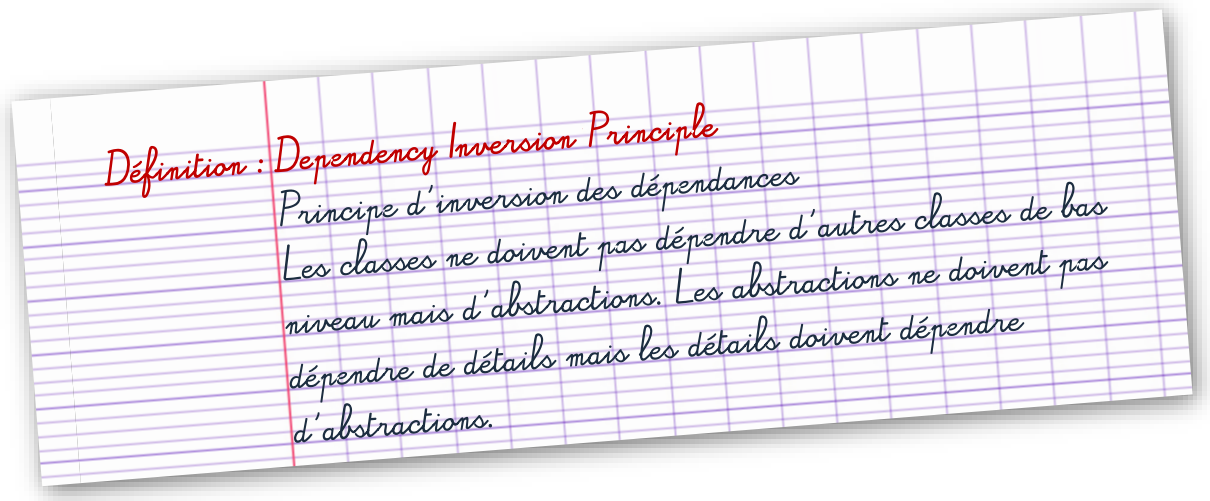
TD 12 – Premiers Principes de Conceptions Orientée Objet

Cours 1 Dependency Inversion Principle

1.1 Définition



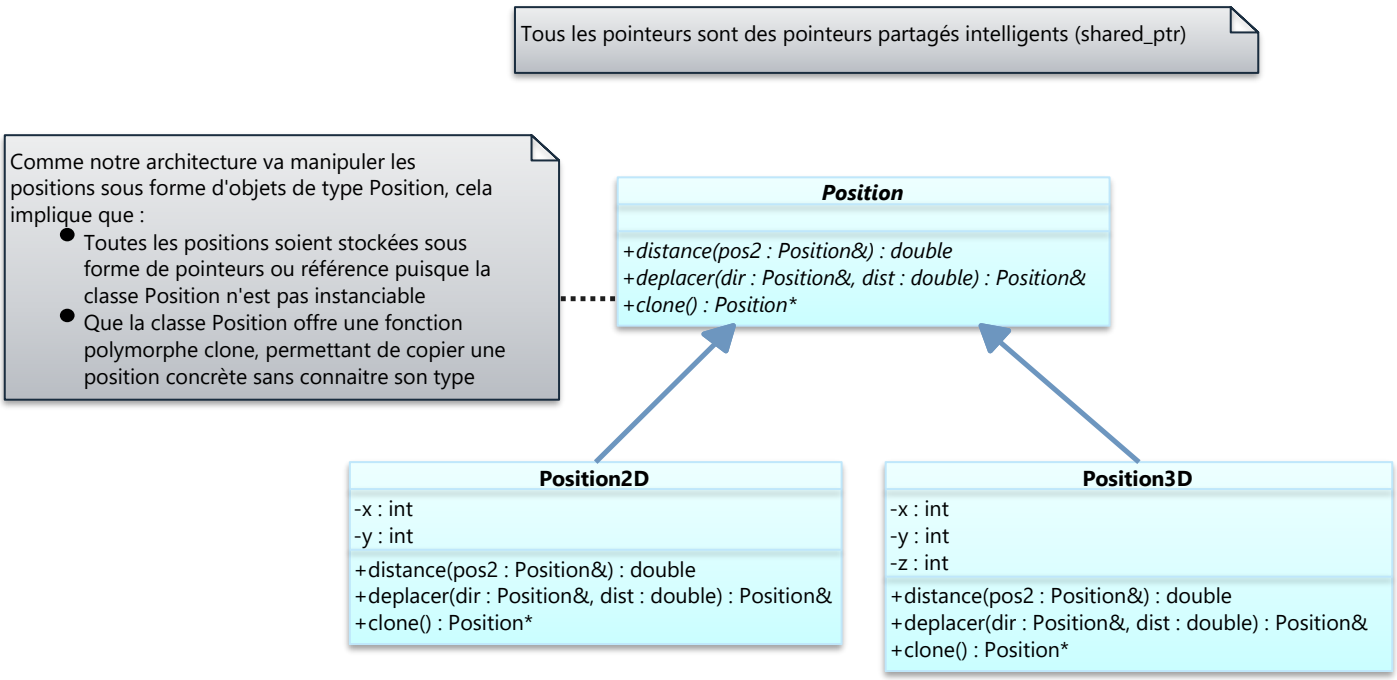
- En d'autres termes, dépendez de classes abstraites, ne dépendez pas de classes concrètes.
  - Faire cela obligera à dériver votre classe concrète de la classe abstraite. Ainsi, si dans le futur, il faut ajouter une nouvelle sorte de classe concrète, il n'y aura qu'à dériver une autre classe de la classe abstraite. Cela se fera donc sans modification des classes dépendantes de l'abstraction.

