COO : Spécification du logiciel - OCL

Philippe Collet

Licence 3 – parcours Informatique et MIAGE

Septembre - Décembre 2012

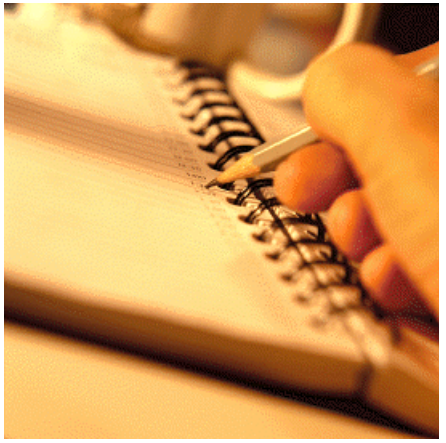
Université Nice Sophia Antipolis

Objectifs

- Comprendre l'activité de spécification logicielle
- Apprendre la spécification par assertions
- Comprendre la nécessité et la portée d'OCL
- Apprendre OCL et la manière de spécifier avec ce langage dans UML
- Comprendre les techniques de vérification des assertions

Plan

- Introduction aux spécifications
- Introduction à la spécification par assertions
- OCL : première approche
- Types et valeurs de base
- Navigation dans les modèles
- Autres éléments du langage
- Collections



- Conception par contrats
- Annexe OCL

Introduction aux spécifications

- Spécification fonctionnelle : définition
- Principaux formalismes de spécification
- Notions autour de la spécification
- Spécification en langage naturel
- Tour rapide d'autres techniques de spécifications « formelles »

Spécification fonctionnelle

- **Définition :**
  - définir ce que doit faire un logiciel ...
  - avant d'écrire ce logiciel !
  - => *le quoi, sans le comment !*
- S'oppose à **implémentation**,
  - comme le plan d'un pont à sa construction.
  - Sauf... qu'en informatique, plan et réalisation ne manipulent que de l'écriture...

Pourquoi spécifier ?

- **Définir** le travail de réalisation, **avant** de le faire ...
  - la spécification est censée être moins coûteuse, plus rapide à obtenir
- Comment **vérifier** la correction d'un programme,
  - si l'on ne sait pas ce qu'il est censé faire ...
  - la spécification est la référence pour toutes les activités de vérification et de validation (V&V) :
    - *tests, preuves, mesures, inspections...*
- Obtenir une description **stable**, plus **abstraite**, moins dépendante des contingences du matériel, des systèmes, des fluctuations de l'environnement.

Qualités recherchées pour une spécification fonctionnelle

- précision, non ambiguïté, non contradiction,
- concision, abstraction,
- complétude,
- facilité d'utilisation : écriture, lecture, vérification
- réalisable avant l'implémentation,
- si possible à un coût réduit,
- référence contractualisable, pour les litiges...

Spécifications fonctionnelles vs extrafonctionnelles

- **En génie logiciel, on distingue :**
  - spécification fonctionnelle : décrit le « fonctionnement » du logiciel, le quoi
  - spécification extrafonctionnelle (ou non fonctionnelle) : les conditions de fonctionnement, le comment

- **Dans le monde industriel, *spécification* sous-entend, *spécifications techniques***

- **En génie logiciel, par défaut *spécification* sous-entend *spécification fonctionnelle***

- **Les *spécifications techniques du logiciel* : coût, temps de réponse, performance, robustesse, capacité de charge, consommation de ressource, confort d'utilisation, ergonomie ... sont appelées :**
  - *qualités de service (QoS)*
  - *spécifications non fonctionnelles*
  - *spécifications extrafonctionnelles*

Principaux formalismes de spécification

Texte informel

- En langage naturel (cahier des charges, commentaires)
- éventuellement encadré par une méthode ou un formalisme graphique (contraintes UML)

Graphique

- sauf exception (réseaux de Petri...) rarement très formel : (SADT, UML)
- pas très précis, mais utile pour la synthèse et en complément

Semi-formel

- Sans ambiguïté, leur expressivité est insuffisante pour établir une preuve, mais on peut souvent les tester :
  - techniques assertionnelles, spécification axiomatique

Formel

- Sans ambiguïté, leur expressivité est suffisante pour tout décrire, donc pour établir des preuves, humaines (« à la main ») ou automatiques

Spécification vs Implémentation

Spécification

- décrit ce que doit faire le logiciel, si possible très tôt dans le processus logiciel, à un coût faible, indépendamment des « détails » :
  - *type de machine, plate-forme, langage utilisé, conditions d'utilisation, fréquences des primitives, représentation, formats...*
- Une spécification fonctionnelle doit se concentrer sur les fonctionnalités, sans considération de performance ou de qualités de services (spécifications extrafonctionnelles)

Implémentation

- doit correspondre aux fonctionnalités décrites (idée de correction),
- mais de manière efficace, performante, en tenant compte:
  - des conditions réelles ou prévues d'utilisation : *langage, spécificité du langage utilisé, fréquence d'emploi des primitives fournies, volumes des données traitées...*
- des contraintes exprimées dans les spécifications extrafonctionnelles

Spécification vs Implémentation (suite)

- **En simplifiant,**
- ***spécifier, c'est définir***
  - *sans se soucier des contingences du monde réel...*
- ***implémenter, c'est optimiser, réifier***
  - *c'est tenir compte de la réalité...*
- **Si l'implémentation nécessite beaucoup de détails, l'écart de niveaux d'abstraction avec la spécification est grand, et cela nécessite des descriptions intermédiaires :**
  - *approches par raffinements successifs*

Spécification vs Implémentation (suite)

- **Relations d'abstraction et de raffinement**
  - Considérons une suite de n descriptions du même problème, à différents niveaux d'abstraction, numérotées par niveau d'abstraction décroissant :
    - desc1 ⊆ desc2... ⊆ descn
  - la desc1, de plus haut niveau d'abstraction,
    - est la spécification,
  - la descn, de plus bas niveau d'abstraction,
    - est l'implémentation

- **Toutes les descriptions sont liées par deux relations :**
  - abstraction :
    - desci = abstract(desci+1) ⇔ desci ⊆ desci+1
  - raffinement :
    - desci = refine(desci−1) ⇔ desci ⊇ desci−1

- **Il existe une théorie qui formalise ces relations : *refinement calculus***

Spécification, Formalité et Exécutabilité

- **Deux sens au mot *formel* :**
  1. *formel* = non ambigu, précis, sans équivoque
     - syntaxe et sémantique précise, interprétable par une machine
  2. *formel* = privilégie la forme au détriment du contenu
     - non ambigu et vérifiable, prouvable à la main ou automatiquement.

- **Exemple:** $(a + b)^2 = a^2 + 2ab + b^2$

- ***formel* n'implique donc pas abstrait (sens 1), mais on confond souvent les deux mots (sens 2)**

- **Exemple :**
  - logique formelle, avec des formules, par opposition à logique philosophique, avec (beaucoup) de mots... (sens 2)
  - spécification (formelle) et implémentation sont formelles ! (sens 1)
    - ... puisque sans ambiguïté !!!

Spécification, Formalité et Exécutabilité (suite)

- **Mais**
  - un programme dans un langage de bas niveau se prête difficilement à des preuves, c'est-à-dire à l'établissement de propositions vraies pour toutes exécutions :
  - *généralement, on teste un programme exécutable*

- **Inversement,**
  - une preuve théorique de spécification ne prouve pas que le logiciel décrit sera utilisable :
  - *temps de réponse ? ergonomie ? adéquation aux besoins ?*

- **La frontière entre spécification et implémentation est assez floue et arbitraire et dépend des époques :**
  - *il y a quelques années, un texte en Prolog aurait été considéré comme une spécification abstraite exécutable…*

Spécification fonctionnelle et correction

❒ Qualité de correction

- un logiciel est correct, si dans des conditions normales d'utilisation, il se comporte comme cela est attendu par ses utilisateurs
- Si la spécification traduit bien ce que veulent les utilisateurs, alors la correction revient à vérifier (prouver, tester) que l'implémentation est conforme à sa spécification

❒ Conséquences :

- Pour vérifier la correction d'une description, il faut une deuxième description : la redondance est indispensable à tout contrôle : *preuve par neuf, contrôle de qualité, vérification des cartes perforées (autrefois)...*
- Comment vérifier la correction d'une spécification ?
  - l'un des gros problèmes des spécifications formelles, lorsqu'elles sont incompréhensibles aux commanditaires ou aux utilisateurs

Utilisation des spécifications fonctionnelles

- **définition du travail d'implémentation,**
  - à un coût normalement inférieur : l'équivalent des plans d'un immeuble, d'une machine, avant sa réalisation.
  - *(sauf qu'ici, spécification et implémentation, c'est de l'écriture !)*

- **référence pour tout ce qui concerne les fonctionnalités du logiciel :**
  - les documentations d'utilisation, de maintenance, les tests externes, les preuves de correction...

Spécification en langage naturel

- **Description en langage naturel (anglais, français...) :**
  - de manière complètement libre, littéraire...
  - de manière très encadrée (structurée) par une méthode qui fournit un plan précis de ce qu'il faut décrire
  - Exemples : normes de cahiers des charges (ISO 900x, IEEE 930), outils d'aide à la documentation pour certains langages de programmation (Java, Eiffel) ou certains systèmes (Unix, Emacs...)

- **Utilisation d'une syntaxe pour les aspects structurels, les signatures de méthodes...**
  - importance de la phase de conception préalable

- **Utilisation de glossaires ou de dictionnaires de données (identificateurs...)**
  - pour éviter d'utiliser des mots voisins ou synonymes : utiliser au contraire un seul mot pour chaque concept

- **Règles de description et d'abréviations**
  - Exemple : un prédicat rend une valeur logique vraie ou faux, donc il suffit de décrire le cas vrai

Exemple d'une (mauvaise) spécification en langage naturel

❒ **Ce module gère une file d'attente de transactions selon l'ordre premier arrivé - premier servi (FIFO)...**

❒ **La queue a au plus 500 transactions et est manipulée par les opérations suivantes :**

- déposer une transaction dans la file, à la fin, si c'est possible, sinon envoyer le message d'erreur "déposer impossible".
- retirer la première transaction de la file (la plus ancienne, en tête), si elle existe, sinon donner le message d'erreur "retirer impossible".
- connaître à tout instant la longueur de la file.

Avantages de la spécification en langage naturel

- (en principe) utilisable et compréhensible par tous, en particulier le commanditaire

- expressivité non limitée : tout concept peut se décrire en langage naturel, mathématique, philosophique, esthétique...

- concision possible, par réutilisation de connaissances antérieures des lecteurs : *il n'est pas toujours nécessaire de tout dire!*

- toujours utile, doivent toujours accompagner les spécifications les plus formelles ou les textes des programmes (commentaire d'en-tête, description résumée)

Inconvénients de la spécification en langage naturel

- **non compréhensibles par des machines ...**
  - et parfois par des humains !

- **défauts fréquents : *ambiguïté, silence, repentir, contradiction, sur-spécification, sous-spécification, changement de terminologie, description verbeuse, répétition, lourdeur de style, charabia, bruit, franglais, ...***

- **nécessité de relectures, qui peuvent être très coûteuses pour un résultat de qualité**

- **utilité d'une description formelle pour rectifier une description informelle**

Quelques défauts de l'exemple de la file

- Ce module gère une *file* **d'attente de transactions** selon l'ordre premier *arrivé* - premier *servi* (FIFO)

- La *queue* a au plus **500** transactions et est manipulée par les opérations suivantes :
  - *déposer* une transaction dans la file, à la fin, si c'est possible, sinon envoyer le **message d'erreur "déposer impossible"**
  - *retirer* la première transaction de la file (la plus ancienne, en tête), si elle existe, sinon donner le **message d'erreur "retirer impossible"**
  - permettre de connaître **à tout instant** la longueur de la file

- Légende :
  - lourdeur de style, repentir, verbiage
  - **sur-spécification**
  - *changement de terminologie*

Spécification structurée en langage naturel

- La structure de l'objet informatique (sa syntaxe) est donnée dans un formalisme précis (UML, langage de classe...).

- La sémantique est donnée en langage naturel, mais avec des conventions d'abréviations et en utilisant un dictionnaire de données (les identificateurs),

- On évite des répétitions par les possibilités de factorisation de propriétés ou d'opérations données : **héritage multiple, généricité ...**

- Les paramétrages (par généricité contrainte si possible) et les signatures précisent de manière claire les propriétés.

- Le cadre syntaxique permet l'usage d'outils : *hypertexte, présentation normalisée, dictionnaire / glossaire des identificateurs...*

Exemple en langage naturel structuré

```
classe File < Element : Object >
  -- file bornée, protocole FIFO, deux extrémités tête et queue.
héritage
  Conteneur < Element >, StructureBornée
opérations
File (n : Naturel)
  -- création d'une file vide d'au plus n éléments
  -- hérité de StructureBornée
tête : Element
  -- prochain élément à retirer, si longueur > 0
queue : Element
  -- dernier élément déposé, si longueur > 0
déposer (e: Element)
  -- dépose e à la queue de self, si longueur < n
retirer
  -- retire l'élément situé en tête
longueur : Naturel
  -- nb d'éléments (hérité de Conteneur)
vide : Booleen
  -- longueur > 0 (hérité de Conteneur)
```

Autres techniques « formelles »

- **Techniques algébriques :**
  - *Clear, ASL, ACT1, Larch, OBj2, LPG, Pluss, Affirm, Reve, Asspegique...*
- **Techniques orientées modèle abstrait :**
  - opérationnelle : langages de haut niveau : *CLU, Euclide, CAML, Prolog...*
  - axiomatiques : *Z, VDM, B*
  - model-checking: logique temporelle, TLA, SMV, Spin, Kronos, etc
- **Techniques orientées lambda calcul :**
  - *théories des types*
- **Techniques orientées logique ordre supérieur, treillis :**
  - *refinement calculus...*,
- **Techniques orientées processus :**
  - *réseaux de Petri, CSP, CCS, arbres JSD...*

Exemple : spécification algébrique

- Une technique très formelle et très abstraite, développée dans les années 1970 : *chaque objet informatique est considéré comme une classe d'algèbre.*

- La **syntaxe** des primitives est donnée par une **signature**, de manière fonctionnelle

- La **sémantique** est donnée par des axiomes (équations ou prédicats) qui combinent les primitives de manière à obtenir des propositions vraies *pour tout objet du type défini*

- Le temps est absent : *les objets ne changent pas d'état, mais toutes les primitives sont des fonctions qui rendent une valeur, éventuellement un nouvel état de l'objet considéré*

- Une spécification algébrique est comme un dictionnaire, par exemple du chinois écrit en chinois : *chaque mot est défini par une combinaison de signes, qui sont eux-même définis dans une entrée du dictionnaire.*

Spécification algébrique : primitives

- **On distingue trois sortes de primitives :**
  - les **générateurs** qui permettent de construire tous les états possibles des objets du type
  - les **accesseurs** qui renseignent sur les objets du type, sans les modidier,
  - les **modifieurs** qui rendent un nouvel état de l'objet

- **Les axiomes expriment toutes les propriétés pertinentes des accesseurs et des modifieurs pour tous les états possibles des objets**
  - Ces états sont décrits par les générateurs, par induction structurale

- **Les cas d'erreurs correspondent à des domaines de définition de fonctions partielles (équivalent des préconditions des langages d'assertions)**

Spec. Algébrique d'une file infinie

type File <E>
primitives
-- *générateurs*
créer : −> File <E>
déposer : File <E> × E −> File <E>

-- *accesseurs*
vide : File <E> −> Booléen
tête : File <E> /−> E

-- *modifieurs*
retirer : File <E> /−> File <E>

préconditions
∀ f: File <E>
(p1) retirer(f) **requiert** ¬ vide (f)
(p2) tête(f) **requiert** ¬ vide (f)

Spec. Algébrique d'une file infinie (suite)

axiomes

∀ e: E, f: File <E>

(a1) vide(créer())

(a2) ¬ vide(déposer(f,e))

(a3) tête(déposer(créer(),e)) = e

(a4) tête(déposer(f,e)) = tête(f)

(a5) retirer(déposer(créer(),e)) = créer()

(a6) retirer(déposer(f,e)) = déposer(retirer(f),e)

Spécification algébrique : bilan de la spec.

