

Modélisation et Conception de logiciels

- Cours 2 -

Maud MARCHAL

15 septembre 2010

Quatrième Année du Département Informatique
INSA de Rennes

1

Plan du cours

- **Modélisation statique UML (fin) :**
 - Diagramme d'objets
 - Diagramme de composants
 - Diagramme de déploiement
- **Object Constraint Language (OCL)**

2

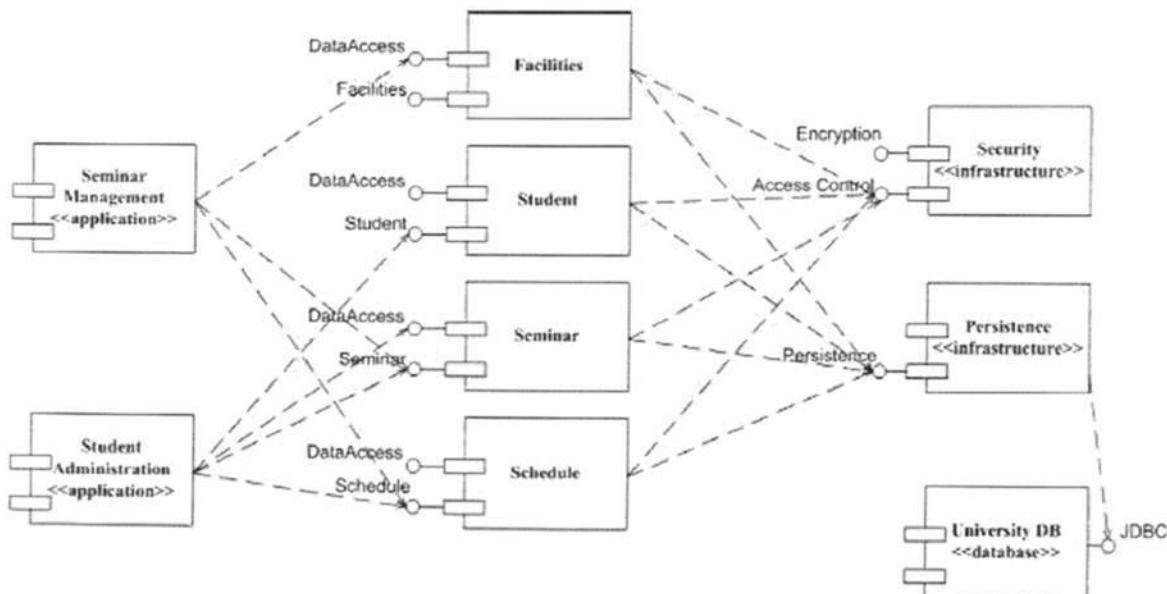
Diagrammes de composants

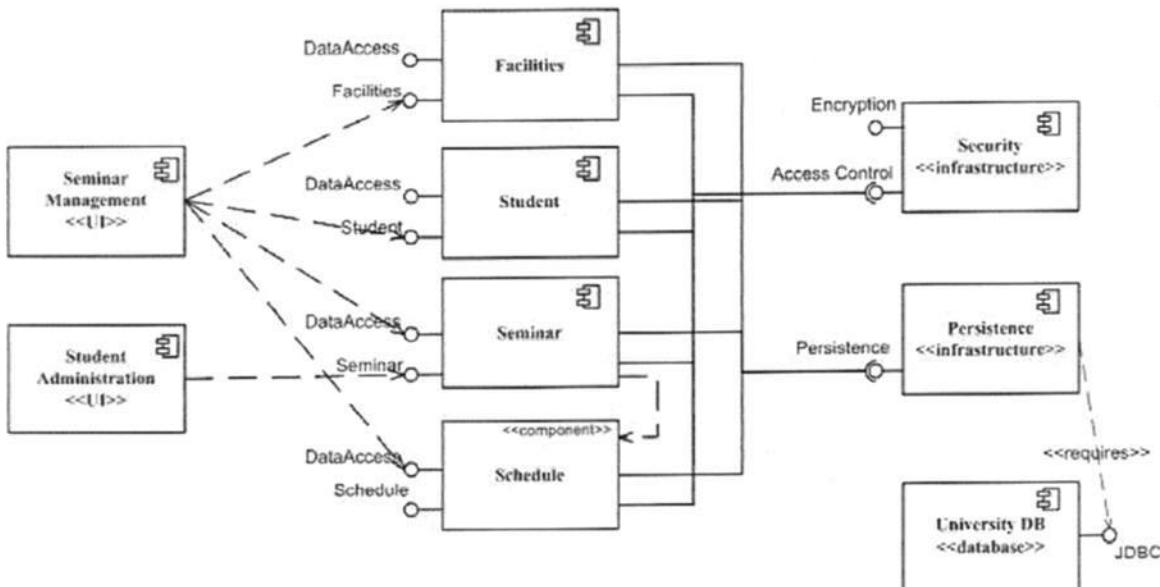
Définition

- **Définition :** un diagramme de composants :
 - ▶ donne une vue statique de l'implémentation du système illustrant les choix de réalisation ;
 - ▶ décrit les composants physiques d'un système et fournit la réalisation d'interface(s).
- **Objectif :**
 - ▶ représenter l'organisation et les dépendances entre les composants logiciels ;
 - ▶ décrire les composants et leurs relations dans le système en construction.
- L'utilisation de composants est assimilable à une approche orientée objet, non pas au niveau du code mais au niveau de l'architecture générale du logiciel.

- Les diagrammes de composants sont composés :
 - des descriptions des implémentations du système (**les composants**) ;
 - des groupes d'implémentations (**les modules**) ;
 - des relations entre les diverses implémentations (**les dépendances**) .
- Remarque :
 - UML1 : composant = n'importe quel élément, y compris fichiers, bibliothèque, etc.
 - UML2 :
 - changement de notation ;
 - utilisation des artefacts (= classe avec stéréotype «artefact») pour représenter les structures physiques (jar, dll, etc.).

Notation : UML 1





Diagrammes de composants

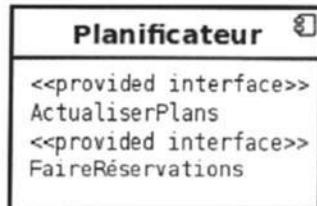
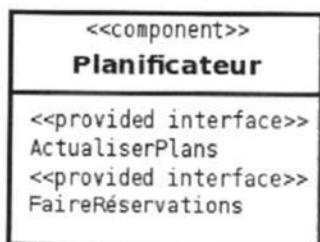
7

Les composants

- **Composant** = élément physique représentant une partie de l'implémentation du système :
 - code (source, binaire, exécutable) ;
 - script, fichier de commande ;
 - fichier de données, table, etc.
- Les fonctionnalités d'un composant doivent être cohérentes entre elles et génériques.
- Un composant implante des services utilisables par d'autres composants.

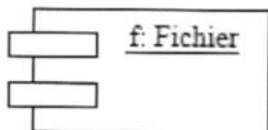
Représentation d'un composant

- Une composant est représenté par un classeur structuré, avec le stéréotype « component ».
- Une icône de composant (petit rectangle avec deux rectangles plus petits dépassant sur son côté gauche) peut être ajoutée à la place ou en plus du mot-clé, dans l'angle supérieur droit.



