

MGL7160 Méthodes formelles et semi-formelles

OCL partie III

Roger Villemaire

Département d'informatique
UQAM

28 janvier 2014

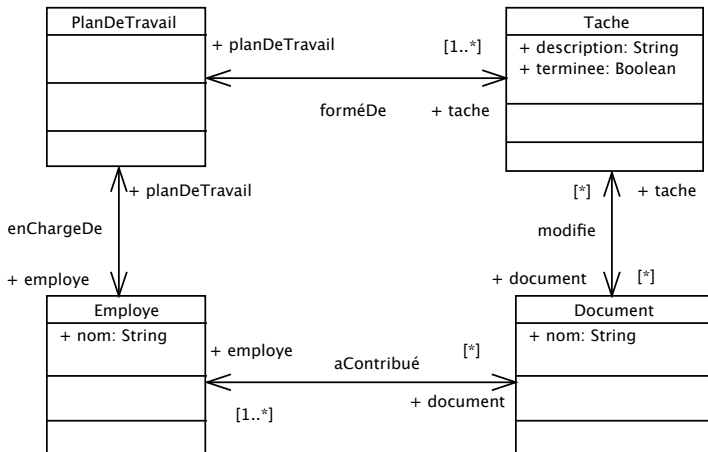
Plan

- 1 Navigation
- 2 Opérations, pré et post-conditions
- 3 Programmation par contrats

Plan

- 1 Navigation
- 2 Opérations, pré et post-conditions
- 3 Programmation par contrats

Exemple



Collect

- L'opération `collect` retourne la collection des valeurs de l'expression.
- `Sequence{1,2,0,3} -> collect(c: c.mod(2)) = Sequence{1,0,0,1}`
- Mais en fait si l'expression s'évalue à une collection, on n'obtient pas une collection de collections mais plutôt une collection de valeurs.
- Dans le contexte `PlanDeTravail`,
 - `tache` est une collection de `Taches`.
 - `tache -> collect(document)` est une collection de `Document`.
- La notation `tache.document` est aussi permise, c'est un *collect implicite*.

Collect

- L'opération `collect` retourne la collection des valeurs de l'expression.
- `Sequence{1,2,0,3} -> collect(c: c.mod(2)) = Sequence{1,0,0,1}`
- Mais en fait si l'expression s'évalue à une collection, on n'obtient pas une collection de collections mais plutôt une collection de valeurs.
- Dans le contexte `PlanDeTravail`,
 - `tache` est une collection de `Taches`.
 - `tache -> collect(document)` est une collection de `Document`.
- La notation `tache.document` est aussi permise, c'est un *collect implicite*.

Collect

- L'opération `collect` retourne la collection des valeurs de l'expression.
- `Sequence{1,2,0,3} -> collect(c: c.mod(2)) = Sequence{1,0,0,1}`
- Mais en fait si l'expression s'évalue à une collection, on n'obtient pas une collection de collections mais plutôt une collection de valeurs.
- Dans le contexte `PlanDeTravail`,
 - `tache` est une collection de `Taches`.
 - `tache -> collect(document)` est une collection de `Document`.
- La notation `tache.document` est aussi permise, c'est un *collect implicite*.

Collect

- L'opération `collect` retourne la collection des valeurs de l'expression.
- `Sequence{1,2,0,3} -> collect(c: c.mod(2)) = Sequence{1,0,0,1}`
- Mais en fait si l'expression s'évalue à une collection, on n'obtient pas une collection de collections mais plutôt une collection de valeurs.
- Dans le contexte `PlanDeTravail`,
 - `tache` est une collection de `Taches`.
 - `tache -> collect(document)` est une collection de `Document`.
- La notation `tache.document` est aussi permise, c'est un *collect implicite*.

