

Contrôle d'accès statique dans les langages à objets

Marianne Huchard

29 novembre 2007

Fondements des langages à objets

Principales caractéristiques :

- ▶ classes, objets
- ▶ messages
- ▶ héritage, polymorphisme

Avec l'objectif de faciliter :

- ▶ la conception de logiciels de grande taille,
- ▶ la maintenance correctrice,
- ▶ la réutilisation et l'extension.

Famille des langages à objets à typage statique

Proposition

contrôler au maximum pendant la phase de compilation (génération du programme exécutable), que le receveur d'un message est du bon type, garantissant qu'il peut répondre au message.

L'objectif est d'éviter :

- ▶ l'échec brutal d'un programme en cours d'exécution,
- ▶ le ralentissement de l'exécution par des tests destinés à éviter cet échec brutal.

Savoir répondre *versus* répondre à n'importe qui

Le fait qu'un objet puisse répondre à un message ne justifie pas forcément que n'importe quel autre objet puisse le lui envoyer.

Exemple, le *masquage de l'implémentation* avec ses avantages :

- ▶ corriger une erreur est facile à effectuer car localisé,
- ▶ modifier le stockage des données ou l'implémentation d'un traitement pour rendre la structure plus efficace est aisé car les programmes appelants n'y font pas référence directement.

Retenons

Deux mécanismes duaux dans les langages à contrôle statique.

- ▶ Le *typage statique* s'assure que le receveur est capable de comprendre un message.
- ▶ Le *contrôle d'accès statique* s'assure que l'émetteur d'un message est autorisé à l'envoyer.

Objectifs du contrôle d'accès statique

- ▶ masquer l'implémentation, favoriser l'abstraction (ex. sur un objet *Pile*, on n'a accès qu'aux opérations légales pour le type abstrait de données) ;
- ▶ faire respecter certaines spécifications (ex. un objet *Enfant* n'a pas accès à la méthode *vendre* d'un objet *MagasinAlcool*) ;
- ▶ réduire la dépendance entre composants logiciels : les accès entre objets sont restreints ;
- ▶ faciliter la maintenance : la restriction des accès possibles limite les modifications à apporter en cas d'évolution ;
- ▶ faciliter la réutilisation : grâce aux qualités d'abstraction acquises.

Terminologie du contrôle d'accès statique

Différents autres termes sont employés pour parler du contrôle d'accès statique, il faut les reconnaître :

- ▶ encapsulation,
- ▶ visibilité.

Entités protégées

- ▶ Classes
- ▶ attributs
- ▶ méthodes
- ▶ types (typedef, classes internes)

Droits contrôlés

- ▶ Accès (pas de distinction lecture/écriture),
- ▶ coercition implicite,
- ▶ instanciation (en protégeant les constructeurs),
- ▶ dérivation (dépend de l'accès à la classe).

public, private et protected pour les propriétés

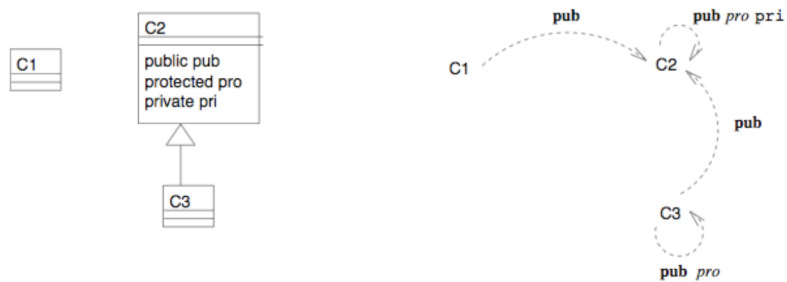


Fig.: 1 - Accès aux propriétés

public, private et protected pour les propriétés

Sans imbrication avec un autre mécanisme :