Avantages

- concise, précise, élégante, abstraite...
- autonome, ne dépend d'aucune autre spécification,
- exhibe les primitives usuelles de manipulation

Inconvénients

- n'explicite pas deux propriétés cachées incontournables des files : *nombre d'éléments*, *accès à la valeur de chaque élément*
  - mais on peut le faire...
- ne fait pas apparaître de relation d'héritage
- pas de commentaires en langage naturel :
  - on doit le faire (absents sur transparents)
- séparation des informations qui marchent ensemble :
  - syntaxe, sémantique, commentaire...

Bilan global sur les spécifications algébriques

- **Comme en programmation, il y a plusieurs manières possibles de spécifier le même problème...**

- **La spécification algébrique est une technique orientée programmation fonctionnelle, peu adaptée au monde objet**
  - L'héritage ne marche pas bien pour les axiomes

- **Possibilité de généricité et d'importation d'autres spécifications comme dans la programmation modulaire classique (Java 5)**

- **Exigence de définition complète : *forme un tout, pas de définition incrémentale de la conception d'un logiciel***

- **Outils d'aide : *vérification syntaxique, complétude, absence de contradiction, prouveurs de théorème***
  - mais il faut les aider, et plutôt lents...

- **Conformité du code ? *preuves de théorèmes* et une adaptation nécessaire aux différents langages**

- **Accessibilité aux programmeurs ?**
  - *difficulté de construction et de compréhension par la majorité des programmeurs, et en cas de logiciel nécessitant des milliers d'axiomes*

Bilan : Avantages des techniques formelles

- **Formalité :**
  - propriétés établies par raisonnement formel : argumentation stricte, non intuitive
  - contradictions et incomplétudes révélées avant l'implémentation
  - outils de preuve calculent et vérifient (semi-) automatiquement,
- **Précision :**
  - meilleure compréhension
  - définition de référence pour l'implémenteur
- **Abstraction :**
  - concision
  - description plus stable, indépendante des techniques d'implémentation, plus réutilisable

Bilan : inconvénients des techniques formelles

- **Difficultés :**
  - manque de lisibilité
  - aptitudes mathématiques nécessaires
  - erreurs possibles, si preuves humaines
- **Manque de maturité :**
  - nombreux formalismes, même si quelques standards existent
  - outils de preuves insuffisants et malaisés
  - manque de structuration, héritage absent, généricité difficile
- **Domaines d'application restreints**
  - généralement inadéquates pour d'autres domaines
- **Les spécifications formelles sont quand même indispensables pour les logiciels où la fiabilité est critique…**

Introduction à la spécification par assertions

- Historique
- Principes
- Accesseurs et modifieurs
- Illustration
- Invariant

Spécification par assertions

- Introduites par Floyd (1967) et Hoare (1969), les assertions servaient au début pour **annoter** les programmes en vue d'en établir la **preuve de correction** (preuve de programme)
  - **C'était donc des commentaires formels**

- Les assertions définissent le comportement d'un *programme* par des formules logiques qui caractérisent, de manière pertinente et synthétique, les **états successifs** de toute exécution

- Pouvant référencer des états, cette technique s'oppose aux techniques formelles purement fonctionnelles, comme le lambda-calcul

Principes de spécification

- **Comme pour toute spécification formelle, une assertion est utilement redondante avec :**
  - les textes en langages naturels : résumé, commentaire, libellé...
  - le code exécutable : instructions d'un programmes pour l'exécuter, comprises par un compilateur, un interprète ou une machine.

- **Les assertions peuvent fournir une spécification complète ou partielle d'un programme.**
  - Elles peuvent être données de manière incrémentale, au cours d'un processus logiciel.

Eléments de syntaxe

- **Une assertion est placée à un endroit caractéristique du moment où elle doit être vraie, et introduite par un mot réservé:**
  - pre:, post:, inv:, etc.

- **Elle est composée d'une ou plusieurs clauses, chacune étant reliée aux autres par une conjonction implicite (and).**

- **Chaque clause est une proposition logique**

- **Chaque type d'assertions a un rôle, afin de**
  - Faciliter la conception des classes
  - Raisonner sous forme de contrats

Termes d'une assertion

- Les **termes** des formules logiques ne sont pas nécessairement liés à des **états** des variables d'un programme.
- Elles peuvent donc exprimer une grande variété de propriétés...
- Exemple d'assertions intégrées au langage de programmation :

```
class Account
...
  deposit (sum: Integer )
  -- deposit sum into the account
   pre:
    trueDeposit: sum >= 0
   post:
    balance = balance@pre + sum
```

Termes d'une assertion (suite)

- Exemple de vue concrète (boîte blanche) :

```
class Account
...
  deposit (sum: Integer )
  -- deposit sum into the account
   pre:
    trueDeposit: sum >= 0
   body:
     balance := balance + sum
   post:
    balance = balance@pre + sum
```

Assertions dans les langages

- **Par nature, les assertions sont associées aux descriptions d'un logiciel :**
  - *schémas de conception, textes des programmes*

- **Elles peuvent être :**
  - intégrées au langage et aux descriptions :
    - *clause assert : C, C++, Ada; assertions d'Eiffel*
    - *D'où code autonome, autodocumenté, autovérifiable, autotestable*
  - définies dans un langage compagnon, mais séparées des descriptions :
    - *contraintes en langage OCL pour UML, assertions pour plates-formes de composants*
  - proposées en extension d'un langage, placées en commentaire :
    - *assertions pour Java : JML, iContract...*

- *NB: concilier **expressivité**, **efficacité** des techniques de vérification et **performance** de l'évaluation des assertions est encore un problème de recherche*

Redondance des assertions

- **Comme pour toute spécification formelle, une assertion est utilement redondante avec :**
  - les textes en langages naturels : *résumé, commentaire, libellé...*
  - le code exécutable : *instructions d'un programmes pour l'exécuter, comprises par un compilateur, un interprète ou une machine.*

- **Les assertions peuvent fournir une spécification complète ou partielle d'un programme**

- **Elles peuvent être données de manière incrémentale, au cours d'un processus logiciel**

Vérification des assertions

- **Ces propriétés peuvent se vérifier par :**
  - des relectures ou des inspections
  - des preuves « à la main » (avec sa tête)
  - des tests, lorsqu'elles sont exécutables en un temps acceptable
    - plus il y a de quantifications universelles ou existentielles, de parcours de collections, plus cela est coûteux !

- **Des assertions exécutables**
  - Les langages qui intègrent des assertions exécutables fournissent
    - des mécanismes d'armement sélectifs (problème du ralentissement de l'exécution) : *quelles assertions évaluer, pour quels objets ou classes, en fonction de leur catégorie et du degré de confiance dans les composants d'un logiciel*
    - des moyens de traitement en cas de violation : d*éclenchement d'une exception, par défaut un diagnostic, possibilité de « programmation défensive »*

Utilisation des assertions

- **Technique de spécification :**
  - *formelle, partielle, incrémentale*
- **Conception et programmation par contrats :**
  - *établissement clair des responsabilités entre les clients et les programmeurs, utilisation dès la conception (OCL)*
- **Aide à la fiabilité :**
  - *pour la correction vs exceptions pour la robustesse*
- **Technique de preuve :**
  - *méthode axiomatique de Hoare, Dijkstra, Morgan, Gries, refinement calculus...*
- **Plus précis que le typage classique :**
  - *héritage des invariants, règles de redéfinitions covariantes, contraintes d'intégrité...*
- **Aide à la maintenance et aux autotests :**
  - *non régression, diagnostic, localisation des erreurs...*
- **Aide à la documentation, à la réutilisation :**
  - *Compléments lisibles des descriptions en langage naturel, documentation testable en accord avec la réalité du code*

Clause d'une assertion (pas forcément OCL)

- **Chaque clause est une proposition logique formée :**
  - de connecteurs logiques (not, and, or, xor, implies)
  - d'opérateurs booléens (=, <>, <, , etc)
  - de valeurs littérales, de constantes symboliques, d'attributs
  - et d'appels de fonctions sans effet de bord

- **Exemple :**

```
x > 0 and premier(x) implies 0 <= result <= y
```

- **Chaque clause peut être introduite par un label (identificateur)**
  - qui rappelle de manière brève l'intention, la propriété énoncée par la clause
  - Cela sert aussi à identifier les formules dans les diagnostics en cas de violation
  - En l'absence de label, les clauses sont numérotées

Rôle des assertions

Préconditions

- doivent être vraies juste avant chaque appel de la méthode.
- expriment les conditions ou hypothèses à satisfaire pour que le développeur sache implémenter la méthode.

Postconditions

- doivent être vraies juste après chaque sortie de la méthode.
- définissent l'essentiel :
  - du travail réalisé sur l'instance ou un système : cas d'un modifieur, générateur, initialiseur
  - ou du résultat retourné : cas d'un accesseur secondaire
- si elles sont assez précises, elles peuvent spécifier complétementou partiellement une primitive.

Exemple : classe Account

```
class Account

feature
balance: Integer
  -- attribut, accesseur primaire
minBalance: Integer = 1000
  -- constante, accesseur primaire

creation
initial (sum: Integer )
  -- initialize account with balance sum
  pre:
    sufficientDeposit: sum >= minBalance
  post:
    balance = sum
```

Accesseurs primaires et secondaires

Accesseur primaire

- permet d'accéder aux états ou aux constantes des objets,
- n'a pas de précondition, la plupart du temps
- ne peut se définir explicitement, mais leur définition implicite apparaît dans leur utilisation dans les autres assertions,
- est implémenté par des variables, attributs, constantes, fonctions.

Accesseur secondaire

- définit des valeurs, à partir d'autres valeurs ou états :
  - *services de confort, prédicats...*
- peut être défini explicitement par une postcondition sur la valeur retournée

Accesseur secondaire : classe Account

```
-- toujours dans la classe Account

mayWithdraw (sum: Integer): Boolean
  -- is account supplied enough to withdraw sum ?
  pre:
    trueWithdraw: sum >= 0
  body:
    Result := balance >= sum + minBalance
  post:
    Result = (balance >= sum + minBalance)
```

Constructions spécifiques aux langages d'assertions

Constructions sans effet de bord

- l'évaluation des assertions **ne doit avoir aucun effet de bord**
- chaque accesseur secondaire (fonction) ne doit avoir aucun effet de bord sur **l'état abstrait** d'un objet, visible à ses clients publiques et décrit par les accesseurs primaires
- chaque accesseur secondaire peut avoir un effet de bord sur l'état concret, privé, pour améliorer les performances : *accélérateurs de primitives d'accès, caches*

Accès aux valeurs antérieures

- Distinguent l'état d'une variable/expression à l'entrée et à la sortie d'une méthode
- Etat à la sortie : *pas de notation particulière*
- Etat à l'entrée : indiqué par un opérateur spécial : **`@pre`** en OCL

```
i = i@pre +1
```

Utilisation d'@pre : classe Account

```
deposit (sum: Integer )
-- deposit sum into the account
  pre:
     trueDeposit: sum >= 0
  body:
    move ( sum )
  post:
     balance = balance@pre + sum

withdraw (sum: Integer )
-- withdraw sum from the account
  pre:
    trueWithdraw: sum >= 0
    suppliedEnough: mayWithdraw(sum)
  body:
    move ( -sum )
  post:
    balance = balance@pre - sum

move (sum: Integer)  -- private
  body:
    balance := balance + sum
  post:
    balance = balance@pre + sum
```

Invariants

- Chaque instance est susceptible de passer par un grand nombre d'états **observables**, lors des périodes de **stabilité** de l'instance :
  - ***pas d'initialiseur ou de modifieur actif***

- Un **invariant d'instance** exprime une propriété remarquable, vraie pour tous ses instants stables et observables.

- Un invariant d'instance doit donc être vrai juste après :
  - chaque primitives d'initialisation,
  - chaque modifieur exporté

- Donc, pour chaque initialiseur/constructeur ou modifieur :

```
postcondition => invariant
```

- Et chaque **précondition**, sauf pour les constructeurs, requiert l'**invariant**.

