
Observateur

Pattern "Observateur"

Le pattern "Observateur" est décrit dans le livre "Design Pattern" de Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides.

En résumé,

- Un "sujet" est une classe notifiant des observateurs lors de ses modifications.
- Un "Observateur" est une classe recevant des notifications d'un sujet.

Chaque observateur doit s'enregistrer chez un sujet. Plusieurs types de notifications doivent parvenir à un "Observateur". Des notifications d'instance, des notifications de classe ou de ses dérivées, et des notifications de classe mais sans les dérivées.

Un sujet doit notifier les observateurs lors de sa modification. Il doit informer le plus précisément possible la cause de la notification. Cela permet à l'observateur de ré-agir précisément suivant les modifications du sujet.

API

L'Api que je propose permet de gérer automatiquement les enregistrements de l'observateur sur le sujet, et de construire automatiquement la fonction C faisant le lien avec une méthode C++. Les risques d'erreurs sont réduits au minimum.

Les sujets et les observateurs n'ont pas de contraintes, ils ne doivent pas hériter d'une classe particulière. Le type d'information décrivant une notification est paramétrable.

Le sujet

Le premier sujet d'une hiérarchie voulant offrir une notification doit ajouter dans la description de la classe la macro `DECLARE_FIRST_SUBJECT(Type)` où `Type` est le type de notification transmise. Dans le source `.cc` il faut ajouter la macro `IMPLEMENT_SUBJECT(Classe,Type)` où `Classe` est le nom de la nouvelle classe, et `Type` le type de notification transmise.

```
class CBook
{ public:
    DECLARE_FIRST_SUBJECT(long)
    //...
};
IMPLEMENT_SUBJECT(CBook, long)
```

Lors d'un traitement quelconque, les méthodes de `CBook` doivent appeler la méthode `notify(Type)` pour prévenir tous les observateurs enregistrés.

Les classes dérivées doivent utiliser une macro de déclaration légèrement différente. Il faut utiliser la macro `DECLARE_SUBJECT(Base,Type)` où `Base` est la classe de base directement héritée, et `Type` est le type de la notification.

```
class CExBook : public CBook
{ public:
    DECLARE_SUBJECT(CBook, long)
};
IMPLEMENT_SUBJECT(CExBook, long)
```

Ces macros doivent être déclarées dans la partie `public` de la classe. Les macros `IMPLEMENT` doivent être dans le `.cc` de chaque classe.

Un observateur ne peut pas s'enregistrer auprès d'une instance pour être notifié de sa création. La notification de classe palie à cette carence.

L'observateur

L'observateur est une classe classique qui possède un objet simulant un pointeur sur le sujet. Cet objet s'utilise comme n'importe quel pointeur, mais enregistre l'observateur chez le sujet et s'occupe également de la suppression de l'enregistrement. La déclaration de ce pointeur est un petit peu complexe car il faut fournir trois paramètres dans le type et ajouter le constructeur correspondant. Il existe trois types de «pointeurs de notification».

Notification d'instance

Le type `CRefNotify<Subject,Observer,noti>` permet d'enregistrer une notification d'instance. `Subject` est le type de sujet concerné, `Observer` est le type de la classe Observateur et `noti` est le type de notification.

```
class ObserverBook
{ CRefNotify<CBook,ObserverBook,long> _book;
  //...
};
```

Cette écriture déclare un pointeur `_book` sur un objet `CBook`. Celui-ci s'utilise comme s'il était déclaré comme : `"CBook* _book;"`.

Attention, il n'est pas recommandé de garder un autre pointeur sur le sujet dans l'observateur. Le template est là pour remplacer un éventuel pointeur vers le sujet.

Pour construire ce pointeur, il faut fournir trois paramètres. Le premier est l'instance `this`, le deuxième est l'adresse de la méthode devant être appelé par le sujet lors de la notification, et le dernier paramètre optionnel est l'adresse de l'instance `sujet` en relation.

```
class ObserverBook
{ CRefNotify<CBook,ObserverBook,long> _book;
  ObserverBook(CBook* book)
  : _book(this,&ObserverBook::notify,book)
  { }
  //...
  void notify(CBook* subject,long x);
};
```

Le type de la méthode de notification reçoit comme premier paramètre l'instance du sujet concerné (constant ou non), et en deuxième paramètre l'objet de description de notification. Toute modification du sujet pointé appelle la méthode `notify` de l'observateur. Si vous désirez modifier la référencée sur un `_book`, utilisez l'affectation. Cela supprimera l'enregistrement de l'ancien book et enregistrera l'observateur dans le nouveau book. Tout le mécanisme d'enregistrement est transparent. Il faut juste indiquer le bon type de pointeur, et appeler le constructeur correctement. Ensuite, il n'y a plus rien à faire.