Collect

- L'opération `collect` retourne la collection des valeurs de l'expression.
- `Sequence{1,2,0,3} -> collect(c: c.mod(2)) = Sequence{1,0,0,1}`
- Mais en fait si l'expression s'évalue à une collection, on n'obtient pas une collection de collections mais plutôt une collection de valeurs.
- Dans le contexte `PlanDeTravail`,
 - `tache` est une collection de `Taches`.
 - `tache -> collect(document)` est une collection de `Document`.
- La notation `tache.document` est aussi permise, c'est un *collect implicite*.

Collect

- L'opération `collect` retourne la collection des valeurs de l'expression.
- `Sequence{1,2,0,3} -> collect(c: c.mod(2)) = Sequence{1,0,0,1}`
- Mais en fait si l'expression s'évalue à une collection, on n'obtient pas une collection de collections mais plutôt une collection de valeurs.
- Dans le contexte `PlanDeTravail`,
 - `tache` est une collection de `Taches`.
 - `tache -> collect(document)` est une collection de `Document`.
- La notation `tache.document` est aussi permise, c'est un *collect implicite*.

Collect

- L'opération `collect` retourne la collection des valeurs de l'expression.
- `Sequence{1,2,0,3} -> collect(c: c.mod(2)) = Sequence{1,0,0,1}`
- Mais en fait si l'expression s'évalue à une collection, on n'obtient pas une collection de collections mais plutôt une collection de valeurs.
- Dans le contexte `PlanDeTravail`,
 - `tache` est une collection de `Taches`.
 - `tache -> collect(document)` est une collection de `Document`.
- La notation `tache.document` est aussi permise, c'est un *collect implicite*.

Collect

- L'opération `collect` retourne la collection des valeurs de l'expression.
- `Sequence{1,2,0,3} -> collect(c: c.mod(2)) = Sequence{1,0,0,1}`
- Mais en fait si l'expression s'évalue à une collection, on n'obtient pas une collection de collections mais plutôt une collection de valeurs.
- Dans le contexte `PlanDeTravail`,
 - `tache` est une collection de `Taches`.
 - `tache -> collect(document)` est une collection de `Document`.
- La notation `tache.document` est aussi permise, c'est un *collect implicite*.

Collect

- L'opération `collect` retourne la collection des valeurs de l'expression.
- `Sequence{1,2,0,3} -> collect(c: c.mod(2)) = Sequence{1,0,0,1}`
- Mais en fait si l'expression s'évalue à une collection, on n'obtient pas une collection de collections mais plutôt une collection de valeurs.
- Dans le contexte `PlanDeTravail`,
 - `tache` est une collection de `Taches`.
 - `tache -> collect(document)` est une collection de `Document`.
- La notation `tache.document` est aussi permise, c'est un *collect implicite*.

Modèle Relationnel

- Chaque navigation d'une association UML représente une relation.
- Lorsqu'on navigue des associations consécutives, on effectue la composition des relations.
 - `planDeTravail -> {t1,t2,t3},`
 - `t1 -> {d1,d2},`
 - `t2 -> {d3,d4,d5},`
 - `t3 -> {d6},`
 - `planDeTravail -> {d1,d2,d3,d4,d5,d6}.`

Modèle Relationnel

- Chaque navigation d'une association UML représente une relation.
- Lorsqu'on navigue des associations consécutives, on effectue la composition des relations.
 - `planDeTravail -> {t1,t2,t3},`
 - `t1 -> {d1,d2},`
 - `t2 -> {d3,d4,d5},`
 - `t3 -> {d6},`
 - `planDeTravail -> {d1,d2,d3,d4,d5,d6}.`

Modèle Relationnel

- Chaque navigation d'une association UML représente une relation.
- Lorsqu'on navigue des associations consécutives, on effectue la composition des relations.