- **Exemple: classe Account**

```
inv:
balance >= minBalance
```

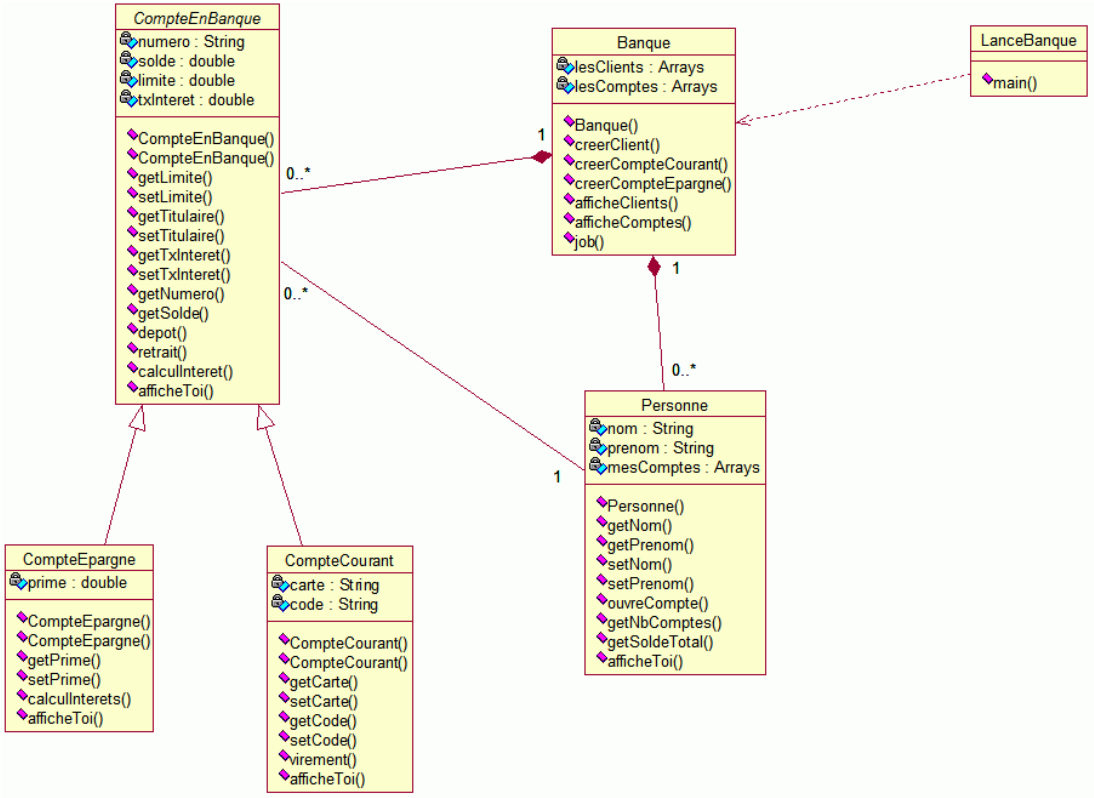
OCL : 1ère partie

- **Introduction**
  - OCL et UML
  - Motivations
  - OCL : principes et forme du langage
- **OCL : première approche**
  - Contraintes et contexte
  - Commentaires
  - Stéréotypes et mots-clés pour structurer les spécifications
  - Types de spécification
- **Types et valeurs de base**
  - Types du modèle
  - Règles de précédence
  - Enumérations
  - Conformance de type

OCL et UML

- UML est un langage de modélisation essentiellement graphique.

- Dans les diagrammes il est difficile, voire impossible dans certains cas, de préciser de manière complète toutes les subtilités d'un modèle.



OCL et UML

- **Contourner le problème ?**
  - écrire des spécifications plus complètes en langue naturelle
  - inconvénient : des ambiguïtés restent possibles

- **l'utilisation d'un langage formel avec une sémantique connue s'impose !**

- **OCL est une réponse à ces attentes**
  - un langage formel
  - pour annoter les diagrammes UML
  - permettant notamment l'expression de contraintes

OCL : objectifs

- **Accompagner les diagrammes UML de descriptions :**
  - précises
  - non ambiguës
- **Eviter les désavantages des langages formels traditionnels**
  - peu utilisables par les utilisateurs et les concepteurs non « matheux »
- **Rester facile à écrire**
  - Tout en étant orienté objet
- **Et facile à lire**

Petit historique

- **OCL s'inspire de Syntropy**
  - méthode basée sur une combinaison d'OMT (Object Modeling Technique) et d'un sous-ensemble de Z.
- **Origine**
  - OCL a été développé à partie de 95 par Jos Warmer (IBM)
  - sur les bases du langage IBEL (Integrated Business Engineering Language).
- **Première définition : IBM, 1997**
- **Formellement intégré à UML 1.1 en 1999**
- **OCL2.0 intégré dans la définition d'UML2.0 en 2003**
  - conforme à UML 2 et au MOF 2.0
  - fait partie du catalogue de spécifications de l'OMG
  - chapitres 7 (OCL Language Description) et 11 (OCL Standard Library)

Bibliographie

- *Steve Cook, John Daniels* , Designing Object Systems: Object-Oriented Modelling with Syntropy, Prentice Hall, 1994

- *Pierre-Alain Muller, Nathalie Gaertner , Modélisation objet avec UML, Eyrolles, 2000*

- *OMG (Object Management Group)* , "Uml 2.0 ocl specification. Technical report, Object Management Group, 2003. ptc/03-10-14«

- *OMG (Object Management Group)* , "Uml 2.0 superstructure specification. Technical report, Object Management Group, 2003. ptc/03-08-02"

OCL : principes

- **La notion de contrainte**
  - **Définition :** Une contrainte est une expression à valeur booléenne que l'on peut attacher à n'importe quel élément UML
  - Elle indique en général une restriction ou donne des informations complémentaires sur un modèle
- **OCL : Langage typé, basé sur**
  - La théorie des ensembles
  - La logique des prédicats
- **OCL : Langage déclaratif**
  - Les contraintes ne sont pas opérationnelles.
    - On ne peut pas invoquer de processus ni d'opérations autres que des requêtes
    - On ne décrit pas le comportement à adopter si une contrainte n'est pas respectée
- **OCL : Langage sans effet de bord**
  - Les instances ne sont pas modifiées par les contraintes

Utilisation des contraintes

- **Description d'invariants sur les classes et les types**
- **Préconditions et postconditions sur les opérations**
- **Contraintes sur la valeur retournée par une opération**
- **Règles de dérivation des attributs**
- **Description de cibles pour les messages et les actions**
- **Expression des gardes**
  - conditions dans les diagrammes dynamiques
- **Invariants de type pour les stéréotypes**
  - Les contraintes servent en particulier à décrire la sémantique d'UML.

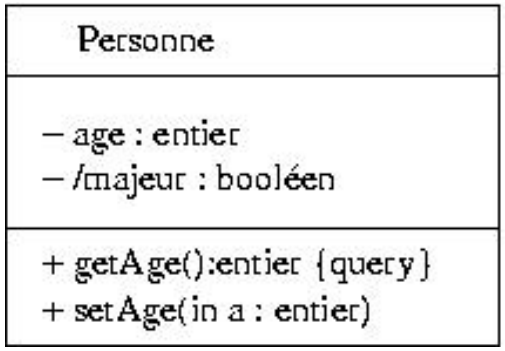
Contraintes et contexte

- **Une contrainte OCL est liée à un contexte :**
  - le type,
  - la méthode
  - ou l'attribut auquel la contrainte se rapporte

```
context moncontexte <stéréotype> :
     Expression de la contrainte
```

- **Exemple :**

```
context Personne inv :
(age <= 140) and (age >=0)
-- l'âge ne peut dépasser 140 ans
```



-- : commentaire en OCL (qui s'achève avec la fin de la ligne)

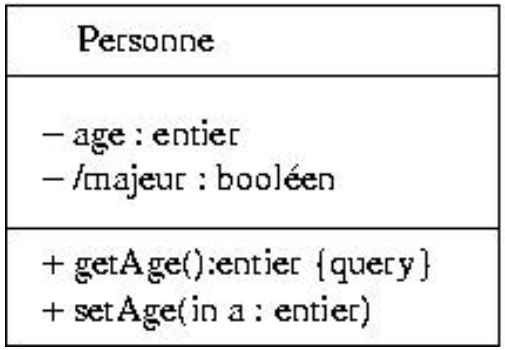
Contexte et package

- Si les contraintes sont placées sans ambiguïté, pas besoin de plus
- Si elles sont dans un fichier global, il faut spécifier le package dans lequel les contraintes se placent

```
package Package::SubPackage

context X inv:
... un invariant ...

context X::operationName(..)
pre: ... une precondition ...

endpackage
```

Contraintes : les stéréotypes

- **Le stéréotype peut prendre les valeurs suivantes :**
  - **inv** invariant de classe
  - Un invariant exprime une contrainte prédicative sur un objet, ou un groupe d'objets, qui doit être respectée en permanence




```
context Personne inv :
(age <= 140) and (age >=0)
```

  - **pre** précondition
  - **post** postcondition
  - Une précondition (respectivement une postcondition) permet de spécifier une contrainte prédicative qui doit être vérifiée avant (respectivement après) l'appel d'une opération

```
context  Personne::setAge(a : integer)
pre:  (a <= 140) and (a >=0) and (a >= age)
post:  age = a
```

Contraintes : les stéréotypes (suite)

- **body** indique le résultat d'une opération *query*
- Ce type de contrainte permet de définir directement le résultat d'une opération

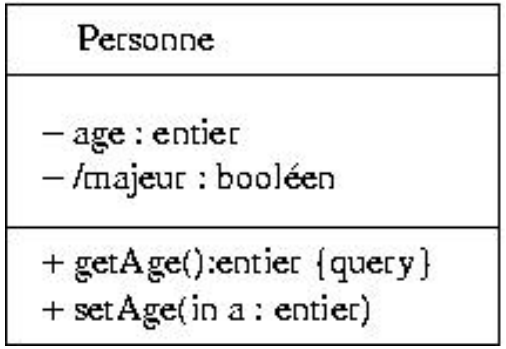
```
context  Personne::getAge() : integer
body:  age
```

- **init** indique la valeur initiale d'un attribut

```
context  Personne::age : integer
init:  0
```

- **derive** indique la valeur dérivée d'un attribut

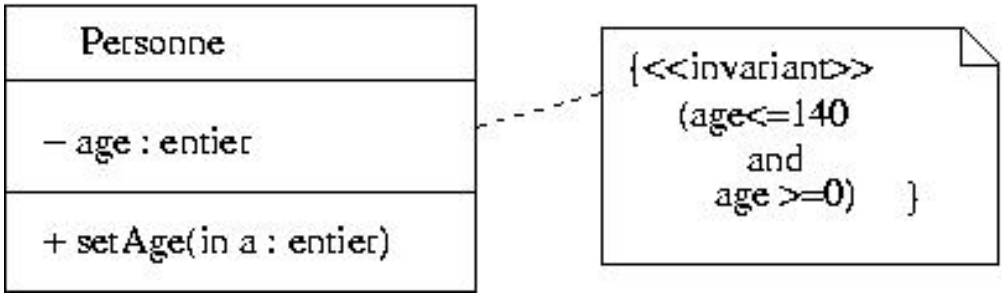
```
context  Personne::majeur : boolean
derive:  age>=18
```



Contraintes : version visuelle

- **Une version alternative, graphique**
  - un commentaire dans le diagramme UML.

- **Le contexte est l'élément UML auquel se rattache le commentaire contenant la contrainte**



- **Les outils UML permettent aussi la saisie de contraintes dans les « propriétés » de l'élément UML**
  - Cela définit automatiquement le contexte

Contrainte nommée / label

- **Une contrainte peut être nommée par un label**
- **Utilité**
  - Rappeler la contrainte dans une autre contrainte
  - Retrouver une contrainte en cas d'évaluation dans un outil, de production de code de vérification, etc.
- **Exemple :**
  - Reprise de la contrainte, nommée « ageBorné »

```
context Personne inv ageBorné:
(age <= 140) and (age >=0)
-- l'âge ne peut depasser 140 ans
```

Référence, nommage des objets

- ***Le mot-clef « self » permet de désigner l'objet à partir duquel part l'évaluation***
  - *Exemple :*

```
context Personne inv:
(self.age <= 140) and (self.age >=0)
-- l'âge ne peut depasser 140 ans
```

  - Caractère « . » : accès à l'attribut (cf. navigation)

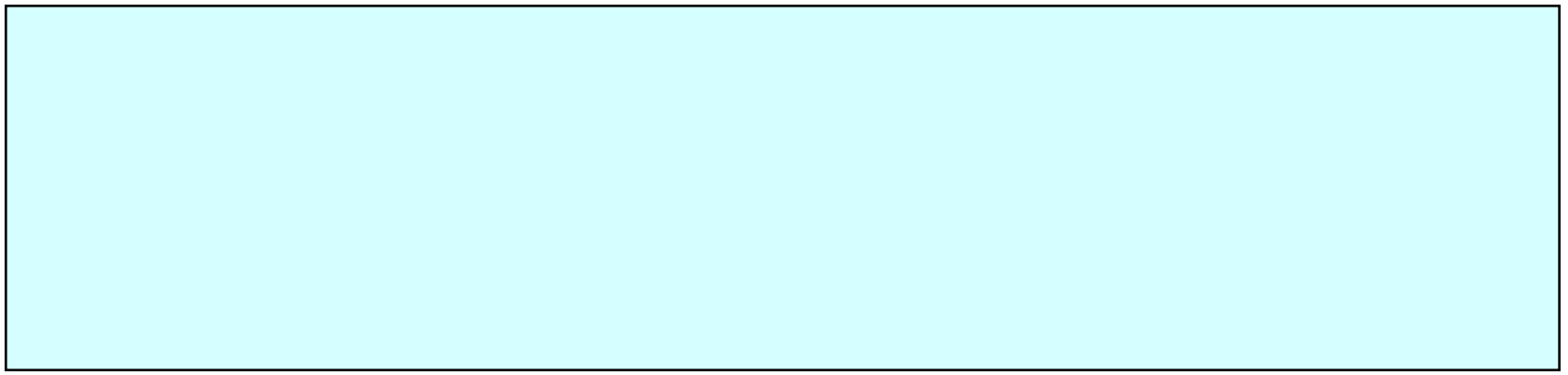
- ***Un nom formel peut être donné à l'objet à partir duquel part l'évaluation***
  - ***Exemple :***

```
context p : Personne inv:
(p.age <= 140) and (p.age >=0)
-- l'age ne peut depasser 140 ans
```

Exercice

- **Ajoutez un attribut mère de type Personne dans la classe Personne.**

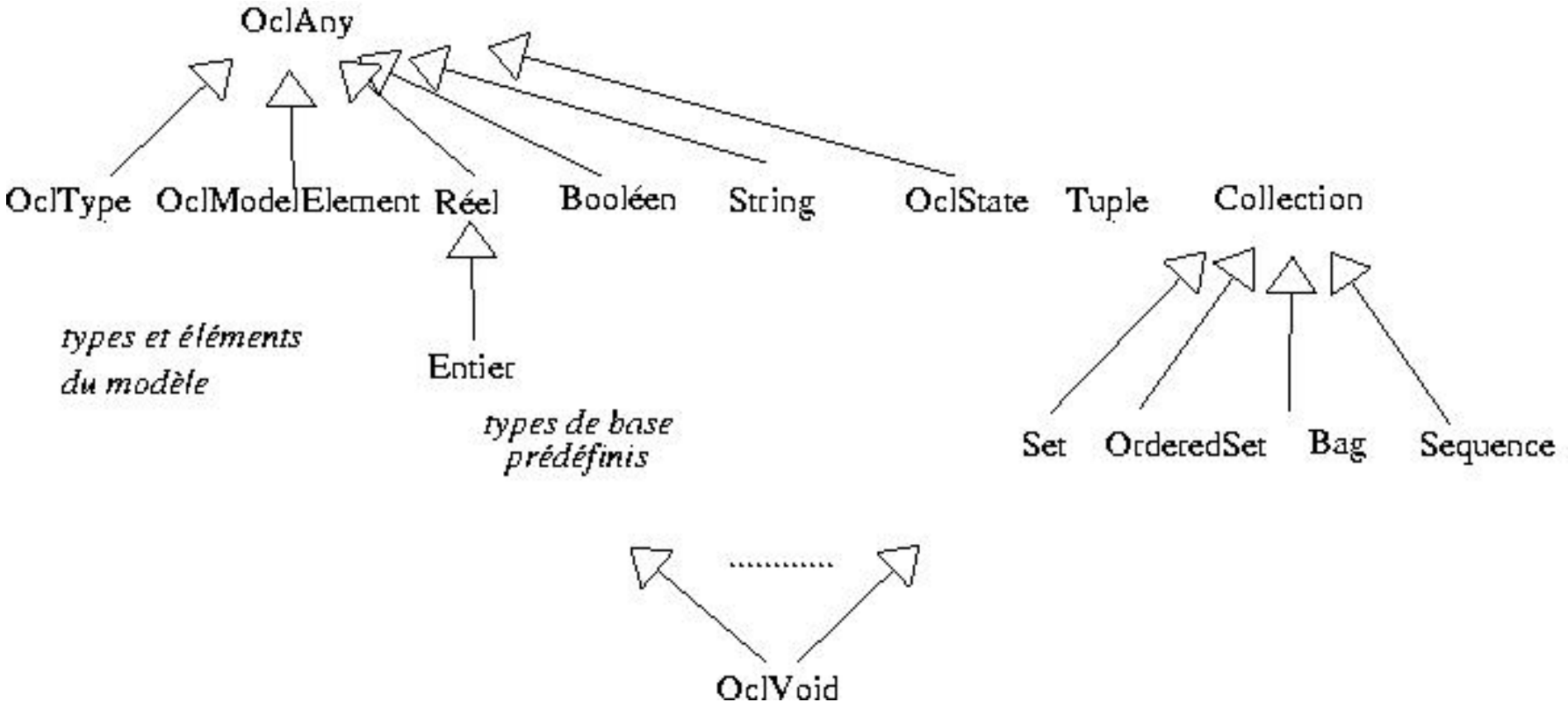
- **Ecrivez une contrainte précisant**
  - que la mère d'une personne ne peut être cette personne elle-même
  - et que l'âge de la mère doit être supérieur à celui de la personne



