Laboratoire
IHDCB335 - Analyse et Modélisation des Systèmes d'Information

Nicolay Matthias
Demonceau Cédric

UNamur

1 Plateau de Jeu

1.1 Diagramme de classe minimaliste du jeu

Voici, un diagramme de classe UML qui fixe les éléments principaux du jeu, c'est à dire le jeu en lui même, les lessons et les niveaux.

La classe principale Jeu possède une ou plusieurs lessons, ces lessons possèdent elles même un ou plusieurs niveaux. Chaque niveau est le précédent ou le suivant d'un autre niveau.

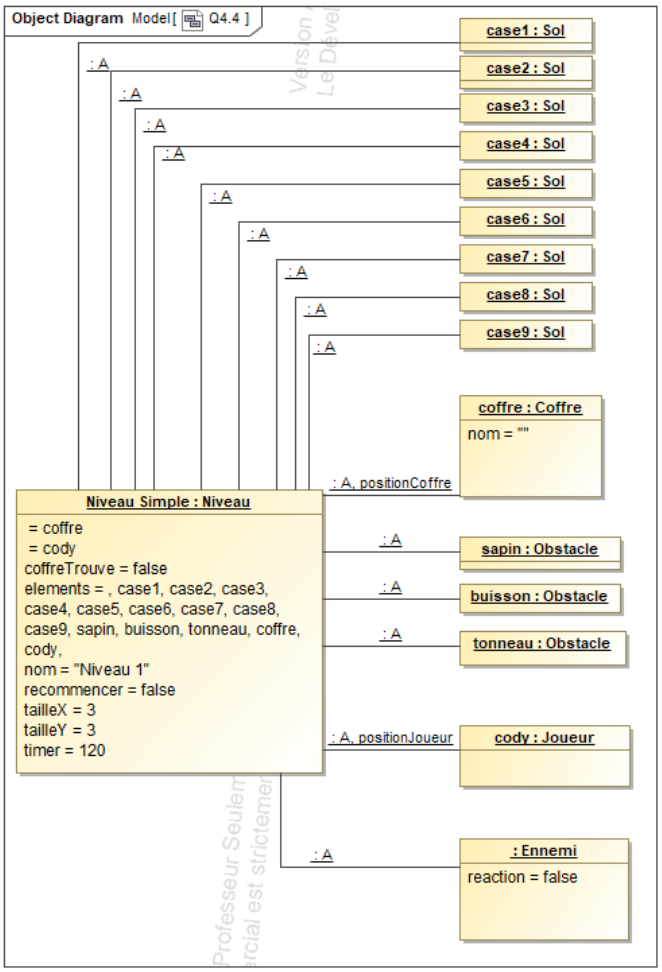

Figure 1: Diagramme de classe des éléments principaux du jeu

1.2 Version enrichie du diagramme de classe du jeu

Dans cette version du diagramme de classe 2, la classe Jeu est est maintenant associée a deux nouvelles classes.

La classe *ProfilUtilisateur*, qui sauvegarde le profil de chaque utilisateur et qui est elle même associée a *HistoriqueDeNiveau* d'une part et d'autre part à *Lesson*, par la classe d'association *Reussite*, qui donne le nombre de lessons réussies. Et la classe *Editeur* qui permet d'éditer un *Niveau* à la fois.

Cette classe *Editeur* est en assiciation avec *ElementNiveau* car il est possible d'éditer un niveau (et donc d'avoir besoins des éléments pour construire un niveau) sans avoir de lessons et de niveau existants.

ProfilUtilisateur est aussi associé à niveau car il faut pouvoir conserver les niveaux édités par l'utilisateur et à *NiveauEnCours* car il faut pouvoir conserver les données de performances de chaque niveau.

La composition se trouvant entre *Niveau* et *NiveauEnCours* est présente car il faut un *Niveau* pour avoir un *NiveauEnCours*. La classe *NiveauEnCours* est associée via la classe d'association *PositionJoueur* à *Joueur* car étant donné que Cody peut se déplacer sur le niveau, il faut pourvoir connaître sa position via *getElementAtPosition*.

La classe *Niveau* est quand à elle associée par une assiciation 1-1 à *Joueur* et *Coffre* car il ne peut y avoir qu'un seul joueur et un seul coffre par *Niveau*.

Elle est aussi associée à *ElementNiveau* qui possède 4 enfants directs (*Surface*, *Personnage*,*Obstacle* et *Teleporteur*).

Surface a comme enfants (*Sol*,*Pont* et *Coffre*), *Personnage* a (*Joueur* et *Ennemi*) et *Teleporteur* a (*Tunnel* et *Levier*).

Pour la classe *Sol*, le changement de *décors* se fait via une énumération *Theme*.

Dans la classe *Ennemi*, le boolean *reaction* permet d'avoir des ennemis agressifs ou non. La classe *Tunnel* est en double association sur elle même car il y a une entrée et une sortie.

Dans le cas contraire , la variable*isOpen* est a false et il n'y a qu'un tunnel fermé. Un *Tunnel* lorsqu'il est double (une entrée et une sortie) est en assiciation 1-1 avec un *Levier* qui permet de l'ouvrir.

