

Class Extension

Objectif

Séparer le modèle « métier » de l'interface utilisateur ou de toute utilisation particulière de celui-ci. Ajouter des services polymorphes aux classes du métier.

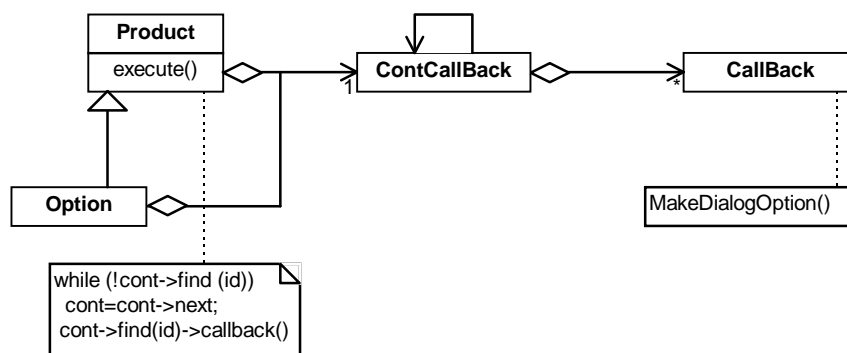
Motivation

L'interface utilisateur d'une classe du métier doit être séparée du modèle afin de garantir une indépendance de celui-ci avec une utilisation particulière. Un modèle peut être présenté par une interface en mode caractère, via des fenêtres graphiques s'exécutant dans différents environnements (X-Windows, Windows, Mac), ou être manipulé par un langage auteur spécifique pour des traitements par lots.

Une interface doit pouvoir lancer un traitement du type : « Affiche la boîte de dialogue permettant de modifier un objet polymorphe ». Lors de l'utilisation du modèle, les objets manipulés ne sont pas forcément identifiés au niveau de leurs types. Par exemple, pour des produits financiers, il est utile d'afficher une boîte de dialogue correspondant au produit manipulé. Chaque produit ayant une boîte différente, deux démarches sont possibles.

- La première consiste à rédiger un `switch` dépendant du type de l'objet et d'appeler la construction de la boîte de dialogue correspondante. Cela entraîne qui faut pouvoir identifier le type d'un objet à l'exécution (ce qui n'est pas toujours possible sans le « RunTime Type Identification »). De plus, toute modification du modèle entraîne la modification du code rédigé pour l'interface utilisateur. Le développement objet n'aime pas les `switchs`. Surtout si ceux-ci dépendent du type d'une instance. Les méthodes virtuelles sont là pour éviter les `switchs`.
- La deuxième technique consiste à ajouter une méthode virtuelle à la classe « produit » s'occupant de créer la boîte de dialogue correspondante pour chaque produit. Cela entraîne une modification du modèle « métier » qui se retrouve à avoir des services fortement liés à l'interface utilisateur. Il n'est alors plus possible de séparer le métier de l'interface.

Pour éviter les deux inconvénients des approches précédentes, il faut offrir une structure permettant d'ajouter des méthodes virtuelles dynamiquement à un modèle. Celui-ci ignore le rôle de ces méthodes. Il n'offre que la possibilité d'enregistrer de nouvelles méthodes et de les appeler.

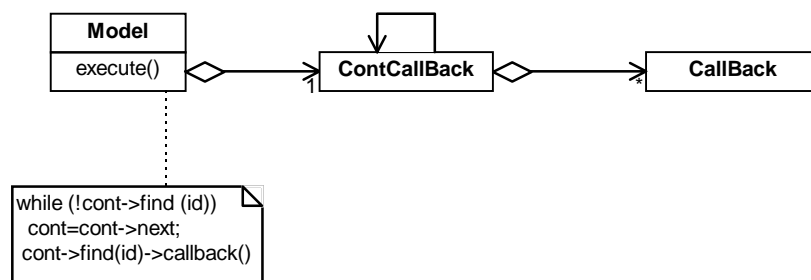


Les conteneurs de Call-Back sont liés entre eux par une relation équivalente au lien d'héritage. L'utilisateur demande l'exécution au modèle d'une méthode identifiée par un id. La Call-Back associée est cherchée dans le conteneur de la classe. Si l'identification de la Call-Back n'est pas trouvée, la recherche s'effectue dans le conteneur de la classe de base.