Types : hiérarchie

❒ Organisation des types dans une hiérarchie

- Des types de base, des collections, des éléments réflexifs sur le typage et certains éléments du langage…



Types

- **Les types de base prédéfinis :**
  - Entier
  - Réel
  - String
  - Booléen
- **Des types spéciaux s'y ajoutent :**
  - OclModelElement (énumération des éléments du modèle)
  - OclType (énumération des types du modèle)
  - OclAny (tout type autre que Tuple et Collection)
  - OclState (pour les diagrammes d'états)
  - OclVoid sous-type de tous les types

Types de base

Le type entier (Integer)

- *Les constantes littérales s'écrivent de manière ordinaire :*
  - *1 -5 22*
- *Les opérateurs sont les suivants :*
  - *= <> + - * / abs div mod max min < > <= >=*
- *L'opérateur - est unaire ou binaire*

Le type réel (Real)

- *Les constantes s'écrivent de manière ordinaire :*
  - *1.56 -5.7 3.14*
- *Les opérateurs sont les suivants :*
  - *= <> + - * / abs floor round max min < > <= >=*
- *L'opérateur - est unaire ou binaire*

Types de base

- **Le type chaîne de caractères (String)**
  - *Les constantes s'écrivent entre simples quotes :*
    - *'ceci est une chaîne OCL'*
  - *Les opérateurs sont (notamment) les suivants :*
    - =
    - *size*
    - *concat(String)*
    - *toUpper*
    - *toLower*
    - *substring(Entier, Entier)*

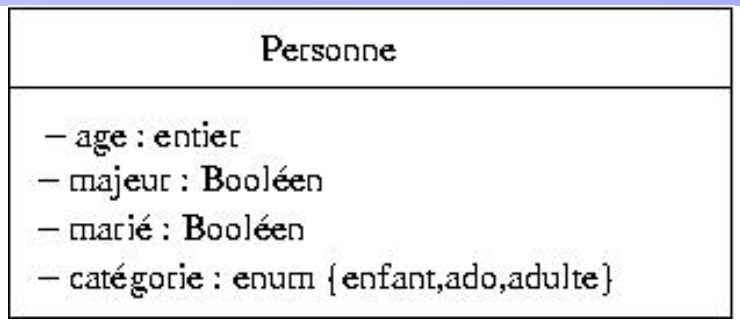
Types de base

Le type booléen (Boolean)

- *Les constantes s'écrivent*
  - *true*
  - *false*
- *Les opérateurs sont les suivants :*
  - *= or xor and not*
  - *b1 implies b2*
  - *if b then expression1 else expression2 endif*

Exercice

- **Avec la classe Personne « étendue »**
  - Indiquez qu'une personne mariée est forcément majeur



  - Trouvez une version plus compacte de l'expression suivante

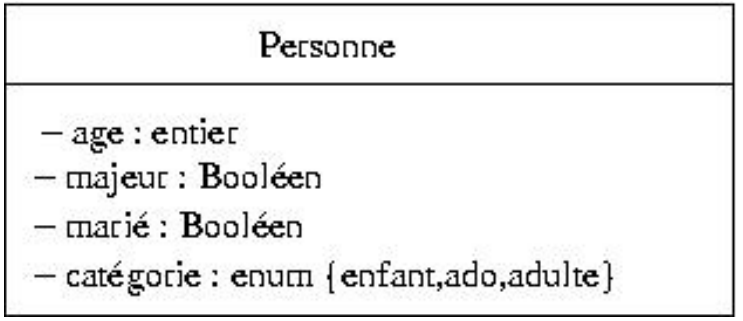
```
context Personne inv majeurIf:
if age >=18 then majeur=vrai
else majeur=faux endif
```

Précédence des opérateurs

- . ->
- *not - unaire*
- * /
- \+ -
- *if then else*
- < > <= >=
- <> =
- *and or xor*
- *implies*

Types énumérés

- **Leur syntaxe est la suivante**
  - `<nom_type_enuméré>::valeur`



- **Le nom du type est déduit de l'attribut de déclaration**
  - `catégorie => Catégorie`

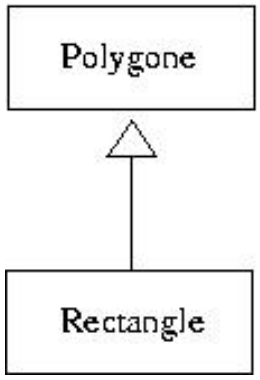
- **Exemple :**

```
context Personne inv:
if age <=12
    then categorie = Categorie::enfant
   else  if age <=18
            then categorie = Categorie::ado
            else categorie = Categorie::adulte
         endif
endif
```

Type des modèles

- **Les types des modèles utilisables en OCL sont**
  - les classifieurs, donc
  - les classes, les interfaces et les associations
- **Manipulation de la relation de spécialisation**
  - oclAsType(t) (conversion ascendante ou descendante de type vers t)
    - la conversion ascendante sert pour l'accès à une propriété redéfinie
  - oclIsTypeOf(t) (vrai si t est supertype direct)
  - oclIsKindOf(t) (vrai si t est supertype indirect)

```
p = r
p.oclAsType(Rectangle)
r.oclIsTypeOf(Rectangle)   (vrai)
r.oclIsKindOf(Polygone)    (vrai)
```



p : Polygone
r : Rectangle

(mauvais) exercice

- **En supposant l'existence**
  - d'un attribut hauteur dans la classe Rectangle
  - d'une méthode hauteur():Réel dans Polygone

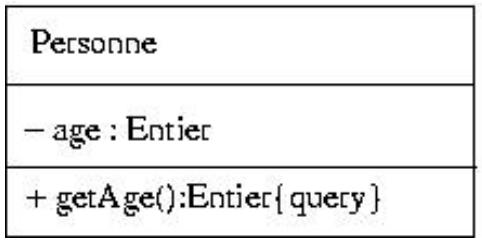
- **Ecrivez un invariant dans Polygone disant que le résultat de hauteur():Réel vaut hauteur pour les polygones qui sont des rectangles, sinon 0**

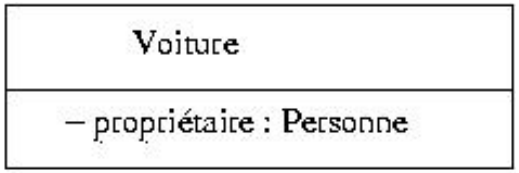
- **Ceci un exemple de très mauvaise conception objet !**

2ème partie

- **Navigation dans les modèles**
  - Accès aux attributs, opérations
  - Navigations sur les associations
  - Navigations sur les classes-association
- **Autres éléments du langage**
  - Structures let et def
  - if
  - @pre
  - result
  - Types OCL avancés
- **Collections**
  - Hiérarchie
  - Opérations spécifiques
  - Opérations collectives
  - Conformance de type
- **Exploitation d'OCL**

Navigation : accès aux attributs





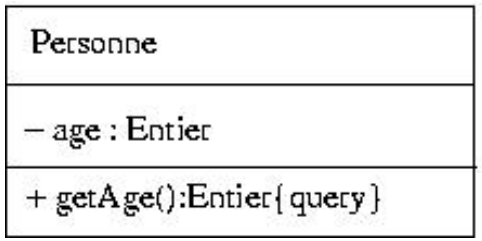
- **Pour faire référence à un attribut de l'objet désigné par le contexte, il suffit d'utiliser le nom de cet élément**

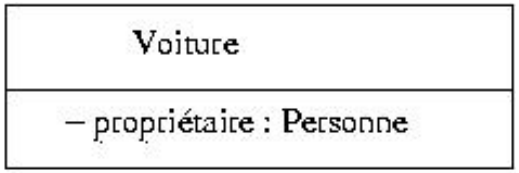
```
context Personne inv:
    …age…
```

- **L'objet désigné par le contexte est également accessible par l'expression *self***
  - On peut donc également utiliser la notation pointée

```
context Personne inv:
    self.age…
```

Navigation : accès aux attributs

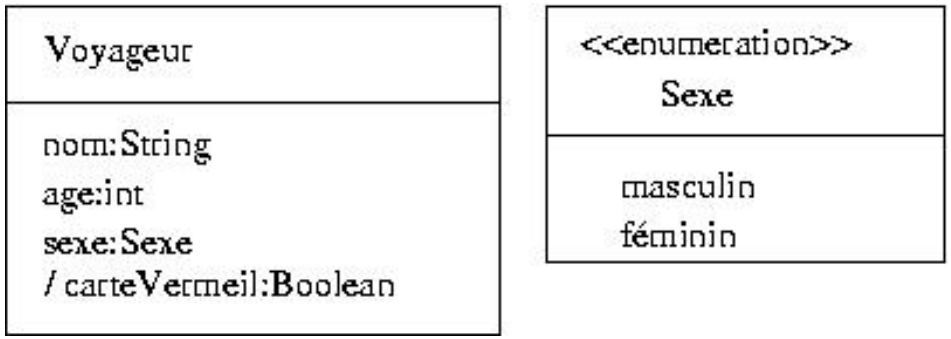




- **L'accès (navigation) vers un attribut s'effectue en mentionnant l'attribut derrière l'opérateur d'accès noté '.'**

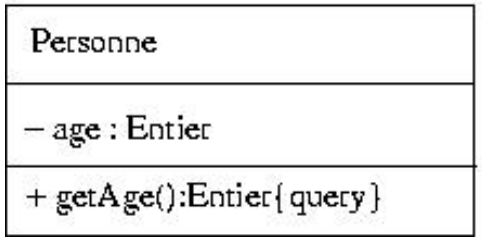
```
context Voiture inv   propriétaireMajeur :
self.propriétaire.age >= 18

context Voiture inv   propriétaireMajeur :
propriétaire.age >= 18

context v : Voiture inv   propriétaireMajeur :
v.propriétaire.age >= 18
```

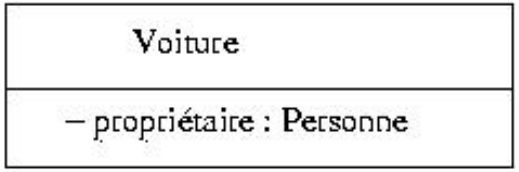
Exercice



- **Ecrivez la contrainte qui caractérise l'attribut dérivé carteVermeil**
  - Un voyageur a droit à la carte vermeil si c'est une femme de plus de 60 ans ou un homme de plus de 65 ans.

Accès aux opérations





❒ **notation identique à la notation utilisée pour l'accès aux attributs (utilisation du '.')**

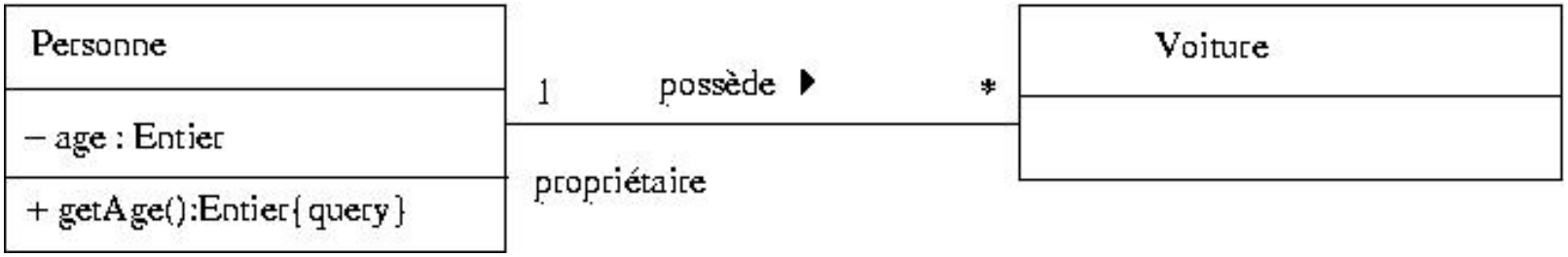
```
context Voiture inv :
self.propriétaire.getAge() >= 18

context Personne inv :
  …getAge()…
```

Seules les opérations de type *query* peuvent apparaître dans les contraintes puisqu'elles ne modifient pas l'objet sur lequel porte la contrainte

- Les contraintes n'ont pas d'effet de bord

Naviguer via les associations

- **La navigation le long des associations se fait en utilisant :**
  - soit les noms de rôles
  - soit les noms des classes extrémités en mettant leur première lettre en minuscule, à condition qu'il n'y ait pas ambiguïté



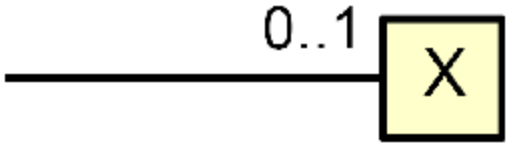

```
context Voiture inv :
self.propriétaire.age >= 18

context Voiture inv :
self.personne.age >= 18
```

- **Que faudrait-il pour qu'il y ait ambiguïté sur cet exemple ?**

Naviguer via les associations - typage

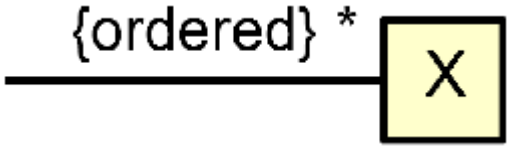
- **Le type du résultat dépend**
  - de la multiplicité du côté de l'objet référencé
  - du type de l'objet référencé