- ▶ les propriétés `public` : sont accessible depuis le code de n'importe quelle classe ou n'importe quelle fonction. Sur la figure 1, toute méthode de `C1`, `C2`, `C3` peut accéder à l'attribut `pub` de la classe `C2` ;
- ▶ les propriétés `private` : ne sont accessibles que dans les autres méthodes de la même classe ;
- ▶ les propriétés `protected` : sont accessibles par une classe et ses sous-classes mais seulement sur leurs propres instances et les instances de leurs sous-classes ; `C3` n'a pas accès à `pro` sur les objets de type statique `C2`.

friend

Cette directive permet à une classe de donner à une classe, à une méthode ou à une fonction le droit d'accéder à sa partie privée (ou protégée). Les limites de `friend` sont les suivantes :

- ▶ il faut anticiper (une fois l'interface de la classe écrite, on ne peut plus lui ajouter des amis) ;
- ▶ ce n'est pas hérité ;
- ▶ ce n'est pas partagé par les classes internes ;
- ▶ ce n'est pas symétrique ;
- ▶ ce n'est pas transitif.

On peut admettre son utilisation pour des cas restreints et anticipés d'accès, tels que l'écriture des opérateurs qui se fait de manière étroite avec celle de la classe concernée.

friend

```
class Voiture
{
private:
Personne* proprietaire;
public:
Voiture(){ }
virtual ~Voiture(){ }
virtual void vendre(Personne* p);
};
```

friend

```
class Personne
{
private:
string nom;
public:
Personne(){ }
virtual ~Personne(){ }
friend ostream& operator<<(ostream& os, const Personne& p);
friend int main();
friend void Voiture::vendre(Personne* p);
// friend class Voiture --
// donnerait accès depuis toutes les méthodes
// de la classe Voiture
};
```

friend

```
void Voiture::vendre(Personne* p)
{cout << "vendue a " << p->nom << endl;
  proprietaire=p;}

ostream& operator<<(ostream& os,const Personne& p)
{os << p.nom << endl;}

int main()
{
  Personne p;
  p.nom="juju";
  cout << p;
}
```

public, private et protected pour les liens d'héritage

L'utilisation des mots-clefs public, private et protected s'étend à la clause dans laquelle une classe déclare ses super-classes.

```
class C3 : protected virtual C2 {};
```

Ces déclarations permettent à la sous-classe de restreindre, sur ses propres objets, l'accès aux propriétés héritées :

- ▶ une propriété public héritée dans une sous-classe par un lien d'héritage protected devient protected

public, private et protected pour les liens d'héritage

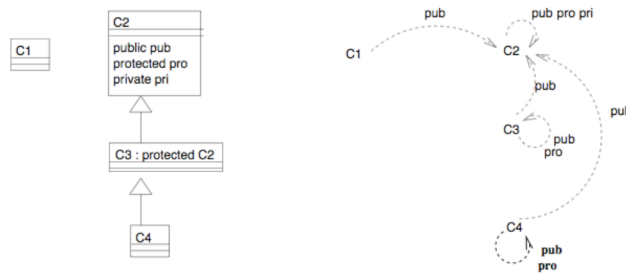
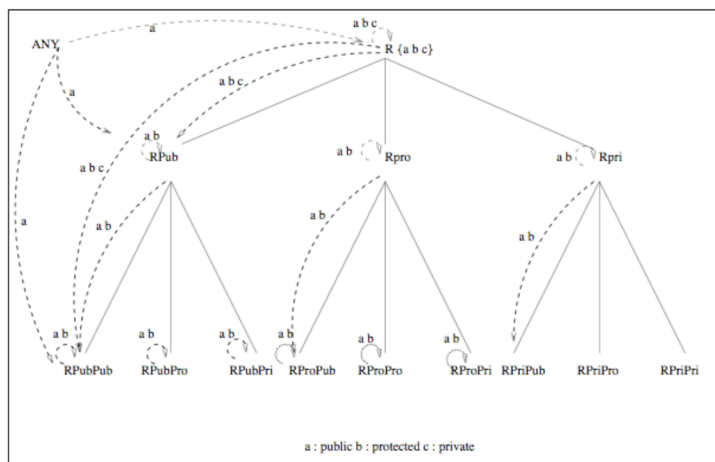