Applicabilité

Utilisez l'extension de classe lorsque que vous voulez ajouter des comportements polymorphiques à un modèle, sans modifier celui-ci.

Structure



Participants

- **Modèle** (Product)
 Implante un objet du métier sans préjuger de ses utilisations.
- **ContCallBack**
 Contient une liste de Call-backs associé à un identifiant. Il existe une instance de ce conteneur par classe. Ces conteneurs sont reliés entre eux pour décrire les liens d'héritages
- **Callback** (MakeDialogProduct)
 Méthodes virtuelles hors du métier à ajouter à celui-ci. Chaque méthode est enregistrée auprès de la classe, et associée à un identifiant.

Collaborations

- L'utilisateur du modèle demande un identifiant unique pour une nouvelle méthode virtuelle à ajouter au modèle.
- Il enregistre des call-backs C associées à un identifiant auprès des classes du métier.
- Le modèle métier offre une interface normalisée permettant d'appeler les call-backs enregistrées suivant l'identifiant fournit.

Conséquences

1. *Le modèle métier est isolé* de l'interface utilisateur ou de toute utilisation de celui-ci
2. *Plusieurs extensions du métier peuvent cohabiter* dans le même projet.
3. *Des méthodes virtuelles peuvent être ajoutée* dynamiquement dans le métier, supprimant ainsi l'utilisation du pattern «Visitor (331) ».
4. *Le modèle ne doit pas être modifié* ou recompilé lors de l'extension de celui-ci.
5. *L'interface du modèle s'adapte* aux modifications de celui-ci

Implementation

L'implémentation déclare un conteneur d'association entre un identifiant et une call-back C. Il existe un conteneur par classe. Les conteneurs sont reliés entre eux suivant l'héritage de chaque classe.

Une méthode virtuelle est ajoutée à chaque classe pour demander l'exécution d'une Call-back par rapport à un identifiant. Cette méthode commence par le conteneur de sa classe et recherche l'identifiant demandé. Si celui-ci est trouvé, la call-back correspondante est appelée. Sinon, la recherche se poursuit dans les conteneurs des classes de base. Il est donc possible de surcharger une call-back identifiée dans les classes dérivées.

La déclaration de tous les templates nécessaires se trouve dans le fichier [extension.hh](#).

Attention, seuls les traitements documentés ici sont valides. Toute utilisation de méthodes non documentées est au risque et péril de l'utilisateur.

Le mode

Les classes du modèle désirant offrir les fonctionnalités de « Class Extension » doivent ajouter dans le corps de la classe dans la section `public`, la macro `DECLARE_CLASSEXTENSION()`. Cette macro n'attend pas de paramètres. Toutes les classes dérivées doivent ajouter cette macro.

De plus, dans le `.cc` de la classe du modèle, il faut ajouter pour la classe de base la macro `IMPLEMENT_FIRST_CLASSEXTENSION(Type)` ou `Type` est le nom de la classe de base. Une classe dérivée doit utiliser la macro `IMPLEMENT_CLASSEXTENSION(Type,Base)`. `Type` représente toujours la classe courante, et `Base` représente l'héritage immédiat.

```
class A
{ public:
  DECLARE_CLASSEXTENSION()
};

class B : public A
{ public:
  DECLARE_CLASSEXTENSION()
};

class C : public B
{ public:
  DECLARE_CLASSEXTENSION()
};

IMPLEMENT_FIRST_CLASSEXTENSION(A)
IMPLEMENT_CLASSEXTENSION(B,A)
IMPLEMENT_CLASSEXTENSION(C,B)
```

Ce sont les seules modifications à apporter au modèle. Celui-ci offrira alors la possibilité d'ajouter des méthodes virtuelles sans être modifié.