- **Si on appelle *X* la classe de l'objet référencé, dans le cas d'une multiplicité de :**
  - 1, le type du résultat est *X*



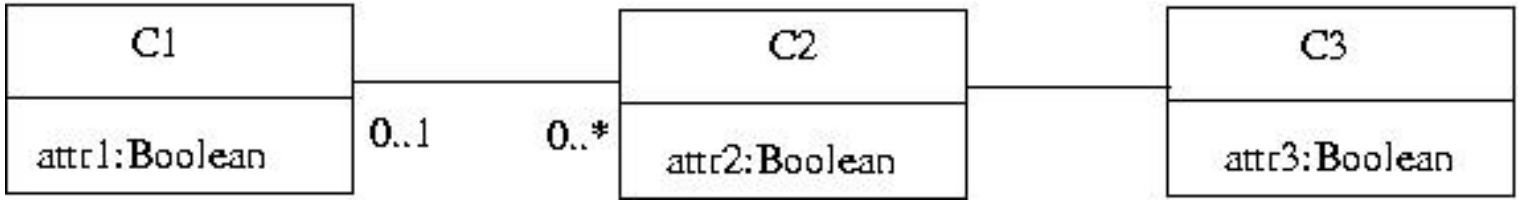
  - * ou 0..*n*, …, le type du résultat est *Set*(*X*)

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



- ***On y reviendra…***

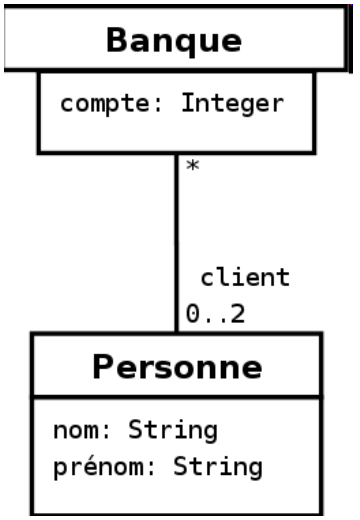
Navigation - ambiguïtés




```
context C1 inv :
c2.attr2=c2.c3.attr3

context C2 inv :
attr2=c3.attr3
```

- Les deux contraintes ci-dessus sont-elles équivalentes ?

Navigation via une association qualifiée



- Une association qualifiée utilise un ou plusieurs qualificatifs pour sélectionner des instances de la classe cible de l'association.
  - En OCL : utilisation de [ ]

- Dans le contexte de banque (context Banque) on fait référence au nom des clients dont le compte porte le numéro 19503800 :

```
self.client[19503800].nom
```

- Dans le cas où il y a plusieurs qualificatifs, il faut séparer chacune des valeurs par une virgule en respectant l'ordre des qualificatifs du diagramme UML.

- Il n'est pas possible de ne préciser la valeur que de certains qualificatifs en en laissant d'autres non définis.

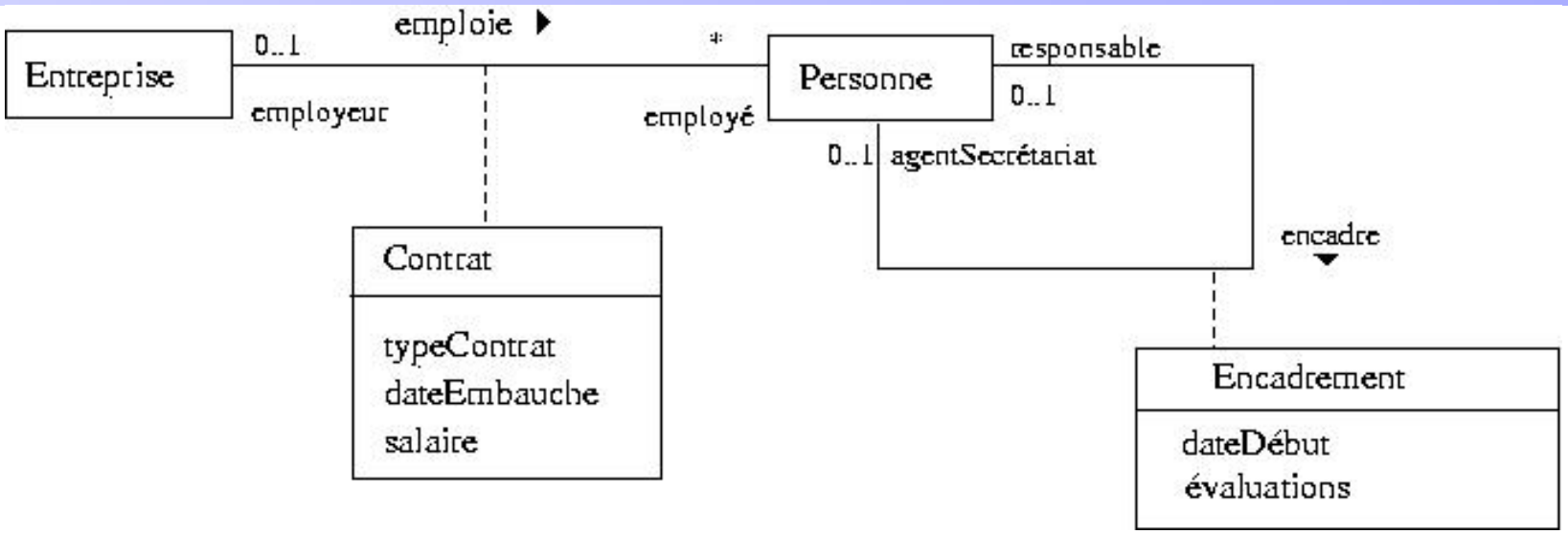
- Par contre, il est possible de ne préciser aucune valeur de qualificatif :

```
self.client.nom
```

- le résultat sera l'ensemble des noms de tous les clients de la banque

Attention à la cardinalité de la cible…

Navigation et classes association



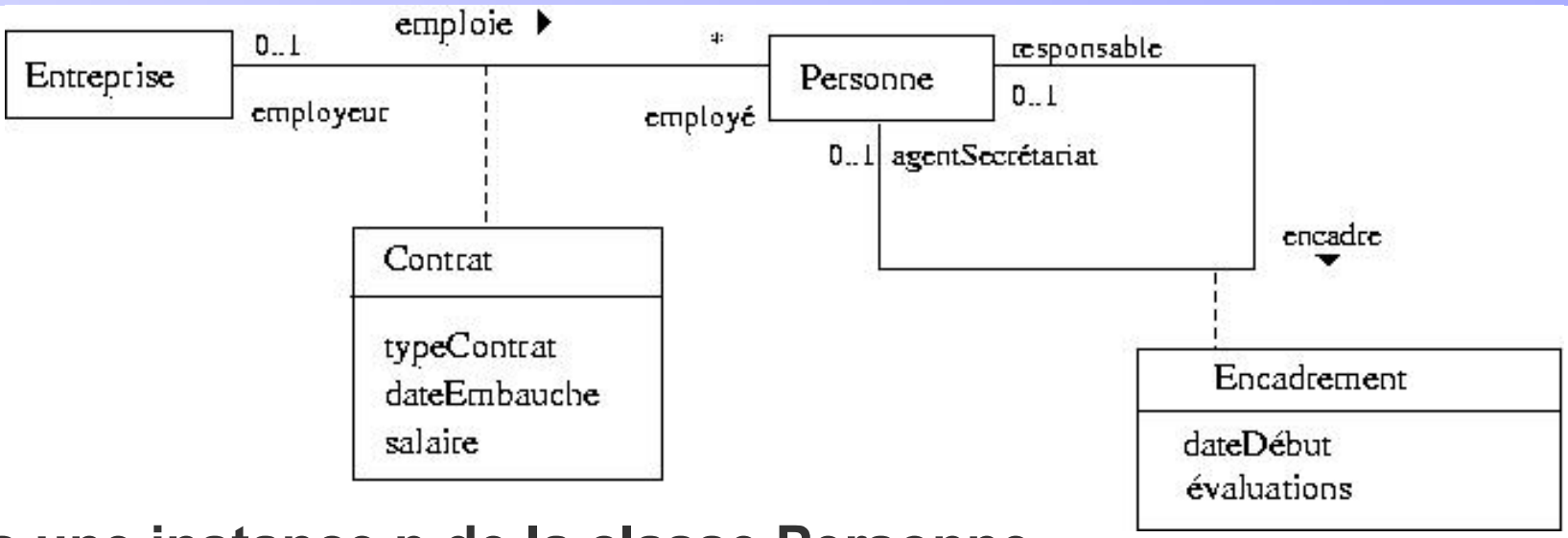
- **Pour naviguer vers une classe association, on utilise le nom de la classe (en mettant le premier caractère en minuscule).**

```
context p : Personne inv :
p.contrat.salaire >= 0
```

- **Une autre manière de naviguer consiste à utiliser le nom de rôle opposé (c'est même obligatoire pour une association réflexive)**

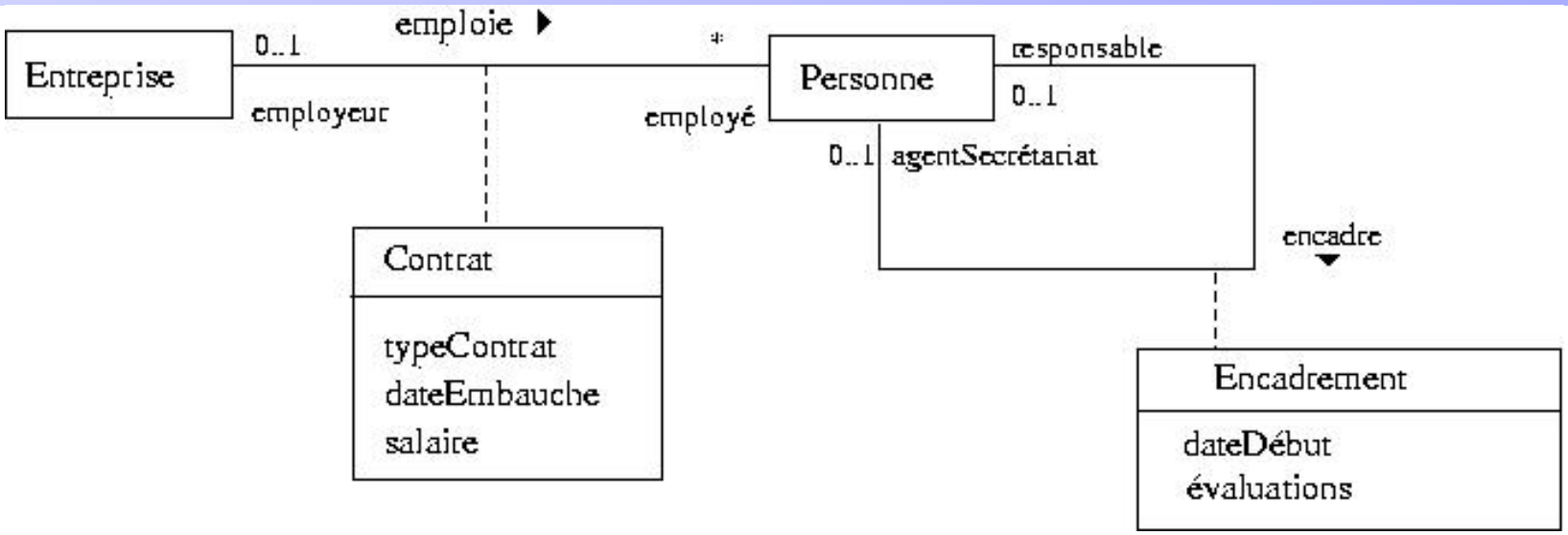
```
context p : Personne inv :
p.contrat[employeur].salaire >= 0
```

Exercice



- **Depuis une instance p de la classe Personne**
  1. Comment naviguer vers l'objet Encadrement de son responsable ?
  2. Comment naviguer vers l'objet Encadrement de son agentSecrétariat ?

Navigation et classes association (suite)

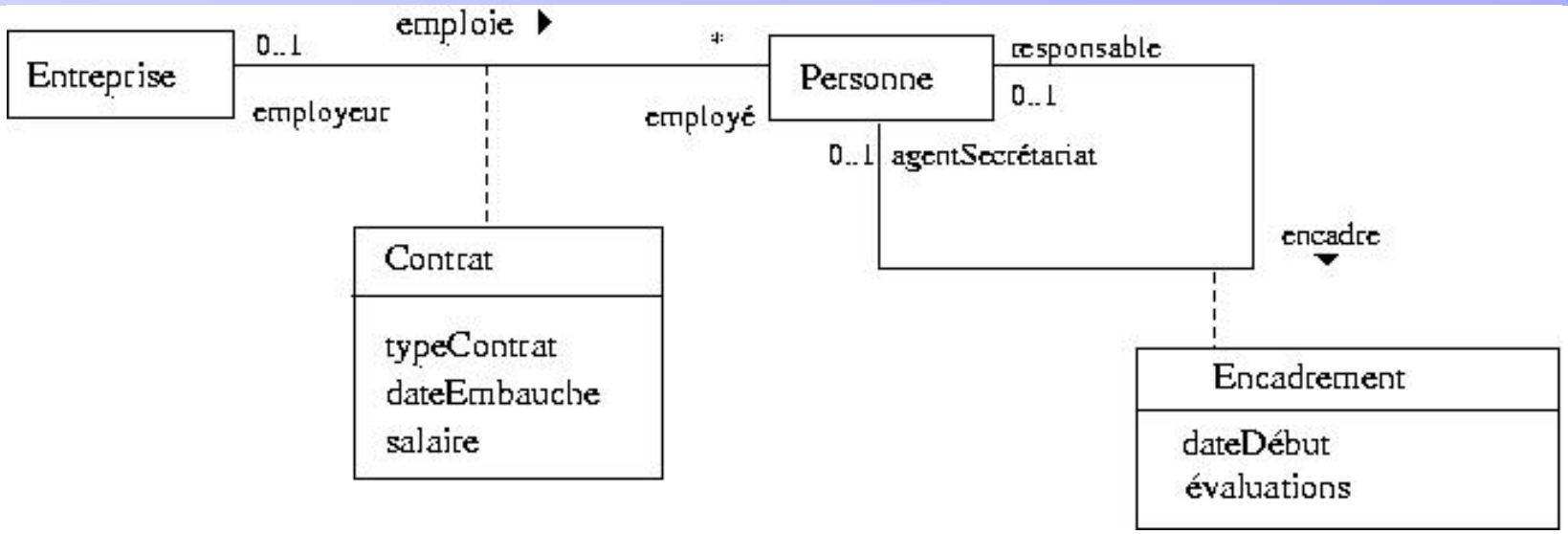


❒ **Pour naviguer <u>depuis</u> une classe association, on utilise les noms des rôles**

- ou s'il n'y en a pas le nom de la classe concernée avec la première lettre en minuscule
- La navigation depuis une classe association vers un rôle ne peut donner qu'un objet.