- `planDeTravail -> {t1,t2,t3},`
- `t1 -> {d1,d2},`
- `t2 -> {d3,d4,d5},`
- `t3 -> {d6},`
- `planDeTravail -> {d1,d2,d3,d4,d5,d6}.`

Modèle Relationnel

- Chaque navigation d'une association UML représente une relation.
- Lorsqu'on navigue des associations consécutives, on effectue la composition des relations.
 - `planDeTravail -> {t1,t2,t3},`
 - `t1 -> {d1,d2},`
 - `t2 -> {d3,d4,d5},`
 - `t3 -> {d6},`
 - `planDeTravail -> {d1,d2,d3,d4,d5,d6}.`

Modèle Relationnel

- Chaque navigation d'une association UML représente une relation.
- Lorsqu'on navigue des associations consécutives, on effectue la composition des relations.
 - `planDeTravail -> {t1,t2,t3},`
 - `t1 -> {d1,d2},`
 - `t2 -> {d3,d4,d5},`
 - `t3 -> {d6},`
 - `planDeTravail -> {d1,d2,d3,d4,d5,d6}.`

Modèle Relationnel

- Chaque navigation d'une association UML représente une relation.
- Lorsqu'on navigue des associations consécutives, on effectue la composition des relations.
 - `planDeTravail -> {t1,t2,t3},`
 - `t1 -> {d1,d2},`
 - `t2 -> {d3,d4,d5},`
 - `t3 -> {d6},`
 - `planDeTravail -> {d1,d2,d3,d4,d5,d6}.`

Plan

- 1 Navigation
- 2 Opérations, pré et post-conditions**
- 3 Programmation par contrats

pré- post-conditions

- Une opération est définie par son type de retour ainsi que les types de ses arguments. OCL permet d'associer à une opération UML :
 - des pré-conditions : propriétés qui devraient toujours être vérifiées *avant* l'appel de l'opération.
 - des post-conditions : propriétés qui devraient toujours être vérifiées *après* l'appel de l'opération.

pré- post-conditions

- Une opération est définie par son type de retour ainsi que les types de ses arguments. OCL permet d'associer à une opération UML :
 - des pré-conditions : propriétés qui devraient toujours être vérifiées *avant* l'appel de l'opération.
 - des post-conditions : propriétés qui devraient toujours être vérifiées *après* l'appel de l'opération.

pré- post-conditions

- Une opération est définie par son type de retour ainsi que les types de ses arguments. OCL permet d'associer à une opération UML :
 - des pré-conditions : propriétés qui devraient toujours être vérifiées *avant* l'appel de l'opération.
 - des post-conditions : propriétés qui devraient toujours être vérifiées *après* l'appel de l'opération.

pré- post-conditions

- Une opération est définie par son type de retour ainsi que les types de ses arguments. OCL permet d'associer à une opération UML :
 - des pré-conditions : propriétés qui devraient toujours être vérifiées *avant* l'appel de l'opération.
 - des post-conditions : propriétés qui devraient toujours être vérifiées *après* l'appel de l'opération.

post-conditions

- Dans une post-condition, en plus des valeurs des attributs après l'appel on peut faire référence à :
 - `result`, la valeur de retour de l'opération,
 - `attribut@pre`, la valeur de l'attribut *avant* l'appel,
 - `a@pre`, l'ancienne valeur de `a`.
 - `a.b@pre`, l'ancienne valeur du `b` pour le `a` actuel.
 - `a@pre.b`, le `b` actuel de l'ancienne valeur de `a`.
 - `a@pre.b@pre`, l'ancien `b` de l'ancien `a`.
 - `ref.isNew()`, permet de savoir si la référence a été créée.

post-conditions

- Dans une post-condition, en plus des valeurs des attributs après l'appel on peut faire référence à :
 - `result`, la valeur de retour de l'opération,
 - `attribut@pre`, la valeur de l'attribut *avant* l'appel,
 - `a@pre`, l'ancienne valeur de `a`.
 - `a.b@pre`, l'ancienne valeur du `b` pour le `a` actuel.
 - `a@pre.b`, le `b` actuel de l'ancienne valeur de `a`.
 - `a@pre.b@pre`, l'ancien `b` de l'ancien `a`.
 - `ref.isNew()`, permet de savoir si la référence a été créée.