L'enregistrement

Chaque méthode à ajouter au modèle doit être identifiée de façon unique. Pour cela, un allocateur d'identifiant permet d'intégrer plusieurs bibliothèques indépendantes d'utilisation du modèle. La fonction `AllocExtension()` retourne un identifiant de type `TIdExtension` unique. Cet entier servira à identifier les noms des méthodes virtuelles à ajouter au modèle lors de l'enregistrement, et permettra de retrouver la bonne méthode lors de l'exécution.

```
static TIdExtension DialogWindows=AllocExtension();
```

La variable statique `DialogWindows` possède un entier unique permettant d'identifier les méthodes ajoutées au modèle. Cette variable doit rester statique et ne pas être déclarée autrement (Sinon problème d'ordre d'initialisation des variables globale).

Une fois le nom de la méthode créé, il faut enregistrer auprès des classes du modèle les call-backs à appeler lors de la demande d'exécution de cette extension. Pour cela, il faut déclarer un objet global à l'application faisant le lien entre l'identification de la méthode, et la fonction C à appeler.

Les paramètres de la méthode C sont variables. Il n'est pas possible avec le C++ de décrire la procédure d'appel d'une call-back. La fonction C doit donc recevoir un pointeur `void` pour pouvoir communiquer avec l'appelant. Il doit y avoir un protocole entre l'appelant et l'appelé pour remplir une structure contenant tous les paramètres d'appel, et un élément particulier permettant de faire transiter le code retour. Cette procédure n'étant pas simple et source de confusions, j'ai rédigé des templates s'occupant de construire une structure de dialogue entre l'appelant et l'appelé, et éclatant celle-ci pour pouvoir enregistrer une fonction C classique.

Cela entraîne une déclaration particulière lors de l'enregistrement de la fonction C. Il faut utiliser une classe template `CRecordRequestX<>` où `X` indique le nombre de paramètres attendu par la fonction C. Ce nombre doit être compris entre zéro et trois inclus. Si vous désirez enregistrer une méthode demandant plus de paramètres, demandez-moi d'ajouter le code nécessaire. Une instance du template doit être créée globalement pour chaque méthode enregistrée. Les signatures de ces templates sont les suivantes :

Pas de paramètres

```
CRecordRequest0<Model,TRet>(TIdExtension id,call-backC)
```

Cette classe enregistre la méthode `id` pour le `Model`, une fonction `call-backC`, ne recevant aucun paramètre et retournant un objet du type `TRet`.

```
int f();
static CRecordRequest0<CProduct,int> record_f(DialogWindows,f);
```

Un paramètre

```
CRecordRequest1<Model,TRet,TPara1>(TIdExtension id,call-backC)
```

Cette classe enregistre la fonction call-back `call-backC`, avec comme identifiant de méthode `id` pour la classe du modèle `Model`. La fonction `call-backC` ne reçoit qu'un paramètre de type `TPara1` et retourne un objet du type `TRet`.

```
int f(CWnd*);
static CRecordRequest1<CProduct,int,CWnd*> record_f(DialogWindows,f);
```

Deux paramètres

`CRecordRequest2<Model,TRet,TPara1,TPara2>(TIdExtension id,call-backC)`
 Cette classe enregistre la fonction call-back `call-backC` avec, comme identifiant de méthode, `id` pour la classe du model `Model`. La fonction `call-backC` reçoit deux paramètres de type `TPara1` et `TPara2` et retourne un objet du type `TRet`.

```
int f(CWnd*,int);
static CRecordRequest2<CProduct,int,CWnd*,int> record_f(DialogWindows,f);
```

Trois paramètres

`CRecordRequest3<Model,TRet,TPara1,TPara2,TPara3>(TIdExtension id,call-backC)`

Cette classe enregistre la fonction call-back `call-backC`, avec comme identifiant de méthode `id` pour la classe du model `Model`. La fonction `call-backC` reçoit trois paramètres de type `TPara1`, `TPara2` et `TPara3` et retourne un objet du type `TRet`.

```
int f(CWnd*,int,long);
static CRecordRequest3<CProduct,int,CWnd*,int,long>
record_f(DialogWindows,f);
```

La déclaration des objets `CRecordRequestX<>` doit être faite dans le même fichier source ou se trouve l'appel de `AllocExtension()`.

