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 joueurs, avatars, matchs, rencontres et les mondes.

Le jeu rassemble des joueurs, qui sauvegardent des avatars. Il possède des matchs qui sont constitués de rencontres qui on lieu dans des mondes. Les mondes sont possédés par le jeu.

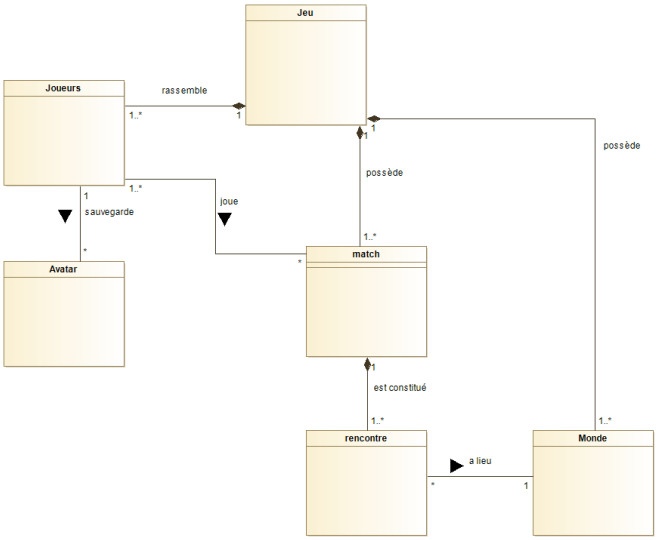

Figure 1: Diagramme de classe des éléments principaux du jeu

1.2 Version enrichie du diagramme de classe du jeu

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

La classe *Position*, qui possède deux attributs (x et y), va conserver la position de chaque objet présent pendant le Jeu.

La classe *Personnage*, qui représente les personnages (avatar ou zombie) avec leurs points de vie, ratio d'attaque, leurs effets et leur position. Cette classe à deux sous-classes : *Zombie* et *Avatar*.

Avatar comporte maintenant des aspects normaux et payants, une portée de visibilité, un ratio de défense et un nom. Cette classe est liée à *Détail* qui va garder le détails des matchs et des opposants rencontrés lors de ceux-ci. Elle est également liée à *Stratégie* qui va contenir la stratégie à appliquer.

La classe *Avatar* est aussi liée à *Rencontre* vu que l'avatar est utilisé lors d'une rencontre. Il possède un *Inventaire*

Joueurs est maintenant une classe abstraite contenant un pseudo et s'il à payé pour des aspects supplémentaires. Ses deux sous-classes sont *Physique* qui représente un joueur physique et *Machine* qui représente une machine avec un niveau de difficulté.

L'*Avatar* possède également un *Inventaire* qui est composé d'une opération permettant d'ajouter un item à celui-ci. Cet *Inventaire* possède des *Objet*. Ces Objets peuvent être de 4 sous-classes. La première sous-classe *Ramassable* qui peut-être *Nourriture*, *Boisson* et *Munition*. La seconde sous-classe *Orientation*, qui peut être *Carte* ou *Radar*. La troisième sous-classe *Aide* Qui peut être un *Activable* (une *Cape* ou une *Cotte de maille*), Ou des *Bottes pare-feu*, des *Bottes à crampons* ou un *Kit de plongée*. La quatrième sous-classe est *Bonus*, qui peut être un *BonusAttaque* ou *BonusDefense*. Au niveau de la classe *Monde*, elle possède maintenant des *Cases*, qui sont toutes à gauche ou à droite et en haut ou en bas d'un autre. Chaque *Cases* peut être vide ou alors un *Obstacle*, un *Item* ou le *Graal*. Si elle est un *Obstacle*, celui-ci peut être une *Zone* un *Franchissable* ou un *Infranchissable*.