Instance d'un composant

- L'instance d'un composant est représentée par un composant dont le nom est souligné :



- Un diagramme de déploiement (cf. prochaine partie) est souvent utilisé pour illustrer les instances des composants.

Représentation d'un composant (2)

- Une composant peut comporter une ou plusieurs interfaces.
- Les interfaces peuvent être représentées explicitement :



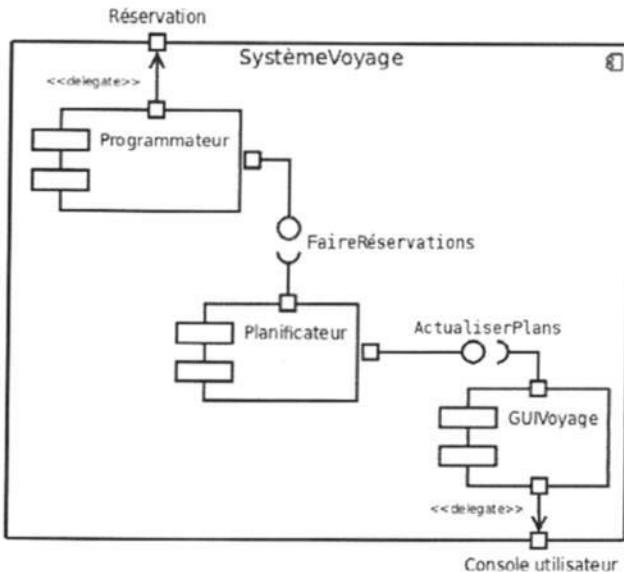
- La représentation la plus classique (interface requise représentée par un demi-cercle et interface offerte par un cercle) :



Structure interne d'un composant

- Le comportement interne d'un composant est totalement masqué : seules ses interfaces sont visibles.
- La seule contrainte pour pouvoir substituer un composant à un autre est de respecter les interfaces requises et offertes.
- L'implémentation d'un composant peut être réalisée par d'autres composants, des classes ou des artefacts.

- Les éléments d'un composant peuvent être représentés dans le symbole du composant ou à côté en les reliant au composant par une relation de dépendance.



Notion de port

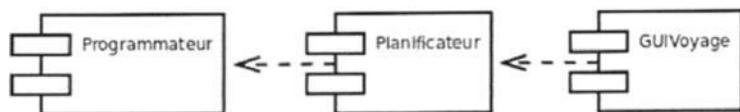
- Un port** = point de connexion entre un classeur et son environnement.
- Représentation graphique : petit carré à cheval sur la bordure du composant. Le nom du port peut également être ajouté à proximité.
- Port et interface sont souvent associés :



La liaison (= connecteur de délégation) est représentée par un trait plein (et éventuellement le stéréotype « delegate »).

Dépendance

- Un diagramme de composant peut comporter des relations de dépendances entre composants/modules.
- La relation de dépendance illustre l'utilisation des services d'un composant par un second composant.

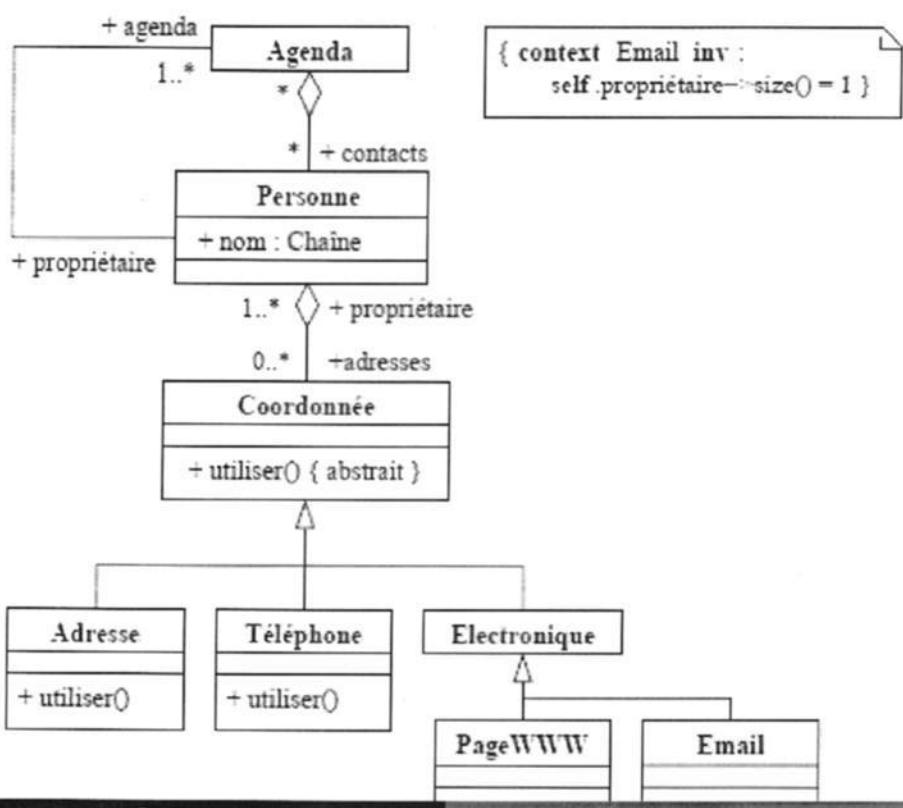


- L'utilisation d'une interface d'un composant par un autre composant peut être représentée en imbriquant le demi-cercle d'une interface requise dans le cercle de l'interface offerte correspondante.

Exemple

- Nous désirons implanter la gestion d'un agenda.
- Un agenda contient un ensemble de personnes.
- Un agenda possède un propriétaire.
- Chaque personne est identifiée par son nom et par un ensemble de coordonnées.
- Une coordonnée peut être postale, téléphonique ou électronique (email ou page web).
- Une adresse mail n'appartient qu'à une seule personne.

