

Des objets à options

Philippe PRADOS

pp@philippe.prados.name



*Préservez l'environnement,
n'imprimez pas ce document*

TABLE DES MATIERES

1.	Le problème.....	3
2.	Les solutions possibles	3
2.1	Héritage	3
2.2	Propriété	5
2.3	Classe incluse.....	7
2.4	Modifications à la volée.....	8
2.5	Avant/Après.....	9
2.6	Observateur.....	9
3.	Paramétrer la construction pour ajouter les options	9
3.1	Méthode fabricante.....	9
3.2	Paramétrage	10
3.3	Prototype	10
3.4	Construction si utilisation.....	10
3.5	Construction de toutes les options avant l'utilisation du métier	11
4.	Proposition de solutions	11
4.1	Capturer l'accès aux objets métier	11
4.2	Options de compilations.....	11
4.3	Options à la place de l'héritage	12
4.4	Subject-oriented programming (SOP)	13
5.	Conclusion	13
6.	Références.....	Error! Bookmark not defined.

Avant propos

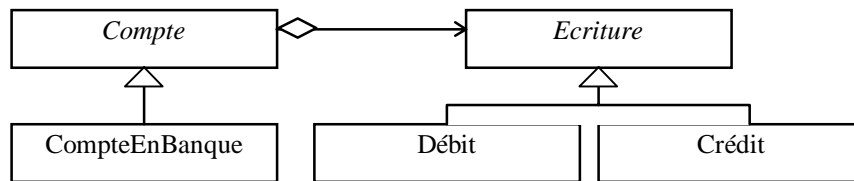
Ce document se propose d'étudier les différentes solutions pour offrir des options d'utilisation à des objets métiers. Comment utiliser un objet métier avec des fenêtres ? Avec une persistance dans une base objets ? Avec une base relationnelle ?

1. LE PROBLEME

Le modèle objet facilite la réutilisation. Il est possible d'utiliser l'héritage et le polymorphisme pour spécialiser un objet suivant un besoin particulier.

Les concepteurs objets conseillent à juste titre, qu'il existe, dans les programmes, une séparation entre les objets métiers et les objets d'interfaces. Cela permet d'extraire un métier sans devoir récupérer une interface gênante. Tout cela est bien joli, mais, à l'usage, ce n'est pas toujours si facile. Étendre un arbre d'héritage pour enrichir un métier ou une interface ne pose pas de problème. Par contre, étendre un métier pour un ou plusieurs usages spécifiques, n'est pas évident.

En effet, il est souvent nécessaire d'ajouter des attributs et des méthodes particulières à un métier correspondant à un usage. Pour éclairer le discours, nous allons prendre un exemple d'objets métiers. Imaginons une classe `CompteEnBanque` héritant de la classe `Compte`. Un compte en banque possède en agrégations des écritures. La classe `Ecriture` est la racine de plusieurs autres classes permettant de spécialiser le type comptable.



Des services permettent de créer, d'ajouter, de consulter ou de supprimer une écriture. Nous voulons utiliser ces objets métiers dans plusieurs environnements différents. C'est un des objectifs de la programmation objet.

- 1^{er} option : Nous désirons sauver ces objets métiers dans une base de données.
- 2^{ème} option : Nous désirons afficher une fenêtre pour le compte en banque, et mémoriser la coordonnée de celle-ci. Nous désirons également une fenêtre ou une boîte de dialogue pour les différents types d'écritures comptables.

Nous voulons pouvoir utiliser toutes les combinaisons possibles :

- Les objets métiers seuls.
- les objets métiers + sauver les objets dans une base de données
- les objets métiers + l'interface utilisateur,
- les objets métiers + sauver les objets dans une base de données + l'interface utilisateur

Nous appellerons les instances correspondant à des options du métier : « des instances options ». Une instance s'occupant d'afficher un métier est une *instance option*. De même, une instance permettant de sauver un objet métier est une *instance option*. Nous aurions pu choisir une option permettant d'utiliser l'objet métier dans une architecture client serveur ou pour l'interfacer avec un langage auteur, la démarche serait la même. Le choix de l'option de présentation permet de faciliter le discours. Comment adapter les objets métiers pour les utiliser dans des contextes très différents ?

2. LES SOLUTIONS POSSIBLES

Il y a plusieurs solutions possibles. Chacune apporte une partie de la solution.

2.1 Héritage

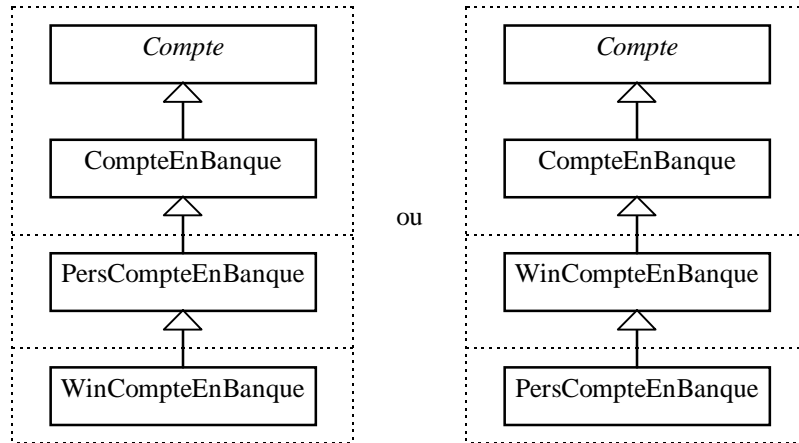
2.1.1 Descriptions

Pour répondre à la première option, nous pouvons déclarer une classe `PersCompteEnBanque` héritant de la classe `CompteEnBanque`. Celle-ci ajoute les services de lecture et d'écriture dans la base de données afin de gérer la persistance.