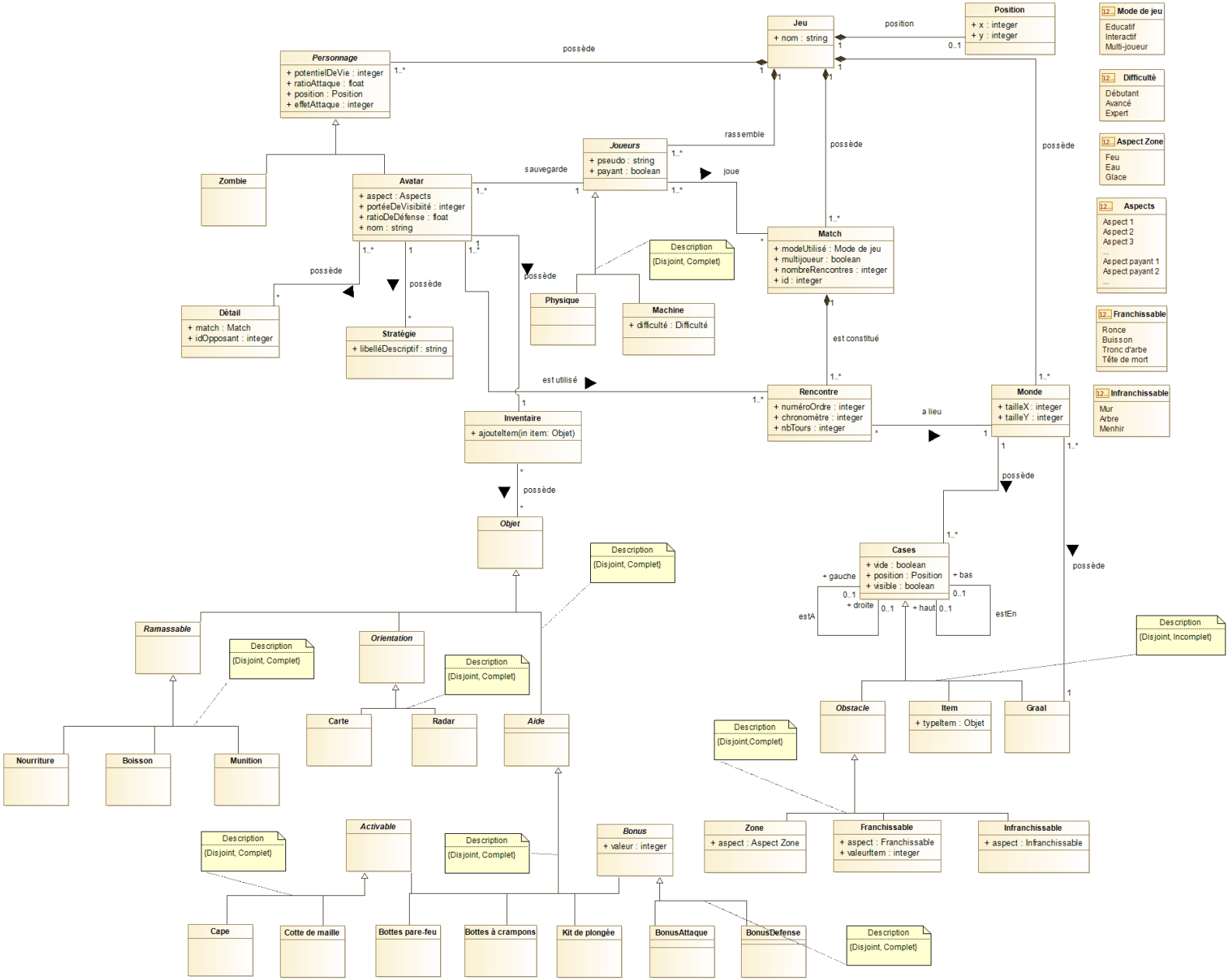

Figure 2: Diagramme enrichit

1.3 Ocl - Contraintes d'unicité

Le fait que les matchs sont identifiés de manière unique se caractérise par cette contrainte Ocl:

```
context Jeu inv matchUnique :
    self.Match.allInstances -> forall( m1,m2 |
        m1.id <> m2.id implies m1 <> m2)
```

Listing 1: Contrainte sur l'unicité d'un match

La contrainte que des joueurs ont des pseudos différents au sein du jeu se caractérise par cette contrainte :

```
context Jeu inv pseudoJoueur :
    self.Joueurs.allInstances -> forall( j1,j2 |
        j1.pseudo <> j2.pseudo implies j1 <> j2)
```

Listing 2: Contrainte sur les pseudos

La contrainte Ocl disant que les personnages/avatars d'un joueur sont nommés différemment est exprimée comme suit:

```
context Joueurs inv nomAvatars :
    self.Avatar.allInstances -> forall( a1,a2 |
        a1.nom <> a2.nom and a1.aspect <> a2.aspect
            implies a1 <> a2)
```

Listing 3: Contrainte sur le nom

La contrainte Ocl obligeant les rencontres à avoir un numéro d'ordre unique est la suivante :

```
context Match inv ordre :
    self.rencontres.allInstances -> forall( r1,r2 |
        r1.numeroOrdre <> r2.numeroOrdre implies r1 <> r2)
```

Listing 4: Contrainte sur le numéro d'ordre unique

1.4 Ocl - Contraintes OCL du diagramme de classe

Le fait que le potentiel de vie d'un joueur est toujours positif est caractérisé par la contrainte suivante:

```
context Personnage inv vie :
  self.potentielDeVie > 0
```

Listing 5: Contrainte sur le potentiel de vie

La contrainte exprimant le fait que les ratios d'attaque et de défense sont des ratios s'exprime comme suit:

```
context Personnage inv ratios :
  self.ratioAttaque > 0 and self.ratioAttaque < 1
    and
      if self.oclIsTypeOf(Avatar)
        then self.ratioDefense > 0 and self.ratioDefense < 1
```

```
endif
```

Listing 6: Contrainte sur les ratios

La contrainte sur la non-parité des rencontres est caractérisée comme suit:

```
context Match inv nbRencontre :
  self.rencontre.size() % 2 = 1
```

Listing 7: Contrainte sur la non-parité des rencontres

La contrainte exprimant que le numéro identifiant la rencontre correspond à son ordre de jeu est la suivante:

```
context Match inv ordone :
  self.rencontre.allInstances -> asOrderedSet()
```

Listing 8: Contrainte sur l'ordre des rencontres

La contrainte sur le les 3 types d'items contenu par l'inventaire est représentée dans le diagramme de classe par l'héritage d'*Objet* (figure 2).

La contrainte qu'un monde ne possède qu'un *Graal* est aussi représentée dans le diagramme de classe par l'association possède entre *Monde* et *Graal* (figure2).