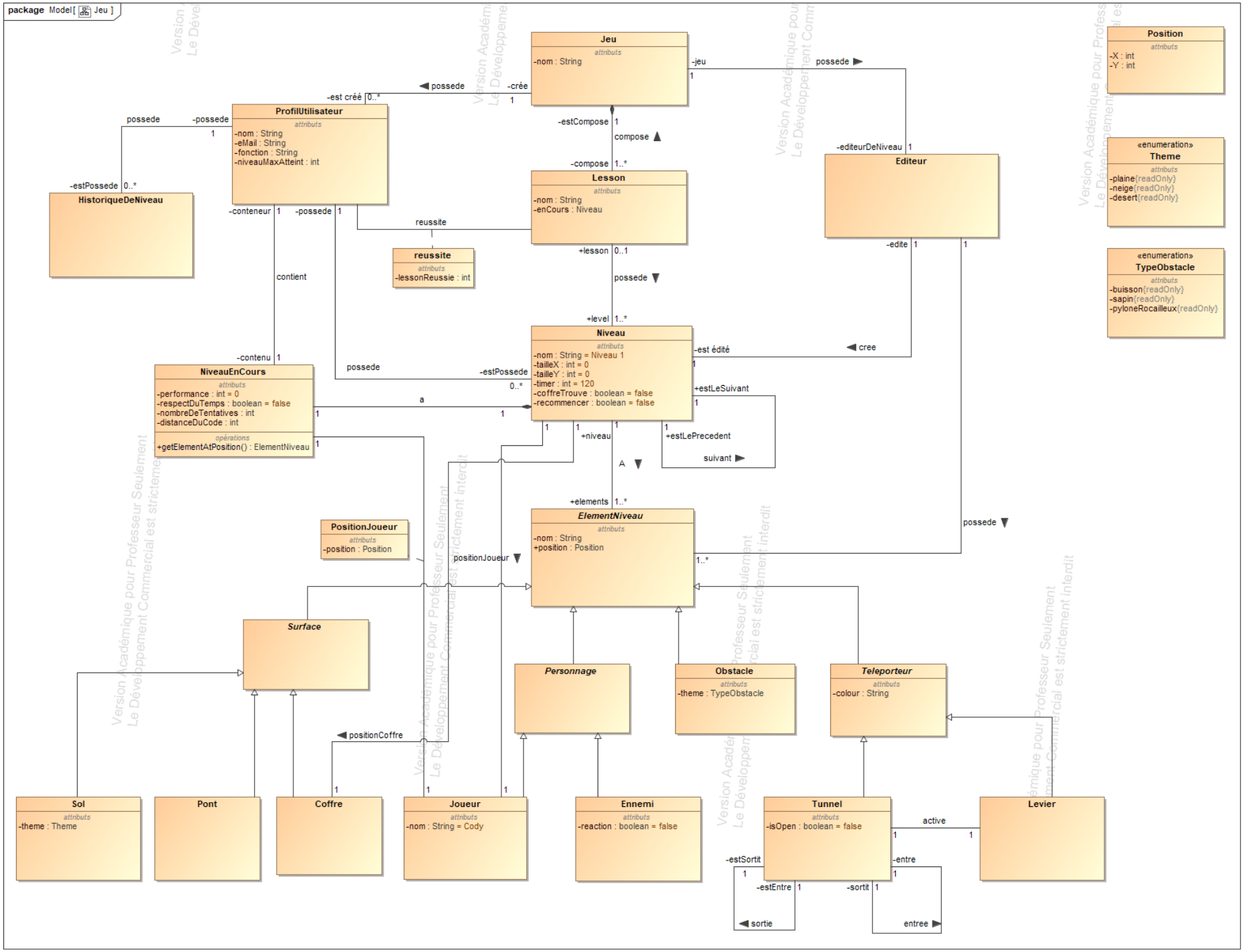

Figure 2: Diagramme enrichit

1.3 Ocl - Contraintes Ocl du diagramme de classe

Le fait que les coordonées d'une case ne peuvent excéder la taille de la carte se caractérise par cette contrainte Ocl:

```
context surface inv taille:
    0 < self.position.x < self.Niveau.tailleX
    && 0 < self.position.y < self.Niveau.tailleY
```

Listing 1: Contrainte sur les coordonées d'une case

La contrainte qu'un niveau ne peut contenir qu'un Cody et un coffre est exprimée dans le diagramme de classe comme repris à la figure 3.

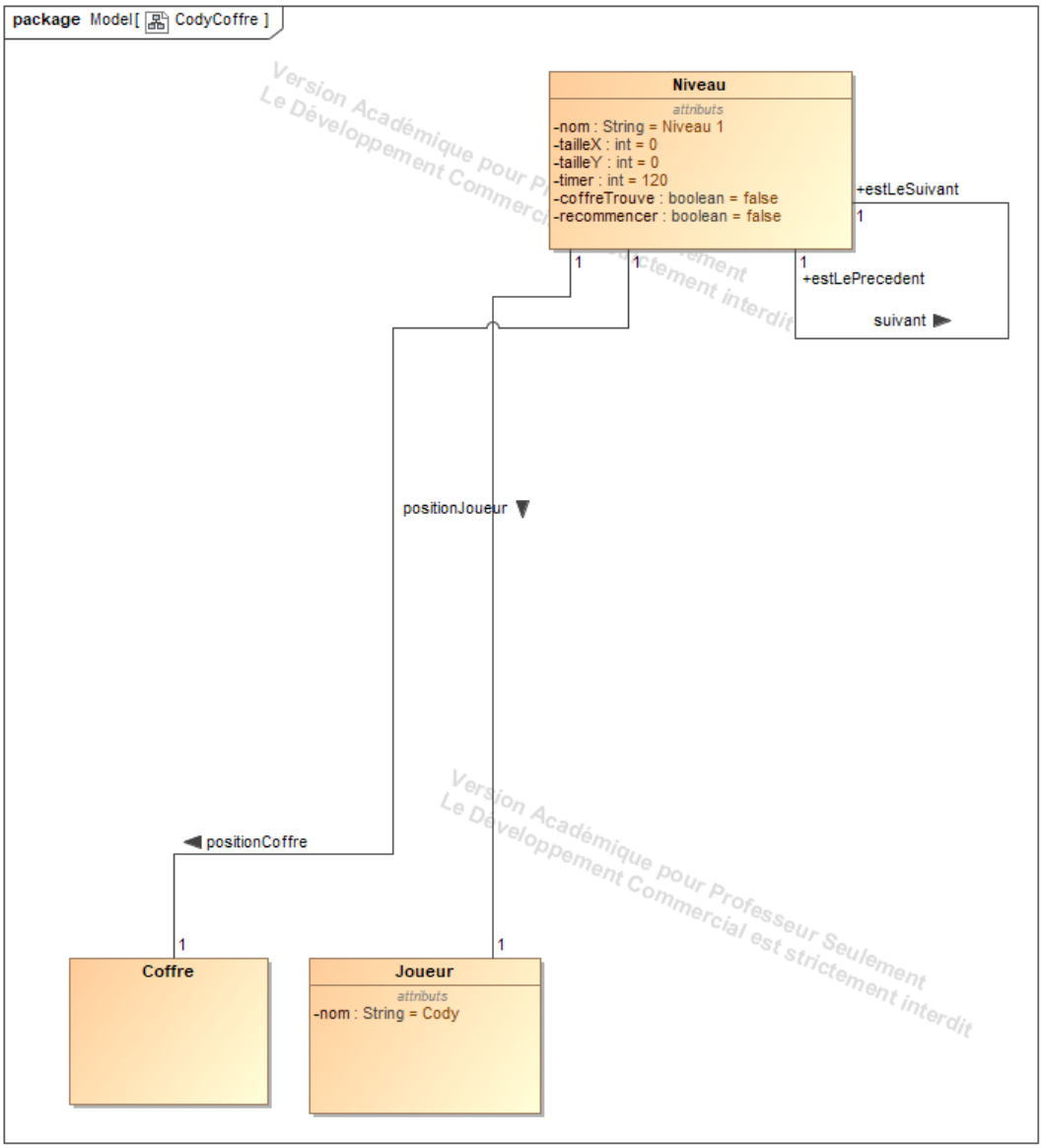

Figure 3: Contrainte de Cody et du coffre

La contrainte Ocl disant qu'un personnage ne peut se trouver sur un obstacle est exprimée comme suit:

```
context Niveau inv posobstacle:
    self.ElementNiveau->forall(p,o |
        p.isOclTypeOf(Personnage) and
        o.isOclTypeOf(Obstacle) and p.position.X <> o.position.X
            and p.position.Y <> o.position.Y)
```

