

Mutation d'instance

Philippe Prados

pp@philippe.prados.name



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

TABLE DES MATIERES

1.	Mutation avec héritage.....	3
2.	Mutation avec association.....	3
3.	Mutation avec héritage multiple	4

Avant propos

Ce document décrit les différentes stratégies pour permettre la mutation d'une instance.

Il est parfois nécessaire d'avoir un objet mutant. Celui-ci est d'un type particulier à un moment de son existence, puis il se transforme en un autre type par la suite. Cet objet se transforme en un autre objet.

Pour prendre un exemple concret, imaginons la modélisation d'un « loup-garou ». Dans les contes fantastiques, un loup-garou est un *homme* qui se transforme en un *loup* les nuits de pleine lune. Plusieurs approches sont possibles pour modéliser cette transformation. L'objectif est de transformer une instance de type *Homme* en une instance de type *Loup*. La première approche consiste à détruire l'instance *Homme* et de construire une nouvelle instance *Loup* à la place. Cette approche est difficile à mettre en place. En effet, les objets sont identifiés par leurs adresses. La nouvelle instance *Loup* n'est pas à la même adresse que l'instance *Homme* précédente. Pour le système, cela ne correspond pas à une mutation de l'objet, mais à la création d'un nouvel objet et la destruction du précédent. Toutes les références sur l'objet *Homme* deviennent invalides après la mutation. Pour écrire correctement cette approche, il faut maintenir l'ensemble des relations avec l'objet *Homme* et les mettre à jour après la mutation. C'est très lourd et compliqué. Le risque de pointeurs flottants est alors très important.

1. MUTATION AVEC HERITAGE

D'autres approches permettent de modéliser réellement la mutation. Tout dépend des caractéristiques de mutation. Si l'objet mutant peut être décrit par la classe héritée, la meilleure approche est de ventiler les méthodes virtuelles suivant l'état courant de la mutation.

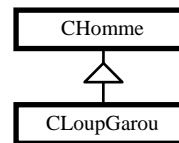
```
class CHomme
{ public:
  virtual ~CHomme()
  {}
  virtual void alimentation() const
  { cout << "omnivore" << endl; }
  virtual void profession() const
  { cout << "journaliste" << endl; }
};

class CLoupGarou : public CHomme
{ enum {THomme, TLoup} _mode;

public:
  CLoupGarou() : _mode(THomme) {}
  virtual void alimentation() const
  { if (_mode==TLoup) cout << "carnivore" << endl;
    else CHomme::alimentation();
  }
  void profession() const
  { assert(_mode==THomme);
    CHomme::profession();
  }
  void mutation()
  { _mode=(_mode==THomme) ? TLoup : THomme;
  }
};

void main()
{ CLoupGarou paul;

  paul.alimentation();
  paul.profession();
  paul.mutation();
  paul.alimentation();
}
```



La classe *CLoupGarou* maintient un mode lui permettant de modifier le comportement des méthodes virtuelles. Ces méthodes appellent les versions héritées, ou bien, elles en modifient le comportement. Dans l'exemple précédent, la méthode *alimentation()* est résolue à l'exécution suivant l'état courant de l'instance. Une méthode particulière de l'objet permet de changer l'état de l'instance, de muter l'objet.

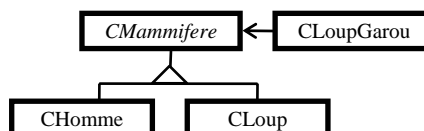
La méthode *profession()* ne peut être appelée que si l'instance mutante est du type *CHomme*. Un *assert()* vérifie cette précondition. La demande de *profession()* d'un loup-garou les nuits de pleine lune entraînent l'arrêt du système pour incohérence.

Si la mutation doit s'effectuer entre deux classes, l'approche précédente n'est pas valide. La classe *CLoup* n'est pas présente dans cet exemple. La classe *CHomme* n'est pas transformée en une classe *CLoup*. C'est le mode de la classe *CLoupGarou* qui identifie la mutation.

