsqu'un objet reçoit un message :

- Soit le message correspond à un traitement défini dans la classe de l'objet auquel cas la méthode correspondante est exécutée. L'objet répond ainsi au message.
- Soit le message ne correspond pas, l'objet refuse le message et signale une erreur.

Un message équivaut à un appel d'une méthode.

Un objet gère lui même son comportement.

Ce qui lui permet soit de traiter des messages en exécutant les méthodes correspondantes soit de rejeter des messages en signalant des erreurs.

Un objet est parfaitement identifié. Comme s'il possédait un attribut (inaccessible directement) qui identifie la classe à laquelle il appartient.

L 'Encapsulation

L'encapsulation est le fait qu'un objet renferme ses propres attributs et ses méthodes.

Une classe encapsule les propriétés (attributs

et méthodes) des objets qu'elle regroupe.

La **modularité** est souvent laissée à la charge du développeur.

Dans l'approche Objet : celle-ci est prise en compte par l'encapsulation.

L'unité de modularité est la classe.

Les classes peuvent être regroupées en **packages** ou en **sous systèmes** (granularité supérieure).

L 'abstraction

L'abstraction est la caractérisation d'un objet par *une* ***partie publique****, une* ***partie privée*** et *une* ***partie implémentation****.*

▪**L'accès public :**

Tout ce qui est accessible par les autres objets. Les méthodes publiques représentent l'interface de l'objet.

Les données quand elles sont publiques n'imposent aucun contrôle ni sur leur structure ni sur la nature des valeurs qu'elles peuvent recevoir.

Il est préférable de mettre les données en accès privé.

▪**L'accès privé :**

Les données privées ne sont modifiables qu'à travers les méthodes publiques qui peuvent les contrôler ainsi.

▪**La partie implémentation :**

Elle est définie par un ensemble de méthodes accessibles que par les autres méthodes de la même classe.

L 'abstraction

Exemple :

La donnée *Capital UV* n'est modifiable
que par la méthode *MajUV*.

Ce concept d'abstraction engendre deux catégories d'acteurs :

- ✓ **les concepteurs des classes**
- ✓ **les utilisateurs des objets**

Ces derniers peuvent utiliser les méthodes d'une classe indépendamment de leurs structures internes.

Ils n'utilisent que les signatures des méthodes (interface de l'objet) .

Ce qui permet aux concepteurs des classes d'objets de modifier la structure interne des méthodes des classes.

L 'Héritage

la classe **Etudiant** :	la classe **Etudiant-Elu** :
nom *capital UV* *diplôme*	*nom* *capital UV* *diplôme* ***Mandat*** ***Syndicat***
VérifierNom *MajUV* *ChangerDiplôme*	*VérifierNom* *MajUV* *ChangerDiplôme* ***DémissionnerMandat*** ***ChangerSyndicat***

- L'objet ***Etudiant-Elu*** a les propriétés (attributs et méthodes) de l'objet ***Etudiant*** mais en plus possède d'autres propriétés.

- La classe ***Etudiant-Elu*** est une spécialisation de la classe *Etudiant*. C'est une sous classe de la classe Etudiant.

- Les objets de la sous classe ***Etudiant-Elu*** héritent des attributs et des méthodes de la classe ***Etudiant***. La sous classe ***Etudiant-Elu*** pourra, si cela est nécessaire pour ses besoins, redéfinir une méthode héritée.

L 'Héritage

♦ Chaque sous classe peut avoir une ou plusieurs sous classes formant ainsi une hiérarchie d'objet. On parle de classe ***ancêtre*** (ou ***mère)*** et de classes ***descendant*** (ou ***fille)***.

♦ L'héritage est un mécanisme qui permet d'assurer une grande variabilité dans la réutilisation des objets. Il existe deux techniques liées à l'héritage : **les classes abstraites** et **l'héritage multiple**.

Les classes abstraites

C'est un type de classe ayant des propriétés qui ne permettent pas de préciser des instances. Ces classes mettent en commun un certain nombre de propriétés à des objets.

Exemple :

Soit la classe ***JeuneAdulte***

Graduation *Adresse* *téléphone* *Service Militaire* *RédigerCV* *AfficherCV*

Et :

La classe **Etudiant** :	La classe **Etudiant-Elu**:
nom *capital UV* *diplôme* *VérifierNom* *MajUV* *ChangerDiplôme*	*nom* *capital UV* *diplôme* ***Mandat*** ***Syndicat*** *VérifierNom* *MajUV* *ChangerDiplôme* ***DémissionnerMandat*** ***ChangerSyndicat***

La Classe *JeuneAdulte* a des propriétés communes aux classes Etudiant et *Etudiant-Elu.* Mais on ne peut l'instancier.

L 'Héritage multiple

L'héritage multiple permet à une classe d'avoir plusieurs classes antécédents et d'hériter ainsi de tous les attributs et méthodes de ces ancêtres.

Exemple

Classe **C1** :
At1 *At2* *Mét1* *Mét2*

Classe **C2** :
At1 ; At21 *At2 ; At22* *At23* *Mét1 ; Mét21* *Mét2 ; Mét22*

Classe **C3** :
At1 ; At31 *At2 ; At32* *At33* *Mét1 ; Mét31* *Mét2 ; Mét32* *Mét33*

Soit la classe **C50** :
At1 ; At2 ; *At21 ;At22* *At23 ; At31 ;* *At32, At33;At51* *Mét1 ; Mét21* *Mét2 ; Mét22* *Mét31; Mét32 .* *Mét33* *Mét51*

La classe *C50* hérite des classes *C1, C2* et *C3.*

Problème :

- La classe *C50* héritera-t-elle 2 fois des attributs *At1* et *At2* ?
- Si la méthode *Mét2* a été modifiée dans *C2* et *C3* alors laquelle des deux héritera la classe *C50 ?*

Le polymorphisme

C'est un mécanisme qui permet à une sous classe de redéfinir une méthode dont elle a hérité tout en gardant la même signature de la méthode héritée.

Ainsi on peut avoir une méthode avec la même tête (même signature) et des corps différents (codes différents) : **polymorphisme**.

Un même message peut ainsi déclencher des traitements différents selon l'objet auquel il fait appel.

Un message polymorphe poserait un problème à la compilation statique car on ne saurait identifier précisément la méthode qu'il vise.

On ne pourra le savoir qu'au moment de l'exécution du programme. C'est la compilation dynamique qui permettra de résoudre ce problème.

Mohamed Chraibi

les différentes étapes :

méthode : **guide de description d'une forme de modèle à une autre.**

formalisme : **langage de représentation graphique.**

- ☑ Expression des besoins
- ☑ Spécification
- ☑ Analyse
- ☑ Conception
- ☑ Implémentation
- ☑ Tests de vérification
- ☑ Validation
- ☑ Maintenance et évolution

Les différentes étapes (1)

✓ Expression des besoins :

...

✓ Spécification :

Ce que le système doit être et comment il peut être utilisé.

✓ Analyse :

L'objectif est de déterminer les éléments intervenant dans le système à construire, ainsi que leur structure et leurs relations .

Elle doit décrire chaque objet selon 3 axes :

- Axe fonctionnel : savoir-faire de l'objet.
- Axe statique : structure de l'objet.
- Axe dynamique : cycle de vie de l'objet au cours de l'application (Etats et messages de l'objet).

Ces descriptions ne tiennent pas compte de contraintes techniques (informatique).

U.M.L. Les différentes étapes (2)

✓ La conception :

Elle consiste à apporter des solutions techniques aux descriptions définies lors de l'analyse : architecture technique ; performances et optimisation ; stratégie de programmation.

On y définit les structures et les algorithmes.

Cette phase sera validée lors des tests.

✓ L'implémentation :

C'est la réalisation de la programmation.

Les différentes étapes (3)

✓ Les tests de vérification :

Ils permettent de réaliser des contrôles pour la qualité technique du système.

Il s'agit de relever les éventuels défauts de conception et de programmation (revue de code, tests des composants,...).

Il faut instaurer ces tests tout au long du cycle de développement et non à la fin pour éviter des reprises conséquentes du travail (programmes de tests robustes ; Logiciels de tests).

✓ Validation :

- Le développement d'une application doit être lié à un contrat ayant une forme de cahier de charges, où doivent se trouver tous les besoins de l'utilisateur. Ce cahier de charge doit être rédigé avec la collaboration de l'utilisateur et peut être par ailleurs complété par la suite.
- Tout au long des ces étapes, il doit y avoir des validations en collaboration également avec l'utilisateur.
- Une autre validation doit aussi être envisagée lors de l'achèvement du travail de développement, une fois que la qualité technique du système est démontrée. Elle permettra de garantir la logique et la complétude du système.

Les différentes étapes (4)

✓ Maintenance et évolution

Deux sortes de maintenances sont à considérer :

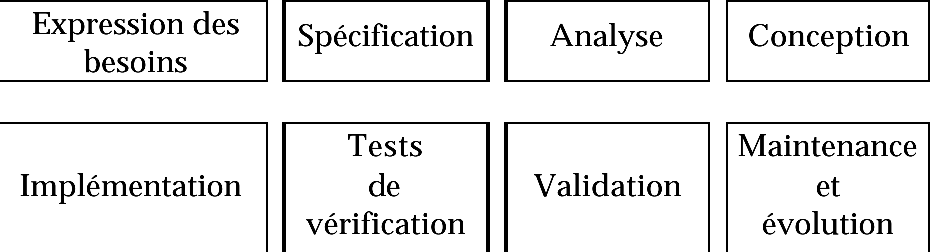
- Une maintenance corrective, qui consiste à traiter les “buggs ”.

- Une maintenance évolutive, qui permet au système d’intégrer de nouveaux besoins ou des changements technologiques.

Les différents cycles de vie

Il existe 2 cycles de vie utilisées dans les approches traditionnelles : le *modèle linéaire* et le *modèle en “ V ”*.

Le modèle linéaire

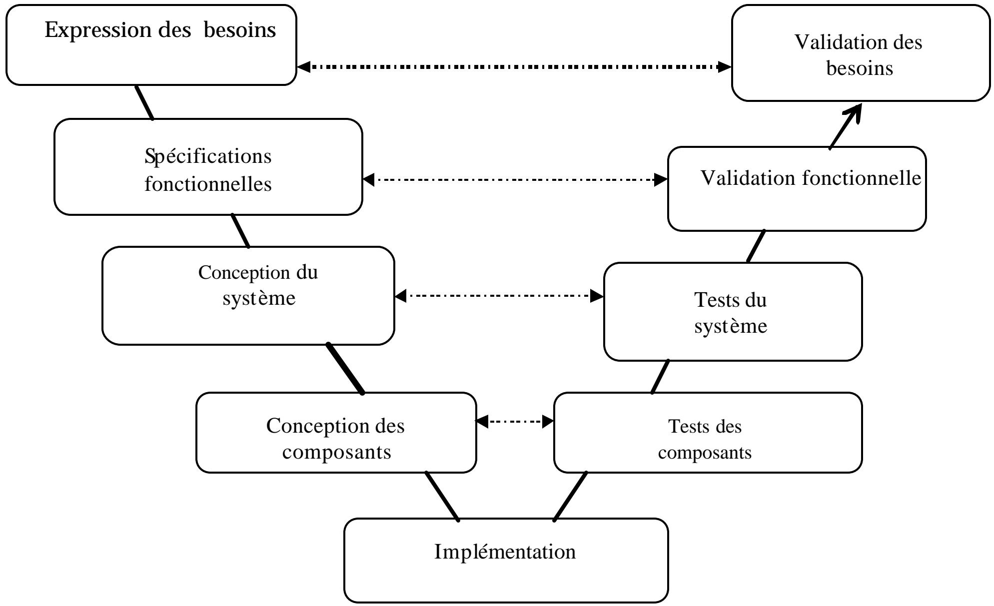


Le principe de cette démarche est que chaque phase est traitée complètement avant que la suivante ne soit entamée.

Ce qui renvoie les tests de vérification et la validation en fin du processus de développement.

S’il y a des erreurs, les retours seront compliqués et coûteraient chers.

Le modèle en “ V ”



Le modèle en “V” permet une **organisation modulaire**.

•A chaque étape de l’analyse et de la conception correspond une étape de tests ou de validation.

•A chaque étape fonctionnelle correspond ainsi une étape technique.

Le processus s’accomplit en deux phases :

- **Une phase descendante** : de spécifications et de conception.
- **Une phase ascendante** : de tests et de validation.

Comme pour le modèle linéaire, l’inconvénient est que la validation et les tests interviennent tardivement.

Les différents cycles de vie

Le cycle de vie Objet

Dans un projet Objet, le cycle de vie répond à 3 caractéristiques essentielles :

- ❖ *La traçabilité entre les étapes*
- ❖ *Un cycle itératif*
- ❖ *Un cycle incrémental*

☞ ***La traçabilité entre les étapes :***

• Les concepts utilisés au cours des différentes étapes sont quasiment identiques (Classes, Objets, Attributs, Méthodes, Héritage, Polymorphisme, ...).

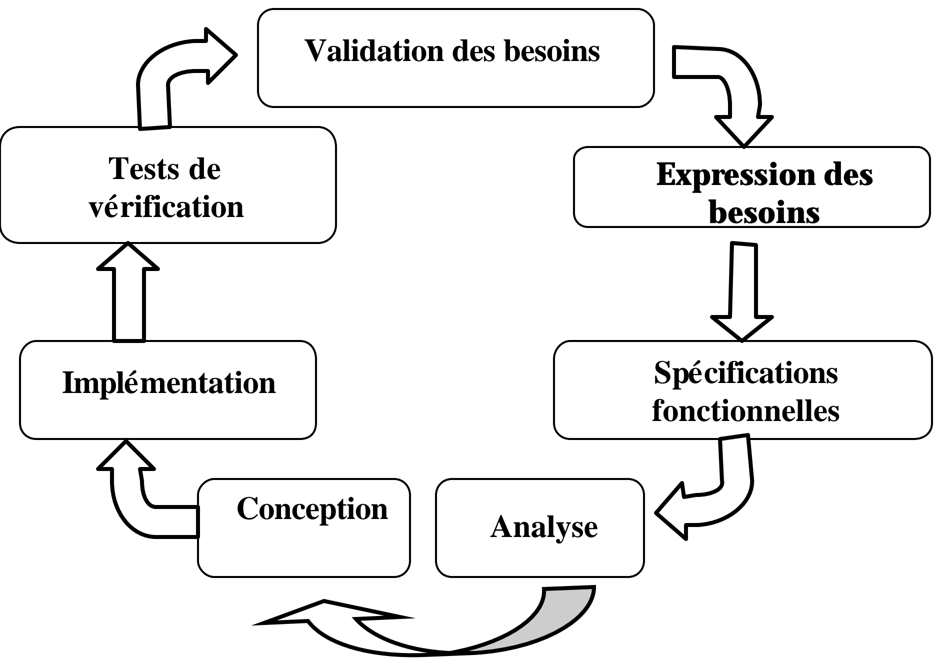
• Ceci permet de conserver le même discours lors de toutes les étapes :

« *Analyse - Conception - Implémentation* ».

• Ce qui n'est pas le cas dans les approches traditionnelles, où l'on utilise une méthode d'analyse et de conception avec des concepts et un langage de programmation avec d'autres concepts.

Le cycle de vie Objet

☞ ***Un cycle itératif :***



☞ ***Un cycle incrémental :***

- Lors du développement, une maquette doit être réalisée pour valider l'ergonomie de l'application et l'enchaînement des écrans.

- Plusieurs versions peuvent être développées. Lors de chacune d'elle, chaque fonctionnalité est améliorée jusqu'à optimisation rendant ainsi le système progressivement robuste.

Les USE CASES

Idée :

Les use cases (cas d'utilisation) sont un concept de la méthode OOSE de Ivar Jacobson.

- Ils permettent d'effectuer une délimitation du système et de décrire son comportement.
- Ils constituent une représentation orientée " fonctionnalités " du système.

Dans la modélisation par les use cases : 2 concepts fondamentaux interviennent :

- ***Les acteurs : utilisateurs du système.***
- ***Les uses cases : utilisation du système***

Les acteurs

Ceux sont les ***utilisateurs du système***

Ils ont une bonne connaissance des fonctionnalités du système. Ils constituent les éléments extérieurs du système.

Ils peuvent être :

»soit des humains ;

»soit des logiciels ;

»soit des automates.

On distingue *:*

- *les acteurs primaires* : ceux sont les utilisateurs du système ;
- *les acteurs secondaires* : ceux qui administrent le système.

Les uses cases

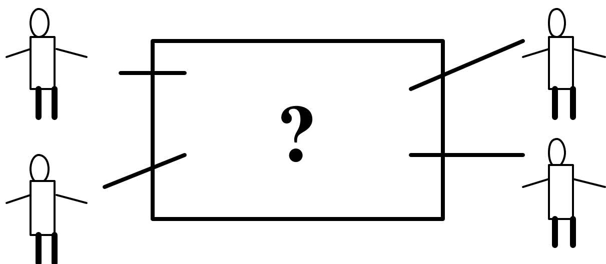
Ceux sont les ***utilisations du système***

Il s'agit de déterminer les éléments constitutifs d'un point de vue fonctionnel.

On pourra trouver des use cases pour décrire :

- chaque tâche de l'utilisateur ;
- les fonctionnalités mal décrites lors des spécifications ;
- les E/S des données ;
- les cas d'anomalies**.**

Représentation des acteurs dans un modèle Use Cases :



Exemple de Use case

U.M.L. Description d'un Use Case

Il existe plusieurs façons de décrire un use case.

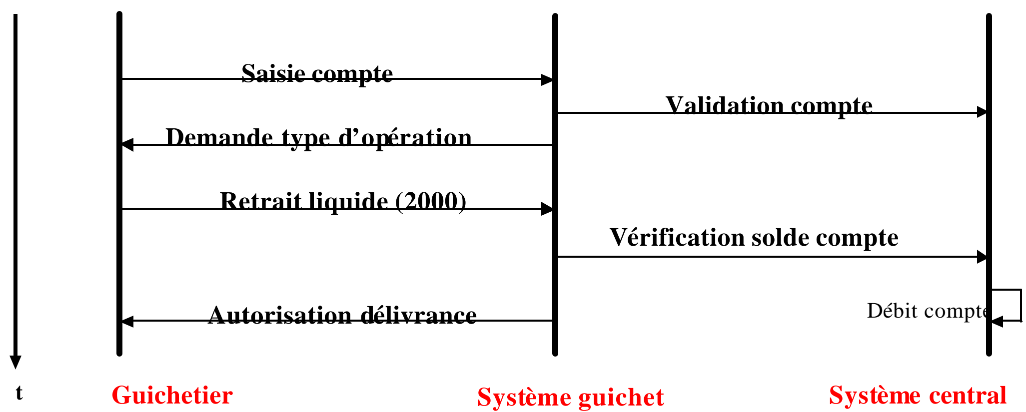
✓ Description textuelle (informelle) :

Exemple :

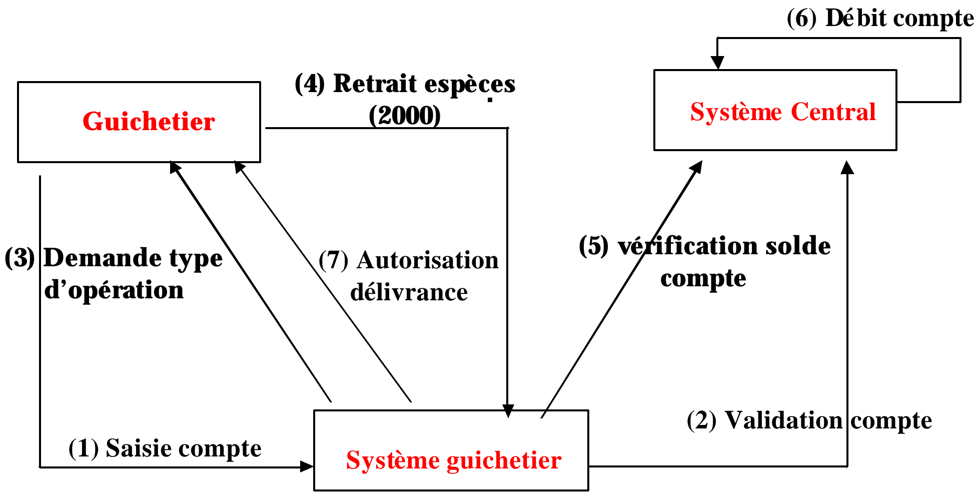
Use case : " ***Retrait en espèce*** " :

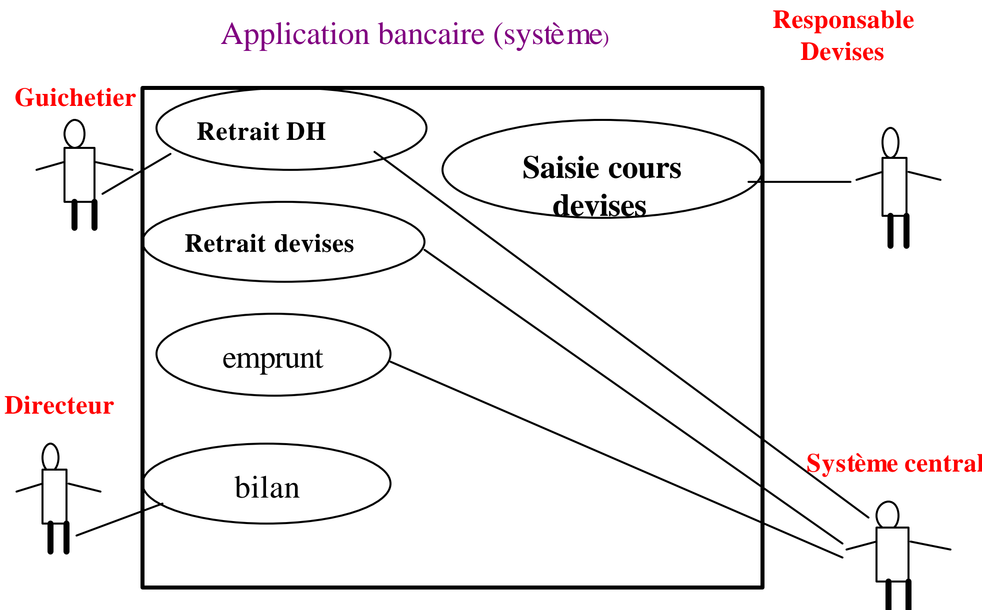
- Le guichetier saisit le n° de compte du client.
- L'application valide le compte auprès du système central.
- L'application demande le type d'opération au guichetier.
- Le guichetier sélectionne un retrait d'espèces de 2000DH.
- Le système " guichetier " interroge le système central pour s'assurer que le compte est suffisamment approvisionné.
- Le système central effectue le débit du compte.
- Le système notifie au guichetier qu'il peut délivrer le montant demandé.

✓ Exemple de scénario



✓ Exemple de diagramme de collaboration





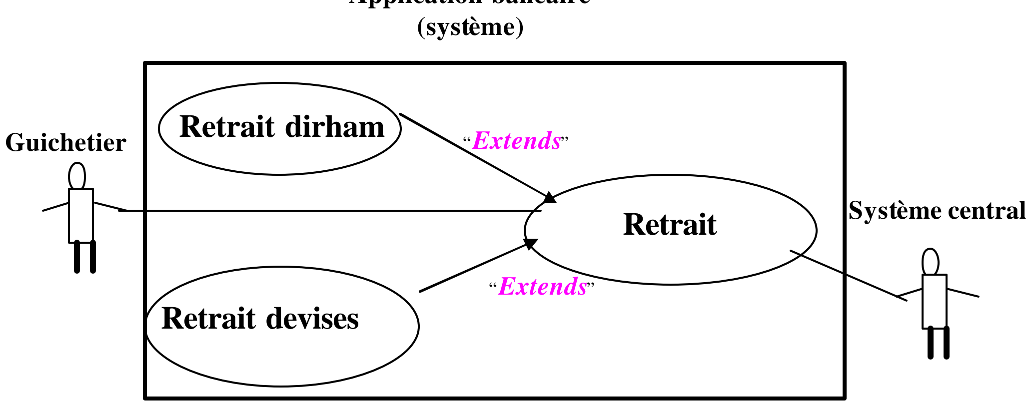
L'intégration dans l'Use Case " *Application bancaire* " des use cases de chaque opération permet d'avoir une vision globale du système. Elle permet également de comprendre le rôle de chaque acteur.

.

Relation « extends »

La relation “extends”

Un ou plusieurs use cases peuvent hériter des caractéristiques d’un autre use case.



Les *Uses Cases fils* ont les mêmes liens avec les acteurs et les autres use cases que le use case dont ils héritent.

Ceux sont de cas particuliers du *Uses Case père*.

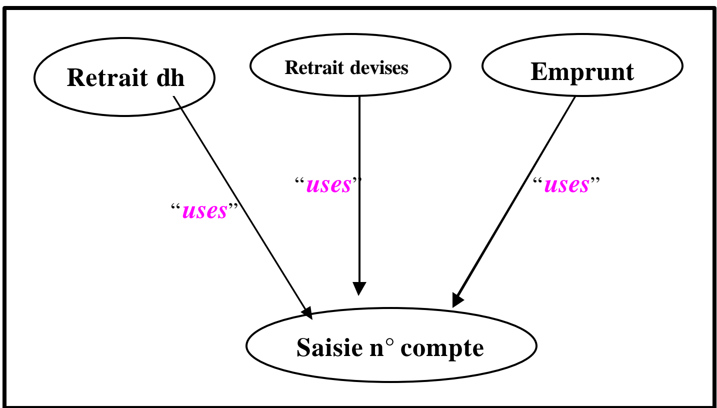
Relation « uses »

La relation “uses”

Soit l’use case “**Saisie n° compte**”

- Le guichetier saisit le code de la banque du compte.
- Il saisit le numéro du compte.
- Il saisit la clé du compte.
- Le système calcule la clé du compte et vérifie qu’elle est bonne.
- Le système interroge le compte sur le système central.
- Le système affiche le compte ainsi que son détenteur.

Application bancaire (système)



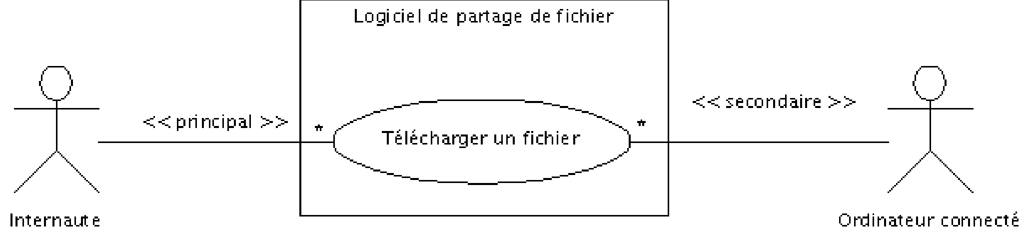
Lorsque une ou plusieurs tâches sont utilisées régulièrement, on peut les factoriser dans un même use case et faire de telle sorte que d’autres use cases l’utilisent en le pointant par une flèche.

Cet use case est en fait une sous partie de chaque use case qui l’utilise. Ce qui permet de décomposer un use case complexe en plusieurs uses cases.