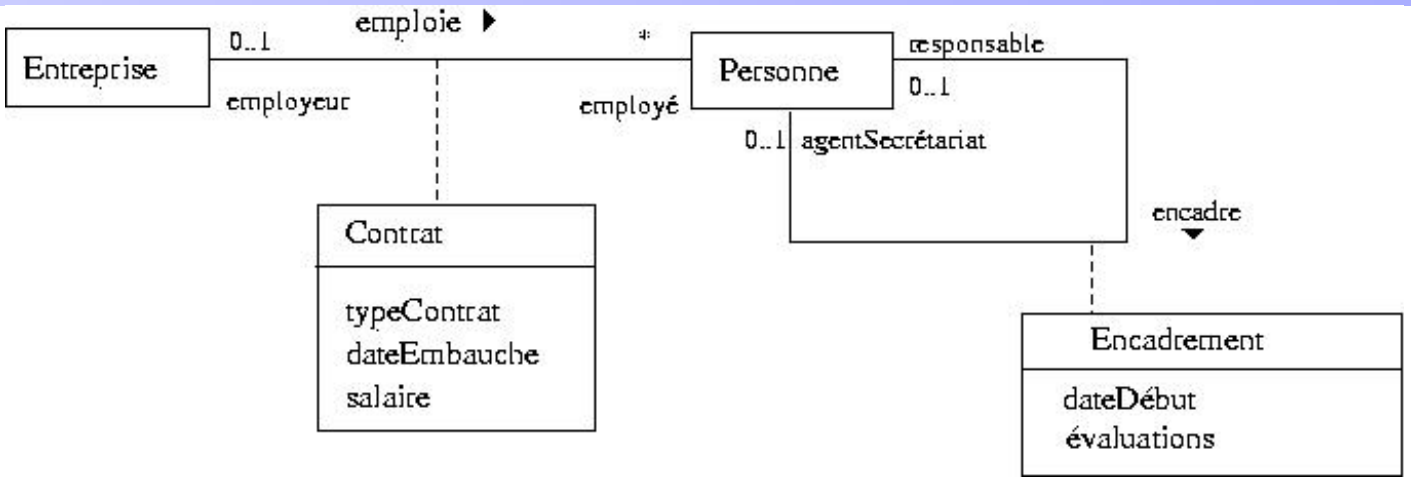
```
context c:Contrat inv :
    c.employé.age >= 16
```

Exercice



- Le salaire d'un agent de secrétariat est inférieur à celui de son responsable ?
- Un agent de secrétariat a un type de contrat ‘agentAdministratif‘ (String) ?

Exercice



De l'aide ?
Un diagramme d'objets

- Un agent de secrétariat a une date d'embauche antérieure à la date de début de l'encadrement (on suppose que les dates sont des entiers)

- Même chose dans le contexte de la classe Personne

Structure let

- **On peut définir des variables pour simplifier l'écriture de certaines expressions. On utilise pour cela la syntaxe suivante :**

```
let variable : type = expression1 in
                      expression2
```

- **Exemple :**
  - *en ajoutant l'attribut dérivé impot dans Personne,*
  - *on pourrait écrire la contrainte suivante pour définir cet attribut*

```
context  Personne inv :
let montantImposable : Real = contrat.salaire*0.8 in
if (montantImposable >= 100000)
        then impot = montantImposable*45/100
      else if  (montantImposable >= 50000)
                          then impot = montantImposable*30/100
                          else impot = montantImposable*10/100
                endif
endif
```

  - *il serait plus correct de vérifier tout d'abord que contrat est valué*

Structure def

- **Si on veut définir une variable utilisable dans plusieurs contraintes de la classe, on peut utiliser la construction def :**

- **Exemple :** *montant imposable*

```
context  Personne
def : montantImposable : Real  = contrat.salaire*0.8
```

- **Lorsqu'il est utile de définir de nouvelles opérations, on peut procéder avec la même construction def**

- **Exemple :** *ageCorrect*

```
context  Personne
def : ageCorrect(a : Real ):Booléen = a>=0 and a<=140
```

Désignation de la valeur antérieure

- ***Dans une post-condition, il est utile de pouvoir désigner la valeur d'une propriété avant l'exécution de l'opération***
  - *Nous faisons de la spécification par **état***
  - *Nous décrivons donc les modifications d'état en exprimant les **différences** avant et après (l'appel d'une opération)*

- ***Ceci se fait en suffixant le nom de la propriété avec @pre***
  - ***Exemple :***

```
context Personne :: feteAnniversaire()
pre : age < 140
post :age = age @pre + 1
```

Désignation du résultat d'une opération

- L'objet retourné par une opération est désigné par result
- Exemple :
  - *Une manière de décrire la post-condition de l'opération getAge vous est indiquée ci-dessous.*

```
context Personne :: getAge() : integer
post : result = age
```

- Lorsque la postcondition se résume à décrire la valeur du résultat, c'est équivalent à l'utilisation de la construction body

Exercice

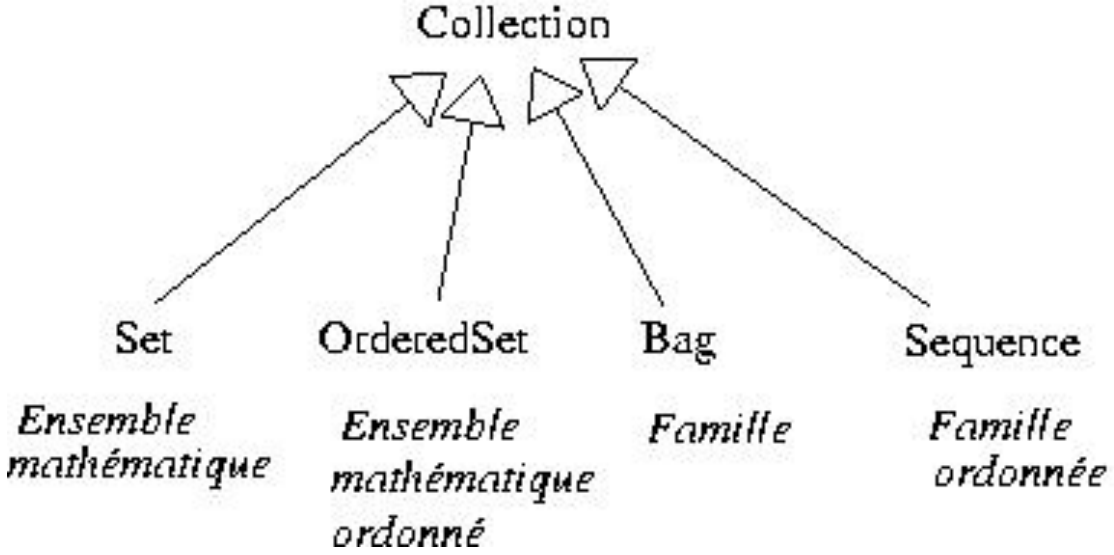
- Imaginez une classe Etudiant, disposant de 3 notes et munie d'une opération mention qui retourne la mention de l'étudiant sous forme d'une chaîne de caractères.

- Ecrivez les contraintes en utilisant let et result pour écrire la postcondition de mention

Types OCL avancés

- **OclInvalid**
  - Type conforme à tous les autres qui ne contient que la valeur invalid (ex : résultat d'une division par 0).
- **OclVoid**
  - Type conforme à tous les autres qui ne contient que la valeur null (pas de référence).
- **Tout appel d'opération appliqué à null produit invalid.**
- **null ou invalid = valeur indéfinie**
  - en général, une expression dont une partie s'évalue à indéfini est elle-même indéfinie.
  - cas particuliers des booléens : true or indéfini = true
- **OclAny**
  - Super-type de tous les types d'OCL, propose des opérations qui s'appliquent à tout objet

Collections : hiérarchie



- **Cinq types de collections existant en OCL**
  - Collection est un type abstrait
  - Set correspond à la notion mathématique d'ensemble
  - OrderedSet correspond à la notion mathématique d'ensemble ordonné
  - Bag correspond à la notion mathématique de famille (un élément peut y apparaître plusieurs fois)
  - Sequence correspond à la notion mathématique de famille, et les éléments sont, de plus, ordonnés

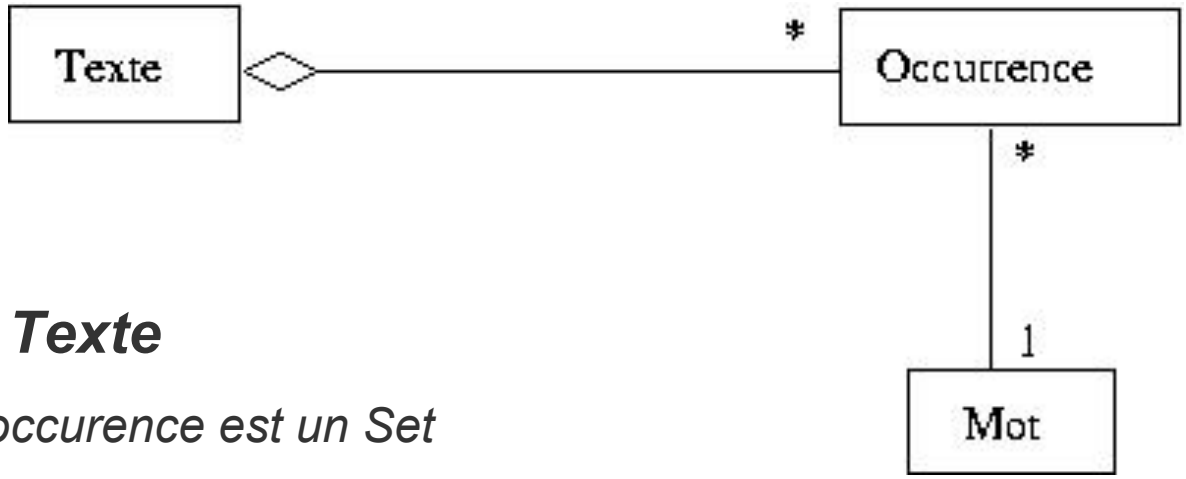
Collections : litéraux

- **On peut définir des collections par des littéraux de la manière suivante.**
  - Set { 2, 4, 6, 8 }
  - OrderedSet { 2, 4, 6, 8 }
  - Bag { 2, 4, 4, 6, 6, 8 }
  - Sequence { 'le', 'chat', 'boit', 'le', 'lait' }
  - Sequence { 1..10 } spécification d'un intervalle d'entiers

- **Les collections peuvent avoir plusieurs niveaux, il est possible de définir des collections de collections, par exemple :**
  - Set { 2, 4, Set {6, 8 }}

Retour sur les navigations

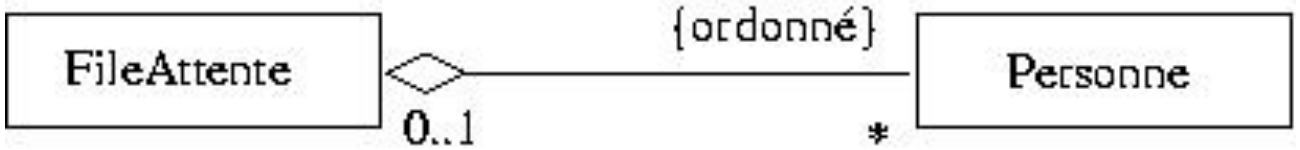
- ***On obtient naturellement un Set en naviguant le long d'une association ordinaire***
  - *ne mentionnant pas les contraintes bag ni seq à l'une de ses extrémités*



- ***Dans le contexte Texte***
  - *l'expression self.occurence est un Set*
  - *l'expression self.occurrence.mot est un Bag*

Retour sur les navigations

- **On peut obtenir un ensemble ordonné en naviguant vers une extrémité d'association munie de la contrainte *ordered***



- **Dans le contexte FileAttente**
  - l'expression personne est un OrderedSet

- **On peut obtenir une séquence en naviguant vers une extrémité d'association munie de la contrainte seq**

- **Dans le contexte Texte**
  - l'expression self.mot est une Sequence

Opérations sur collection

- **Nous présentons les principales opérations applicables à toutes les collections**
  - Pour une liste plus exhaustive, voir l'annexe
- **Nous considérons des collections d'éléments de type T**
  - Les opérations sur une collection sont mentionnées avec la flèche formée de deux caractères ->
- **Itérateurs**
  - Les opérations qui prennent une expression comme paramètre peuvent déclarer optionnellement un itérateur
  - toute opération de la forme operation(expression) existe également sous deux formes plus complexes.
    - operation(v | expression-contenant-v)
    - operation(v : Type | expression-contenant-v)

Opérations sur toutes types de collection

- Deux opérations permettent de tester si la collection est vide ou non

- Elles servent en particulier à tester l'existence d'un lien dans une association dont la multiplicité inclut 0

```
isEmpty() : Boolean
notEmpty : Boolean
```

- Exemple :
  - *Retour sur la contrainte de cohérence de la date d'embauche*
  - *nous pouvons à présent l'écrire de manière plus précise, en vérifiant que la personne a bien un agent de secrétariat*

```
context p : Personne
inv :
  p.agentSecretariat->notEmpty() implies

        p.agentSecrétariat.contrat.dateEmbauche <=
                p.encadrement[agentSecrétariat].dateDebut
```

Opérations sur toutes types de collection

- **Taille d'une collection**

```
size() : integer
```

- **Nombre d'occurrences d'un objet dans une collection**

```
count(unObjet : T) : integer
```

- **Addition des éléments de la collection**
  - Il faut que le type T supporte une opération + associative

```
sum() : T
```

- **Appartenance d'un élément à la collection**
  - Deux opérations retournent vrai (resp. faux) si et seulement si un certain objet est (resp. n'est pas) un élément de la collection

```
includes(unObjet : T) : Boolean
excludes(o : T) : Boolean
```

Opérations sur toutes types de collection

Appartenance des éléments d'une collection à la collection

- Deux opérations retournent vrai (resp. faux) si et seulement si la collection contient tous les (resp. ne contient aucun des) éléments d'une certaine collection passée en paramètre

```
includesAll(uneCollection : Collection(T)) : Boolean
excludesAll(uneCollection : Collection(T)) : Boolean
```

Vérification d'une propriété pour tous les éléments d'une collection

- Une opération permet de vérifier qu'une certaine expression passée en paramètre est vraie pour tous les éléments de la collection

```
forAll(uneExpression : Boolean) : Boolean
```

- Cette opération a une variante permettant d'itérer avec plusieurs itérateurs sur la même collection. On peut ainsi parcourir des ensembles produits de la même collection

```
forAll( t1,t2 : T | uneExpression-contenant-t1-et-t2) : Boolean
```

Opérations sur toutes types de collection

Existence d'un élément satisfaisant une expression

- L'opération suivante vaut vrai si et seulement si au moins un élément de la collection satisfait une certaine expression passée en paramètre

```
exists(uneExpression : Boolean) : Boolean
```

