

TP Supplémentaire 04 – Implémentation de BAKaway



L'objectif de ce TP est d'implémenter la gestion des positions dans le code source de BAKaway du TD 12 – Dependency Inversion Principle.

Exercice 0 Initialisation

Comme cela devient une habitude, vous allez devoir récupérer le code source initial depuis un dépôt git.



Clonez ce dépôt : [git@gitlab-lepuy.iut.uca.fr:TPs-QualiteDeDev/tp-bakaway.git](https://gitlab-lepuy.iut.uca.fr:TPs-QualiteDeDev/tp-bakaway.git)

Exercice 1 Remise en forme

Pour rappel, la figure suivante représente le diagramme de classes établi en fin de TD (n'hésitez pas à utiliser le zoom !).

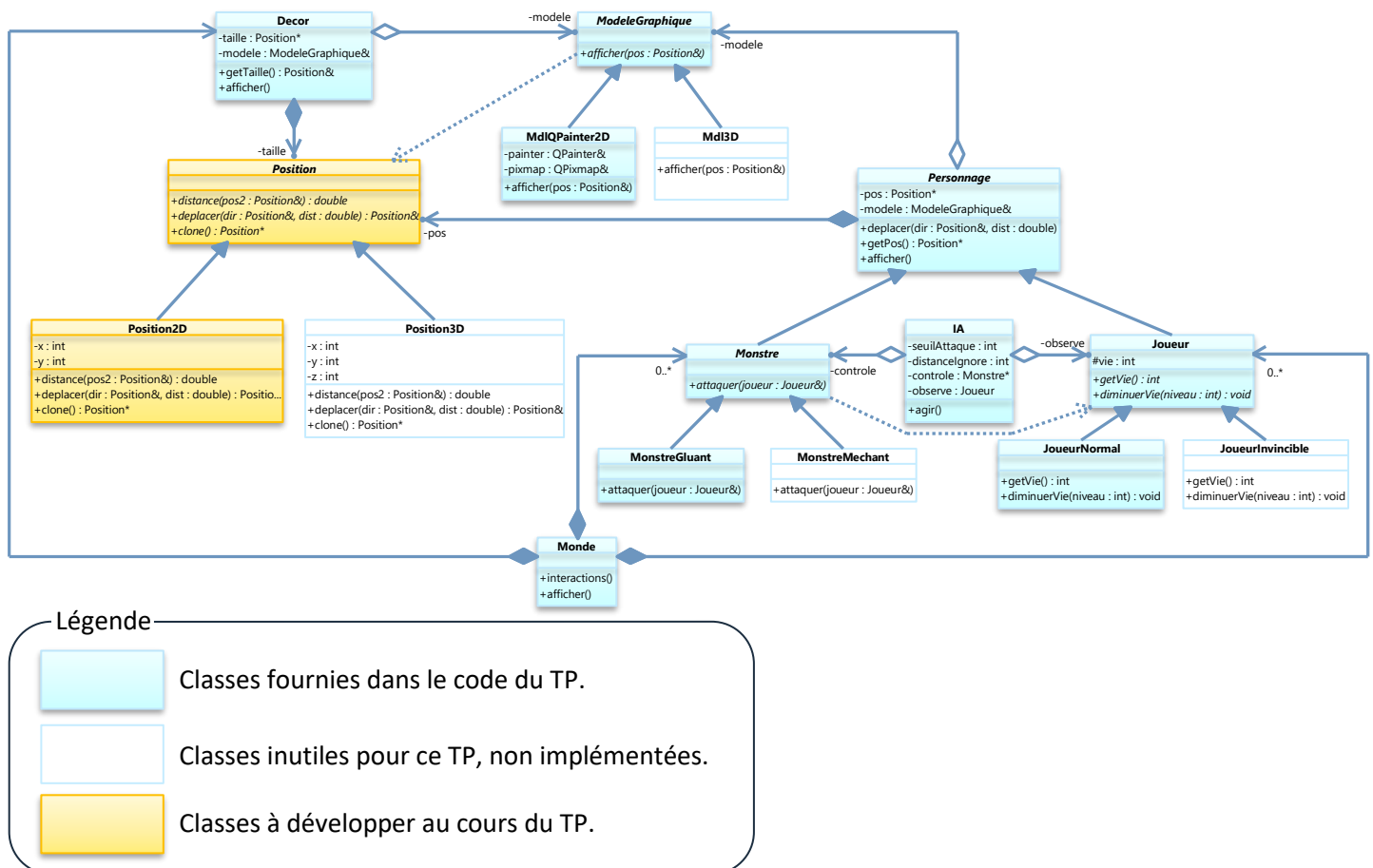


Figure 1 : Diagramme de classe du programme BAKaway en fin de TD



Compilez le programme, vous devriez obtenir ce résultat à l'exécution :