La contrainte du fait que les cases ne peuvent excéder la longueur d'un monde est la suivante:

```
context Monde inv posCases :
  self.case.allInstances -> forall ( c |
    c.position.x >= 0 and c.position.x < self.tailleX and
    c.position.y >= 0 and c.position.y < self.tailleY)
```

Listing 9: Contrainte sur la position des cases

La contrainte sur la disposition des cases est la suivante :

Le fait que le joueur se trouve dans la case correspondant à sa position absolue est caractérisé par la contrainte suivante :

```
context Avatar inv posAbsolue :
  self.rencontre.monde.case.allInstances -> forall ( c1,c2 |
    self.position.x = c1.position.x and
    self.position.y = c1.position.y implies
    self.position.x <> c2.position.x and
    self.position.y <> c2.position.y)
```

Listing 10: Contrainte sur la position du joueur

Le fait que la bordure d'un plateau contienne toujours des éléments infranchissables est caractérisé par la contrainte suivante :

```
context Monde inv Infranchissable :
  self.case.allInstances -> forall(c |
    (c.position.x = 0 and c.position.y <= 0 and
    c.position.y > self.tailleY or
    c.position.x = self.tailleX - 1 and
    c.position.y <= 0 and c.position.y > self.tailleY or
    c.position.y = 0 and c.position.x <= 0 and
    c.position.x > self.tailleX or
```

```
    c.position.y = self.tailleY - 1 and c.position.x <= 0 and
     c.position.x > self.tailleX) implies
      self.case.oclIsTypeOf(Infranchissable)
```

Listing 11: Contrainte sur la bordure du plateau

La contrainte sur la visibilité est la suivante :

Le fait qu'un personnage ne puisse se trouver sur une case portant un obstacle infranchissable est caractérisé par la contrainte suivante :

```
context Monde inv posJoueurCase :
  if self.case.allInstances -> oclIsTypeOf(Infranchissable) and
    self.case.position.x = self.rencontre.avatar.position.x and
    self.case.position.y = self.rencontre.avatar.position.y
  then
    false
  else
    true
  endif
```

Listing 12: Contrainte sur la position du joueur sur case infranchissable

Le fait que 2 personnages (zombie compris) ne puissent se trouver sur la même case est caractérisé par la contrainte suivante :

```
context Personnage inv memeCase :
  self.allInstances -> forall ( p1,p2 |
    p1.position.x = p2.position.x and
    p1.position.y = p2.position.y implies
    p1 = p2)
```

2 Description de Stratégie

2.1 Modélisation du concept de stratégie

Voici un diagramme de classe qui fixe les éléments principaux d'une stratégie :

Une *Stratégie* est composé d'une *vision court terme* et une *vision long terme* qui sont toutes les deux articulées par des objectifs eux même réalisés par des règles.

Une *Stratégie* comporte également des *Déclaration* pouvant être des modules ou des variables.

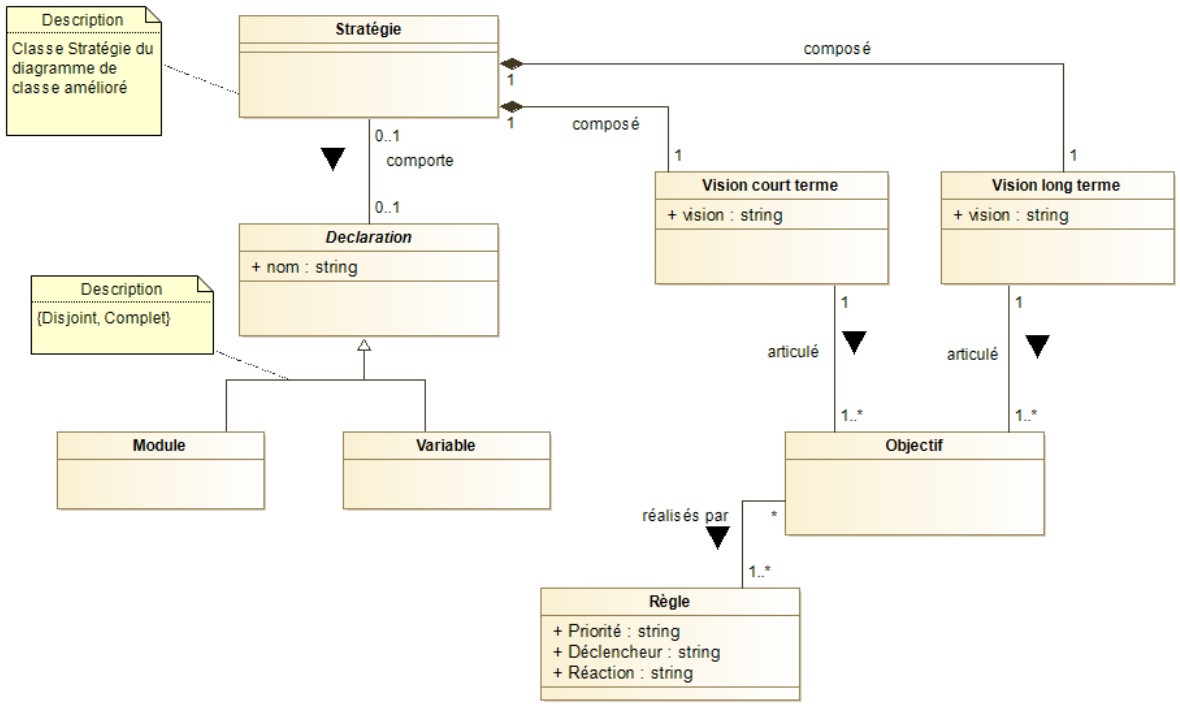

Figure 3: Diagramme de classe d'une stratégie

2.2 Modélisation du concept de Type

Le concept de *Type* est modélisé de la manière présentée à la figure 4.La classe *Type* est parente de plusieurs classes présentes dans ce diagramme *Enumération*, *TypePrimitif*, *Tableau*.