post-conditions

- Dans une post-condition, en plus des valeurs des attributs après l'appel on peut faire référence à :
 - `result`, la valeur de retour de l'opération,
 - `attribut@pre`, la valeur de l'attribut *avant* l'appel,
 - `a@pre`, l'ancienne valeur de `a`.
 - `a.b@pre`, l'ancienne valeur du `b` pour le `a` actuel.
 - `a@pre.b`, le `b` actuel de l'ancienne valeur de `a`.
 - `a@pre.b@pre`, l'ancien `b` de l'ancien `a`.
 - `ref.isNew()`, permet de savoir si la référence a été créée.

post-conditions

- Dans une post-condition, en plus des valeurs des attributs après l'appel on peut faire référence à :
 - `result`, la valeur de retour de l'opération,
 - `attribut@pre`, la valeur de l'attribut *avant* l'appel,
 - `a@pre`, l'ancienne valeur de `a`.
 - `a.b@pre`, l'ancienne valeur du `b` pour le `a` actuel.
 - `a@pre.b`, le `b` actuel de l'ancienne valeur de `a`.
 - `a@pre.b@pre`, l'ancien `b` de l'ancien `a`.
 - `ref.isNew()`, permet de savoir si la référence a été créée.

post-conditions

- Dans une post-condition, en plus des valeurs des attributs après l'appel on peut faire référence à :
 - `result`, la valeur de retour de l'opération,
 - `attribut@pre`, la valeur de l'attribut *avant* l'appel,
 - `a@pre`, l'ancienne valeur de `a`.
 - `a.b@pre`, l'ancienne valeur du `b` pour le `a` actuel.
 - `a@pre.b`, le `b` actuel de l'ancienne valeur de `a`.
 - `a@pre.b@pre`, l'ancien `b` de l'ancien `a`.
 - `ref.isNew()`, permet de savoir si la référence a été créée.

post-conditions

- Dans une post-condition, en plus des valeurs des attributs après l'appel on peut faire référence à :
 - `result`, la valeur de retour de l'opération,
 - `attribut@pre`, la valeur de l'attribut *avant* l'appel,
 - `a@pre`, l'ancienne valeur de `a`.
 - `a.b@pre`, l'ancienne valeur du `b` pour le `a` actuel.
 - `a@pre.b`, le `b` actuel de l'ancienne valeur de `a`.
 - `a@pre.b@pre`, l'ancien `b` de l'ancien `a`.
 - `ref.isNew()`, permet de savoir si la référence a été créée.

post-conditions

- Dans une post-condition, en plus des valeurs des attributs après l'appel on peut faire référence à :
 - `result`, la valeur de retour de l'opération,
 - `attribut@pre`, la valeur de l'attribut *avant* l'appel,
 - `a@pre`, l'ancienne valeur de `a`.
 - `a.b@pre`, l'ancienne valeur du `b` pour le `a` actuel.
 - `a@pre.b`, le `b` actuel de l'ancienne valeur de `a`.
 - `a@pre.b@pre`, l'ancien `b` de l'ancien `a`.
 - `ref.isNew()`, permet de savoir si la référence a été créée.

post-conditions

- Dans une post-condition, en plus des valeurs des attributs après l'appel on peut faire référence à :
 - `result`, la valeur de retour de l'opération,
 - `attribut@pre`, la valeur de l'attribut *avant* l'appel,
 - `a@pre`, l'ancienne valeur de `a`.
 - `a.b@pre`, l'ancienne valeur du `b` pour le `a` actuel.
 - `a@pre.b`, le `b` actuel de l'ancienne valeur de `a`.
 - `a@pre.b@pre`, l'ancien `b` de l'ancien `a`.
 - `ref.oclIsNew()`, permet de savoir si la référence a été créée.