De même, pour répondre à la deuxième option, nous allons rédiger une classe héritant de `WinCompteEnBanque`. Celle-ci ajoute les paramètres permettant d'afficher un `Compte` et de mémoriser les coordonnées de la fenêtre. La classe `WinCompteEnBanque` est bien séparée de la classe `CompteEnBanque`. Cela respecte le principe de la séparation du métier et de l'interface.

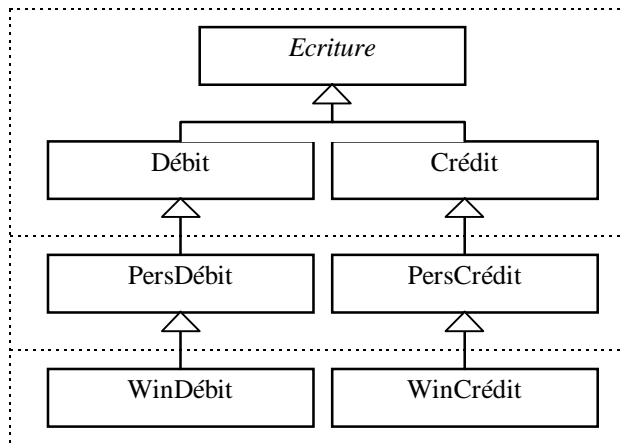
Il aurait été possible de rédiger ces classes dans le sens inverse : d'abord la présentation, puis la persistance (Figure 1).

Figure 1



Nous procédons de même pour les classes dérivées de *Ecriture* (Figure 2).

Figure 2



Possédant une référence sur une écriture, il est possible de demander l'affichage de la fenêtre correspondante. Cette approche permet de bénéficier du polymorphisme. Il n'est pas nécessaire de connaître le type exact des écritures pour afficher sa fenêtre. L'objet s'en charge.

2.1.2 Avantages

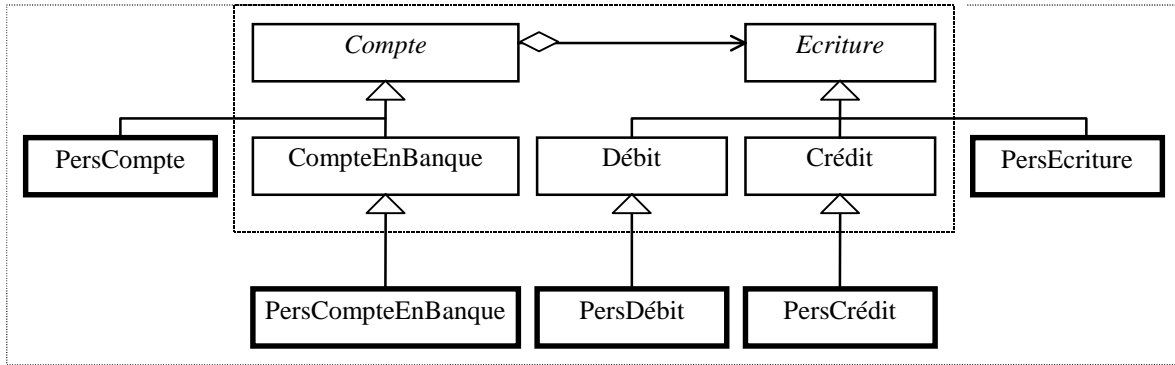
L'application utilise le niveau d'héritage correspondant à son besoin. Il est facile de n'utiliser que la classe métier *CompteEnBanque*, sans utiliser la version utilisant les fenêtres par exemple.

2.1.3 Inconvénient

Si l'on désire utiliser la version de la classe *WinCompteEnBanque* possédant la persistance, il faut alors également prendre la classe *PersCompteEnBanque*. Il n'est pas possible de sélectionner les options nécessaires à une nouvelle application. Les options héritant les unes des autres, il n'est pas possible de les choisir suivant le contexte d'utilisation. Il faut choisir l'option du plus haut niveau d'héritage permettant d'englober toutes les autres. Cela apporte des classes inutiles qui parfois, interdisent la réutilisation.

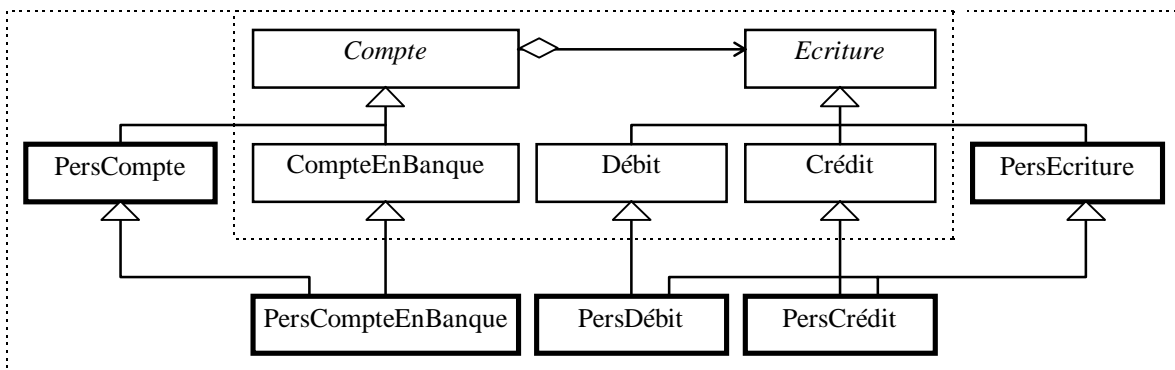
Il est impossible d'ajouter des services pour toutes les classes du métier sans détériorer la souplesse de l'héritage. Une classe d'option ne pourra pas facilement être héritée par une option plus spécialisée. Par exemple, si l'on désire ajouter une classe persistance pour toutes les classes du métier, cela donne l'arbre d'héritage de la Figure 3.