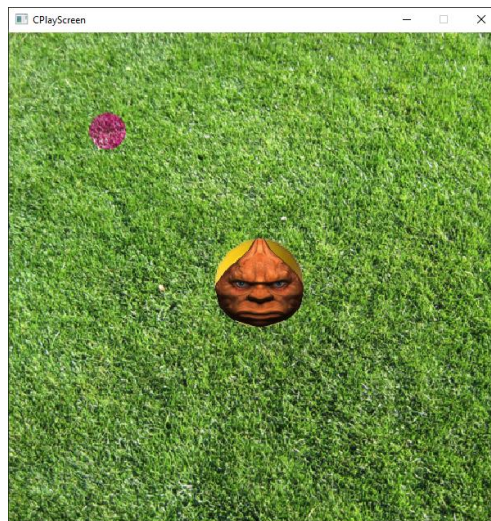


Figure 2 : Projet à l'initial avec aucune interaction.

Comme vous pouvez le remarquer, rien ne bouge. Toutes traces des classes **CPosition** et **CPosition2D** ont été supprimées du code qui vous a été fourni. Votre objectif sera de recréer et réintégrer ces classes.

Exercice 2 La classe mère CPosition

2.1 Première tentative

Cette classe est destinée à représenter de façon abstraite la position de nos différents éléments du jeu (personnages et décor). Sachant qu'elle est abstraite, elle ne peut être instanciée, et sera donc exclusivement manipulée via des pointeurs (intelligents). Afin d'obliger l'usage de pointeurs intelligents, nous allons mettre en place un mécanisme interdisant l'instanciation de cette famille de classes autrement que par l'usage de pointeurs intelligents. Pour se faire, nous allons utiliser ce genre de structure :

```
class CMaClasse
: public std::enable_shared_from_this<CMaClasse>
{
public:
    //Raccourcis de types
    using ptr = std::shared_ptr<CMaClasse>;
    using const_ptr = std::shared_ptr<const CMaClasse>;

    //Procure une copie polymorphe
    virtual ptr clone() const = 0;
    //Surcharge de l'opérateur & qui retourne l'adresse via un pointeur
    //intelligent
    ptr operator &() { return shared_from_this(); }
    const_ptr operator &() const { return shared_from_this(); }
};
```

Déclaration d'une classe capable de connaître le pointeur intelligent associé à this (voir enable_shared_from_this)

```

class CMaClasseFille
: public CMaClasse
{
    //Constructeur privé -> la classe ne peut être instancié que par
    //une méthode membre
    CMaClasseFille() = default;

    //Constructeur par recopie privé -> Les objets de type
    CMaClasseFille
    //ne peuvent être copiés que depuis une méthode membre
    CMaClasseFille(const CMaClasseFille&) = default;

public:
    //Raccourcis de types
    using ptr = std::shared_ptr<CMaClasseFille>;
    using const_ptr = std::shared_ptr<const CMaClasseFille>;

    //Méthode statique de fabrique
    static ptr create()
    {
        return ptr(new CMaClasseFille);
    }

    //Implémentation de la copie polymorphe
    CMaClasse::ptr clone() const override
    {
        return ptr(new CMaClasseFille(*this));
    }
};

```

Déclaration d'une classe fille concrète non instanciable autrement que via un pointeur intelligent.

Avec de telles classes, la création d'objets de type **CMaClasseFille** n'est possible que par l'usage de pointeurs intelligents, comme l'indique l'extrait de code ci-dessous :

```

CMaClasse MonObjet;
//|-> Impossible comme tout type de création d'objet de type CMaClasse,
//    la classe étant abstraite par la présence d'une fonction virtuelle
//    pure (clone)

CMaClasseFille MonObjetFils;
//|-> Impossible car le constructeur de la classe CMaClasseFille est
//    privé

CMaClasseFille::ptr pMonObjetFils = CMaClasseFille::create();
//|-> OK : création d'un objet par instanciation dynamique et
//    association à un pointeur intelligent

CMaClasseFille MonObjetCopie(*pMonObjetFils);
//|-> Impossible car le constructeur par recopie de la classe CMaClasse
//    est privé

```

De plus, la recherche de l'adresse d'un objet de cette famille par l'opérateur **&** retourne un pointeur intelligent de type **CMaClasse::ptr** (i.e. **std::shared_ptr<CMaClasse>**) par la surcharge de l'opérateur **&** dans la classe **CMaClasse** :

```
CMaClasse& RefVersObjet = *pMonObjetFils;
CMaClasse::ptr pMonObjet = &(RefVersObjet);
```

Utilisation de l'opérateur **&** personnalisé. La première ligne définit une référence vers un objet, la seconde ligne retrouve le pointeur intelligent vers cet objet référencé.