Itération

- Cette opération, plus complexe que les autres, permet de les généraliser
  - *i est un itérateur sur la collection*
  - *acc est un accumulateur initialisé avec uneExpression.*
  - *L'expression ExpressionAvecietacc est évaluée pour chaque i et son résultat est affecté dans acc.*
  - *Le résultat de iterate est acc.*

```
iterate(i : T ; acc : Type = uneExpression | ExpressionAvec-i-et-
   acc) : Type
```

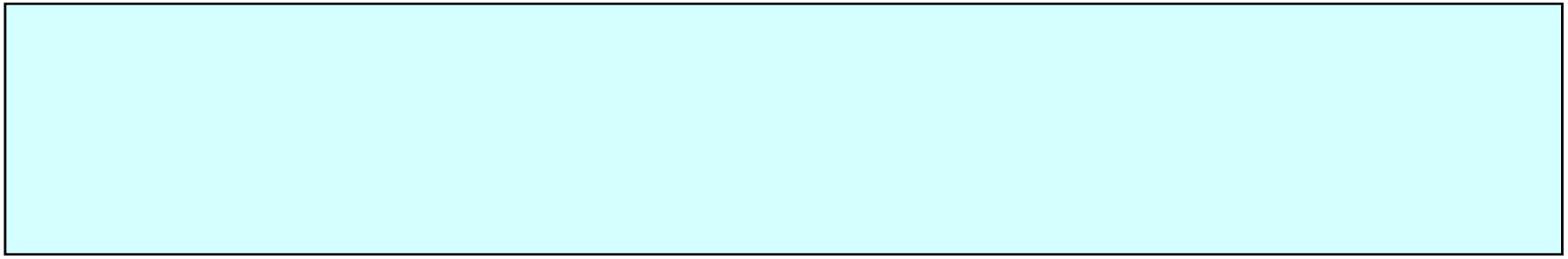
- Exemple : masse salariale de l'entreprise

```
context Entreprise :: masseSalariale():Real
post : result =  employé->iterate(p : Personne ;
                                  ms : Real=0 | ms+p.contrat.salaire)
```

Exercice

- Ecrivez, dans le contexte de la classe Collection, l'opération size à l'aide de l'opération iterate

- Ecrivez, dans le contexte de la classe Collection, l'opération forAll à l'aide de l'opération iterate.



Opérations sur toutes types de collection

Evaluation unique

- L'opération suivante vaut vrai si et seulement si uneExpression s'évalue avec une valeur distincte pour chaque élément de la collection

```
isUnique(uneExpression : BooleanExpression) : Boolean
```

sortedBy(uneExpression) : Sequence

- *L'opération retourne une séquence contenant les éléments de la collection triés par ordre croissant suivant le critère décrit par une certaine expression.*
- *Les valeurs résultant de l'évaluation de cette expression doivent donc supporter l'opération <.*

```
sortedBy(uneExpression) : Sequence
```

- Exemple: les salariés, ordonnés par leur salaire

```
context Entreprise :: salariesTries() : orderedSet(Personne)
post :  result = self.employe->sortedBy(p | p.contrat.salaire)
```

Opérations communes... mais spécifiques

- ***L'opération select***
  - *est commune aux trois types de collection concrets, n'est pas définie dans Collection, mais seulement dans les types concrets avec des signatures spécifiques :*

    ```
    select(expr:BooleanExpression): Set(T)
    select(expr:BooleanExpression): Bag(T)
    select(expr:BooleanExpression): Sequence(T)
    ```

  - *Pour simplifier*

    ```
    select(expr:BooleanExpression): Collection(T)
    ```

Opérations communes… mais spécifiques

❒ Sélection et rejet d'éléments

- *Deux opérations retournent une collection du même type construite par sélection des éléments vérifiant (resp. ne vérifiant pas) une certaine expression.*

select(uneExpression : BooleanExpression) : Collection(T)

reject(uneExpression : BooleanExpression) : Collection(T)

❒ Exemple : expression représentant dans le contexte d'une entreprise les employés âgés de plus de 60 ans :

```
context Entreprise …
…  self.employé->select(p : Personne | p.age>=60)  …
```

Opérations communes... mais spécifiques

- **L'opération collect retourne une collection composée des résultats successifs de l'application d'une certaine expression à chaque élément de la collection.**

```
collect(expr:BooleanExpression): Collection(T)
```

- **Exemple :**
  - *Les trois manières standards d'atteindre les mots d'un texte s'écrivent ainsi dans le contexte d'un texte*

```
self.occurrence->collect(mot)
self.occurrence->collect(o | o.mot)
self.occurrence->collect(o : Occurrence | o.mot)
```

- cette notation admet un raccourci :
  - *self.occurrence.mot*

Opérations communes... mais spécifiques

Ajout/retrait d'éléments

- *Ces deux opérations retournent une collection résultant de l'ajout (resp. du retrait) d'un certain objet à la collection.*

```
including(unObjet : T) : Collection(T)
excluding(unObjet : T) : Collection(T)
```

Union de deux collections

```
union(c : Collection) : Collection
```

Opérations de conversion

- Chaque sorte de collection (set, orderedSet, sequence ou bag) peut être transformée dans n'importe quelle autre sorte.

- Par exemple un bag peut être transformé en :
  - séquence par l'opération asSequence():Sequence(T)
    - L'ordre résultant est indéterminé
  - ensemble par l'opération asSet():Set(T)
    - Les doublons sont éliminés
  - ensemble ordonné par l'opération asOrderedSet():OrderedSet(T)
    - Les doublons sont éliminés ; l'ordre résultant est indéterminé

- Illustration : il y a plus de 2 occurrence d'un certain mot :

```
context Texte inv:
  self.occurrence->collect(mot)->asSet()->size() >= 2
```

Opérations propres à Set et Bag

- L'intersection est une opération commune aux sets et aux bags

```
context Set :: intersection (s : Set(T)) : Set(T)

post : result -> forAll(elem | self -> includes(elem) and
                               s -> includes(elem) )

post : self -> forAll(elem | s -> includes(elem) =
                             result -> includes(elem) )

post : s -> forAll(elem | self -> includes(elem) =
                             result -> includes(elem) )
```

Opérations liées à l'ordre (OrderedSet, Sequence)

- Certaines opérations sont propres à la manipulation des collections dont les éléments sont ordonnés

```
append(unObjet):Sequence(T) ; ajout d'un objet à la fin
prepend(unObjet):Sequence(T) ; ajout d'un objet au début
```

- partie d'une collection ordonnée délimitée par deux indices

```
subSequence(inf:Entier, sup:Entier):Sequence(T); spécifique des
   séquences
subOrderedSet(inf:Entier, sup:Entier):OrderedSet(T); spécifique
   des ordered sets
```

- D'autres opérations du même genre

```
at(inf:Entier):T
first():T
last():T
insertAt(i:Integer, o:T):Sequence(T)
indexOf(o:T):Integer
```

Compléments

allInstances()

- S'applique à une classe et non un objet.
- Retourne l'ensemble de toutes les instances d'une classe et de ses sous-types.
  - allInstances() : Set(T)

Itérateur isUnique(expr)

- Retourne vrai si chaque évaluation du corps pour les éléments de la collection source produit un résultat différent, vrai sinon.

```
c->isUnique(expr: OclExpression): Boolean
-- true if expr evaluates to a different value
-- for each element in c.
post: result = c->collect(expr)->forAll(e1, e2 | e1 <> e2)
```

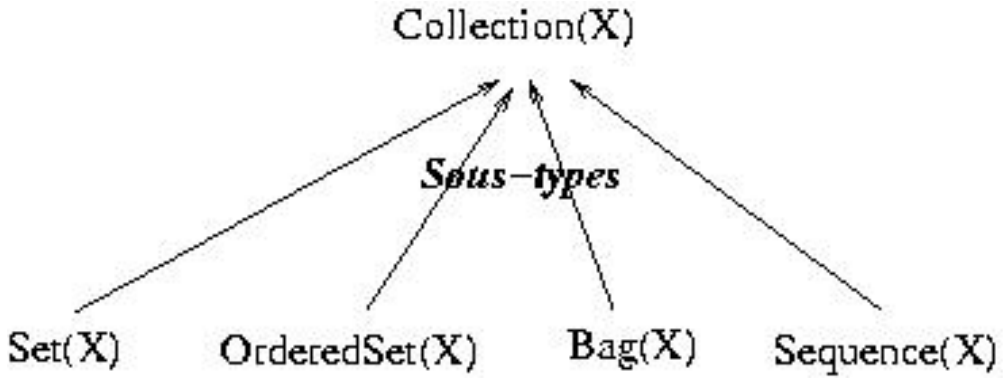
Exercice

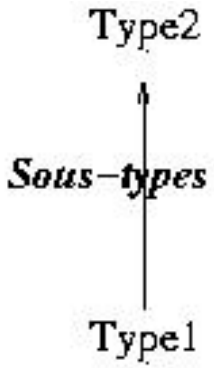
- **Quelle est la signification de cette expression ?**

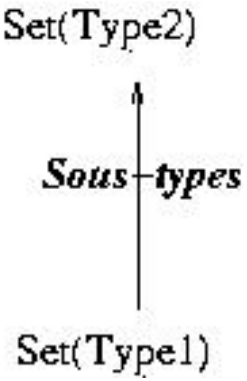
```
context Personne inv:

Personne.allInstances()->forAll(p1, p2 |

          p.1 <> p2 implies p1.nom <> p2.nom )
```

- **Comment l'écrire avec isUnique ?**

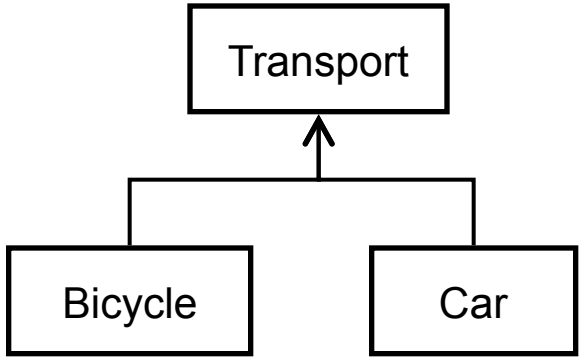
Conformance de type







- Sous-typage
  - **Set(Bicycle) conforme à Set(Transport)**
  - **Set(Bicycle) conforme à Collection(Bicycle)**
  - **Set(Bicycle) conforme à Collection(Transport)**
  - Set(Bicycle) non conforme à Bag(Bicycle)



Limites d'expressivité d'OCL

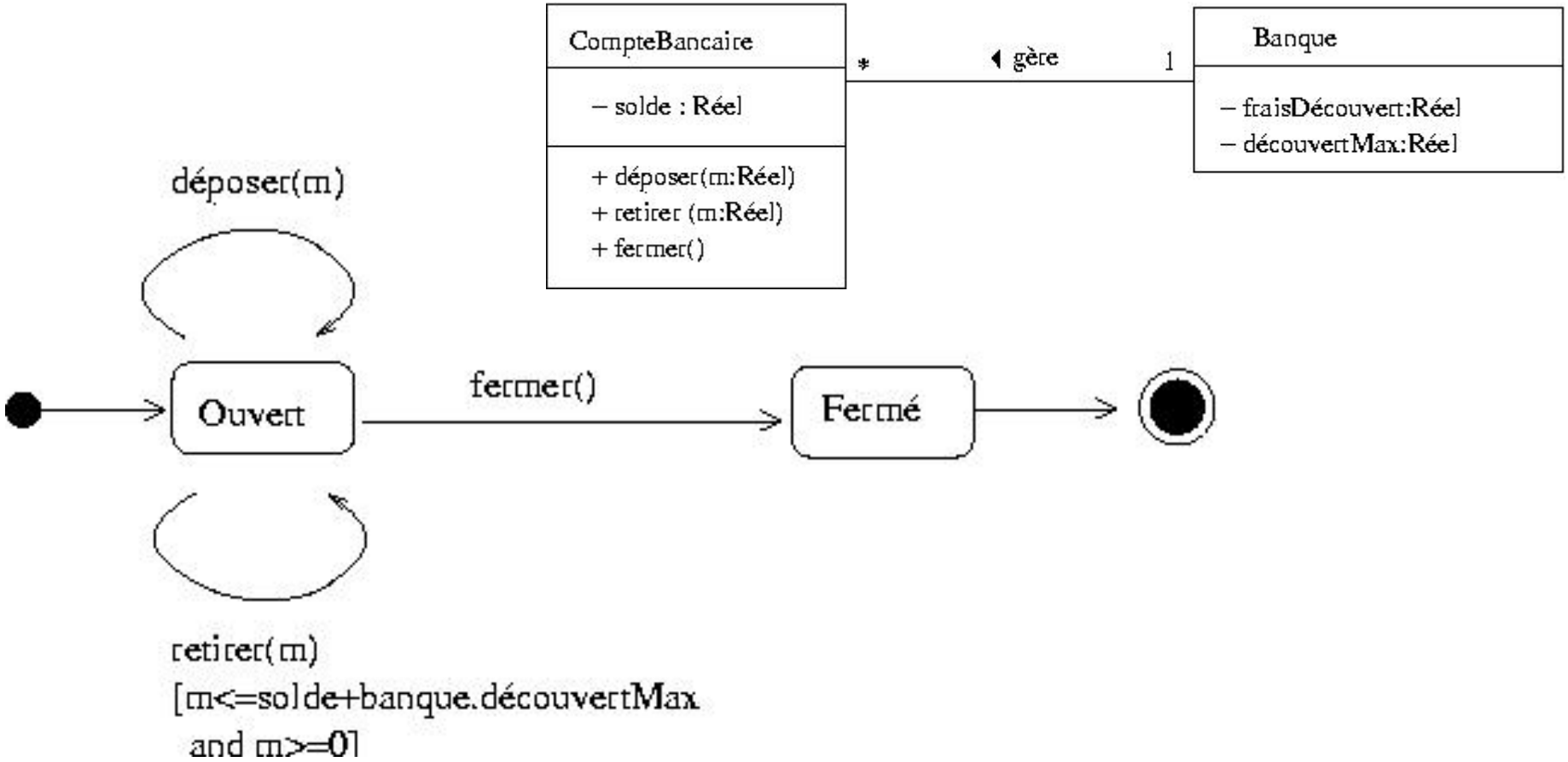
- **Limites de la logique des prédicats**

- **Comment les contourner ?**

- **Check List**
  1. Manque-t-il une primitive à ma classe ?
     - Défaut de conception
     - Ajouter la primitive (avec « def » si on ne peut toucher à la classe)
  2. Manque-t-il un moyen d'exprimer une propriété importante ?
     - Défaut de d'expressivité d'OCL
     - Ajouter une primitive qui représente tout ou partie de la propriété recherchée
     - Spécifier cette primitive, même partiellement

