

L'accès aux attributs

Philippe PRADOS

pp@philippe.prados.name



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

TABLE DES MATIERES

1.1	Accès avec get et set.....	3
1.2	Accès direct.....	5
1.3	Duplication des services	5
1.4	Accès à une agrégation du type pointeur	5
1.5	Accès aux conteneurs	6

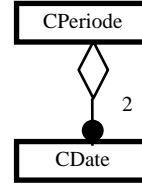
Il arrive fréquemment que l'on désire offrir l'accès à un attribut d'une classe. Plusieurs solutions sont alors possibles. La plus simple consiste à rendre public cet attribut. Cette approche est à déconseiller car vous auriez alors des difficultés à modifier votre classe. Si vous désiriez supprimer, par la suite, l'attribut, l'interface de votre classe changerait, et vous devriez réviser les accès à celle-ci. Il semble préférable d'offrir un accès aux attributs à l'aide de méthodes inline.

Prenons un exemple d'agrégation simple :

```
class CDate
{
    int _jour;
    int _mois;
    int _annee;
public:
    CDate(int jour,int mois,int annee)
    : _jour(jour), _mois(mois), _annee(annee)
    { }
    CDate(const CDate& x)
    : _jour(x._jour), _mois(x._mois), _annee(x._annee)
    { }
    int jour() const { return _jour; }
    int mois() const { return _mois; }
    int annee() const { return _annee; }
    CDate& operator ++()
    { // Ajoute un jour
      // ...
      return *this;
    }
};

class CPeriode
{
    CDate _debut;
    CDate _fin;
public:
    CPeriode(const CDate& dateDebut,const CDate& dateFin)
    : _debut(dateDebut), _fin(dateFin) {}

    //...
};
```



La méthode `CDate& CDate::operator++()` n'est pas rédigée complètement. Elle permet d'ajouter un jour à la date. Le changement de mois et d'année devant être pris en compte. Cet opérateur n'est pas du type `const` car il modifie l'objet.

1.1 Accès avec get et set

Nous désirons offrir des accès aux attributs `_debut` et `_fin` de `CPeriode`. La première approche consiste à offrir pour ces deux attributs les services `get` et `set`, souvent appelés "accesseurs".

```
CDate getDebut() const { return _debut; }
void setDebut(CDate date) { _debut=date; }
CDate getFin() const { return _fin; }
void setFin(CDate date) { _fin=date; }
```

Les services `getX()` retournent une copie de l'attribut concerné. Les services `setX()` copient dans les attributs un nouvel objet `CDate`. Il est possible de réduire les copies des objets `CDate` nécessaires en modifiant légèrement ces méthodes.

```
const CDate& getDebut() const { return _debut; }
void setDebut(const CDate& date) { _debut=date; }
const CDate& getFin() const { return _fin; }
void setFin(const CDate& date) { _fin=date; }
```

Ces nouvelles versions réduisent le nombre de copies nécessaires. Les retours des méthodes `getX()` sont des références sur des constantes. Elles pointent directement sur les attributs de l'objet `CPeriode`. Les paramètres des méthodes `setX()` étant des références sur des constantes, l'objet `CDate` fourni par l'utilisateur de la classe n'est pas copié dans le paramètre de la méthode. Celui-ci aurait ensuite dû être recopié dans l'attribut, et enfin détruit avant de sortir de celle-ci.

L'accès à ces attributs est parfaitement contrôlé par la classe `CPeriode`. Regardons comment l'utilisateur peut modifier un de ceux-ci. Par exemple, cherchons à ajouter un jour à l'attribut `debut` d'une instance `CPeriode`.

```
void main()
{
    CPeriode periode(CDate(1,1,1995),CDate(31,1,1995));

    CDate x=periode.getDebut();
    ++x;
    periode.setDebut(x);
}
```

Il faut prendre une copie de l'attribut `_debut`, puis appeler l'opérateur `++` de cette copie, et enfin demander la mise à jour de l'attribut. Cette mise à jour sera faite par l'opérateur d'assignation de `CDate` appelé dans `setDebut()`. On constate que la manipulation d'un attribut d'une classe avec cet accès, est extrêmement lourd. Imaginez que l'attribut soit un arbre complexe, auquel vous désirez ajouter un élé-

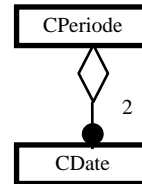
ment. Il faudra créer une copie de cet arbre, ajouter l'élément voulu, puis détruire l'arbre de CPeriode, et recopier l'arbre temporaire modifié pour enfin détruire l'arbre de la copie.