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

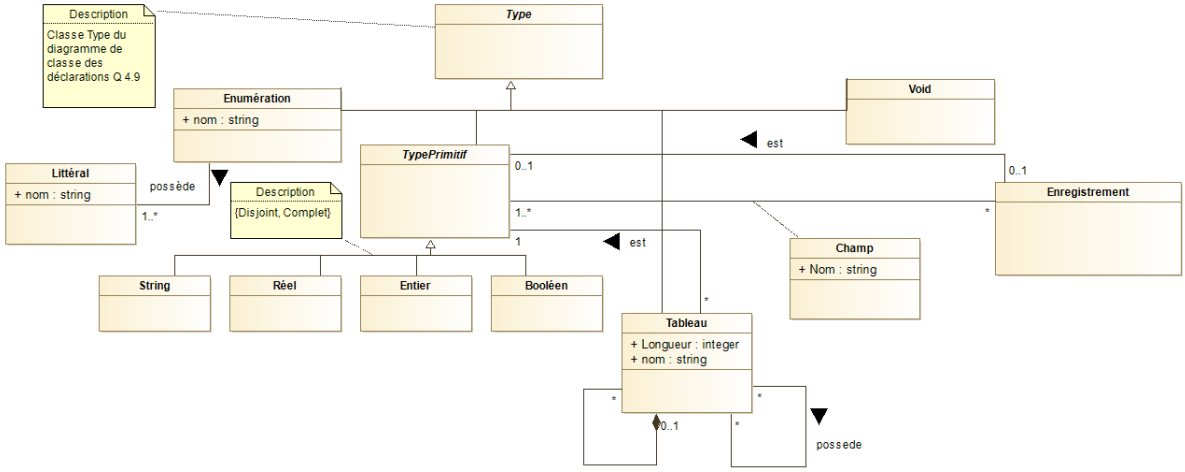

Figure 4: Diagramme de classe d'un type

2.3 Modélisation du concept de déclaration

Présentée à la figure 5, la modélisation d'une *Declaration* montre que celle-ci est comportée dans une *Stratégie* et à 5 enfants : *Variable*, *Instruction*, *Expression*, *Module* et *Paramètres*.

Un *Module* comporte des variables et ces deux-ci ont un *Type*. Le *Module* prend également des *Paramètres*.

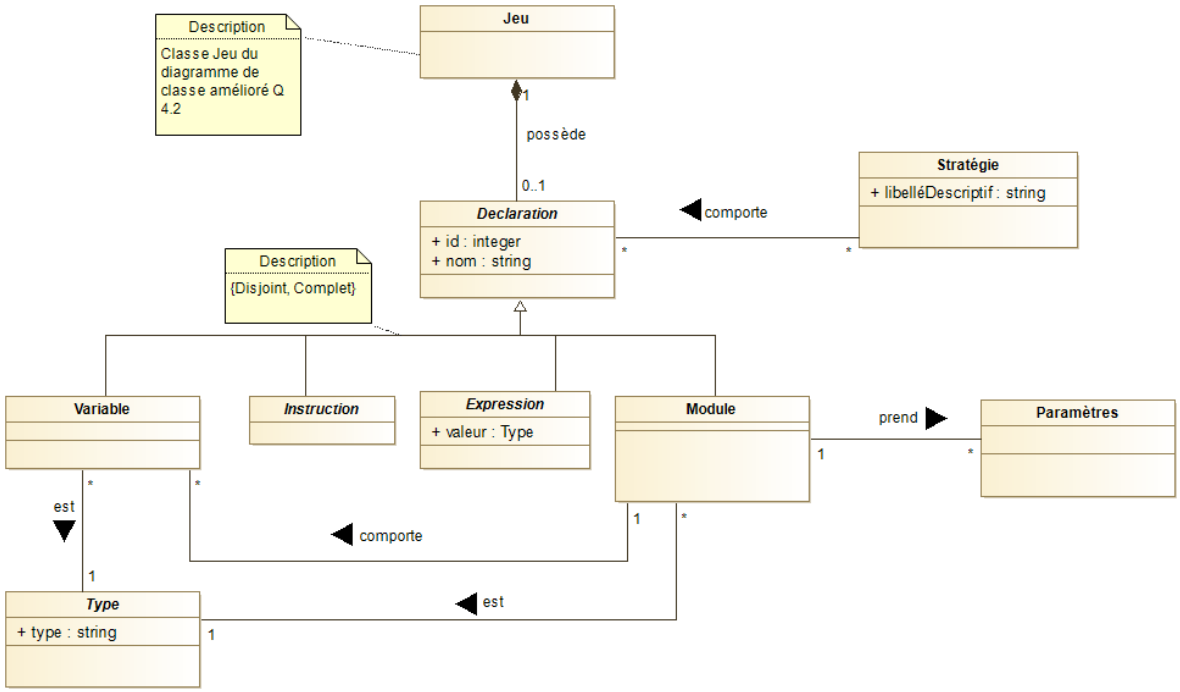

Figure 5: Diagramme de classe d'une déclaration

2.4 Modélisation explicite d'un objectif

Le concept d'objectif est présenté à l'aide de la figure 6. Un *Objectif* est parent de 4 enfants : *Neant* qui est l'objectif par défault. *Combinable* qui est lui même parent de *AllerVers*, *Contourner* et *Eviter*. *CollecterMax* et *Combattre* sont les deux derniers enfants. Les *Objectif* sont réalisés par des règles qui possèdent des *Reaction*.