Exercice 2 BAKaway en ouvert-fermé en inversant les dépendances

- Appliquez ce principe à la conception de ***BAKaway***.
  - Rappelez-vous qu'aucun code n'a encore été écrit, vous pouvez tout reprendre sans vous soucier des modifications de code.

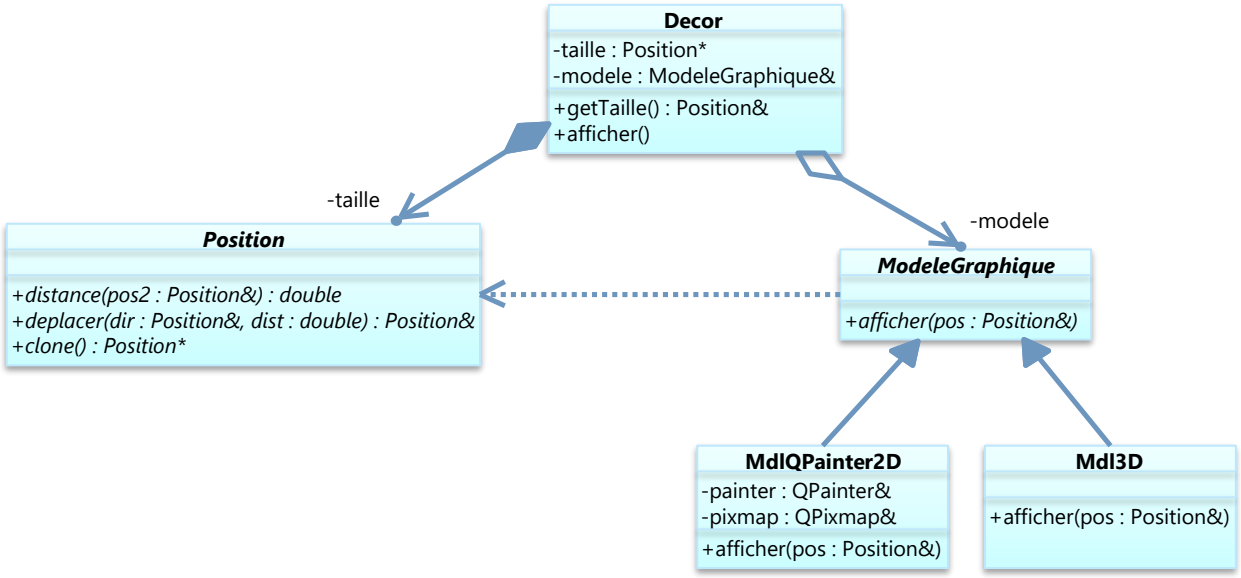
2.1 Solution proposée

Pour résoudre l'ouverture vers la 2D et la 3D, je propose de faire une classe abstraite `Position` dont va dériver une position 2D et une position 3D.