2. MUTATION AVEC ASSOCIATION

Pour résoudre une mutation entre deux classes, une des approches consiste à maintenir une relation avec l'instance réelle de l'objet. Pour la mutation de la classe *CHomme* en classe *CLoup*, la classe mutante offre une interface sur ces deux classes.

```
class CMammifere
{ public:
  virtual void alimentation() const =0;
  virtual ~CMammifere()
  {}
};
```



```

    {}
};

class CHomme : public CMammifere
{ public:
    virtual void alimentation() const
    { cout << "omnivore" << endl; }

    virtual void profession() const
    { cout << "journaliste" << endl; }
};

class CLoup : public CMammifere
{ public:
    virtual void alimentation() const
    { cout << "carnivore" << endl; }
};

class CLoupGarou
{ enum {THomme,TLoup} _mode;
  CMammifere* _instance;

public:
    CLoupGarou() : _mode(THomme), _instance(new CHomme()) {}
    void alimentation() const
    { _instance->alimentation(); }
    ~CLoupGarou()
    { delete _instance; }
    void profession() const
    { assert(_mode==THomme);
      ((CHomme*)_instance)->profession(); }
    void mutation()
    { delete _instance;
      if (_mode==THomme)
      { _mode=TLoup;
        _instance=new CLoup();
      }
      else
      { _mode=THomme;
        _instance=new CHomme();
      }
    }
};

void main()
{ CLoupGarou paul;

  paul.alimentation();
  paul.profession();
  paul.mutation();
  paul.alimentation();
}

```

Un pointeur sur la classe de base est présent dans la classe mutante. C'est l'adresse de cette instance mutante qui identifie l'objet. Cette classe manipule deux types d'instances suivant l'état courant de l'objet. La mutation se traduit par la destruction de l'instance `CHomme` et par la création d'une instance `CLoup`. L'ensemble des méthodes offertes par les deux objets `CHomme` et `CLoup` est redéfini pour sous-traiter celles-ci à l'instance associée. La classe `CLoupGarou` est un « proxy » des deux classes.

L'inconvénient de cette approche est qu'il n'est pas possible d'utiliser le polymorphisme entre les classes `CLoupGarou`, `CLoup` et `CHomme`. Ces classes ne sont pas en relation d'héritage. Les méthodes de `CLoupGarou` *simulent* les méthodes de `CLoup` et de `CHomme`. Un `CLoupGarou` n'est pas une sorte de `CLoup` ni une sorte de `CHomme`.

3. MUTATION AVEC HERITAGE MULTIPLE

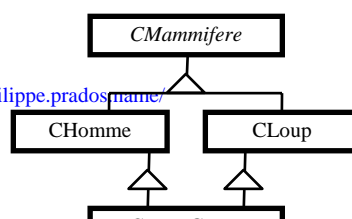
Pour régler la mutation en gardant les relations d'héritages, il faut utiliser une nouvelle approche qui permet de garder les capacités de polymorphisme entre les classes.

La classe `CLoupGarou` doit hériter des deux classes `CHomme` et `CLoup`. Un `CLoupGarou` est une sorte d'homme et une sorte de loup, mais pas en même temps. Nous cherchons à implanter une sorte d'héritage dynamique d'instance. Chaque instance désire indiquer la classe qu'elle hérite à un moment donné de son existence.

```

class CMammifere
{ public:

```



```

    virtual void alimentation() const =0;
    virtual ~CMammifere()
    {}
};

class CHomme : public CMammifere
{
public:
    virtual void alimentation() const
    { cout << "omnivore" << endl; }
    virtual void profession() const
    { cout << "journaliste" << endl; }
};

class CLoup : public CMammifere
{ public:
    virtual void alimentation() const
    { cout << "carnivore" << endl;
    }
};

class CLoupGarou : public CHomme, public CLoup
{ public:
    enum {THomme,TLoup} _mode;

    CLoupGarou() : _mode(THomme) {}
    void alimentation() const
    { (_mode==THomme) ? CHomme::alimentation() : CLoup::alimentation();
    }
    void profession() const
    { assert(_mode==THomme);
      CHomme::profession();
    }
    void mutation()
    { _mode=(_mode==THomme) ? TLoup : THomme;
    }
};

void main()
{ CLoupGarou paul;
  CHomme* pHomme;
  CLoup* pLoup;

  pHomme=&paul;
  paul.alimentation();
  paul.profession();

  paul.mutation();
  paul.alimentation();

  pLoup=&paul;
  pHomme=&paul;          // conversion incorrecte
  pHomme->alimentation();
  pHomme->profession();  // conversion incorrecte
}