Listing 2: Contrainte sur la position

La contrainte Ocl interdisant deux personnage de se trouver sur la même case est la suivante :

```
context Niveau inv posperso:
    self.>ElementNiveau->forall(p1,p2 |
        p1.isOclTypeOf(Personnage)
        and p2.isOclTypeOf(Personnage) and p1 <> p2 implies
            p1.position.X <> p2.position.X and
                p1.position.Y <> p2.position.Y)
```

Listing 3: Contrainte sur l'interdiction de deux personnages sur la même case

Le fait que le coffre doive se trouver sur une surface franchissable est caractérisé par la contrainte suivante:

```
context Niveau inv poscoffre:
    self.ElementNiveau->forall(a,b | a.isOclTypeOf(Coffre)
        and b.isOclTypeOf(Sol) and a.position.X = b.position.X
            and a.position.Y = b.position.Y)
```

Listing 4: Contrainte sur la position du coffre

La contrainte exprimant le fait que chaque niveau comporte soit une paire de tunnels de téléportation de même couleur, soit un tunnel unique d'une couleur mais qui est initialement fermé s'exprime comme suit:

```
context Niveau inv tunnel:
    (self.ElementNiveau->forall(t1,t2 | t1.isOclTypeOf(Tunnel)
        and t2.isOclTypeOf(Tunnel) and t1.couleur = t2.couleur
        and t1.isOpen = true and t2.isOpen = true) or
        self.ElementNiveau->forall(t1|
            t1.size() = 1 and t1.isOpen = false))
```

Listing 5: Contrainte sur les tunnels

La propriété qu'un levier ne peut être présent que si il existe un tunnel de la même couleur est caractérisée comme suit:

```
context Niveau inv levier:
    self.ElementNiveau->forall(l | l.isOclTypeOf(Levier)
        and self.ElementNiveau->exists(t | t.isOclTypeOf(Tunnel)
        and l.couleur = t.couleur))
```

Listing 6: Contrainte sur la présence d'un levier

La contrainte exprimant qu'un obstacle ne peut se trouver que sur une surface franchissable est la quivante:

```
context Niveau inv obstacles:
    self.ElementNiveau->forall(a,b| a.isOclTypeOf(Obstacle)
        and b.isOclTypeOf(Sol) and a.position.X == b.position.X
            and a.position.Y == b.position.Y)
```

Listing 7: Contrainte sur les obstacles

1.4 Diagramme d'objet d'un niveau

La propriété du jeu qui n'est pas satisfaite par ce niveau est celle de l'accesibilité du coffre par le joueur. En effet dans ce niveau, il n'est pas possible au joueur d'atteindre le coffre.

Le diagramme d'objet suivant décrit le niveau à la figure 3:

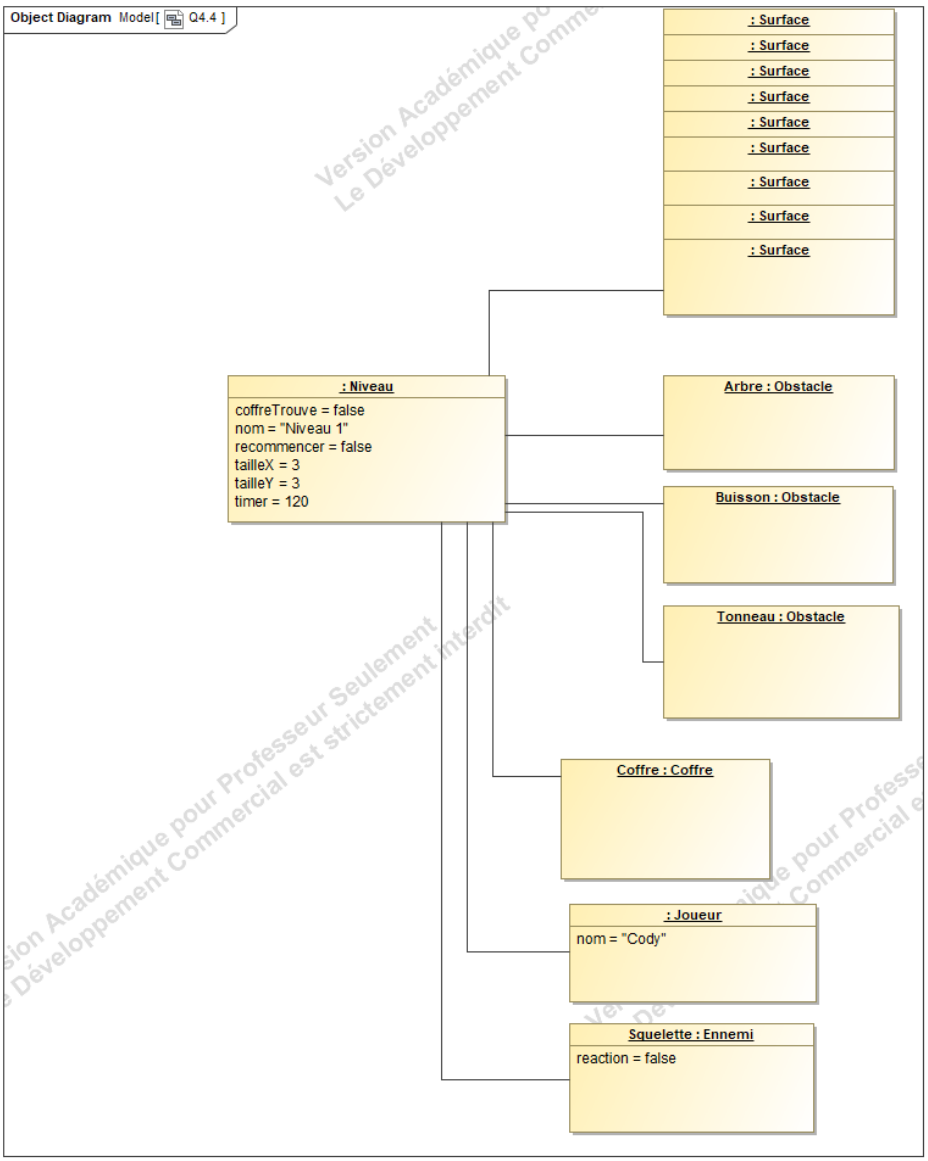

Figure 4: Diagramme d'objet de la figure 4.4

2 Play : Un langage d'Action

2.1 Diagramme d'un programme

Le diagramme présenté en figure 5, représente une première version du langage *Play*. Un programme étant «simplement» un ensemble de déclaration.

Une *Declaration* hérite d'une classe virtuelle *Named* qui permet de faire hériter, à l'ensemble des enfants de *Declaration*, une propriété *name*.