Fig.: Accès aux propriétés avec protected sur un lien d'héritage

public, private et protected pour les liens d'héritage



Conversion implicite (affectation polymorphe)

```
class PersonnePri : protected virtual Personne{...};

int main()
{
    PersonnePri* pi = ....;
    Personne *pe=pi; // interdit : 'Personne' is an
                    // inaccessible base of 'PersonnePri'
}
```

using

using permet, sur une propriété héritée par un lien d'héritage portant une restriction, de ... lever la restriction

```
class Personne
{
private:
    string nom;
protected:
    string adresse;
public:
    string numInsee;
    Personne(){}
    virtual ~Personne(){}
};
```

using

using permet, sur une propriété héritée par un lien d'héritage portant une restriction, de ... lever la restriction

```
class PersonnePro : protected virtual Personne{
public:
// (interdit car prive) using Personne::nom;
using Personne::adresse;
using Personne::numInsee;
virtual ~PersonnePro(){
};
int main(){
PersonnePro *pp=new PersonnePro();
cout << pp->adresse
<< pp->numInsee <<endl;
}
```

Modification lors des redéfinitions

Sans contraintes!

```
class A{
private:
virtual void f(){cout << "f de A"<<endl;}
public:
virtual void g(){this->f();}
virtual void h(){cout << "h de A"; this->f();}
};
class B : public virtual A{
public:
virtual void f(){cout << "f de B"<<endl;}
private:
virtual void h(){cout << "h de B"; this->f();}
};
```

Modification lors des redéfinitions

```
int main(){
A *pa=new A(); cout << "cest pour A" << endl;
pa->g(); //f de A
pa->h(); //h de Af de A

A *pb=new B(); cout << "cest pour B" << endl;
pb->g(); // f de B
pb->h(); // h de Bf de B

B *pbb=new B(); cout << "cest pour BB" << endl;
pbb->g(); //f de B
//(accès interdit cf type statique) pbb->h();
}
```

Contournements

Exemple : contournement du contrôle d'accès à un attribut hérité `protected` sur un objet de la super-classe.

```
class Personne
{
private:
string nom;
protected:
string adresse;
public:
Personne(){}
Personne(string a){adresse=a;}
virtual ~Personne(){}
virtual void affiche(){cout<< adresse;}
};
```

Contournements

Exemple : contournement du contrôle d'accès à un attribut hérité `protected` sur un objet de la super-classe.

```
class PersonnePro : protected virtual Personne{
public:
PersonnePro(){}
PersonnePro(string a):Personne(a){}
virtual ~PersonnePro(){}
virtual void echangeAdresses(Personne&p){
// (accès interdit) p.adresse;
PersonnePro aux1;
(Personne&)aux1=p;
string s_aux=aux1.adresse;
PersonnePro aux2;
aux2.adresse=adresse;
p=(Personne&)aux2;
adresse=s_aux;
}
```

Contournements

Exemple : contournement du contrôle d'accès à un attribut hérité `protected` sur un objet de la super-classe.

```
int main()
{
Personne* p=new Personne("chemin du bois");
PersonnePro *ppro = new PersonnePro("rue des jardins");
cout << "adresse p ="; p->affiche();
cout << " adresse ppro = "; ppro->affiche(); cout << endl;
ppro->echangeAdresses(*p);
cout << "adresse p ="; p->affiche();
cout << " adresse ppro = "; ppro->affiche(); cout << endl;
}
```

Entités protégées

- ▶ Classes,
- ▶ attributs,
- ▶ méthodes.

Droits contrôlés

- ▶ Accès (pas de distinction lecture/écriture),
- ▶ instanciation (en protégeant les constructeurs),
- ▶ dérivation d'une classe,
- ▶ redéfinition d'une propriété.