Relations dans les diagrammes de cas d'utilisation

Relations entre acteurs et cas d'utilisation

Relation d'association



Une relation d'association est chemin de communication entre un acteur et un cas d'utilisation et est représenté un trait continu

Multiplicité

Il est possible d'ajouter une multiplicité sur l'association du côté du cas d'utilisation. Le symbole * *n*, *n..m*

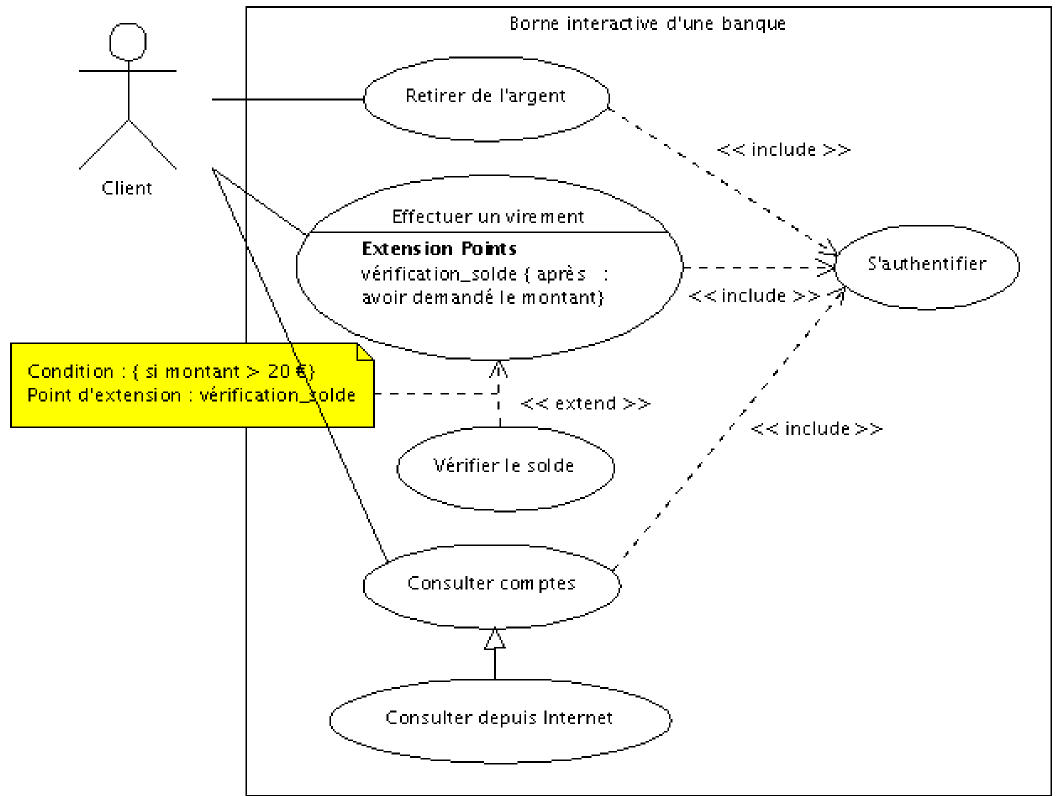
Préciser une multiplicité sur une relation n'implique pas nécessairement que les cas sont utilisés en même temps.

Acteurs principaux et secondaires

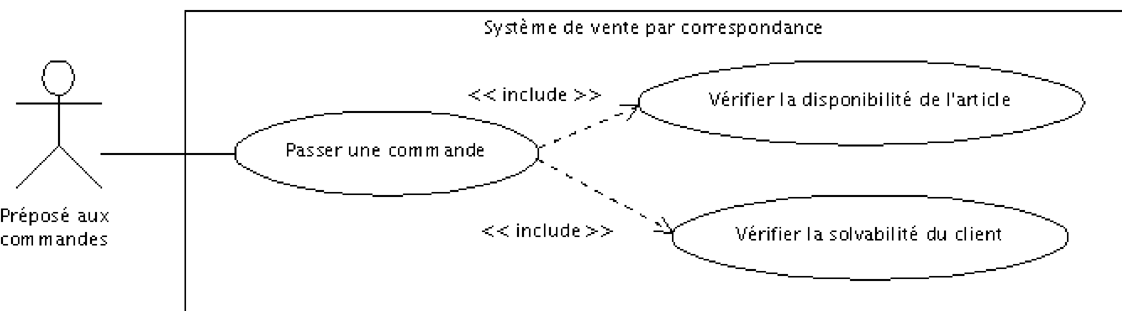
Un cas d'utilisation a au plus un acteur principal.

Un acteur principal obtient un résultat observable du système tandis qu'un acteur secondaire est sollicité pour des informations complémentaires.

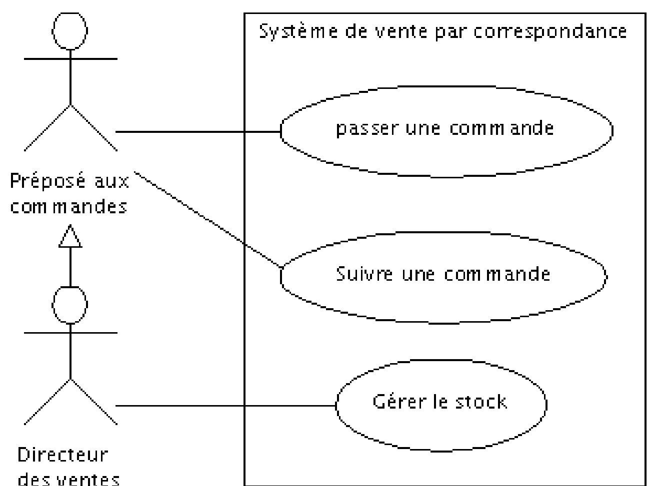
Relations entre cas d'utilisation



Relations entre cas pour décomposer un cas complexe

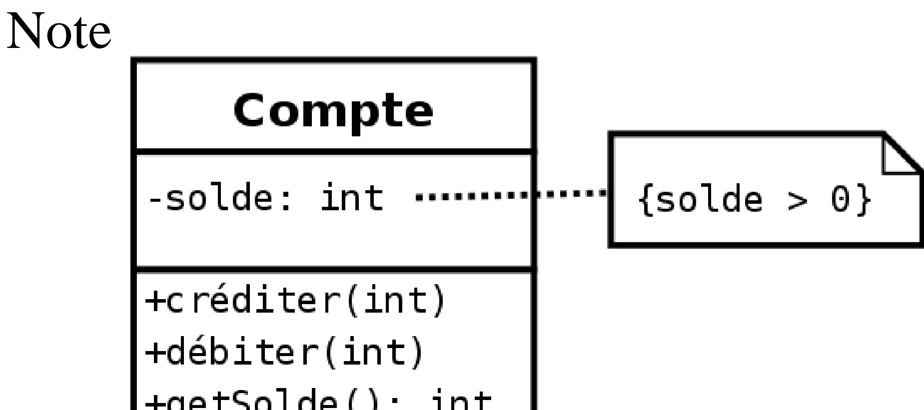


Relations entre acteurs



La seule relation possible entre deux acteurs est la généralisation : un acteur A est une généralisation d'un acteur B si l'acteur A peut être substitué par l'acteur B. Dans ce cas, tous les cas d'utilisation accessibles à A le sont aussi à B, mais l'inverse n'est pas vrai.

Note



Une note contient une information textuelle comme un commentaire, un corps de méthode ou une contrainte

- **Comment identifier les acteurs ?**

UML n'emploie pas le terme *d'utilisateur* mais d'*acteur*. Les acteurs d'un système sont les entités externes à ce système qui interagissent (saisie de données, réception d'information, ...) avec lui. Les acteurs sont donc à l'extérieur du système et dialoguent avec lui. Ces acteurs permettent de cerner l'interface que le système va devoir offrir à son environnement.

Oublier des acteurs ou en identifier de faux conduit donc nécessairement à se tromper sur l'interface et donc la définition du système à produire.

- **Comment recenser les cas d'utilisation**

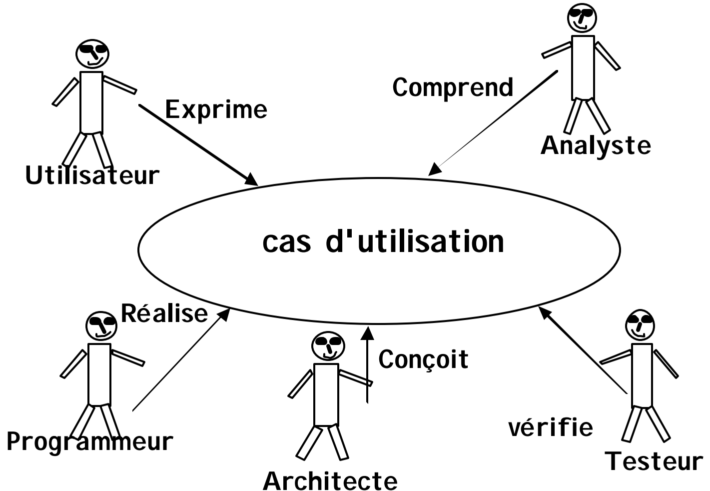
L'ensemble des cas d'utilisation doit décrire exhaustivement les exigences fonctionnelles du système. Chaque cas d'utilisation correspond donc à une fonction métier du système

- **Description textuelle des cas d'utilisation**

Le diagramme de cas d'utilisation décrit les grandes fonctions d'un système du point de vue des acteurs, mais n'expose pas de façon détaillée le dialogue entre les acteurs et les cas d'utilisation.

Description textuelle des cas d'utilisation

- **Nom :**
  - Utiliser une tournure à l'infinitif (ex : Réceptionner un colis).
- **Objectif :**
  - Une description résumée permettant de comprendre l'intention principale du cas d'utilisation. Cette partie est souvent renseignée au début du projet dans la phase de découverte des cas d'utilisation.
- **Acteurs principaux :**
  - Ceux qui vont réaliser le cas d'utilisation (la relation avec le cas d'utilisation est illustrée par le trait liant le cas d'utilisation et l'acteur dans un diagramme de cas d'utilisation)
- **Acteurs secondaires :**
  - Ceux qui ne font que recevoir des informations à l'issue de la réalisation du cas d'utilisation
- **Dates :**
  - Les dates de créations et de mise à jour de la description courante.
- **Responsable :**
  - Le nom des responsables.
- **Version :**
  - Le numéro de version.
- **Les préconditions :**
  - elles décrivent dans quel état doit être le système (l'application) avant que ce cas d'utilisation puisse être déclenché.
- **Des scénarii :**
  - Ces scénarii sont décrits sous la forme d'échanges d'évènements entre l'acteur et le système. On distingue le scénario nominal, qui se déroule quand il n'y a pas d'erreur, des scénarii alternatifs qui sont les variantes du scénario nominal et enfin les scénarii d'exception qui décrivent les cas d'erreurs.
- **Des postconditions :**
  - Elle décrivent l'état du système à l'issue des différents scénarii.



Les cas d'utilisation servent de fil conducteur pour l'ensemble du proie

Le Modèle Objet

DÉMARCHE D'APPLICATION D'UML

Nous proposons de suivre une démarche structurée en sept étapes.

Étape 1 : **élaboration d'un diagramme de contexte du système à étudier** : comme nous l'avons déjà dit dans la présentation d'OMT, il est important de démarrer une analyse par le positionnement le plus précis possible du champ du système à étudier. No recommandons donc d'élaborer un diagramme de contexte du système à étudier.

Étape 2 : **identification et représentation des cas d'utilisation** : les fonctions du système sont identifiées en recherchant les cas d'utilisation du système qui seront mis en oeuvre par les différents acteurs. Le *diagramme des cas d'utilisation* est élaboré.

Étape 3 : **description et représentation des scénarios** *chaque* cas d'utilisation se traduit par un certain nombre de scénarios. Chaque scénario fait l'objet d'une description. Chaque scénario est ensuite décrit sous forme graphique à l'aide du diagramme *de séquence* et/ou *diagramme de collaboration.*

Étape 4 : **identification des objets et classes** *: une* première identification des objets classes est fournie par la synthèse des diagrammes de séquence et/ou de collaboration Ainsi une liste de tous les objets et toutes les classes manipulés peut être dressée.

Étape 5 : **élaboration du diagramme de classe** :

à partir des classes identifiées, une première version du modèle objet est élaborée. Les classes du modèle objet corresponde soit à des préoccupations métier soit à des nécessités techniques.

Étape 6 : **élaboration du diagramme état-transition**:

pour chaque classe importante c'est à dire présentant un intérêt pour le système à modéliser, un diagramme état-transition est élaboré.

Étape 7 : **consolidation et vérification des modèles** :

le concepteur doit ensuite dérouler les étapes 3, 4, 5 et 6 jusqu'au moment où il considérera qu'il atteint le niveau suffisant pour la description du système.

Le Modèle Objet

Cela consiste à définir les objets qui vont modéliser les besoins qui ont été exprimé en termes de fonctionnalités.

Le passage de cet aspect fonctionnel à un aspect objet n'est certes pas évident.

La description des objets est **structurelle**.

Par ailleurs, on déterminera les **liens** entre les différents objets.

Les Objets et leurs liens représentent ainsi le

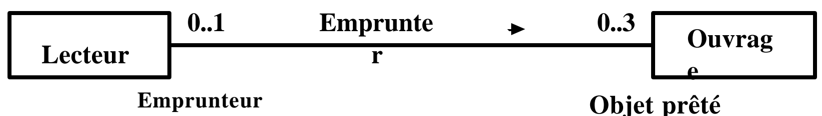
modèle statique

Les objets déterminés serviront lors des phases analyse, conception et plus tard à l'implémentation.

Concept de classe et d' objet

Les objets du modèle statique sont une représentation (modélisation) des objets (monde réel), qui seront en général ceux qu'on retrouve lors de l'implémentation sous la même forme ou sous une forme différente.

Ils sont munis de données encapsulées dans les objets, représentant leurs attributs et leurs opérations (méthodes).

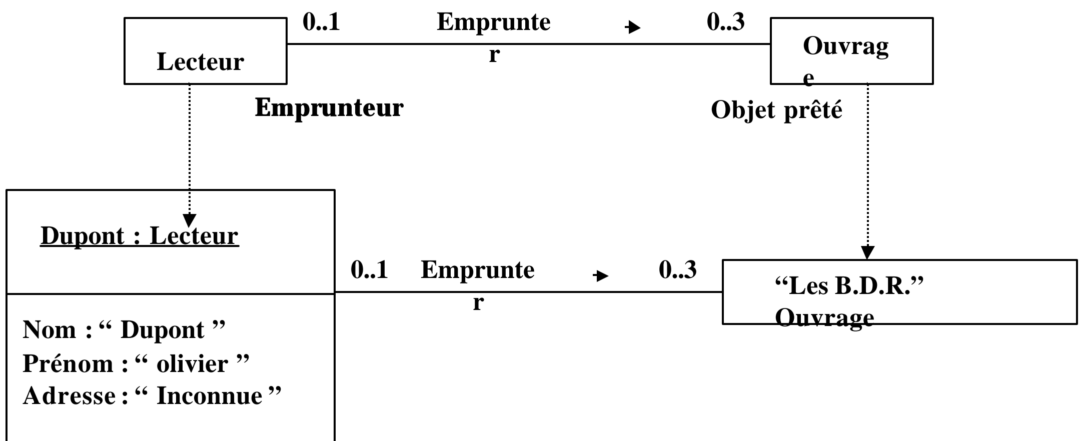


Exemple de classes

Chaque objet d'une classe a une identité propre et n'a donc pas besoin d'un identificateur, sauf si celui-ci est un identificateur métier préexistant comme un n° CIN.

- Le nombre d'attributs et de méthodes qu'on définira dépend du niveau de granularité qu'on veut obtenir.

Exemple :



Exemple de classes et d'objet

Association et Classe d 'associations

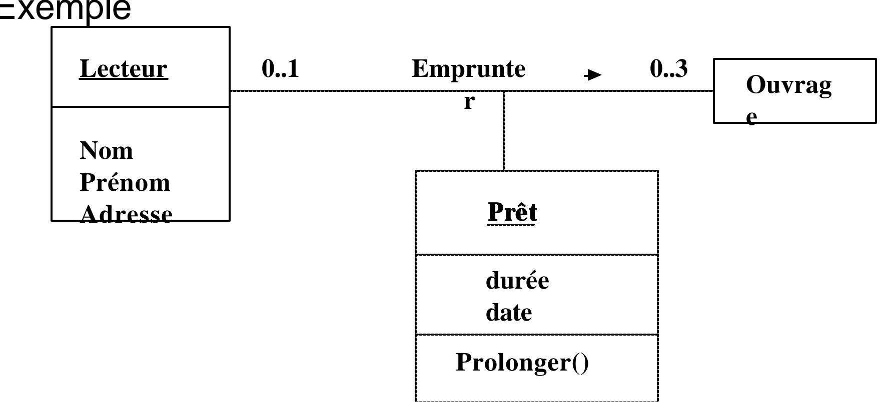
Entre les 2 classes Lecteur et Ouvrage , il existe un lien qui représente un emprunt d'ouvrage par un lecteur. Il est matérialisé par un lien entre les 2 classes et il peut être caractérisé par :

- **Un nom**
- **Deux noms de rôle facultatifs**
- **Un sens de lecture**
- **Deux cardinalités.**

Une cardinalité peut être représentée par un **nombre**, une "*" par l'infini ou un **intervalle**.

Une association peut nécessiter des données et aussi des opérations : il est alors tout indiqué de lui construire une classe.

Exemple



Classe d'association

On peut choisir parfois entre rajouter une donnée dans une classe ou créer une classe propre.

D'autre part, il est possible de mettre la donnée dans une structure classique mais ceci peut s'avérer lourd à gérer et ne peut d'autre part assurer la persistance de la donnée.

Diagramme des Classes

Le modèle objet sera représenté par un **diagramme de classes** où chacune sera décrite avec ses attributs et ses méthodes :

Nom de la classe
attribut : type = valeur initiale
opération(liste d'arguments) : résultat renvoyé

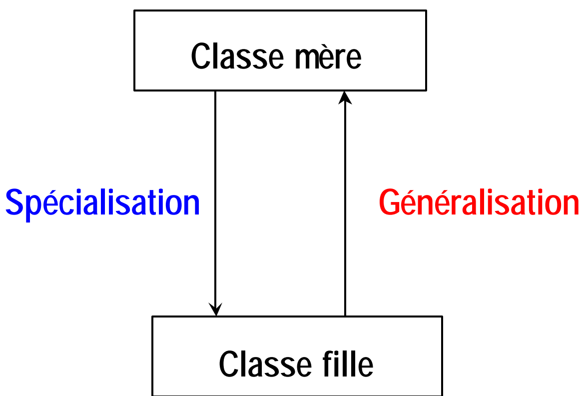
Diagramme d'une classe

- La détermination des classes lors de la phase d'analyse n'est pas évidente. **Elle suit une méthode plutôt intuitive**, basée sur l'expérience de l'analyste qui a (ou n'a pas) l'habitude de reconnaître plus ou moins facilement les classes, les objets , les associations, les attributs et les méthodes des classes.

- L'analyste pourra se demander alors quels sont **les objets de gestion** dans le problème étudié et se référer également aux **règles de gestion** pour identifier les **objets réels** ainsi que leurs **liens** et qu'il va falloir **modéliser sous formes de classes** et **d'associations.**

- **Le parallèle avec le modèle E/A est intéressant à faire lors de la phase d'analyse**.

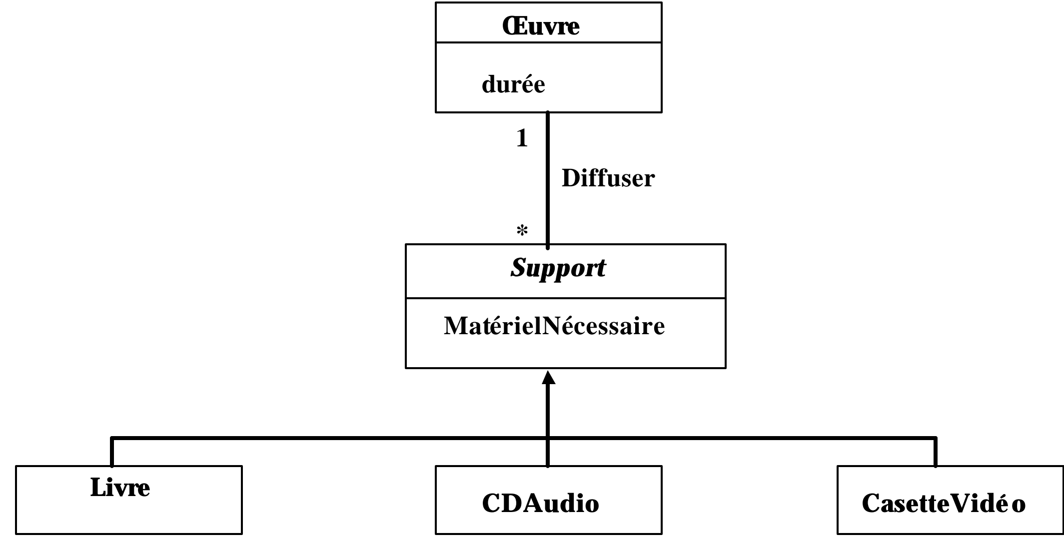
Généralisation ; Spécialisation



Concepts de Spécialisation et de Généralisation

La modélisation de la **notion d'héritage** dans un modèle statique peut se faire par l'intermédiaire de la **généralisation** qui permet une **organisation des classes** et des **regroupements sémantiques** de classes.

Exemple :



Exemple de classe abstraite

La **généralisation** permet de simplifier le diagramme des classes. La **spécialisation** permet d'établir une relation de type « ***est un*** » ou « ***est une sorte de*** ».

Classes abstraites

Il existe des classes qu'on ne peut instancier, car elle sont trop générales.

Elle servent à **mettre en commun des caractéristiques** communes à certaines classes, c'est le cas de la classe *Support* : C'est une **classe abstraite**.

Celle-ci doit toujours être suivie de ***classes dérivées***.

Dans l'exemple précédent les classes dérivées sont *Livre*, *CDAudio* et *CassetteVidéo*.

Elle permettent de représenter des concepts importants dans une application.

U.M.L. Héritage simple ou multiple (1)

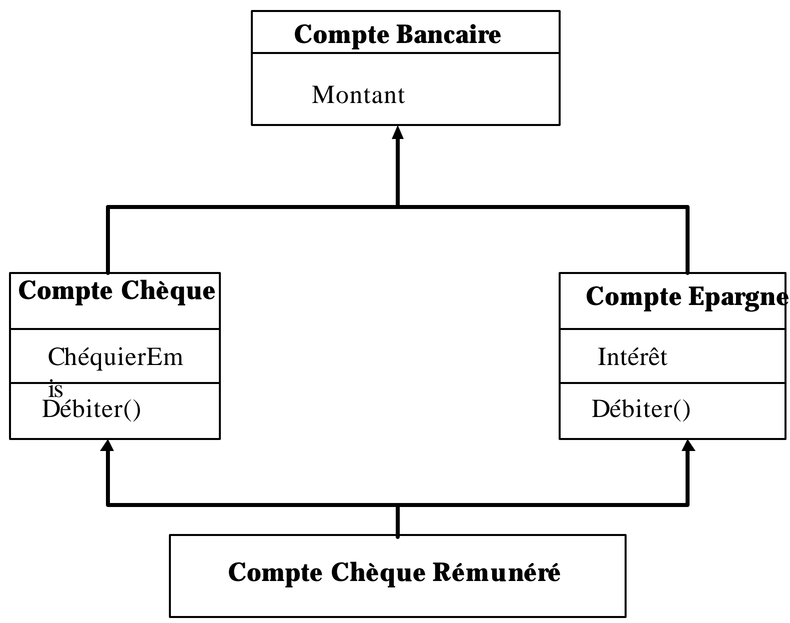
☑ **L'héritage** peut être **simple** (c.a.d. une classe qui hérite n'a qu'une seule classe mère) ou **multiple** (c.a.d. la classe a plusieurs classes mères).

☑ Ce dernier permet une structuration multiple du diagramme des classes, cependant il induit tout de même une certaine complexité.

☑ Tous les langages de programmation ne gèrent pas l'héritage multiple.

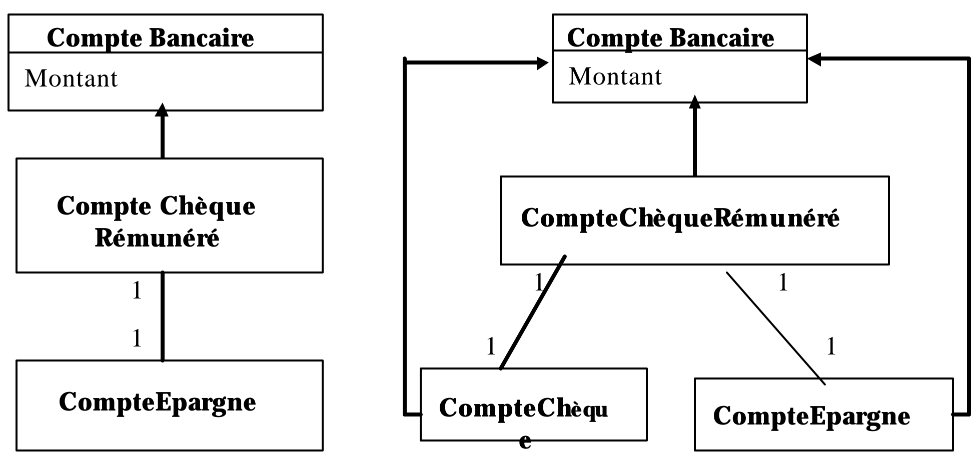
☑ On peut contourner l'héritage multiple dès la phase d'analyse.

Exemple :



Héritage Multiple

U.M.L. Héritage simple ou multiple (2)



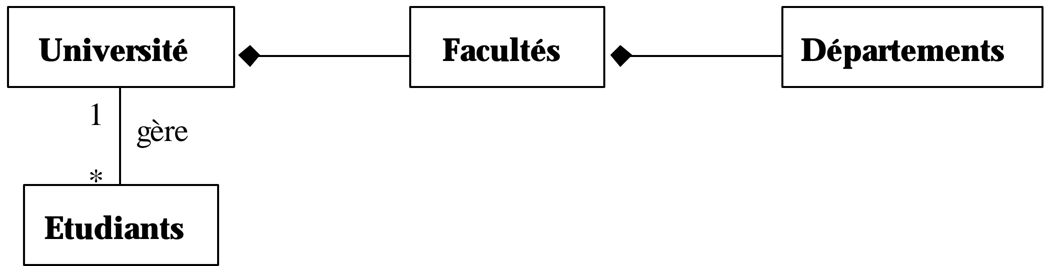
Héritage Simple : 2 solutions

Agrégation

Lorsqu'une association entre deux instances d'une classe a en plus une particularité dont le sens est du style : "*une instance est composée d'une ou plusieurs autres instances* ", on peut alors qualifier cette association d'***agrégation***.

On peut dire également qu'une agrégation est une association de type " **Composé–Composant** ". Où, l'instance ***composé*** est **l'agrégat** et *les* ***composants*** sont les **instances** agrégées.

Exemple :



Agrégation

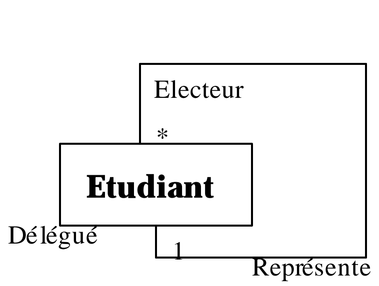
Agrégation (2)

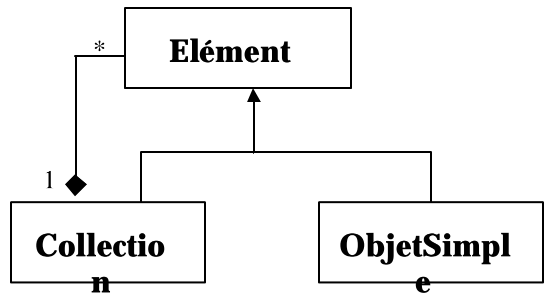
Une agrégation peut être perçue comme une association. Cependant une association ne peut être une agrégation.

Si une association a les caractéristiques suivantes, elle peut alors être représentée par une agrégation :

- L'association a une sémantique de style " *est composée de ...* " ou *" est une partie de ... "*.
- Il existe une forte différence de granularité entre une classe (l'agrégat) et d'autres classes (les agrégés).
- La suppression d'un objet agrégat ferait disparaître les objets agrégés.
- La modification d'un attribut d'un objet agrégat porte aussi sur les attributs des objets agrégés.
- La définition d'une méthode de l'objet agrégat repose sur celles des objets agrégés et peuvent porter dailleurs le même nom.

