UNSA
Université de Nice Sophia-Antipolis

UFR Sciences
Département Informatique

Licence d'informatique – Module génie logiciel

Spécification du logiciel

Roger Rousseau
CNRS-UNSA/I3S

Version 1.1 *Mars 2005*

Eléments de Génie Logiciel 2007-2008 : cours dispensé par Ph. Collet

Plan du cours

Bibliographie résumée (1)

1) **James Rumbaugh, Ivar Jacobson, Grady Booch**
"The Unified Modeling Language Reference Manual"
Addison-Wesley, 1999, 550 p.
(le manuel de référence UML).

2) **Jos Warmer et Anneke Kleppe**
"The Object Constraint Language: Precise Modeling with UML"
Addison-Wesley, 1999, 144 p.
(le manuel de référence OCL).

3) **Desmond F. D'Souza et Alan C. Wills**
"Object, Components and Frameworks with UML: The Catalysis Approach"
Addison-Wesley, 1998, 816 p.
(une méthode de conception avec UML).

4) **Pascal André et Alain Vailly**
"Spécification des logiciels
deux exemples de pratiques récentes : Z et UML"
Ellipses, 2001, 317 p.
(ouvrage accessible sur les spécifications).

Bibliographie résumée (2)

1) **Bertrand Meyer** *"Object Oriented Software Construction"*, 2 e édition, Prentice-Hall, 1997, 1254 p.
(la « bible » sur Eiffel et sa méthode)

2) **Bertrand Meyer** *"Eiffel : the language"*, Prentice-Hall, 1991, 1254 p. *(définition du langage Eiffel)*

3) **Bertrand Meyer** *"Introduction to the Theory of Programming"*, Prentice-Hall, 1992
(à voir pour les spécifications)

4) **Kim Waldén & Jean-Marc Nerson** *"Seamless O-O Software Architecture Analysis and Design"* Prentice-Hall, 1994, 302 p.
(Méthode de conception OO « BON »)

5) **Groupe LMO** (Langages et Modèles à objets)
École internationale d'été du CIMPA, Nice, juillet 1996 :
Langages et Modèles à objets, R. Ducournau, J. Euzénat, G. Masini, A. Napoli (Eds), INRIA, Collection Didactique, 1998, ISBN 2-7261-1131-9
Synthèse des différentes approches à objets en 1996

Bibliographie résumée (3)

1) **Dijkstra, E.W.** *"A discipline of programming"*, Prentice Hall, 1976.
(programmation réfléchie à partir de spéc.)

2) **Gries, David** *"The Science of Programming"*, Springer Verlag, 1981
(programmation réfléchie à partir de spéc.)

3) **Jones, C.B.** *"Systematic Software Development using VDM"*, Prentice Hall, 1986.
(programmation réfléchie à partir de spéc.)

4) **Morgan, Carroll** *"Programming from Specifications"*, Prentice Hall, 1990, 2nd edition 1994.
(programmation réfléchie à partir de spéc.)

5) **Wirth, Niklaus** *"Systematic Programming : An Introduction"*, Prentice Hall, 1973.
(programmation réfléchie à partir de spéc.)

Bibliographie résumée (4)

1) **Gries, D & Schneider, F.B.** *"First Order Logic and Automated Theorem Proving"*, Springer, 1995
(bases théoriques pour les preuves automatiques)

2) **Hoare, C.A.R.** *"An axiomatic basis for computer programming"*, Comm. ACM, 1969, Vol 12, Num 10
(programmation réfléchie à partir de spéc.)

3) **Jackson, M. & Zave, Pamela** *"Where do operations come from ? A multiparadigm spec technique"*, IEEE Trans. on S.E., 1996
(technique de spec multi-paradigmes)

4) **Lano, Kevin** *"Formal Object-Oriented Development"*, Springer Verlag, 1995.
(programmation réfléchie à partir de spéc.)

5) **Manna, Z. & Pnueli, A.** *"The temporal logic of reactive and Concurrent Systems : specification"*, Springer Verlag, 1992.
(bases théoriques pour la spécification d'aspects concurrents.)

Chapitre 1

Spécification fonctionnelle : une introduction

1) **Problématique des spécifications**
2) **Spécification en langage naturel**
3) **Spécification structurée en langage naturel**
4) **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**

Spécifications extrafonctionnelles

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 .

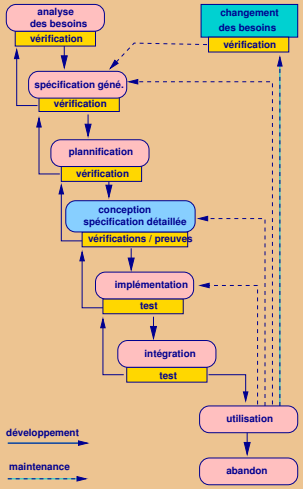
Quand spécifier ?

Vu la difficulté, il vaut mieux se limiter à des logiciels simples :

*contrairement au modèle de la cascade, les **spécifications doivent suivre** (ou s'intégrer) à la phase de conception / décomposition :*

ex. méthode Catalysis.

Quand spécifier ? suite(1)



Principaux formalismes de spécification

Texte informel

En langage naturel (cahier des charges, commentaires),

éventuellement encadré par une méthode ou un formalisme graphique (contraintes UML).

Principaux formalismes de spécification

Graphique

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

Principaux formalismes de spécification

suite(1)

Semi-formel

Sans ambiguïté, leur expressivité est insuffisante pour établir une **preuve**,
mais on peut souvent les tester :

- techniques assertionnelles, spécification axiomatique

Principaux formalismes de spécification

suite(2)

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).

Spécification vs Implémentation

implémentation

doit correspondre aux fonctionalités 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(1)

En simplifiant,

- *spécifier, c'est définir*
  sans se soucier des contingences du monde réel...
- *implémenter, c'est optimiser, réifier* $\Rightarrow$
  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(2)

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 décrois. :

$desc_1 \sqsubseteq desc_2 ... \sqsubseteq desc_n$

- la $desc_1$, de plus haut niveau d'abstraction,
  est la **spécification**,
- la $desc_n$, de plus bas niveau d'abstraction,
  est **l'implémentation,**,

Spécification vs Implémentation suite(3)

- toutes les descriptions sont liées par deux relations :
  - **abstraction** :
    $desc_i = abstract(desc_{i+1}) \Leftrightarrow desc_i \sqsubseteq desc_{i+1}$
  - **raffinement** :
    $desc_i = refine(desc_{i-1}) \Leftrightarrow desc_i \sqsupseteq desc_{i-1}$

Il existe une théorie qui formalise ces relations : refinement calculus [Morgan 92, Sekerinsky 94, Back & Wright 98]

Spécification, Formalité et Exécutabilité

Deux sens au mot *formel* :

1) *formel* = non ambigu, précis, sans équivoque
   $\Rightarrow$ syntaxe et sémantique précise, interprétable par une machine
2) *formel* = privilégie la forme au détriment du contenu
   $\Rightarrow$ 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).

Spécification, Formalité et Exécutabilité

suite(1)

Exemple :
logique formelle, avec des formules, par oposition 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(2)

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 ?

Spécification, Formalité et Exécutabilité

suite(3)

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.*

Utilisation des spec. fonctionnelles : 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 (ou ses instigateurs, commanditaires).

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.

Utilisation des spec. fonctionnelles suite(1)

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 ?
  $\Rightarrow$ l'un des gros problèmes des spécifications formelles, lorsqu'elles sont incompréhensibles aux commanditaires ou aux utilisateurs.

Utilisation des spec. fonctionnelles suite(2)

- **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...)

Spécification en langage naturel suite(1)

- utilisation d'une syntaxe pour les aspects structurels, les signatures de méthodes... $\Rightarrow$ 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.

Spécification en langage naturel : avantages

- (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).

Spéc. en lang. naturel : inconvénients

- 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 (e.g. projet Ada, rôle de l'Internet, durée d'un produit)
- utilité d'une description formelle pour rectifier une description informelle.

Défauts évidents de l'exemple de la file

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

légende

lourdeur de style, repentir, verbiage
sur-spécification
changement de terminologie

Quelques défauts évidents de l'exemple de la file suite(1)

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.

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) 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` *)*

Exemple en langage naturel structuré suite(1)

`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`

Exemple en langage naturel structuré suite(2)

`retirer`

-- *retire l'élément situé en tête*

`longueur : Naturel`

-- *nb d'éléments (hérité de* `Conteneur`*)*

`vide : Booleen`

-- `longueur` > *0 (hérité de* `Conteneur`*)*

Exemple de spécification structurée en langage naturel suite(3)

sous-entendus : self, prédicats faux, hiérarchie d'héritage...

Typiquement, c'est ce que l'on fait en UML (sans utiliser OCL).

Cette spécification est un progrès, mais manque encore de précision.

Elle sera améliorée avec les techniques suivantes...

Principales 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

Principales techniques formelles suite(1)

- 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...*

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*.

Spécification algébrique : principes

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.* Cela facilite la combinaison des formules dans les axiomes.

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 suite(1)

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

Spécification algébrique suite(2)

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).

Les erreurs sont indiquées par différents procédés : *préconditions, valeur spéciale notée "ϵ", universelle et transitive...*

Spéc. algébrique d'une file infinie (v1)

type File <E>

primitives

-- *générateurs*

créer : $\longrightarrow$ File <E>

déposer : File <E> $\times$ E $\longrightarrow$ File <E>

-- *accesseurs*

vide : File <E> $\longrightarrow$ Booléen

tête : File <E> $\not\longrightarrow$ E

-- *modifieurs*

retirer : File <E> $\not\longrightarrow$ File <E>

Spéc. algébrique d'une file suite(1)

préconditions

$\forall$ f: File <E>

(p1) retirer(f) **requiert** $\neg$ vide (f)

(p2) tête(f) **requiert** $\neg$ vide (f)

Spéc. algébrique d'une file suite(2)

axiomes

$\forall$ e: E, f: File <E>

(a1) vide(créer())

(a2) $\neg$ vide(déposer(f,e))

(a3) tête(déposer(créer(),e)) = e

(a4) tête(déposer(f,e)) = tête(f)

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

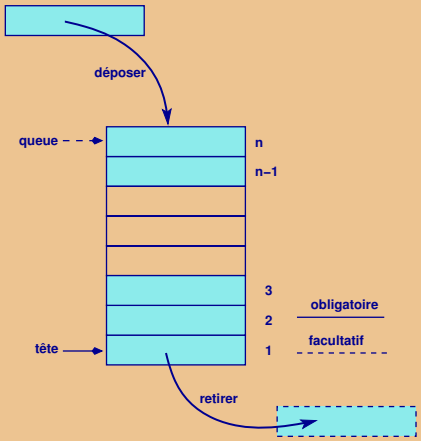
(a6) retirer(déposer(f,e)) =
déposer(retirer(f),e)

Qualités de la version 1

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

Schéma d'une file

mais au fait ?
qu'est-ce qu'une file ?



Défauts de la version 1

- n'explicite pas deux propriétés cachées incontournables des files :
  - *nombre d'éléments*,
  - *accès à la valeur de chaque élément (par indexation)*

  $\Rightarrow$ mais on peut le faire...
- ne fait pas apparaître de relation d'héritage :
  `Container, Dispenser, Linear, Collection`...

Défauts de la version 1 suite(1)

- pas de commentaires en langage naturel :
  $\Rightarrow$ on doit le faire (absents sur transparents)
- séparation des informations qui marchent ensemble : syntaxe, sémantique, commentaire...

Spéc. algébrique d'une file v2

type `File <E>`

primitives

-- générateurs

`créer :` $\longrightarrow$ `File <E>`

`déposer :` `File <E>` $\times$ `E` $\longrightarrow$ `File <E>`

-- accesseurs

`vide :` `File <E>` $\longrightarrow$ `Booléen`

`tête :` `File <E>` $\not\longrightarrow$ `E`

-- nouveautés :

`longueur :` `File <E>` $\longrightarrow$ `Naturel`

`élément:` `File <E>` $\times$ `Naturel` $\not\longrightarrow$ `E`

Spéc. algébrique d'une file v2 suite(1)

-- modifieurs

retirer : File <E> ⟶/ File <E>

préconditions

∀ f: File <E>, i: Naturel, e: E

(p1) retirer (f) **requiert** ¬ vide (f)

(p2) tête (f) **requiert** ¬ vide (f)

-- nouveautés :

(p3) élément (f, i) **requiert**

¬ vide (f) ∧ 1 ≤ i ≤ longueur

Spéc. algébrique d'une file v2 suite(2)

...

axiomes

∀ e: E, i: Naturel, 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)

Spéc. algébrique d'une file v2 suite(3)

-- nouveautés :

(a7) longueur(créer()) = 0

(a8) longueur(déposer(f,e)) = 1+longueur(f)

(a9) élément(déposer(f,e),i) =

si i = longueur(f)+1

alors e **sinon** élément(f,i)

Remarque (1)

La définition de `vide`
peut maintenant être réécrite avec `longueur` *:*

axiomes

-- (a1) vide (créer())

-- (a2) ¬ vide (déposer(f, e))

(a1b) vide (f) = (longueur(f) = 0)

Remarque (2)

On peut éviter les tests conditionnels par des analyses cas par cas et par l'utilisation d'implications logiques :

```
(a9b) élément(déposer(f,e),
                longueur(f)+1) = e
(a9c) i ≠ longueur(f)+1 ⇒
    élément(déposer(f,e),i) = élément(f,i)
```


Remarque (3)

On pourrait aussi définir élément à partir de retirer :

```
(a9e) élément(f,i) =
      si i = 1
        alors tête(f)
        sinon élément(retirer(f),i-1)
```

Mais pour avoir une méthode de spécification systématique, il vaut mieux définir les axiomes sur des états obtenus par les générateurs.