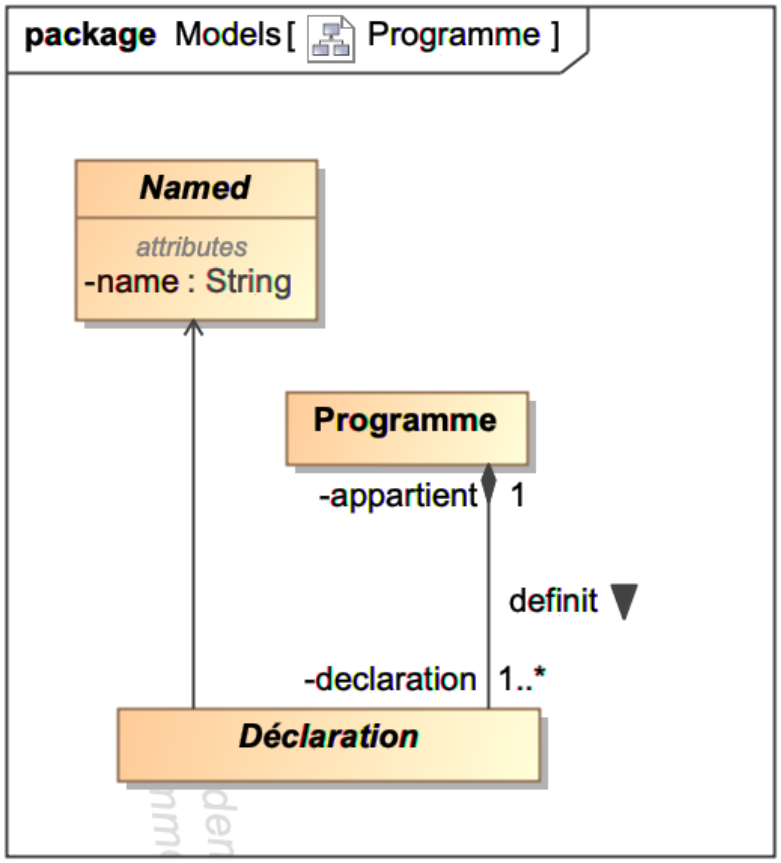

Figure 5: Diagramme de classe d'un programme

2.2 Modélisation du concept de déclaration

La classe *Déclaration* (Figure 5) est raffinée dans la figure 6. Dans cette figure, nous avons conservé les classes *Named* et *Programme* afin de faciliter la lecture du diagramme.

2.3 Raffinement du concept de Type

Le concept de *Type* est raffiné de la manière présentée à la figure 7. La classe *Type* est liée à plusieurs classes présentes dans la figure 6 (*Variable*, *Procédure* et *Paramètre*).

On peut remarquer que le type *void* n'est pas considéré comme un type primitif et est donc directement relié à la classe *Type*.