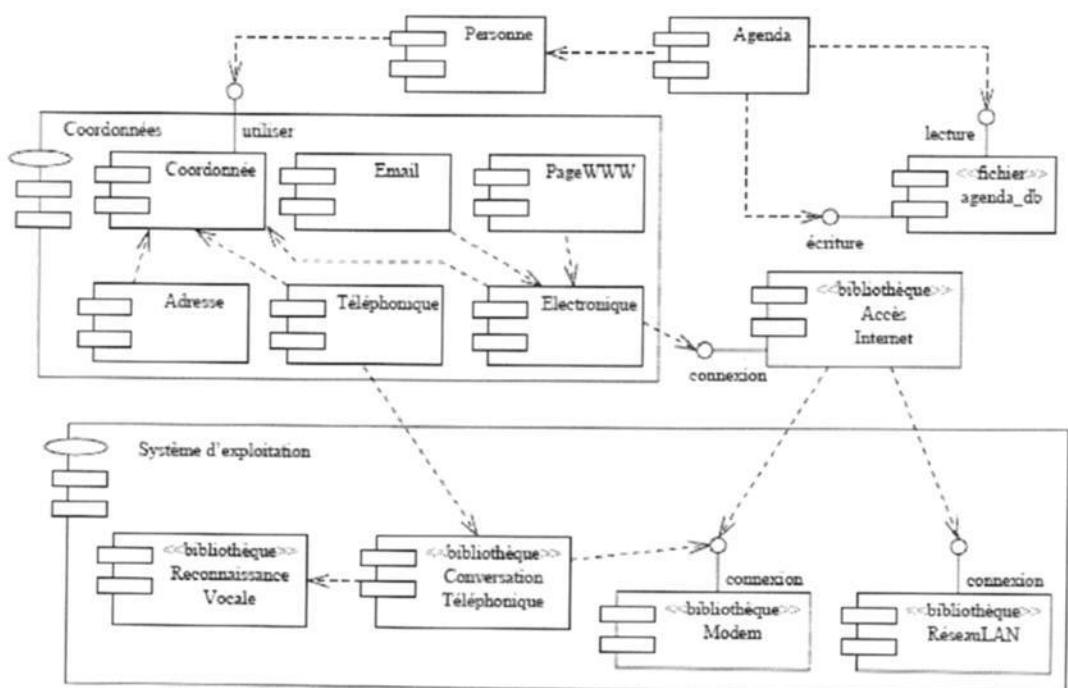
Exemple : diagramme de classes



Diagrammes de composants

17

Exemple : diagramme de composants



Diagrammes de composants

18

Diagrammes de déploiement

Définition

- **Objectifs d'un diagramme de déploiement :**

- ▶ Rendre compte de la disposition physique des différentes ressources matérielles qui entrent dans la composition d'un système ;
- ▶ Rendre compte de la disposition des programmes exécutables et de la répartition des composants sur ces matériels.

- **Ressource = Noeud**

- ▶ Le diagramme de déploiement décrit la répartition des composants sur les noeuds ;
- ▶ Il décrit également les connexions entre les composants ou les noeuds.

- **Deux formes de diagrammes de déploiement : spécification et instance.**

- Un diagramme de déploiement est composé :
 - de dispositifs physiques (les **noeuds**) ;
 - d'objets d'implantation attachés aux noeuds (les **composants**) ;
 - de liens représentant les moyens de communication entre les noeuds (les **supports de communication**).

Noeud

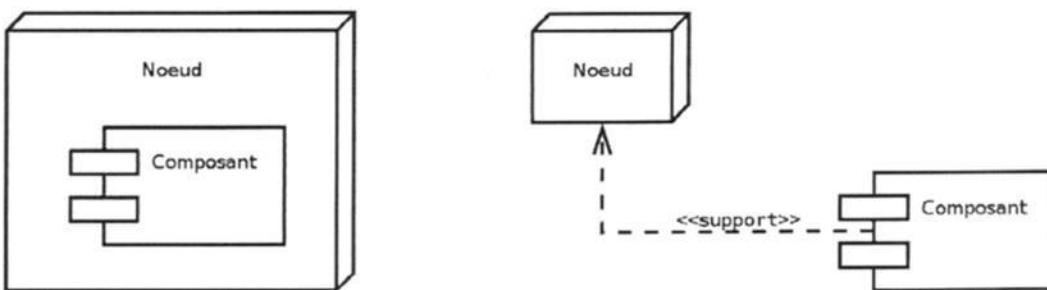
- Chaque ressource est matérialisée par un noeud.
- Représentation graphique : cube comportant un nom.



- Un noeud peut posséder des attributs (mémoire, informations sur le processeur, etc.)

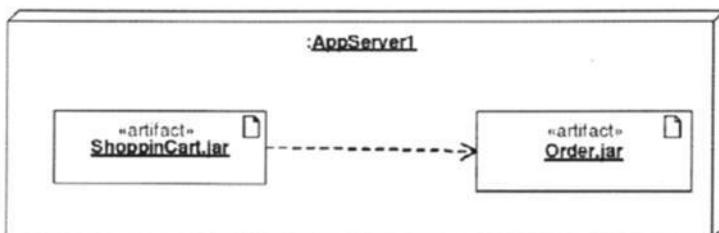
Relation entre composant et noeud

- Deux possibilités pour représenter l'affectation d'un composant à un noeud :
 - Placement du composant à l'intérieur du noeud ;
 - Relation de dépendance avec un stéréotype « support ».



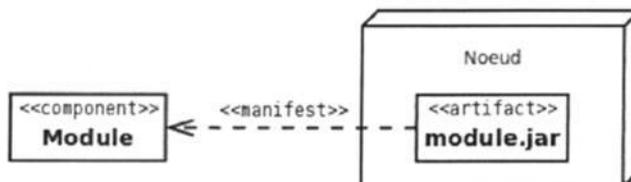
Artefact

- **Définition** : élément concret existant dans le monde réel (fichier, exécutable, script, ect.).
- Représentation graphique : rectangle contenant le mot-clé « artifact » suivi du nom de l'artefact.



Artifact (2)

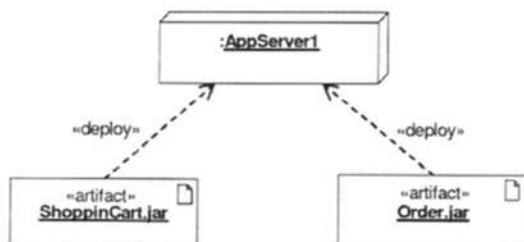
- **Rôle** : l'implémentation des modèles (classes, etc.) se fait sous la forme de jeu d'artefacts.
 - Un artefact peut manifester (= résulter et implémenter) un ensemble d'éléments de modèle.
 - Manifestation = relation entre un élément de modèle et l'artefact qui l'implémente.
 - Représentation graphique : relation de dépendance avec le stéréotype « manifest »



(ici : déploiement dans un noeud d'un artefact manifestant un composant)

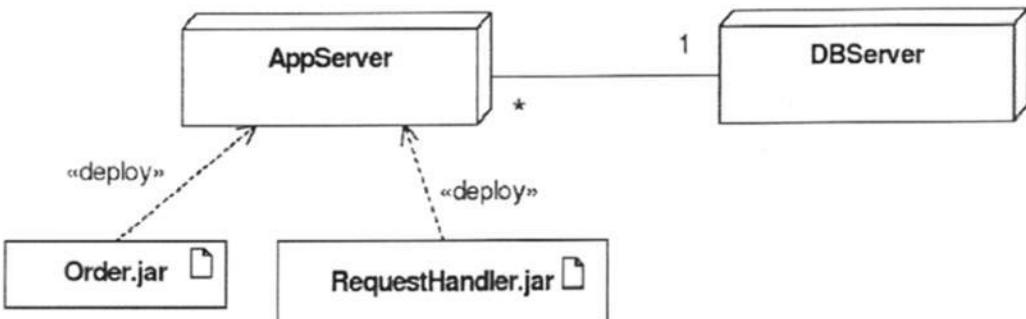
Artifact(3)

- Une instance d'un artefact déploie sur une instance de noeud.
- Représentation graphique : relation de dépendance avec le stéréotype « deploy » pointant vers le noeud.



- L'artefact peut également être directement inclus dans le cube représentant le noeud.
- De manière rigoureuse : seul des artefacts peuvent être déployés sur des noeuds. (Un composant doit être manifesté par un artefact qui peut être déployé sur un noeud).