Bilan des spécifications algébriques

- comme en programmation, il y a plusieurs manières possibles de spécifier le même problème...
- spécification algébrique est une technique orientée programmation fonctionnelle, et est 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 (Ada) et par objets (Eiffel, C++, Java5...),

Bilan des spécifications algébriques suite(1)

- 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 (Larch)
  ⇒ mais il faut les aider, et plutôt lents...
- Conformité du code ? *preuves de théorèmes*
  et une adaptation aux différents langages (Larch-Ada, Larch C++...)

Bilan des spécifications algébriques suite(2)

- Accessibilité aux programmeurs ?
  difficulté de construction et de compréhension par la majorité des programmeurs, et en cas de logiciel nécessitant des milliers d'axiomes
- Les techniques très formelles sont peu prisées par les industriels américains, mais le sont davantage en Europe, notamment en G.B.

Avantages des Spécifications 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,

Avantages des Spécifications formelles suite(1)

- précision :
  - meilleure compréhension :
    « dans un dialogue, il est utile que l'un au moins des deux interlocuteurs sache de quoi il parle... »
    (Claude Pair, 1977)
  - définition de référence pour l'implémenteur
- abstraction :
  - concision,
  - description plus stable, indépendante des techniques d'implémentation, plus réutilisable

Inconvénients des Spécifications formelles

- difficultés :
  - manque de lisibilité
  - aptitudes mathématiques nécessaires
  - erreurs possibles, si preuves humaines

Inconvénients des Spécifications formelles

suite(1)

- manque de maturité :
  - nombreux formalismes, mais standards (CASL)
  - 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 indispensables pour les logiciels où la fiabilité est critique

Chapitre 2

Spécification par assertions exécutables

1) **Introduction aux assertions**
2) **Assertions exécutables non quantifiées**

Introduction aux 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

Introduction aux assertions suite(1)

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.

Introduction aux assertions suite(2)

Une **assertion** (ou *affirmation*, *contrainte*),
est une formule de logique des prédicats
qui « **doit** » être vraie en un **point précis**
de l'**espace-temps** d'un programme
(ou d'un schéma de conception),
pour toute exécution.

Espace : une assertion est attachée à un contexte précis du programme.

Temps : une assertion doit être vraie à des moments précis du processus d'exécution.

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...

Termes d'une assertion suite(1)

- d'**états** des variables d'un programme (attributs) : *programmation impérative*,
- d'enchaînement des **événements** : *logique temporelle, programmation concurrente, réactive...*
- de **valeurs** (immutables) : typage, *de nature mathématique*,
- d'**organisation** d'un logiciel : relations, architecture, *utile en conception*,
- des **processus logiciel** (de construction) : *génie logiciel*,

Exemple d'assertion : class Account

Vue abstraite (boîte noire) :

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

⇒ *ici intégrée au texte du programme*

Exemple d'assertion : class Account suite(1)

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
...
```


Exemple d'assertion : class Account suite(2)

Vue abstraite (boîte noire) :

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

⇒ *ici externe au texte du programme*

Assertion 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

⇒ *code autonome, autodocumenté, autovérifiable, auto-testable*

Assertion et langage suite(1)

- définies dans un langage compagnon, mais séparées des descriptions : *contraintes en langage OCL pour UML*
  Assertions pour plates-formes de composants : CCL-J de ConFract...
- 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.

Redondance des assertions suite(1)

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 d'une assertion

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 difficile pour les assertions quantifiées : $\exists$, $\forall$

Assertion exécutable

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 :
  déclenchement d'une exception, par défaut un diagnostic
  $\Rightarrow$ *possibilité de « programmation défensive »*.

Assertion exécutable suite(1)

- des diagnostics qui peuvent être très précis :
  type, instance, méthode, type d'assertion, identifiant de la formule (label, numéro)... clause, numéro de ligne...

$\Rightarrow$ **Le deboguage devient rarement utile**

Utilisation des assertions

- technique de spécification :
  formelle, partielle, incrémentale,
- conception et programmation par contrats :
  établissement clair des responsabilités entre les clients et les programmeurs,
- aide à la fiabilité :
  pour la correction vs exceptions pour la robustesse,
- technique de preuve :
  méthode axiomatique de Hoare, Dijkstra, Morgan, Gries, refinement calculus...

Utilisation des assertions suite(1)

- 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*.
- processus logiciel : *intégrées au logiciel, possibilités de traçabilité, rétroconception, métamanipulations*
- support de méthodes de conception : *Catalysis avec OCL*

Utilisation des assertions suite(2)

⇒ **consensus sur l'intérêt des assertions** ...

⇒ chaque nouveau paradigme de programmation ou de conception s'intéresse aux assertions :
design pattern, programmation générative, séparation des préoccupations (aspects, sujets...), composants, etc.

... mais pratique les assertions sont encore peu utilisées, car les langages les plus utilisés en sont officiellement dépourvus : C, C++, Java, Ada...

Classification des assertions

Deux points de vue :

- **physique, syntaxique** : selon l'endroit de l'assertion dans la description, qui indique à quels moments l'assertion doit être vraie :
  précondition, postcondition, invariant d'instance, invariant d'itération, assertion d'étape d'un calcul...
- **logique, sémantique** : selon l'intention des concepteurs ou des programmeurs, la généralité de la propriété, son intérêt dans le processus logiciel, le degré d'importance... :
  axiome, théorème, définition, lemme, invariant, antécédent, conséquent, contrôle, contrainte d'intégrité...

Classification des assertions suite(1)

On peut utiliser des *stéréotypes* ou des *mots-réservés* pour distinguer toutes ces catégories.

L'intérêt des catégories d'assertions est de leur associer des rôles dans le processus logiciel et des techniques d'évaluation adaptées.

Assertion et Objet

Les assertions sont bien adaptées à l'approche objet :

- état, évolution,
- connaissance partielle,
- définitions incrémentales, factorisables
- règles pour l'héritage et les redéfinitions de méthodes,

Assertion et Objet suite(1)

L'approche par objets (persistants) est utile aux assertions :

- réification : *états d'une assertion, classes, méthodes de manipulation*,
- métamanipulation : *système d'armement...*
- accès aux extensions : *des types ou autres collections (quantifications...),*
- lien avec les systèmes persistants : *pour les contraintes d'intégrité...*

Assertion exécutable : niveau Eiffel

Le langage Eiffel a joué un rôle de pionnier pour les assertions exécutables, intégrées au langage, embarquées dans les composants.

Les assertions d'Eiffel ne sont pas quantifiées.
mais il existe des extensions avec quantifications (Oqual).

Dans la suite, la syntaxe utilisée est inspirée d'Eiffel, OCL, Java, Oqual et est sans importance...

Catégories physiques d'assertion

Eiffel distingue 5 catégories (physiques) d'assertions:

- **assertion externe:**
  perceptible aux utilisateurs des classes (clients ou héritiers)
  - precondition de méthode (fonction ou procédure):
    precondition, **require**, **pre:**,
  - postcondition de procédure (opération, action):
    postcondition, **ensure**, **post:**
  - invariant d'instance :
    invariant, **inv:**

Catégories physiques d'assertion suite(1)

- **assertion interne :**
  visible seulement à l'intérieur d'une méthode
  - invariant d'itération :
    invariant, **inv:**
  - étape d'un algorithme:
    check, **assert:**

Syntaxe d'une assertion

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**).

Clause

Chaque clause est une proposition logique formée:

- de connecteurs logiques (not, and, or, xor, implies),
- d'opérateurs booléens (=, <>, <, $\leq$, etc),
- de valeurs littérales, de constantes symboliques, d'attributs et d'appels de fonctions sans effet de bord.
- Mais, en Eiffel, pas de quantificateurs.

Exemple:

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


Label

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

Exemples de labels

```
marier ( conjoint:  Personne)
pre
non_marié:
  not conjoint.marié
mariage_hétéro:
  conjoint.sexe /= self.sexe
age_légal:
  self.age >= 18 and conjoint.age >= 18
```

Exemple: classe Account

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


Exemple: classe Account suite(1)

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


Rôle des préconditions (pre:)

- doivent être vraies juste avant chaque appel de la méthode.
- expriment les conditions ou hypothèses à satisfaire pour que le programmeur sache implémenter la méthode.
- inutile de retester la précondition dans le code : elle peut jouer le rôle d'une **garde** pour la méthode.

Rôle des postconditions (post:)

- 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** complétement ou partiellement une primitive.

Rôle des postconditions (post:) suite(1)

Remarques:

Observer sur l'exemple de l'initialiseur la parfaite redondance entre le code et la postcondition.

Pour les méthodes complexes, les postconditions sont beaucoup plus synthétiques que le code.

Accesseur primaire

- permet d'accéder aux états ou aux constantes des objets,
- joue le même rôle que les **générateurs** des spécifications algébriques,
- n'a pas de précondition,
- 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,
- Exemples : `balance`, `minBalance`.

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 suite(1)

Exemple: 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)
```

Effet de bord et assertion

- 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 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*

Effet de bord et assertion suite(1)

- la partie visible aux clients d'un objet est définie par les accesseurs primaires publiques (exportés à tous les clients).
- les langages comme Eiffel qui autorisent l'exportation sélective fournissent des états abstraits différents selon les clients.

Opérateurs d'états

La plupart des opérateurs d'état ne peuvent s'utiliser que dans les **postconditions** :

- **old** ou **@pre**,
- **nochange**, **strip**, **isQuery**,
- **new**,
- **garantee**,
- **rely**...

valeur antérieure: opérateurs old, @pre

Distinguent l'état d'une variable/expression à l'entrée et à la sortie d'une méthode.

- état à la sortie : *pas de notation particulière*,
- état à l'entrée : indiqué par un opérateur spécial
  - variable' (Hoare),
  - **old** expression (Eiffel),
  - expression@pre (OCL)

valeur antérieure: opérateurs old, @pre suite(1)

L'opérateur **@pre** a un sens précis :

seule l'expression immédiatement avant **@pre** *est évaluée à l'entrée.*

Il faut répéter autant de fois que nécessaire **@pre** *:*

```
item(i)@pre /= item(i@pre)@pre
```


opérateurs old, @pre : exemples

- incrémentation d'une variable : `i = old i +1`
- incrémentation d'une variable : `i = i@pre +1`
- `subject.mentor@pre` :
  sujet dans son état actuel,
  mentor dans l'état initial (à l'entrée).
- `subject.mentor.mentee@pre` :
  sujet dans son état actuel,
  mentor dans l'état actuel, disciple dans l'état initial.

opérateur @pre : exemples

- `subject.mentor@pre.mentee@pre` :
  sujet dans l'état actuel,
  mentor dans l'état initial, disciple dans l'état initial.
- `subject.(mentor.mentee)@pre` : *idem*
- `subject.mentor@pre.mentee` :
  sujet dans l'état actuel,
  mentor dans l'état initial, disciple dans l'état actuel.

La notation OCL **@pre** est plus précise que le **old** d'Eiffel.

Exemple de spécification avec @pre
class Account

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

Exemple de spécification avec @pre

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


Exemple de spécification avec @pre

```
feature {None} -- private
  move (sum:  Integer)
    body:
      balance := balance + sum
    post:
      balance = balance@pre + sum
```


prédicats de constance : nochange, isQuery

On veut souvent indiquer dans une postcondition qu'un système n'a pas été affecté par une méthode :

- Eiffel2 fournissait un prédicat **nochange** pour indiquer qu'un méthode n'avait pas changé l'état du système.
- OCL/UML indique la même chose avec le méta-attribut **isQuery** appliqué à une méthode réifiée : *les accesseurs (ou Queries) rendent la valeur* `true`, *les modifieurs (ou Operations) rendent* `false`.

Opérateur de changement partiel : @strip

suite(1)

- Eiffel3 fournit un opérateur pour sélectionner un **état complet ou partiel d'un objet:**
  strip(liste d'attributs) :
  liste des attributs d'un objet,
  sauf ceux indiqués dans la liste **strip**.

Exemples :

- absence d'un changement d'état : strip() = **strip**()**@pre**
- changement uniquement de la variable x :
  strip(x) = **strip**(x)**@pre**

opérateur de nouvelles créations : new

Catalysis, puis OCL1.3 ont introduit la fonction **new** qui rend, **dans une postcondition**, l'ensemble de toutes les instances qui ont été créées par la méthode.

Cet opérateur n'est intéressant que si le langage d'assertion fournit des quantificateurs sur des collections :

Exemple: fabrique d'objets ayant une propriété caractéristique

opérateurs de durée : rely, garantee

Pour les interactions en parallèle (collaborations...), il est parfois utile de spécifier une condition qui doit être vraie **durant toute une action** :

- **garantee:** *condition*
  exprime une condition essentielle, un objectif, comme une postcondition, une propriété **fournie**.
- **rely:** *condition*
  exprime une condition nécessaire pour que l'objectif poursuivi par la garantie soit atteint: comme une precondition, une propriété **requise**.

opérateurs de durée : rely, garantee suite(1)

Exemples :

- **garantee:** dépôt.stock.size() $>$ 0
- **rely:** grossiste.enActivité

Invariant d'instance

Chaque instance est susceptible de passer par un grand nombre d'états (trace) **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.

Invariant d'instance suite(1)

Donc, pour chaque initialiseur/constructeur ou modifieur :
postcondition $\Rightarrow$ **invariant**

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

Exemple: classe Account

inv:

```
balance >= minBalance
```

Explications ...