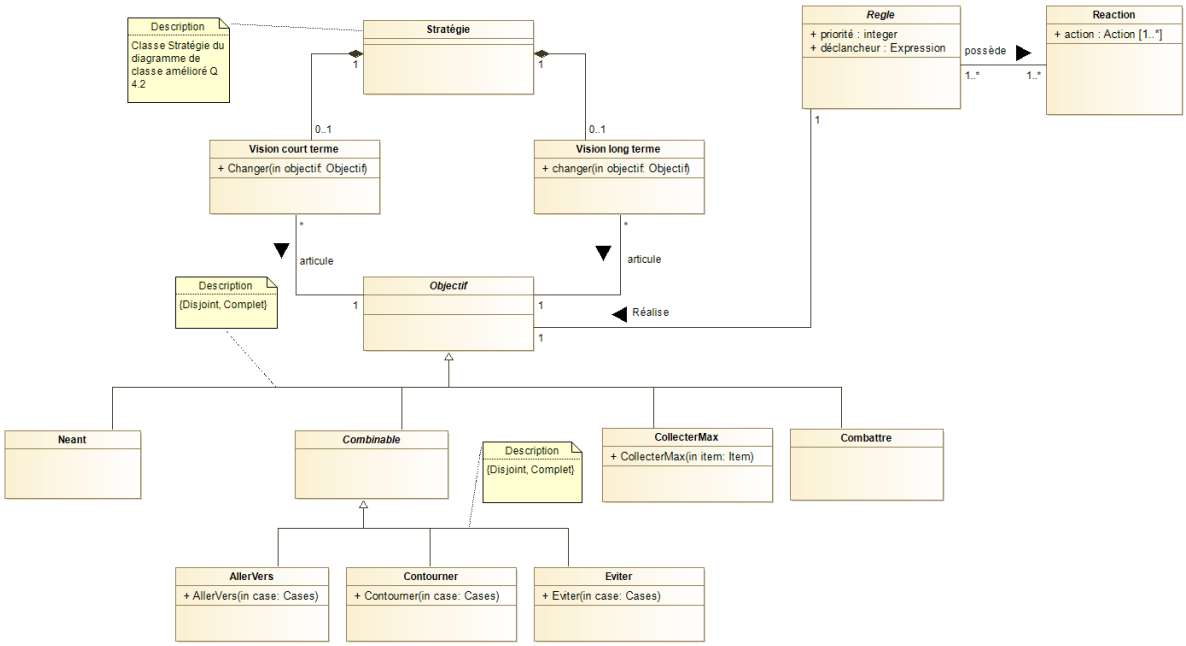

Figure 6: Diagramme de classe d'un objectif

2.5 Modélisation explicite d'une action

Le concept d'action est présenté à l'aide de la figure 7. Une *Action* est parent de 6 enfants : *SeDeplacer*, *UtiliserItem*, *Revetir*, *Frapper*, *Tirer* et *ConsulterRadar*.

C'est un personnage qui dispose de ces actions.

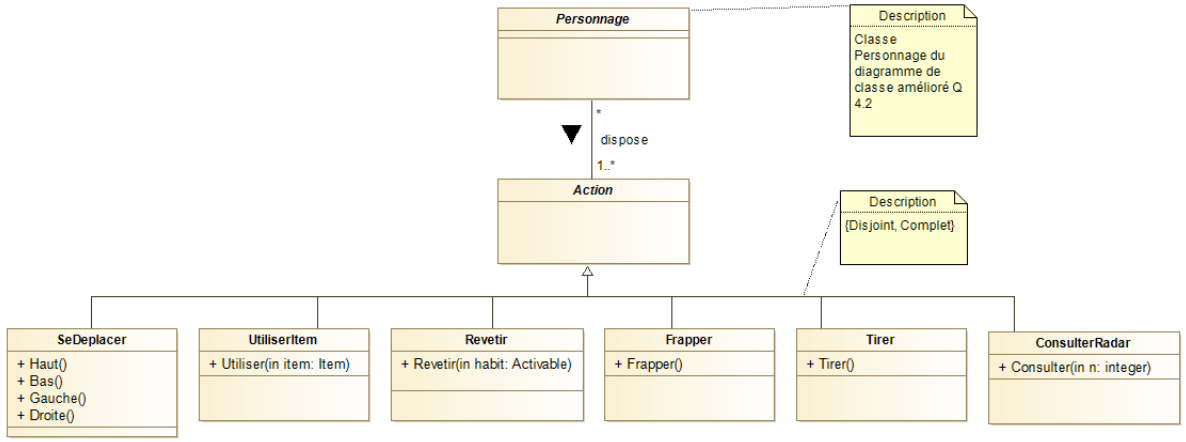

Figure 7: Diagramme de classe d'une action

2.6 Modélisation explicite d'une expression

Le concept d'expression est présenté à l'aide de la figure 8. Une *Expression* contient une *Parenthese* et peut prendre la forme de celle-ci. Une Expression peut aussi être un *Literal* qui est une variable, une *ExpressionGauche* qui elle même peut être un *AppelVariable* qui invoque une variable, un *AppelChamp* qui provient d'un *Enregistrement*, un *AppelCellule* qui a comme source un *Tableau*. Une *Expression* peut aussi être un *AppelModule* comportant des *Parametre*, une *Expression Unaire* ou *Binaire* comportant des *Operateur*.