Figure 3



La classe de persistance `PersCompteEnBanque` ne bénéficie pas de la classe `PersCompte`. `PersCompteEnBanque` n'hérite pas de `PersCompte`. Pour palier à cela, il faut utiliser l'héritage virtuel et l'héritage multiple. Cela donne la Figure 4.

Figure 4



`PersCompte` et `CompteEnBanque` doivent hériter virtuellement de `Compte` pour n'avoir qu'une seule instance de `Compte` dans la classe `PersCompteEnBanque`. De même, `Débit`, `Crédit` et `PersEcriture` doivent hériter virtuellement de `Ecriture`.

Si une classe métier construit elle-même un autre objet métier, celui-ci ne sera pas étendu par les fonctionnalités de l'héritage. Par exemple, la méthode `creerEcriture` de la classe `CompteEnBanque` ne peut pas construire une classe `PersDebit` à la place de `Débit` sans modification du métier. Le chapitre "Paramétrer la construction pour ajouter les options" propose différentes solutions pour régler ce problème.

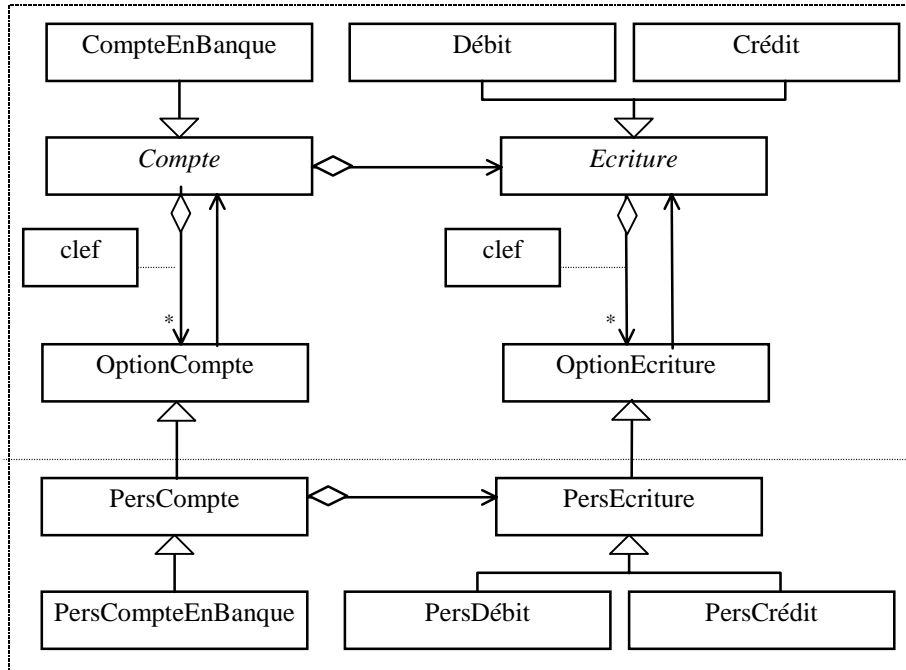
2.2 Propriété

2.2.1 Descriptions

Pour pouvoir bénéficier de l'héritage entre les classes d'interface utilisateurs, sans utiliser l'héritage multiple, il peut être intéressant de sortir cet arbre des classes métiers et d'utiliser la délégation.

Un arbre d'héritage est construit pour chaque option du métier (un arbre pour la persistance et un arbre pour la présentation). Ceux-ci sont reliés au métier à l'aide d'un attribut particulier, appelé propriété (Figure 5).

Figure 5

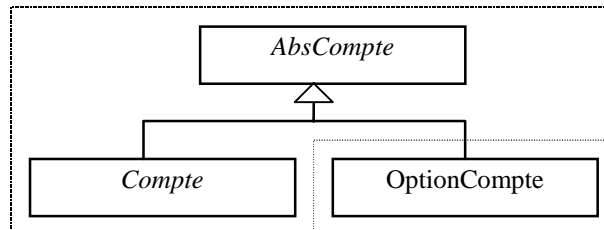


Un service du métier identifie chaque option par une clef. Celle-ci permet de retrouver l'instance de l'option correspondant au métier. Les services de l'option peuvent déléguer les traitements au métier. Un service demandé à `PersCompte` sera éventuellement délégué à l'instance `Compte` correspondante.

Cette approche est très similaire au pattern *Bridge* (cf. Design Pattern). Contrairement à celui-ci, il permet d'avoir simultanément plusieurs options. Une instance `Compte` peut posséder simultanément une instance `PersCompte` et une instance `WinCompte`. Ces options sont accessibles par des clefs.

Avec les langages fortement typés (C++ et Java), il peut être nécessaire de déclarer une classe abstraite héritée par `Compte` et `OptionCompte` (Figure 6).

Figure 6



Cela permet de garantir la compatibilité d'interface entre `Compte` et `OptionCompte` et permet de bénéficier du polymorphisme. Une classe `OptionCompte` est une sorte de classe `AbsCompte`. Le polymorphisme s'effectue à l'aide d'instances de type `AbsCompte`.

Les C++ utilise un protocole similaire pour permettre l'addition d'attribut aux objets de flux. La méthode `ios::xalloc()` permet de réserver une place mémoire dans le flux. Elle retourne un entier servant à identifier un entier ou un pointeur ajouté dans le flux. Les méthodes `ios::iword()` et `ios::pword()` permettent de manipuler ces attributs supplémentaires. L'initialisation de ces attributs n'est pas gérée. Ils ont une valeur aléatoire tant qu'ils ne sont pas valorisés.