Une classe peut avoir des **instances qui sont en association entre elles**.





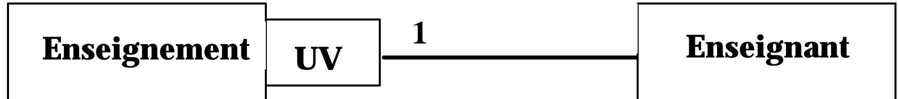
Association unaire (réflexive)

Agrégation récursive

Le rôle d'un **qualificateur** est de réduire la cardinalité d'une association et **joue le rôle semblable à une clé primaire** dans une BDR.

Il permet de tenir un dictionnaire composé de :

qualificateur ⇨ objet(s) qualifié(s).

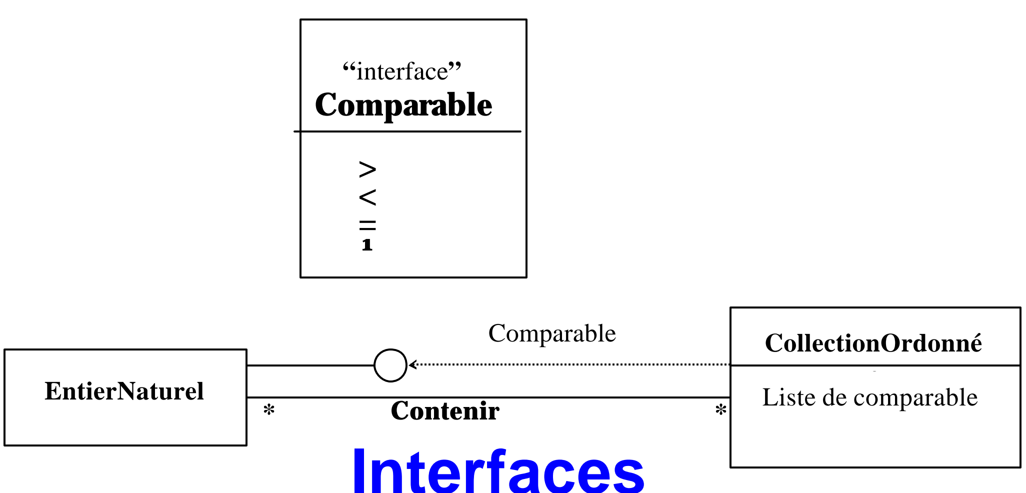


Association unaire Agrégation récursive(réflexive)

Une **interface** est une classe qui ne peut contenir que des opérations.

Elle ne véhicule que la sémantique de ses opérations et ne dit rien sur la façon de les implémenter.

On dit alors qu'une classe qui implémente ces opérations implémente l'interface.



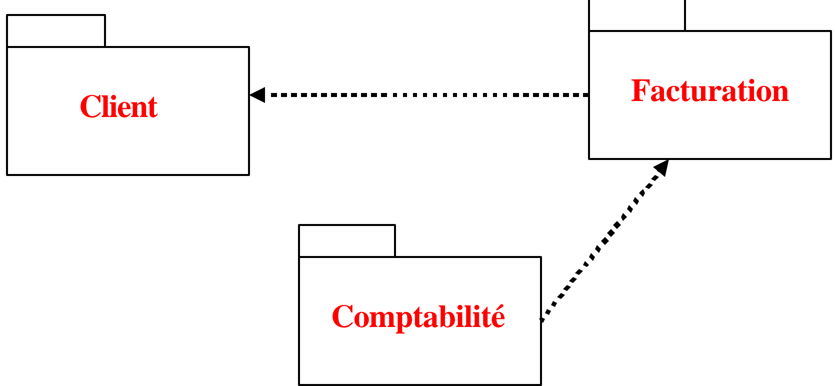
Interfaces

- On peut créer plusieurs interfaces et les faire hériter entre elles.
- Les interfaces présentent un caractère d'aptitude que d'autres classes ne peuvent encapsuler.
- C'est ce qui permet de distinguer entre une généralisation et une interface.

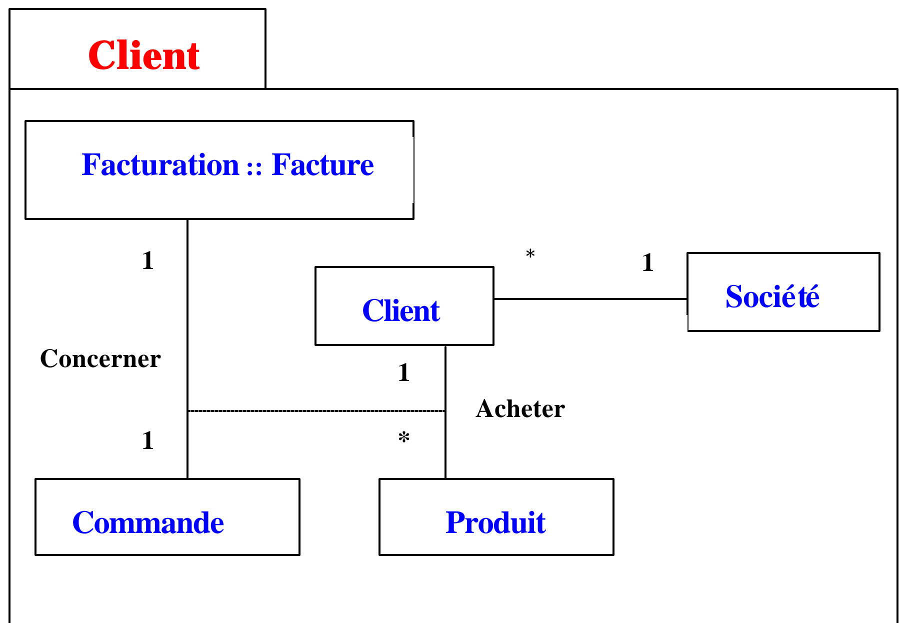
Un **package** permet de regrouper un ensemble de classes, d'associations et éventuellement d'autres packages.

Il permet de découper une système en plusieurs parties représentées chacune par un package.

Les packages peuvent avoir des actions entre eux.



Organisation par packages



Contenu d'un package

Une classe peut apparaître dans différents packages (avec le même nom).

On y trouve même des classes qui n'appartiennent pas au package mais qui sont référencées par les classes propres.

On désigner une classe d'un package par :

nomPackage :: nomClasse

C’est un concept qui permet des **regroupements** de classes, d’associations, de méthodes, d’attributs, de packages…

Il permet de créer des **familles** d’éléments associés à un même stéréotype.

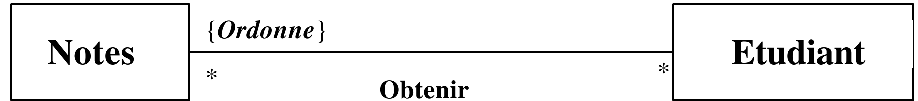
D’autre part, il permet de modifier la sémantique des éléments associés et crée par la même occasion des concepts propres à une application.

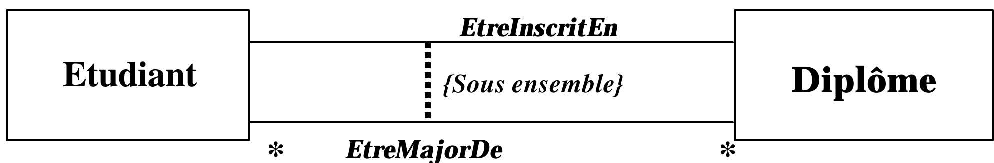
L’interface est un exemple de stéréotype.

Contenu d’un package

Elles permettent d'apporter plus de **précisions** à un élément du modèle.

Exemple :





Ile vient juste après **l'analyse statique** du modèle.

Cette dernière décrit la structure des éléments du modèle ainsi que leurs relations.

Quant à **l'analyse dynamique**, elle a pour but de **décrire** les **états** des objets au cours du fonctionnement du systèmes :

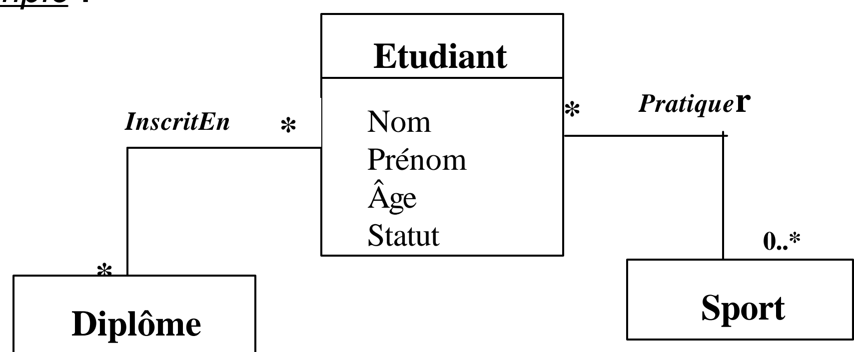
- modifications des attributs ;
- exécutions des méthodes ;
- réactions à des sollicitations externes au système;…

La notion d 'Etat

Un **état** d'un objet est défini à la fois par la **valeurs de ses attributs** et de ses **liens avec les autres objets**.

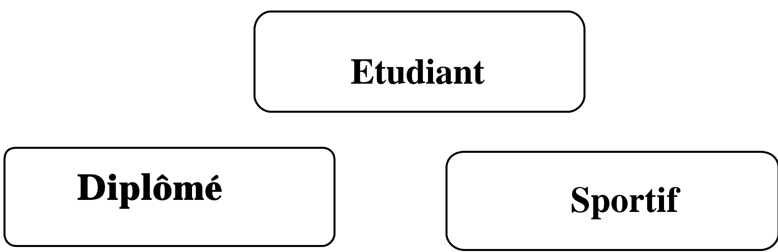
Il représente ainsi un ***intervalle de temps***.
L'objet est dans un état initial et peut alors changer d'état.

Exemple :



Ici l'objet *Etudiant* peut passer d'un ***état Etudiant*** à un ***état de diplômé*** à un ***état de sportif***, …
C'est l'attribut ***Statut*** qui va changer de valeur.

Représentation d 'un état :



Evènements et messages

Un **événement** est produit par un fait et véhicule une information qui va solliciter un ou plusieurs objets.

Soit ils répondront en **activant une méthode** ; soit ils ne réagiront pas du tout. Cela dépend de l'état dans lequel se trouve(nt) l' (les) objet(s).

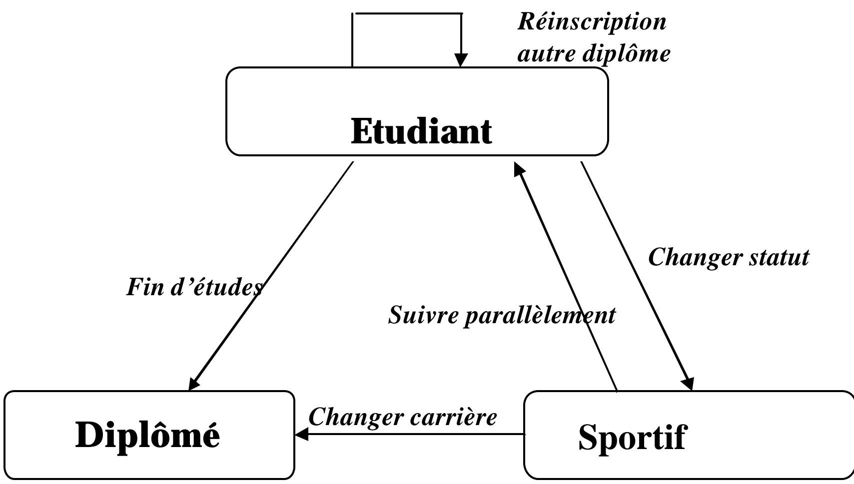
Un événement peut être **interne** ou **externe** au système.

Lorsqu'un objet provoque un événement en direction d'un autre objet, ce dernier peut répondre en déclenchant une de ses méthodes.

C'est une **interaction** entre ces deux objets. L'événement est qualifié alors de **message** entre ces deux objets.

Transition

Une **transition** est le passage d'un objet d'un état à un autre dû à un événement.



• Le point de vue spécification met l'accent sur les interfaces des classes plutôt que sur leurs contenus.
• Le point de vue conceptuel capture les concepts du domaine et les liens qui les lient. Il s'intéresse peu ou prou à la manière éventuelle d'implémenter ces concepts et relations et aux langages d'implémentation.
• Le point de vue implémentation, le plus courant, détaille le contenu et l'implémentation de chaque classe.

Une démarche couramment utilisée pour bâtir un diagramme de classes consiste à :

Trouver les classes du domaine étudié.

Cette étape empirique se fait généralement en collaboration avec un expert du domaine. Les classes correspondent généralement à des concepts ou des substantifs du domaine.

Trouver les associations entre classes.

Les associations correspondent souvent à des verbes, ou des constructions verbales, mettant en relation plusieurs classes, comme << est composé de >>, << pilote >>, << travaille pour >>. *Attention, méfiez vous de certains attributs qui sont en réalité des relations entre classes.*

Trouver les attributs des classes.

Les attributs correspondent souvent à des substantifs, ou des groupes nominaux, tels que << la masse d'une voiture >> ou << le montant d'une transaction >>. Les adjectifs et les valeurs correspondent souvent à des valeurs d'attributs. Vous pouvez ajouter des attributs à toutes les étapes du cycle de vie d'un projet (implémentation comprise). N'espérez pas trouver tous les attributs dès la construction du diagramme de classes.

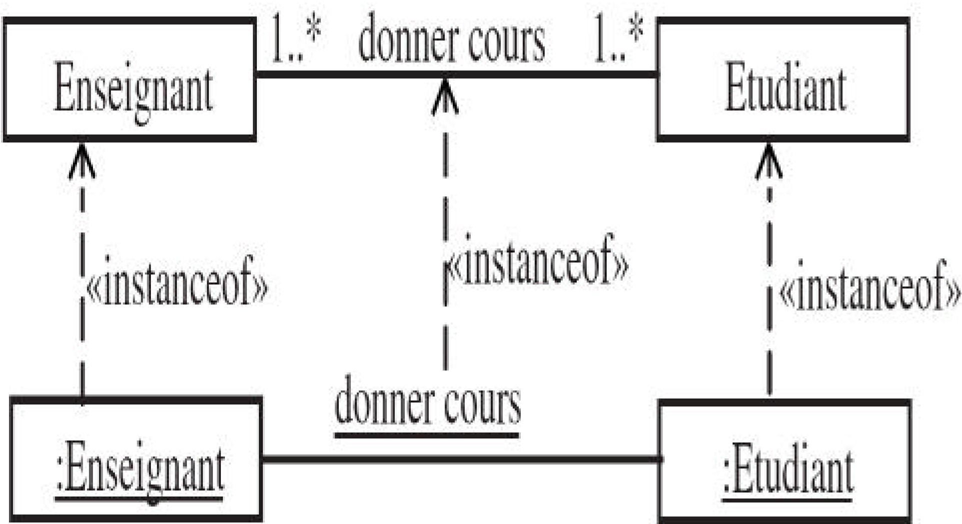
Organiser et simplifier le modèle

en éliminant les classes redondantes et en utilisant l'héritage.

Vérifier les chemins d'accès aux classes.

Itérer et raffiner le modèle.

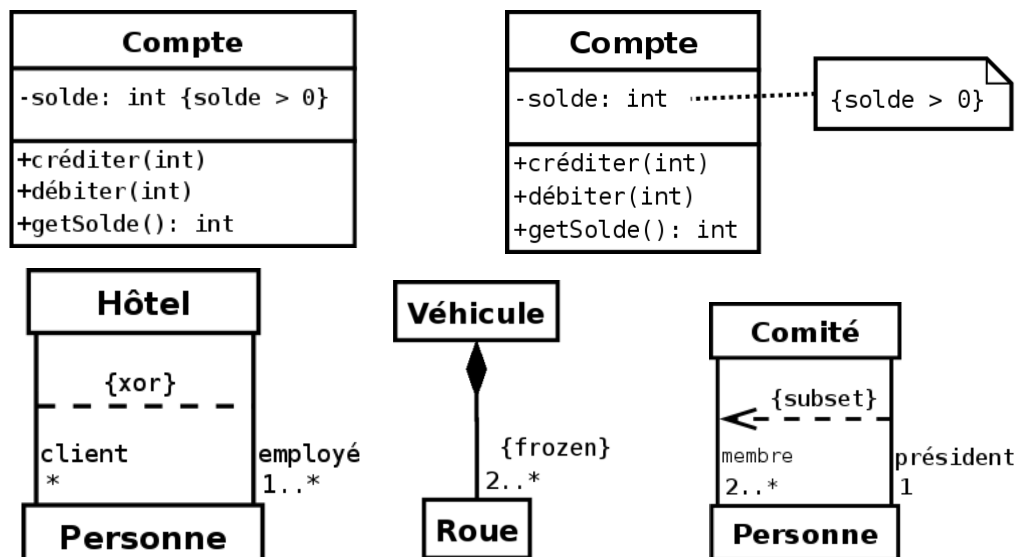
Un modèle est rarement correct dès sa première construction. La modélisation objet est un processus non pas linéaire mais itératif.



La relation de dépendance d'instanciation (stéréotypée << *instanceof* >>) décrit la relation entre un classeur et ses instances. Elle relie, en particulier, les liens aux associations et les objets aux classes.

Langage de contraintes objet

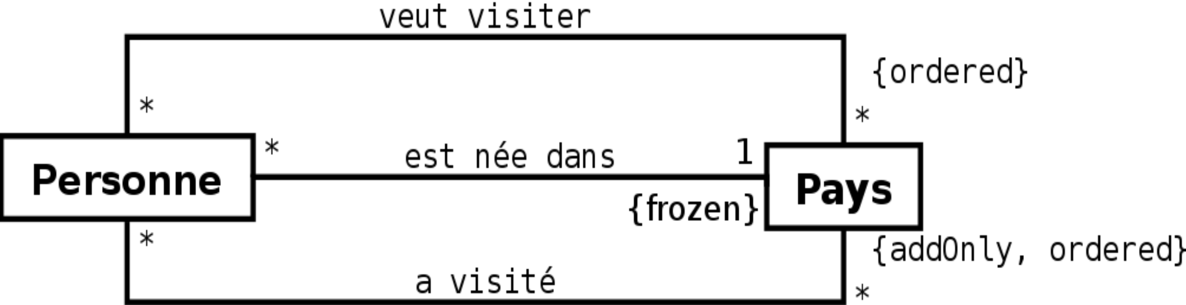
- **Expression des contraintes en UML**
- **Contraintes structurelles :**
  - les attributs dans les classes, les différents types de relations entre classes (généralisation, association, agrégation, composition, dépendance), la cardinalité et la navigabilité des propriétés structurelles, etc.
- **Contraintes de type :**
  - typage des propriétés, etc.
- **Contraintes diverses :**
  - les contraintes de visibilité, les méthodes et classes abstraites (contrainte *abstract*), etc.



{frozen} précise que le nombre de roues d'un véhicule ne peut pas varier

{subset} précise que le président est également un membre du comité

{xor} (*ou* exclusif) précise que les employés de l'hôtel n'ont pas le droit de prendre une chambre dans ce même hôtel



Ce diagramme exprime que : une personne est née dans un pays, et que cette association ne peut être modifiée ; une personne a visité un certain nombre de pays, dans un ordre donné, et que le nombre de pays visités ne peut que croître ; une personne aimerait encore visiter tout une liste de pays, et que cette liste est ordonnée (probablement par ordre de préférence).

UML permet d'associer une contrainte à un, ou plusieurs, élément(s) de modèle de différentes façons

- en plaçant directement la contrainte à côté d'une propriété ou d'une opération dans un classeur ;
- en ajoutant une note associée à l'élément à contraindre ;
- en plaçant la contrainte à proximité de l'élément à contraindre, comme une extrémité d'association par exemple ;
- en plaçant la contrainte sur une flèche en pointillés joignant les deux éléments de modèle à contraindre ensemble,

Mise en situation

Plaçons-nous dans le contexte d'une application bancaire. Il nous faut donc gérer :

des comptes bancaires,

des clients,

et des banques.

De plus, on aimerait intégrer les contraintes suivantes dans notre modèle :

un compte doit avoir un solde toujours positif ;

un client peut posséder plusieurs comptes ;

une personne peut être cliente de plusieurs banques ;

un client d'une banque possède au moins un compte dans cette banque ;

un compte appartient forcément à un client ;

une banque gère plusieurs comptes ;

une banque possède plusieurs clients.

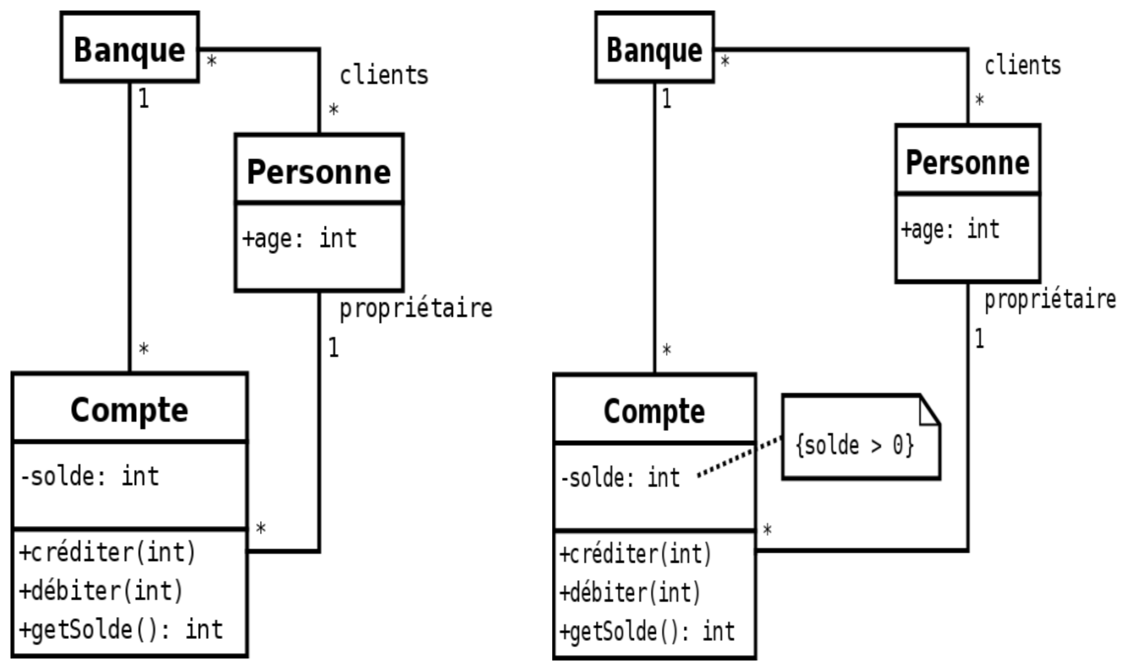


Illustration par l'exemple Suite…

Diagramme d'objets cohérent avec le diagramme de classes

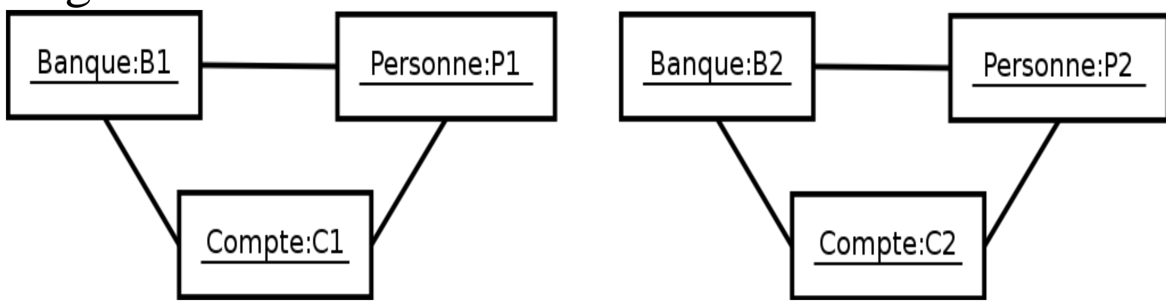
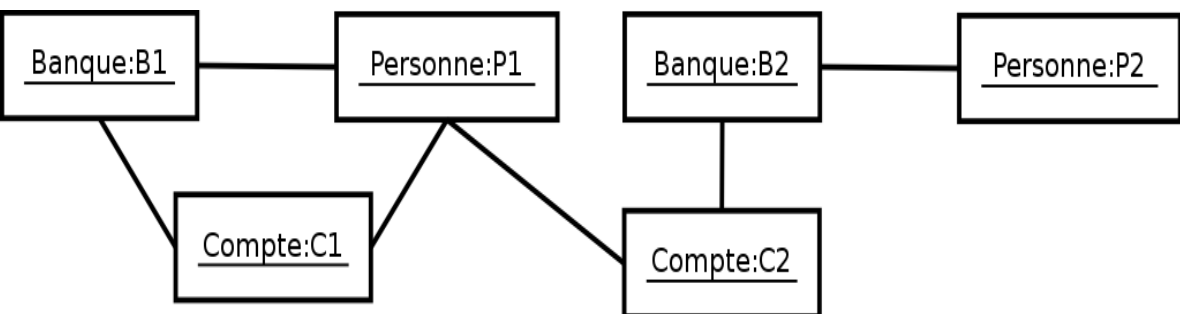


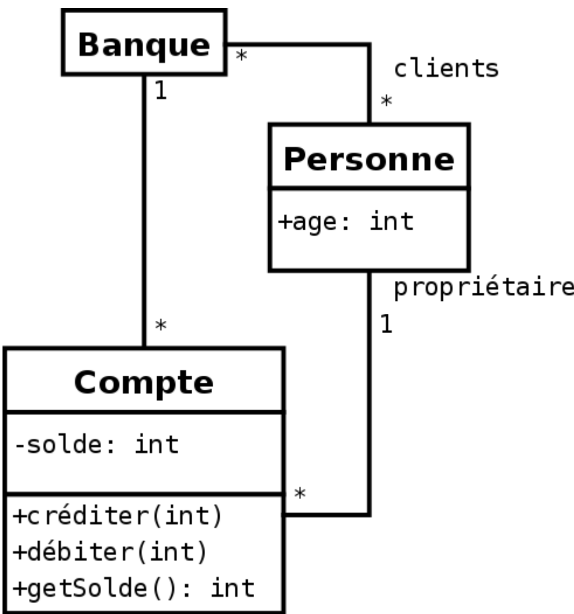
Diagramme d'objets cohérent avec le diagramme de classes, mais représentant une situation inacceptable



En effet, ce diagramme d'objets montre une personne (P1) ayant un compte dans une banque sans en être client. Ce diagramme montre également un client (P2) d'une banque n'y possédant pas de compte.