Pour avoir un pointeur constant sur le sujet, utilisez la classe `CCRefNotify<Subject,Observer,noti>`. Pour supprimer une relation de notifi-

cation, il faut valoriser le pointeur avec `NULL`. Si vous détruisez le sujet, il faut auparavant reseter le pointeur avec `NULL`. Sinon, une postcondition du sujet indiquera qu'il existe encore des observateurs enregistrés pour ce sujet.

Notification de classe

Il n'est pas possible de s'enregistrer pour une instance n'existant pas encore. La construction d'un sujet ne peut pas notifier un observateur. Pour palier à cela, il est possible de s'enregistrer pour une classe entière. La technique est équivalente à la technique décrite précédemment. Toutes modifications des instances de la classe `sujet` ou de ses dérivées vont notifier l'observateur. La description de la notification permet éventuellement de filtrer les notifications intéressantes. Pour cela, il faut déclarer un objet dans l'instance `observateur` s'occupant de l'enregistrement de la notification. La classe `CRecordNotify<Subject,Observer,noti>` permet d'enregistrer un observateur pour une classe d'un sujet. Le constructeur d'une instance de ce type reçoit deux paramètres : l'instance observateur et la méthode notifiée.

```
class ObserverBook
{ CRecordNotify<CBook,ObserverBook,long> _notifyClass;
  ObserverBook()
  : _notifyClass(this,&ObserverBook::notifyClass)
  { }
  //...
  void notifyClass(CBook* subject,long x);
};
```

Si la classe `CBook` appelle sa méthode `notify()` dans son constructeur; alors la méthode `notifyClass` de `ObserverBook` est appelée. Lors des notifications de toutes les instances de la classe `CBook` et de ses dérivées, la méthode `notifyClass` est appelé. Il est préférable de rédiger un conteneur de `CBook` et d'enregistrer une notification sur ce conteneur. Il faut éviter d'utiliser la notification de classe car cela représente une notification d'un conteneur virtuel de toutes les instances d'une classe. L'utilisateur peut construire des objets temporaires qui n'ont pas à notifier les observateurs. Il faut écrire un observateur de conteneur plutôt qu'un observateur de classe.

Notification de classe exclusive

Il peut être utile d'être notifié pour une classe particulière mais pas pour ses dérivées. Dans ce cas, il faut utiliser la classe `CRecordNotifyEx<Subject,Observer,noti>` pour initialiser une notification exclusive. La syntaxe est la même que précédemment.

Guide d'utilisation

Implantation

La déclaration des macros se trouve dans le fichier "`observer.hh`". Ce fichier doit être inclus dans le `.hh` du sujet. L'observateur doit inclure le `.hh` du sujet.

Le Sujet

Lors de la rédaction des méthodes du sujet; il faut offrir une information pertinente de notification. Celle-ci peut être simple comme un entier, ou plus complexe comme un objet décrivant l'action et les acteurs impliqués dans l'action. Par exemple, la classe `CBook` peut notifier simplement comme ceci :

```

class CBook
{ public:
  DECLARE_FIRST_SUBJECT(int)
  enum { AddTrade,SubTrade };
  //...
  void addTrade(CTrade* Trade)
  { //...
    notify(AddTrade);
  }
};

```

Il peut être intéressant pour les observateurs de connaître plus de détail sur l'action.

```

struct CNotifyBook
{ enum TCmd { AddTrade,SubTrade, Calcul };
  TCmd _cmd;
  CTrade* _Trade;
  CNotifyBook(TCmd cmd,CTrade* trade=NULL)
  : _cmd(cmd), _trade(trade)
  { }
};

```

```

class CBook
{ public:
  DECLARE_FIRST_SUBJECT(const CNotifyBook&)
  void addTrade(CTrade* Trade)
  { //...
    notify(CNotifyBook(CNotifyBook::AddTrade,Trade));
  }
  void calcul()
  { //...
    notify(CNotifyBook::Calcul);
  }
};

```

Il est parfaitement possible d'être enregistré sur plusieurs sujets différent. Un même observateur peut avoir plusieurs références sur différent sujet de même type ou de type différent. Le pointeur reçu en paramètre dans la Call-Back de la notification permet de différencier le sujet concerné par la notification. La même Call-Back peut être partagé entre plusieurs sujets du même type.