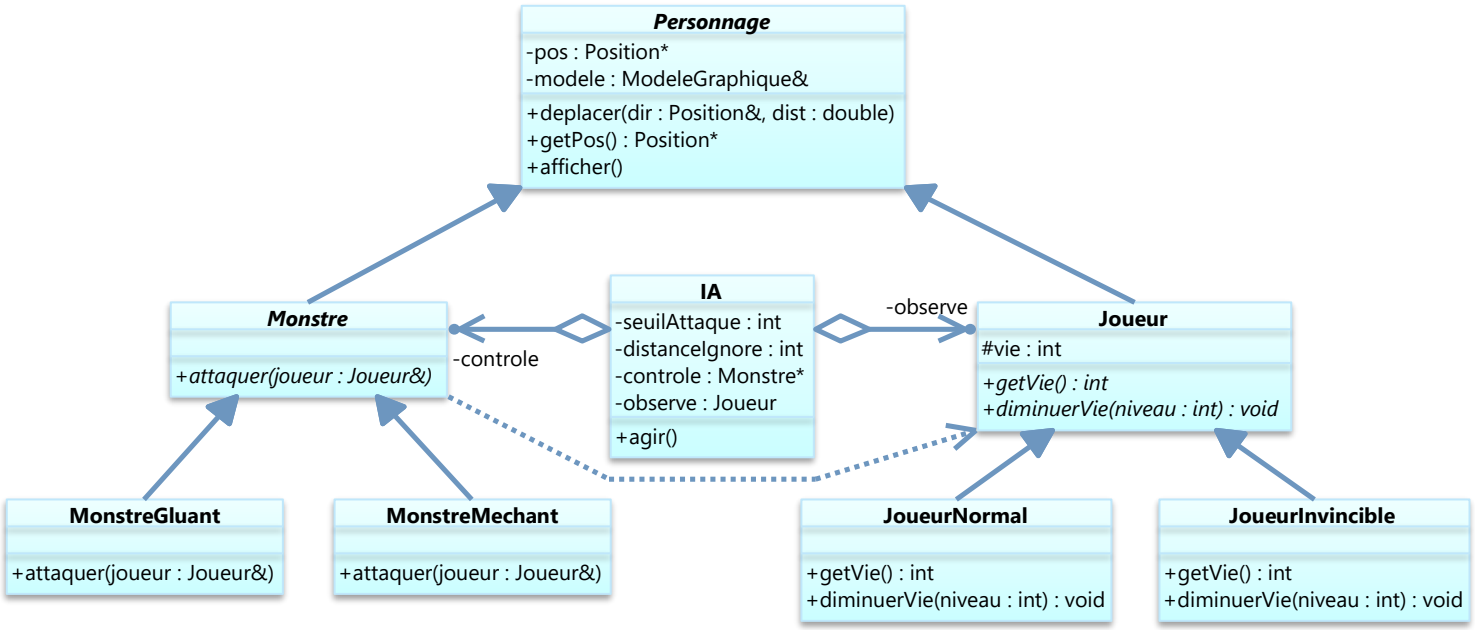
La méthode clone est là pour permettre la copie polymorphe des positions, ce qui sera très probablement utile. (Déjà vu lors d'un TD il me semble, il y a fort longtemps) Le corps d'une telle fonction est simplement **`return new Position2D(*this);`** pour la classe Position2D.