Illustration par l'exemple Suite…

- Exemple d'utilisation du langage de contrainte OCL sur l'exemple bancaire



context Compte
inv : solde > 0

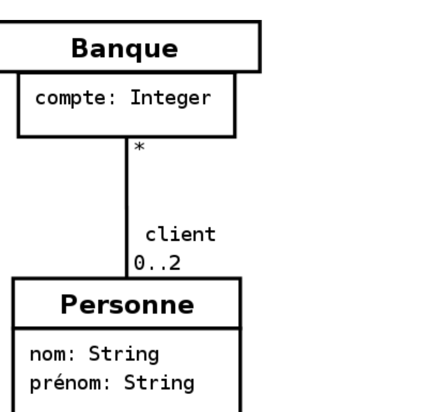
context Compte :: débiter(somme : int)
pre : somme > 0
post : solde = solde@pre - somme

context Compte
inv : banque.clients -> includes (propriétaire)

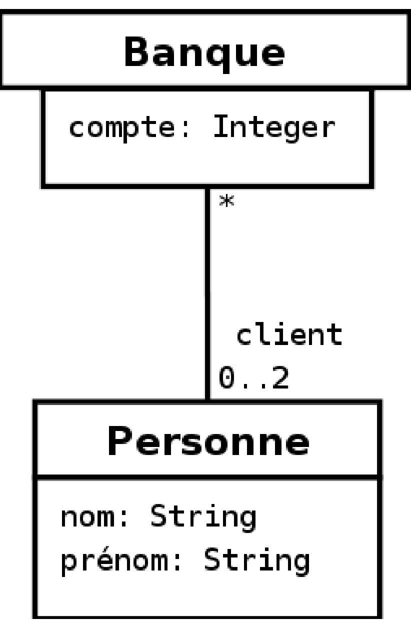
Navigation via une association

- Pour faire référence à un objet, ou un groupe d'objets, en association avec l'objet désigné par le contexte
- Le type du résultat dépend de la propriété structurelle empruntée pour accéder à l'objet référencé, et plus précisément de la multiplicité du côté de l'objet référencé, et du type de l'objet référencé proprement dit. Si on appelle *X* la classe de l'objet référencé, dans le cas d'une multiplicité de :

- 1, le type du résultat est *X*
- * ou 0..*n*, ..., le type du résultat est *Set*(*X*)
- * ou 0..*n*, ..., et s'il y a en plus une contrainte *{ordered}*, le type du résultat est *OrderedSet*(*X*)



Navigation via une association qualifiée



Une association qualifiée utilise un ou plusieurs qualificatifs pour sélectionner des instances de la classe cible de l'association. Pour emprunter une telle association, il est possible de spécifier les valeurs, ou les instances, des qualificatifs en utilisant des crochets ([]).

Opérations sur les collections

- **«::» –**
  - permet de désigner un élément (comme une opération) dans un élément englobant (comme un classeur ou un paquetage) ;
- **«.» –**
  - permet d'accéder à une propriété (attributs, terminaisons d'associations, opérations) d'un objet ;
- **«->» –**
  - permet d'accéder à une propriété d'une collection.
- Quelques opérations de base sur les collections :
- **size():Integer –**
  - retourne le nombre d'éléments (la cardinalité) de *self*.
- **includes(objet:T):Boolean –**
  - vrai si *self* contient l'objet *objet*.
- **excludes(objet:T):Boolean –**
  - vrai si *self* ne contient pas l'objet *objet*.
- **count(objet:T):Integer –**
  - retourne le nombre d'occurrences de *objet* dans *self*.
- **includesAll(c:Collection(T)):Boolean –**
  - vrai si *self* contient tous les éléments de la collection *c*.
- **excludesAll(c:Collection(T)):Boolean –**
  - vrai si *self* ne contient aucun élément de la collection *c*.
- **isEmpty() –**
  - vrai si *self* est vide.
- **notEmpty() –**
  - vrai si *self* n'est pas vide.
- **sum():T**
  - retourne la somme des éléments de *self*. Les éléments de *self* doivent supporter l'opérateur *somme* (+) et le type du résultat dépend du type des éléments.
- **product(c2:Collection(T2)):Set(Tuple(first:T,second:T2)) –**
  - le résultat est la collection de Tuple correspondant au produit cartésien de *self* (de type Collection(T)) par *c2*.

Opérations de base sur les ensembles (*Set*)

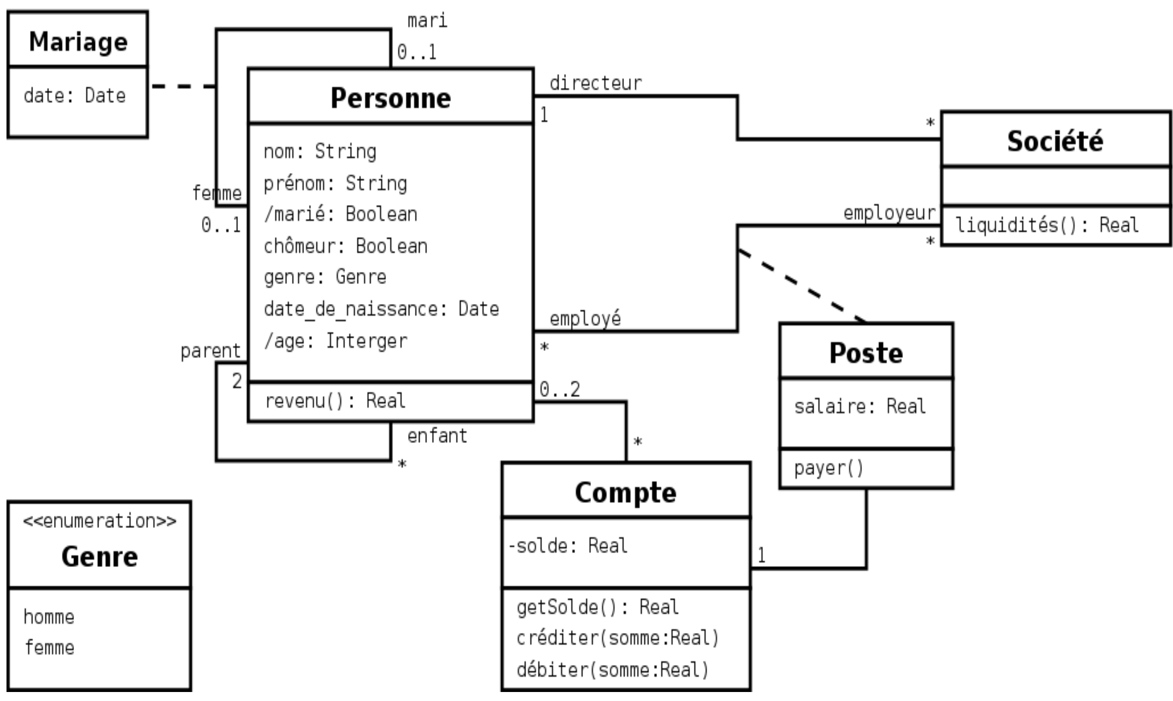
- **Quelques opérations de base sur les ensembles (type *Set*)**
- **union(set:Set(T)):Set(T) –**
  - retourne l'union de *self* et *set*.
- **union(bag:Bag(T)):Bag(T) –**
  - retourne l'union de *self* et *bag*.
- **=(set:Set(T)):Boolean –**
  - vrai si *self* et *set* contiennent les mêmes éléments.
- **intersection(set:Set(T)):Set(T) –**
  - intersection entre *self* et *set*.
- **intersection(bag:Bag(T)):Set(T) –**
  - intersection entre *self* et *bag*.
- **including(objet:T):Set(T) –**
  - Le résultat contient tous les éléments de *self* plus l'objet *objet*.
- **excluding(objet:T):Set(T) –**
  - Le résultat contient tous les éléments de *self* sans l'objet *objet*.
- **-(set:Set(T)):Set(T) –**
  - Le résultat contient tous les éléments de *self* sans ceux de *set*.
- **asOrderedSet():OrderedSet(T) –**
  - permet de convertir *self* du type *Set(T)* en *OrderedSet(T)*.
- **asSequence():Sequence(T) –**
  - permet de convertir *self* du type *Set(T)* en *Sequence(T)*.
- **asBag():Bag(T) –**
  - permet de convertir *self* du type *Set(T)* en *Bag(T)*.
- **Exemples**

Une société a au moins un employé : **context** Société **inv :** self.employé->notEmpty()

Une société possède exactement un directeur : **context** Société **inv :** self.directeur->size()=1

Le directeur est également un employé : **context** Société **inv :** self.employé->includes(self.directeur)

Exemples de contraintes



Exemples de contraintes

- Dans une société, le directeur est un employé, n'est pas un chômeur et doit avoir plus de 40 ans. De plus, une société possède exactement un directeur et au moins un employé. context Société inv : self.directeur->size()=1 and not(self.directeur.chômeur) and self.directeur.age > 40 and self.employé->includes(self.directeur)
- Une personne considérée comme au chômage ne doit pas avoir des revenus supérieurs à 100 € context Personne inv : let revenus : Real = self.poste.salaire->sum() in if chômeur then revenus < 100 else revenus >= 100 endif
- Une personne possède au plus 2 parents (référencés). context Personne inv : parent->size()<=2
- Si une personne possède deux parents, l'un est une femme et l'autre un homme. context Personne inv : parent->size()=2 implies (parent->exists(genre=Genre::homme) and parent->exists(genre=Genre::femme))
- Tous les enfants d'une personne ont bien cette personne comme parent et inversement. context Personne inv : enfant->notEmpty() implies enfant->forAll(p : Personne | p.parents->includes(self)) context Personne inv : parent->notEmpty() implies parent->forAll (p : Personne | p.enfant->includes (self))
- Pour être marié, il faut avoir une femme ou un mari. context Personne::marié derive : self.femme->notEmpty() or self.mari->notEmpty()
- Pour être marié, il faut avoir plus de 18 ans. Un homme est marié avec exactement une femme et une femme avec exactement un homme. context Personne inv : self.marié implies self.genre=Genre::homme implies (self.femme->size()=1 and self.femme.genre=Genre::femme) and self.genre=Genre::femme implies (self.mari->size()=1 and self.mari.genre=Genre::homme) and self.age >=18

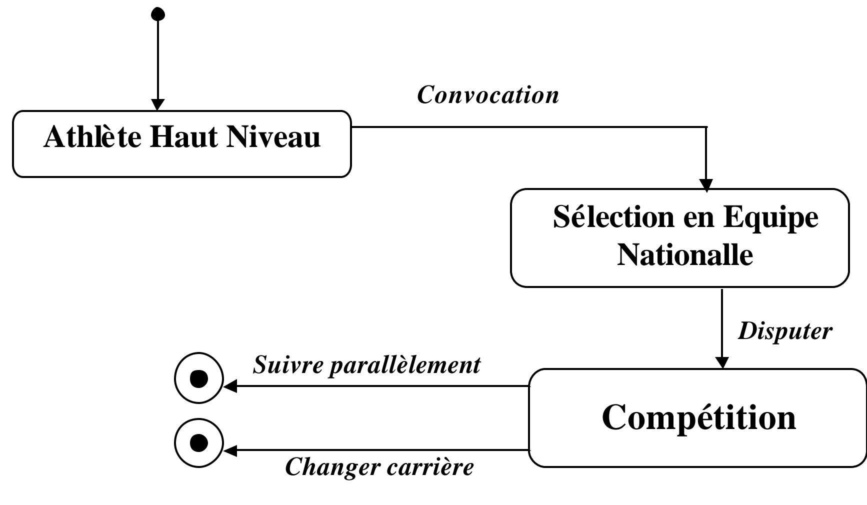
Diagramme d ’états

- C’est un **graphe** composé de **nœuds** représentant des états d’un objet d’une classe et les **arcs** sont les transitions portant des événements.
- Un diagramme d’états est propre à une classe d’objets.
- Un état d’un objet peut correspondre à des sous états . Cela dépend du niveau de granularité qu’on désire.

Exemple :

l’état ***Sportif*** peut être représenté par trois sous états : ***Athlète Haut Niveau ; Sélection en E.N.*** et ***Compétition***.

- Les sous états sont représentés comme des états.
- Dans un diagramme d’états on peut développer un état d’un objet par un **sous diagramme d'états** avec des points dentrée et des points de sortie. De telle sorte on peut passer d'un état à un sous état et inversement.



Sous diagramme d’états.

✓Les attributs

Ceux sont des paramètres portés par des événements. Ils sont représentés dans une liste (utilisation des ( )) . Une transition peut porter une liste d'attributs.

✓Les gardiens

Ceux sont des fonctions booléennes qui conditionnent le déclenchement d'une transition. (utilisation des [ ]).

✓Les activités

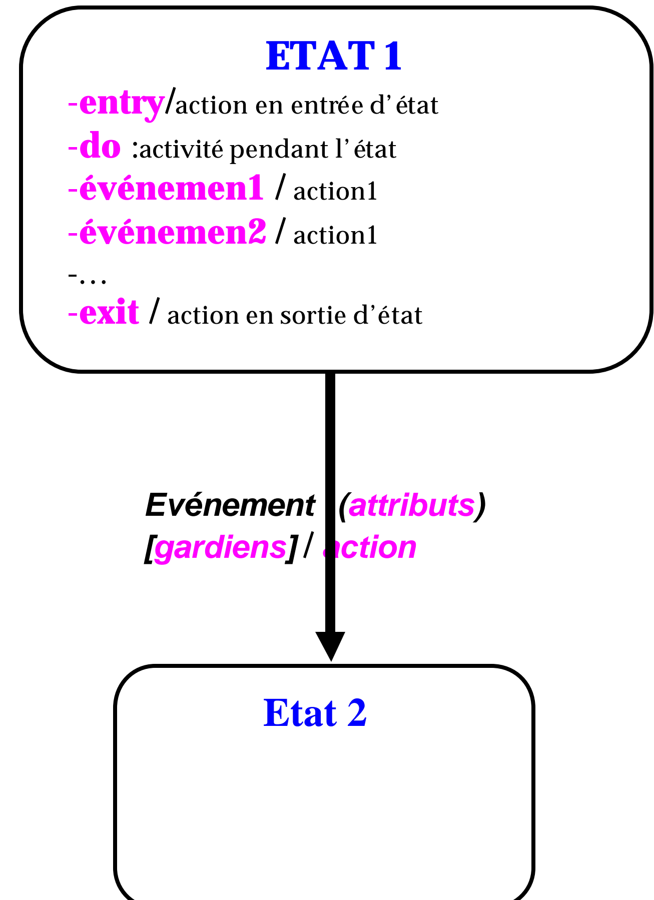
Ceux sont des opérations continues dans le temps et s'exécutent tardivement. Une activité est forcément associée à un état. Il est précédée du mot clé "***do***".

✓Les actions

Ceux des opérations qui s'exécutent instantanément. Une action peut être associé à un état ou à une transition. Elle peut intervenir :

- soit en entrée d'état (elle sera préfixée par ***entry/…)*** ;
- soit en sortie d'état (elle sera préfixée par ***exit/…)*** ;
- soit en réponse à un événement déclencheur (elle sera préfixée par le nom d'événement ***événement1/…)*** ;
- soit enfin au cours d'une **transition** **(**elle est préfixée **par */…*).**

- ✓Les attributs
- ✓Les gardiens
- ✓Les activités
- ✓Les actions

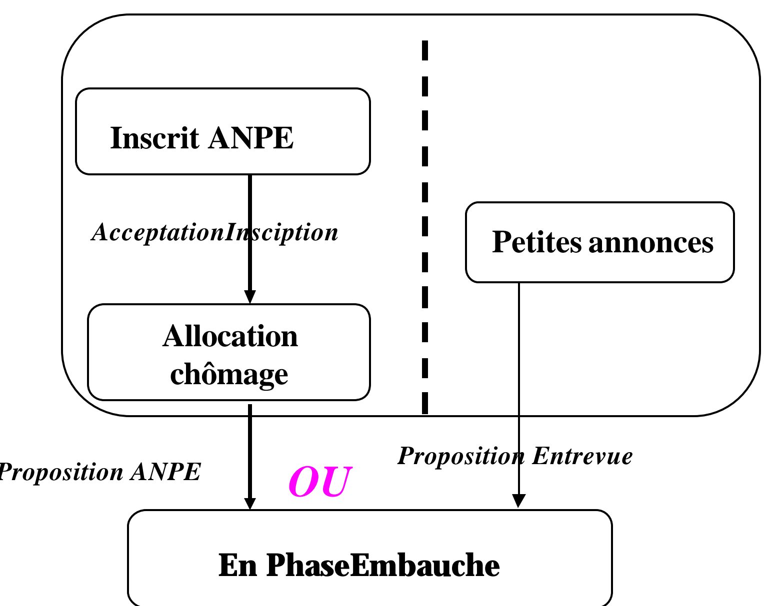


Remarque

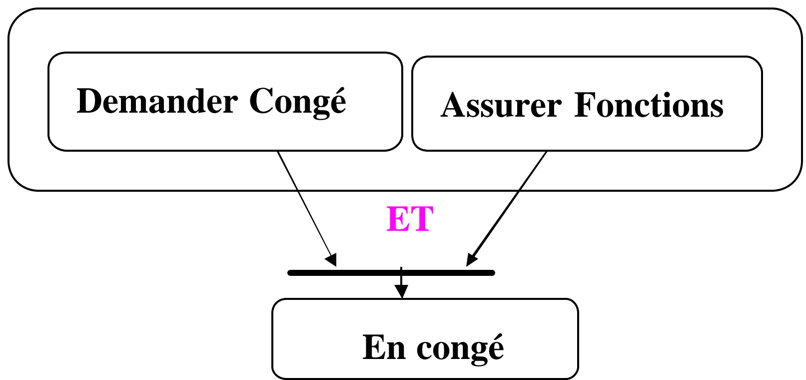
Plusieurs sous diagrammes d'états peuvent intervenir en même temps. Ils se déroulent en **concurrence** ou en **synchronisation**.

- Lorsqu'ils sont en **concurrence**, le premier sous diagramme d'état **bloque les autres** et fait quitter l'état englobant vers un autre état.

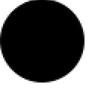
- Lorsqu'ils sont en **synchronisation**, la transition de l'état englobant vers un autre état **ne s'effectue que lorsque tous les sous diagrammes le permettent**. Aucun sous diagramme ne peut être interrompu.

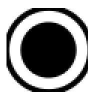


- Sous diagramme d'état en concurrence.



- Sous diagramme d'état en synchronisation.

État initial 

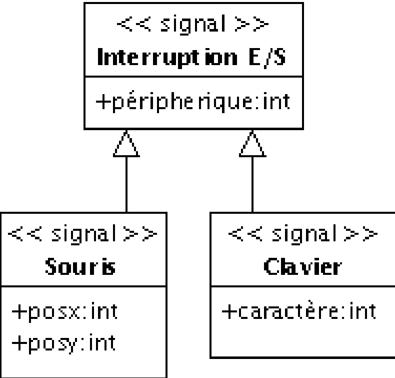
État final 

L'état initial est un pseudo état qui indique l'état de départ, par défaut, lorsque le diagramme d'états-transitions, ou l'état enveloppant, est invoqué. Lorsqu'un objet est créé, il entre dans l'état initial

L'état final est un pseudo état qui indique que le diagramme d'états-transitions, ou l'état enveloppant, est terminé.

Événement

- Un événement est quelque chose qui se produit pendant l'exécution d'un système et qui mérite d'être modélisé. Les diagrammes d'états-transitions permettent justement de spécifier les réactions d'une partie du système à des événements discrets.



Événement d'appel (*call*)

Un événement d'appel représente la réception de l'appel d'une opération par un objet. Les paramètres de l'opération sont ceux de l'événement d'appel. La syntaxe d'un événement d'appel est la même que celle d'un signal. Par contre, les événements d'appel sont des méthodes déclarées au niveau du diagramme de classes.

Événement de changement (*change*)

Un événement de changement est généré par la satisfaction (passage de faux à vrai) d'une expression booléenne sur des valeurs d'attributs. Il s'agit d'une manière déclarative d'attendre qu'une condition soit satisfaite. La syntaxe d'un événement de changement est la suivante : when (<condition_booléenne>)

Transition

- Une transition définit la réponse d'un objet à l'occurrence d'un événement

Condition de garde

- Une transition peut avoir une condition de garde (spécifiée par '[' <garde> ']' dans la syntaxe).

Effet d'une transition

- Lorsqu'une transition se déclenche (on parle également de tir d'une transition), son effet (spécifié par '/' <activité> dans la syntaxe) s'exécute.

Transition externe

- Une transition externe est une transition qui modifie l'état actif. Il s'agit du type de transition le plus répandu. Elle est représentée par une flèche allant de l'état source vers l'état cible.

Transition d'achèvement

- Une transition dépourvue d'événement déclencheur explicite se déclenche à la fin de l'activité contenue dans l'état source (y compris les état imbriqués).

Transition interne

- Les règles de déclenchement d'une transition interne sont les mêmes que pour une transition externe excepté qu'une transition interne ne possède pas d'état cible et que l'état actif reste le même à la suite de son déclenchement.

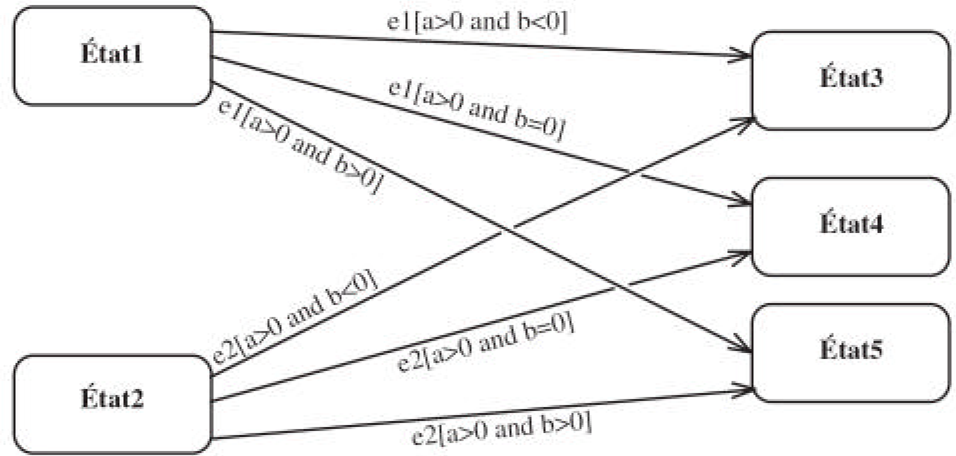
saisie mot de passe
entry/set echo invisible exit/set echo normal character/traiter caractère help/afficher aide clear/remise à zéro mot de passe et chronomètre after(20s)/exit

- Les transitions internes possèdent des noms d'événement prédéfinis correspondant à des déclencheurs particuliers : *entry*, *exit*, *do* et *include*
- ***entry –***
  - *entry permet de spécifier une activité qui s'accomplit quand on entre dans l'état.*
- ***exit –***
  - *exit permet de spécifier une activité qui s'accomplit quand on sort de l'état.*
- ***do –***
  - *Une activité do commence dès que l'activité entry est terminée. Lorsque cette activité est terminée, une transition d'achèvement peut être déclenchée, après l'exécution de l'activité exit bien entendu. Si une transition se déclenche pendant que l'activité do est en cours, cette dernière est interrompue et l'activité exit de l'état s'exécute.*
- ***include –***
  - *permet d'invoquer un sous-diagramme d'états-transitions.*

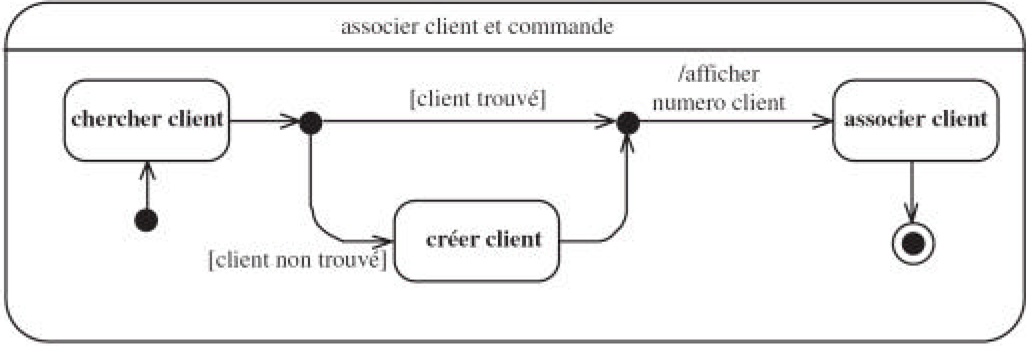
Point de choix

- Il est possible de représenter des alternatives pour le franchissement d'une transition. On utilise pour cela des pseudo-états particuliers : les points de jonction (représentés par un petit cercle plein) et les points de décision (représenté par un losange).

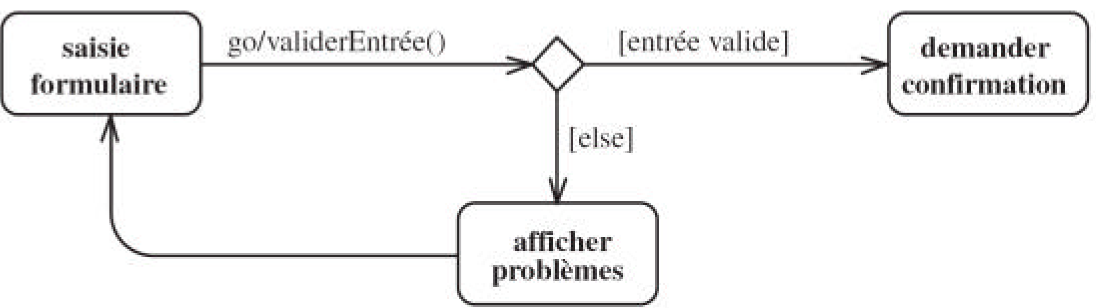
- **Point de jonction**



- En haut, un diagramme sans point de jonction. En bas, son équivalent utilisant un point de jonction



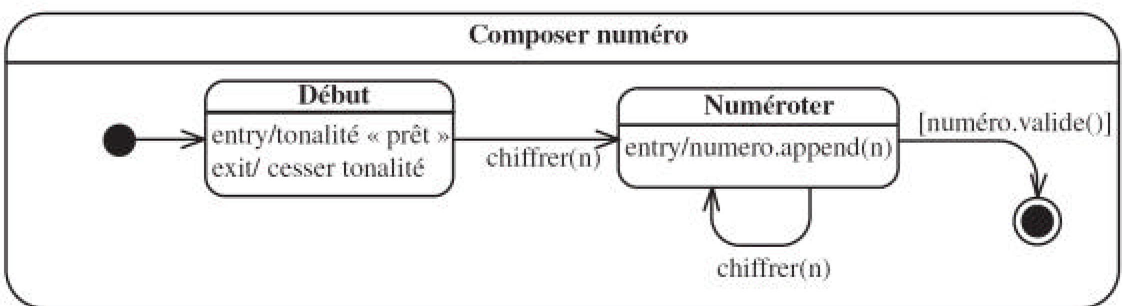
- **Point de décision**



- Un point de décision possède une entrée et au moins deux sorties. Contrairement à un point de jonction, les gardes situées après le point de décision sont évaluées au moment où il est atteint. Cela permet de baser le choix sur des résultats obtenus en franchissant le segment avant le point de choix. Si, quand le point de décision est atteint, aucun segment en aval n'est franchissable, c'est que le modèle est mal formé.

États composites

- Un état simple ne possède pas de sous structure mais uniquement, le cas échéant, un jeu de transitions internes. Un état composite est un état décomposé en régions contenant chacune un ou plusieurs sous états.

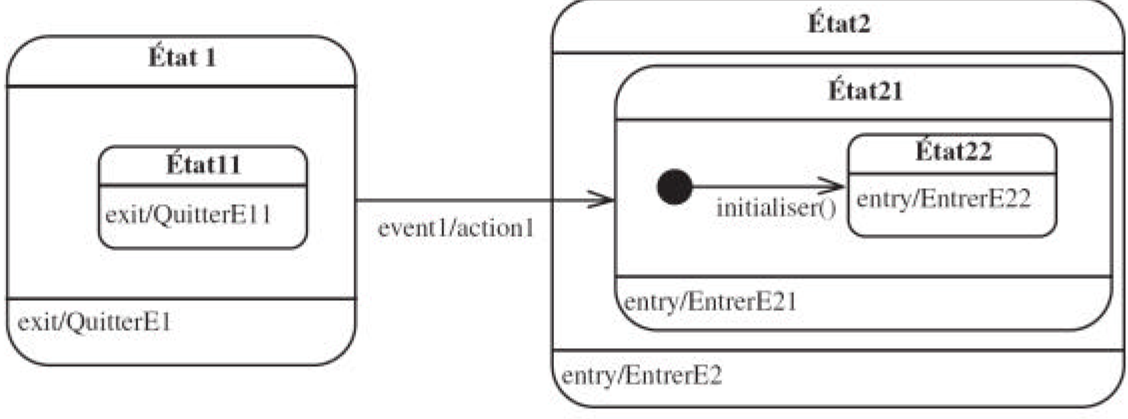


Composer numéro

- L'utilisation d'états composites permet de développer une spécification par raffinements. Il n'est pas nécessaire de représenter les sous états à chaque utilisation de l'état englobant. Une notation abrégée. Permet d'indiquer qu'un état est composite et que sa définition est donnée sur un autre diagramme.