Invariant de type : axiome

Si toutes les instances d'une classe ont une propriété remarquable d'invariance, *indépendante de tout état*, cette propriété est de nature **mathématique** :

- c'est un invariant de type, donc un **axiome**

Relations entre les assertions externes

Par ordre d'importance décroissante, les invariants de classe, puis d'instance sont les propriétés les plus remarquables qui doivent être perçues très tôt lors de la conception.

Puis, les postconditions doivent impliquer les invariants, et se décrivent ensuite.

Enfin les préconditions sont nécessaires à la réalisation des postconditions et se décrivent en dernier.

Exemple avec la classe `Account`

Héritage des invariants

Les invariants d'instance sont également hérités par les classes héritières (conjonction implicite) et jouent le rôle de **contraintes sémantiques très fortes**, substancielles des classes : *sorte de* ***code génétique****, qui protège un auteur contre toute utilisation infidèle par héritage.*

C'est particulièrement important lors d'héritage multiple et profond...

Exemple d'héritage d'invariant d'instance

```
class Herbivore inherit Animal
  estomac: Conteneur <Nourriture>
  manger(n: Nourriture)
    -- absorbe la nourriture n, si non rassasié
  pre:
    honnêté: ¬ n.vide
    salubrité: n.estVégétal
    sobriété:
        estomac.capacité >= n.quantité
  post:
    estomac = estomac@pre + n
```


Exemple d'héritage d'invariant suite(1)

```
  ...
inv:
  ¬ estomac.vide ⇒
      estomac.contenu.estVégétal
```


Exemple d'héritage d'invariant suite(2)

```
class Bovin
-- expérience d'un chercheur fou de l'INRA
inherit Herbivore redefine manger
  manger(n: FarineAnimale)
    -- absorbe la nourriture n, si non rassasié
  ...
inv:  -- hérité de Herbivore, pas moyen d'échapper à son contrôle !
caractéristique: ¬ estomac.vide ⇒
      estomac.contenu.estVégétal
```


Exemple d'héritage d'invariant suite(3)

```
-- test du chercheur fou ...
v: Vache ; v := new v
v.manger(carcasseMouton)
⇒assertion violated !
instance: v (#325765) : Bovin
last method: manger(carcasseMouton)
invariant: caractéristique of class Bovin,
inherited from Herbivore
  ¬ estomac.vide ⇒
      estomac.contenu.estVégétal
message:  -- Convocation immédiate dans le bureau du directeur !
```

Assertions d'itération : invariant & variant

Comme toute assertion, les assertions d'itération peuvent être prouvées ou testées.

L'**invariant d'itération** doit être vrai à chaque cycle de l'itération.

Le **variant** est une fonction entière ≥ 0 décroissante qui atteint 0 lors de la sortie :*pour tester la finitude*

Assertions d'itération : exemple suite(1)

```
search ( e:  Element )
pre eExists:  e /= Void
body:
   from start
   invariant:  1 <= pos <= length + 1
   variant:  length - pos -1
   until pos= length+1 || e = currentItem
   loop pos:= pos+1 end
post:
     pos <= length ⇒ e = currentItem
```


Spécification d'une classe Vue abstraite des clients

En Eiffel, on a des outils (`ebench`, `short`, `flat`...) pour extraire automatiquement la spécification des classes, à partir des assertions externes ⇒
spécifications formelles, complète ou partielle.

Exemple avec la classe Account

```
class interface Account
creation initial
feature
  balance:  Integer
  minBalance:  Integer = 1000
  ...
```

Vue abstraite des clients suite(1)

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


Vue abstraite des clients suite(2)

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


Vue abstraite des clients suite(3)

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


Vue abstraite des clients suite(4)

```
withdraw (sum:  Integer )
  -- withdraw sum from the account
  pre:
    trueWithdraw:  sum >= 0
    suppliedEnough:  mayWithdraw(sum)
  post:
    balance = balance@pre - sum
inv:
  balance >= minBalance
end Account
```

Vue abstraite des clients suite(5)

Les outils d'extraction offrent **différentes vues** possibles : *clients, héritiers, grasses ou allégées...* et selon des **directives des programmeurs**...

Les clauses qui citent des entités cachées sont automatiquement masquées.

Chapitre 3

OCL : langage d'assertions non exécutables pour UML

1) Introduction au langage OCL
2) Types et Valeurs de Base
3) Accès aux objets et aux propriétés

Sommaire suite(1)

4) Types fondamentaux
5) Types simples
6) Types collectifs
7) Grammaire d'OCL

Introduction au langage OCL

Le langage OCL *Object Constraint Language* est un langage expressif d'assertions, **non exécutables**, quantifiées :

- défini pour la première fois en 1997 par IBM,
- adopté pour UML (dern. version : 2, 2003) et plusieurs méthodes : *Syntropy, Catalysis*
- rapport officiel : *OMG Adopted specifications ptc/03-10-14*

Introduction au langage OCL suite(1)

- aspects sémantiques d'UML : *tous décrits en OCL.*
- OCL est compatible avec les modèles de l'OMG : ODMG, CORBA...
- syntaxe : *proche d'Eiffel, mêmes notions (pré/post/invariant)*
- langage abstrait d'expressions :
  pas d'instructions d'implémentation ou d'affectations à effet de bord.

Place des contraintes dans UML

UML est un langage graphique ⇒
assertions raccrochées aux figures par des clauses de **context** :

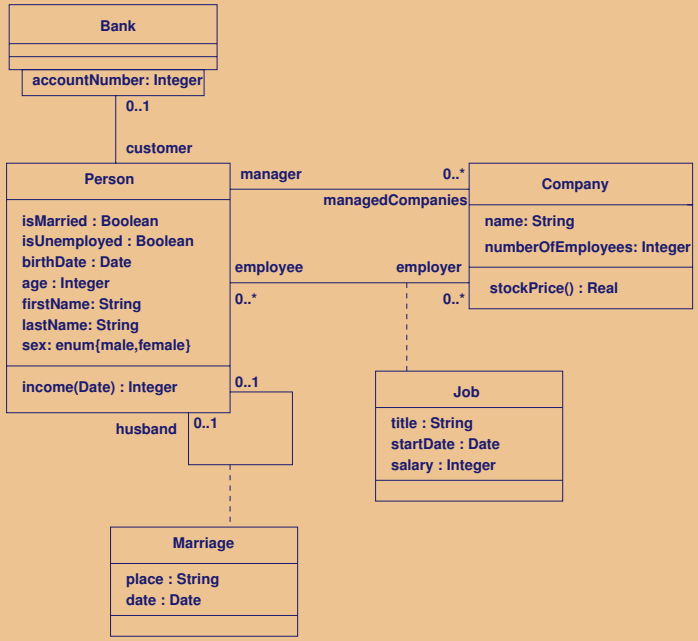
```
context TypeName inv :
   -- expression OCL avec stereotype invariant
   -- dans le contexte de TypeName = "autre chaîne"
```


Place des contraintes dans UML

OCL est un langage **typé** : toute expression a un type.

Chaque **classifieur** (type, classe, interface, association...) du modèle UML représente un **type** OCL.

Il existe un ensemble de **types prédéfinis** qui permettent de normaliser les descriptions *(voir plus loin).*



Contexte

La notation : `self.numberOfEmployees > 50` serait ambiguë.

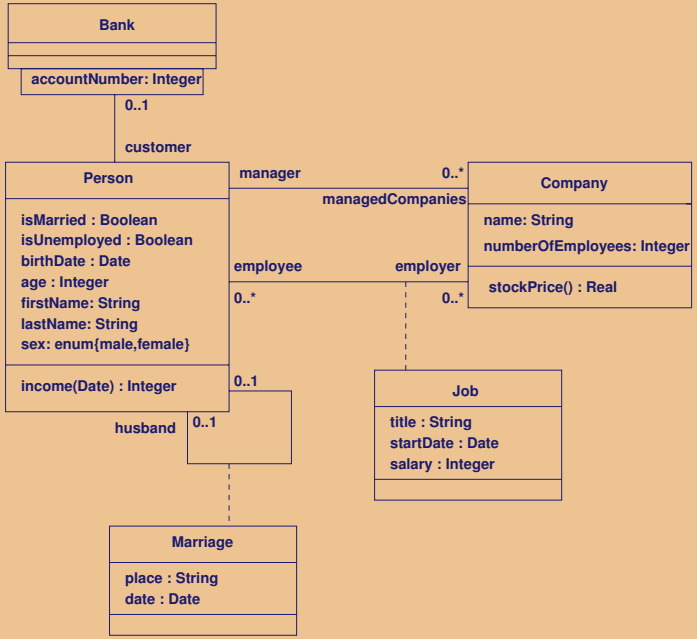
On précise le contexte de chaque contrainte, qui peuvent être regroupées sous le même contexte.

Chaque contrainte précise sa nature (**pre**:, **inv**: ...) suivie d'un label éventuel comme en Eiffel:

```
inv enoughEmployees:  c.nbOfEmp > 50
```


Exemples d'invariants suite(1)

```
-- Invariants de types
context Company
   inv:  self.numberOfEmployees > 50
-- ou
context Company
   inv:  numberOfEmployees > 50
context c : Company
   inv:  c.numberOfEmployees > 50
context c : Company
   inv enoughEmployees:
         c.numberOfEmployees > 50
```

Exemples de pré/post conditions

```
context Typename::operationName
          ( p1:  Type1, ...):ReturnType
  pre :  param1 > ...
  post:  result = ...
context Person::income(d :  Date) :  Integer
  post:  result = 5000
```


Expression let

```
context Person inv:
  let income :  Integer =
      self.job.salary->sum in
  if isUnemployed then
    income < 100
  else
    income >= 100
  end if
```


Types de base

très classiques, le minimum vital...

Notation des valeurs littérales :

type	values
Boolean	true, false
Integer	1, -5, 2, 34, 26524, ...
Real	1.5, 3.14, ...
String	'To be or not to be...'

Types de bases suite(1)

Opérations sur les types de base :

type	operations
Boolean	**and**, **or**, **xor**, **not**, **implies**, **if-then-else**
Integer	*, +, -, /, abs
Real	*, +, -, /, floor
String	toUpper, concat

Type énuméré

Déclaration :

```
enum{ value1, value2, value3 }
```

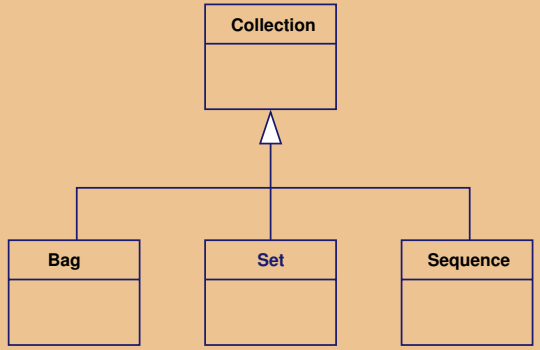
Utilisation dans une expression :

```
value1 -- si aucune ambiguïté
#value1 -- si ambiguïté
```

voir la spécification complète du type enum plus loin

Conformité des types

Hiérarchie minimale :



Conformité des types suite(1)

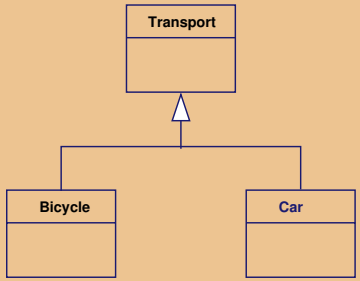
Type	Conforms to / Is a subtype of
Set(T)	Collection(T)
Sequence(T)	Collection(T)
Bag(T)	Collection(T)
Integer	Real

En fait, `Collection` et `Bag` sont identiques...

Conformité des types suite(2)

Respectent les règles de conformité du langage Eiffel, mais pas celles de Java5 avec généricité

Exemple:



Système de typage suite(3)

- `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)`
- 12 + 13.5 OK, `Integer` conforme à `Real`

En Java5, toutes ces compatibilités ne marchent pas : `Set(Bicycle)` n'est pas conforme à `Set(Transport)`...

Système de typage suite(4)

Retypage : *(Casting)*

Soit T2 un sous-type de T1, obj1, une variable de type T1.

```
obj1.oclAsType(T2)  -- evaluates to object with type T2
```

est légal, si on est sûr que le type effectif de obj1 est T2

Système de typage suite(5)

Valeur Non définie :

Tout opérande ou argument peut avoir une valeur "non définie". Le résultat de l'expression est alors "undefined".
exceptions :

- **or** rend **true** si l'un de ses argument est **true**
- **and** rend **false** si l'un de ses argument est **false**

(même si les autres arguments sont undefined)

Système de typage suite(6)

Opérateurs Infixes :

```
a + b
-- est une abréviation d'écriture de :
a.+(b)
-- active la méthode '+' de l'objet a (receveur) avec l'argument b
```

Accès aux Objets et aux propriétés

Chaque expression OCL peut se référer :

- aux classifiers : *type, classe, interface, association...*
- aux propriétés (features, properties, primitives) : attributs, extrémités d'association (rôles), méthodes, opérations, pourvu qu'elles n'aient **aucun effet de bord**. : leur prédicat `isQuery` est **true**.