post-conditions

- Dans une post-condition, en plus des valeurs des attributs après l'appel on peut faire référence à :
 - `result`, la valeur de retour de l'opération,
 - `attribut@pre`, la valeur de l'attribut *avant* l'appel,
 - `a@pre`, l'ancienne valeur de `a`.
 - `a.b@pre`, l'ancienne valeur du `b` pour le `a` actuel.
 - `a@pre.b`, le `b` actuel de l'ancienne valeur de `a`.
 - `a@pre.b@pre`, l'ancien `b` de l'ancien `a`.
 - `ref.isNew()`, permet de savoir si la référence a été créée.

Autres usages d'OCL

- `def` : donner un nom parlant à une expression OCL pour pouvoir la réutiliser.
- `body` : donne le résultat d'une *requête* (opération qui ne change pas l'état du système).
- `init` : donne la valeur initiale d'un attribut.
- `derive` : donne la valeur d'un attribut *dérivé* (calculé à partir des autres).
- `let s : Integer = tache -> size() in` valeur locale.

Autres usages d'OCL

- `def` : donner un nom parlant à une expression OCL pour pouvoir la réutiliser.
- `body` : donne le résultat d'une *requête* (opération qui ne change pas l'état du système).
- `init` : donne la valeur initiale d'un attribut.
- `derive` : donne la valeur d'un attribut *dérivé* (calculé à partir des autres).
- `let s : Integer = tache -> size() in valeur locale.`

Autres usages d'OCL

- `def` : donner un nom parlant à une expression OCL pour pouvoir la réutiliser.
- `body` : donne le résultat d'une *requête* (opération qui ne change pas l'état du système).
- `init` : donne la valeur initiale d'un attribut.
- `derive` : donne la valeur d'un attribut *dérivé* (calculé à partir des autres).
- `let s : Integer = tache -> size() in valeur locale.`

Autres usages d'OCL

- `def` : donner un nom parlant à une expression OCL pour pouvoir la réutiliser.
- `body` : donne le résultat d'une *requête* (opération qui ne change pas l'état du système).
- `init` : donne la valeur initiale d'un attribut.
- `derive` : donne la valeur d'un attribut *dérivé* (calculé à partir des autres).
- `let s : Integer = tache -> size() in valeur locale.`

Autres usages d'OCL

- `def` : donner un nom parlant à une expression OCL pour pouvoir la réutiliser.
- `body` : donne le résultat d'une *requête* (opération qui ne change pas l'état du système).
- `init` : donne la valeur initiale d'un attribut.
- `derive` : donne la valeur d'un attribut *dérivé* (calculé à partir des autres).
- `let s : Integer = tache -> size() in valeur locale.`

Autres usages d'OCL

- `def` : donner un nom parlant à une expression OCL pour pouvoir la réutiliser.
- `body` : donne le résultat d'une *requête* (opération qui ne change pas l'état du système).
- `init` : donne la valeur initiale d'un attribut.
- `derive` : donne la valeur d'un attribut *dérivé* (calculé à partir des autres).
- `let s : Integer = tache -> size() in valeur locale.`

Plan

- 1 Navigation
- 2 Opérations, pré et post-conditions
- 3 Programmation par contrats**

Contrats

- Métaphore inspirée des contrats entre clients et fournisseurs introduite par Bertrand Meyer pour aider à réalisation des logiciels.
- Un invariant définit la signification d'une classe.
- Une pré-condition est
 - une obligation pour le client qui veut utiliser l'opération.
 - une garantie pour le fournisseur que l'opération sera utilisée dans un contexte correct.
- Une post-condition est
 - une obligation pour le fournisseur qui doit l'assurer.
 - une garantie pour le client qui utilise l'opération.