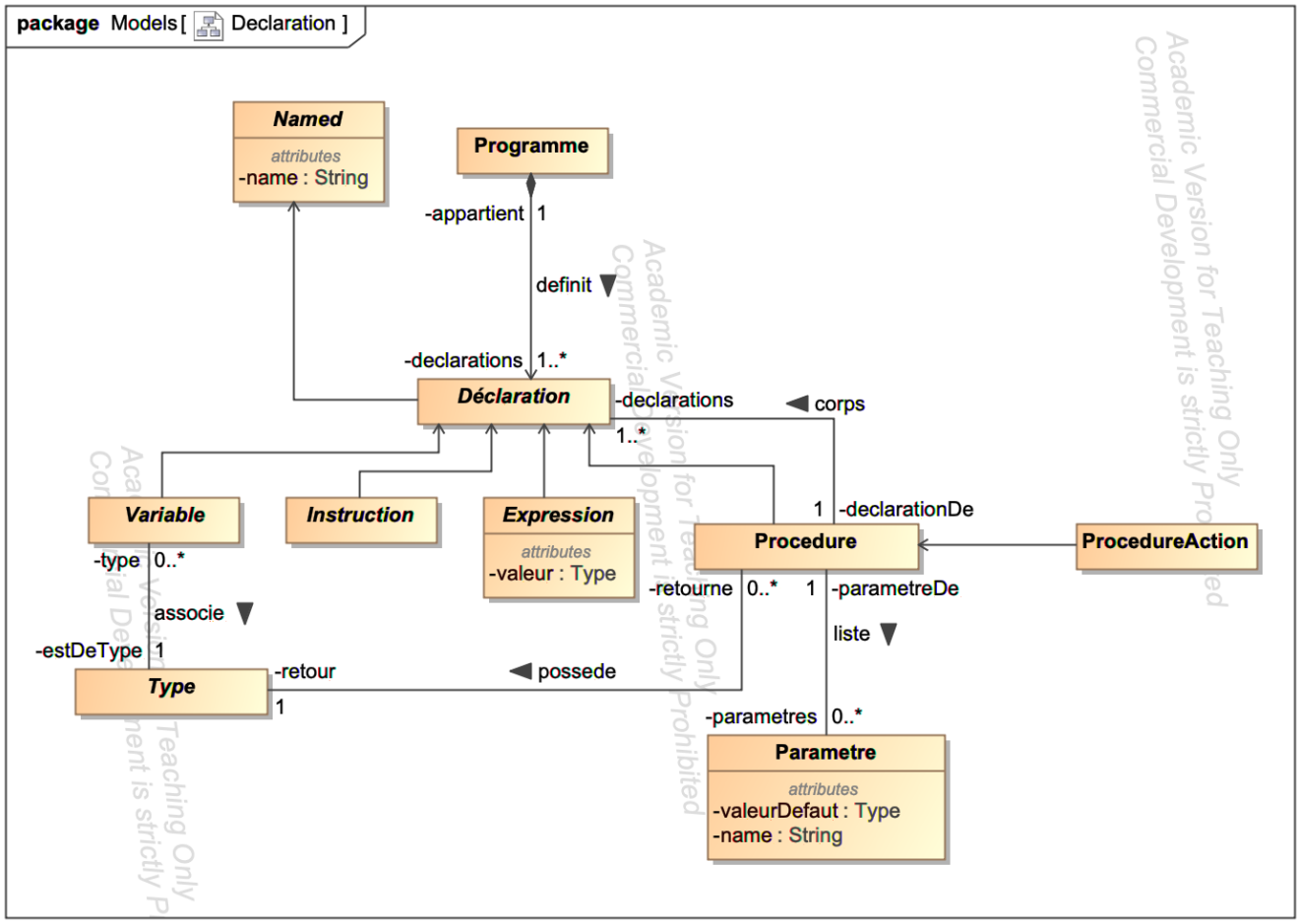

Figure 6: Diagramme de classe d'une déclaration

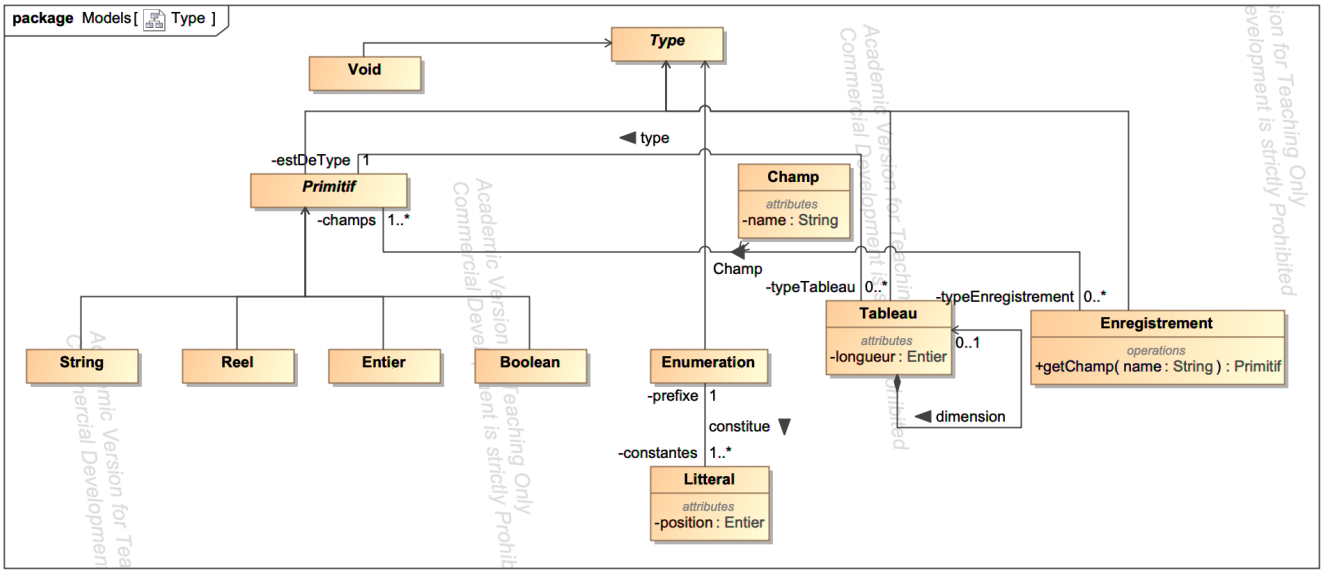

Figure 7: Diagramme de classe d'un type

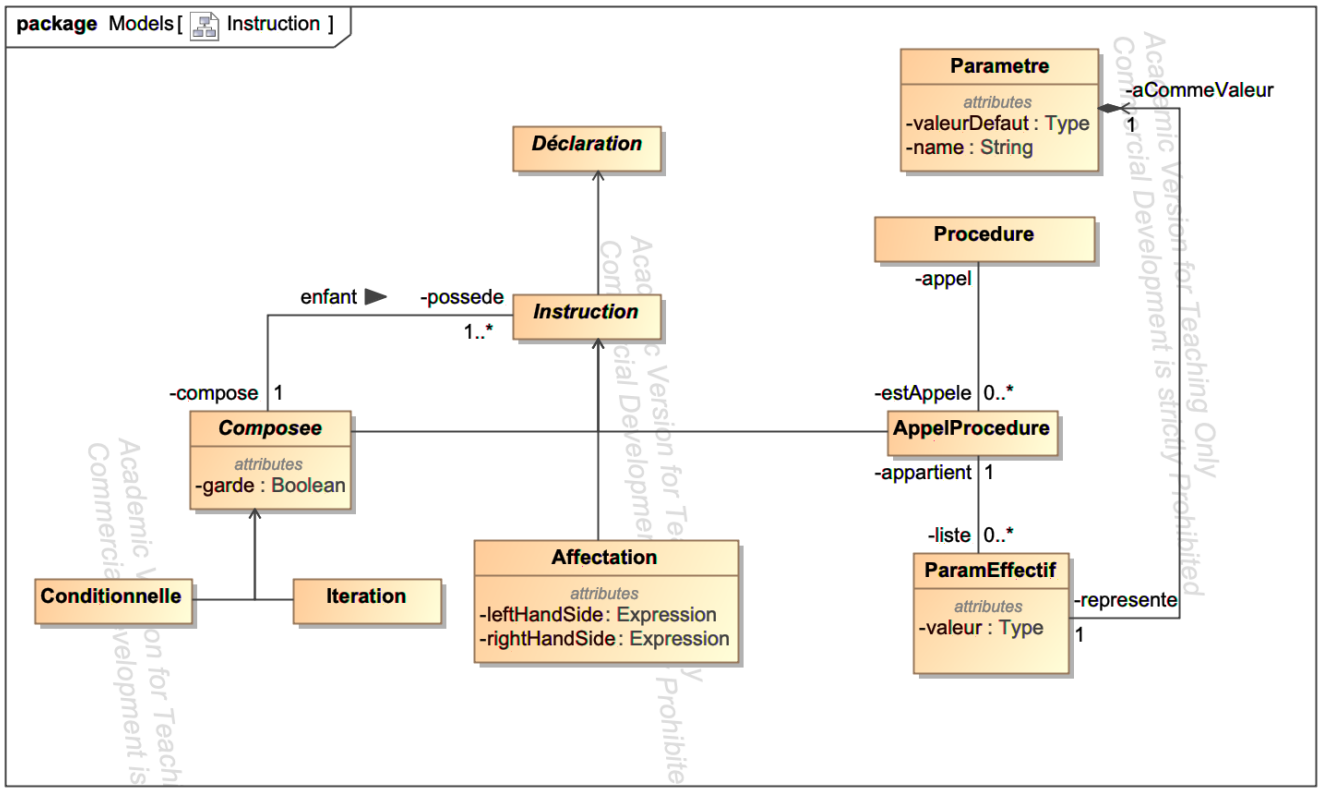

Figure 8: Diagramme de classe d'une instruction

2.4 Raffinement du concept d'Instruction

Présenté à la figure 8, le raffinement d'*Instruction* montre qu'une *Instruction* possède trois enfants direct (*Composée*, *Affectation* et *AppelProcédure*).

Le diagramme montre aussi qu'une Instruction est enfant de *Déclaration* (comme présenté dans la figure 6). Cela permet d'éviter d'avoir des tableaux de *Void*.

Sur le diagramme, nous pouvons remarquer la classe *Parametres*, celle-ci a été représentée dans la figure 6.

2.5 Raffinement du concept d'Expression

Le concept d'expression est présenté à l'aide de la figure 9. Une *Expression* étant elle aussi enfant de *Déclaration* qui était présenté à la figure 6.

2.6 OCL - Contrainte d'unicité

La contrainte suivante (Listing 8) permet de vérifier que le nom d'une procédure est unique.

```
context Procedure inv:
  Procedure.allInstance()->forAll(p1, p2
    | p1 <> p2 implies p1.name <> p2.name)
```

Listing 8: Unicité des noms des procédures

Afin de vérifier que la procédure **Cody** est bien unique, nous utilisons la contrainte OCL ci-dessous (Listing 9).