Cet interface oblige à avoir :

- Une copie via le constructeur de copie.
- La modification de la copie de l'attribut.
- Une affectation qui correspond à un destructeur suivi d'un constructeur de copie.
- Une destruction de la copie.

Regardons comment la classe CPeriode peut évoluer. Admettons qu'il existe une classe CDateEtendu héritant de CDate. Une nouvelle version de CPeriode utilise dorénavant deux attributs de type CDateEtendu. Cette nouvelle version modifie l'opérateur d'incréméntation pour tenir compte des années bissextiles.

```
class CDateEtendu : public CDate
{ public:
    CDateEtendu(int jour,int mois,int annee)
    : CDate(jour,mois,annee)
    { }
    CDateEtendu(const CDate& x)
    : CDate(x)
    { }
    CDateEtendu& operator ++()
    { // Ajoute un jour en tenant compte des années bissextiles
      return *this;
    }
};
```



```
class CPeriode // Version 2
{ CDateEtendu _debut;
  CDateEtendu _fin;
  public:
  // ...
};
```

Plusieurs approches sont possibles pour adapter la nouvelle classe CPeriode en maintenant une compatibilité ascendante avec la version précédente. La première consiste à maintenir intégralement l'interface.

```
class CPeriode // Version 2.1
{ public:
    CPeriode(const CDate& dateDebut,const CDate& dateFin)
    : _debut(dateDebut), _fin(dateFin) {}

    const CDate& getDebut() const { return _debut; }
    void setDebut(const CDate& date) { _debut=date; }
    const CDate& getFin() const { return _fin; }
    void setFin(const CDate& date) { _fin=date; }
    //...
};
```

L'utilisateur devant manipuler une copie de l'attribut, il manipule une instance de CDate et non une instance de CDateEtendu. L'appel de l'opérateur ++() ne tiendra pas compte des années bissextiles. Modifions légèrement l'interface, pour tenir compte du nouveau type.

```
class CPeriode // Version 2.2
{ public:
    CPeriode(const CDate& dateDebut,const CDate& dateFin)
    : _debut(dateDebut), _fin(dateFin) {}

    const CDateEtendu& getDebut() const { return _debut; }
    void setDebut(const CDateEtendu& date) { _debut=date; }
    const CDateEtendu& getFin() const { return _fin; }
    void setFin(const CDateEtendu& date) { _fin=date; }
    //...
};
```

L'utilisateur aura une erreur lors de l'appel des méthodes setX() car celles-ci désirent maintenant une classe de type CDateEtendu et non une instance de CDate.

```
void main()
{ CPeriode periode(CDate(1,1,1995),CDate(31,1,1995));

  CDate x=periode.getDebut();
  ++x;
  periode.setDebut(x); // Erreur !
}
```

L'utilisateur de la classe devra modifier toutes les utilisations de setX() pour tenir compte du nouveau type.

```

void main()
{ CPeriode periode(CDate(1,1,1995),CDate(31,1,1995));

  CDateEtendu x=periode.getDebut();
  ++x;
  periode.setDebut(x);
}

```

L'interface à l'aide de `setX()` et `getX()` n'est pas efficace et n'est pas évolutive.

1.2 Accès direct

En changeant l'interface, il est possible d'offrir l'accès à l'attribut *in situ*. Retournons une référence non constante sur l'attribut.

```
CDate& debut() { return _debut; }
```

Avec cet accès, il est possible de modifier l'attribut directement dans l'objet.

```
++periode.debut();
```

Il faut également offrir un accès à celui-ci en constante.

```
const CDate& debut() const { return _debut; }
```

Si par la suite, la nouvelle version de `CPeriode` utilise la classe `CDateEtendu`, il faut adapter les méthodes d'accès.

```
CDateEtendu& debut()           { return _debut; }
const CDateEtendu& debut() const { return _debut; }
```

L'utilisation directe de l'attribut sera toujours valide.

C'est strictement identique à présenter l'attribut en `public`. La seule différence entre cette approche et l'attribut `public` est qu'il est possible, par la suite, de modifier la méthode `debut()` pour retourner un objet simulant le type `CDate` et offrant une modification du nouvel attribut de la classe `CPeriode`. Si vous modifiez la classe `CPeriode` en supprimant les attributs `CDate`, vous pouvez offrir une interface de la classe, compatible avec l'interface précédente, en retournant un objet simulant `CDate`. Mais, pour un simple changement de type d'un attribut, c'est très compliqué.