Contrats

- Métaphore inspirée des contrats entre clients et fournisseurs introduite par Bertrand Meyer pour aider à réalisation des logiciels.
- Un invariant définit la signification d'une classe.
- Une pré-condition est
 - une obligation pour le client qui veut utiliser l'opération.
 - une garantie pour le fournisseur que l'opération sera utilisée dans un contexte correct.
- Une post-condition est
 - une obligation pour le fournisseur qui doit l'assurer.
 - une garantie pour le client qui utilise l'opération.

Contrats

- Métaphore inspirée des contrats entre clients et fournisseurs introduite par Bertrand Meyer pour aider à réalisation des logiciels.
- Un invariant définit la signification d'une classe.
- Une pré-condition est
 - une obligation pour le client qui veut utiliser l'opération.
 - une garantie pour le fournisseur que l'opération sera utilisée dans un contexte correct.
- Une post-condition est
 - une obligation pour le fournisseur qui doit l'assurer.
 - une garantie pour le client qui utilise l'opération.

Contrats

- Métaphore inspirée des contrats entre clients et fournisseurs introduite par Bertrand Meyer pour aider à réalisation des logiciels.
- Un invariant définit la signification d'une classe.
- Une pré-condition est
 - une obligation pour le client qui veut utiliser l'opération.
 - une garantie pour le fournisseur que l'opération sera utilisée dans un contexte correct.
- Une post-condition est
 - une obligation pour le fournisseur qui doit l'assurer.
 - une garantie pour le client qui utilise l'opération.

Contrats

- Métaphore inspirée des contrats entre clients et fournisseurs introduite par Bertrand Meyer pour aider à réalisation des logiciels.
- Un invariant définit la signification d'une classe.
- Une pré-condition est
 - une obligation pour le client qui veut utiliser l'opération.
 - une garantie pour le fournisseur que l'opération sera utilisée dans un contexte correct.
- Une post-condition est
 - une obligation pour le fournisseur qui doit l'assurer.
 - une garantie pour le client qui utilise l'opération.

Contrats

- Métaphore inspirée des contrats entre clients et fournisseurs introduite par Bertrand Meyer pour aider à réalisation des logiciels.
- Un invariant définit la signification d'une classe.
- Une pré-condition est
 - une obligation pour le client qui veut utiliser l'opération.
 - une garantie pour le fournisseur que l'opération sera utilisée dans un contexte correct.
- Une post-condition est
 - une obligation pour le fournisseur qui doit l'assurer.
 - une garantie pour le client qui utilise l'opération.

Contrats

- Métaphore inspirée des contrats entre clients et fournisseurs introduite par Bertrand Meyer pour aider à réalisation des logiciels.
- Un invariant définit la signification d'une classe.
- Une pré-condition est
 - une obligation pour le client qui veut utiliser l'opération.
 - une garantie pour le fournisseur que l'opération sera utilisée dans un contexte correct.
- Une post-condition est
 - une obligation pour le fournisseur qui doit l'assurer.
 - une garantie pour le client qui utilise l'opération.

Contrats

- Métaphore inspirée des contrats entre clients et fournisseurs introduite par Bertrand Meyer pour aider à réalisation des logiciels.
- Un invariant définit la signification d'une classe.
- Une pré-condition est
 - une obligation pour le client qui veut utiliser l'opération.
 - une garantie pour le fournisseur que l'opération sera utilisée dans un contexte correct.
- Une post-condition est
 - une obligation pour le fournisseur qui doit l'assurer.
 - une garantie pour le client qui utilise l'opération.

Contrats

- Métaphore inspirée des contrats entre clients et fournisseurs introduite par Bertrand Meyer pour aider à réalisation des logiciels.
- Un invariant définit la signification d'une classe.
- Une pré-condition est
 - une obligation pour le client qui veut utiliser l'opération.
 - une garantie pour le fournisseur que l'opération sera utilisée dans un contexte correct.
- Une post-condition est
 - une obligation pour le fournisseur qui doit l'assurer.
 - une garantie pour le client qui utilise l'opération.