L'utilisation

Pour appeler la call-back enregistrée dans le modèle, il faut appeler la fonction template `classExtension()`. Cette fonction a une signature très particulière. En effet, elle accepte de trois à six paramètres. Seul le deuxième paramètre possède un type obligatoire. Tous les autres peuvent être de n'importe quel type.

```
classExtension(Model& m,TIdExtension id,TRet*[,TPara1 [,TPara2
[,TPara3]]]);
```

Le premier paramètre doit être l'objet du modèle polymorphe dont on désire appeler une méthode additionnelle.

Le deuxième paramètre est l'identifiant de la méthode ajoutée. Cet identifiant doit avoir été obtenu par l'initialisation d'un variable statique appelant `AllocExtention()`.

Le troisième paramètre doit être un pointeur sur le type de retour de la fonction C enregistré.

Les trois paramètres optionnels suivants sont les paramètres attendus par la méthode C enregistrée.

Il faut absolument encapsuler l'appel de cette fonction dans une autre fonction C. En effet, les types des paramètres ne sont pas vérifiés par le compilateur lors de l'appel de `classExtension()`. L'API le vérifie à l'exécution et indique une erreur par une assertion si l'appel de `classExtension` est erroné. Malheureusement, le polymorphisme ne peut pas être correctement détecté (pour des explications complémentaires, me demander). L'encapsulation de cet appel permet de régler cela.

```
CWnd* DlgProduct(CProduct& p,CWnd* father)
{ CWnd* ret;
```

```

    classExtension(p,DialogWindows,&ret,father);
    return ret;
}

```

Il faut toujours encapsuler l'appel à une méthode `classExtension`. De plus, cela permet d'avoir un code retour traditionnel.

Les erreurs

- La surcharge n'est pas gérée par cet Api. Il n'est pas possible d'enregistrer deux fonctions C différent avec le même identifiant pour la même classe.
- La demande d'exécution d'un identifiant non enregistré génère une assertion à l'exécution.
- L'exécution d'une méthode enregistrée doit avoir strictement la même signature d'appel que les fonctions C associées.

Exemple de Code

À la place d'une modification du modèle comme ceci :

```

class CProduct
{ //...
    CWnd* Dlg(CWnd* father)=0;
};
class COption : public CProduct
{ //...
    CWnd* Dlg(CWnd* father)
    { // Make Dlg for Option...
        return rc;
    }
};

//...
void CWndApplication::buttonEdit()
{ product.Dlg(this); }

```

Il faut rédiger le code comme ceci :

Fichier product.hh

```

class CProduct
{ //...
    DECLARE_CLASSEXTENSION()
};
class COption : public CProduct
{ // ...
    DECLARE_CLASSEXTENSION()
};

```

Fichier product.cc

```

IMPLEMENT_FIRST_CLASSEXTENSION(CProduct)
IMPLEMENT_CLASSEXTENSION(COption,CProduct)

```

Fichier wininit.cc

```

static TIdExtension DialogWindows=AllocExtension();
CWnd* DlgOption(CWnd* father)
{ // Make Dlg for Option...
    return rc;
}
static CRecordRequest1<CProduct,CWnd*,CWnd*>
record_DlgOption(DialogWindows,DlgOption);

```

Fichier winproduct.cc

```
CWnd* ProductDlg(CProduct& m,CWnd* father)
{ CWnd* rc;
  classExtension(m,DialogWindows,&rc,father);
  return rc;
}
//...
void CWndApplication::buttonEdit()
{ ProductDlg(product,this); }
```

Utilisations connues

Dans le projet Fox action, l'édition des jeux de paramètres ou des produits utiliseront ce pattern.

Patterns en relation

- Visitor (331) est une autre technique pour ajouter des comportements à un modèle sans le modifier. Par contre, ce pattern oblige à recompiler tous le modèle lors de l'ajout d'une classe. L'évolution de celui-ci n'est plus incrémentale mais global.
- L'Observer (293) est un pattern régulièrement utilisé avec Class Extension. Pour ajouter une interface à un modèle sans modifier celui-ci, le couple « Class Extension » et « Observer (293) » permet de régler tous les désirs d'extensions. Lorsque le modèle est modifié, l'interface est prévenue par l'Observer (293). Pour appeler des traitements spécifiques à l'interface, mais dépendant du polymorphisme du modèle, « Class Extension » est utilisé.