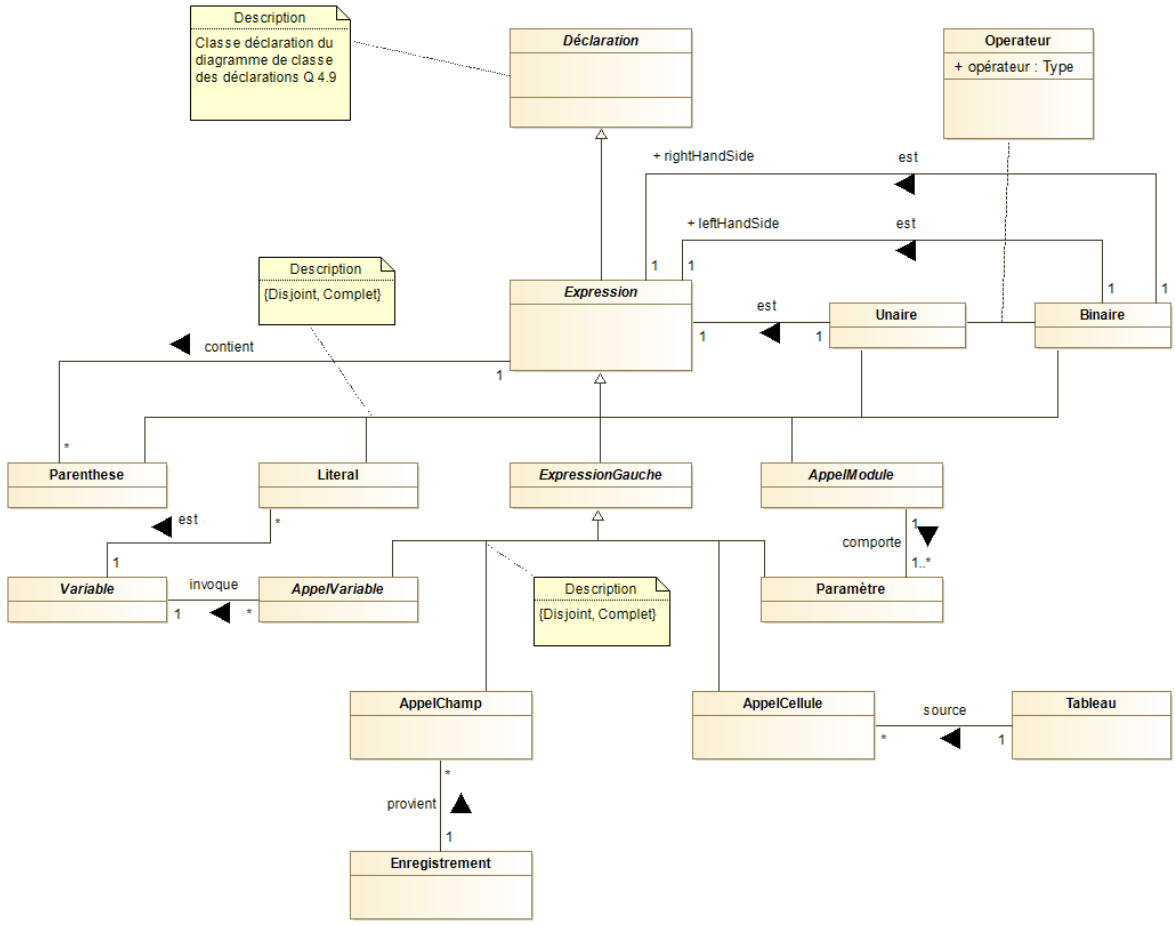

Figure 8: Diagramme de classe d'une expression

2.7 Modélisation explicite d'une instruction

Le concept d'instruction est présenté à l'aide de la figure 9. Une *Instruction* peut être *Skip* qui est une instruction pour passer son tour. Elle à 4 autres enfants qui sont *Conditionnelle*, *Iteration* (qui est composé de plusieurs instructions), *Affectation* et *Action* qui est lié aux actions suivantes.

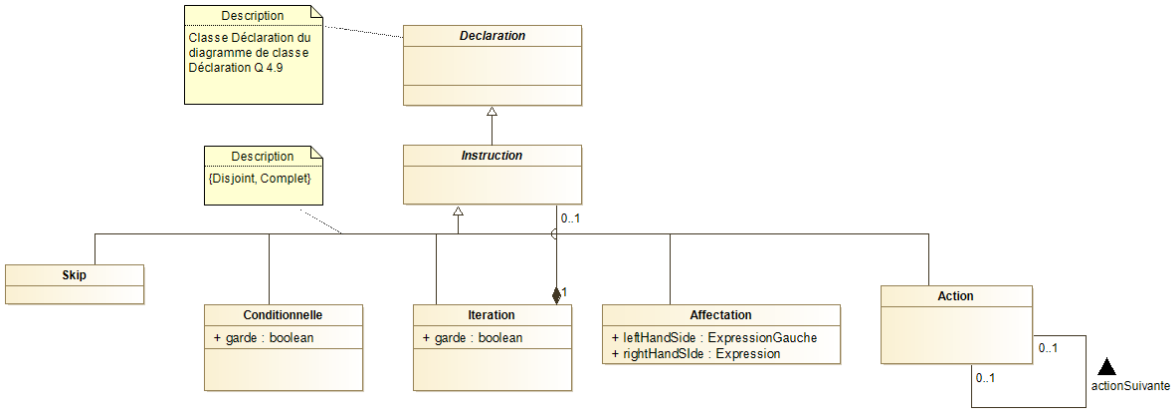

Figure 9: Diagramme de classe d'une instruction

2.8 OCL - Contraintes d'unicité

La première contrainte d'unicité sur le nom unique des modules est spécifiée comme suit :

```
context Strategie inv nomUnique :
  if self.Declaration.allInstances -> oclIsTypeOf(Module) and
    self.Declaration.forall( m1, m2 | m1.nom <> m2.nom)
  then
    true
  else
    false
  endif
```