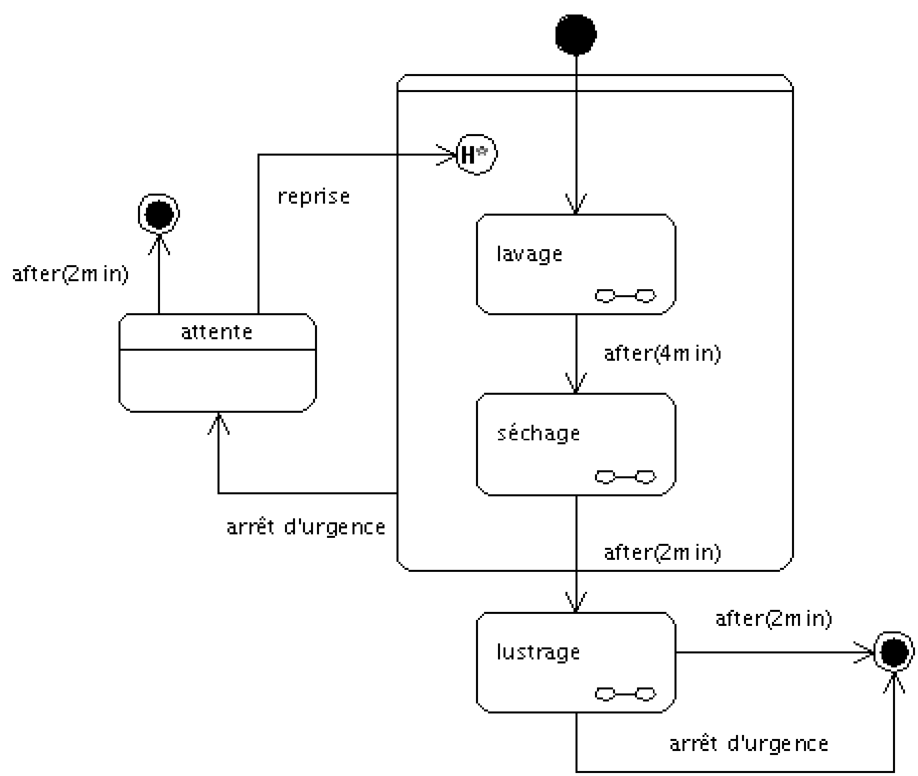
Transition

- Les transitions peuvent avoir pour cible la frontière d'un état composite et sont équivalentes à une transition ayant pour cible l'état initial de l'état composite.
  Une transition ayant pour source la frontière d'un état composite est équivalente à une transition qui s'applique à tout sous état de l'état composite source. Cette relation est transitive : la transition est franchissable depuis tout état imbriqué, quelle que soit sa profondeur.



État historique

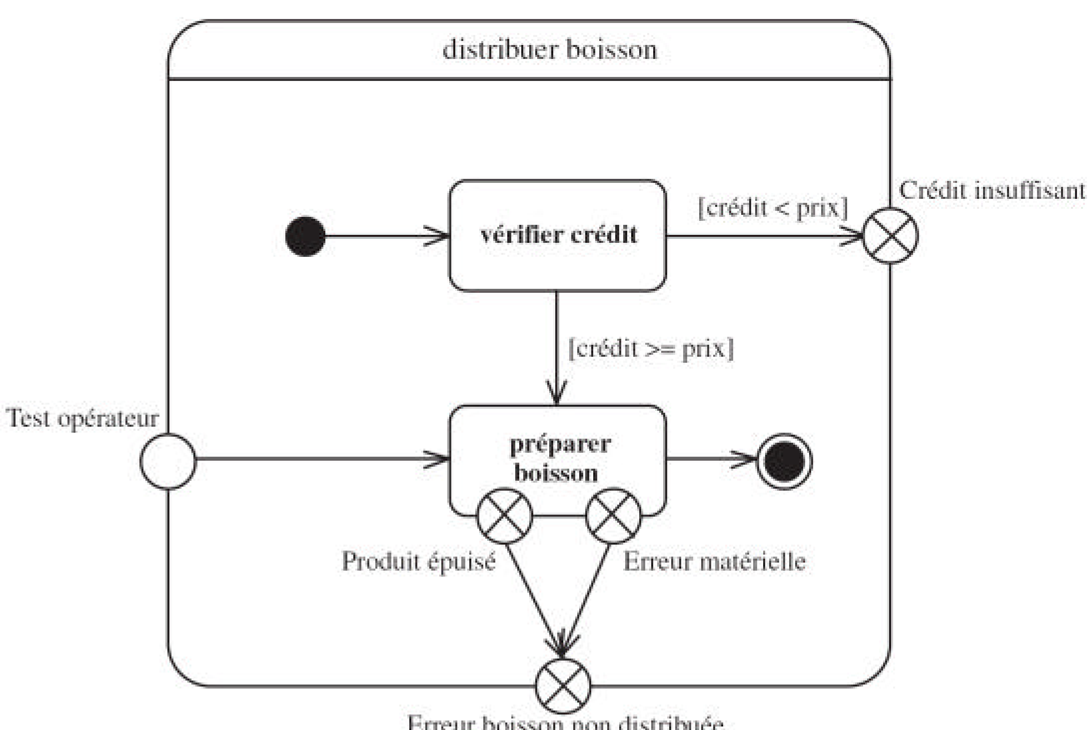
- Un état historique, également qualifié d'*état historique plat*, est un pseudo état qui mémorise le dernier sous état actif d'un état composite. Graphiquement, il est représenté par un cercle contenant un *H*.



Exemple de diagramme possédant un état historique profond permettant de reprendre le programme de lavage ou de séchage d'une voiture à l'endroit où il était arrivé avant d'être interrompu.

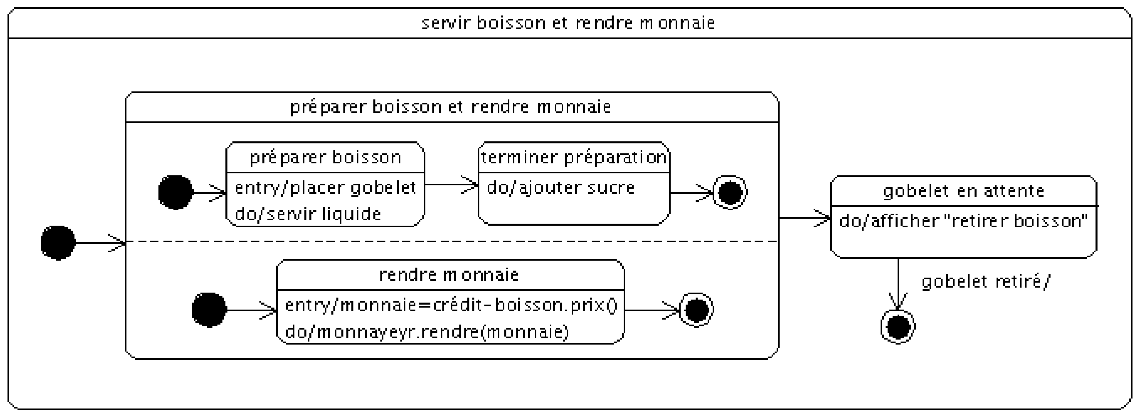
Interface : les points de connexion

- il est possible de masquer les sous états d'un état composite et de les définir dans un autre diagramme. Cette pratique nécessite parfois l'utilisation de pseudo états appelés *points de connexion*.

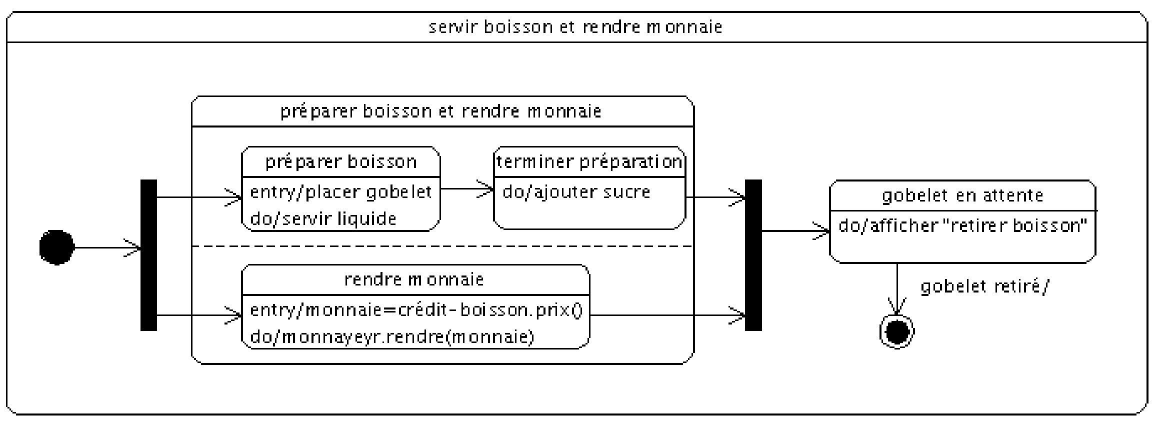


Concurrence

- Les diagrammes d'états transitions permettent de décrire efficacement les mécanismes concurrents grâce à l'utilisation d'*états orthogonaux*. Un état orthogonal est un état composite comportant plus d'une région, chaque région représentant un flot d'exécution. Graphiquement, dans un état orthogonal, les différentes régions sont séparées par un trait horizontal en pointillé allant du bord gauche au bord droit de l'état composite.



Exemple d'utilisation de transitions complexes.



- Les diagrammes d'activités permettent de mettre l'accent sur les traitements. Ils sont donc particulièrement adaptés à la modélisation du cheminement de flots de contrôle et de flots de données. Ils permettent ainsi de représenter graphiquement le comportement d'une méthode ou le déroulement d'un cas d'utilisation.

Action (*action*)

- Une action est le plus petit traitement qui puisse être exprimé en UML. Une action a une incidence sur l'état du système ou en extrait une information. Les actions sont des étapes discrètes à partir desquelles se construisent les comportements. La notion d'action est à rapprocher de la notion d'instruction élémentaire d'un langage de programmation (comme C++ ou Java).
- une affectation de valeur à des attributs ;
- un accès à la valeur d'une propriété structurelle (attribut ou terminaison d'association) ;
- la création d'un nouvel objet ou lien ;
- un calcul arithmétique simple ;
- l'émission d'un signal ;
- la réception d'un signal ;

- **Action appeler (*call operation*) –**
  - L'action *call operation* correspond à l'invocation d'une opération sur un objet de manière synchrone ou asynchrone. Lorsque l'action est exécutée, les paramètres sont transmis à l'objet cible. Si l'appel est asynchrone, l'action est terminée et les éventuelles valeurs de retour seront ignorées. Si l'appel est synchrone, l'appelant est bloqué pendant l'exécution de l'opération et, le cas échéant, les valeurs de retour pourront être réceptionnées.
- **Action comportement (*call behavior*) –**
  - L'action *call behavior* est une variante de l'action *call operation* car elle invoque directement une activité plutôt qu'une opération.
- **Action envoyer (*send*) –**
  - Cette action crée un message et le transmet à un objet cible, où elle peut déclencher un comportement. Il s'agit d'un appel asynchrone (*i.e.* qui ne bloque pas l'objet appelant) bien adapté à l'envoi de signaux (*send signal*).
- **Action accepter événement (*accept event*) –**
  - L'exécution de cette action bloque l'exécution en cours jusqu'à la réception du type d'événement spécifié, qui généralement est un signal. Cette action est utilisée pour la réception de signaux asynchrones.
- **Action accepter appel (*accept call*) –**
  - Il s'agit d'une variante de l'action *accept event* pour les appels synchrones.
- **Action répondre (*reply*) –**
  - Cette action permet de transmettre un message en réponse à la réception d'une action de type *accept call*.
- **Action créer (*create*) –**
  - Cette action permet d'instancier un objet.
- **Action détruire (*destroy*) –**
  - Cette action permet de détruire un objet.
- **Action lever exception (*raise exception*) –**
  - Cette action permet de lever explicitement une exception.

Activité (*activity*)

- Une activité définit un comportement décrit par un séquencement organisé d'unités dont les éléments simples sont les actions. Le flot d'exécution est modélisé par des noeuds reliés par des arcs (transitions). Le flot de contrôle reste dans l'activité jusqu'à ce que les traitements soient terminés.

Groupe d'activités (*activity group*)

- Un groupe d'activités est une activité regroupant des noeuds et des arcs. Les noeuds et les arcs peuvent appartenir à plus d'un groupe. Un diagramme d'activités est lui-même un groupe d'activités

Noeud d'activité (*activity node*)

- Représentation graphique des noeuds d'activité.

- Le noeud représentant une action, qui est une variété de noeud exécutable,
- Un noeud objet,
- Un noeud de décision ou de fusion,
- Un noeud de bifurcation ou d'union,
- Un noeud initial,
- Un noeud final et un noeud final de flot.

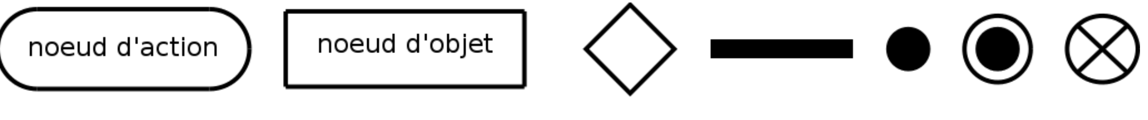
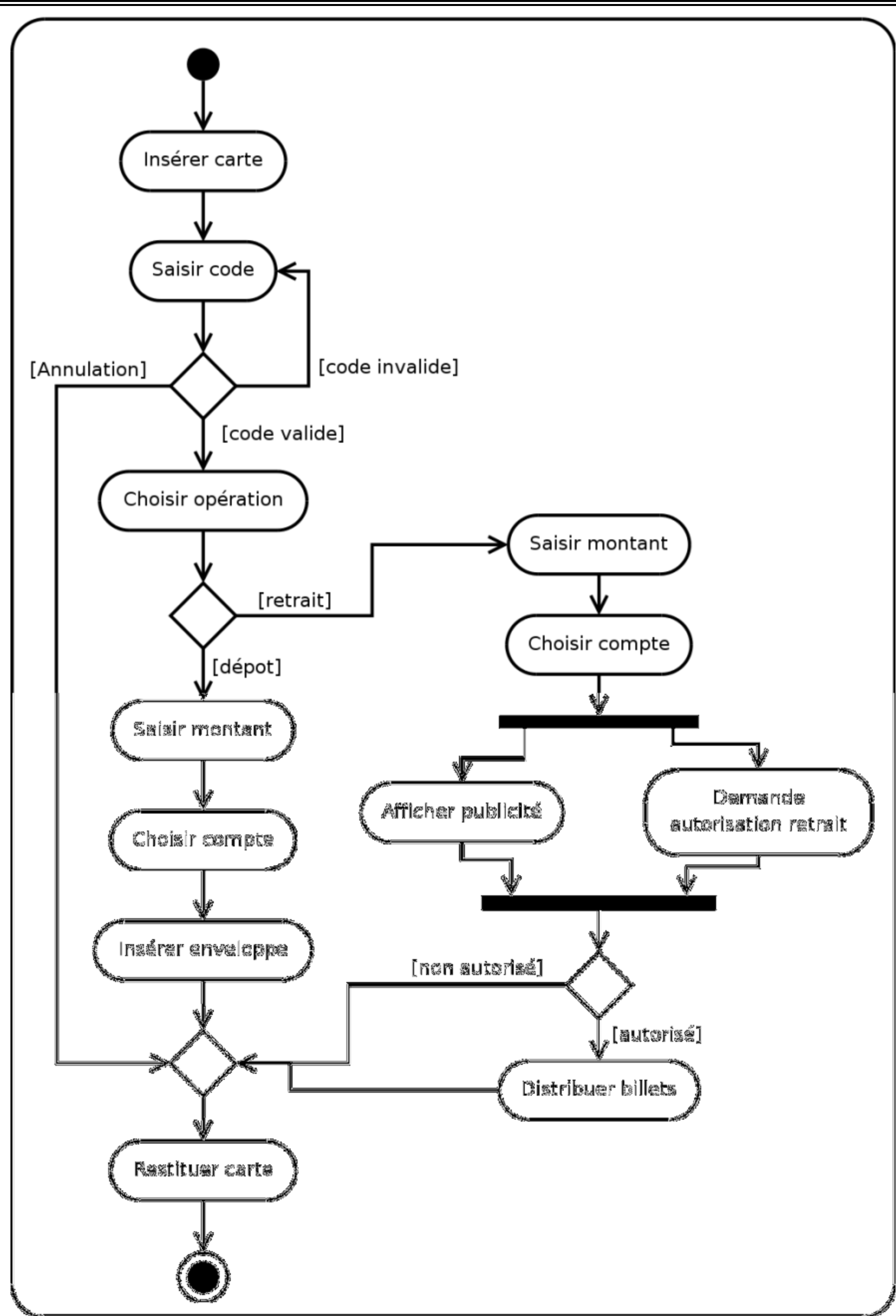


Diagramme d'activités modélisant le fonctionnement d'une borne bancaire.



Transition

- Le passage d'une activité vers une autre est matérialisé par une transition. Graphiquement les transitions sont représentées par des flèches en traits pleins qui connectent les activités entre elles

Noeud exécutable (*executable node*)

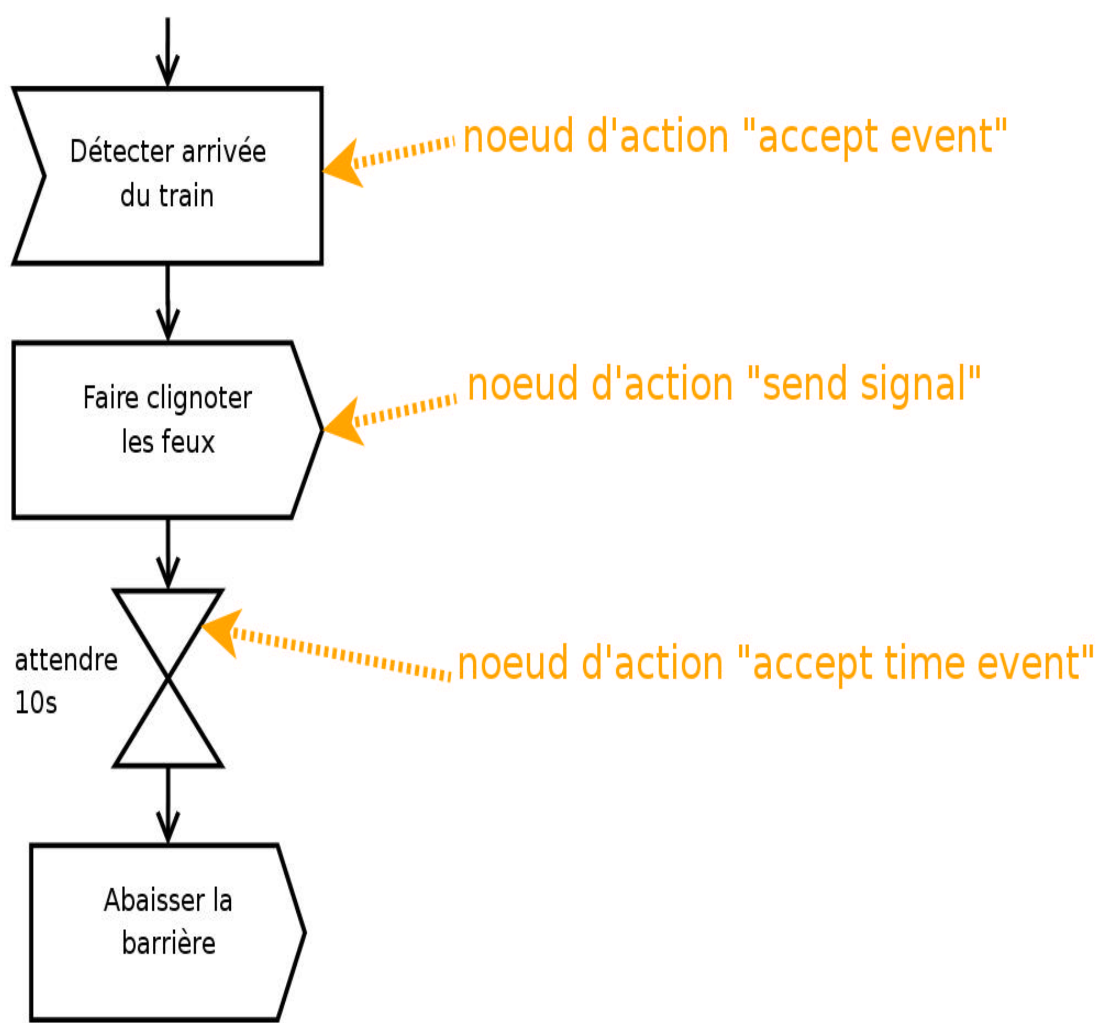
- Un noeud exécutable est un noeud d'activité qu'on peut exécuter (*i.e.* une activité). Il possède un gestionnaire d'exception qui peut capturer les exceptions levées par le noeud, ou un de ses noeuds imbriqués.

Noeud d'action

- Un noeud d'action est un noeud d'activité exécutable qui constitue l'unité fondamentale de fonctionnalité exécutable dans une activité. L'exécution d'une action représente une transformation ou un calcul quelconque dans le système modélisé

Saisir code

Représentation particulière des noeuds d'action de communication



Noeud de contrôle (*control node*)

Noeud initial

- Un noeud initial est un noeud de contrôle à partir duquel le flot débute lorsque l'activité enveloppante est invoquée. Une activité peut avoir plusieurs noeuds initiaux. Un noeud initial possède un arc sortant et pas d'arc entrant.

Noeud final

- Un noeud final est un noeud de contrôle possédant un ou plusieurs arcs entrants et aucun arc sortant.

Noeud de fin d'activité

- Lorsque l'un des arcs d'un noeud de fin d'activité est activé (*i.e.* lorsqu'un flot d'exécution atteint un noeud de fin d'activité), l'exécution de l'activité enveloppante s'achève et tout noeud ou flot actif au sein de l'activité enveloppante est abandonné. Si l'activité a été invoquée par un appel synchrone, un message (*reply*) contenant les valeurs sortantes est transmis en retour à l'appelant.

Noeud de fin de flot

- Lorsqu'un flot d'exécution atteint un noeud de fin de flot, le flot en question est terminé, mais cette fin de flot n'a aucune incidence sur les autres flots actifs de l'activité enveloppante.

  Les noeuds de fin de flot sont particuliers et à utiliser avec parcimonie.

Noeud de décision et de fusion

Noeud de décision (*decision node*)

Un noeud de décision est un noeud de contrôle qui permet de faire un choix entre plusieurs flots sortants. Il possède un arc entrant et plusieurs arcs sortants. Ces derniers sont généralement accompagnés de conditions de garde pour conditionner le choix. Si, quand le noeud de décision est atteint, aucun arc en aval n'est franchissable (*i.e.* aucune condition de garde n'est vraie), c'est que le modèle est mal formé. L'utilisation d'une garde [else] est recommandée après un noeud de décision car elle garantit un modèle bien formé. En effet, la condition de garde [else] est validée si et seulement si toutes les autres gardes des transitions ayant la même source sont fausses. Dans le cas où plusieurs arcs sont franchissables (*i.e.* plusieurs conditions de garde sont vraies), seul l'un d'entre eux est retenu et ce choix est non déterministe.

Noeud de fusion (*merge node*)

Un noeud de fusion est un noeud de contrôle qui rassemble plusieurs flots alternatifs entrants en un seul flot sortant. Il n'est pas utilisé pour synchroniser des flots concurrents (c'est le rôle du noeud d'union) mais pour accepter un flot parmi plusieurs.

Remarque

Graphiquement, il est possible de fusionner un noeud de fusion et un noeud de décision, et donc d'avoir un losange possédant plusieurs arcs entrants et sortants. Il est également possible de fusionner un noeud de décision ou de fusion avec un autre noeud, ou avec une activité. Cependant, pour mieux mettre en évidence un branchement conditionnel, il est préférable d'utiliser un noeud de décision (losange).

Noeud de bifurcation et d'union

Noeud de bifurcation ou de débranchement (*fork node*)

Un noeud de bifurcation, également appelé noeud de débranchement est un noeud de contrôle qui sépare un flot en plusieurs flots concurrents. Un tel noeud possède donc un arc entrant et plusieurs arcs sortants. On apparie généralement un noeud de bifurcation avec un noeud d'union pour équilibrer la concurrence

Noeud d'union ou de jointure (*join node*)

Un noeud d'union, également appelé noeud de jointure est un noeud de contrôle qui synchronise des flots multiples. Un tel noeud possède donc plusieurs arcs entrants et un seul arc sortant. Lorsque tous les arcs entrants sont activés, l'arc sortant l'est également.

Remarque

Graphiquement, il est possible de fusionner un noeud de bifurcation et un noeud d'union, et donc d'avoir un trait plein possédant plusieurs arcs entrants et sortants.

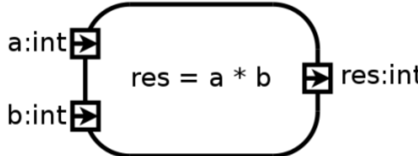
Noeud de contrôle (*control node*)

Diagramme d'activité illustrant l'utilisation de noeuds de contrôle. Ce diagramme décrit la prise en compte d'une commande.

Noeud d'objet (*object node*)

Pin d'entrée ou de sortie

- Pour spécifier les valeurs passées en argument à une activité et les valeurs de retour, on utilise des noeuds d'objets appelés pins (*pin* en anglais) d'entrée ou de sortie. L'activité ne peut débuter que si l'on affecte une valeur à chacun de ses pins d'entrée. Quand l'activité se termine, une valeur doit être affectée à chacun de ses pins de sortie.



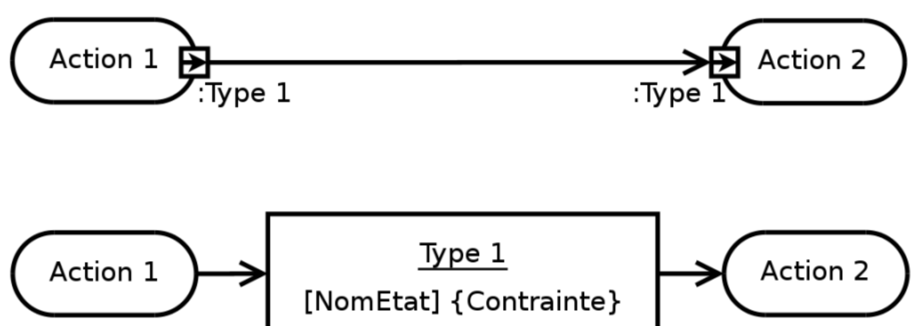
Représentation des pins d'entrée et de sortie sur une activité.