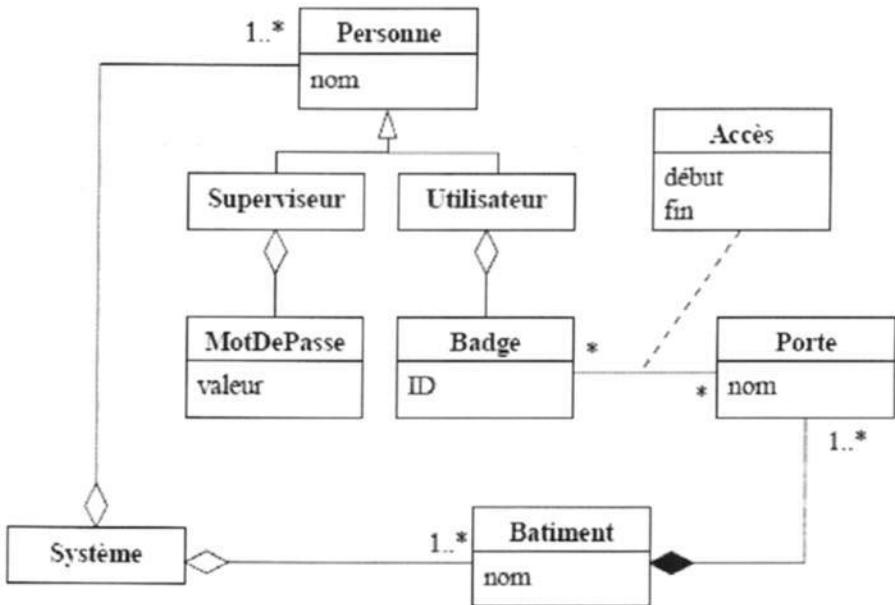
Diagramme de déploiement



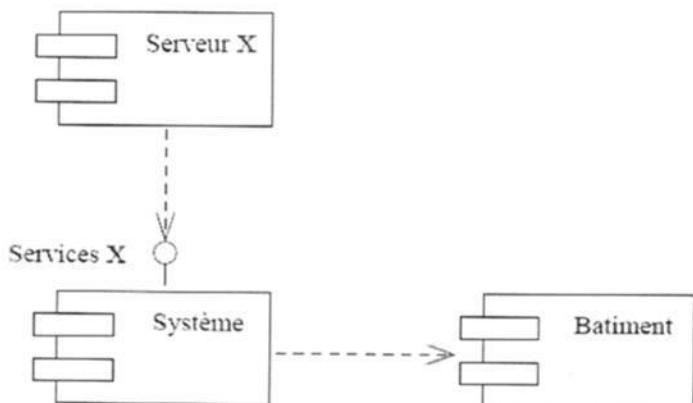
Exemple d'un système de sécurité : point de vue spécification

- Retour sur l'exercice d'un système de sécurité (issu du livre "Instant UML").
- Rappel : système de sécurité limitant les accès à des parties d'un bâtiment à l'aide de cartes magnétiques.

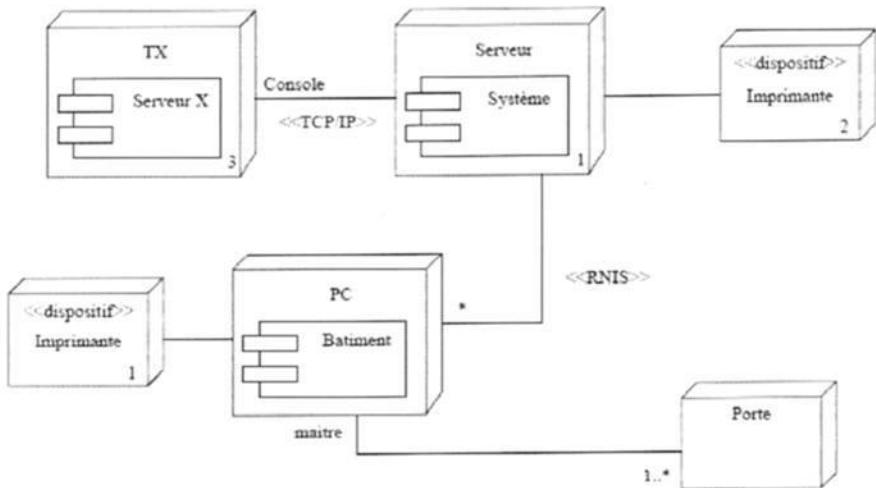
Exemple : diagramme de classes



Exemple : diagramme partiel de composants



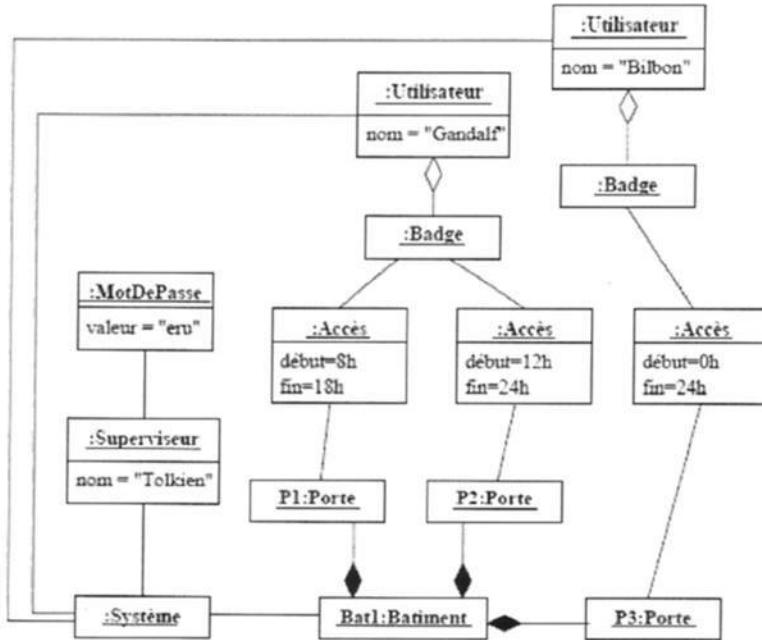
Exemple : diagramme partiel de déploiement



Exemple d'un système de sécurité : point de vue instance

- Le système gère un seul bâtiment contenant trois portes.
- Le système peut être géré par une personne nommée Tolkien.
- Deux utilisateurs peuvent accéder au bâtiment :
 - Gandalf a accès à la première (8h-18h) et seconde porte (12h-14h) ;
 - Bilbon a accès à la troisième porte toute la journée.

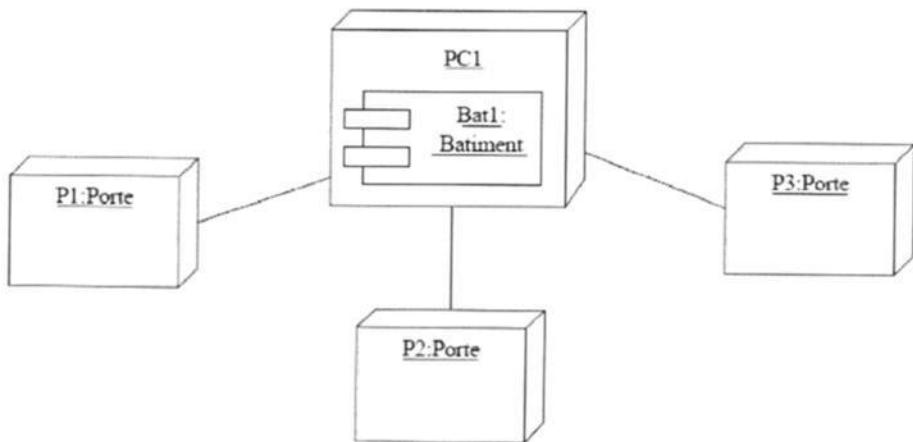
Exemple : diagramme d'objets



Diagrammes de déploiement

33

Exemple : diagramme partiel de déploiement



Diagrammes de déploiement

34

Object Constraint Language

OCL

35

Plan

- ➊ Pourquoi OCL ?
- ➋ Les principaux concepts d'OCL
- ➌ Exemple d'application
- ➍ Utilisation en pratique d'OCL lors d'un développement logiciel