En suivant cette idée de structure, vous allez commencer dans les fichiers *position.h* et *position.cpp* par réalisez le travail suivant :

- 📁 Implémentez une classe abstraite **CPosition** qui a pour membres :
 - ➡ Une fonction virtuelle pure **norme()** dont l'objectif des classes filles sera de retourner en **double** la norme du vecteur position.
 - ➡ Une fonction virtuelle pure **normer()** dont l'objectif des classes filles sera de changer la taille du vecteur position de façon qu'il soit de norme égale à 1. Cette fonction devra retourner un pointeur intelligent vers l'objet courant.
 - ➡ Une fonction virtuelle pure **distance()** dont l'objectif des classes filles sera de calculer la distance entre 2 positions du même type. La fonction prendra donc en paramètre un pointeur intelligent vers un objet **CPosition** et retournera un **double**.
 - ➡ Une fonction virtuelle pure **deplacer()** dont l'objectif des classes filles sera de modifier la position selon une direction et pour une certaine distance. La fonction prendra donc en paramètre un pointeur intelligent vers un objet **CPosition** servant de vecteur directeur et un **double** pour la distance de déplacement. Enfin, elle retournera un pointeur intelligent vers l'objet courant, qui a donc été déplacé.

2.2 La taille du décor

- 📁 Munissez la classe **CDecor** d'un membre **m_pTaille** de type pointeur intelligent vers un **CPosition**.
 - ➡ Ajoutez un accesseur **getTaille()** à la classe **CDecor**
 - ➡ Ajoutez un paramètre de type référence vers un objet **CPosition** au constructeur de la classe **CDecor**. Ce paramètre sera destiné à initialiser le membre **m_pTaille**.
 - ➡ **LISEZ LA SUITE AVANT DE CRIER QUE CA NE MARCHE PAS !**

Là, il y a un problème. D'un côté, nous avons un pointeur intelligent vers un **CPosition** **m_pTaille** et de l'autre une référence vers un objet **CPosition**. Pour rappel, **m_pTaille** est un pointeur car nous ne pouvons pas stocker d'objet **CPosition** directement puisque **CPosition** est une classe abstraite. Ce pointeur doit alors être initialisé en pointant vers une copie de la position passée en paramètre du constructeur. Cette position est passée par référence (nous aurions pu choisir de la passer par un pointeur intelligent également, mais en aucun cas par valeur, puisque la classe **CPosition** est abstraite). Voici deux tentatives de code qui ne peuvent pas fonctionner :

```
CDecor::CDecor(const CPosition& taille, CModeleGraphique& modele)
: m_modele(modele)
{
    m_pTaille = std::make_shared<CPosition>(taille);
    //Appel du constructeur par copie CPosition(taille)
}
```



C'est mal car on ne peut pas créer un objet CPosition car la classe est abstraite → ne compile pas

```
CDecor::CDecor(const CPosition& taille, CModeleGraphique& modele)
: m_modele(modele)
{
    m_pTaille = &taille;
    //Stockage d'une adresse (pointeur "con") dans un pointeur intelligent
}
```

C'est mal car on ne peut pas affecter un pointeur « con » à un pointeur intelligent → ne compile pas. C'est d'autant plus mal que l'idée serait de stocker la référence d'un objet qui va sûrement être rapidement détruit.

Le problème à résoudre est donc : « *comment réaliser la copie d'un objet manipulé par une interface abstraite ?* » Ce qui revient à dire « comment réaliser la copie d'un objet qu'on ne connaît pas précisément ? »

La solution est d'implémenter une *copie polymorphe*, appelée aussi *clonage* ou encore *design pattern Prototype*. Il s'agit d'ajouter à la classe abstraite dont on veut avoir la possibilité d'avoir des copies d'instances la fonction virtuelle pure **clone()**. Dans chacune des classes filles, cette fonction doit retourner un pointeur vers une copie de l'objet courant.




-  Munissez la classe **CPosition** d'une fonction virtuelle pure **clone()** dont le type de retour est un pointeur intelligent vers un objet **CPosition**.
-  Définissez le constructeur de la classe **CDecor** de façon à initialiser le membre **m_pTaille** avec un clone du paramètre **taille**.