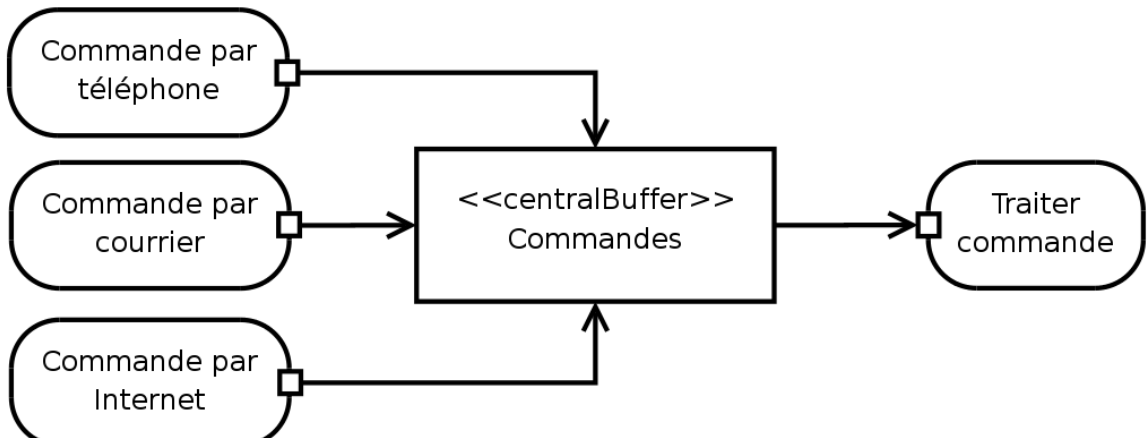
Pin de valeur (*value pin*)

- Un pin valeur est un pin d'entrée qui fournit une valeur à une action sans que cette valeur ne provienne d'un arc de flot d'objets. Un pin valeur est toujours associé à une valeur spécifique.

Flot d'objet

- Un flot d'objets permet de passer des données d'une activité à une autre. Un arc reliant un pin de sortie à un pin d'entrée est, par définition même des pins, un flot d'objets. Dans cette configuration, le type du pin récepteur doit être identique ou parent (au sens de la relation de généralisation) du type du pin émetteur



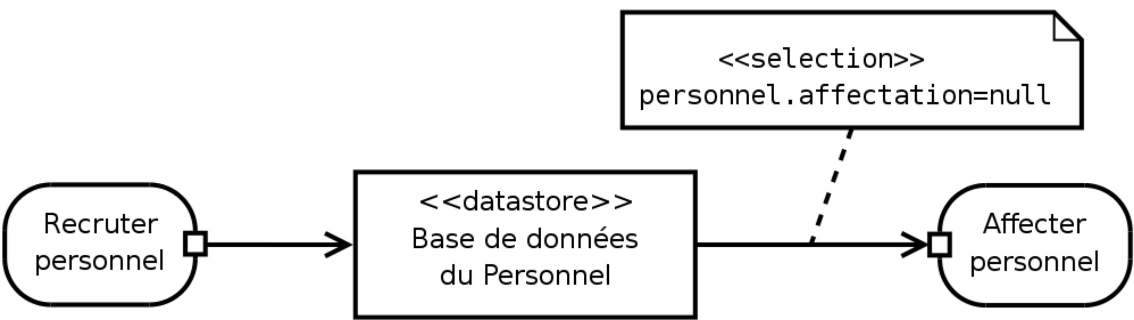


Noeud tampon central (*central buffer node*)

Un noeud tampon central est un noeud d'objet qui accepte les entrées de plusieurs noeuds d'objets ou produit des sorties vers plusieurs noeuds d'objets. Les flots en provenance d'un noeud tampon central ne sont donc pas directement connectés à des actions. Ce noeud modélise donc un tampon traditionnel qui peut contenir des valeurs en provenance de diverses sources et livrer des valeurs vers différentes destinations.

Noeud de stockage des données (data store node)

- Un noeud de stockage des données est un noeud tampon central particulier qui assure la persistance des données. Lorsqu'une information est sélectionnée par un flux sortant, l'information est dupliquée et ne disparaît pas du noeud de stockage des données comme ce serait le cas dans un noeud tampon central. Lorsqu'un flux entrant véhicule une donnée déjà stockée par le noeud de stockage des données, cette dernière est écrasée par la nouvelle.

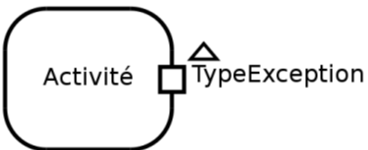


Dans cette modélisation, le personnel, après avoir été recruté par l'activité Recruter personnel, est stocké de manière persistante dans le noeud de stockage Base de donnée du Personnel. Bien qu'ils restent dans ce noeud, chaque employé qui n'a pas encore reçu d'affectation (étiquette stéréotypée «selection» : personnel.affectation=null) est disponible pour être utilisé par l'activité Affecter personnel.

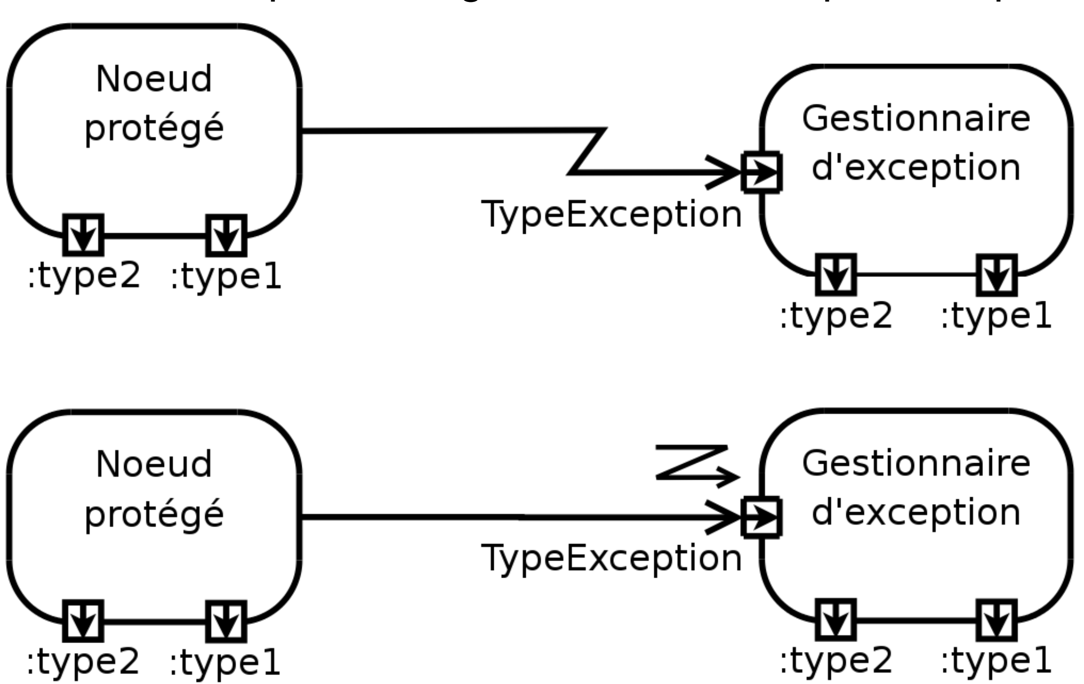
Partitions

Les partitions, souvent appelées couloirs ou lignes d'eau (*swimlane*) du fait de leur notation, permettent d'organiser les noeuds d'activités dans un diagramme d'activités en opérant des regroupements

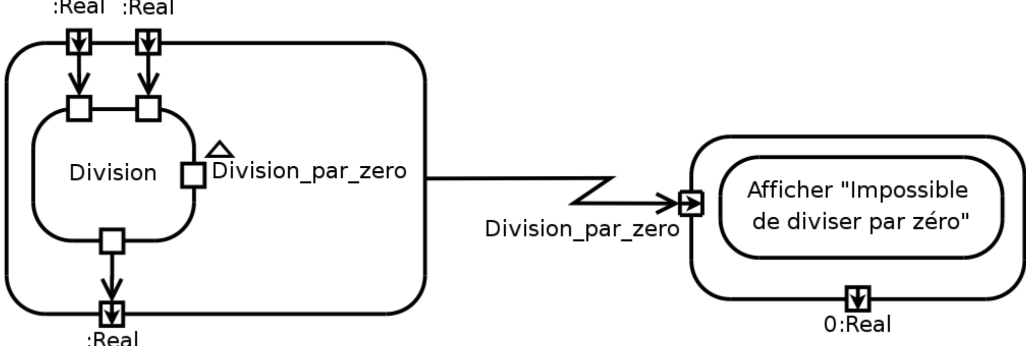
Exceptions



- Une exception est générée quand une situation anormale entrave le déroulement nominal d'une tâche. Elle peut être générée automatiquement pour



Un gestionnaire d'exception est une activité possédant un pin d'entrée du type de l'exception qu'il gère et lié à l'activité qu'il protège par un arc en zigzag ou un arc classique orné d'une petite flèche en zigzag. Le gestionnaire d'exception doit avoir les mêmes pins de sortie que le bloc qu'il protège

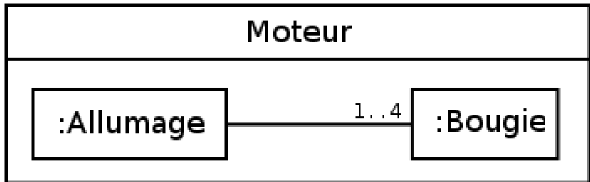


Lorsqu'une exception survient, l'exécution de l'activité en cours est abandonnée sans générer de valeur de sortie. Le mécanisme d'exécution recherche alors un gestionnaire d'exception susceptible de traiter l'exception levée ou une de ses classes parentes.

Diagrammes d'interaction

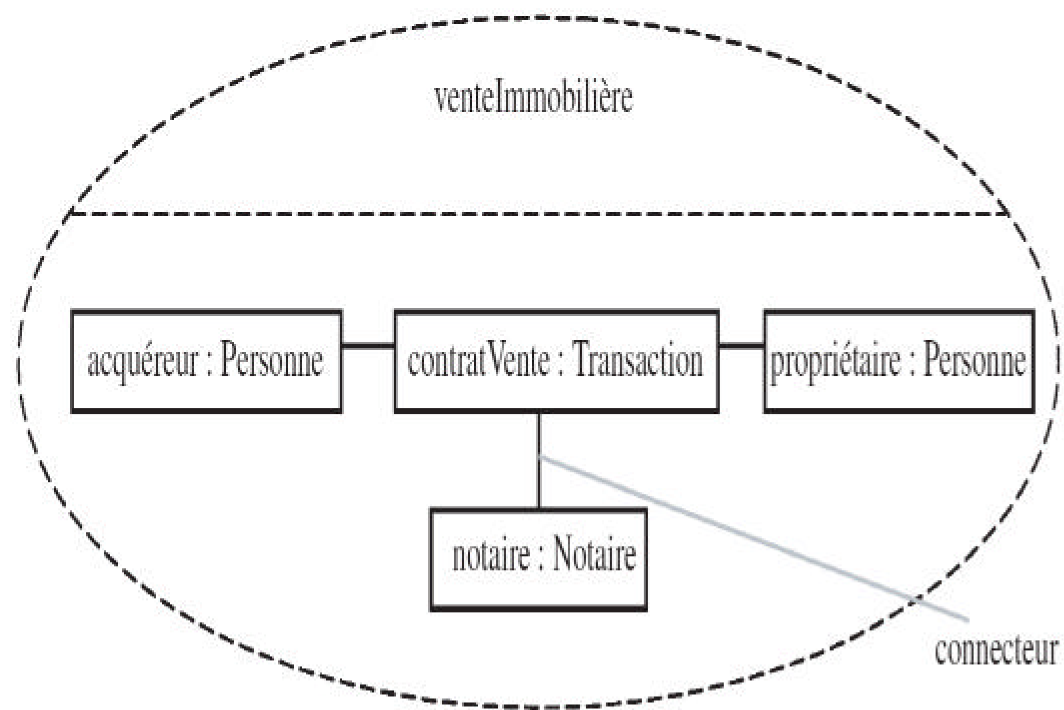
- Un objet interagit pour implémenter un comportement. On peut décrire cette interaction de deux manières complémentaires : l'une est centrée sur des objets individuels (diagramme d'états transitions) et l'autre sur une collection d'objets qui coopèrent (diagrammes d'interaction).

- Les diagrammes d'interaction permettent d'établir un lien entre les diagrammes de cas d'utilisation et les diagrammes de classes : ils montrent comment des objets (i.e. des instances de classes) communiquent pour réaliser une certaine fonctionnalité. Ils apportent un aspect dynamique à la modélisation du système.
- **Classeur structuré**



- Les classes découvertes au moment de l'analyse (celles qui figurent dans le diagramme de classes) ne sont pas assez détaillées pour pouvoir être implémentées par des développeurs. UML propose de partir des classeurs découverts au moment de l'analyse (tels que les classes, les sous-systèmes, les cas d'utilisation, ...) et de les décomposer en éléments suffisamment fins pour permettre leur implémentation. Les classeur ainsi décomposés s'appellent des classeurs structurés.

Collaboration



Une collaboration montre des instances qui collaborent dans un contexte donné pour mettre en oeuvre une fonctionnalité d'un système.
Graphiquement, une collaboration se représente par une ellipse en trait pointillé comprenant deux compartiments. Le compartiment supérieur contient le nom de la collaboration et le compartiment inférieur montre les participants à la collaboration

- **Interactions et lignes de vie**

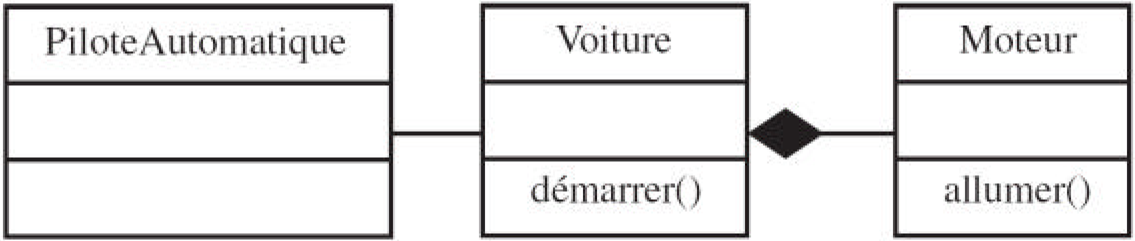


Diagramme de classe d'un système de pilotage.

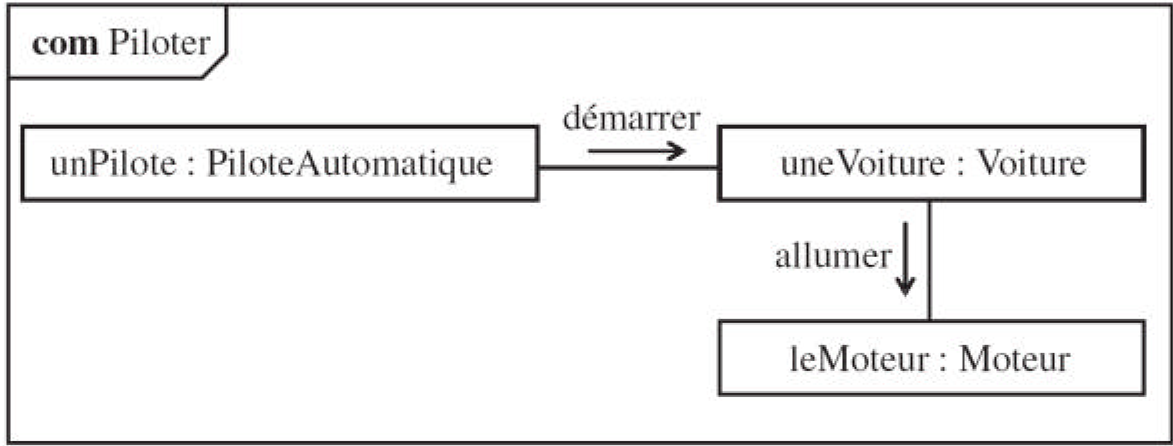


Diagramme de communication d'un système de pilotage.

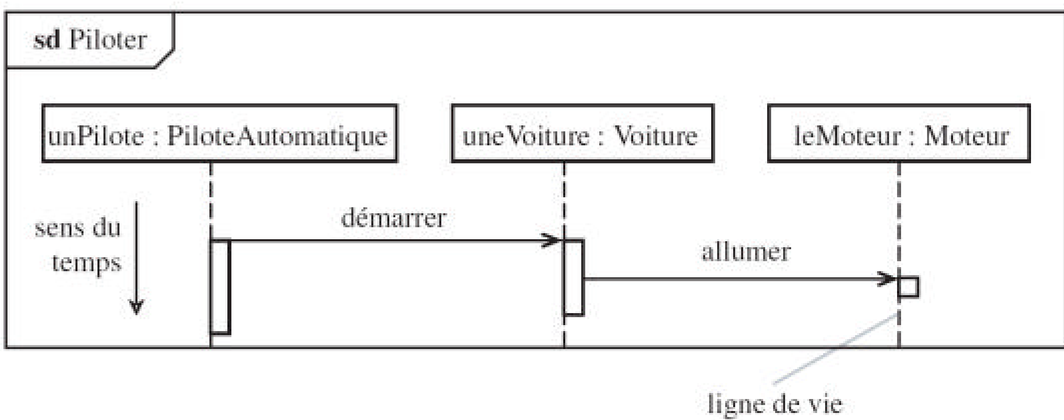


Diagramme de séquence d'un système de pilotage.

Une interaction montre le comportement d'un classeur structuré ou d'une collaboration en se focalisant sur l'échange d'informations entre les éléments du classeur ou de la collaboration. Une interaction contient un jeu de ligne de vie. Chaque ligne de vie correspond à une partie interne d'un classeur ou de la collaboration et représente une instance ou un jeu d'instances sur une période donnée. L'interaction décrit donc l'activité interne des éléments du classeur ou de la collaboration, appelés lignes de vie, et des messages qu'ils échangent.

Diagramme de communication
(Communication diagram)

- Contrairement à un diagramme de séquence, un diagramme de communication rend compte de l'organisation spatiale des participants à l'interaction, il est souvent utilisé pour illustrer un cas d'utilisation ou pour décrire une opération.

Représentation des lignes de vie

- Les lignes de vie sont représentées par des rectangles contenant une étiquette dont la syntaxe est : [<nom_du_rôle>] : [<Nom_du_type>] Au moins un des deux noms doit être spécifié dans l'étiquette, les deux points (:) sont, quand à eux, obligatoire.

Représentation des connecteurs

- Les relations entre les lignes de vie sont appelées connecteurs et se représentent par un trait plein reliant deux lignes de vies et dont les extrémités peuvent être ornées de multiplicités.

Représentation des messages

- Dans un diagramme de communication, les messages sont généralement ordonnés selon un numéro de séquence croissant.
- Un message est, habituellement, spécifié sous la forme suivante: ['['cond']' [séq] [*[||] '['iter']'] :] [r :=] msg([par])
- **cond**
  - est une condition sous forme d'expression booléenne entre crochets.
- **séq**
  - est le numéro de séquence du message. On numérote les messages par envoi et sous-envoi désignés par des chiffres séparés par des points : ainsi l'envoi du message 1.4.3 est antérieur à celui du message 1.4.4 mais postérieur à celui du message 1.3.5. La simultanéité d'un envoi est désignée par une lettre : les messages 1.6.a et 1.6.b sont envoyés en même temps.
- **iter**
  - spécifie (en langage naturel, entre crochets) l'envoi séquentiel (ou en parallèle, avec ||). On peut omettre cette spécification et ne garder que le caractère "*" (ou "*||") pour désigner un message récurrent envoyé un certain nombre de fois.
- **r**
  - est la valeur de retour du message, qui sera par exemple transmise en paramètre à un autre message.
- **msg**
  - est le nom du message.
- **par**
  - désigne les paramètres (optionnels) du message.
- Cette syntaxe un peu complexe permet de préciser parfaitement l'ordonnancement et la synchronisation des messages entre les objets du diagramme de communication. La direction d'un message est spécifiée par une flèche pointant vers l'un ou l'autre des objets de l'interaction, reliés par ailleurs avec un trait continu (connecteur).

Diagramme de séquence *(Sequence diagram)*

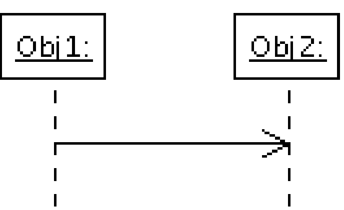
Les principales informations contenues dans un diagramme de séquence sont les messages échangés entre les lignes de vie, présentés dans un ordre chronologique. Ainsi, contrairement au diagramme de communication, le temps y est représenté explicitement par une dimension (la dimension verticale) et s'écoule de haut en bas.

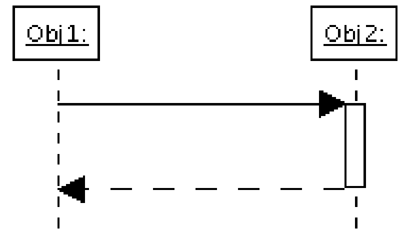
Représentation des lignes de vie

- Une ligne de vie se représente par un rectangle, auquel est accroché une ligne verticale pointillée, contenant une étiquette dont la syntaxe est : [<nom_du_rôle>] : [<Nom_du_type>] Au moins un des deux noms doit être spécifié dans l'étiquette, les deux points (:) sont, quand à eux, obligatoire.

Représentation des messages

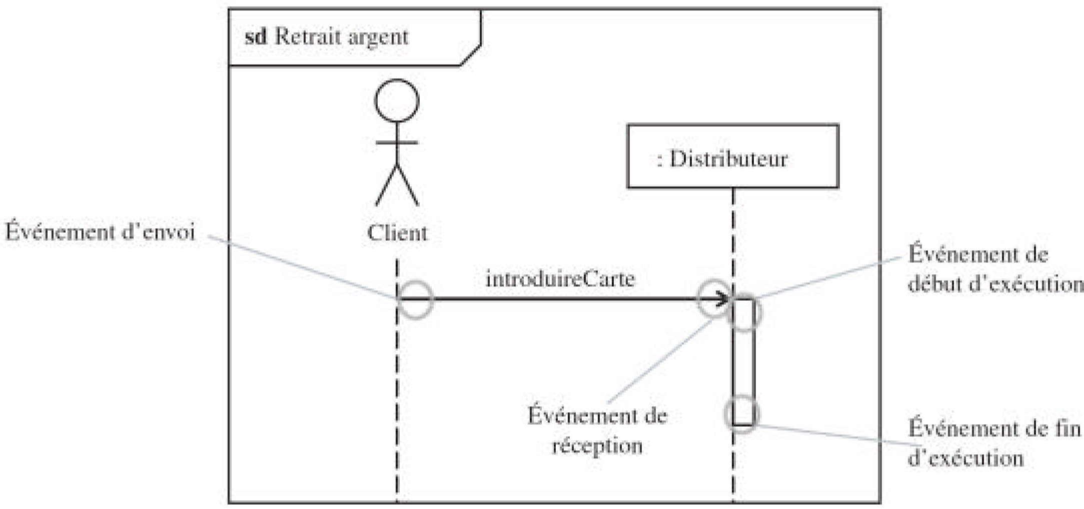
- **Messages synchrones et asynchrones, création et destruction d'instance**
- Un message définit une communication particulière entre des lignes de vie. Plusieurs types de messages existent, les plus commun sont :
- l'envoi d'un signal ;
- l'invocation d'une opération ;
- la création ou la destruction d'une instance.





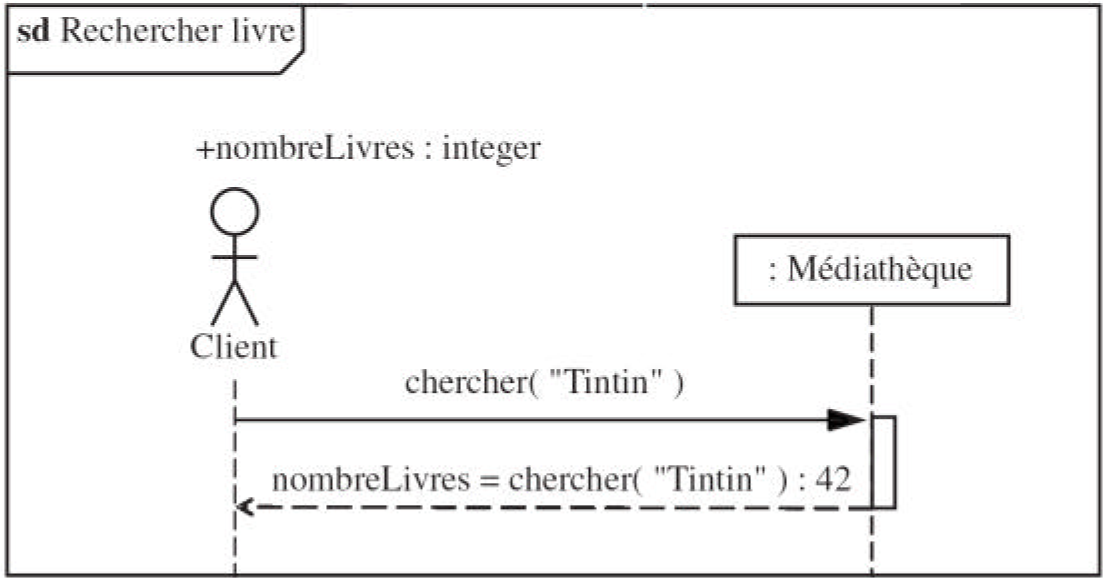
Message asynchrone. Message synchrone.

Événements et messages



Les différents évènement correspondant à un message asynchrone.

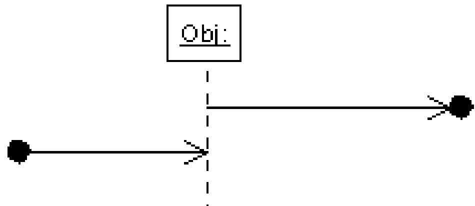
Syntaxe des messages et des réponses



- la direction du message est directement spécifiée par la direction de la flèche qui matérialise le message, et non par une flèche supplémentaire au dessus du connecteur reliant les objets comme c'est le cas dans un diagramme de communication ;
- les numéros de séquence sont généralement omis puisque l'ordre relatif des messages est déjà matérialisé par l'axe vertical qui représente l'écoulement du temps.
- La syntaxe de réponse à un message est la suivante : [<attribut> =] message [: <valeur_de_retour>]

Message perdu et trouvé

- Un message complet est tel que les événements d'envoi et de réception sont connus. Comme nous l'avons déjà vu, un message complet se représente par une simple flèche dirigée de l'émetteur vers le récepteur.



Porte

- Une porte est un point de connexion qui permet de représenter un même message dans plusieurs fragments d'interaction. Ces messages entrants et sortants vont d'un bord d'une diagramme à une ligne de vie (ou l'inverse).

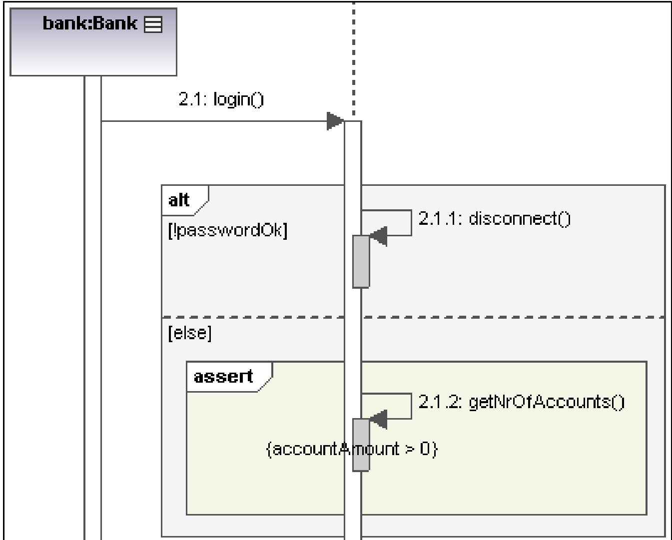
Exécution de méthode et objet actif

- Un objet actif initie et contrôle le flux d'activités. Graphiquement, la ligne pointillée verticale d'un objet actif est remplacée par un double trait vertical.