1.3 Duplication des services

Si on ne désire pas offrir l'accès direct aux attributs d'un objet, il faut offrir l'interface avec toutes les méthodes des attributs. Dans l'exemple précédent, la classe `CPeriode` doit être rédigée comme suit :

```

class CPeriode // Version 2
{ public:
  CPeriode(const CDateEtendu& dateDebut,const CDateEtendu& dateFin)
  : _debut(dateDebut), _fin(dateFin) {}

  void incDebut() { ++_debut; }
  void incFin()   { ++_fin; }
  //...
};

```

La classe `CPeriode` doit avoir les méthodes de manipulation pour modifier directement les attributs sans avoir à faire de copies et sans devoir leur fournir un accès `public`. Les méthodes `getX()` et `setX()` ne sont pas obligatoires. Elles doivent être ajoutées si l'on désire vraiment offrir ces accès. La manipulation des attributs par la classe est plus efficace. Il faut éviter d'inciter l'utilisateur à copier les objets. Les méthodes `getX()` seront souvent présentes mais pas les méthodes `setX()`. Avec cette approche, il est possible de modifier radicalement la classe `CPeriode` sans changer l'interface `public` de celle-ci. Les attributs `CDate` peuvent même disparaître de la classe `CPeriode`.

1.4 Accès à une agrégation du type pointeur

Si votre objet possède une agrégation effectuée par un pointeur, il faut offrir une interface compatible avec l'agrégation indépendamment du fait qu'elle soit rédigée avec un pointeur. Supposons que la classe `CPeriode` possède deux pointeurs sur `CDate` à la place de deux attributs.

```

class CPeriode
{ CDate* _debut;
  CDate* _fin;
public:
  CPeriode(const CDate& dateDebut,const CDate& dateFin)
  : _debut(new CDate(dateDebut)), _fin(new CDate(dateFin)) {}
  ~CPeriode()
  { delete _debut;
    delete _fin;
  }
  //...
};

```

Il est possible d'offrir des accès à ces attributs en retournant des pointeurs.

```
CDate* getDebut() const { return _debut; }
```

Cette méthode peut être constante car l'objet `CPeriode` n'est pas modifié. Par contre, l'attribut `_debut` peut l'être par l'appelant, ce qui revient, par effet de bord, à modifier l'objet `CPeriode`. Il faut, pour respecter l'agrégation, offrir un accès légèrement différent :

```
const CDate* getDebut() const { return _debut; }
```

Avec cette écriture, la notion d'agrégation est préservée. L'attribut `const` d'une méthode doit indiquer la possibilité de modifier l'objet, quels que soient les moyens d'accès. C'est différent d'une relation. Un objet en relation peut, par principe, être modifié tout seul. Dans ce cas, une méthode d'accès constante peut retourner un pointeur non constant. La classe `CEmploye` ci-dessous possède une *relation* avec une entreprise.

```
class CEmploye
{ CEntreprise* _entreprise;
public:
  CEntreprise* getEntreprise() const
  { return _entreprise; }
};
```

Plusieurs employés peuvent être en relation avec la même entreprise.

Si l'attribut pointeur doit toujours être présent, la valeur `NULL` étant impossible, il est préférable de retourner une référence sur l'attribut.

```
const CDate& getDebut() const { return *_debut; }
```

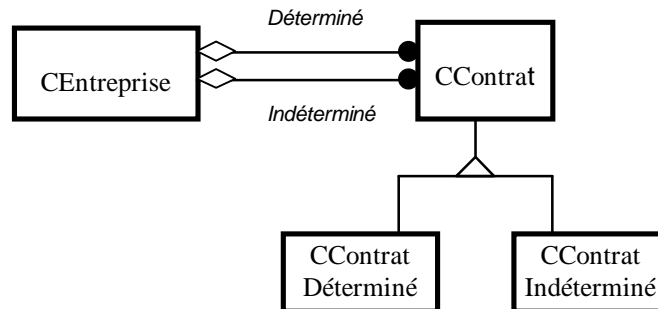
Le choix de l'implantation de l'agrégation en pointeur ne concerne pas l'utilisateur de la classe qui doit pouvoir utiliser cette dernière comme si l'agrégation était classique, comme si la classe utilisait des attributs.

1.5 Accès aux conteneurs

Un "conteneur" est un objet regroupant un ensemble d'objets. Un tableau ou une liste chaînée sont des conteneurs. Un conteneur est généralement associé à un "itérateur". Un itérateur est un curseur permettant de parcourir l'ensemble des éléments d'un conteneur.

Les objets possèdent très souvent des attributs étant des conteneurs. En effet, cela permet de rédiger les relations entre objets. Cela permet également d'écrire une agrégation multiple. Tous les objets contenus appartiennent au conteneur.

