

TP Supplémentaire 05 – Implémentation de BAKt

L'objectif de ce TP est d'implémenter la gestion des positions dans le code source de BAKt du TD 13 – Interface Segregation 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-bakt.git](https://gitlab-lepuy.iut.uca.fr:TPs-QualiteDeDev/tp-bakt.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 !). Les classes réalisant le template **EventManager** implémentent le pattern **Observateur** avec les classes associées **I*Listener**.

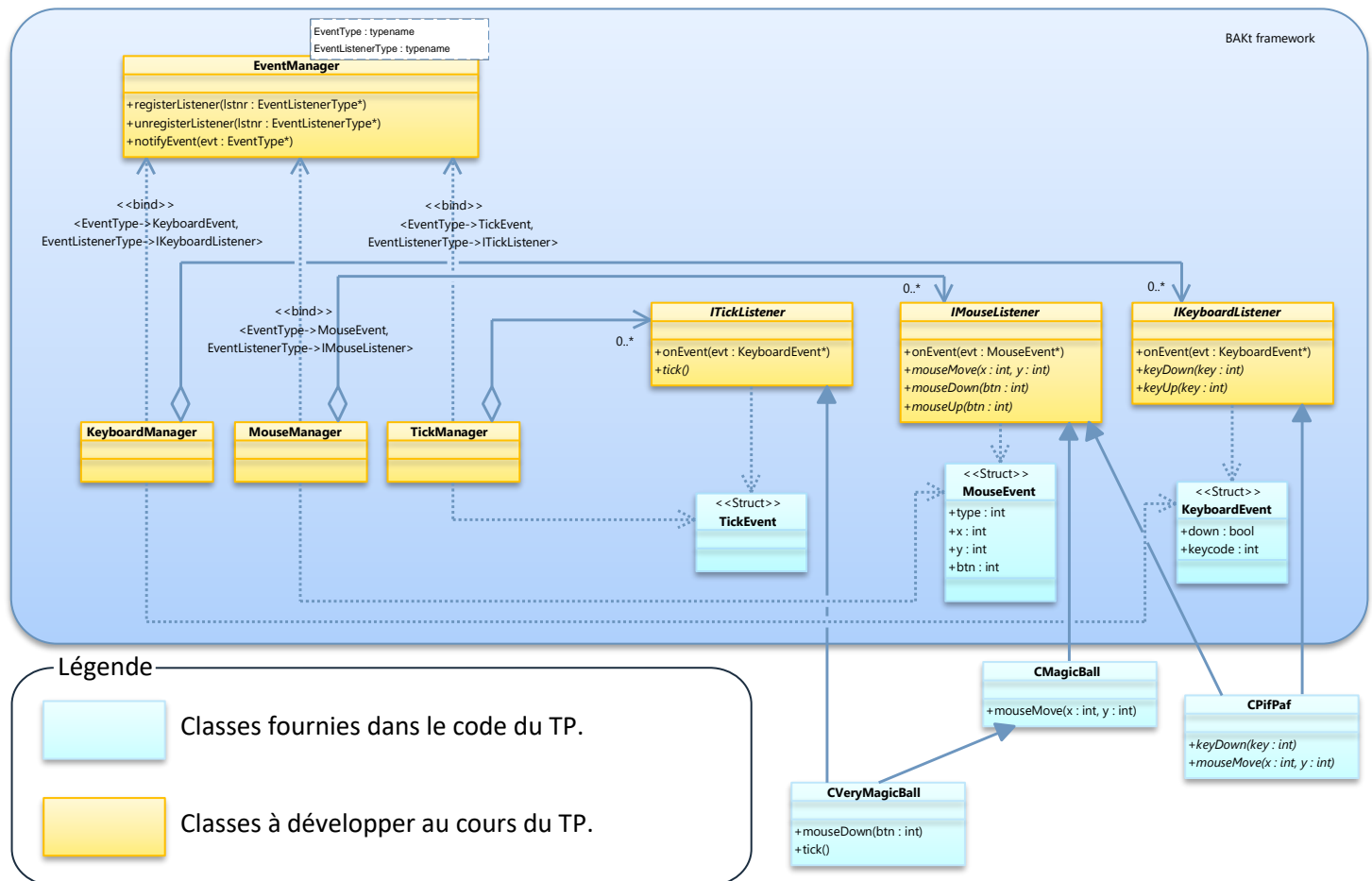