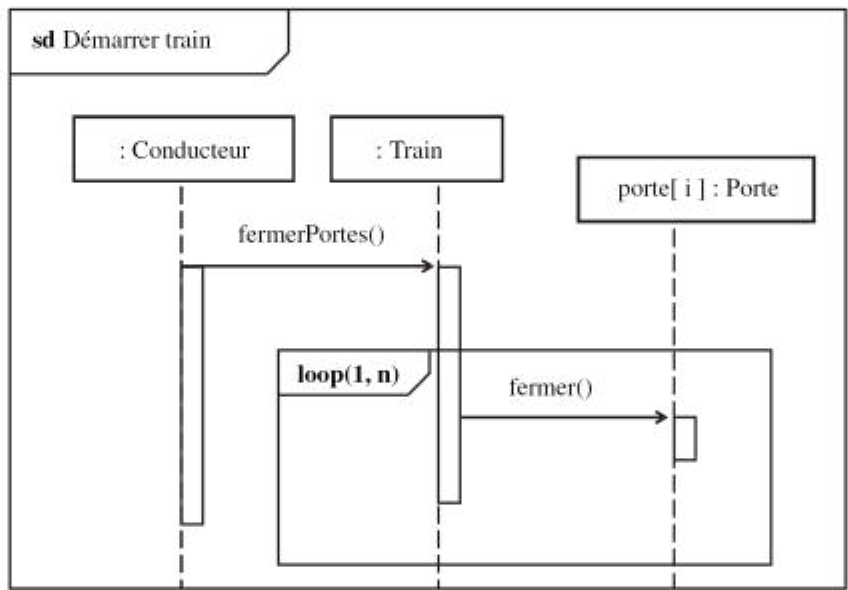
- **Fragments d'interaction combinés**
- Un fragment combiné se représente de la même façon qu'une interaction. Il est représenté un rectangle dont le coin supérieur gauche contient un pentagone. Dans le pentagone figure le type de la combinaison, appelé *opérateur d'interaction*. Les opérandes d'un opérateur d'interaction sont séparés par une ligne pointillée. Les conditions de choix des opérandes sont données par des expressions booléennes entre crochets ([ ]).

  La liste suivante regroupe les opérateurs d'interaction par fonctions :
- les opérateurs de choix et de boucle : **alternative**, **option**, **break** et **loop** ;
- les opérateurs contrôlant l'envoi en parallèle de messages : **parallel** et **critical region** ;
- les opérateurs contrôlant l'envoi de messages : **ignore**, **consider**, **assertion** et **negative** ;
- les opérateurs fixant l'ordre d'envoi des messages : **weak sequencing** , **strict sequencing**.

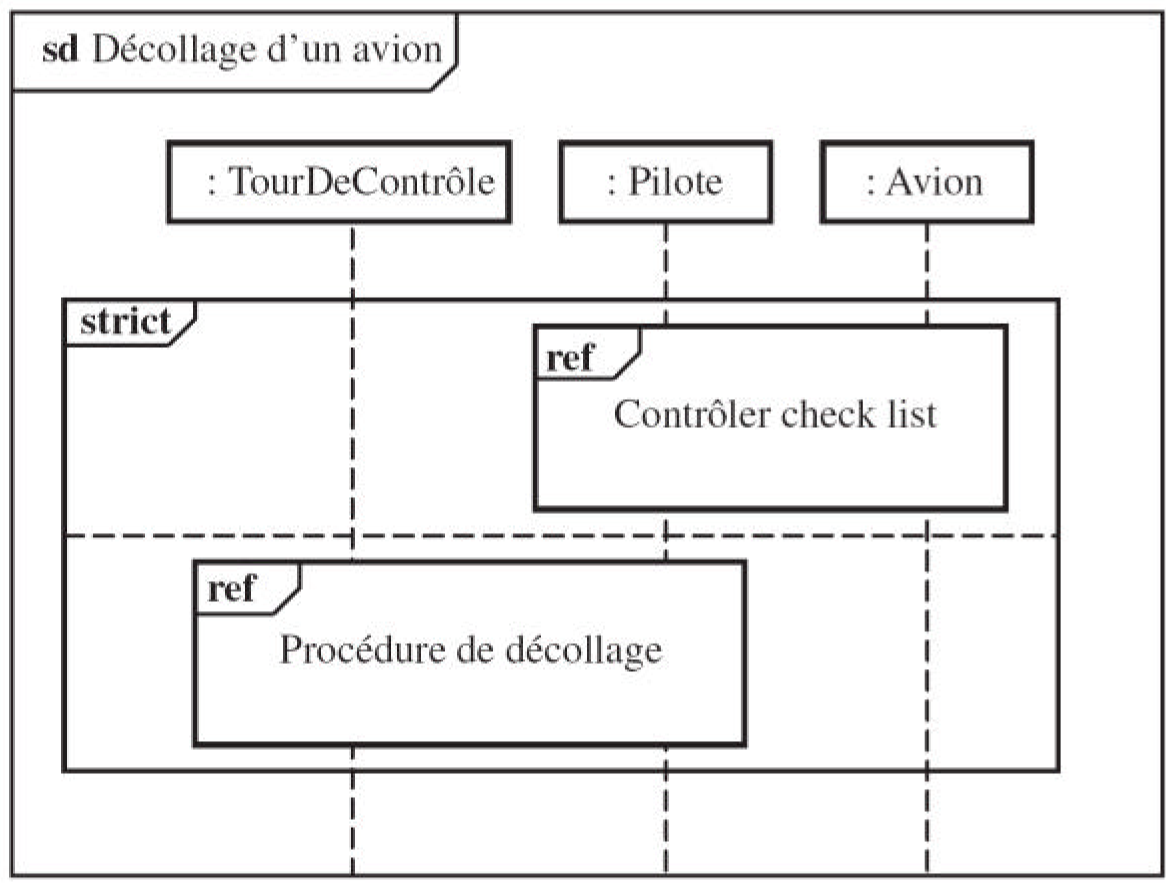
Opérateurs *alt* et *opt*



Opérateur *loop*



- **Opérateur *strict***



Un fragments combiné de type **strict sequencing**, ou *strict*, possède au moins deux sous-fragments. Ceux-ci s'exécutent selon leur ordre d'apparition au sein du fragment combiné. Ce fragment combiné est utile surtout lorsque deux parties d'un diagramme n'ont pas de ligne de vie en commun

- Les diagrammes de composants et les diagrammes de déploiement sont les deux derniers types de vues statiques en UML. Les premiers décrivent le système modélisé sous forme de composants réutilisables et mettent en évidence leurs relations de dépendance.

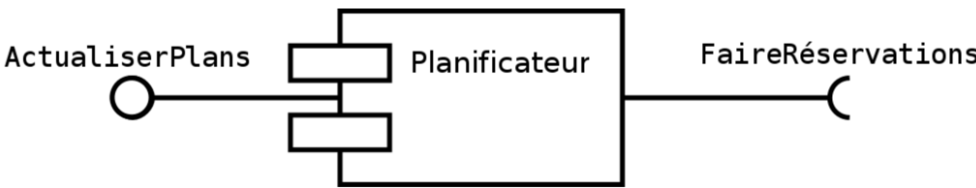
Diagrammes de composants

<<component>>
Planificateur
<<provided interface>>
ActualiserPlans
<<provided interface>>
FaireRéservations

Planificateur
<<provided interface>>
ActualiserPlans
<<provided interface>>
FaireRéservations

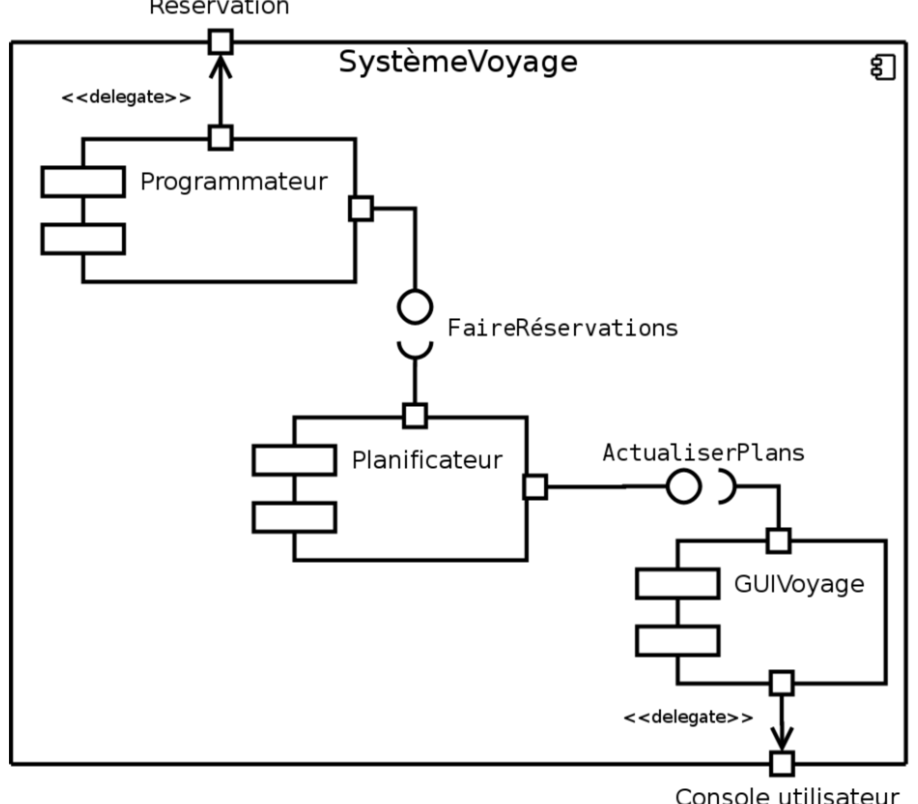
Représentation d'un composant et de ses interfaces requises ou offertes sous la forme d'un classeur structuré stéréotypé *«component»*. Au lieu ou en plus du mot clé, on peut faire figurer une icône de composant (petit rectangle équipé de deux rectangles plus petits dépassant sur son côté gauche) dans l'angle supérieur droit (comme sur la figure de droite).

Représentation d'un composant accompagnée de la représentation explicite de ses interfaces requise et offerte.



Représentation classique d'un composant et de ses interfaces requise (représenté par un demi-cercle) et offerte (représentée par un cercle).
Cette représentation est souvent utilisée dans les diagrammes de composants. Sur la figure du bas, le stéréotype « *component* » est rendu inutile par la représentation même du composant.

Représentation d'un composant et de ses interfaces requise et offerte avec la représentation explicite de leur port correspondant.



Représentation de l'implémentation d'un composant complexe contenant des sous-composants

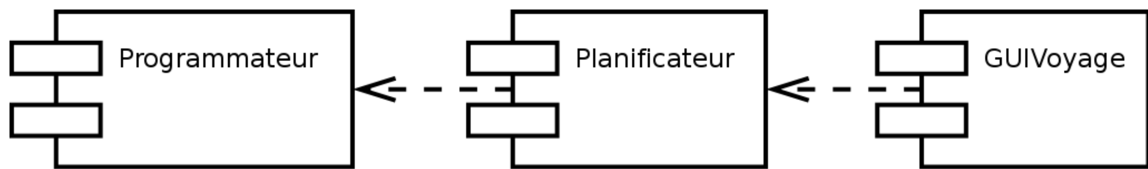
Notion de composant

- Un composant doit fournir un service bien précis. Les fonctionnalités qu'il encapsule doivent être cohérentes entre elles et génériques (par opposition à spécialisées) puisque sa vocation est d'être réutilisable.

Notion de port

- Un port est un point de connexion entre un classeur et son environnement. Graphiquement, un port est représenté par un petit carré à cheval sur la bordure du contour du classeur. On peut faire figurer le nom du port à proximité de sa représentation.

Diagramme de composants



La relation de dépendance est utilisée dans les diagrammes de composants pour indiquer qu'un élément de l'implémentation d'un composant fait appel aux services offerts par les éléments d'implémentation d'un autre composant

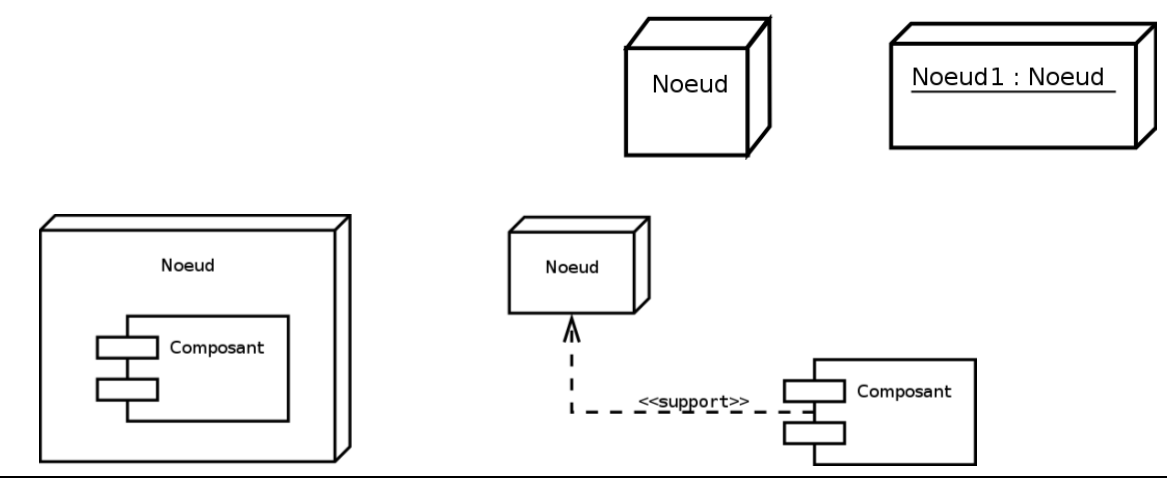
Diagramme de déploiement

Objectif du diagramme de déploiement

- Un diagramme de déploiement décrit la disposition physique des ressources matérielles qui composent le système et montre la répartition des composants sur ces matériels. Chaque ressource étant matérialisée par un noeud, le diagramme de déploiement précise comment les composants sont répartis sur les noeuds et quelles sont les connexions entre les composants ou les noeuds. Les diagrammes de déploiement existent sous deux formes : spécification et instance.

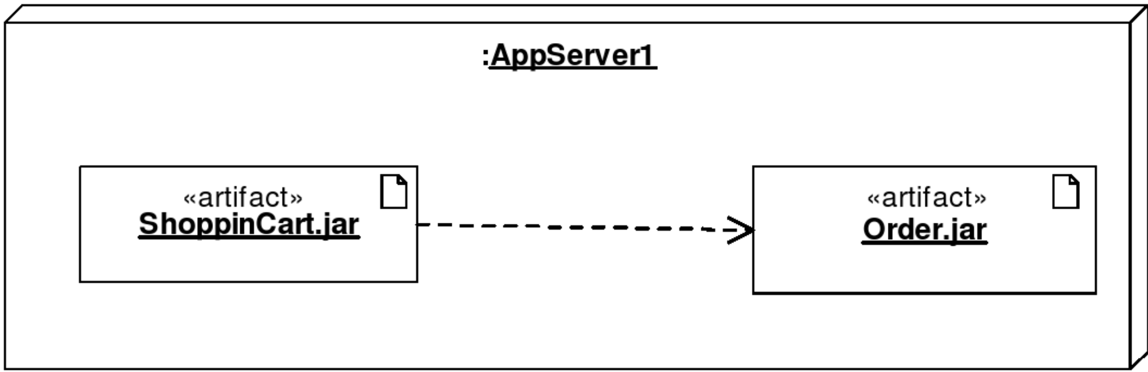
Représentation des noeuds

- Représentation d'un noeud (à gauche) et d'une instance de noeud (à droite).

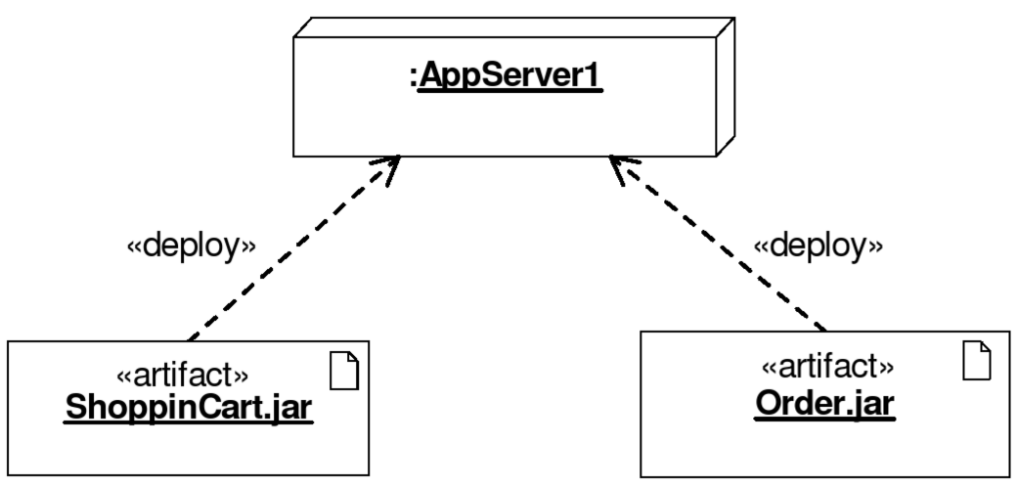


Notion d'artefact (*artifact*)

- Représentation du déploiement de deux artefacts dans un noeud. La dépendance entre les deux artefacts est également représentée. *



- Représentation du déploiement de deux artefacts dans un noeud utilisant la relation de dépendance stéréotypée *«deploy»*.



- Représentation du déploiement dans un noeud d'un artefact manifestant un composant.

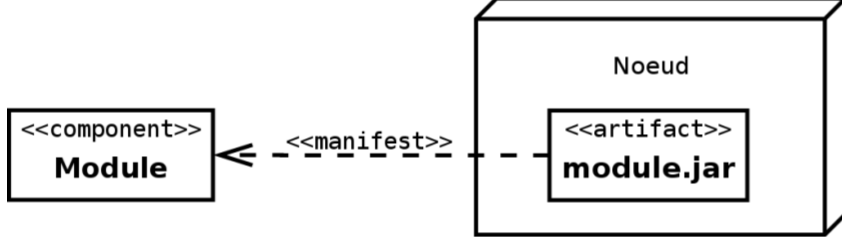
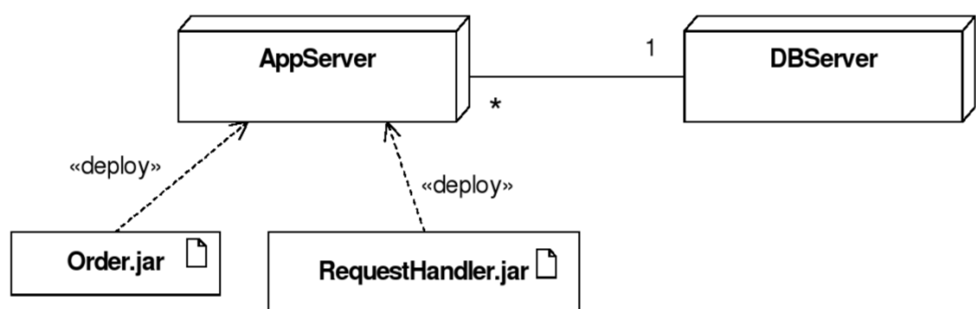


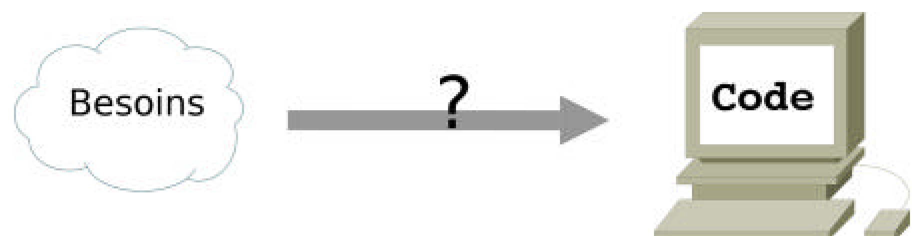
Diagramme de déploiement

- Exemple de diagramme de déploiement illustrant la communication entre plusieurs noeuds.



- Dans un diagramme de déploiement, les associations entre noeuds sont des chemins de communication qui permettent l'échange d'informations

Mise en oeuvre d'UML



- **Pilotée par les cas d'utilisation :**
  - La principale qualité d'un logiciel étant son utilité, c'est-à-dire son adéquation avec les besoins des utilisateurs, toutes les étapes, de la spécification des besoins à la maintenance, doivent être guidées par les cas d'utilisation qui modélisent justement les besoins des utilisateurs.

- **Centrée sur l'architecture :**
  - L'architecture est conçue pour satisfaire les besoins exprimés dans les cas d'utilisation, mais aussi pour prendre en compte les évolutions futures et les contraintes de réalisation. La mise en place d'une architecture adaptée conditionne le succès d'un développement. Il est important de la stabiliser le plus tôt possible.

- **Itérative et incrémentale :**
  - L'ensemble du problème est décomposé en petites itérations, définies à partir des cas d'utilisation et de l'étude des risques. Les risques majeurs et les cas d'utilisation les plus importants sont traités en priorité. Le développement procède par des itérations qui conduisent à des livraisons incrémentales du système.

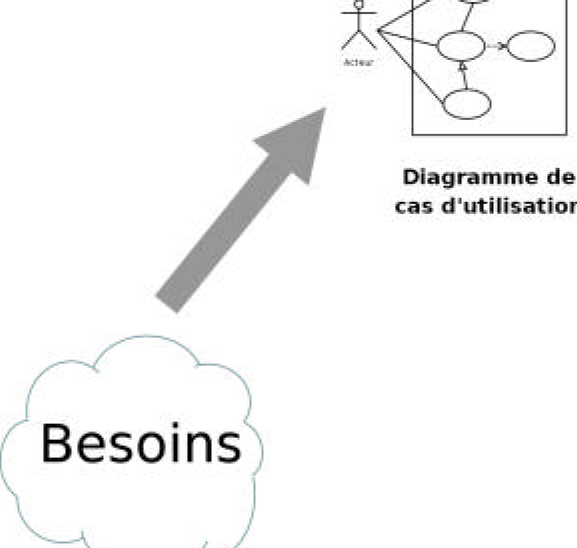
Identification et représentation des besoins : diagramme de cas d'utilisation

Une méthode simple et générique

- Les cas d'utilisation sont utilisés tout au long du projet. Dans un premier temps, on les crée pour identifier et modéliser les besoins des utilisateurs. Ces besoins sont déterminés à partir des informations recueillies lors des rencontres entre informaticiens et utilisateurs. Il faut impérativement proscrire toute considération de réalisation lors de cette étape.

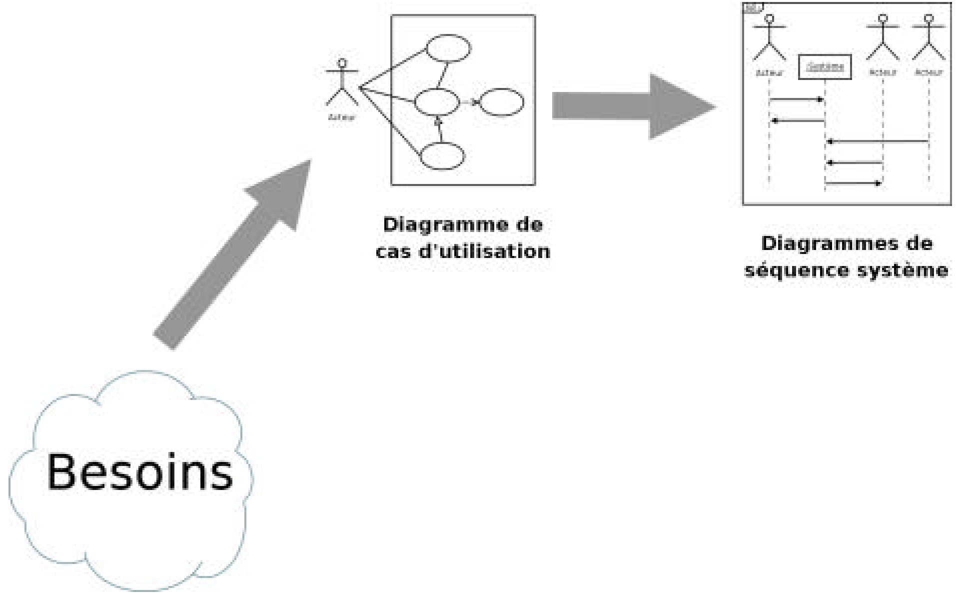
  Durant cette étape, vous devrez déterminer les limites du système, identifier les acteurs et recenser les cas d'utilisation (cf. section 2.5). Si l'application est complexe, vous pourrez organiser les cas d'utilisation en paquetages.

  Dans le cadre d'une approche itérative et incrémentale, il faut affecter un degré d'importance et un coefficient de risque à chacun des cas d'utilisation pour définir l'ordre des incréments à réaliser.



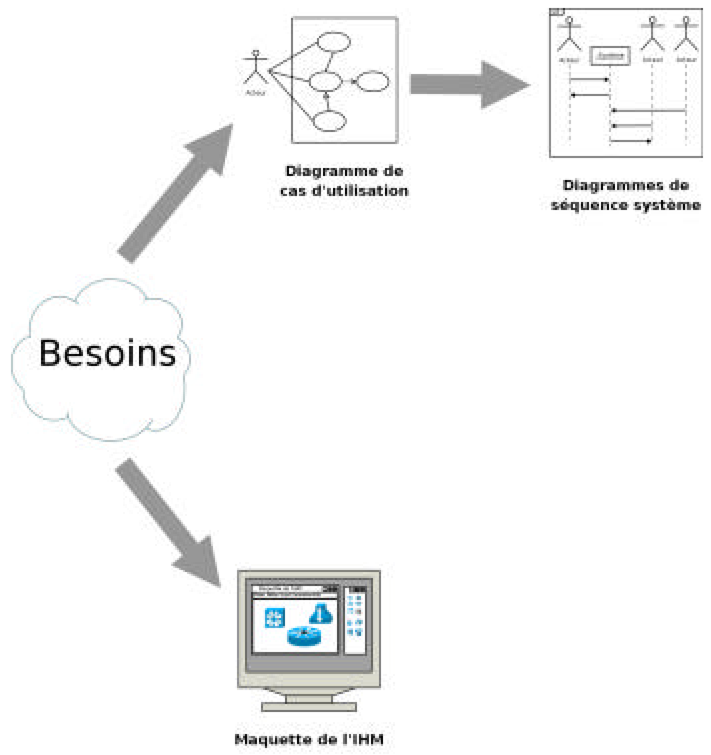
Spécification détaillée des besoins : diagrammes de séquence système

Dans cette étape, on cherche à détailler la description des besoins par la description textuelle des cas d'utilisation et la production de diagrammes de séquence système illustrant cette description textuelle. Cette étape amène souvent à mettre à jour le diagramme de cas d'utilisation puisque nous somme toujours dans la spécification des besoins.
Les scénarii de la description textuelle des cas d'utilisation peuvent être vus comme des instances de cas d'utilisation et sont illustrés par des diagrammes de séquence système. Il faut, au minimum, représenter le scénario nominal de chacun des cas d'utilisation par un diagramme de séquence qui rend compte de l'interaction entre l'acteur, ou les acteurs, et le système. Le système est ici considéré comme un tout et est représenté par une ligne de vie. Chaque acteur est également associé à une ligne de vie.
Lorsque les scénarii alternatifs d'un cas d'utilisation sont nombreux et importants, l'utilisation d'un diagramme d'états-transitions ou d'activités peut s'avérer préférable à une multitude de diagrammes de séquence.



Maquette de l'IHM de l'application (non couvert par UML)

- Une maquette d'IHM (Interface Homme-Machine) est un produit jetable permettant aux utilisateurs d'avoir une vue concrète mais non définitive de la future interface de l'application. La maquette peut très bien consister en un ensemble de dessins produits par un logiciel de présentation ou de dessin. Par la suite, la maquette pourra intégrer des fonctionnalités de navigation permettant à l'utilisateur de tester l'enchaînement des écrans ou des menus, même si les fonctionnalités restent fictives. La maquette doit être développée rapidement afin de provoquer des retours de la part des utilisateurs.