Prenons un exemple d'attribut de type conteneurs. Une entreprise emploie des employés. Deux types de contrats sont possibles : les contrats à durée indéterminée, et les contrats à durée déterminée. Admettons que l'on désire avoir deux attributs de type conteneurs pour l'objet `CEntreprise`, un pour les contrats à durée indéterminée, et un autre pour les contrats à durée déterminée.



```
class CEntreprise
{ CList<CContrat*> _determine;
  CList<CContrat*> _indetermine;
public:
  //...
};
```

Comment offrir une interface sur ces conteneurs ? Une première approche consiste à offrir un accesseur.

```
const CList<CContrat*>& getContratDetermine() const
{ return _determine; }
const CList<CContrat*>& getContratIndetermine() const
{ return _indetermine; }
```

L'utilisateur de la classe peut alors parcourir les conteneurs à l'aide de l'itérateur approprié. Pour pouvoir modifier par la suite le type de conteneur de la classe, il faut encapsuler les conteneurs dans des `typedef`. Si vous désirez plus tard, modifier le type `CList<T>` par une liste double-chaînée ou un algorithme de H-code, vous ne modifierez pas l'interface de la classe.

```
class CEntreprise
{ public:
  typedef CList<CContrat*> TContContrat;
  typedef CListIterator<CContrat*> TContratIterator;
  const TContContrat& getDetermine() const
  { return _determine; }
  const TContContrat& getIndetermine() const
  { return _indetermine; }
private:
```

```

TContContrat _determine;
TContContrat _indetermine;
//...
};

```

L'utilisateur utilisera les types déclarés dans la classe CEntreprise pour parcourir les conteneurs.

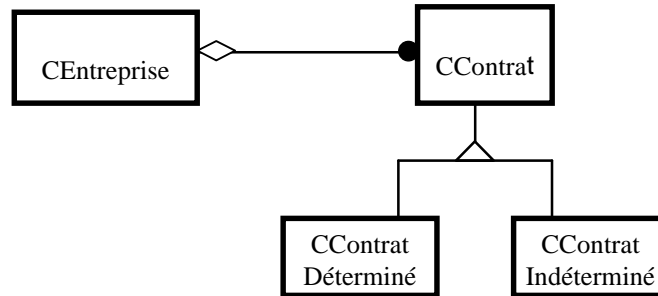
```

CEntreprise entreprise;

CEntreprise::TContratIterator i(entreprise.getDetermine());
for (i.first(); !i.last(); ++i)
{ //...
}

```

Que se passe-t-il si on désire modifier l'implantation de classes pour n'avoir qu'un seul conteneur, mais cette fois-ci attribué ?



L'interface précédente n'est plus valide. Vous ne pouvez donc pas modifier le corps de votre classe dans ce cas.

Pour pouvoir offrir une interface robuste aux évolutions de la classe, il faut procéder autrement. La classe CEntreprise peut être vue comme un conteneur de contrat. Il faut alors rédiger deux itérateurs spécifiques pour cette classe. Un itérateur s'occupera de parcourir les contrats à durée déterminée, et un autre itérateur s'occupera des contrats à durée indéterminée.

```

class CEntreprise
{
    CList<CContrat*> _determine;
    CList<CContrat*> _indetermine;
    //...
    friend class CEntrepriseDetermineIterator;
    friend class CEntrepriseIndetermineIterator;
};

class CEntrepriseDetermineIterator : public CListIterator<CContrat*>
{
public:
    CEntrepriseDetermineIterator(const CEntreprise& entreprise)
        : CListIterator<CContrat*>(entreprise._determine)
    { }
};

class CEntrepriseIndetermineIterator : public CListIterator<CContrat*>
{
public:
    CEntrepriseIndetermineIterator(const CEntreprise& entreprise)
        : CListIterator<CContrat*>(entreprise._indetermine)
    { }
};

```

L'utilisateur peut alors parcourir les conteneurs en utilisant les nouvelles classes.

```

CEntreprise entreprise;

void f()
{
    CEntrepriseDetermineIterator i(entreprise);
    for (i.first(); !i.last(); ++i)
    { //...
    }
}

```

L'interface est plus agréable. Si par la suite, le corps de la classe évolue en ne gardant qu'un seul conteneur, il suffit de modifier les classes itérateurs pour filtrer les éléments parcourus suivant le critère correspondant.

```

class CEntrepriseDetermineIterator : public CListIterator<CContrat*>
{
public:
    CEntrepriseDetermineIterator(const CEntreprise& entreprise)
        : CListIterator<CContrat*>(entreprise._contrats)
    { }
    void first()
    {
        CListIterator<CContrat*>::first();
        for (; !last() && (data()->duree())!=0; ++*this)
        ;
    }
};

```

```
}  
//...  
};
```

Comme pour les attributs simples, il ne faut pas offrir un accès direct à ces éléments, mais offrir les services de manipulation à travers la classe.