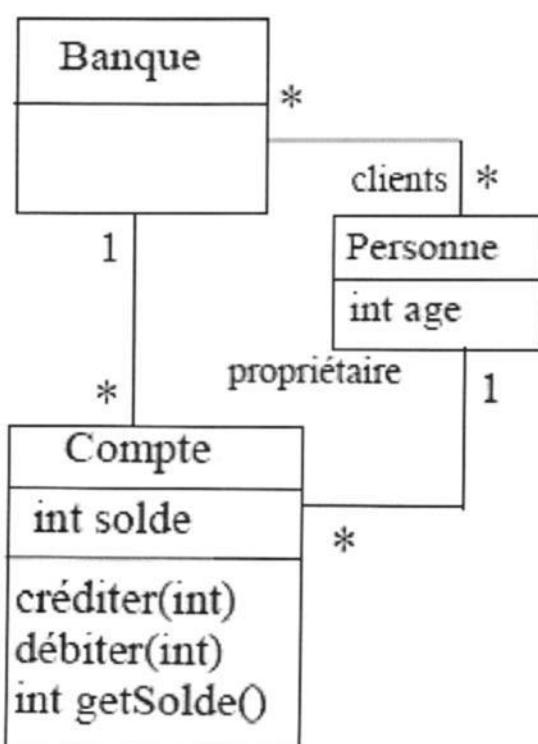
OCL

36

Exemple d'application

- Application bancaire :
 - Elle comporte des comptes, des clients, des banques.
- Spécifications :
 - Un compte doit avoir un solde toujours positif ;
 - Un client peut posséder plusieurs comptes ;
 - Un client peut être client de plusieurs banques ;
 - Un client d'une banque possède au moins un compte dans cette banque ;
 - Une banque gère plusieurs comptes ;
 - Une banque possède plusieurs clients.

Diagramme de classe



Manque de précision dans le diagramme

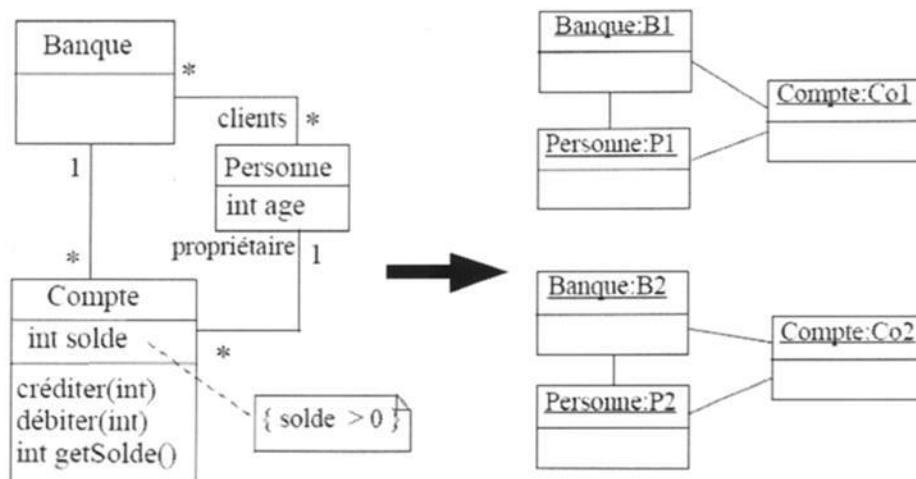
- Le diagramme de classes ne permet pas d'exprimer toutes les spécifications informelles.
 - Exemple : le solde d'un compte doit toujours être positif : ajout d'une contrainte sur cet attribut.
- Toutes les contraintes sur les relations entre les classes ne peuvent pas être détaillées uniquement avec le diagramme de classes.

OCL Pourquoi OCL ?

39

Diagramme d'objets

- Exemple de diagramme d'objets valide vis-à-vis du diagramme de classes et de la spécification attendue :

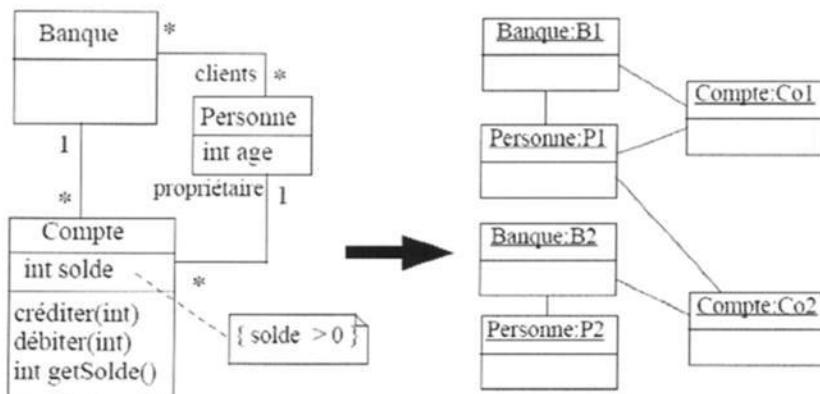


OCL Pourquoi OCL ?

40

Diagramme d'objets

- Exemple de diagramme d'objets valide vis-à-vis du diagramme de classes mais pas de la spécification attendue :



- Une personne a un compte dans une banque où elle n'est pas cliente ;
- Une personne est cliente d'une banque mais sans y avoir de compte.

OCL

Pourquoi OCL ?

41

Expression des contraintes

- Les diagrammes UML sont insuffisants pour spécifier complètement une application : nécessité de rajouter des contraintes.
- Comment exprimer les contraintes ?
 - Langue naturelle peut manquer de précision, voire être ambiguë ;
 - Langage formel avec sémantique précise : par exemple OCL.

OCL

Pourquoi OCL ?

42

- Langage de contraintes orienté objet ;
- Langage formel avec une syntaxe, une grammaire et une sémantique ;
- Langage simple à utiliser, manipulable par un outil ;
- Langage s'appliquant entre autres sur les diagrammes UML ou MOF.

Plan

1 Pourquoi OCL ?

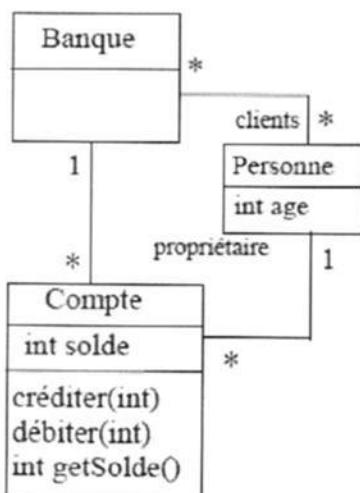
2 Les principaux concepts d'OCL

3 Exemple d'application

4 Utilisation en pratique d'OCL lors d'un développement logiciel

- Version courante : version 2.2
 - Le langage peut s'appliquer sur tout type de modèle, indépendamment d'un langage de modélisation donné (cf. cours MDE).
- OCL permet d'exprimer principalement 2 types de contraintes sur l'état d'un objet ou d'un ensemble d'objets :
 - des invariants qui doivent être respectés en permanence ;
 - des pré et post-conditions pour une opération :
 - pré-condition : doit être vérifiée avant l'exécution ;
 - post-condition : doit être vérifiée après l'exécution.
- Remarque importante : une expression OCL décrit une contrainte à respecter et non le code d'une méthode.