Phases d'analyse

- **Analyse du domaine : modèle du domaine**

Diagramme de cas d'utilisation

Diagrammes de séquence système

Besoins

Modèle du domaine

Maquette de l'IHM

- La modélisation des besoins par des cas d'utilisation s'apparente à une analyse fonctionnelle classique. L'élaboration du modèle des classes du domaine permet d'opérer une transition vers une véritable modélisation objet. L'analyse du domaine est une étape totalement dissociée de l'analyse des besoins. Elle peut être menée avant, en parallèle ou après cette dernière.

  La phase d'analyse du domaine permet d'élaborer la première version du diagramme de classes appelée modèle du domaine. Ce modèle doit définir les classes qui modélisent les entités ou concepts présents dans le domaine (on utilise aussi le terme de *métier*) de l'application. Il s'agit donc de produire un modèle des objets du monde réel dans un domaine donné. Ces entités ou concepts peuvent être identifiés directement à partir de la connaissance du domaine ou par des entretiens avec des experts du domaine. Il faut absolument utiliser le vocabulaire du métier pour nommer les classes et leurs attributs. Les classes du modèle du domaine ne doivent pas contenir d'opérations, mais seulement des attributs. Les étapes à suivre pour établir ce diagramme sont :
- identifier les entités ou concepts du domaine ;
- identifier et ajouter les associations et les attributs ;
- organiser et simplifier le modèle en éliminant les classes redondantes et en utilisant l'héritage ;
- le cas échéant, structurer les classes en paquetage selon les principes de cohérence et d'indépendance.
- L'erreur la plus courante lors de la création d'un modèle du domaine consiste à modéliser un concept par un attribut alors que ce dernier devait être modélisé par une classe. Si la seule chose que recouvre un concept est sa valeur, il s'agit simplement d'un attribut. Par contre, si un concept recouvre un ensemble d'informations, alors il s'agit plutôt d'une classe qui possède elle-même plusieurs attributs.

Diagramme de classes participantes

Diagramme de cas d'utilisation

Diagrammes de séquence système

Besoins

Modèle du domaine

Diagramme de classes participantes

Maquette de l'IHM

Lors de l'élaboration du diagramme de classes participantes, il faut veiller au respect des règles suivantes :

- Les entités, qui sont issues du modèle du domaine, ne comportent que des attributs.
- Les entités ne peuvent être en association qu'avec d'autres entités ou avec des contrôles, mais, dans ce dernier cas, avec une contrainte de navigabilité interdisant de traverser une association d'une entité vers un contrôle.
- Les contrôles ne comportent que des opérations. Ils implémentent la logique applicative (*i.e.* les fonctionnalités de l'application), et peuvent correspondre à des règles transverses à plusieurs entités. Chaque contrôle est généralement associé à un cas d'utilisation, et *vice versa*. Mais rien n'empêche de décomposer un cas d'utilisation complexe en plusieurs contrôles.
- Les contrôles peuvent être associés à tous les types de classes, y compris d'autres contrôles. Dans le cas d'une association entre un dialogue et un contrôle, une contrainte de navigabilité doit interdire de traverser l'association du contrôle vers le dialogue.
- Les dialogues comportent des attributs et des opérations. Les attributs représentent des informations ou des paramètres saisis par l'utilisateur ou des résultats d'actions. Les opérations réalisent (généralement par délégation aux contrôles) les actions que l'utilisateur demande par le biais de l'IHM.
- Les dialogues peuvent être en association avec des contrôles ou d'autres dialoques, mais pas directement avec des entités.
- Il est également possible d'ajouter les acteurs sur le diagramme de classes participantes en respectant la règle suivante : un acteur ne peut être lié qu'à un dialogue.

Diagrammes d'activités de navigation

Besoins

Diagramme de cas d'utilisation

Diagrammes de séquence système

Modèle du domaine

Diagramme de classes participantes

Maquette de l'IHM

Diagramme d'activité de navigation

- Les IHM modernes facilitent la communication entre l'application et l'utilisateur en offrant toute une gamme de moyens d'action et de visualisation comme des menus déroulants ou contextuels, des palettes d'outils, des boîtes de dialogues, des fenêtres de visualisation, etc. Cette combinaison possible d'options d'affichage, d'interaction et de navigation aboutis aujourd'hui à des interfaces de plus en plus riches et puissantes.

  UML offre la possibilité de représenter graphiquement cette activité de navigation dans l'interface en produisant des diagrammes dynamiques. On appelle ces diagrammes des diagrammes de navigation. Le concepteur à le choix d'opter pour cette modélisation entre des diagrammes d'états-transitions et des diagrammes d'activités. Les diagrammes d'activités constituent peut-être un choix plus souple et plus judicieux.

  Les diagrammes d'activités de navigation sont à relier aux classes de dialogue du diagramme de classes participantes. Les différentes activités du diagramme de navigation peuvent être stéréotypées en fonction de leur nature : « *fenêtre* », « *menu* », « *menu contextuel* », « *dialogue* », etc.

  La modélisation de la navigation à intérêt à être structurée par acteur.

Phase de conception
Diagrammes d'interaction

Besoins

Diagramme de cas d'utilisation

Diagrammes de séquence système

Diagrammes d'interaction

Modèle du domaine

Diagramme de classes participantes

Maquette de l'IHM

Diagramme d'activité de navigation

Maintenant, il faut attribuer précisément les responsabilités de comportement, dégagée par le diagrammes de séquence système, aux classes d'analyse du diagramme de classes participantes élaboré. Les résultats de cette réflexion sont présentés sous la forme de diagrammes d'interaction UML. Inversement, l'élaboration de ces diagrammes facilite grandement la réflexion.

Parallèlement, une première ébauche de la vue statique de conception, c'est-à-dire du diagramme de classes de conception, est construite et complétée. Durant cette phase, l'ébauche du diagramme de classes de conception reste indépendante des choix technologiques qui seront faits ultérieurement .

Pour chaque service ou fonction, il faut décider quelle est la classe qui va le contenir. Les diagrammes d'interactions (*i.e* de séquence ou de communication) sont particulièrement utiles au concepteur pour représenter graphiquement ces décisions d'allocations des responsabilités. Chaque diagramme va représenter un ensemble d'objets de classes différentes collaborant dans le cadre d'un scénario d'exécution du système.

Dans les diagrammes d'interaction, les objets communiquent en s'envoyant des messages qui invoquent des opérations sur les objets récepteurs. Il est ainsi possible de suivre visuellement les interactions dynamiques entre objets, et les traitements réalisés par chacun d'eux. Avec un outil de modélisation UML (comme Rational Rose ou PowerAMC), la spécification de l'envoi d'un message entre deux objets crée effectivement une opération publique sur la classe de l'objet cible. Ce type d'outil permet réellement de mettre en oeuvre l'allocation des responsabilités à partir des diagrammes d'interaction.

sd : diagramme de séquence système

Acteur

Le système

action

réponse

sd : diagramme d'interaction

Le système

Acteur

: Dialogue

:Contrôle

:Entitée

action

opération()

getDonnée()

rep=getDonnée()

setDonnée(val)

réponse

Diagramme de classes de conception

Diagramme de cas d'utilisation

Diagrammes de séquence système

Diagrammes d'interaction

Besoins

Modèle du domaine

Diagramme de classes participantes

Code

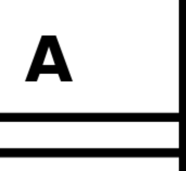
Diagramme de classes de conception

Maquette de l'IHM

Diagramme d'activité de navigation

Implémentation en Java

- **Classe**



Parfois, la génération automatique de code produit, pour chaque classe, un constructeur et une méthode finalize comme ci-dessous. Rappelons que cette méthode est invoquée par le *ramasse miettes* lorsque celui-ci constate que l'objet n'est plus référencé. Pour des raisons de simplification, nous ne ferons plus figurer ces opérations dans les sections suivantes.

```
public class A {
  public A() {
    ...
  }
  protected void finalize() throws Throwable {
    super.finalize();
    ...
  }
}
```

Classe avec attributs et opérations

```
public class A {
  public    String a1;
  package   String a2;
  protected String a3;
  private   String a4;
  public void op1() {
    ...
  }
  public void op2() {
    ...
  }
}
```

A

+a1: String
 a2: String
#a3: String
-a4: String

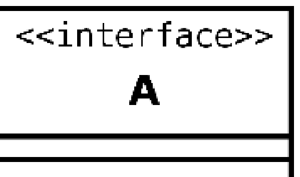
+op1()
+op2()

```
public abstract class A {
  ...
}
```

A {abstract}

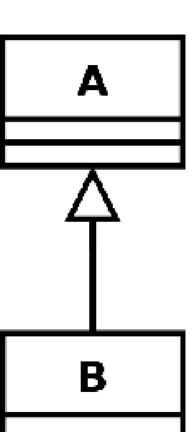
Interface

```
public interface A {
  ...
}
```

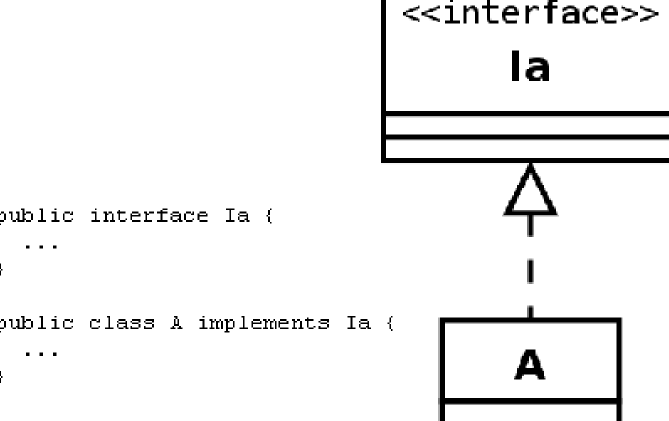


Héritage simple

```
public class A {
  ...
}

public class B extends A {
  ...
}
```

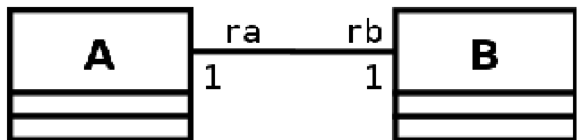


Réalisation d'une interface par une classe



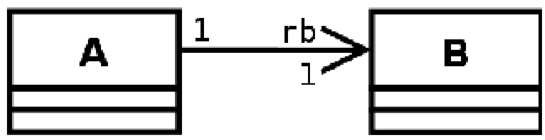

```
public interface Ia {
  ...
}

public class A implements Ia {
  ...
}
```

Association bidirectionnelle 1 vers 1



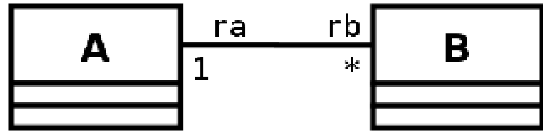

```
public class A {
  private B rb;
  public void addB( B b ) {
  if( b != null ){
    if ( b.getA() != null ) {   // si b est déjà connecté à un autre A
      b.getA().setB(null);    // cet autre A doit se déconnecter
    }
    this.setB( b );
    b.setA( this );
  }
  public B getB() { return( rb ); }
  public void setB( B b ) { this.rb=b; }
}

public class B {
  private A ra;
  public void addA( A a ) {
    if( a != null ) {
      if (a.getB() != null) {   // si a est déjà connecté à un autre B
        a.getB().setA( null );  // cet autre B doit se déconnecter
      }
      this.setA( a );
      a.setB( this );
    }
  }
  public void setA(A a){ this.ra=a; }
  public A getA(){ return(ra); }
}
```

Association unidirectionnelle 1 vers 1



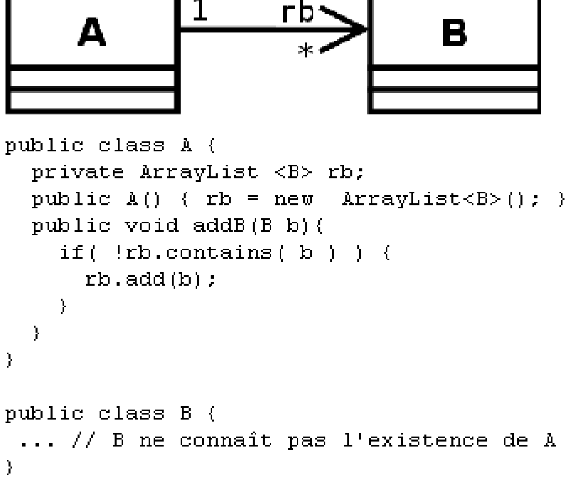

```
public class A {
  private B rb;
  public void addB( B b ) {
  if( b != null ) {
    this.setB( b );
  }
  public void setB( B b ) { this.rb=b; }
}

public class B {
 ... // La classe B ne connaît pas l'existence de la classe A
}
```

Association bidirectionnelle 1 vers plusieurs



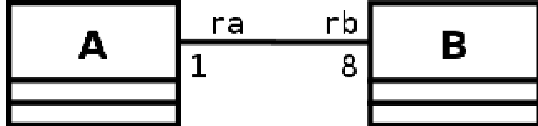

```
public class A {
  private ArrayList <B> rb;
  public A() { rb = new ArrayList<B>(); }
  public void addB( B b ) {
    if( !rb.contains( b ) ) {
      b.setA( this );
      rb.add( b );
    }
  }
  public ArrayList <B> getRB() { return(rb); }
}

public class B {
  private A ra;
  public void addA( A a ) {
     if( a != null ) {
       if( !a.getArray().contains( this ) ) {
         this.setA( a );
         ra.getRB().add( this );
       }
     }
  }
  public void setA(A a) { this.ra=a; }
}
```

Association unidirectionnelle 1 vers plusieurs




```
public class A {
  private ArrayList <B> rb;
  public A() { rb = new  ArrayList<B>(); }
  public void addB(B b){
    if( !rb.contains( b ) ) {
      rb.add(b);
    }
  }
}

public class B {
 ... // B ne connaît pas l'existence de A
}
```

Association 1 vers N

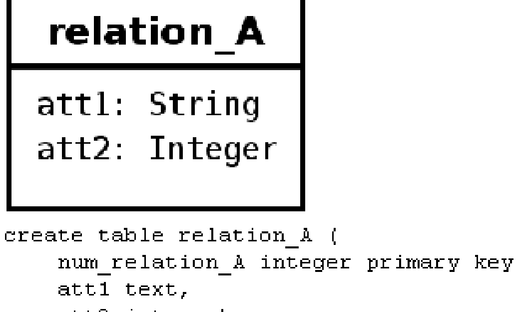


Dans ce cas, il faut utiliser un tableau plutôt qu'un vecteur. La dimension du tableau étant données par la cardinalité de la terminaison d'association.

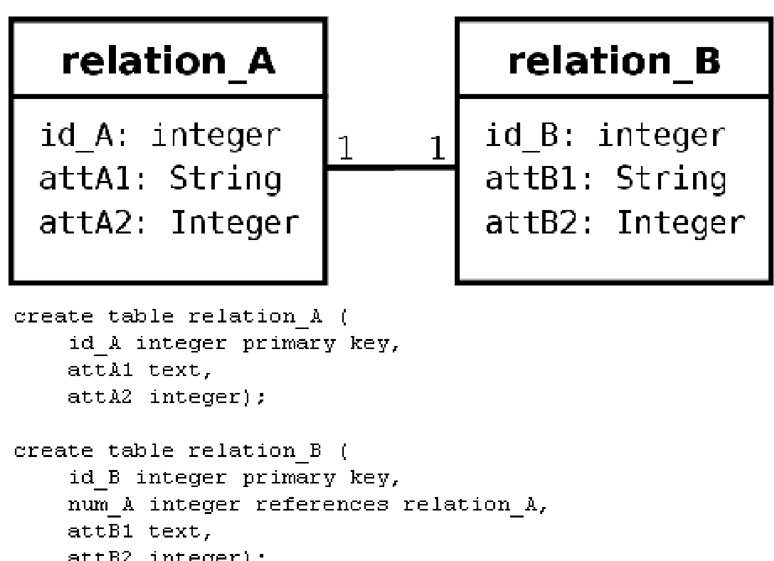
Implémentation en SQL

- Il est possible de traduire un diagramme de classe en modèle relationnel. Bien entendu, les méthodes des classes ne sont pas traduites. Aujourd'hui, lors de la conception de base de données, il devient de plus en plus courant d'utiliser la modélisation UML plutôt que le traditionnel modèle entités-associations.

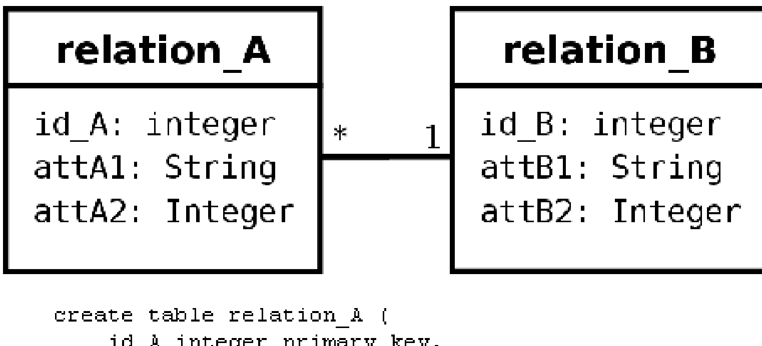
Classe avec attributs



Association 1 vers 1

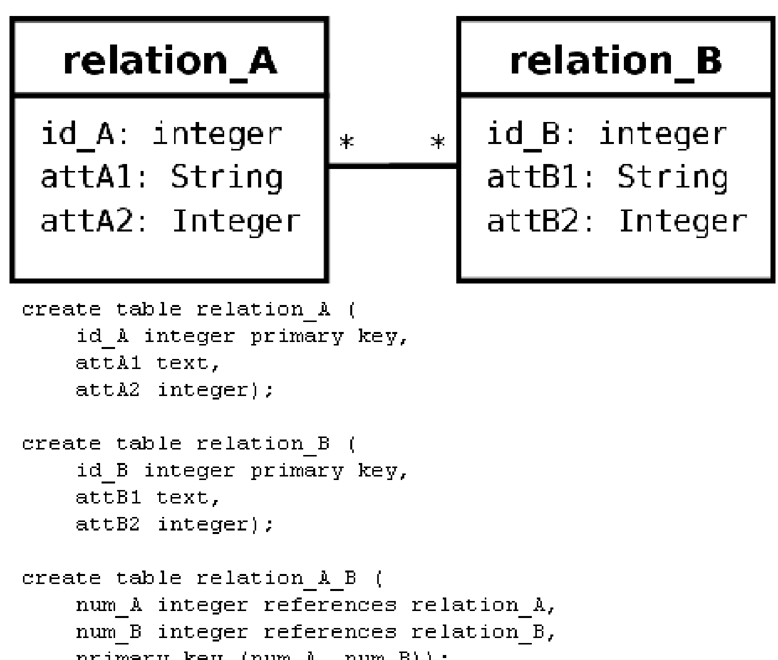


Association 1 vers plusieurs




```
create table relation_A (
    id_A integer primary key,
    num_B integer references relation_B,
    attA1 text,
    attA2 integer);

create table relation_B (
    id_B integer primary key,
    attB1 text,
    attB2 integer);
```

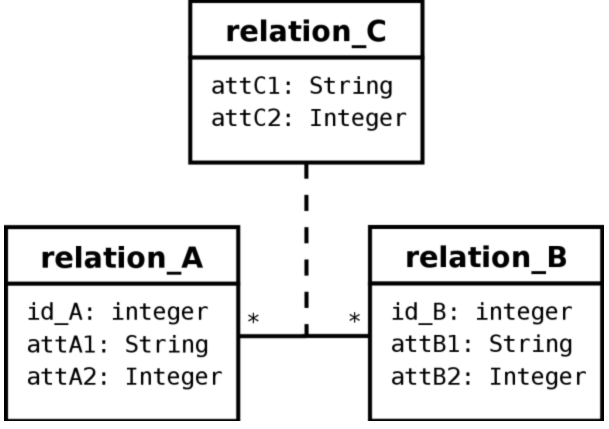
Association plusieurs vers plusieurs




```
create table relation_A (
    id_A integer primary key,
    attA1 text,
    attA2 integer);

create table relation_B (
    id_B integer primary key,
    attB1 text,
    attB2 integer);

create table relation_A_B (
    num_A integer references relation_A,
    num_B integer references relation_B,
    primary key (num_A, num_B));
```

Classe-association plusieurs vers plusieurs

```
create table relation_A (
    id_A integer primary key,
    attA1 text,
    attA2 integer);

create table relation_B (
    id_B integer primary key,
    attB1 text,
    attB2 integer);

create table relation_C (
    num_A integer references relation_A,
    num_B integer references relation_B,
    attC1 text,
    attC2 integer,
    primary key (num_A, num_B));
```



Héritage

```
create table relation_C (
    id_C integer primary key,
    attC1 text,
    attC2 integer,
    type text);

create table relation_A (
    id_A references relation_C,
    attA1 text,
    attA2 integer,
    primary key (id_A));

create table relation_B (
    id_B references relation_C,
    attB1 text,
    attB2 integer,
    primary key (id_B));
```

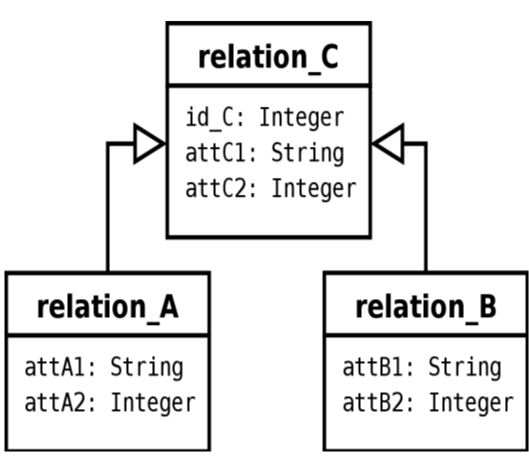
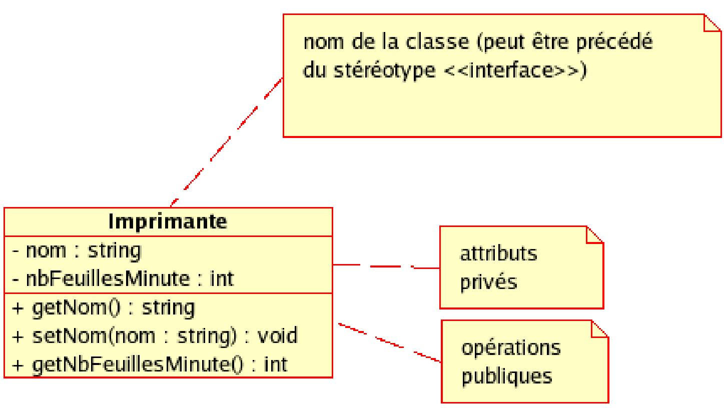


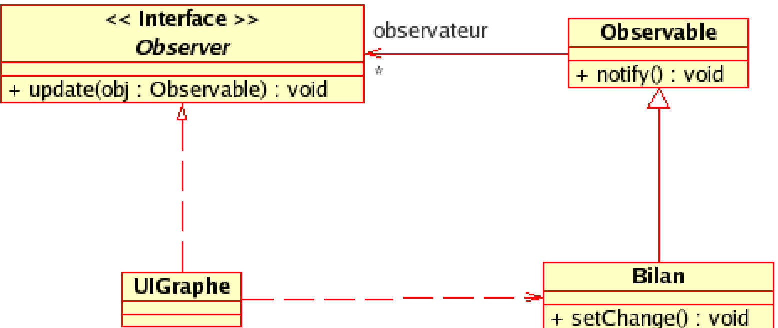
Figure 1. Exemple de représentation d'une classe avec UML



Vous concevrez un diagramme de classes UML mettant en avant les relations significatives entre classes décrites par l'extrait de code java ci-dessous.

Extrait du code source :

```
public interface Observer {
  public void update(Observable o);
}

public class Observable {
  Collection observateurs;
  public void notify() {
    Iterator it = this.iterator();
    while (it.hasNext()) {
      ((Observer) it.next()).update(this);
    }
  }
  public void addObserver(Observer o) { observateurs.add(o); }
    ...
  }

public class Bilan extends Observable {
  void setChange() { notify(); }
  ...
}

public class UIGraphe implements Observer {
  public void update(Observable o) {
    Bilan unbilan = (Bilan) o;
    double compteResultat = unbilan.getCompteResultat();
    ...
  }
 ...
}
```


<< Interface >>
Observer
+ update(obj : Observable) : void
observateur
*
Observable
+ notify() : void
UIGraphe
Bilan
+ setChange() : void

Exercice 1

Une entreprise de fabrication et de distribution de matériels possède une usine et plusieurs lieux de stockage/expédition.

Un produit est caractérisé par un numéro (NOP), un libellé (LIB), un prix unitaire (PU).

Chaque produit peut être stocké dans un ou plusieurs dépôts. Un dépôt est caractérisé par un numéro (NOD). Dans chaque dépôt on connaît la quantité en stock de chaque produit (QTS) et la quantité disponible (QTD) (la différence représente la quantité réservée pour des commandes déjà validées mais non livrées).

Un client est déterminé par son numéro (NOCLI), son nom (NOM), son adresse (ADR), le total de son chiffre d'affaire (CA), le taux de réduction (RED). Chaque client est livré à partir d'un dépôt privilégié, ou à partir d'un dépôt de secours en cas de défaillance du premier.

À un client peuvent être associées une ou plusieurs commandes, chacune étant caractérisée par un numéro (NOCOM) et une date (DAC). Une ligne comporte un code produit, une quantité commandée (QTC), un délai de livraison (DEL) et un code livraison (CL) indiquant si la livraison est intervenue.

À chaque commande peuvent être associées une ou plusieurs factures, une facture étant élaborée dès qu'une livraison est intervenue. Une facture est caractérisée par un numéro de facture (NOF), une date (DAP), un montant (MOF). Une facture peut concerner plusieurs produits. Chaque ligne comprend la quantité facturée (QTF) et le montant correspondant (MOP).

Proposer un schéma conceptuel des données à l'aide d'un diagramme de classes UML.

Exercice 2

Une compagnie aérienne veut mettre en œuvre une base de données pour gérer ses différents vols.

Un VOL est un parcours aérien caractérisé par un NUMERO, une VILLE-DEPART, une VILLE-ARRIVEE, une HEURE-DEPART, une HEURE-ARRIVEE, une DISTANCE, une FREQUENCE.

Lorsqu'un VOL est programmé pour une DATE déterminée il constitue un DEPART. Un VOL n'est programmé qu'une seule fois dans une journée à l'heure prévue.

Un certain nombre de PASSAGERS peut être enregistré pour un DEPART. Un PASSAGER est caractérisé par son NOM, son ADRESSE et son NO-TELEPHONE.

Un AVION est affecté à chaque DEPART. Un AVION est caractérisé par un NUMERO, un TYPE, une CAPACITE. Un AVION utilise une certaine QUANTITE DE CARBURANT pour accomplir le trajet. Cette dernière dépend des conditions atmosphériques, donc de la DATE.

Un certain nombre de PERSONNELS est affecté à chaque DEPART. On distingue les personnels non-navigants des personnels navigants. Parmi ces derniers, on distingue le(s) pilote(s). Un membre du personnel est caractérisé par son NOM, son ADRESSE, son NO-TELEPHONE.

1) On désire utiliser cette base pour produire (entre autres) les listes suivantes :

- passagers enregistrés pour un départ,
- personnels affectés à un départ pour chacune des trois catégories,
- départs programmés pour un vol donné,
- départs assignés à un avion donné pour la semaine à venir,
- caractéristiques du vol correspondant à un départ.

Proposer un diagramme de classes UML pour cette base (tous les éléments figurant en majuscules dans l'énoncé doivent être pris en compte), en exploitant au maximum la relation d'héritage.

2) Un VOL peut en fait être constitué de plusieurs tronçons. Par ailleurs on souhaite pouvoir établir pour chaque VILLE les vols au départ et les vols à l'arrivée.