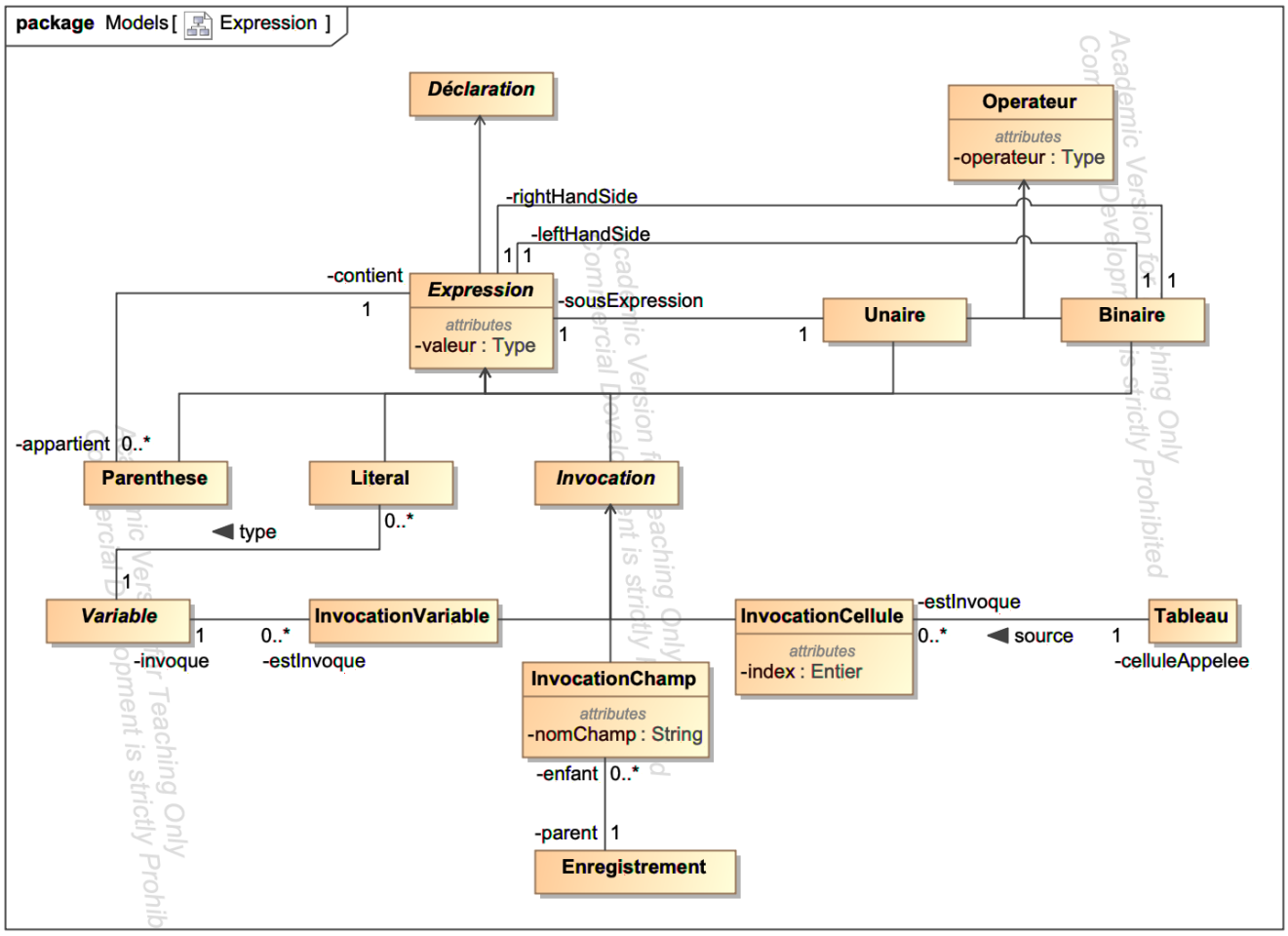

Figure 9: Diagramme de classe d'une expression

```
context Programme inv:
  self.declarations.size(declaration |
    declaration.oclIsTypeOf(Procedure) and
    declaration.name == "Cody") == 1
```

Listing 9: Unicité de la procédure Cody dans le programme

La contrainte permettant de vérifier qu'une variable globale porte un nom unique est défini comme suit (Listing 10) :

```
context Programme inv:
  self.declarations.forAll(d1, d2 |
    d1.oclIsTypeOf(Variable) and
    d2.oclIsTypeOf(Variable) and
    d1 <> d2 implies d1.name <> d2.name)
```

Listing 10: Unicité des noms des variables

La contrainte ci-après (Listing 11) permet l'unicité des noms des paramètres d'une procédure.

```
context Procedure inv:
  self.parametres.forAll(p1, p2 |
    p1 <> p2 implies p1.name <> p2.name)
```

Listing 11: Unicité des noms des paramètres d'une procédure

Une procédure va aussi devoir avoir une unicité pour les noms des variables de celle-ci. Cette unicité est indiqué via la contrainte OCL suivante (Listing 12)

```
context Procedure inv:
  self.declarations.forAll(d1, d2 |
    d1.oclIsTypeOf(Variable) and
    d2.oclIsTypeOf(Variable) and
    d1 <> d2 implies d1.name <> d2.name)
```

Listing 12: Unicité des noms des variables d'une procédure

Enfin, la contrainte OCL ci-dessous (Listing 13) permet l'unicité des noms des champs d'un Enregistrement

```
context Enregistrement inv:
  self.champs.forAll(c1, c2 |
    c1 <> c2 implies c1.name <> c2.name)
```

Listing 13: Unicité des noms des champs d'un enregistrement

2.7 OCL - Contrainte permettant de vérifier qu'une déclaration est bien formée

La contrainte de multiplicité (1..*) sur la liaison entre *Enregistrement* et *Primitif* (nommé *possède* - Figure 8) force à ce que la liste des champs d'un enregistrement ne soit pas vide.

Un tableau est toujours au minimum d'une dimension lors de l'initialisation d'une class *Tableau*. La contrainte OCL permettant d'avoir une longueur positive est ci-après (Listing 14).

```
context Tableau inv:
  self.longueur >= 0
```

Listing 14: Dimension strictement positive

2.8 OCL - Contrats sur l'opération *type(exp : Expression) : Type*

Dans les différents contrats, la fonction *oclType()* est utilisée. Il s'agit d'une fonction fourni par OCL qui permet d'évaluer le type de l'instance sur laquelle il est appelé (**?**).