2.3 Un peu de concret

Pour que cela fonctionne, il va falloir changer la création du décor. Cela est fait dans le constructeur de la classe **CPlayScreen** :

```
CDecor decor(m_md1Decor);
```

Il faut donc ajouter le paramètre de la taille du décor. Nous ne pouvons pas passer un objet **CPosition** car **ON NE PEUT PAS AVOIR D'INSTANCE DE CPosition** ! Il va donc falloir créer une classe fille **CPosition2D**.

-  Implémentez la classe **CPosition2D**
 -  Ajoutez donc les surcharges des fonctions virtuelles pures.
 -  Ajoutez deux membres **m_dX** et **m_dY** en **double** ainsi que leurs accesseurs (getter et setter).

- ☞ Modifiez la ligne de création du décor dans le constructeur de **CPlayScreen** de façon à passer une position 2D de taille 50x50 au constructeur de **CDecor**.
- ☞ Compilez, exécutez.
 - ☞ La compilation doit fonctionner, l’affichage doit rester le même (nous ne l’avons pas encore géré).

Nous avons mis en place la mécanique de la position virtuelle du décor, mais nous ne l’avons pas fait pour son affichage. La méthode d’affichage du décor est la suivante :

```
void afficher() { m_modele.afficher(); }
```

Le membre **m_modele** est une référence vers un modèle graphique de type **CModeleGraphique**. Concrètement, il ne peut être un objet **CModeleGraphique** car la classe **CModeleGraphique** est abstraite, à l’instar de la classe **CPosition**. Concrètement, la référence se fait vers un objet de type **CMdlQPainter2D**. La méthode **afficher()** de la classe **CModeleGraphique** est virtuelle pure et destinée à afficher le modèle. Elle ne prend pas en compte pour le moment la position du modèle, et c’est ce que nous nous apprêtons à ajouter.

- ☞ Modifiez la signature de la fonction **CModeleGraphique::afficher()** de façon qu’elle accepte un pointeur intelligent de type **CPosition** qui sera par défaut égal à **nullptr**.
- ☞ Répercutez cette modification dans sa classe fille **CMdlQPainter2D**
 - ☞ N’oubliez pas de modifier le corps de la fonction **CMdlQPainter2D::afficher()** pour prendre en compte le nouveau paramètre :

```
mPainter.drawPixmap(25 - m_size.width()/2, 25 - m_size.height()/2, ...
```

- ☞ Dans ce code, les « 25 » doivent être remplacés respectivement par les X et Y des positions 2D dans le cas où le pointeur passé en paramètre est différent de nul.
- ☞ Compilez et exécutez
 - ☞ Vous devriez avoir le même résultat graphique qu’au début.
 - ☞ Ca ne sert à rien ? Bon OK... mais ce n’est que le décor... Si vous modifiez les valeurs de taille du décor à la construction de l’objet **CDecor**, l’affichage devrait être modifié.

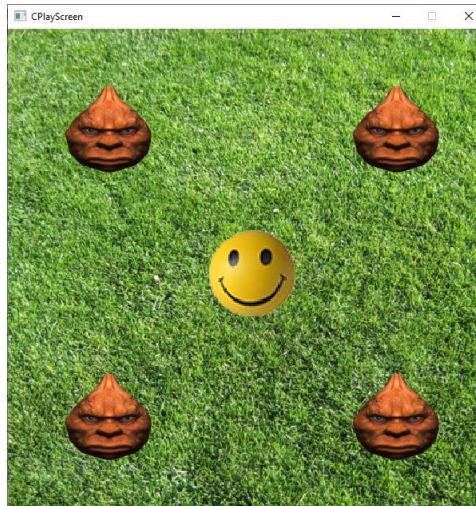
Exercice 3 Faisons bouger les personnages !

Il va donc falloir maintenant nous occuper de la position des personnages.

3.1 Initialisons-les

- ☞ Comme indiqué dans le diagramme de classe, munissez la classe **CPersonnage** d’un pointeur intelligent **m_pPos** vers **CPosition**.
 - ☞ Ajouter les accesseurs (getter et setter)
 - ☞ Modifiez la méthode **afficher** pour ajouter le paramètre **m_pPos**
 - ☞ Modifiez le constructeur pour ajouter un paramètre de position
 - ☞ Répercutez ces changements dans toutes les classes filles
- ☞ Dans le corps du constructeur **CPlayScreen**, modifiez les constructions des objets **m_pJoueur**, **m_pMonstre1**, **m_pMonstre2**, **m_pMonstre3** et **m_pMonstre4** comme suit :
 - ☞ Le joueur doit s’initialiser à la position 25, 25