```

La classe `CMammifere` peut être héritée virtuellement par `CHomme` et `CLoup` si les caractéristiques `CMammifere` du loup-garou ne se modifient pas lors de la mutation. Cette écriture permet de convertir un `CLoupGarou` en `CHomme` ou en `CLoup`. Le polymorphisme est respecté. Par contre, la conversion n'est valide que lors d'un état particulier de l'instance. Il ne devrait pas être possible de convertir un `CLoupGarou` en `CHomme` lors des nuits de pleine lune. Il serait intéressant d'offrir une protection à l'exécution du type :

```

CHomme* operator &()
{ assert(_mode==THomme);
  return this;
}
CLoup* operator &()
{ assert(_mode==TLoup);
  return this;
}

```

mais cela ne fonctionne pas car le langage ne peut pas différencier deux méthodes par leurs codes retour. Il faut alors ajouter un intermédiaire s'occupant de vérifier la validité de la conversion lors de l'exécution. L'`operator &()` doit retourner un objet ayant deux opérateurs de conversion vers les deux types possibles de la mutation, `CHomme` et `CLoup`. Cet objet intermédiaire vérifie la validité de la conversion.

Nous allons ajouter une classe `PLoupGarou` dans la classe `CLoupGarou`. Celle-ci doit être `private` car personne d'autre que `CLoupGarou` ne doit pouvoir en créer une instance. Cette classe possède un pointeur vers un `CLoupGarou`. L'opérateur `&()` va alors retourner une instance de `CPLoupGarou`.

```
class CLoupGarou : public CHomme, public CLoup
{ //...
private:
class CPLoupGarou
{ CLoupGarou* _pLoupGarou;
public:
CPLoupGarou(CLoupGarou* pLoupGarou)
: _pLoupGarou(pLoupGarou) {}
operator CHomme* ()
{ assert(_pLoupGarou->_mode==THomme);
return _pLoupGarou;
}
operator CLoup* ()
{ assert(_pLoupGarou->_mode==TLoup);
return _pLoupGarou;
}
};
public:
CPLoupGarou operator &()
{ return CPLoupGarou(this); }
};
```

La récupération de l'adresse d'une instance de `CLoupGarou` est vérifiée à l'exécution. Le mode courant de l'instance doit être valide. Par exemple :

```
{ CLoupGarou LoupGarou;
CLoup* p=&LoupGarou; // Erreur
}
```

Malheureusement, cette technique ne fonctionne pas dans tous les cas. Un pointeur `CLoupGarou*` peut être converti en `CLoup*` sans vérification.

```
CLoupGarou* pLoupGarou=&paul
CLoup*      pLoup=pLoupGarou; // Valide même si mode=THomme
```

Pour utiliser l'opérateur déclaré précédemment, il faut utiliser une écriture inhabituelle

```
CLoup*      pLoup=&*pLoupGarou; // Verification de la conversion
```

Si les méthodes vérifient le contexte courant de l'instance en précondition, les conversions erronées seront détectées à l'exécution des méthodes mais pas lors de la conversion. Il n'est alors pas nécessaire d'ajouter l'artifice précédent. Celui-ci étant incomplet comme on vient de le voir.

Cette technique doit rester rarissime. Il est préférable de modifier la conception que de devoir écrire ce type de modélisation. La plupart des mutations peuvent se régler à l'aide de l'approche modale expliquée au paragraphe « Mutation avec héritage ».