2.2.2 Avantages

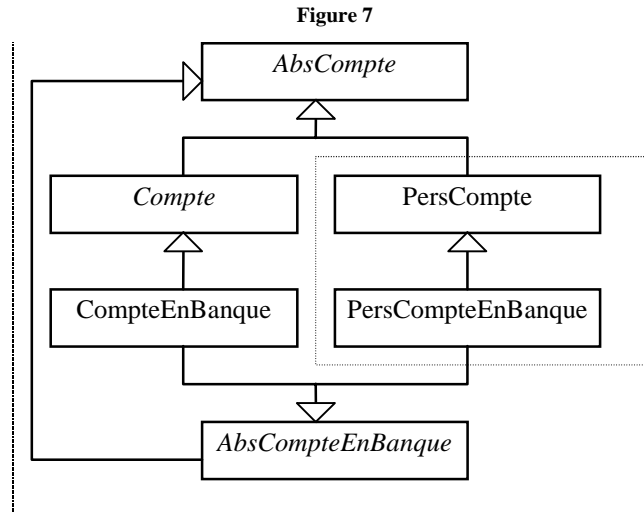
Il y a une nette séparation entre le métier et les options de celui-ci.

Il n'est pas nécessaire d'apporter des classes inutiles lors de l'utilisation d'une option. On peut choisir indifféremment d'utiliser le métier, le métier et la persistance, le métier et la présentation ou le métier, la persistance et la présentation.

2.2.3 Inconvénient

Les accès aux attributs et aux méthodes des options sont complexes. Il faut rechercher l'option du métier puis accéder aux attributs.

Si l'on désire utiliser des classes abstraites pour garantir le polymorphisme à l'aide d'un langage fortement typé, le modèle objet se complique à chaque niveau d'héritage (Figure 7).



L'utilisation de l'héritage multiple devient obligatoire, et il faut enrichir cela d'héritages virtuels !

Cela ne règle pas le problème de la construction des options. Si une classe métier construit elle-même un autre objet métier, celui-ci ne sera pas étendu par les fonctionnalités de l'héritage. Le chapitre "Paramétrer la construction pour ajouter les options" propose différentes solutions pour régler ce problème.

2.3 Classe include

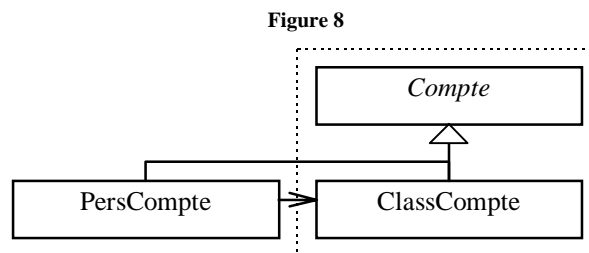
La version 1.1 de Java propose un nouveau type d'héritage. Une classe peut être déclarée à l'intérieur d'une autre. La particularité des classes incluses est qu'elles gardent un lien avec l'instance qui les a générées. Il est ainsi possible d'utiliser les attributs de cette instance, comme s'il s'agissait d'attribut de la classe interne.

```

// Code Java
class MainClass
{ int mainAttr;

    public class InnerClass
    { public int innerAttr;
      void innerMethode()
      { innerAttr=0;
        mainAttr=0; // Cela fonctionne !
      }
    }
}
  
```

La classe `InnerClass` peut être considérée comme une option de la classe `MainClass`. Le corps de l'option est facile à rédiger car elle possède un lien avec la classe métier. Par contre, il faut organiser les classes pour leurs faire partager la même interface. Cela permet de voir l'option comme une classe métier. Les méthodes de l'option devront utiliser la délégation pour s'exécuter sur l'instance du métier correspondant (Figure 8).



```

interface Compte
{ public float solde();
}
class ClassCompte implements Compte
{
    public float solde()
    { //...
    }

    // L'option
    class PersCompte implements Compte
    { public float solde()
      { return ClassCompte.this.solde();
      }
    }
}
  
```

```

    }
    void save()
    { //...
    }
    void load()
    { //...
    }
}

```

2.3.1 Avantages

Cette approche est sémantiquement intéressante. Elle permet de décrire les options sans se soucier des artifices techniques faisant le lien avec l'instance métier.

Les accès aux attributs et aux méthodes du métier sont simples dans le corps de l'option.

Il y a une nette séparation entre le métier et les options de celui-ci.

2.3.2 Inconvénient

Toutes les options doivent être codées dans la classe métier. Il n'est pas possible d'ajouter une nouvelle option.

Il n'est pas possible d'accéder directement aux attributs du métier à partir d'une référence sur une option.

Il n'est pas possible de retrouver une option à partir d'une instance du métier (conversion inverse).

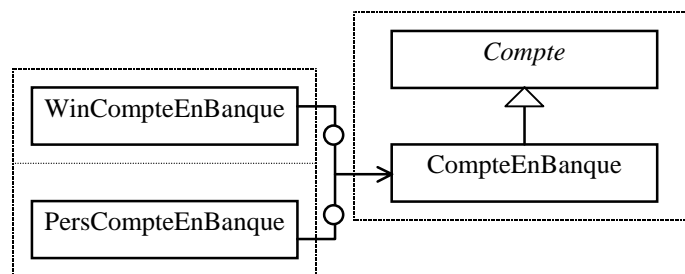
Il reste encore à régler le problème de la construction des instances options. Celui-ci est étudié au chapitre "Paramétrer la construction pour ajouter les options".

2.4 Modifications à la volée

2.4.1 Descriptions

Certains langages permettent de modifier dynamiquement les attributs et les méthodes d'une instance (Smalltalk ou CLOS). Dans ce cas, une instance du métier peut être enrichie dynamiquement d'options. Les attributs et les méthodes additionnelles d'un objet métier sont ajoutés pour correspondre à l'usage qui en est fait (Figure 9).