Usage d'OCL sur l'application bancaire



```

context Compte
inv : solde > 0

context Compte : debiter(somme : Integer)
pre : somme > 0
post : solde = solde@pre - somme

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

- Avantage d'OCL : langage formel permettant de préciser clairement de la sémantique sur les modèles UML.
- OCL peut servir à spécifier :
 - ▶ les invariants sur des classes ;
 - ▶ les pré et post-conditions sur des opérations ;
 - ▶ les précautions sur les diagrammes d'états-transitions ou les messages sur les diagrammes de séquences ;
 - ▶ des ensembles d'objets destinataires pour un envoi de message ;
 - ▶ des attributs dérivés ;
 - ▶ des stéréotypes, etc.

Expressions OCL

Les expressions OCL sont construites en 4 parties :

- le **contexte** : définit l'environnement de la contrainte ;
- le(s) **propriété(s)** : représente les caractéristiques du contexte (exemple : si le contexte est une classe, une propriété peut être un attribut)
- l(es) **opération(s)** : manipule ou qualifie la propriété ;
- les **mots-clés** : sont utilisés pour spécifier les expressions conditionnelles (ex : if, then, etc.)

- Une expression OCL est toujours définie dans un contexte.
- Mot-clé : `context`
- Exemple :
 - ▶ `context Compte`
 - ▶ L'expression OCL s'applique à la classe Compte, c'est à dire à toutes les instances de cette classe.
- Un contexte peut porter sur différents types de cibles :
 - ▶ Un type (une classe par exemple : `context Compte`)
 - ▶ Une opération (`context Compte : nom_op(liste_param)`)
 - ▶ Un attribut ou extrémité d'association (`context Compte : nom_attribut : type_attribut`)

Invariants

- Un invariant exprime une contrainte sur un objet ou un groupe d'objets qui doit être respectée en permanence.
- Mot-clé : `inv`
- Exemple :

```
▶ context Compte  
inv : solde > 0
```

- ▶ Pour toutes les instances de la classe Compte, l'attribut solde doit toujours être positif.

- Syntaxe pour une opération :

```
context ma_classe :: mon_op(liste_param) : type_retour
```

- Pour spécifier une opération, on peut utiliser :

- une pré-condition : état qui doit être respecté avant l'appel à l'opération ;
- une post-condition : état qui doit être respecté après l'appel à l'opération ;
- Mots-clés : pre et post

- Les pré et post-conditions servent à décrire les contraintes sur l'état avant et après l'exécution de l'opération ; elles ne décrivent pas comment l'opération est réalisée.

Pré et post-conditions

- Deux éléments particuliers pour la post-condition :

- Attribut result : référence la valeur renvoyée par l'opération
- mon_attribut@pre : référence la valeur de mon_attribut avant l'appel de l'opération.

- Exemples :

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

- La somme à débiter doit être positive pour que l'appel de l'opération soit valide ;
- Après l'exécution de l'opération, l'attribut solde doit avoir pour valeur la différence de sa valeur avant l'appel et de la somme passée en paramètre.

```
context Compte : getSolde() : int
post : result=solde
```

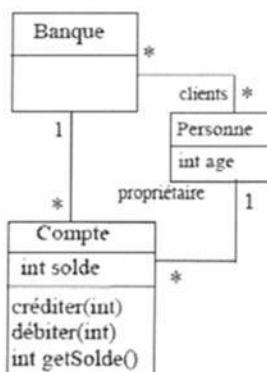
- Le résultat renvoyé doit être le solde courant.

- Dans une contrainte OCL associée à un objet, on peut :
 - ▶ accéder à l'état interne de cet objet (ses attributs) ;
 - ▶ Naviguer dans le diagramme : accéder de manière transitive à tous les objets avec qui il est en relation.
- Accès aux éléments :
 - ▶ Attributs ou paramètres d'une opération : utilisation de leur nom ;
 - ▶ Objet en association : utilisation du nom de la classe associée (en minuscule) ou le nom du rôle d'association du côté de cette classe ;
 - ▶ Association :
 - Cardinalité = 1 : référence 1 objet ;
 - Cardinalité > 1 : référence une collection d'objets.

Navigation dans le modèle

- Exemples : dans le contexte de la classe Compte :
 - ▶ solde : attribut référencé directement ;
 - ▶ banque : objet de la classe Banque associé au compte (référence via le nom de la classe) ;
 - ▶ propriétaire : objet de la classe Personne associée au compte (référence via le nom de rôle d'association) ;
 - ▶ banque.clients : ensemble des clients de la banque associée au compte (référence par transitivité) ;
 - ▶ banque.clients.age : ensemble des âges de tous les clients de la banque associée au compte ;
- Contrainte : le propriétaire doit avoir plus de 18 ans :

```
context Compte
inv : propriétaire.age >= 18
```



```
package Package::Sub::Package

context X
inv : ...

context X::nomOperation()
pre : ...

endpackage
```

Corps d'opération : syntaxe

- body spécifie le résultat d'une opération.

```
context nom_classe::nom_op( liste_param ):type_retour
inv : ...
pre : ...
post : ...
body : ...
```

- Valeur initiale d'un attribut ou d'une association :

```
context nom_classe::nom_attribut : type_attribut  
init ...
```

- Valeur dérivée spécifie la règle de dérivation d'un attribut ou d'une association :

```
context nom_classe::nom_association : type  
derive ...
```

- Exemple :

```
context Personne::argentPoche : Integer  
init : parents.salaire->sum() *1%  
derive : if mineur  
then parents.salaire->sum() *1%  
else job.salaire  
endif
```

Variables

- Définition de variables : peut faciliter l'utilisation de certains attributs ou calculs de valeurs.
- Syntaxe OCL : let...in...

```
▶ context Personne  
inv : let argent=compte.solde->sum() in age>= 18  
implies argent>0
```

- ▶ Une personne majeure doit avoir de l'argent.
- ▶ sum() fait la somme de tous les objets de la collection.

- Pour utiliser la variable partout, syntaxe : def

```
▶ context Personne  
def : argent : int = compte.solde->sum()
```

- Accès à une propriété d'un élément pouvant être :
 - ▶ un attribut ;
 - ▶ un bout d'association ;
 - ▶ une opération ou méthode de type requête ;
 - ▶ une collection ;
- Syntaxe :
 - ▶ accès à la propriété d'un objet avec ":" ;
 - ▶ accès à la propriété d'une collection avec "->".
- Exemple :