Notation pointée

```
context AType inv:
  self.property
-- Exemple : accès à un attribut
context Person inv:
  self.age > 0
```


Notation pointée

```
-- Exemple : accès à une méthode
-- (les appels aux propriétés peuvent être récursifs)
aPerson.income(aDate)
context Person::income (d: Date) : Integer
  post: result = age * 1000 -- plutôt stupide
context Company inv:
  self.stockPrice() > 0
```


Accès aux rôles

(extrémités d'associations)

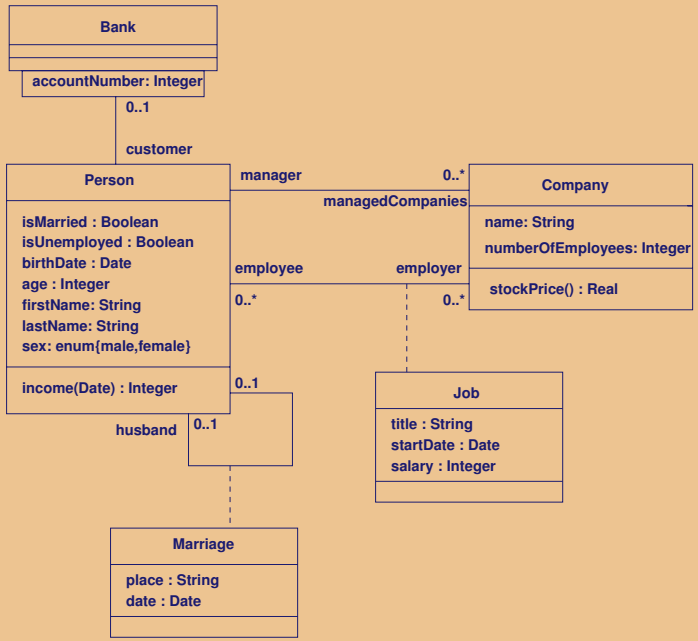
```
object.rolename
  -- = collection des objets de l'autre côté
object.rolename->property
  -- = accès à la propriété de la collection elle-même
```

Navigation via les rôles

- **Cardinalités simples : 0 ou 1** ⇒ rend un objet
- **Cardinalités multiples, sans ordre** ⇒ rend un Set
- **Cardinalités multiples, avec ordre** ⇒ rend une Sequence

Exemples

```
-- Accès avec le nom d'un rôle
context Company
   inv :  self.manager.isUnemployed = false
   inv :  self.employee->notEmpty
```

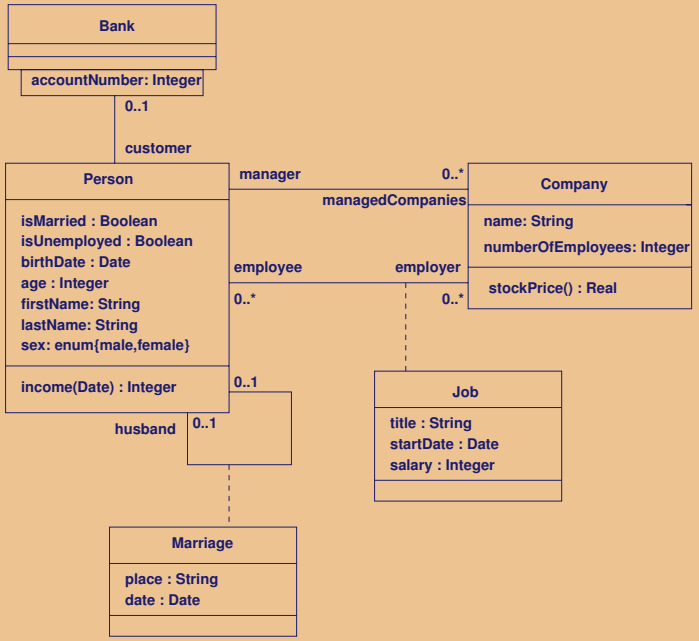



Exemples suite(1)

```
context Person inv :
   -- pas plus de 3 employeurs par salarié
   Emmanuelle.employer->size < 3
   -- l'ensemble des employeurs de Paul est vide
   Paul.employer->isEmpty
```

Exemples suite(2)

```
context Person inv:
-- Accès sans nom d'un rôle
   Paul.bank
      -- Nom du type de l'autre extrémité en minuscule :
      -- Ne doit pas avoir d'ambiguïtés,
      -- sinon le nom de rôle est nécessaire
   Paul.company
      -- managedCompanies ou employer ?
```

Associations de cardinalité 0..1

Lorsqu'une navigation conduit à un objet, celui-ci peut être considéré comme un ensemble « *singleton* »

```
-- Utilisation comme ensemble singleton
context Company inv:  self.manager->size = 1
-- Utilisation comme objet
context Company inv:  self.manager.age> 40
-- Utilisation comme ensemble et objet
context Person inv:
  self.wife->notEmpty implies
          self.wife.sex = #female
```

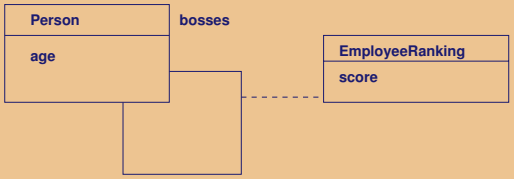

Combinaisons de propriétés

```
-- Personnes mariées agées d'au moins 18 ans
context Person inv:
  self.wife->notEmpty implies
    self.wife.age >= 18
  and
  self.husband->notEmpty implies
    self.husband.age >= 18
```

Combinaisons de propriétés suite(1)

```
-- Une société a au moins 50 employés
context Company inv:
  self.employee->size >= 50
```

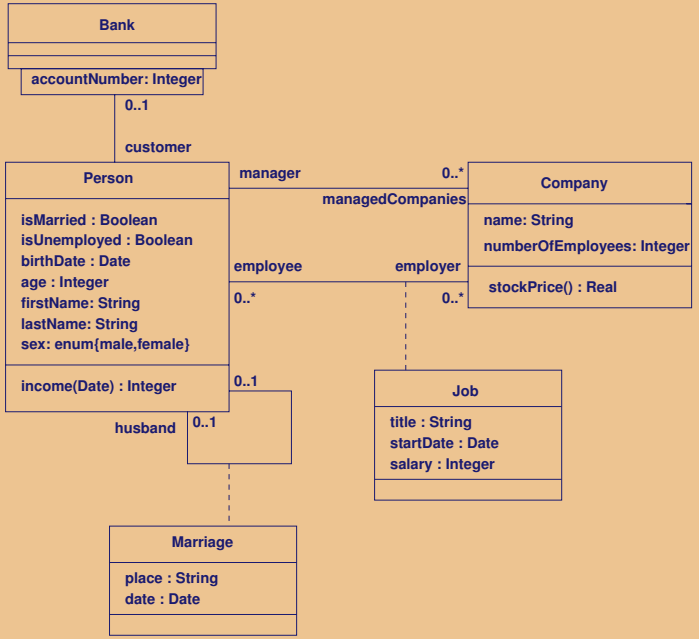

Associations de cardinalité *



Associations de cardinalité * suite(1)

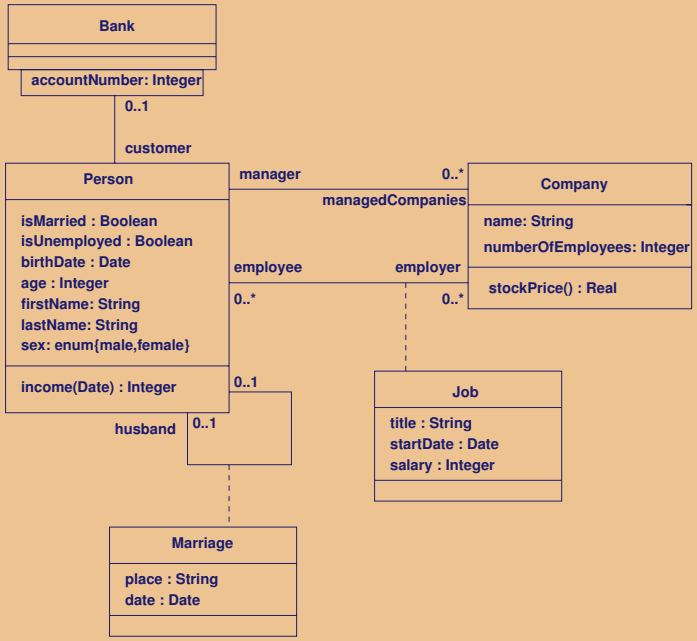
Bien distinguer le point de la flêche...

```
-- Ensemble des jobs d'une personne
context Person inv:  self.job
context Person inv:
  self.employeeRanking[bosses]->sum > 0
context Person inv:
  self.employeeRanking[employees]->sum > 0
context Person inv:
  self.employeeRanking->sum > 0 -- INVALID!
context Person inv:  self.job[employer]
```

Autres navigations

```
-- Navigation depuis une classe d'association
context Job
   inv:self.employer.numberOfEmployees >= 1
   inv:self.employee.age > 21
-- Navigation à travers une association qualifiée
context Bank inv:
   self.customer
context Bank inv:
   self.customer[8764423]
```

Autres navigations suite(1)

```
-- Navigation à travers des Paquetages
Packagename::Typename
Packagename1::Packagename2::Typename
-- Accès aux propriétés des supertypes
context B inv:
   self.oclAsType(A).p1
      -- accesses the p1 property defined in A
```

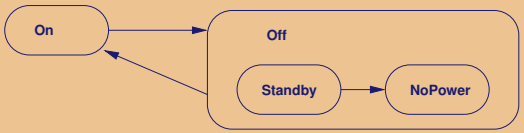

Métaconcepts

Extension d'un type

```
Type OclType
t.allInstances -- t : an instance of OclType
   -- Rend l'ensemble (Set) de tous les objets du type
   -- Non défini pour les types Integer, Real, String...
Exemple:
context Person inv:
  Person.allInstances->forAll(p1, p2 |
    p1 <> p2 implies p1.name <> p2.name)
```

OclState

Dans un schéma de conception UML, on peut utiliser des diagrammes d'états avec des noms :



OclState suite(1)

OCL admet de citer ces noms d'états dans l'opération `oclInState` :

```
object.oclInState(On)
object.oclInState(Off)
object.oclInstate(Off::Standby)
object.oclInState(Off::NoPower)
```


OclExpression

Chaque expression OCL est un objet d'un type bien déterminé, dans un contexte OCL. Ce type est utilisé pour :

- définir la sémantique des propriétés associées au type de cette expression, par exemple : select, collect, forAll...
- possibilité d'emploi dans la variable ou le type optionnel d'un curseur (appelé `iterator`, voir les primitives itératives de Collection)
- possibilité d'emploi dans la variable ou le type optionnel d'un accumulateur (voir les primitives itératives de Collection)

OclType

Tous les types OCL sont réifiés et sont de type OclType : pour les modélisations avancées, outils de GL...

Type `OclType` -- *Let t an instance of OclType*

```
t.name :  String
   -- The name of t.
t.attributes :  Set(String)
   -- The set of names of the attributes of t,
   -- as they are defined in the model.
...
```

OclType suite(1)

```
t.associationEnds : Set(String)
   -- The set of names of the navigable associationEnds of t,
   -- as they are defined in the model.
t.operations : Set(String)
   -- The set of names of the operations of t,
   -- as they are defined in the model.
...
```


OclType suite(2)

```
t.supertypes : Set(OclType)
   -- The set of all direct supertypes of t.
   post: t.allSupertypes->includesAll(result)
t.allSupertypes : Set(OclType)
   -- The transitive closure of the set of all supertypes of t.
t.allInstances : Set(t)
   -- The set of all instances of t and all its subtypes in existence
   -- at the snapshot at the time that the expression is evaluated.
End OclType
```


OclAny

```
Type OclAny
-- Let o, o1, o2... instances of OclAny
o1 = (o2 : OclAny) : Boolean
   -- True if o1 is the same o1 as o2.
o1 <> (o2 : OclAny) : Boolean
   -- True if o1 is a different o1 from o2.
   post: result = not (o1 = o2)
o1.oclIsKindOf(t : OclType) : Boolean
   -- True if t is one of the types of o1, or one of the
   -- supertypes (transitive) of the types of o1.
```


OclAny suite(1)

```
o1.oclIsTypeOf(t : OclType) : Boolean
   -- True if t is equal to one of the types of o1.
o1.oclAsType(t : OclType) : OclAny
   -- Results in o1, but of known type t.
   -- Results in Undefined if the actual type of o1
   -- is not t or one of its subtypes.
   pre : o1.oclIsKindOf(t)
   post: result = o1
   post: result.oclIsKindOf(t)
```

OclAny suite(2)

```
o1.oclInState(state : OclState) : Boolean
   -- True if o1 is in the state state,
   -- The argument is a name of a state in the state machine
   -- corresponding with the class of o1.
o1.oclIsNew : Boolean
   -- Can only be used in a postcondition.
   -- True if the o1 is created during performing the operation.
   -- I.e. it didn't exist at precondition time.
```


OclAny suite(3)

```
-- Pas de méthodes en OCL pour connaître la réification des types
-- Ce qui suit n'est PAS défini en OCL
o1.oclType : OclType
   -- reification of the object type
o1.oclType.isGeneric : Boolean
   -- Is the type of object generic ?
o1.oclType.genericNumber : Integer
   -- nb of generic parameters
o1.oclType.generic(n:Integer) : OclType
   -- reification of the the nth generic parameter
End OclAny
```


OclAny: exemples suite(4)

```
context Person
   inv : self.oclIsTypeOf( Person ) -- is true
   inv : self.oclIsTypeOf( Company) -- is false
```


Structures collectives

Pour exprimer des **quantifications**, il faut parcourir des collections d'objets :

- dans une structure de données parcourable séquentiellement,
- ou dans extension de type (qui est aussi une collection).