Toujours dans le contexte de la 2D / 3D, le moteur graphique sera très probablement différent. Par exemple utilisation d'un QPixmap et d'un QPainter pour la 2D et utilisation d'OpenGL pour la 3D. Cela nécessite de découpler l'affichage des classes concrètes. Pour réaliser cela, chaque élément graphique (le décor, les monstres, les joueurs) incorporera non seulement leur position, mais aussi leur modèle graphique. Le modèle graphique sera implémenté concrètement dans l'un ou l'autre des moteurs graphiques :

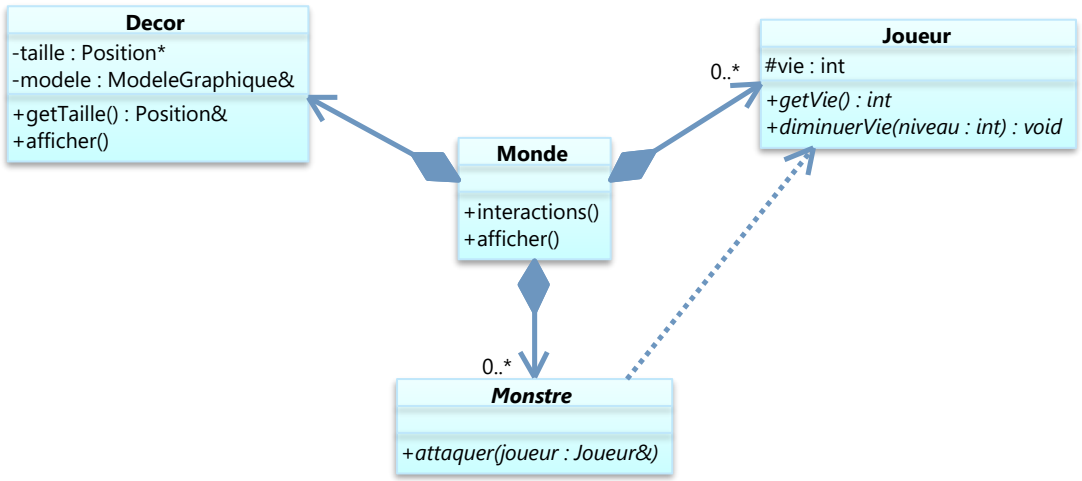


Enfin, les monstres doivent pouvoir être sous-classés en différents monstres avec des caractéristiques d'attaques différentes (plus ou moins méchant). De même pour les joueurs qui