Figure 9



2.4.2 Avantages

Il n'est pas nécessaire d'avoir une hiérarchie de classe en parallèle au métier.

Les accès aux attributs des options sont directs. Il ne faut pas ajouter des tuyauteries compliquées pour accéder à tous les attributs et toutes les méthodes des options.

2.4.3 Inconvénient

Il faut organiser l'addition des attributs dans un concept général pour pouvoir extraire une option du programme. Il faut pouvoir n'extraire que les ajouts concernant la persistance ou que ce concernant la présentation.

Cela ne règle pas le problème de la construction d'un objet métier par un autre objet métier.

Pour ne pas modifier la construction des objets métiers, il faut ajouter ces attributs lors de chaque utilisation s'ils ne sont pas présents. Par exemple, si un attribut d'une option n'est pas présent lors de l'utilisation d'un objet métier, l'appelant doit ajouter cette option.

Possédant une référence sur un objet `Compte` du métier, je désire accéder à l'attribut `window` de l'option présentation. S'il n'est pas présent, cela veut dire que l'instance `Compte` référencé, ne possède pas encore cette option. Je l'ajoute alors, et je recommence à demander l'attribut.

Il y a perte de la notion de classe. Chaque instance peut être différente de sa voisine. Cela abîme le modèle objet traditionnel. On se retrouve avec un langage proche de Javascript ou de Self. Il y a perte de la structuration du programme en classe. Chaque objet représente un dictionnaire de service.

2.5 Avant/Après

2.5.1 Descriptions

Certains langages (CLOS par exemple) permettent d'ajouter des comportements avant, pendant ou après une méthode d'un métier. Cela permet d'enrichir celui-ci sans le modifier.

2.5.2 Avantages

L'ajout de comportement est facile. Cela est un moyen d'enrichir un métier sans le modifier.

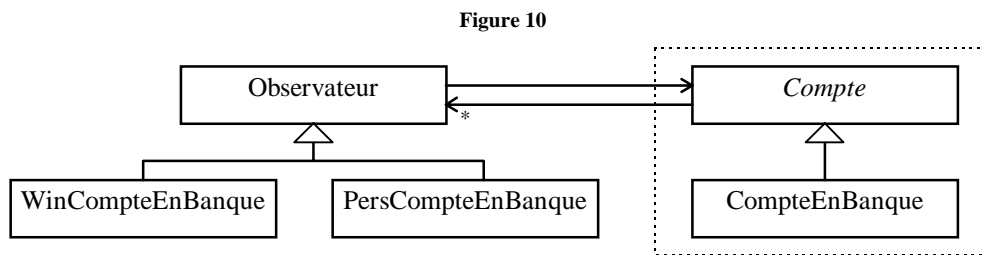
2.5.3 Inconvénient

Cela ne règle pas le problème de la construction des instances options pour le métier. Sans modification du métier, il est difficile de modifier la construction d'un objet au sein d'une méthode.

2.6 Observateur

2.6.1 Descriptions

Pour connaître les modifications d'un objet métier, on peut rédiger une classe étant à l'écoute de celui-ci. Lorsque le métier évolue, il prévient les observateurs. Ceux-ci peuvent alors s'adapter aux évolutions (Figure 10).



Les instances `WinCompteEnBanque` et `PersCompteEnBanque` sont prévenues de toutes les modifications de l'instance `CompteEnBanque`.

2.6.2 Avantages

Le métier est parfaitement séparé de ses options.

2.6.3 Inconvénient

Suivant les langages, il faut que le métier informe de toutes ses modifications avec une sémantique forte. Il doit décrire précisément les modifications qu'il a subit.

Il n'est pas possible de notifier un observateur lors de la création d'une instance du métier. En effet, lors de la création d'une instance, l'observateur ne connaît pas encore son existence. Il ne peut pas s'y enregistrer.

Il n'est pas possible de modifier une méthode du métier pour tenir compte d'une option particulière. L'observateur peut réagir à des modifications du métier, mais ne peut pas modifier le comportement de celui-ci.

Cela ne règle pas le problème de la construction des instances options pour le métier. Ce point est éclairé ci-dessous.

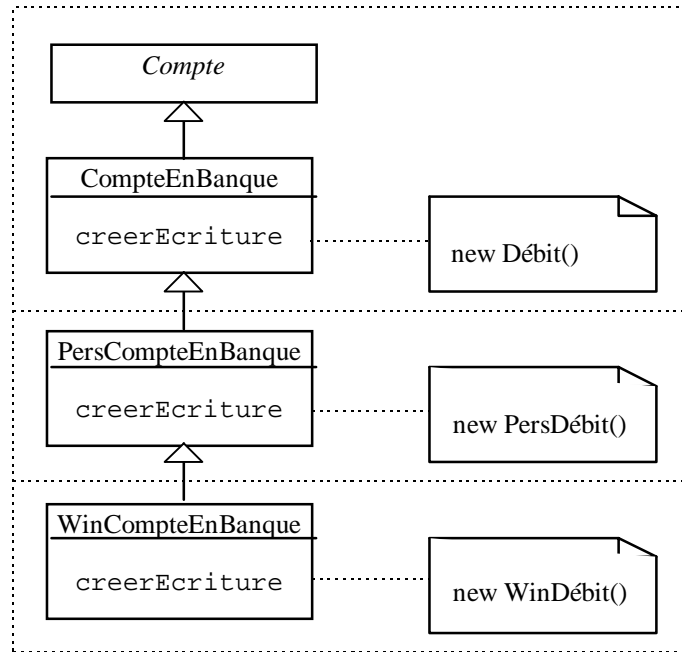
3. PARAMETRER LA CONSTRUCTION POUR AJOUTER LES OPTIONS

Les différentes propositions ci-dessus organisent les instances d'options par rapport aux instances du métier. Plusieurs approches sont envisagées, mais toutes sont insuffisantes pour permettre à un objet métier de construire dans une méthode un autre objet métier ayant les options nécessaires.