La primitive de base est `iterate`*, qui permet de définir les autres opérations de parcours ou les quantifications.*

Structures collectives suite(1)

Littéraux de structures collectives:

```
Set {}
Set { 1, 2, 5, 88 }
Set { 'apple' , 'orange', 'strawberry' }
Sequence { 1, 3, 45, 2, 3 }
Sequence { 'ape', 'nut' }
Bag { 1, 3, 4, 3, 5 }
```


Structures collectives suite(2)

```
Sequence{ 1..(6 + 4) }
Sequence{ 1..10 }
-- les deux sont identiques à
Sequence{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
Set{ Set{1, 2}, Set{3, 4}, Set{5, 6} }
-- mêmes valeurs que
Set{ 1, 2, 3, 4, 5, 6 }
```


Itération

```
Type Collection (T)
-- Let c, c1, c2... instances of Collection(T)
c->iterate(
   elm:  T;
   -- T peut être omis, puisque elm est forcément de type T
   acc:  R = initial-value-expr
   | expr-with-elm-and-acc :  OclExpression
)
-- "|" se lit "tel que" en français...
-- Rend l'accumulation de expr-with-elm-and-acc dans acc
-- lors du parcours complet de c via le curseur elm.
```


Itération suite(1)

@@ Dessin

Itération suite(2)

```
-- ce qui correspond, en Java-like pseudocode, à :
iterate(elm:  T;
    acc:  R = initial-value-expr) {
  acc = initial-value-expr;
  for ( Collection c ;
      c.hasMoreElements(); ){
    elm = c.nextElement();
    acc = expr-with-elm-and-acc
  }
return acc;
}
```


Exemple d'utilisation de iterate

```
context Collection(T)::collect(expr) :
                Collection(expr.oclType)
-- The Collection of values which results from applying expr
-- to every member of self : map of lisp
post:  result =
  iterate(e;
    acc :  Collection(expr.oclType)
      = Collection{}
      | acc->including(expr)
  )
```


Cardinal d'une collection C

$$R = |C|$$

s'écrit :

```
c->size :  Integer
  -- The number of elements in the collection c.
post:
result= c->iterate(e; a:Integer=0 | a+1)
```


Collection vide ? suite(1)

$$R = (C = Collection\{\})$$

s'écrit :

```
c->isEmpty :  Boolean
  -- Is c the empty collection?
post:  result = ( c->size = 0 )
```

Collection non vide ? suite(2)

$$R = (C \neq Collection\{\})$$

s'écrit :

```
c->notEmpty : Boolean
  -- Is c not the empty collection?
post: result = ( c->size <> 0 )
```


Nombre d'occurences d'une valeur

$$R = |Collection\{x_i \mid \forall x_i \in C\}|$$

s'écrit :

```
c->count(x : OclAny) : Integer
  -- The number of times that x occurs in the collection C.
  post:
    result = c->iterate(e; a:Integer=0 |
      if e = x then a+1 else a end if)
```


Test d'appartenance suite(1)

$$R = (x \in C)$$

s'écrit :

```
c->includes(x : OclAny) : Boolean
  -- True if x is an element of C.
  post: result = (c->count(x) > 0)
```


Test de non appartenance suite(2)

$$R = \neg(x \in C)$$

s'écrit :

```
c->excludes(x : OclAny) : Boolean
  -- True if x is not an element of C.
  post: result = (c->count(x) = 0)
```

forAll (curseur implicite)

$$R = \bigwedge_e expr(p(e)) \ \forall e \in C$$

s'écrit :

```
c->forAll(expr) : Boolean
   -- True if boolean expr evaluates to true for each element in c.
   -- expr have to cite some properties of elements of c.
post:
result= c->iterate(e; a: Boolean =true |
            a and booleanExpr)
Exemple:
context Company inv:
   employee->forAll(firstName = 'Jack')
-- booleanExpr utilise le curseur e implicitement
```


forAll (curseur explicite) suite(1)

$$R = \bigwedge_e expr(e) \ \forall e \in C$$

s'écrit :

```
c->forAll(e|booleanExpr-with-e):Boolean
c->forAll(e: c.oclType.generic(1) |
          booleanExpr-with-e):Boolean
-- true if booleanExpr-with-e evaluates to true for each element in c
post:
result = c->iterate(e; a: Boolean =true |
            a and booleanExpr-with-e)
```


Exemples de forAll avec curseur explicite

```
context Company
inv:
   self.employee->forAll(p |
               p.firstName = 'Jack')
inv:
   self.employee->forAll( p : Person |
               p.firstName = 'Jack' )
```


forAll (curseurs multiples) suite(1)

```
c->forAll(e1,e2 | booleanExpr-with-e1,e2)
            : Boolean
c->forAll(e1, e2 : c.oclType.generic(1)
    | booleanExpr-with-e1,e2) : Boolean
   -- true if booleanExpr-with-e1,e2 evaluates to true
   -- for each element in c.
post: result = forAll(e1 | c->
          forAll(e2 |
            booleanExpr-with-e1,e2))
```

Cette construction se **généralise à n curseurs** ...

Exemples de forAll avec deux curseurs suite(2)

```
context Company inv:
  self.employee->forAll( e1, e2 |
    e1 <> e2 implies
            e1.firstName <> e2.firstName)
context Company inv:
  self.employee->forAll( e1, e2 : Person |
    e1 <> e2 implies
            e1.firstName <> e2.firstName)
```


exists (curseur explicite) suite(1)

$$R = \exists e \in C \mid expr(e)$$

s'écrit :

```
c->exists(e |
  booleanExpr-wih-e) : Boolean
c->exists(e : c.oclType.generic(1) |
  booleanExpr-wih-e) : Boolean
  -- true if booleanExp-with-e evaluates to true
  -- for at least one element in c.
post:
result = c->iterate(e; a: Boolean =false |
          a or booleanExp-with-e)
```


exists (curseur implicite)

$$R = \exists e \in C \mid expr(p(e))$$

s'écrit :

```
c->exists(expr) : Boolean
  -- true if boolean exp is true for at least 1 element in c
  -- expr have to cite some properties of elements of c.
post:
result = c->iterate(e; a:Boolean =false |
          a or expr)
Exemple:
context Company inv:
  self.employee->exists(firstName = 'Jack')
```


Exemples de exists avec curseur explicite

```
context Company inv:
  self.employee->exists(p |
            p.firstName = 'Jack')
context Company inv:
  self.employee->exists(p: Person |
            p.firstName='Jack')
```

includesAll

$C_1 \supset C_2$

s'écrit :

```
c1->includesAll(c2:Collection(T)): Boolean
  -- Does c1 contain all the elements of c2 ?
post:
  result= c2->forAll(e | c1->includes(e))
```


excludesAll

$C_2 \cap C_1 = \varnothing$

s'écrit :

```
c1->excludesAll(c2:Collection(T)): Boolean
  -- Does c1 contain none of the elements of c2 ?
post:
  result= c2->forAll(e | c1->excludes(e))
```

Remarque:

```
c1->excludesAll(c2) <=> c2.excludesAll(c1)
```


égalité

$C_1 = C_2$

s'écrit :

```
c1 = (c2:  Collection(T)) : Boolean
  -- Evaluates to true if c1 and c2 contain the same elements.
post:
  result = (c1->forAll(e | c2->includes(e))
      and c2->forAll(e | c1->includes(e)) )
```


collect (curseur implicite)

$R = Collection\{expr(e_i)\ \forall e_i \in C\}$

s'écrit :

```
c->collect(expr:  OclExpression) :
                  Collection(T)
  -- The Collection of elements which results from applying expr
  -- to every member of c (~ map of lisp)
post:
result= c->iterate(e;
  a:Collection(expr.oclType)= Collection{}
      | a->including(expr)
```

collect (curseur explicite) suite(1)

```
c->collect(e |expr-with-e): Collection(T)
c->collect(e: c.oclType.generic(1) |
    expr-with-e) : Collection(T)
  -- The Collection of elements which results from applying
  -- expr-with-e to every member of c
post:
  result= c->iterate(e;
    a: Collection(expr.oclType)=
                Collection{}
      | a->including(expr-with-e)
    )
```


Exemples d'utilisation de collect

context

```
self.employee->collect(birthDate)
self.employee->collect(person|
        person.birthDate )
self.employee->collect(person: Person |
        person.birthDate )
```


Raccourci pour collect

```
c.propertyname
collection.propertyname(par1, par2, ...)
-- sont équivalents à :
c->collect(propertyname)
collection->collect(propertyname(par1, par2,
```

⇒ assez contestable, pour une économie réduite

Exemples de raccourcis de collect suite(1)

```
self.employee.birthdate
-- est équivalent à :
self.employee->collect(birthdate)
```

isUnique

Prédicat d'unicité:

```
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)
```


select (curseur implicite)

```
c->select(expr):
        Collection(c.oclType.generic(1))
-- The subcollection of c for which boolean expr is true.
-- expr have to cite some properties of elements of c.
post:
result = c->iterate(e;
  a:  Collection(c.oclType.generic(1)=
                 Collection{} |
      if expr
      then a->including(e)
      else a end if )
```


select (curseur explicite) suite(1)

```
c->select(e| booleanExpr-with-e):
        Collection(c.oclType.generic(1))
c->select(e:  c.oclType.generic(1)|
        booleanExpr-with-e):
        Collection(c.oclType.generic(1))
post:  result = c->iterate(e;
    a:  Collection(c.oclType.generic(1)=
                   Collection{} |
    if booleanExpr-with-e
      then a->including(e) else a end if)
```


reject (curseur implicite)

```
c->reject(expr:
    Collection(c.oclType.generic(1))
   -- Boolean expr have to cite some properties of elements of c.
   -- The subset of c for which expr criteria is false.
post:
  result= c->select(not expr)
```

reject (curseur explicite)

```
c->reject(e | expr-with-e) :
    Collection(c.oclType.generic(1))
c->reject(e:  c.oclType.generic(1) |
    expr-with-e)
    :  Collection(c.oclType.generic(1))
  -- The subset of c for which expr-with-e criteria is false.
post:
  result= c->select(not booleanExpr-with-e)
```


sum

```
c->sum :  T
  -- The addition of all elements in c.
  -- Elements must be of a type supporting the + operation.
  -- The + operation must take one parameter of type T
  -- and be both associative: (a+b)+c = a+(b+c),
  -- and commutative: a+b = b+a.
  -- Integer and Real fulfill this condition.
post:
  result= c->iterate(e; a:T =0 | a+e)
```


Collection: sortedBy

```
c->sortedBy(expr:  OclExpression):
                Sequence(T)
  -- The sorted Sequence containing all expressions evaluated
  -- for each element of c.
  -- The expr which has the lowest value comes first
  -- The type of the expr must have the < operation defined.
  -- The < operation must be transitive i.e.
  -- if a < b and b < c then a < c
end Collection
```

On déplore ici l'absence de généricité contrainte.

Bag

« `Bag` est une collection d'éléments avec possibilité de duplication »
Stockage sans idée de l'emplacement.

(en fait une abstraction de la réalité)

D'après la définition, il n'y a donc pas de différence entre `Bag` et `Collection` !!! Et en examinant de près leurs primitives, elles sont identiques, des généralisations des primitives de `Set` et `Sequence` :

Bag et Collection suite(1)

- Chaque précondition de Bag implique les préconditions correspondantes de Set ou Sequence, (en fait, pas de préconditions),
- Chaque postcondition est impliquée par les postconditions correspondantes de Set ou Sequence,

(règle de l'entonnoir) $\Rightarrow$ La classe Bag est inutile et équivalente à Sequence.

Bag et Collection

Les primitives suivantes ne sont pas définies pour les Collections en OCL, mais en fait, leur définition fait appel à des primitives définies dans les Collections. Ce sont donc aussi des primitives de Collection

union

Type `Bag (T)`

-- *Let* `b, b1, b2...` *instances of* `Bag(T)`

```
b1->union(b2 :  Bag(T)) : Bag(T)
```

-- *The union of b1 and b2.*

```
post:  result->forAll(e | result->count(e) =
    b1->count(e) + b2->count(e) )
post:  b1->forAll(e | result->count(e) =
    b1->count(e) + b2->count(e) )
post:  b2->forAll(e | result->count(e) =
    b1->count(e) + b2->count(e) )
```


union suite(1)

```
b1->union(s :  Set(T)) : Bag(T)
```

-- *The union of b1 and s.*

```
post:  result->forAll(e | result->count(e) =
       b1->count(e) + s->count(e))
post:  b1->forAll(e | result->count(e) =
       b1->count(e) + s->count(e))
post:  s->forAll(e | result->count(e) =
       b1->count(e) + s->count(e))
```

intersection suite(2)

```
b1->intersection(b2 :  Bag(T)) : Bag(T)
   -- The intersection of b1 and b2.
post:  result->forAll(e | result->count(e) =
       b1->count(e).min(b2->count(e)) )
post:  b1->forAll(e | result->count(e) =
       b1->count(e).min(b2->count(e)) )
post:  b2->forAll(e | result->count(e) =
       b1->count(e).min(b2->count(e)) )
```


intersection suite(3)

```
b1->intersection(s :  Set(T)) : Set(T)
   -- The intersection of b1 and s.
post:  result->forAll(e | result->count(e) =
           b1->count(e).min(s->count(e)) )
post:  b1->forAll(e | result->count(e) =
           b1->count(e).min(s->count(e)) )
post:  s->forAll(e | result->count(e) =
           b1->count(e).min(s->count(e)) )
```


inclusion suite(4)

```
b1->including(o :  T) : Bag(T)
   -- all elements of b1 plus o.
post:  result->forAll(e |
   if e = o
   then result->count(e) = b1->count(e) + 1
   else result->count(e) = b1->count(e)
   end if)
post:  b1->forAll(e |
   if e = o
   then result->count(e) = b1->count(e) + 1
   else result->count(e) = b1->count(e)
   end if)
```


exclusion suite(5)

```
b1->excluding(o :  T) : Bag(T)
   -- all elements of b1 apart from all occurrences of o.
post:  result->forAll(e | if e = o
   then result->count(e) = 0
   else result->count(e) = b1->count(e)
   end if)
post:  b1->forAll(e | if e = o
   then result->count(e) = 0
   else result->count(e) = b1->count(e)
   end if)
```

conversion suite(6)

```
b1->asSet :  Set(T)
   -- The Set containing all the elements from b1,
   -- with duplicates removed.
post:  result->forAll(e |
           b1->includes(e) )
post:  b1->forAll(e |
           result->includes(e))
```


conversion suite(7)

```
b1->asSequence :  Sequence(T)
   -- A Sequence that contains all the elements from b1,
   -- in undefined order.
post:  result->forAll(e |
           b1->count(e) = result->count(e))
post:  b1->forAll(e |
           b1->count(e) = result->count(e))
end Bag
```


Set

Les ensembles, au sens informatique :

finis, sans duplication

Toutes les primitives sont des redéfinitions de celles de du type `Collection`, plus précises (règle de l'entonnoir) ou nouvelles.