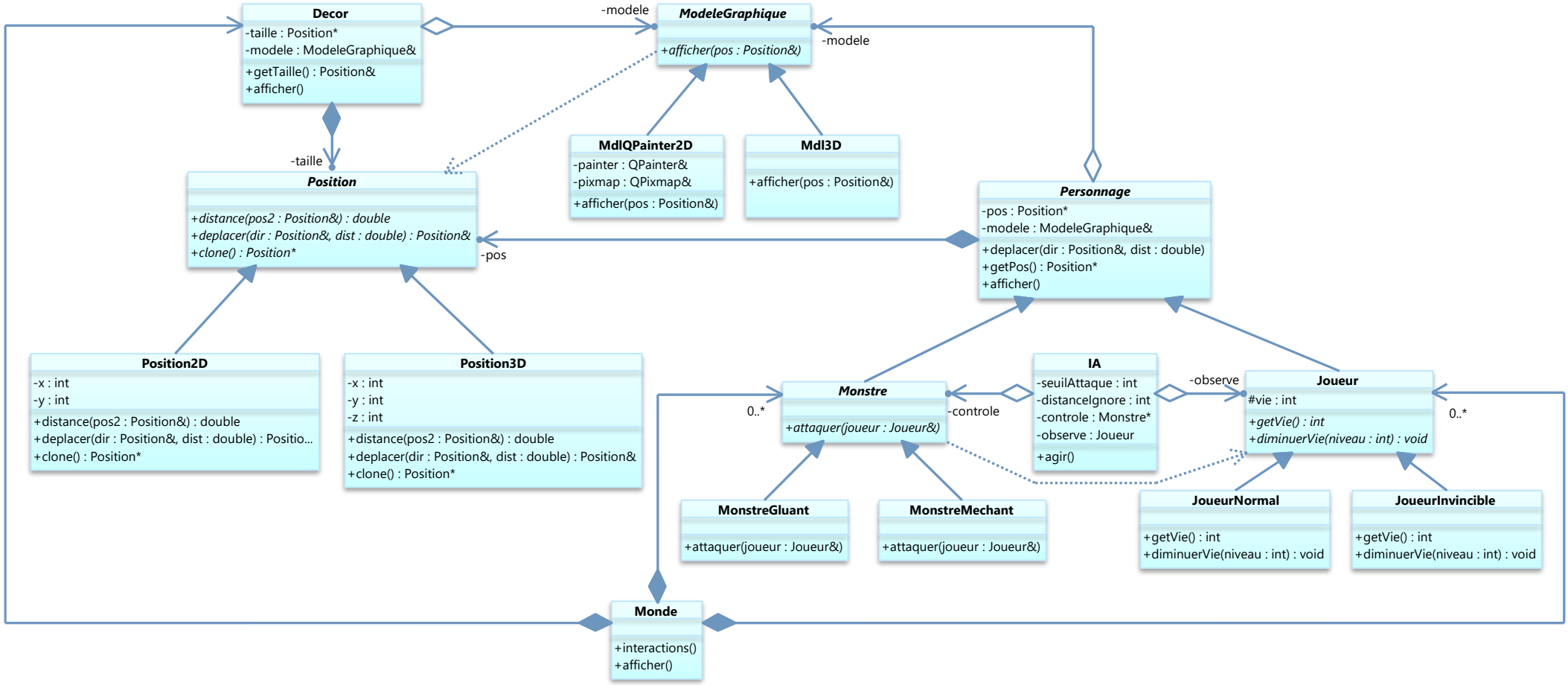
peuvent avoir des comportements différents pour la perte de points de vie (par exemple pour gérer le niveau de difficulté du jeu, ou pour avoir un joueur invincible…). Les Joueurs et les Monstres partagent tous le fait d'être des personnages graphiques (une position, un modèle graphique) qui peuvent se déplacer. Je propose donc de faire une super-classe Personnage regroupant Monstres et Joueurs. Ainsi, le diagramme au niveau des personnages devient :



Enfin, que se passe-t-il autour du monde :



Finallement, voici le diagramme complet, incompréhensible si on le regarde d'un coup, mais pas si bête que ça :


Decor
-taille : Position*
-modele : ModeleGraphique&
+getTaille() : Position&
+afficher()
-modele
ModeleGraphique
+afficher(pos : Position&)
-modele
-taille
Position
+distance(pos2 : Position&) : double
+deplacer(dir : Position&, dist : double) : Position&
+clone() : Position*
-pos
MdlQPainter2D
-painter : QPainter&
-pixmap : QPixmap&
+afficher(pos : Position&)
Mdl3D
+afficher(pos : Position&)
Personnage
-pos : Position*
-modele : ModeleGraphique&
+deplacer(dir : Position&, dist : double)
+getPos() : Position*
+afficher()
Position2D
-x : int
-y : int
+distance(pos2 : Position&) : double
+deplacer(dir : Position&, dist : double) : Positio...
+clone() : Position*
Position3D
-x : int
-y : int
-z : int
+distance(pos2 : Position&) : double
+deplacer(dir : Position&, dist : double) : Position&
+clone() : Position*
Monstre
+attaquer(joueur : Joueur&)
0..*
-controle
IA
-seuilAttaque : int
-distanceIgnore : int
-controle : Monstre*
-observe : Joueur
+agir()
-observe
Joueur
#vie : int
+getVie() : int
+diminuerVie(niveau : int) : void
0..*
MonstreGluant
+attaquer(joueur : Joueur&)
MonstreMechant
+attaquer(joueur : Joueur&)
JoueurNormal
+getVie() : int
+diminuerVie(niveau : int) : void
JoueurInvincible
+getVie() : int
+diminuerVie(niveau : int) : void
Monde
+interactions()
+afficher()