Figure 1 : Diagramme de classe du programme BAKt en fin de TD

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

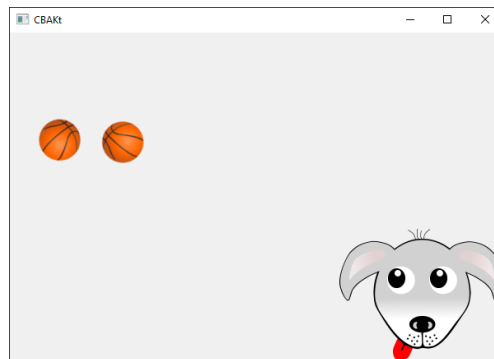
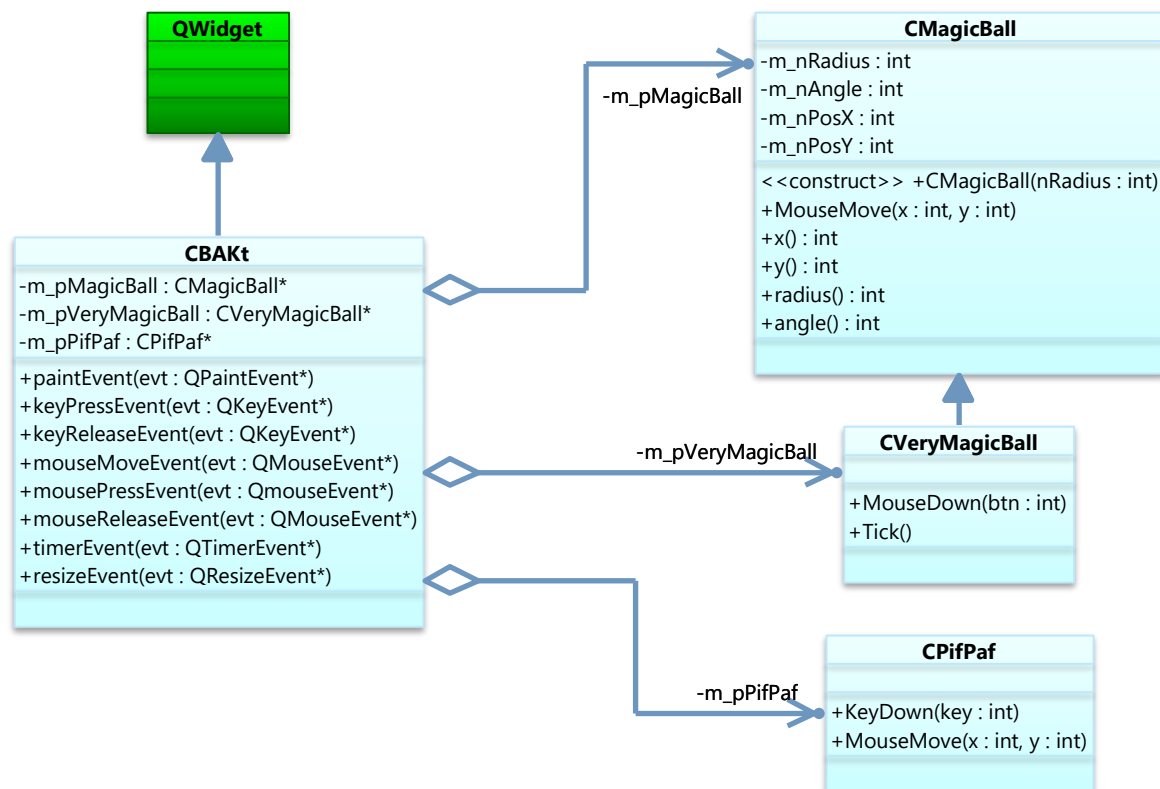
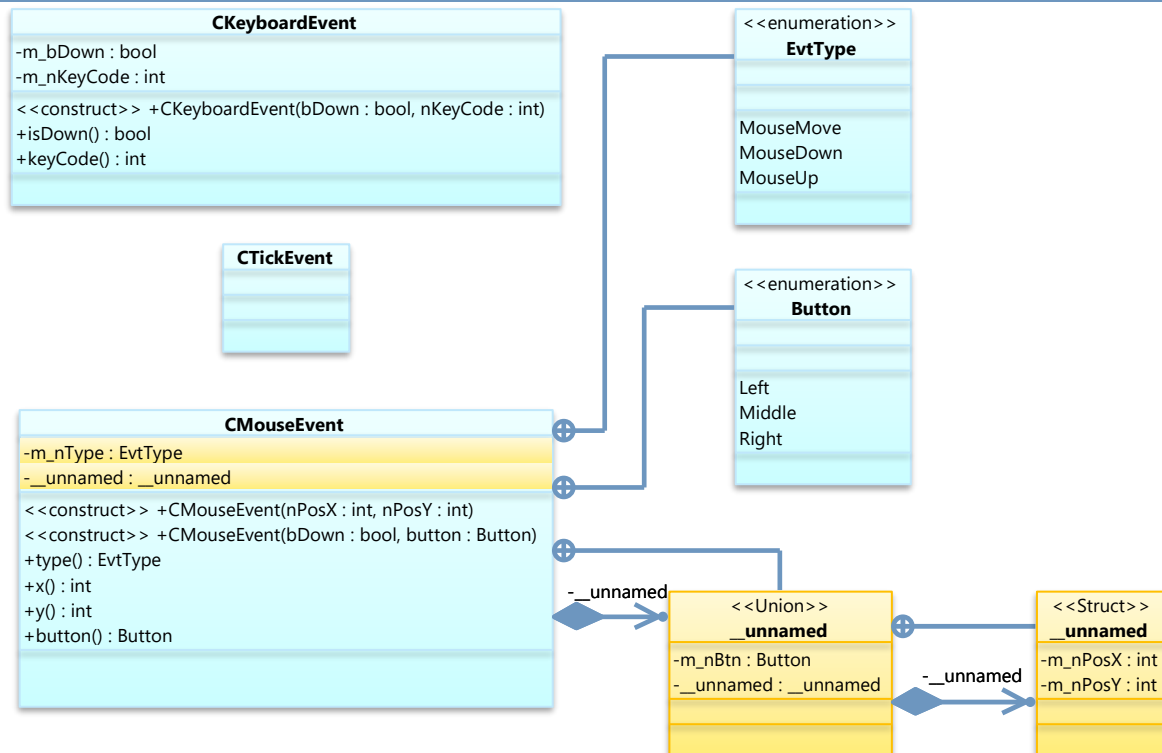