Héritage

Lors d'un héritage, il faut faire attention à l'emplacement de la notification. En effet, il est facile d'avoir deux notifications pour le même service. Imaginons la classe `CVehicule`. Celle-ci peut être rédigée comme cela :

```

class CVehicule
{ enum TEtat { Avance,Stop };
  TEtat _etat;
  public:
  DECLARE_FIRST_SUBJECT(int)
  CVehicule()
  : _etat(Stop) {}
  virtual void avance()
  { _etat=Avance;
    notify(Avance);
  }
  //...
};

```

La notification intervient lorsque le traitement `avance()` est complètement terminé. Si maintenant on rédige la classe `CVoiture` héritant de `CVehicule`, cela donne :

```
class CVoiture : public CVehicule
{ public:
  DECLARE_SUBJECT(CVehicule,int)
  virtual void avance()
  { Vehicule::avance();
    //... Lance le moteur
    notify(Avance);
  }
};
```

Une classe `Observateur` recevra deux notifications lors de l'appel de la méthode `CVoiture::avance()`. En effet, cette méthode appelle la méthode de la classe de base qui finit par appeler la notification, puis la notification de `CVoiture::avance()` est appelé. Pour éviter ce type de problème il faut rédiger les services autrement.

```
class CVehicule
{ enum TEtat { Avance,Stop };
  TEtat _etat;
  public:
  DECLARE_FIRST_SUBJECT(int)
  CVehicule()
  : _etat(Stop) {}
  virtual void _avance()
  { _etat=Avance;
  }
  void avance()
  { _avance();
    notify(Avance);
  }
  //...
};
class CVoiture : public CVehicule
{ public:

  DECLARE_SUBJECT(CVehicule,int)
  virtual void _avance()
  { Vehicule::_avance();
    //... Lance le moteur
  }
};
```

L'Observateur

Lors de la réception d'une notification, il faut faire attention en manipulant le sujet. Il y a un risque de récurrence. En effet, la notification est synchrone, c'est-à-dire que c'est la méthode du sujet qui appelle directement la call-back de l'observateur. Si celui-ci appelle alors un service du sujet entraînant une nouvelle notification, la call-back est de nouveau appelé. Si le code est rédigé correctement, cela peut fonctionner, mais le problème réside dans l'absence de connaissance des autres observateurs associés au sujet.

Pour éclairer le problème, imaginons la situation suivante. Un sujet et deux observateurs `Obs1` et `Obs2`. Le sujet peut envoyer deux notifications `A` et `B` suivant si le service `a()` ou `b()` est appelé. Maintenant, l'observateur `Obs1` déclare que lors de la réception d'une notification `A` il appelle le service `b()` du sujet. Pas de problème pour le moment, cela fonctionne car la notification `B` pour cet observateur est simplement

ignorée. Ajoutons un nouvel observateur `Obs2` pour le même sujet, mais celui-ci appelle la méthode `a()` lors de la réception d'une notification `B`. Si cet observateur est présent seul, avec le sujet, cela fonctionne. Le problème arrive lorsque les deux observateurs sont présents simultanément pour le même sujet. Dans ce cas de figure, lors de l'appel de la méthode `a()`, le premier sujet détecte cet appel, et appelle la méthode `b()`. Cet appel entraîne une notification vers le deuxième observateur, qui a son tour, appelle la méthode `a()`. Cet appel entraîne, via le premier observateur, l'appel de `b()`. Cette boucle infernale continue tant que la pile du processus le permet. Nous sommes en présence d'une récursivité infinie. Pour éviter ce problème, il faut éviter d'appeler des méthodes du sujet entraînant une notification lors d'une callback. Il faut éventuellement ajouter un mécanisme asynchrone chez l'observateur pour appeler le service voulu du sujet en dehors de la notification. Il faut alors espérer qu'à la longue, le sujet sera stabilisé.

En résumé, la méthode de Call-back devrait recevoir un pointeur `const` sur le sujet afin d'interdire la modification de celui-ci.