- ➡ Le premier monstre doit s'initialiser à la position 10, 10
- ➡ Le deuxième monstre doit s'initialiser à la position 10, 40
- ➡ Le troisième monstre doit s'initialiser à la position 40, 10
- ➡ Le quatrième monstre doit s'initialiser à la position 40, 40
- 📁 Compilez, exécutez. Vous devriez obtenir l'affichage suivant :



3.2 Contrôlons le joueur

Les personnages doivent pouvoir être déplacés (soit par le clavier pour le joueur, soit par une IA pour les monstres). Ajoutons cette capacité aux personnages :

- 📁 Ajoutez à la classe **CPersonnage** une méthode **deplacer()** prenant en paramètre une **CPosition** de direction et une distance.
 - ➡ Son objet sera de déplacer la position du personnage (**m_pPos**) en déléguant le travail à la fonction **CPosition::deplacer()**
- 📁 Dans la fonction **CPlayScreen::keyPressEvent()**, ajoutez la réponse aux événements suivant :
 - ➡ **Qt::Key_Up** : déplacer le joueur (**m_pJoueur**) du vecteur 2D (0, -1) pour une distance égale à un dixième de ses points de vie.
 - ➡ **Qt::Key_Down** : déplacer le joueur (**m_pJoueur**) du vecteur 2D (0, 1) pour une distance égale à un dixième de ses points de vie.
 - ➡ **Qt::Key_Left** : déplacer le joueur (**m_pJoueur**) du vecteur 2D (-1, 0) pour une distance égale à un dixième de ses points de vie.
 - ➡ **Qt::Key_Right** : déplacer le joueur (**m_pJoueur**) du vecteur 2D (1, 0) pour une distance égale à un dixième de ses points de vie.
- 📁 Compilez, exécutez. Le joueur doit être contrôlable au clavier.

3.3 Contrôlons les monstres

C'est le rôle des objets de type **CIA**. Vous allez devoir modifier la méthode **CIA::agir()**

- 📁 Dans la portion du code qui parcourt les objets observés par l'IA :

```
for(auto& pObserved : m_lstObserved)
{
    double dDist = std::numeric_limits<double>::max();
    if(dDist < dMinDist)
    {
        pPlusProche = pObserved;
        dMinDist = dDist;
    }
}
```

➡ La variable **dDist** doit être calculée comme étant la distance entre l'objet observé **pObserved** et l'objet contrôlé **m_pControlled**.

📖 Dans la portion de code suivante :

```
if(pPlusProche && pPlusProche->getVie() < m_nSeuilVieAttaque)
    m_pControlled;
else
    m_pControlled;
```

➡ Dans le premier cas (joueur faible), l'objet **m_pControlled** doit être déplacé de 1 vers le joueur (il sera peut-être pratique d'implémenter un opérateur « - » entre deux **CPosition**).

➡ Dans le cas contraire, le déplacement doit être d'une distance de -1 (fuite).

📖 Compilez, exécutez. Les monstres devraient donc maintenant vous fuir !

C'est bien, mais les monstres et le joueur peuvent sortir de l'espace de jeu.

📖 Modifiez la fonction **CPlayScreen::keyPressEvent()** afin de limiter les positions des personnages à l'espace du décor.

Exercice 4 Appliquons les règles du jeu

4.1 Tuons les monstres

📖 Modifiez la méthode **CMonde::interactions()** de façon que si le joueur se trouve à une distance de moins de 5 d'un monstre, le monstre l'attaque.

➡ Vous trouverez un **if(false)** non loin de l'appel de la méthode **attaquer()** du monstre.



📖 Compilez, exécutez. A chaque fois que vous êtes près d'un monstre, celui-ci meurt et vous perdez un point de vie. Une fois trois monstres tués, le quatrième doit s'approcher de vous.

4.2 Gagnons la partie

📖 A l'instar du membre **m_pTaille**, ajoutez à la classe **CDecor** un membre **m_pObjectif** qui représentera le lieu à atteindre pour gagner la partie.

➡ Modifiez en conséquence son constructeur et ajouter le getter adéquat.

📖 Dans le constructeur de **CPlayScreen**, créez le décor en lui fixant l'objectif en position 10, 10.

-  Dans la méthode **CMonde::interactions()**, modifiez le dernier **if(false)** de façon que le signal **JoueurGagne(*itJoueur)** soit émis lorsque le joueur ***itJoueur** se trouve à une distance de moins de 1 de l'objectif.
-  Compilez, exécutez, jouez !



Vous êtes ici ? Félicitations !