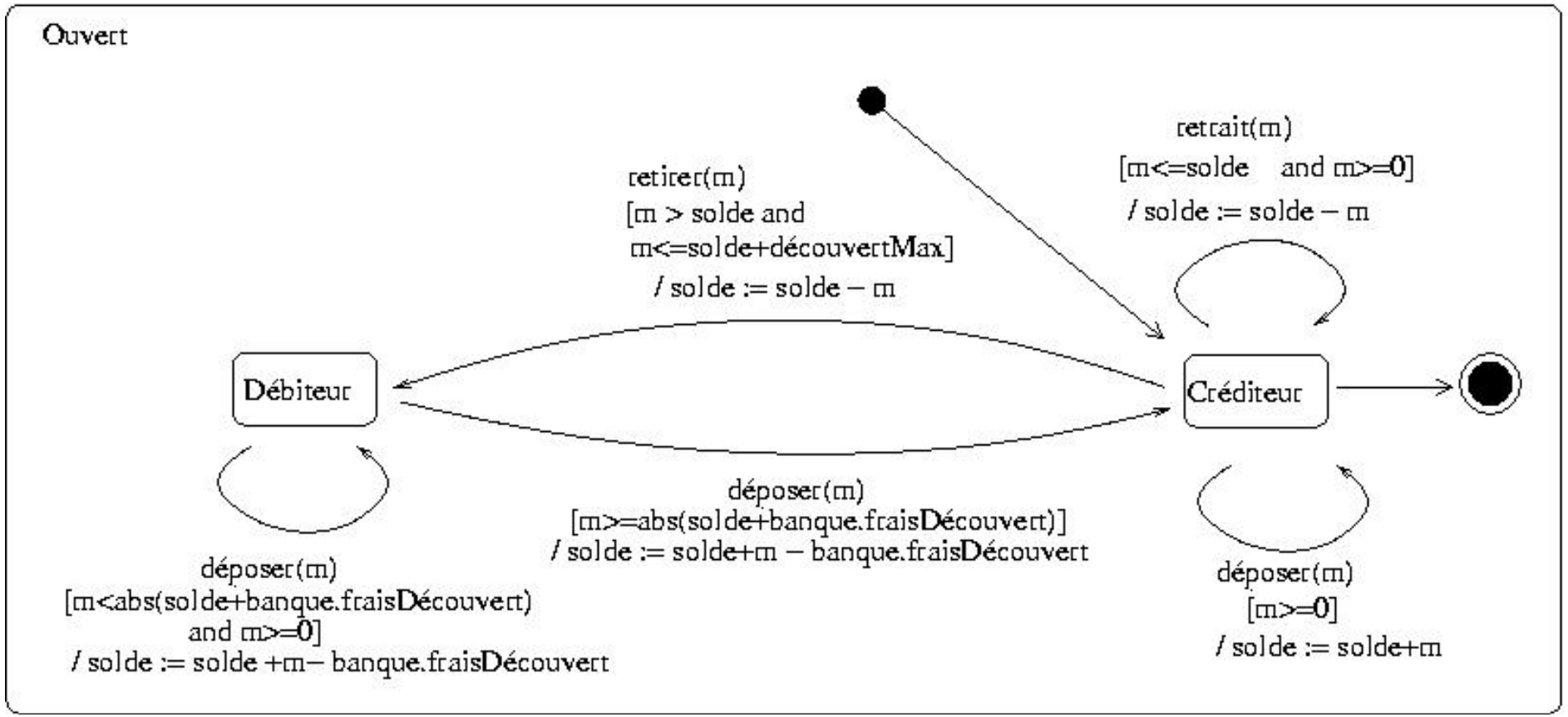
Exploitation d'ocl : Garde

- **Utilisation d'OCL pour exprimer des gardes dans les diagrammes d'automates à états finis :**
  - Diagrammes d'états
  - Diagrammes d'activités



Actions

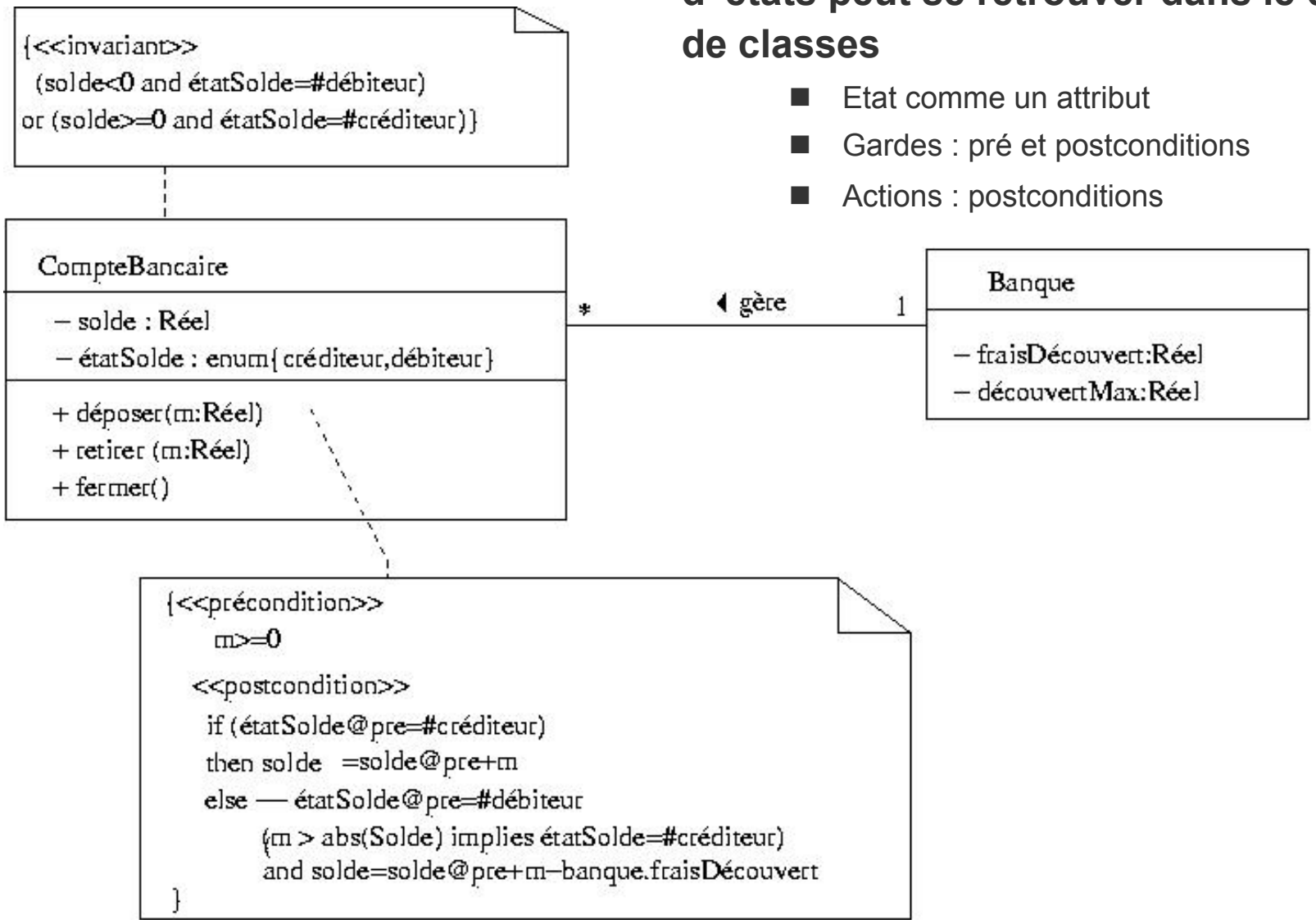
- **Utilisation dans la description des actions qui étiquettent les transitions**
  - OCL est utilisé comme langage de navigation
  - On n'écrit pas forcément une expression booléenne



Redondance des informations

❒ **Une partie des informations du diagramme d'états peut se retrouver dans le diagramme de classes**

- Etat comme un attribut
- Gardes : pré et postconditions
- Actions : postconditions



V& V, et contrats

- V&V : introduction
- Approche par contrats

Définitions

- Validation : *construisons-nous le bon produit ?*
- Vérification : *le construisons-nous bien ?*

Pour vérifier, il faut une spécification précise, si possible formelle, du fonctionnement du logiciel

Applications

- Validation : *recherche des défauts de conception*
  - essentiellement un technique de tests dynamiques, fonctionnels (en boîte noire)
- Vérification : *des erreurs* des développeurs
  - à partir cahier des charges (ou spécifications précises)
  - Ce qui permet plusieurs techniques : revues, inspections, preuves, analyses statiques et dynamiques, tests fonctionnels et structurels, en boîte noire ou blanche...

Techniques statiques et dynamiques : complémentarité

❒ Techniques statiques

- Elles portent sur des documents (en particulier les programmes), *sans exécuter le logiciel*
- Avantages :
  - contrôle systématique valable, pour toute exécution, applicable à tout document
- Inconvénients :
  - ne portent pas forcément sur le code réel (qui peut évoluer) ni l'environnement réel
  - Sauf pour les preuves, les vérifications statiques sont sommaires
  - Les preuves complètes sont difficiles et longues et nécessitent des spécifications formelles et complètes ... également difficiles

❒ Techniques dynamiques

- Elles nécessitent une exécution du logiciel, une parmi des multitudes d'autres possibles
- Avantages :
  - contrôle qui porte sur des conditions proches de la réalité
  - plus à la portée du commun des programmeurs
- Inconvénients :
  - il faut provoquer des expériences, donc écrire du code et construire des données d'essais
  - **un test qui réussit ne prouve pas qu'il n'y a pas d'erreurs**

Approche par contrats

- **L'idée de contrat a été introduite en 1988 par Bertrand Meyer, l'auteur du langage Eiffel**

- **L'intérêt d'un contrat est de bien préciser les responsabilités entre les acteurs d'une conception, d'un développement logiciel**

- **Par abus de langage, on parle parfois de contrats entre classes ou composants : ce sont en fait des contrats entre les utilisateurs des classes, des composants**

- **Un contrat définit explicitement les droits et les devoirs de chaque intervenant dans la conception et l'utilisation d'une unité**
  - aucun contrôle ne manque
  - aucun contrôle n'est superflu

Types de contrats

- **Contrat de** clientèle
  - entre les utilisateurs externes des services (primitives exportées, publiques) d'une classe, les **clients**, et l'auteur de cette classe **fournisseur**

- **Contrat** d'héritage
  - entre les (re)utilisateurs par adaptations
  - dans des classes héritières, **descendantes** et l'auteur de la classe héritée, la classe ancêtre.

Contrat de clientèle : exemple

```
Context MathUtil:: sqrt (x, eps: Real): Real
-- Square Root of x with precision eps
pre: x >= 0
pre: eps >= 10^-6
post: abs(result^2 - x) <= 2*eps*result
```

	Devoirs	Droits
client	Appeler sqrt uniquement si la précondition est satisfaite	Obtenir le résultat correct avec la précision attendue, spécifiée dans la postcondition
fournisseur	Rendre le résultat correct avec la précision attendue, spécifiée dans la postcondition	Refuser le calcul (dégager sa responsabilité) si la précondition n'est pas satisfaite

Assertions et héritage

- **Une classe héritière hérite de tous les invariants de ses classes parentes**
  - Ils sont tous conjugués par des « and »

- **Lorsqu'il y a redéfinition de méthodes lors de l'héritage, on peut avoir** nouvelles pré/post conditions
  - **précondition moins exigeante, moins forte**
  - **postcondition plus précise, plus forte**

Les préconditions moins exigeantes, c'est parfois contre intuitif !

```
context NewMathUtil:: sqrt (x, eps: Real): Real
pre: eps >= 10^-9
```

Contrats d'héritage

	Devoirs	Droits
Classe parente	Fournir aux héritiers des services fiables, performants et réutilisables	Les héritiers ne doivent pas déformer la pensée initiale Invariants = code génétique
Classe héritière	Respecter les invariants Redéfinir en respectant les règles pre/post	La classe parente doit fournir des services de qualité. Les assertions initiales doivent être cohérentes

Annexe : Opérations de manipulation des collections

- **Les opérations de collection utilisent l'opérateur ->**
  - (tous les indices commencent à 1) :
  - = <> Collections identiques ou différentes
  - append(obj) Rend la valeur de la collection ordonnée avec obj ajouté à la fin
  - asSet(), asOrderedSet(), asSequence() Conversion de type entre collections
  - at(idx) Rend l'objet à l'indice idx dans une collection ordonnée
  - count(obj) Nombre d'apparition de obj dans une collection
  - excludes(obj) Rend count(obj) = 0 ?
  - excludesAll(coll) Est-ce que tous les éléments de coll satisfont count(obj) = 0 ?
  - excluding(obj) Rend la valeur de la collection sans l'objet obj
  - first() Rend le premier élément d'une collection ordonnée

Annexe : Opérations de manipulation des collections

- includes(obj) Rend count(obj) > 0 ?
- includesAll(coll) Est-ce que tous les éléments de coll satisfont count(obj) > 0 ?
- including(obj) Rend la valeur de la collection avec l'objet obj ajouté
- indexOf(obj) Rend l'indice de la 1ère occurrence de obj dans une séquence
- insertAt(idx,obj) Rend la valeur de la collection avec obj ajouté à l'indice idx
- intersection(coll) Intersection des collections non ordonnées self et coll
- isEmpty() Rend size() = 0 ?
- last() Rend le dernier élément d'une collection ordonnée
- notEmpty() Rend size() > 0 ?
- prepend(obj) Rend la valeur de la collection ordonnée avec obj ajouté en tête
- size() Nombre d'éléments dans la collection
- subOrderedSet(start,end) Sous-partie d'un ensemble ordonné entre les indices
- subSequence(start,end) Sous partie d'une séquence extraite entre les indices
- sum() Somme des éléments d'une collection (Integer ou Real)
- union(coll) Union des collections self et c

Annexe : opérations collectives

- collect(expr) Rend un bag contenant la valeur de exp appliquée à chaque élément de la collection
  - maCollection.valeur est équivalent à maCollection->collect(valeur)
- exists(expr) Est-ce l'expression expr est satisfaite pour l'un des éléments de la collection ?
- forAll(expr) Est-ce l'expression expr est satisfaite pour tous les éléments de la collection ?
- isUnique(expr) Rend vrai si expr donne une valeur différente pour chaque élément de la collection
- iterate(i : Type; a : Type | expr)
  - Opération à partir de laquelle les autres sont définies. La variable « i » est l'itérateur et « a » est l'accumulateur, qui récupère la valeur de expr après chaque évaluation
- one(expr) Rend l'expression coll->select(expr)->size()=1
- reject(expr) Rend la sous-collection des éléments pour lesquelles expr n'est pas satisfaite
- select(expr) Rend la sous-collection des éléments pour lesquelles expr est satisfaite
- sortedBy(expr) Rend la séquence contenant tous les éléments de la collection, ordonnés par la valeur
  - expr (le type des éléments doit posséder un opérateur « < »)