3.1 Méthode fabricante

Le pattern "factory method" propose une solution. Si l'on organise toutes les constructions d'objet métier au sein d'un métier dans des méthodes distinctes, il est possible de les redéfinir lors d'un héritage (Figure 11).

Figure 11



Il faut organiser la classe `CompteEnBanque` pour que celle-ci possède une méthode protégée pour chaque construction d'objet. Lorsque le métier construit un autre objet, cela doit se faire dans une méthode particulière pouvant être redéfini dans une classe dérivée. Par exemple, la méthode `creerEcriture` peut être redéfinie par les classes dérivées de `CompteEnBanque`.

Cette technique fonctionne pour l'approche *héritage* des options mais est incompatible avec les approches *propriété*, *Modification à la volée* et *Observateur*.

3.2 Paramétrage

Il est possible de paramétrer la construction des objets métiers pour que ceux-ci construisent automatiquement les options nécessaires à un contexte particulier d'exécution. Un mécanisme d'enregistrement permet de paramétrer une construction afin que celle-ci appelle une succession de méthodes pour construire toutes les options.

Cela peut se faire par un enregistrement des options dans la classe. Avant toute exécution du métier, le programme initialise les options qu'il désire auprès des classes du métier. Ensuite, il peut construire des instances de celui-ci, cela fabriquera automatiquement les options correspondantes.

Il est également possible de construire un objet global ayant en charge la construction de tous les objets métier. Cet objet sera paramétré suivant les options désirées pour l'application. Le pattern "template method" propose cette solution. Malheureusement, cela viole le principe "open/close" qui garantit l'évolution. Il n'est pas possible de greffer une nouvelle option au programme sans modifier cet objet global. L'extension du programme s'effectue en *modifiant* celui-ci, et non en *étendant*. Le programme n'est pas évolutif.

Cette technique est incompatible avec l'approche *héritage* des options. Elle fonctionne pour l'approche *Propriété*, *Modifications à la volée* et *Observateur*.

3.3 Prototype

Le pattern "prototype" propose de construire tous les objets à l'aide d'un modèle. Celui-ci est paramétré suivant les options nécessaires au programme. Un protocole permet d'identifier les modèles d'objets métiers (nom de variable globale, dictionnaire,...). Lorsqu'un métier construit une instance, il demande une duplication du modèle.

Le choix des options nécessaire à une application s'effectue en paramétrant les modèles. L'inconvénient de cette approche est qu'il faut construire les modèles à la main pour l'utilisation des options. Une librairie ne peut pas être intégrée sans la présence d'un fichier source complexe, unifiant toutes les options possibles. Cette approche n'est pas compatible avec le principe "open/close".

Cette technique est incompatible avec l'approche *héritage* des options. Elle fonctionne pour l'approche *Propriété*, *Modifications à la volée* et *Observateur*.

3.4 Construction si utilisation

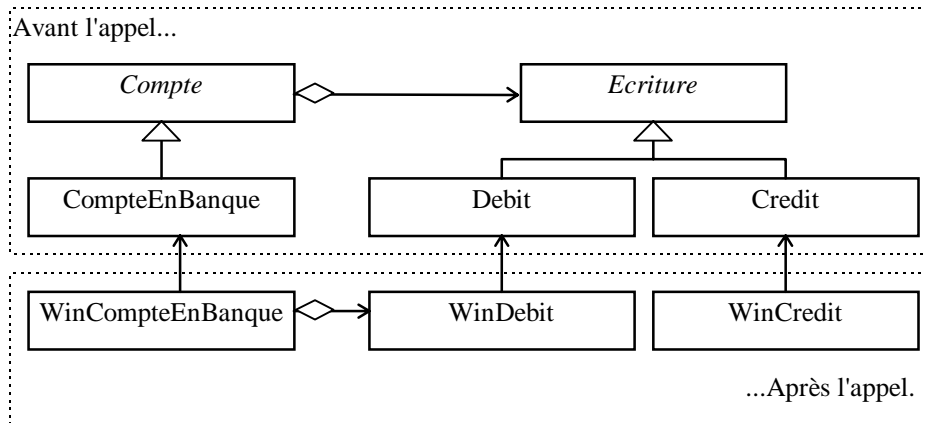
Il est également possible de construire les instances des options si et seulement si un objet métier n'en possède pas lors de l'utilisation. Par exemple, ayant accès à un objet `Compte`, un service demande l'option `Win` de celui-ci. En cas d'absence, le programme la rajoute au métier pour les usages ultérieurs. Cela complique beaucoup le programme. Pour chaque appel à une option il faut se préparer à la construire si nécessaire.

3.5 Construction de toutes les options avant l'utilisation du métier

On peut construire toutes les instances des options avant d'utiliser le métier. Une méthode parcourt l'arbre d'instance du métier et construit toutes les options correspondantes. Une fois cette méthode terminée, il suffit de manipuler les instances des options pour avoir accès aux instances du métier.

Cette approche fonctionne si, en cours d'exécution, il n'y a pas de nouvelles instances du métier de construites. Il y a un fort risque de dissonance entre l'arbre des options et l'arbre du métier. Le métier doit pouvoir évoluer sans problème. Cette approche n'est pas sûre à la longue. Rien, en effet, ne garantit qu'un arbre d'instance du métier ne bougera pas lors de l'appel d'une méthode du métier (Figure 12).