Listing 13: Nom unique au sein d'une stratégie

La contrainte sur le nom des variables globales est la suivante :

```
context Strategie inv nomVarGlobal :
  if self.Declaration.allInstances ->oclIsTypeOf(Variable) and
    self.Declaration.forall( v1,v2 | v1.nom <> v2.nom)
  then
    true
  else
    false
  endif
```

Listing 14: Nom unique d'une variable globale

La contrainte sur les variables locales est la suivante :

```
context Module inv uniqueLocale :
  self.Variable.allInstances -> forall(v1,v2 |
  v1.nom <> v2.nom implies v1 <> v2)
```

Listing 15: Nom unique des variables locales

La contrainte sur les noms des paramètres d'un module est la suivante :

```
context Module inv uniqueParam :
  self.Parametre.allInstances -> forall(p1,p2 |
  p1.nom <> p2.nom implies p1 <> p2)
```

Listing 16: Nom unique des paramètres

La contrainte sur le nom des types soit globalement unique est la suivante :

```
context Type inv difNom :
  self.allInstances -> forall( e,t |
  e.oclIsTypeOf(Enumeration) and t.oclIsTypeOf(Tableau)
  and e.nom <> t.nom)
```

Listing 17: Nom unique des types

2.9 OCL - Contrainte sur les déclarations

La première contrainte sur l'unicité des littéraux est la suivante :

```
context enumeration inv literauxUnique :
  self.Literal.allInstances -> forall( l1,l2 |
    l1.nom <> l2.nom implies l1 <> l2)
```

Listing 18: unicité des litéraux

La contrainte sur la liste des champs est la suivante :

```
context Enregistrement inv champNonVide :
  self.Champ.allInstances -> size() > 0
```

Listing 19: champ non vide

La contrainte sur l'unicité du nom des champs est la suivante :

```
context Enregistrement inv nomChamp :
  self.Champ.allInstances -> forall( c1,c2 |
  c1.nom <> c2.nom implies c1 <> c2)
```

Listing 20: Nom unique des champs

La contrainte sur la dimension d'un tableau est la suivante :

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

Listing 21: dimension d'un tableau

La contrainte sur la dimension positive des tableaux est la suivante :

```
context Tableau inv dimPositive :
  self.allInstances -> forall(t | t.possede.longueur > 0)
```

Listing 22: Dimension positive

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

Ce premier contrat porte sur le fait que le type du littéral retourné corresponde au littéral :

```
context Expression :: type() : Type
  post:
    if Type.oclIsTypeOf(TypePrimitif) then
      return Type.oclType()
```

Listing 23: Contrat OCL sur le type des litéraux

Le contrat sur l'opérateur unaire est la suivante :

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

Listing 24: Contrat OCL sur le type unaire

Le contrat correspondant à l'opérateur binaire est le suivant :

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

Listing 25: Contrat OCL sur le type binaire

Le contrat sur le type de la sous-expression dans une parenthèse est le suivant :

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

Listing 26: Contrat OCL sur le type paranthésée

Le contrat sur le type de la déclaration est le suivant :
Le contrat sur le type de renvoi d'une énumération de littéraux est le suivant :

```
context Expression :: type() : Type
  post :
    if Type.oclIsTypeOf(Litteral)
    then
      return
        Type.est.oslType()
```

Listing 27: Contrat OCL sur le type de renvoi d'une énum de litéraux

Le contrat sur le type de retour d'un champ est le suivant :

```
context Expression :: type() : Type
  post :
    if Type.oclIsTypeOf(Champ)
    then
      return
      Type.TypePrimitif.oclType()
```

Listing 28: Contrat OCL sur le type de retour d'un champ

Le contrat sur le type du contenu d'un tableau est le suivant :

```
context Expression :: type() : Type
  post :
    if Type.oclIsTypeOf(Tableau)
    then
     return
      Type.est.oclType()
```

Listing 29: Contrat OCL sur le type de retour d'un tableau

2.11 Opération estTraversableGraceAuxItems() : Boolean

La classe de notre diagramme qui pourrait contenir cette opération est Case.

Le contrat de cette définition est le suivant :

```
context estTraversableGraceAuxItems(Case, Avatar) : Boolean :
    post :
        if Case.oclIsTypeOf(Zone) and (
            Case.aspect = Feu and
            Avatar.Inventaire.Objet.allInstances ->
                forall(o | o.oclIsTypeOf(Bottes Pare-Feu))
            or
            Case.aspect = Eau and
            Avatar.Inventaire.Objet.allInstances ->
                forall(o | o.oclIsTypeOf(Kit De Plongee))
            or
            Case.aspect = Glace and
            Avatar.Inventaire.Objet.allInstances ->
                forall(o | o.oclIsTypeOf(Bottes a crampon))
        )
        then
            return true
        endif
```

Listing 30: Contrat OCL sur l'opération estTraversableGraceAuxItems

2.12 Opération ramasser() : Void

L'opération est définie sur Avatar et ses paramètres sont Case et Avatar

Le contrat d'une telle opération est le suivant :

```
context ramasser(Case, Avatar) : Void
    post:
        if Case.typeItem.oclIsTypeOf(bonusAttaque)
            then
                Avatar.ratioAttaque x Case.typeItem.valeur
        endif

        if Case.typeItem.oclIsTypeOf(bonusDefense)
            then
                Avatar.ratioDefense x Case.typeItem.valeur
        endif

        if Case.typeItem.oclIsTypeOf(Radar)
            then
                Avatar.porteeDeVisibilite x 2
        endif

        if Case.typeItem.oclIsTypeOf(Ramassable) or
            Case.typeItem.oclIsTypeOf(Aide)
```

```
        then
            Avatar.Inventaire.ajouterItem(case.typeItem)
    endif
```