```
context Compte
inv : self.solde > 0
```

Types OCL : types de base

- Integer :
 - ▶ Opérations associées : *, +, -, /, abs()
- Real :
 - ▶ Opérations associées : *, +, -, /, floor()
- String :
 - ▶ Opérations associées : concat(), size(), substring()
- Boolean :
 - ▶ Opérations associées : and, or, xor, not, implies, if-then-else
 - ▶ La plupart des expressions OCL sont de type Boolean (notamment les expressions pour les invariants, les pré et post-conditions)

- Types de collection d'objets :
 - **Set** : ensemble au sens mathématique, pas de doublons, pas d'ordre.
 - Exemple : {1,4,3,5}
 - **OrderedSet** : idem mais avec ordre
 - **Bag** : comme Set pas avec possibilité de doublons
 - Exemple : {1,4,1,3,5,4}
 - **Sequence** : Bag dont les éléments sont ordonnés
 - Exemple : {1,1,3,4,4,5}
- Possibilité de transformer un type de collection en un autre type de collection avec opérations OCL dédiées.

Types OCL : collections imbriquées

- En naviguant dans le modèle, on peut récupérer des collections ayant pour éléments d'autres collections.
- 2 modes de manipulation :
 - Explicitement comme une collection de collections de ...
 - Collection unique : on aplatis le contenu de toutes les collections imbriquées en un seul niveau :
 - Opération `flatten()`
- Tuples/n-uplet :
 - Données contenant plusieurs champs :
 - Exemple : tuple {nom :String ='toto', age :Integer = 21}
 - Possibilité de manipuler ce type de données en OCL.

- Conformité de types :

- ▶ Prise en compte des spécialisations entre classes du modèle ;
- ▶ Opérations OCL dédiées à la gestion des types :
 - `oclIsTypeOf(type)` : vrai si l'objet est du type `type`;
 - `oclIsKindOf(type)` : vrai si l'objet est du type `type` ou un de ses sous-types;
 - `oclAsType(type)` : cast de l'objet en type `type`.

- Types internes à OCL :

- ▶ Conformité entre les types de collection :
 - Collection est le super-type de Set, Bag et Sequence ;
 - Conformité entre collection et types des objets contenus : `Set(T1)` est conforme à `Collection(T2)` si `T1` est sous-type de `T2` ;
- ▶ Integer est un sous-type de Real.

Opérations sur objets et collections

- OCL propose des primitives utilisables sur les collections :

- ▶ Syntaxe : `objetOuCollection -> primitive`

- Primitives OCL :

- ▶ `size()` : retourne le nombre d'éléments de la collection ;
- ▶ `isEmpty()` : retourne vrai si la collection est vide ;
- ▶ `notEmpty()` : retourne vrai si la collection n'est pas vide ;
- ▶ `includes(obj)` : vrai si la collection inclut l'objet `obj` ;
- ▶ `excludes (obj)` : vrai si la collection n'inclut pas l'objet `obj` ;
- ▶ `including(obj)` : la collection référencée doit être cette collection en incluant l'objet `obj` ;
- ▶ `excluding(obj)` : idem en excluant l'objet `obj` ;
- ▶ `includesAll(ens)` : la collection contient tous les éléments de la collection `ens` ;
- ▶ `excludesAll(ens)` : la collection ne contient aucun des éléments de la collection `ens` ;

- `self` : pseudo-attribut référençant l'objet courant.

- Invariants dans le contexte de la classe Compte :

- ▶ `proprietaire->notEmpty()` : il y a au moins un objet Personne associé à un compte ;
- ▶ `proprietaire->size() = 1` : le nombre d'objets Personne associés à un compte est de 1 ;
- ▶ `banque.clients->size() >=1` : une banque a au moins un client ;
- ▶ `banque.clients->includes(proprietaire)` : l'ensemble des clients de la banque associée au compte contient le propriétaire du compte ;
- ▶ `banque.clients.compte->includes(self)` : le compte appartient à un des clients de sa banque.

Opérations sur objets et collections

- `oclIsNew()` : primitive indiquant qu'un objet doit être créé pendant l'appel de l'opération (à utiliser dans une post-condition) ;
- `and` :
 - ▶ permet de définir plusieurs contraintes pour un invariant, une pré ou post-condition ;
 - ▶ équivalent au "et logique" : vrai si toutes les expressions reliées sont vraies.
- Exemple :

```
context Banque:::creerCompte(p: Personne): Compte
post : result.oclIsNew() and
compte = compte@pre->including(result) and
p.compte = p.compte@pre->including(result)
```

Un nouveau compte est créé. La banque doit gérer ce nouveau compte. Le client passé en paramètre doit posséder ce compte. Le nouveau compte est retourné par l'opération.

- `union` : retourne l'union de 2 collections ;
- `intersection` : retourne l'intersection de 2 collections ;
- Exemples :
 - ▶ `(col1->intersection(col2))->isEmpty()` : vrai si les collections `col1` et `col2` n'ont pas d'élément en commun.
 - ▶ `col1 = col2->union(col3)` : la collection `col1` doit être l'union des éléments de `col2` et `col3`.

Opérations sur les éléments d'une collection

- OCL permet de vérifier des contraintes sur chaque élément d'une collection ou de définir une sous-collection à partir d'une collection en fonction de certaines contraintes.
- Primitives OCL :
 - ▶ `select/reject`
 - ▶ `collect`
 - ▶ `exists`
 - ▶ `forAll`
 - ▶ `iterate`

- Trois usages des opérations (primitives appliquées sur la collection *col*) :
 - ▶ `col->primitive(expr)` : primitive appliquée aux éléments de la collection. Pour chacun d'eux, l'expression est vérifiée. On accède aux attributs/relations d'un élément directement.
 - ▶ `col->primitive(elem : type|expr)` : le type des éléments de la collection est explicité. On accède aux attributs/relations de l'élément courant en utilisant *elem*.
 - ▶ `col->primitive(elem|expr)` : l'attribut courant *elem* est nommé mais sans préciser son type.

Opérateurs select et reject

- **Objectif** : retourne le sous-ensemble de la collection dont les éléments (ne) respectent (pas) la contrainte spécifiée.
- **Syntaxe** :
 - ▶ `col->select(expr)`
 - ▶ `col->select(elem : type|expr)`
 - ▶ `col->select(elem|expr)`
- Exemple : dans le contexte de la classe Banque :
 - ▶ `compte->select(c | c.solde > 1000)` : retourne une collection contenant tous les comptes bancaires dont le solde est supérieur à 1000 euros.
 - ▶ `compte->select(solde > 1000)` : retourne une collection contenant tous les comptes bancaires dont le solde n'est pas supérieur à 1000 euros.

Opérateur collect

- **Objectif** : retourne une collection (de taille identique) construite à partir des éléments de la collection initiale. Le type des éléments contenus dans la nouvelle collection peut être différent de celui de la collection initiale.
 - ▶ Remarque : collect renvoie toujours un Bag.
- **Syntaxe :**
 - ▶ col->collect(expr)
 - ▶ col->collect(elem : type|expr)
 - ▶ col->collect(elem|expr)
- Exemple : dans le contexte de la classe Banque :
 - ▶ compte->collect(c : Compte | c.solde) : retourne une collection contenant l'ensemble des soldes de tous les comptes.
 - ▶ (compte->select(solde>1000))->collect(c |c.solde) : retourne une collection contenant tous les soldes des comptes dont le solde est supérieur à 1000 euros.
- Remarque : forme raccourcie et simplifiée de compte->collect(solde) est compte.solde qui renvoie l'ensemble des soldes de tous les comptes (syntaxe pour l'accès aux attributs pour les collections).

Opérateur exists

- **Objectif** : retourne vrai si au moins un élément de la collection respecte la contrainte spécifiée et faux sinon.
- **Syntaxe :**
 - ▶ col->exists(expr)
 - ▶ col->exists(elem : type|expr)
 - ▶ col->exists(elem|expr)
- Exemple : dans le contexte de la classe Banque :
 - ▶ **context** Banque
inv : **not**(clients->exists(age<18))

Il n'existe pas de clients de la banque dont l'âge est inférieur à 18 ans.
not prend la négation d'une expression.

- **Objectif** : retourne vrai tous les éléments de la collection respectent la contrainte spécifiée (pouvant impliquer à la fois plusieurs éléments de la collection).
- **Syntaxe** :
 - `col->forAll(expr)`
 - `col->forAll(elem : type|expr)`
 - `col->forAll(elem|expr)`
- Exemple : dans le contexte de la classe Personne :