Figure 12



4. PROPOSITION DE SOLUTIONS

4.1 Capturer l'accès aux objets métier

4.1.1 Description

Des techniques permettent de capturer tous les accès aux objets du métier. Les objets sont créés dans un espace mémoire particulier. Lorsque le programme désire accéder à un objet du métier, le microprocesseur génère une exception pour tentative d'accès à une zone mémoire invalide. Cela est capturé par une couche logicielle spéciale qui peut ainsi détecter automatiquement les évolutions du métier. Des produits comme ObjectStore™ utilisent cette technologie.

4.1.2 Avantages

Cela permet d'ajouter la persistance à des objets métiers sans que celui-ci soit préoccupé. Cela peut également permettre l'ajout de la distribution à un objet métier.

4.1.3 Inconvénient

Il est difficile d'ajouter des attributs à un métier ou de modifier un traitement.

C'est une technique complexe à développer. Il est difficile d'envisager l'intégration de plusieurs options simultanées sur le même métier. Les frameworks proposant cela sont faciles à utiliser, par contre, il est difficile de rédiger un framework équivalent.

Les liaisons entre les métiers et les options ne sont pas apparentes. Les options sont développées avec une forte dépendance vis à vis du métier. Si celui-ci évolue, les options sont à revoir.

4.2 Options de compilations

4.2.1 Descriptions

Nous cherchons à avoir des objets à options. Nous désirons un objet métier ayant des capacités de persistance et/ou des capacités de présentation. Pourquoi ne pas offrir explicitement la notion d'option ? Toutes les méthodes et les attributs d'un objet possèdent alors un adjectif identifiant l'option qui les concerne. Lors de l'utilisation d'un objet métier, le développeur indique explicitement au compilateur les options qu'il désire. Celui-ci sélectionne les informations pertinentes, et propose un objet métier sur mesure.

4.2.2 Avantages

Seules les informations pertinentes à un usage particulier sont présentes. "On ne paye que pour ce qu'on utilise."

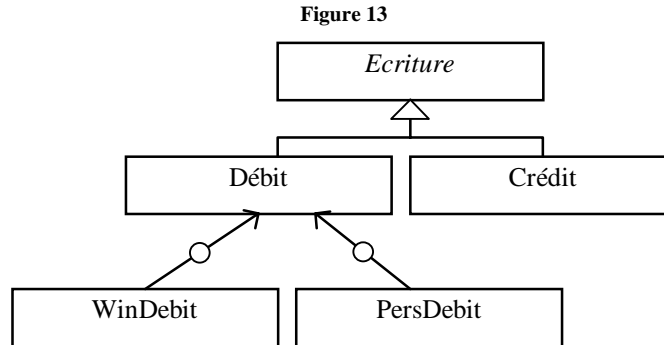
4.2.3 Inconvénient

Il faut modifier directement le fichier source original pour lui apporter les modifications nécessaires pour une option. Une option peut désirer un attribut utile à une autre option. Certains langages permettent d'annoter une classe pour intégrer cette notion d'option. Avec Smalltalk par exemple, il est envisageable d'annoter les attributs et les méthodes d'un métier.

4.3 Options à la place de l'héritage

4.3.1 Descriptions

On peut également imaginer une syntaxe permettant d'ajouter des options à un objet sans intervenir sur le source original. La syntaxe permet de déclarer un additif à une classe. L'additif est toujours présent lors de la création d'une instance, alors qu'un héritage dépend d'un choix lors de la construction. La syntaxe pourrait être strictement la même que pour l'héritage, mais avec un autre mot-clef signifiant qu'il s'agit d'une option (Figure 13).



L'utilisateur de ce modèle peut construire une instance `Debit` ou une instance `Credit`. Il ne peut pas créer d'instance `WinDebit` ou `PersDebit`. Par contre, il peut les utiliser. S'il le fait quelque part dans le programme, cela entraîne que lors de la création de toutes les instances `Debit`, les options `WinDebit` et `PersDebit` seront présentes.

4.3.2 Avantages

Seules les informations pertinentes à un usage particulier sont présentes.

Le langage peut continuer à être fortement typé. L'utilisateur déclare quelle option il utilise pour un objet. Le compilateur se charge de vérifier cela lors de la phase de lien.

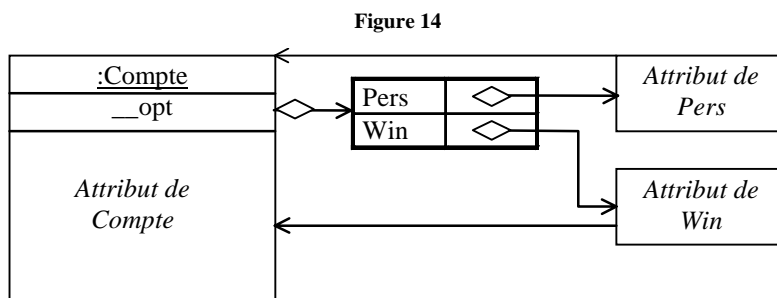
4.3.3 Inconvénient

Il est difficile d'envisager le passage de paramètre lors de la construction d'une option. Comment fournir un paramètre à un constructeur pour que celui-ci le redonne à l'option ? Le constructeur de l'objet ne doit pas être modifié par la présence d'une option. Cela entraîne que les constructeurs des options ne doivent pas recevoir de paramètre. Seul les constructeurs par défaut sont valides pour les options.

Le compilateur se complique. Il doit connaître l'ensemble des options nécessaires au programme avant de pouvoir générer le code. On peut imaginer deux versions de compilation : une en phase de développement sacrifiant la vitesse d'exécution au bénéfice de la vitesse de compilation, et une autre optimisant au mieux le code généré.