Listing 31: Contrat OCL sur l'opération ramasser

2.13 Opération Instruction :: estValide : Boolean

Le contrat d'une telle opération est le suivant :
- L'instruction "skip" est toujours valide :

```
context Instruction :: estValide() : Boolean
    post :
        if Instruction.oclIsTypeOf(Skip)
        then
            true
        endif
```

Listing 32: Contrat OCL sur l'opération estValide

- La garde d'une *Conditionelle* ou d'une *Iteration* est booleéne :

```
context Instruction :: estValide() : Boolean
    post :
        if Instruction.oclIsTypeOf(Conditionelle) or
            Instruction.oclIsTypeOf(Iteration)
        then
            Instruction.garde.oclIsTypeOf(Boolean)
        endif
```

Listing 33: Contrat OCL sur la garde

- La partie gauche et droite d'une *Affectation* sont du même type :

```
context Instruction :: estValide() : Boolean
    post :
        if Instruction.oclIsTypeOf(Affectation)
        then
            Instruction.leftHandSide.oclIsTypeOf(rightHandSide)
        endif
```

Listing 34: Contrat OCL les parties d'une affectation

2.14 Contrainte sur la cohérence des règles

Les contraintes sur la cohérence des règles sont les suivantes :
- Toutes les règles d'un même objectif ont des priorités différentes :

```
context Objectif inv prioRegle :
    self.Regle.allInstances -> forall( r1,r2 |
        r1 <> r2 implies r1.priorite <> r2.priorite)
```

Listing 35: Contrainte sur les priorités

- Une réaction de règle ne contient qu'une seule action SeDéplacer :

```
context Regle inv seDeplacer :
    self.Reaction.action.forall (a |
    a.oclIsTypeOf(SeDeplacer).size() <= 1)
```

Listing 36: Contrainte sur SeDéplacer

- La contrainte des métarègles est présente dans le diagramme de classe des *Objectif*.

2.15 Diagramme d'objet d'une stratégie

Voici, le diagramme d'objet d'une stratégie dont la vision à long terme est de se rendre vers le graal et à court terme de ramasser un maximum de nourriture.

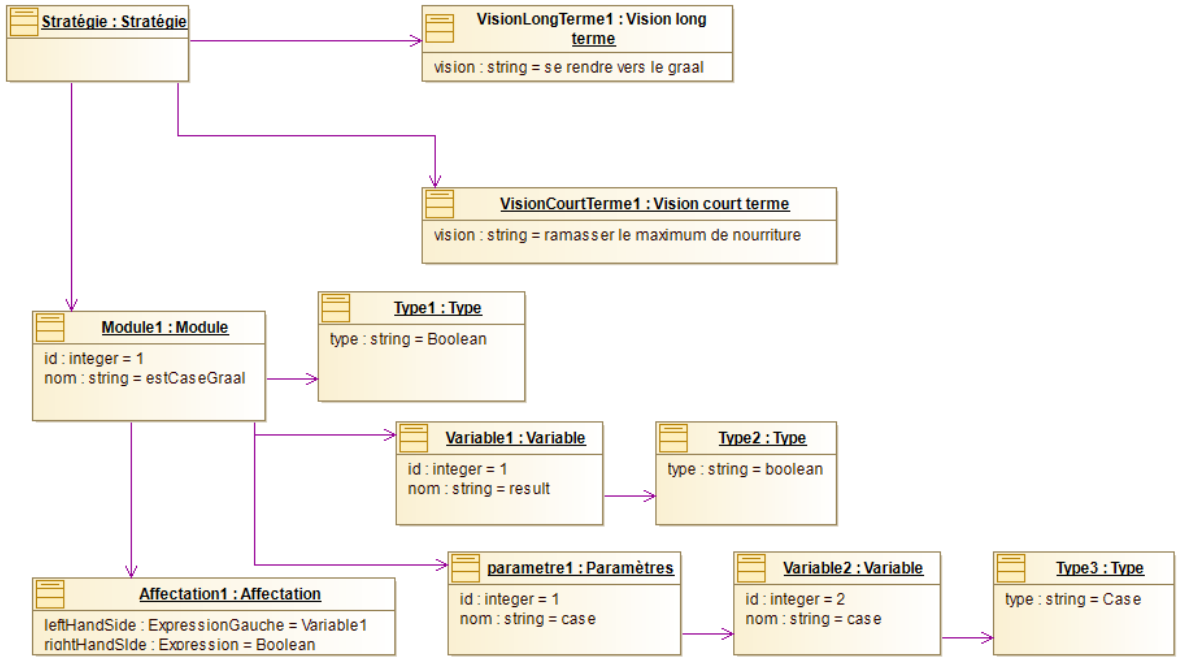

Figure 10: Diagramme d'objet d'une stratégie

2.16 Maturité et Modularité

La modification du diagramme de classe pour qu'un item se trouve sur un *zone* particulière est très simple, il suffit de supprimer l'héritage venant de obstacle vers *zone* et faire hériter *zone* de *Case*.

Cela n'aura pas d'impact sur la modélisation du langage mais certaines contraintes *OCL* devraient être adaptées étant donné le changement d'héritage.