Le contrat OCL présenté ci-dessous (Listing 15) spécifie que le type des littéraux est le type qui leur correspond.

```
context Expression::type(exp: Expression)
  post:
    if exp.oclIsTypeOf(Primitif) then
      return exp.oclType()
```

Listing 15: Le type des littéraux est le type qui leur correspond

Le contrat du listing 16 indique que le type d'une expression unaire est lié au type de l'opérateur, à condition que sa sous-expression y corresponde.

```
context Expression::type(exp: Expression)
  post:
    if exp.oclIsTypeOf(Unaire) then
      if exp.operateur.oclIsTypeOf(
        exp.sous-expression.oclType()
      ) then
        return exp.operateur.oclType()
```

Listing 16: Contrat OCL sur le type unaire

Le contrat OCL suivant (Listing 17) défini qu'un type d'une expression binaire est lié au type de son opérateur.

```
context Expression::type(exp: Expression)
  post:
    if exp.oclIsTypeOf(Binaire) then
      if exp.operateur.oclIsTypeOf(
        exp.leftHandSide.oclType()
      ) and exp.operateur.oclIsTypeOf(
        exp.rightHandSide.oclType()
      ) then
        return exp.operateur.oclType()
```

Listing 17: Contrat OCL sur le type binaire

Le contrat ci-dessous (Listing 18) spécifie que le type d'une expression paranthésée est le type de sa sous-expression.

```
context Expression::type(exp: Expression)
  post:
    if exp.oclIsTypeOf(Parenthese) then
      return exp.sous-expression.oclType()
```

Listing 18: Contrat OCL sur le type parenthèsée

Le contrat suivant (Listing 19) spécifie que le type d'une expression gauche correspondant à l'accès à un champ est le type de sa déclaration dans l'enregistrement.

```
context Expression::type(exp: Expression)
  post:
    if exp.oclIsTypeOf(Binaire) then
      if exp.rightHandSide.oclIsTypeOf(InvocationChamp) then
        return exp.rightHandSide.parent
          .getChamp(exp.rightHandSide.nomChamp)
          .oclType()
```

Listing 19: Contrat OCL sur le type d'un enregistrement

Le dernier contrat présenté dans cette section (Listing 20) spécifie que le type d'une expression gauche correspondant à l'accès à une case d'un tableau est le type de la déclaration du tableau.

```
context Expression::type(exp: Expression)
  post:
    if exp.oclIsTypeOf(Binaire) then
      if exp.rightHandSide.oclIsTypeOf(InvocationCellule) then
        return exp.rightHandSide.source.type.oclType()
```

Listing 20: Contrat OCL sur le type d'un tableau

2.9 OCL - Contrats sur l'opération *estValide() : Boolean*

Ce premier contrat OCL (Listing 21) spécifie qu'un appel de procédure doit référer à une déclaration de procédure dont le nom existe dans le programme.

```
context Declaration::estValide()
  post:
    if self.oclIsTypeOf(AppelProcedure) then
      return Procedure.allInstance()
        ->include(p1 | p1.name == self.appel.name)
```

Listing 21: Contrat sur l'existance du nom de la procédure appellée

Les gardes des instructions sont une propriété boolean de la classe *Composée* (Figure 8), il n'y a donc pas besoin de contrat OCL pour vérifier que les gardes soient toujours des boolean et que l'instruction soit valide.

Le contrat OCL ci-dessous (Listing 22) spécifie que tous les paramètres d'une instruction *Action* doivent être des entiers.

```
context Declaration::estValide()
  post:
    if self.oclIsTypeOf(ProcedureAction) then
      return self.parametres.forAll(
        p1 | p1.type.oclIsTypeOf(Entier)
      )
```

Listing 22: Les paramètres d'une instruction *Action* doivent être des entiers

Le contrat OCL présenté à la figure 23 spécifie que les paramètres effectifs de rang i doivent être de même type dans un appel de procédure que dans la déclaration du même nom.

```
context Declaration::estValide()
  if self.oclIsTypeOf(AppelProcedure) then
    return self.list.forAll(p1 |
      self.parametres.exists(p2 |
        p1.oclType() == p2.oclType() and
        p1.name == p2.name
      )
    )
```

Listing 23: Contrat OCL indiquant que les paramètres effectifs doivent être de même type que dans la déclaration

Le contrat suivant (Listing 24) indique que la partie gauche et la partie droite d'une affectation doivent être de même type.

```
context Declaration::estValide()
  post:
    if self.oclIsTypeOf(Affectation) then
      return Expression::type(self.leftHandSide) ==
          Expression::type(self.rightHandSide)
```

Listing 24: Contrat OCL spécifiant que la partie gauche et droite d'une affectation ont le même type

Enfin, ce dernier contrat OCL (Listing 25) spécifie que le type de retour d'une procédure doit toujours être void.

```
context Declaration::estValide()
  post:
    if self.oclIsTypeOf(Procedure) then
      return self.retour.oclIsTypeOf(Void)
```

Listing 25: Contrat OCL indiquant que le type de retour d'une procédure est toujours void

2.10 OCL - Contrats sur l'opération *prec_mouv() : Déplacement*

Le premier contract OCL (Listing 26) va permet de vérifier que lors d'un déplacement, si on tente d'accéder à une case ou se trouve un obstacle, le déplacement n'est pas effectué.

```
context ProcedureAction::right()
  def: player = NiveauEnCours::elements.select(e -> e.name == "Cody"
        and e.oclIsTypeOf(Player))
  post rightObstacle:
    if  NiveauEnCours::getElementAtPosition(player.Position.X + 1,
      player.Position.Y).oclIsTypeOf(Obstacle) then
      player.Position.X == player.Position.X@pre and
      player.Position.Y == player.Position.Y@pre
```

```
context ProcedureAction::left()
  def: player = NiveauEnCours::elements.select(e -> e.name == "Cody"
          and e.oclIsTypeOf(Player))
  post leftObstacle:
    if  NiveauEnCours::getElementAtPosition(player.Position.X - 1,
      player.Position.Y).oclIsTypeOf(Obstacle) then
      player.Position.X == player.Position.X@pre and
      player.Position.Y == player.Position.Y@pre

context ProcedureAction::up()
  def: player = NiveauEnCours::elements.select(e -> e.name == "Cody"
          and e.oclIsTypeOf(Player))
  post upObstacle:
    if  NiveauEnCours::getElementAtPosition(player.Position.X ,
      player.Position.Y - 1).oclIsTypeOf(Obstacle) then
      player.Position.X == player.Position.X@pre and
      pPlayer.Position.Y == player.Position.Y@pre

context ProcedureAction::down()
  def: player = NiveauEnCours::elements.select(e -> e.name == "Cody"
          and e.oclIsTypeOf(Player))
  post downObstacle:
    if  NiveauEnCours::getElementAtPosition(player.Position.X,
      player.Position.Y + 1).oclIsTypeOf(Obstacle) then
      player.Position.X == player.Position.X@pre and
      player.Position.Y == player.Position.Y@pre
```