Figure 2 : Projet à l'initial avec aucune interaction.

Comme vous pouvez le remarquer, rien ne bouge... Toute la gestion d'événement a été retirée du code qui vous a été fourni. L'objectif de ce TP est donc de le redévelopper. Avant de passer à l'**Erreur ! Source du renvoi introuvable.**, prenez le temps de comprendre l'architecture du code.



La classe **CBAkt** est un **QWidget** personnalisé qui tient lieu de classe principale de notre application de démonstration. C'est dans ses méthodes de réponse aux événements Qt que vous devrez générer les événements de votre framework.



Les trois classes **CKeyboardEvent**, **CTickEvent** et **CMouseEvent** sont des versions encapsulées des structures **KeyboardEvent**, **TickEvent** et **MouseEvent** du diagramme de la figure 1 ; elles jouent le rôle de ces structures dans le code.

Observons la classe **CMouseEvent**, il y a une chose que vous n'avez probablement encore jamais rencontré. Tout d'abord, rappelons que la liaison \oplus signifie que la classe reliée est contenue dans la première. Cela signifie en C++, que la classe incluse est déclarée dans l'espace de nom de la classe englobante. Ainsi à titre d'exemple, l'énumération **Button** est déclarée à l'intérieur de la classe **CMouseEvent**. Ensuite, pour comprendre la partie du diagramme mis en exergue, il faut connaître le concept d'**union C++**. Le C++ permet de créer des structures dont les membres partagent le même espace mémoire : les unions. Cela signifie donc qu'un seul membre d'une union ne peut être actif. Par exemple, le code suivant permet d'avoir des variables **nb1** et **nb2** qui chacune stockent soit un entier, soit un double :

```

union UVal {
    int n;
    double d;
};

UVal nb1{1}; //nb1 stocke l'entier 1. Il est accessible via nb1.n
UVal nb2{2.5}; //nb2 stocke le double 2.5. Il est accessible via nb2.d;
//Accéder à nb1.d retournera une valeur loufoque car la représentation
//binaire de l'entier 1 sera interprété comme un double.
//De même accéder à nb2.n retournera également une valeur loufoque.

```

Dans notre cas, le code correspondant à la partie du diagramme mis en exergue est le suivant :

```
class CMouseEvent
{
    EvtType m_nType;
    union {
        Button m_nBtn;
        struct {
            int m_nPosX;
            int m_nPosY;
        };
    };
};
```

Ce code fait appel à une structure anonyme incluse dans une union anonyme. En C++, Les éléments anonymes apportent leurs membres dans l'espace de nom supérieur. Donc ici, les membres **m_nPosX** et **m_nPosY** de la structure anonyme sont accessibles directement depuis l'union. Mais l'union est également anonyme donc les membres **m_nBtn**, **m_nPosX** et **m_nPosY** sont accessibles directement depuis la classe **CMouseEvent**. Donc depuis la classe **CMouseEvent**, on peut accéder soit à **m_nBtn**, soit à la position d'un point avec **m_nPosX** et **m_nPosY**. Pour savoir quel membre est actif, le membre **m_nType** est tout à fait approprié, car c'est le type d'événement souris qui détermine si la position du curseur est utile ou si c'est l'identification du bouton. Ce genre de classe s'appelle une **tagged union** ou **variant** ou encore en français « **type somme** » (il y a encore plein de synonyme). Pour finir cette explication, le code suivant est un cas d'usage présentant comment distinguer le type d'événement et comment accéder aux membres de l'union.

```
void CMouseEvent::QueFaireAvec()
{
    switch (m_nType)
    {
        case CMouseEvent::EvtType::MouseDown:
        case CMouseEvent::EvtType::MouseUp:
            std::cout << "Evénement de type bouton de code " << m_nBtn;
            break;
        case CMouseEvent::EvtType::MouseMove:
            std::cout << "Déplacement de souris en " << m_nPosX << ", " << m_nPosY;
            break;
        default:
            break;
    }
}
```

Exercice 2 Mise en place d'un gestionnaire d'événement souris

2.1 La classe `IMouseListener`

Pour commencer, dans les fichiers *IEventListener.h* et *IEventListener.cpp*, nous allons créer une classe **`IMouseListener`** telle que décrite dans la figure 1 en suivant les étapes suivantes :

- 📄 Créez les méthodes **`MouseMove(int x, int y)`**, **`MouseDown(CMouseEvent::Button btn)`** et **`MouseUp(CMouseEvent::Button btn)`** avec un corps vide.
- 📄 Créez la méthode **`OnEvent`** prenant en paramètre une référence constante vers un objet de type **`CMouseEvent`**
 - ➡ Cette fonction est chargée d'appeler les fonctions **`MouseMove`**, **`MouseDown`** et **`MouseUp`** en fonction de l'état du paramètre de type **`CMouseEvent`**.

2.2 La classe `CMouseManager`

La classe **`IMouseListener`** est prête, il faut s'occuper de créer le **`CMouseManager`**. Pour rappel, son rôle est de notifier tous les **`IMouseListener`** qui y sont associés dès qu'un événement souris se produit. Il va falloir gérer ces associations :

- 📄 Dans les fichiers *EventManager.h* et *EventManager.cpp*, créez la classe **`CMouseManager`** :
 - ➡ Munie d'une collection de pointeurs vers des objets de type **`IMouseListener`**
 - ➡ Munie d'une méthode **`registerListener`** qui, prenant en paramètre un pointeur vers un **`IMouseListener`**, associe l'objet **`IMouseListener`** en ajoutant son pointeur dans la collection.
 - ➡ Munie d'une méthode **`unregisterListener`** qui, prenant en paramètre un pointeur vers un **`IMouseListener`**, dissocie l'objet **`IMouseListener`** en retirant son pointeur de la collection.
 - ➡ Munie d'une méthode **`notifyEvent`** qui, prenant en paramètre une référence vers un **`CMouseEvent`**, appelle les fonctions **`OnEvent`** de tous les **`IMouseListener`** associés.