Set suite(1)

```
Type Set (T)
-- Let s, s1, s2... instances of Set(T)
s->count(o :  T) : Integer -- Redefinition
   -- The number of occurrences of o in s.
post:  result <= 1
```

conversion

```
s->asSequence :  Sequence(T)
  -- A Sequence that contains all the elements from s,
  -- in undefined order.
post:  result->forAll(e | s->includes(e))
post:  s->forAll(e | result->count(e) = 1)
```


union

```
s1->union(s2 :  Set(T)) : Set(T)
  -- The union of s1 and s2.
post:  result->forAll(e |
       s1->includes(e) or s2->includes(e))
post:  s1->forAll(e | result->includes(e))
post:  s2->forAll(e | result->includes(e))
```


union suite(1)

```
s->union(b :  Bag(T)) : Bag(T)
  -- The union of s and b.
post:  result->forAll(e |
       result->count(e) = s->count(e)
                + b->count(e))
post:  s->forAll(e | result->includes(e))
post:  b->forAll(e | result->includes(e))
```


intersection

```
s1->intersection(s2 :  Set(T)) : Set(T)
  -- The intersection of s1 and s2
post:  result->forAll(e |
  s1->includes(e) and s2->includes(e))
post:  s1->forAll(e |
  s2->includes(e) = result->includes(e))
post:  s2->forAll(e |
  s1->includes(e) = result->includes(e))
s->intersection(b :  Bag(T)) : Set(T)
  -- The intersection of s and b.
post:  result = s->intersection( b->asSet )
```

inclusion

```
s->including(o : T) : Set(T)
  -- The s containing all elements of s plus o.
post: result->forAll(e |
  s->includes(e) or (e = o))
post: s->forAll(e | result->includes(e))
post: result->includes(o)
```


exclusion

```
s->excluding(o : T) : Set(T)
  -- The s containing all elements of s without o.
post: result->forAll(e |
  s->includes(e) and (e <> o))
post: s->forAll(e |
  result->includes(e) = (o <> e))
post: result->excludes(o)
```


différence

```
s1 - (s2 : Set(T)) : Set(T)
  -- The elements of s1, which are not in s2.
post: result->forAll(e |
  s1->includes(e) and s2->excludes(e))
post: s1->forAll(e |
  result->includes(e) = s2->excludes(e))
```


différence symétrique

```
s1->symmetricDifference(s2: Set(T)): Set(T)
  -- The sets containing all the elements that are in s1 or s2,
  -- but not in both.
post: result->forAll(e |
  s1->includes(e) xor s2->includes(e))
post: s1->forAll(e |
  result->includes(e) = s2->excludes(e))
post: s2 ->forAll(e |
  result->includes(e) = s1->excludes(e))
end Set
```

Séquence (suite)

Une collection d'éléments, avec possibilité de duplication.
Stockage avec idée de l'emplacement :
notion abstraite de position, incarnée par un indice entier.

Là encore la plupart des primitives sont des redéfinitions de celles de Collection/Bag

longueur

```
Type Sequence (T)
-- Let s1 an instance of Sequence(T)
s->count(o : T) : Integer
   -- The number of occurrences of o in s.
```


égalité

```
s = (s2 : Sequence(T)) : Boolean
   -- True if s contains the same elements as s2 in the same order.
post: result = Sequence{1..s->size}->
  forAll(i: Integer |
  s->at(i)=s2->at(i)) and s->size=s2->size
```


union

```
s->union (s2 : Sequence(T)) : Sequence(T)
   -- The s consisting of all elements in s,
   -- followed by all elements in s2.
post: result->size = s->size + s2->size
post: Sequence{1..s->size}->forAll(
  i: Integer |
  s->at(i) = result->at(i))
post: Sequence{1..s2->size}->forAll(
  i : Integer |
  s2->at(i) = result->at(i + s->size)))
```

inclusion

```
s->including(o : T) : Sequence(T)
   -- The sequence containing all elements of s
   -- plus o added as the last element.
post: result = s.append(o)
```


exclusion

```
s->excluding(o : T) : Sequence(T)
   -- The sequence containing all elements of s
   -- apart from all occurrences of o.
   -- The order of the remaining elements is not changed.
post: result->includes(o) = false
post: result->size = s->size - s->count(o)
post: result = s->
  iterate(e; a: Sequence(T) =Sequence{}|
    if e = o then a else a->append(e) end if)
```


itération

```
s->iterate(el: Type1;
  acc: Type2 = initial-value |
expr-with-el-and-all : OclExpression )
-- Redefinition
   -- Iteration will be done from element at position 1
   -- up until the element at the last position
   -- following the order of the s.
```


conversion

```
s->asBag() : Bag(T)
   -- The Bag containing all the elements from s,
   -- including duplicates.
post: result->forAll(e |
    s->count(e) = result->count(e) )
post: s->forAll(e |
    s->count(e) = result->count(e) )
```

conversion suite(1)

```
s->asSet() :  Set(T)
   -- The Set containing all the elements from s,
   -- with duplicated removed.
post:  result->forAll(e | s->includes(e))
post:  s->forAll(e | result->includes(e))
```


indiçage

```
-- Primitives spécifiques des Séquences
s->at(i :  Integer) :  T
   -- The i-th element of s.
pre :  i >= 1 and i <= s->size
```


premier élément

```
s->first :  T
   -- The first element in s.
post:  result = s->at(1)
```


dernier élément

```
s->last :  T
   -- The last element in s.
post:  result = s->at(s->size)
```

ajout à la fin

```
s->append (o:  T) : Sequence(T)
   -- The sequence of elements, consisting of all elements of s,
   -- followed by o.
post:  result->size = s->size + 1
post:  result->at(result->size) = o
post:  Sequence{1..s->size}->forAll(i :  Integ
            result->at(i) = s->at(i))
```


insertion en tête

```
s->prepend(o:  T) : Sequence(T)
   -- The sequence consisting of o, followed by all elements in s.
post:  result->size = s->size + 1
post:  result->at(1) = o
post:  Sequence{1..s->size}->forAll(
          i:  Integer |
            s->at(i) = result->at(i + 1))
```


sous-suite

```
s->subSequence(l,u:  Integer):  Sequence(T)
   -- The sub-sequence of s starting at number l,
   -- up to and including element number u.
pre :  1 <= l and l <= u and u <= s->size
post:  result->size = u-l+1
post:  Sequence{l..u}->forAll( i |
            result->at(i-l+1) = s->at(i))
end Sequence
```


Real

```
Type Real
   -- Let r1 an instance of Real
r1 = (r2 :  Real) :  Boolean
   -- True if r1 is equal to r2.
r1 <> (r2 :  Real) :  Boolean
   -- True if r1 is not equal to r2.
post:  result = not (r1 = r2)
r1 + (r2 :  Real) :  Real
   -- The value of the addition of r1 and r2.
```

Real suite(1)

```
r1 - (r2 : Real) : Real
   -- The value of the subtraction of r2 from r1.
r1 * (r2 : Real) : Real
   -- The value of the multiplication of r1 and r2.
r1 / (r2 : Real) : Real
   -- The value of r1 divided by r2.
```


Real suite(2)

```
r1.abs : Real
   -- The absolute value of r1.
post: if r1 < 0 then result = - r1
      else result = r1 end if
r1.floor : Integer
   -- The largest integer which is less than or equal to r1.
post: (result <= r) and (result + 1> r)
```


Real suite(3)

```
r1.round : Integer
   -- The integer which is closest to r1.
   -- When there are two such integers, the largest one.
post: ((r1-result)<r).abs < 0.5) or
      ((r1-result).abs = 0.5 and (result>r))
r1.max(r2 : Real) : Real
   -- The maximum of r1 and r2.
post: if r1 >= r2 then result = r1
      else result = r2 end if
```


Real suite(4)

```
r1.min(r2 : Real) : Real
   -- The minimum of r1 and r2.
post: if r1 <= r2 then result = r1
      else result = r2 end if
r1 < (r2 : Real) : Boolean
   -- True if r1 is less than r2.
```

Real suite(5)

```
r1 > (r2 : Real) : Boolean
   -- True if r1 is greater1 than r2.
post: result = not (r1 <= r2)
r1 <= (r2 : Real) : Boolean
   -- True if r1 is less than or1 equal to r2.
post: result = (r1 = r2) or (r1 < r2)
r1 >= (r2 : Real) : Boolean
   -- True if r1 is greater than or equal to r2.
post: result = (r1 = r2) or (r1 > r2)
end Real
```


Integer

```
Type Integer
-- Let i1 an instance of Integer
i1 = (i2 : Integer) : Boolean
   -- True if i1 is equal to i2.
i1 + (i2 : Integer) : Integer
   -- The value of the addition of i1 and i2.
i1 - (i2 : Integer) : Integer
   -- The value of the subtraction of i2 from i.
```


Integer suite(1)

```
i1 * (i2 : Integer) : Integer
   -- The value of the multiplication of i1 and i2.
i1 / (i2 : Integer) : Real
   -- The value of i1 divided by i2.
i1.abs : Integer
   -- The absolute value of i1.
post:
    if i1 < 0 then result= -i1
    else result = i1 end if
```


Integer suite(2)

```
i1.div( i2 : Integer) : Integer
   -- The number of times that i2 fits completely within i1.
pre : i2 <> 0
post:
    if i1 / i2 >= 0
    then result = (i1/i2).floor
    else result = -((-i/i2).floor) end if
i1.mod( i2 : Integer) : Integer
   -- The result is i1 modulo i2.
post: result = i1 - (i1.div(i2) * i2)
```

Integer suite(3)

```
i1.max(i2 : Integer) : Integer
  -- The maximum of i1 an i2.
post:
    if i1 >= i2 then result = i1
    else result = i2 end if
i1.min(i2 : Integer) : Integer
  -- The minimum of i1 an i2.
post: if i1 <= i2 then result=i1
            else result = i2 end if
end Integer
```


String

```
Type String
-- Let s1 an instance of String
s1 = (s2 : String) : Boolean
  -- True if s1 and s2 contain the same characters,
  -- in the same order.
s1.size : Integer
  -- The number of characters in s1.
```


String suite(1)

```
s1.substring(lower,upper : Integer) : Strin
  -- The sub-string of s1 starting at character number lower,
  -- up to and including character number upper.
s1.concat(s2 : String) : String
  -- The concatenation of s1 and s2.
post: result.size = s1.size + s2.size
post: result.substring(1, s1.size) = s1
post: result.substring(s1.size+1, result.size
```


String suite(2)

```
s1.toUpper : String
  -- The value of s1 with all lowercase characters converted
  -- to uppercase characters.
post: result.size = s1.size
s1.toLower : String
  -- The value of s1 with all uppercase characters converted
  -- to lowercase characters.
post: result.size = s1.size
end String
```

Boolean

```
Type Boolean
-- Let b1 an instance of Boolean
b1 = (b2 : Boolean) : Boolean
   -- Equal if b1 is the same as b2.
b1 or (b2 : Boolean) : Boolean
   -- True if either b1 or b2 is true.
b1 xor (b2 : Boolean) : Boolean
   -- True if either b1 or b2 is true, but not both.
   post: (b or b2) and not (b = b2)
```


Boolean suite(1)

```
b1 and (b2 : Boolean) : Boolean
   -- True if both b1 and b2 are true.
not b1 : Boolean
   -- True if b1 is false.
post:
   if b then result = false
   else result = true end if
```


Boolean suite(2)

```
b1 implies (b2 : Boolean) : Boolean
   -- True if b1 is false, or if b1 is true and b2 is true.
post: (not b) or (b and b2)
if b1
then (expr1 : OclExpression)
else (expr2 : OclExpression)
end if : expr1.evaluationType
   -- If b1 is true, the result is the value of evaluating expr1;
   -- otherwise, result is the value of evaluating expr2.
end Boolean
```


Enumeration

```
Type Enumeration
-- Let e1 an instance of Enumeration
e1 = (e2 : Enumeration) : Boolean
   -- Equal if e1 is the same as e2.
e1 <> (e2 : Enumeration) : Boolean
   -- Equal if e1 is not the same as e2.
post: result = not ( e1 = e2)
end Enumeration
```

Grammaire d'OCL

```
constraint := contextDeclaration
      (stereotype name?  ":" expression)+
contextDeclaration := "context"
      (classifierContext | operationContext)
classifierContext := (<name> ":")?  <typeName>
operationContext := <typeName> "::" <name>
      "(" formalParameterList?  ")"
      ( ":" <typeName> )?
formalParameterList :=
      formalParameter (";" formalParameter)*
formalParameter := <name> ":" <typeName>
stereotype := "inv" | "pre" | "post"
expression := letExpression* logicalExpression
```