Listing 26: On empêche le déplacement si la case est occupée par un obstacle

Le contrat OCL (Listing 27) suivant indique que si le joueur accède à une case ou ce trouve un tunnel, il ressort dans la case suivant le dernier mouvement à partir de l'autre tunnel.

```
context ProcedureAction::right()
  def: tunnel : Position
  def: player = NiveauEnCours::elements.select(e -> e.name == "Cody"
          and e.oclIsTypeOf(Player))
  post:
    tunnel = NiveauEnCours::getElementAtPosition(
    player.PositionX + 1, Player.PositionY)

    if tunnel.oclIsTypeOf(Tunnel) then
      player.Position.X == tunnel.Sortie.X + 1
      and player.Position.Y == tunnel.Sortie.Y

context ProcedureAction::left()
  def: tunnel : Position
  def: player = NiveauEnCours::elements.select(e -> e.name == "Cody"
          and e.oclIsTypeOf(Player))
  post:
```

```
    tunnel = NiveauEnCours::getElementAtPosition(
    player.PositionX - 1, Player.PositionY)

    if tunnel.oclIsTypeOf(Tunnel) then
      player.Position.X == tunnel.Sortie.X - 1
      and player.Position.Y == tunnel.Sortie.Y

context ProcedureAction::up()
  def: tunnel : Position
  def: player = NiveauEnCours::elements.select(e -> e.name == "Cody"
        and e.oclIsTypeOf(Player))
  post:
    tunnel = NiveauEnCours::getElementAtPosition(
    player.PositionX, Player.PositionY - 1)

    if tunnel.oclIsTypeOf(Tunnel) then
      player.Position.X == tunnel.Sortie.X and
      player.Position.Y == tunnel.Sortie.Y - 1

context ProcedureAction::down()
  def: tunnel : Position
  def: player = NiveauEnCours::elements.select(e -> e.name == "Cody"
        and e.oclIsTypeOf(Player))
  post:
    tunnel = NiveauEnCours::getElementAtPosition(
    player.PositionX, Player.PositionY + 1)

    if tunnel.oclIsTypeOf(Tunnel) then
      player.Position.X == tunnel.Sortie.X and
      player.Position.Y == tunnel.Sortie.Y + 1
```

Listing 27: Contract OCL pour le passage dans un tunnel

Enfin, le dernier contract OCL (Listing 28) permet d'indiquer que si le joueur saute, il atterit deux cases plus loin dans la même direction. Par contre, s'il y a un obstacle deux cases plus loin, le joueur ne bouge pas.

```
context ProcedureAction::jump()
  def: newPosition : Position
  def: player = Niveau::elements.select(e -> e.name == "Cody" and
        e.oclIsTypeOf(Player))
  post:
    if prec_mouv() == right then
      newPosition = NiveauEnCours::getElementAtPosition(
      player.PositionX + 2, player.PositionY)

      if newPosition.oclIsTypeOf(Surface) then
        player.Position.X == player.Position.X@pre + 2
        and player.Position.Y == player.Position.Y@pre
      else
```

```
      player.Position.X == player.Position.X@pre
      and player.Position.Y == player.Position.Y@pre
  if prec_mouv() == left then
    newPosition = NiveauEnCours::getElementAtPosition(
    player.PositionX - 2, player.PositionY)

    if newPosition.oclIsTypeOf(Surface) then
      player.Position.X == player.Position.X@pre - 2
      and player.Position.Y == player.Position.Y@pre
    else
      player.Position.X == player.Position.X@pre
      and player.Position.Y == player.Position.Y@pre
  if prec_mouv() == up then
    newPosition = NiveauEnCours::getElementAtPosition(
    player.PositionX, player.PositionY - 2)

    if newPosition.oclIsTypeOf(Surface) then
      player.Position.X == player.Position.X@pre
      and player.Position.Y == player.Position.Y@pre
    else
      player.Position.X == player.Position.X@pre
      and player.Position.Y == player.Position.Y@pre - 2
  if prec_mouv() == down then
    newPosition = NiveauEnCours::getElementAtPosition(
    player.PositionX, player.PositionY + 2)

    if newPosition.oclIsTypeOf(Surface) then
      player.Position.X == player.Position.X@pre
      and player.Position.Y == player.Position.Y@pre + 2
    else
      player.Position.X == player.Position.X@pre
      and player.Position.Y == player.Position.Y@pre
```

Listing 28: Contract OCL psur le saut

2.11 Diagramme d'objet UML pour le «niveau du sapin»

Le code de résolution du niveau présenté à la figure 10, une fois l'arbre retiré, est le suivant (Listing 29) :

```
up(2)
right(2)
/* Permet de changer de direction
   vers le bas sans se deplacer */
directionDown()
fight()
down(2)
dig()
```

Listing 29: Le code permettant de résoudre le niveau

Figure 10: Le niveau à résoudre

Dans la résolution, nous avons utilisé une instruction d'action *directionDown* qui permet de modifier la direction dans laquelle regarde le joueur sans changer de position. Cela permet de se mettre dans la direction pour frapper l'ennemi.

A partir du code de la solution et de notre diagramme UML, nous pouvons représenter le diagramme d'objet avec la structure présentée ci-après (Figure 11).

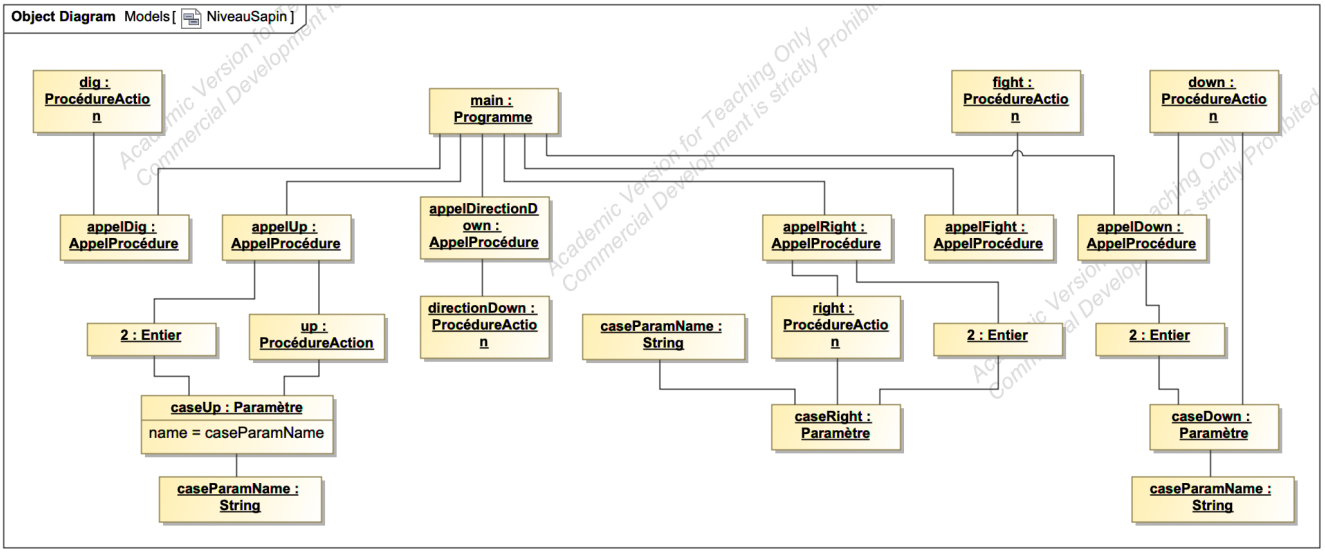

Figure 11: Diagramme d'objet du programme du niveau du sapin