Un modèle mémoire pour le C++ pourrait être organisé comme ceci :

Un pointeur caché est ajouté à chaque objet. Celui-ci pointe sur un dictionnaire. Cela permet de retrouver les attributs appartenant aux options (Figure 14).



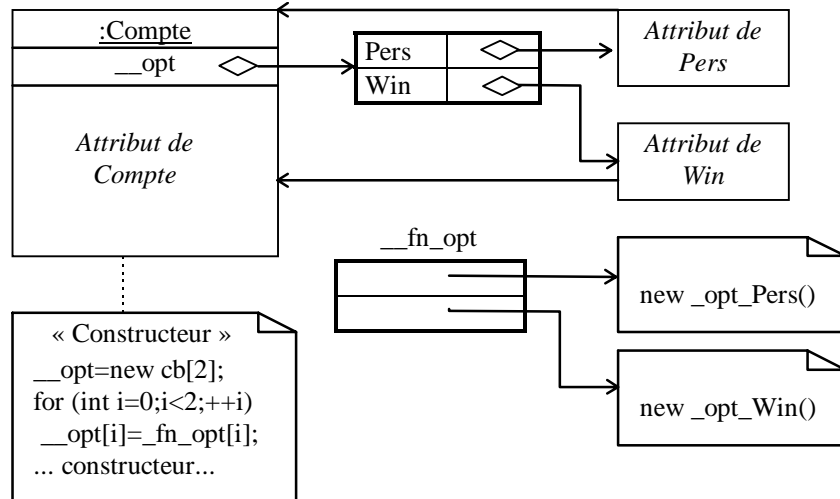
Une méthode ayant un nom particulier est générée automatiquement dans le fichier compilé. Celle-ci s'occupe de construire l'option pour une instance. Elle pourrait avoir la forme suivante :

```
void* __ctr_opt_Pers()
{ return new _opt_Pers(); }
```

Le programme de liaison (`link`) détecte ces fonctions pour les réunir dans un tableau permettant de regrouper toutes les options d'une classe. Le mécanisme est le même que pour initialiser les objets globaux des programmes C++.

Ensuite, le constructeur de l'objet `Compte` est modifié pour appeler successivement les fonctions présentes dans le tableau de fonction créée par le `link`. Au final, lors de l'exécution, la mémoire est organisée comme Figure 15.

Figure 15



Le code du constructeur n'est pas tout à fait exact car il faut normalement construire le dictionnaire avec sa clef, mais le principe est là.

Lors de la conversion d'une référence sur un objet du métier vers une référence sur une de ces options, le code généré pourrait être celui-là :

```
PersCompteEnBanque* OptionPersCpt=(PersCompteEnBanque*)cpt;
```

sera compilé en :

```
PersCompteEnBanque* OptionPersCpt=cpt->__opt.find("Pers");
```

Et lors de l'utilisation d'un attribut du métier à partir d'une option, cela donne :

```
OptionPersCpt->solde=100;
```

compilé en :

```
OptionPersCpt->_metier->solde=100;
```

Cet accès aux attributs est similaire aux classes internes de Java.

Il faut envisager un mécanisme similaire pour permettre la surcharge dans une option.

Une version optimisée pourra supprimer toute cette mécanique pour gérer la mémoire comme si les options n'avaient jamais existé. Le compilateur peut enrichir l'objet avec les attributs et les méthodes des options, comme si ceux-ci avaient été déclarés dans l'objet métier. Pour cela, tous les sources devront être recompilés en spécifiant les options utilisées par le programme. Une compilation totale en deux phases est envisageable. La première détecte les options utiles, la seconde compile le programme en conséquence.

Je n'ai pas expérimenté cela. Un dérivé de `cfront` d'ATT devrait pouvoir tester cette idée relativement facilement.

4.4 Subject-oriented programming (SOP)

Bill Harrison et Harold Ossher de IBM Research proposent une extension au C++ permettant de traiter ce problème [JV]. Leurs idées est d'augmenter les langages orientés objets par des facilités de regroupement de code suivant un usage particulier. Leurs approche permet d'augmenter les capacités d'un objet en décrivant des règles de composition.

```
equate(class AbstractFactory, (AbstractFactory, ConcreteFactory));
```

permet d'indiquer lors de la compilation que toutes les créations d'objet `AbstractFactory` seront remplacées par la création de `ConcreteFactory`.

```
merge(class AbstractFactory, (AbstractFactory, ExtensionC));
```

permet d'enrichir l'interface de `AbstractFactory` à l'aide de l'interface de `ExtensionC`.

```
override(class Clock, (Clock, MyClock));
```

Permet de modifier le code de `Clock` avec les méthodes de la classe `MyClock`. Si la classe `Clock` possède une méthode `setTime()`, la classe `MyClock` peut la modifier pour toutes les instances `Clock` du programme. Les autres méthodes ne sont pas modifiées.

Ces règles permettent de paramétrer le code de l'application sans intervenir sur les sources existants. Il faut par contre, recompiler tout le programme. L'ensemble des règles permet de gérer les options à sélectionner pour un usage particulier.

5. CONCLUSION

Je pense que la difficulté de gérer correctement les options avec le modèle objet est un des défauts majeurs de ce modèle de données. Comme on vient de le voir, malgré plusieurs propositions différentes pour résoudre cela, aucune n'apporte une réelle satisfaction avec les langages objets actuels. Peut-être existe-t-il une approche non encore envisagée ? Dans l'immédiat, il faut choisir au cas par cas, la solution la moins gênante.

6. REFERENCES

- [DP] « Design Patterns », 1994, de Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley (ISBN : 0-201-63361-2)
- [JV] « Subject-Oriented Design », de John Vlissides, C++ Report February 1998, Vol 10/No 2