Grammaire d'OCL suite(1)

```
ifExpression := "if" expression
    "then" expression "else" expression "end if"
logicalExpression := relationalExpression
    ( logicalOperator relationalExpression )*
relationalExpression:= additiveExpression
    ( relationalOperator additiveExpression )?
additiveExpression := multiplicativeExpression
    ( addOperator multiplicativeExpression )*
multiplicativeExpression := unaryExpression
    ( multiplyOperator unaryExpression )*
unaryExpression :=
    ( unaryOperator postfixExpression ) |
                          postfixExpression
```


Grammaire d'OCL suite(2)

```
postfixExpression := primaryExpression
        ( ("." | "->") featureCall )*
primaryExpression :=
    literalCollection
    | literal
    | pathName timeExpression?  qualifier?
              featureCallParameters?
    | "(" expression ")"
    | ifExpression
featureCallParameters :=
    "(" declarator?  actualParameterList?  ")"
letExpression := "let" <name>
      ( ":" pathTypeName )?  "=" expression "in"
literal := <STRING> | <number> | "#" <name>
```


Grammaire d'OCL suite(3)

```
enumerationType := "enum"
      "{" "#" <name> ( "," "#" <name> )* "}"
simpleTypeSpecifier := pathTypeName
          | enumerationType
literalCollection :=
  collectionKind "{" expressionListOrRange?  "}"
expressionListOrRange := expression
    ( "," expression)+ | (".." expression) )?
featureCall := pathName timeExpression?
           qualifiers?  featureCallParameters?
qualifiers := "[" actualParameterList "]"
declarator := <name> ( "," <name> )*
           ( ":" simpleTypeSpecifier )?  "|"
```

Grammaire d'OCL suite(4)

```
pathTypeName := <typeName> ( "::" <typeName> )*
pathName := ( <typeName> | <name> )
            ( "::" ( <typeName> | <name> ) )*
timeExpression := "@" <name>
actualParameterList :=
      expression ("," expression)*
logicalOperator :=
        "and" | "or" | "xor" | "implies"
collectionKind := "Set" | "Bag" | "Sequence"
               | "Collection"
relationalOperator := "=" | ">" | "<" | ">="
                   | "<=" | "<>"
addOperator := "+" | "-"
multiplyOperator := "*" | "/"
```


Grammaire d'OCL suite(5)

```
unaryOperator := "-" | "not"
typeName :=
  ( ["a"-"z"] | ["A"-"Z"] | "_" )
  ( ["a"-"z"] | ["0"-"9"] | ["A"-"Z"] | "_")*
name :=
  ( ["a"-"z"] | ["A"-"Z"] | "_" )
  ( ["a"-"z"] | ["0"-"9"] | ["A"-"Z"] | "_")*
number := ["0"-"9"] (["0"-"9"])*
string := "'" ( (~["'","
","\n","\"])
  | ("
" ( ["n","t","b","r","f","
","'","\"]
  | ["0"-"7"] ( ["0"-"7"] )?
  | ["0"-"3"] ["0"-"7"] ["0"-"7"] ) ) )* "'"
```


Chapitre 4

Vérification des assertions : contrats, tests et preuves

1) Vérification et Validation
2) Approche contractuelle

Sommaire suite(1)

3) Preuves avec assertions
4) Exceptions vs assertions
5) Conclusion sur les assertions exécutables

Vérifications et Validations

Rappel :

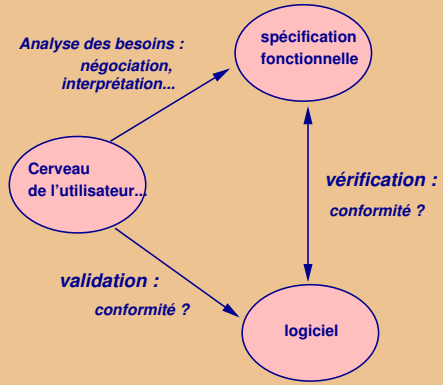
- validation : *construisons-nous le bon produit ?*
- vérification : *le construisons-nous bien ?*

⇒ pour vérifier, il faut une spécification précise, si possible formelle, du fonctionnement du logiciel.

Vérifications et Validations suite(1)

Mais ce qui est plus important, c'est de construire ce qui est attendu des utilisateurs...

Problème:
Comment le savoir ?



Vérifications et Validations suite(2)

Que cherche-t-on ?

- **validation :** *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...

Vérifications et Validations suite(3)

La validation ne devrait être effectuée que par les utilisateurs, sans tenir compte du cahier des charges, en utilisant la documentation du produit.

En pratique, la validation s'appuie souvent sur le cahier des charges et utilise des tests externes d'acceptation.

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

Techniques statiques : inconvénients

- ne portent pas forcément sur le code réel (qui peut évoluer),
- ne sont pas en situation réelle, avec tous les éléments en interaction,
- sauf pour les preuves, les vérifications statiques sont sommaires (typage, inspections, analyse...)
- 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, avec les éléments présents en exécution normale.
- plus à la portée du commun des programmeurs,

Techniques dynamiques : 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,

Les techniques statiques et dynamiques sont donc complémentaires

Approche contractuelle

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

Cette idée s'est aujourd'hui tellement imposée, qu'on confond souvent contrats et assertions.

Les contrats peuvent être moraux ou juridiques, entre les différents responsables d'un projet logiciel et selon différentes relations entre classes.

Contrats

Dans le langage courant, un contrat est :

- un **document** élaboré par **négociation**,
- entre plusieurs (souvent deux) **parties**,
- constitué de **dispositions** qui précisent les **droits** et les **devoirs** de chaque partie.

Responsabilités

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 :*

Responsablités suite(1)

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**,

Responsabilités suite(2)

En cas de problème, **on sait qui est responsable**, qui doit agir. C'est particulièrement important pour les **traitements d'exceptions.**

Les contrats doivent être équitables :

- ni **léonin**,
- ni **sinécure** !

Types de contrats pour PPO

- 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`.

Autres types de contrats : approche par composants

- contrats d'assemblage :
  l'assemblage respecte-il des propriétés de bonne architecture ?
- contrats de composition :
  l'organisation et le fonctionnement interne sont-ils compatibles avec les spécifications externes ?
- contrats extrafonctionnel : négociations possibles

Contrat de clientèle : exemple

Considérons la spécification suivante :

```
sqrt (x: Real): Real
  -- Square root of x
  define
    x >= 0 => result^2 = x
```

C'est une définition purement mathématique.

contrat de clientèle : exemple suite(1)

Aucun informaticien ne peut trouver un algorithme qui satisfasse cette définition, car la représentation des nombres réels n'est pas exacte dans un ordinateur :

$x = x'$

ne veut rien dire : *avec des réels, toujours utiliser des inégalités, sauf pour des valeurs entières pour lesquelles on a une représentation exacte : 0, 1, 100...*

Sans contrat précis, cette définition est irréaliste du point de vue du génie logiciel.

C'est un contrat léonin pour les clients !

contrat de clientèle : exemple suite(2)

Pour obtenir un contrat équitable, il faut une précision relative :

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


contrat de clientèle : exemple suite(3)

Remarques

- Ici, la spécification est complète
- Faire intervenir une définition mathématique de la précision relative comme $\frac{|\sqrt{x}-r|}{x} \leq \varepsilon$ ne serait pas vérifiable.
  Il faut une définition approchée, sans $\sqrt{x}$.

Contrat de clientèle

Contrat entre les clients et leurs fournisseurs :

	Devoirs	Droits
Client	appeler sqrt seulement si la précondition $(x \geq 0) \wedge (\varepsilon \geq 10^{-6})$ est satisfaite	obtenir le résultat correct avec la précision demandée, spécifiée dans la postcondition: $\lvert\sqrt{x} - r\rvert \leq \varepsilon \times x$
Fournisseur	Rendre le résultat correct avec la précision demandée spécifiée dans la postcondition	Refuser le calcul, (dégager sa responsabilité) si la précondition n'est pas satisfaite : $(x<0) \vee (\varepsilon<10^{-6})$

Comparaison de conditions logiques

Une condition $\mathcal{C}_2$ est **moins forte** (moins exigeante) que $\mathcal{C}_1$, ssi $\mathcal{C}_2$ **est impliquée par** $\mathcal{C}_1$:

$\mathcal{C}_2 \neq \mathcal{C}_1 \wedge \mathcal{C}_1 \Rightarrow \mathcal{C}_2$
par exemple : $\mathcal{C}_2 = \mathcal{C}_1 \vee ...$

Une condition $\mathcal{C}_2$ est **plus forte** (plus exigeante) que $\mathcal{C}_1$, ssi $\mathcal{C}_2$ **implique** $\mathcal{C}_1$:

$\mathcal{C}_2 \neq \mathcal{C}_1 \wedge \mathcal{C}_2 \Rightarrow \mathcal{C}_1$
par exemple : $\mathcal{C}_2 = \mathcal{C}_1 \wedge ...$

Précondition moins forte suite(1)

Si une **précondition est moins forte** ou **absente**,
ou **toujours vraie** cela **avantage les clients** :
il y a plus de cas acceptables d'utilisation!

Précondition moins forte suite(2)

Exemple de contrat léonin (pour les clients) :

```
sqrt (x: Real): Real
  -- Square Root of x
  -- pas de précondition ou
  pre: true
  ...
```

Comment traiter le cas x négatif ?

Postcondition plus forte

Si la **postcondition est plus forte** (plus exigeante), cela avantage aussi les clients :

il y a moins de résultats acceptables,
donc ils sont plus précis.

Postcondition plus forte: exemple

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


Sinécure

```
sqrt (x, eps:  Real):  Real
  -- Square Root of x with precision eps
  pre:
    0 >= 1 ...
    false
  body:
    -- n'importe quoi ...
  post:
    -- tout ce que l'on veut ou rien du tout...
    (result = 0) ∨ (result = 7) ...
```


Précondition vs test interne

Le code ne **doit pas** contenir de tests redondants avec les préconditions...

Précondition vs test interne suite(1)

```
sqrt (x, eps:  Real):  Real
  pre:
    x >= 0 ; eps >= 10^-6
  body:
    if x < 0 then ...
    elseif eps < 10 ^-6 then ...
    else -- calcul normal de la racine..
    fi
  post:
    abs (result^2-x) <= 2*eps*result
```

Précondition vs test interne suite(2)

Le compilateur peut décider de ne pas exécuter le corps des méthodes si leur précondition est violée avec certitude.

Le compilateur peut ne pas produire de code pour les méthodes dont la précondition est évaluée statiquement à faux.

Assertion vs exception

```
sqrt (x, eps:  Real):  Real
pre:
  x >= 0 ; eps >= 10^-6
body:     ...
post:
  abs (result ^2-x) <= 2*eps*result
...
```


Assertion vs exception suite(1)

```
calculInteractif local x:  Real
body:
  print("Taper un nombre positif :  ")
  readReal ( out x )
  if x >= 0
  then ...  sqrt(x, 10^-4) ...
  else
  -- prévenir l'utilisateur et éventuellement recommencer...
  -- Si l'utilisateur persiste, déclencher une exception d'échec...
  fi
```


Assertion vs exception suite(2)

⇒

*Les **assertions** sont une aide à la **correction***
*Les **exceptions** sont une aide à la **robustesse***

Assertion et Héritage

Nous avons déjà vu (« la vache folle ») qu'une classe héritière de plusieurs classes parentes hérite des invariants de tous ses classes parentes :

$$InvariantRel = InvariantLocal \wedge_i Invariant_{parent_i}$$

Redéfinition de méthodes lors d'un héritage

Les instances d'une **classe héritière** doivent pouvoir se **substituer** aux instances de **tous leurs ancêtres** : *nécessaire pour le polymorphisme.*

Cela suppose des **contraintes sémantiques** sur la nouvelle méthode :

- compatibilité des signatures,
- règles de typage : nonvariance, covariance, contravariance,
- compatibilité sémantique de substitution contrôlée par les assertions,

Assertion et Héritage suite(1)

Pour les assertions, il faut de nouvelles pré/post conditions qui vérifient :

- précondition moins exigeante, moins forte
- postcondition plus précise, plus forte

C'est la règle de l'**entonnoir**...

Assertion et Héritage suite(2)

Comment vérifier ces nouvelles exigences ?
⇒ *simplement par la syntaxe !!!*

Dans une méthode redéfinie, au lieu d'utiliser les mots clefs **pre:** et **post:**, il faut utiliser :

pre-else: `alternative Precondition`
`...`
post-then: `extra Postcondition`

Assertion et Héritage suite(3)

Dans de tels cas, les méthodes redéclarées auront comme conditions effectives :

```
pre:
  pre_1 or else ... pre_n
        or else alternative Precondition
  ...
post:
  post_1 and then ... post_n
         and then extra Postcondition
```


Assertion et Héritage suite(4)

Ainsi, nous avons :

```
ancestors precondition ⇒
          redeclared precondition
ancestors postcondition ⇐
          redeclared postcondition
```


Contrat d'héritage: exemple

```
class NewMath inherit Math redefine sqrt
feature
  sqrt (x, eps: Real): Real
  -- Square root of x, with precision eps
  pre-else:
    eps >= 10^-9
  post-then:
    -- Pas de nouvelle post-condition,
    -- l'ancienne est toujours active
```


Contrat d'héritage suite(1)

	Devoirs	Droits
Classe Parente	Fournir aux héritiers des services fiables, performants et réutilisables	Les héritiers ne doivent pas déformer la pensée initiale : invariants = droit d'auteur
Classe héritière	Maintenir les invariants, redéfinir en respectant la règle de l'entonnoir.	La classe parente doit fournir effectivement des services de qualité. Les assertions initiales doivent être respectées.