2.3 Intégration dans le code existant

Il va falloir maintenant faire fonctionner tout cela...

- 📄 Dans la classe **`CBAkt`** :
 - ➡ Ajoutez un membre de type **`CMouseManager`**, il sera votre objet de gestion des événements souris pour votre framework.
 - ➡ Dans le constructeur, associez les objets pointés par **`m_pMagicBall`**, **`m_pVeryMagicBall`** et **`m_pPifPaf`** au nouveau **`CMouseManager`**.
 - ➡ Vous devrez donc faire en sorte que les **`CMagicBall`**, **`CVeryMagicBall`** et **`CPifPaf`** soient considérés comme des **`IMouseEventListener`**.
 - ➡ Dans les méthodes de réception des événements Qt lié à la souris (**`mouseMoveEvent`**, **`mousePressEvent`** et **`mouseReleaseEvent`**), demandez la notification de tous les **`IMouseListener`** associés au **`CMouseManager`**.
 - ➡ Je vous ai gentiment laissé les objets **`CMouseEvent`** initialisés correctement.




Le plus gros est fait.

- 📄 Compilez, exécutez.
 - ➡ Cela doit fonctionner pour tous les événements souris.

Exercice 3 Mise en place d'un gestionnaire d'événement de temps



L'objectif de cet exercice est d'implémenter l'architecture autour des classes **ITickListener** et **CTickListener**.

3.1 La classe **ITickListener**

-  A l'instar de la classe **IMouseListener**, implémentez la classe **ITickListener** dans les fichiers *IEventListener.h* et *IEventListener.cpp*. Elle doit avoir :
 -  Une méthode **Tick**.
 -  Une méthode **OnEvent** prenant en paramètre une référence vers un objet **CTickEvent**. Le rôle de cette fonction sera d'appeler la méthode **Tick**.

3.2 La classe **CTickManager**



Il est aisé de remarquer que cette classe est sensiblement identique à la classe **CEventManager**. Les différences sont uniquement sur les types d'objets manipulés, mais pas sur la façon de les manipuler. Il serait donc intéressant, plutôt que de copier – coller le code, d'utiliser la capacité du C++ à écrire du code générique en utilisant un template de classe :

-  Transformez la classe **CEventManager** en un template de classe **TEventManager**, paramétré par le type d'événement envoyé (**CMouseEvent**, **CTickEvent**, ...) et par le type de listener associé (**IMouseListener**, **ITickListener**, ...)
-  En utilisant le mot clé **using**, vous pouvez recréer la classe **CEventManager** simplement par un code du style suivant :







```
using CEventManager = TEventManager<CMouseEvent, IMouseListener>;
```

-  Sur le même principe, créez la classe **CTickManager**.

3.3 Intégration dans le code existant

-  En vous inspirant de l'exercice 2.3, faites fonctionner les événements de temps.
-  Compilez, exécutez. La balle très magique doit se mettre à tourner automatiquement.

Exercice 4 Mise en place d'un gestionnaire d'événement clavier

-  Sur le même principe que l'exercice 3, implémentez la mécanique des événements clavier
-  Compilez, exécutez. Le programme doit être totalement fonctionnel :
 -  PifPaf suit la souris des yeux.
 -  La balle magique tourne autour de la souris lorsque cette dernière est déplacée.
 -  La balle très magique tourne automatiquement autour de la souris et s'éloigne / s'approche sur les clics gauche / droit de la souris.
 -  PifPaf tire la langue vers la gauche sur l'appuie de la touche ← et vers la droite sur l'appuie de la touche →.



Vous êtes ici ? Félicitations !