```
context Personne
inv : Personne.allInstances()->forAll(p1, p2 | p1<math>\diamondsuit</math>p2
implies p1.nom <math>\diamondsuit</math> p2.nom
```

- Il n'existe pas deux instances de la classe Personne pour lesquelles l'attribut nom a la même valeur : deux personnes différentes ont un nom différent.
- Primitive `allInstances()` : s'applique à une classe (pas à un objet) et retourne toutes les instances de la classe référencée.

- **Objectif** : forme générale d'une itération sur une collection et permet de redéfinir les opérateurs vus précédemment.
- **Syntaxe** :
 - `col->iterate(elem : type; reponse : type
=<valeur> | <expression_avec_elem_et_reponse>)`
- Exemple : dans le contexte de la classe Banque :

```
context Banque
def : moyenneDesComptes : Real = compte->collect(c :
  Compte | c.solde)->sum()/compte->size()
-- est identique à :
context Compte
def : moyenneDesComptes : Real =
compte->iterate(c : Compte; lesComptes : Bag{Integer})
  = Bag{} | lesComptes->including(c.solde)->sum()
  /compte->size()
```

- Certaines contraintes sont dépendantes d'autres contraintes :

► **if expr1 then expr2 else expr3 endif**

- Si l'expression *expr1* est vraie alors *expr2* doit être vraie sinon *expr3* doit être vraie ;
- La clause *else* est nécessaire et les deux expressions *expr2* et *expr3* doivent être du même type.

expr1 implies expr2

-
- Si l'expression *expr1* est vraie, alors *expr2* doit être vraie.
 - Si *expr1* est fausse, alors l'expression complète est vraie.

Conditionnelles : exemple

• **context Personne**
inv : if age<18
then compte->isEmpty()
else compte->notEmpty()
endif

- Une personne de moins de 18 ans n'a pas de compte bancaire alors qu'une personne de plus de 18 ans possède au moins un compte.

context Personne
inv : compte->notEmpty() implies banque->notEmpty()

- ► Si une personne possède au moins un compte bancaire, alors elle est cliente d'au moins une banque.

- Commentaire en OCL : utilisation de --
- On peut également nommer des contraintes :
Exemple :

```
► context Compte  
inv : soldePositif : solde>0
```

```
context Compte::debiter(somme : Integer)  
pre : sommePositive:somme>0  
post : sommeDebitee:solde=solde@pre-somme
```

Appels d'opération des classes

- Possibilité d'accéder aux attributs, objets en mode "lecture" ;
- Possibilité d'utiliser une opération d'une classe dans une contrainte : attention aux effets de bord !
- Exemple :

```
► context Banque  
inv : compte->forAll(c | c.getSolde()> 0)
```

► `getSolde()` est une opération de la classe `Compte`. Elle calcule une valeur mais sans modifier l'état d'un compte.

- @pre
- . et ->
- not et -
- * et /
- + et -
- if then else endif
- >, <, <= et >=
- = et <>
- and, or et xor
- implies

Remarque : les parenthèses permettent de changer cet ordre.

Plan

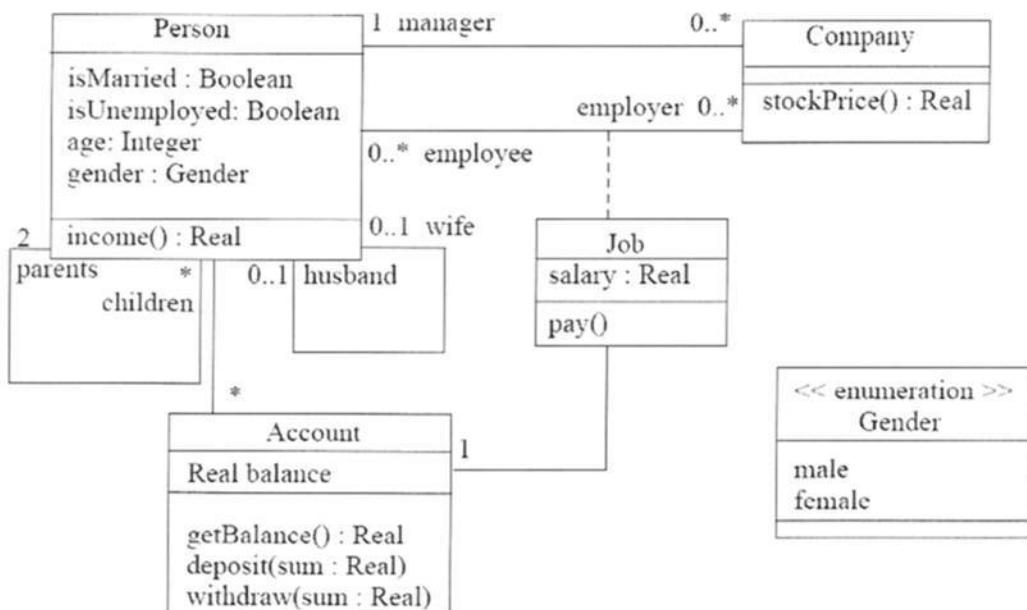
1 Pourquoi OCL ?

2 Les principaux concepts d'OCL

3 Exemple d'application

4 Utilisation en pratique d'OCL lors d'un développement logiciel

Diagramme de classes



(d'après les normes OCL)

OCL

Exemple d'application

81

Contraintes sur les employés d'une compagnie

- Dans une compagnie, un manager doit travailler et avoir plus de 40 ans ;
- Le nombre d'employé d'une compagnie est non nul ;

OCL

Exemple d'application

82

Lien salaire/chômage pour une personne

- Une personne considérée comme au chômage ne doit pas avoir des revenus supérieurs à 500 euros.

Contraintes sur les parents/enfants

- Un enfant a un père et une mère.

Contraintes sur les parents/enfants

- Tous les enfants d'une personne ont bien cette personne comme parent et inversement.

Contraintes de mariage

- Pour être marié, il faut avoir plus de 18 ans.
- Un homme est marié avec une femme et une femme avec un homme.

- Un employé nouvellement embauché n'appartenait pas déjà à la compagnie.

Revenus selon l'âge

- Selon l'âge de la personne, ses revenus sont :
 - 1% des revenus des parents quand elle est mineure (argent de poche) ;
 - ses salaires quand elle est majeure.

- Versement du salaire :

```
context Job :: pay()  
post : account.balance = account.balance@pre + salary
```

- Depuis OCL 2.0, on peut aussi préciser que l'opération deposit doit être appelée :

- ▶ Syntaxe : objet[^]operation(param1,...) : renvoie vrai si un message operation est envoyé à objet avec la liste de paramètres précisée.

- ▶

```
context Job :: pay()  
post : account^deposit(salary)
```

Plan

1 Pourquoi OCL ?

2 Les principaux concepts d'OCL

3 Exemple d'application

4 Utilisation en pratique d'OCL lors d'un développement logiciel