Preuves avec assertions

A partir de la **précondition et de propriétés héritées**
(souvent dérivées du langage de programmation sous-jacent et vérifiées statiquement par typage),
il faut, **par un raisonnement prouver la postcondition**.

(et non plus seulement la tester sur un jeu d'essai)

Preuve avec assertions : principes suite(1)

Technique de preuve introduite par C.A.R Hoare en 1969
(technique dite **axiomatique**).

Principe :

- utiliser des assertions, éventuellement **quantifiées** ($\forall$, $\exists$...)
- placer des assertions intermédiaires appelées **conséquents** (**assert**, **check**) qui sont déduites:
  - des assertions précédentes appelées **antécédents**,
  - et des règles sémantiques du langage utilisé.

Preuve assistée d'assertions

$\Rightarrow$ **possibilité de combiner**
test et preuve des assertions

- **preuve** (authentique) : toutes les assertions doivent être démontrées rigoureusement, avec des règles de déduction déduites de la sémantique du langage,
  (Hoare, Wirth pour le langage Pascal)
- **preuve assistée** : les assertions intermédiaires sont testées ou partiellement testées.

Remarque: les preuves assistées sont des tests en boîte blanche

Preuve assistée d'assertions suite(1)

$\Rightarrow$ *des techniques sophistiquées sont nécessaires pour tester des quantifications :*
échantillonnage, prise en compte de la signification d'une assertion, de l'état de stabilité de l'entité attachée...

Preuve assistée : recherche binaire

```
chercher (key: ELEM) : INTEGER is
require est_ordonné
ensure
  if ∃ i ∈ lower..upper | self @ i = key
    then Result = i
    else Result = lower -1
  end if
end -- chercher
est_ordonné : BOOLEAN
define Result = ∀ i ∈ lower .. upper-1 |
    self @ i <= self @ (i+1)
```


Exemple : recherche binaire suite(1)

```
require ...
local ...
do
  -- Déduire de require et ARRAY :
  check est_ordonné and lower <= upper end
  from ...
  invariant
    -- invariant de boucle (avant test de sortie)
  variant
    -- expression entière >= 0 et décroissante
  until trouvé xor bas > haut
  loop
    ...
  end loop
    ...
ensure ...
```


Exemple : recherche binaire suite(2)

```
chercher (key: ELEM) : INTEGER is
local
  bas,haut,milieu: INTEGER ; trouvé : BOOLEAN
do
  from
    bas:= lower ; haut:= upper
    milieu:= (bas + haut) / 2
    trouvé:= self @ milieu = key
  invariant -- invariant de boucle (avant test de sortie)
    ...
  until trouvé xor bas > haut
  loop ...
```


Exemple : recherche binaire suite(3)

```
  invariant -- invariant de boucle (avant test de sortie)
    lower <= milieu <= upper
    if trouvé
    then
      (self @ milieu = key) and
      lower <= bas <= milieu <= haut <= upper
    else -- key peut être dans [ self@bas .. self@haut ]
      not ∃ i ∈ lower .. bas-1 ∪ haut+1 .. upper
        | self @ i = key
    end if
  until ...
```

Exemple : recherche binaire suite(4)

```
...
  until trouvé xor bas > haut
  loop
    check -- invariant de début de boucle (après test)
      bas <= milieu <= haut
      not trouvé
      not ∃ i ∈ lower .. bas-1 ∪ haut+1 .. upper
          | self @ i = key
    end check
      ...
  end loop
  ...
```


Exemple : recherche binaire suite(5)

```
loop
  check ... end check
  milieu:= (bas + haut) / 2
  if self @ milieu = key then
    trouvé:= true
    check self @ milieu = key and trouvé end check
  else if self @ milieu < key then -- élément plus haut ?
    bas:= milieu + 1
    check not trouvé and
      not ∃ i ∈ lower .. bas-1 | self @ i = key
    end check
  else -- élément plus bas ?
    haut:= milieu - 1
    check not trouvé and
      not ∃ i ∈ haut+1 .. upper | self @ i = key
    end check
  end if
end loop
```


Preuve de terminaison

- soit l'élément est dans la table et l'on sort par *trouvé* = *true*,
- soit l'élément n'y est pas et l'on sort par *bas* > *haut :* (dans l'avant dernière itération, bas = milieu = haut)
- il suffit donc de prouver que (haut - bas) est décroissante, ce que l'on peut démontrer et/ou vérifier par le **variant** de la boucle :

NB: le langage Eiffel exige que le variant atteigne 0 à la sortie, donc `haut - bas +1` est le variant cherché

Preuve de terminaison suite(1)

```
from ...
invariant -- invariant de boucle (avant test de sortie)
variant
  -- expression entière >= 0 et
  -- décroissante qui atteint 0 à la sortie
  haut - bas + 1
until trouvé xor bas > haut
loop
...
end loop
```

Fin de la preuve assistée

```
from ... invariant ... variant ...
until ... loop ... end loop
check
  if trouvé
  then
    self @ milieu = key and
      milieu ∈ lower .. upper
  else
    bas > haut and
      not ∃ i ∈ lower .. upper
            | self @ i = key
  end if
end check
```


Fin de la preuve assistée suite(1)

```
if trouvé then Result := milieu
else Result := lower - 1 end if
ensure
  if ∃ i ∈ lower .. upper | self @ i = key
    then Result = i else Result = lower -1 en
end -- chercher
```

c.q.f.d.

Preuve authentique

La démarche est la même que précédemment, mais les assertions intermédiaires sont démontrées au lieu d'être testées.

Pour y parvenir, il faut des règles de déduction pour chaque construction du langage : affectation, énoncé de choix binaire, appel de méthode, itérations, etc.

Preuve authentique suite(1)

Ces règles sont :

- assez simples pour un langage comme Pascal,
- beaucoup plus compliquées pour un langage à objets avec héritage,
- inutilisables pour un langage laxiste ou pléthorique en constr tions redondantes.

Comme la plupart des langages actuels sont complexes et « tordus », les preuves authentiques de programmes deviennent inaccessibles ...

Exception vs Assertion

Les exceptions peuvent sont levées pendant l'exécution dans les cas suivants :

- erreur détectée par le support d'exécution,
- violation d'assertion (si les assertions sont armées),
- signaux émis par le système d'exploitation sous-jacent,
- exceptions explicites levées par le programmeur.

Traitements d'exceptions suite(1)

Chaque routine, méthode, bloc peut traiter les exceptions levées dans son contexte par une clause de récupération (**rescue**, **catch** clause...)

Par défaut, il peut il y avoir une clause de récupération héritée au niveau de l'objet, du composant...

Analyse des causes d'exception

L'analyse de la cause peut être effectuée par des classes, contrôleurs, constructions de service :

identification de la nature, du lieu, des responsabilités, etc

éventuellement par introspection...

Mécanisme de déroutement suite(1)

Lorsqu'une exception est levée, le flux de contrôle normal est dérouté vers la clause de récupération...

A la fin du traitement, plusieurs lieux de continuation sont possibles: reprise, propagation...

$\Rightarrow$ d'où danger de programmes spaghettis...

Déroutement contrôlé par les contrats

Dans une programmation avec contrats, les déroutements ne doivent pas outrepasser les obligations des contrats...

Le déroutement doit être rigoureux et motivé...

une méthode ne peut que réussir ou échouer !!!

- soit elle réussit, et remplit son contrat : *sa post-condition est vraie*
  ⇒ *son appelant ignore les difficultés rencontrées*

Déroutement contrôlé par les contrats suite(1)

- soit elle échoue : *sa post-condition est fausse et le retour se fait dans l'appelant, en déclenchant une exception.*
  La clause de secours de l'appelant (si elle existe) sera exécutée en suivant les mêmes règles :
  ⇒ **propagation** possible jusqu'à la routine racine.

Déroutement contrôlé par les contrats suite(2)

- ou, si le problème peut être résolu dans la clause de secours, (après analyse), i.e. la précondition et l'invariant peut être restauré, une nouvelle chance est donnée pour reprendre l'exécution : clause (**retry**):
  la routine, le bloc reprend depuis le début pour tenter de remplir son contrat, usuellement en changeant de stratégie...

Compte-rendu vs Exceptions
Version avec compte-rendu

```
class DIALOGUE inherit STANDARD_FILES
feature

  maxAttempts :  INTEGER := 3
  exitCode    :  enum {ok, syntax, negative }
  lastPositive:  REAL -- last positive number read

  readPositiveReal ()
    -- update lastPositive and exitCode
```

Version avec compte-rendu suite(1)

```
  dialogueWithExitCode ()
    local tryNb:  INTEGER
    do
      ...  see next slide ...
    end
  end -- dialogueWithExitCode
end -- class DIALOGUE
```


Version avec compte-rendu suite(2)

```
do -- dialogueWithExitCode continued
  from tryNb := 0
  until exitCode=ok or else tryNb>maxAttempts
  loop
    put("Give a positive nb:  ")
    readPositiveReal()
    if exitCode = syntax
      then put("=> syntax error !\n")
    elseif exitCode = negative
      then put("=> negative nb !\n")
    elseif exitCode = ok
      then put(" Square_root = " +
        sqrt(lastPositive) + "\n")
    end
```


Version avec compte-rendu suite(3)

```
    if exitCode /= ok then
      if tryNb < maxAttempts -1 then
        put(" New attempt...\n")
      elseif tryNb < maxAttempts then
        put(" Last attempt...\n")
      else
        put(" Renonciation...\n")
      end
      tryNb := tryNb +1
    end
  end -- loop
end -- dialogueWithExitCode
```


Version avec exception

```
class DIALOGUE inherit ...
feature
  maxAttempts:  INTEGER := 3
  lastPositive:  REAL
  readPositiveReal ()
    -- update lastPositive and raise exceptions
  dialogueWithExceptions ()
    local tryNb:  INTEGER
    do ...  see next slide ...
    rescue ...  see next slide ...  end
end -- class DIALOGUE
```

version avec exception suite(1)

```
do -- dialogueWithExceptions continued
   put("Give a positive nb:  ")
   lastPositive := readPositiveReal()
   put(" Sqrt = ")+sqrt(lastPositive)+"\n"
```


version avec exception suite(2)

```
rescue
   if exception_name.is_equal("syntax")
      then
         put("=> syntax error !\n")
   elseif exception_name.is_equal("negative")
      then
         put("=> negative nb !\n")
   end
```


version avec exception suite(3)

```
   if tryNb < maxAttempts -1 then
        put(" New attempt...\n")
        tryNb := tryNb +1
        retry
   elseif tryNb < maxAttempts then
        put(" Last attempt...\n")
        tryNb := tryNb +1
        retry
   else
     put(" Propagation...\n")
   end
end -- dialogueWithExceptions
```


Exceptions vs Assertions

Dans les deux versions précédentes, le contrat entre les méthodes `dialogue` et `sqrt` est satisfait

```
class SINGLE_MATH
feature
epsilon:  REAL is 10^-6
sqrt (x:  REAL): REAL
   require x >= 0
   ensure abs(Result^2 -x) <=
        2 * epsilon * Result
end -- class SINGLE_MATH
```

Exceptions vs Assertions suite(1)

Ici, lever une exception serait une erreur méthodologique !!!

- assertions sont faites pour détecter les erreurs de programmation,
- exceptions détectent des situations anormales durant l'exécu erreurs d'utilisation, signaux, manque de performance...

Conclusion sur les assertions exécutables

Qu'est-ce qu'un test ?

Une expérience d'exécution, pour mettre en évidence un défaut ou une erreur (conception, programmation).

Il faut aussi :

- un diagnostic : quel est le défaut ?
  $\Rightarrow$ besoin d'**oracle** qui indique si le résultat est juste, si l'expérience est conforme aux spécifications.
- localisation : où est la cause du défaut ?

Tests avec assertions exécutables suite(1)

$\Rightarrow$ intérêt des **assertions embarquées** dans le code, qui réalisent des diagnostics et des localisations

$\Rightarrow$ intérêt des **tests embarqués** :
classes, composants auto-testables...

Intérêt des assertions exécutables

L'écriture de **tests est indispensable, mais très fastidieuse**, même avec des outils (cf infrastructure `check`) et est redondante avec les spécifications (extreme programming)

Intérêt des assertions exécutables suite(1)

L'écriture de **spécifications formelles** est indispensable pour les preuves, mais plus difficile que l'utilisation des assertions.

Les preuves sans outils sont longues et difficiles.

Les preuves avec outils d'aide (prouveurs de théorèmes) ne prouvent que des propriétés des spécifications formelles, **pas du code réel**.

L'approche la plus efficace est la **génération automatique de tests** à partir des spécifications formelles.

Intérêt des assertions exécutables suite(2)

Les tests externes **ne localisent pas la cause des défauts** (exemple `check`).

Il faut ensuite trouver la cause : **débogage**, long et difficile, même avec des superoutils graphiques comme `ddd`.

Un test sans identification automatique de la cause d'échec ne peut pas s'intègrer dans un processus logiciel de type « autonomic computing » (logiciel qui évolue en cours d'exécution, de manière automatique et autonome)

Intérêt des assertions exécutables suite(3)

Les asssertions exécutables donnent des réponses positives à toutes ces questions :

- spécifications faciles à écrire et à comprendre,
- oracles des tests, débrayables ou persistants dans le code,
- bonne localisation de la cause des problèmes (tests internes en boite blanche),
- bonne intégration avec les traitements d'exception : veille de sécurité, d'intégrité des propriétés essentielles,