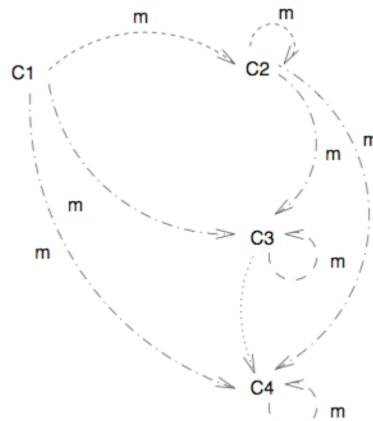
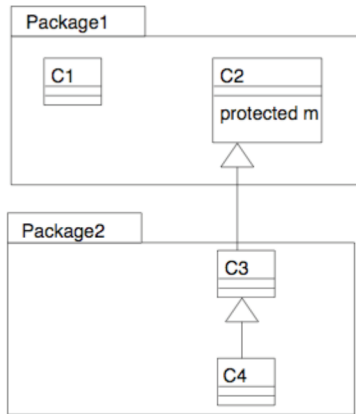
public, private, protected et *package* pour les propriétés

- ▶ *package* (*default*) pour les propriétés auxquelles on n'a pas attaché de mot-clef de contrôle d'accès. Elles sont alors accessibles à tout code écrit dans le même paquetage.
- ▶ *private* et *protected* n'ont pas tout à fait la même signification qu'en C++
 - ▶ *protected* = accessible dans tout code du même paquetage ;
 - ▶ les propriétés *protected* sont accessibles également hors du paquetage chez les sous-classes, sur leurs propres objets ou les objets des sous-classes, tant qu'il n'y a pas redéfinition ;
 - ▶ en C++ une classe a accès sur les objets des sous-classes aux propriétés *private* héritées, tandis qu'en Java ce n'est pas le cas.

Ordre entre les directives (par affaiblissement croissant)

private < *package* < *protected* < *public*

Règle d'accès à une méthode protected



Cas problématique d'accès à une méthode protected

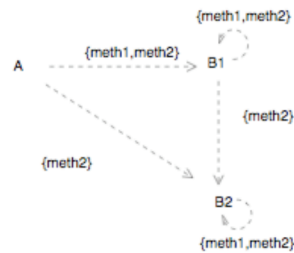
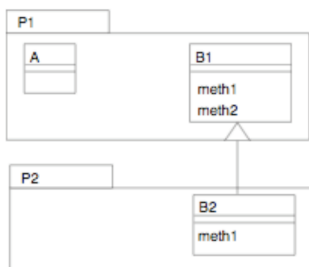


Fig.: Accès aux propriétés en Java (toutes sont protected)

Règle d'accès à une méthode `protected`

Domageable pour la conception : les objets d'une sous-classe n'ont pas la même interface (ne répondent pas aux mêmes messages) que ceux de la super-classe : `meth2` est accessible sur les objets de B1 mais pas de B2 dans certains environnements.

Le code de l'exemple problématique

```
//----- A.java -----  
package P1;  
import P2.*;  
public class A{  
public void test(){  
B1 b1=new B1();  
b1.meth1();  
b1.meth2();  
B1 b12=new B2();  
b12.meth1();  
b12.meth2();  
B2 b22=new B2();  
//(interdit) b22.meth1(); ... sauf si B2 redéfinit meth1 comme public  
b22.meth2();  
}  
}
```

Le code de l'exemple problématique

```
//----- B1.java -----  
package P1;    import P2.*;  
public class B1{  
    protected void meth1(){  
    protected void meth2(){  
    public void test(){  
        B1 b1=new B1();  
        b1.meth1();  
        b1.meth2();  
        B1 b12=new B2();  
        b12.meth1();  
        b12.meth2();  
        B2 b22=new B2();  
        //(interdit) b22.meth1(); ... sauf si B2 redéfinit meth1 comme public  
        b22.meth2();  
    }  
}
```



Le code de l'exemple problématique

```
//----- B2.java -----  
package P2;    import P1.*;  
public class B2 extends B1{  
    protected void meth1(){  
  
    public void test(){  
        B1 b1=new B1();  
        //(interdit) b1.meth1();  
        //(interdit) b1.meth2();  
        B1 b12=new B2();  
        //(interdit) b12.meth1();  
        //(interdit)b12.meth2();  
        B2 b22=new B2();  
        b22.meth1();  
        b22.meth2();  
    }  
}
```



public, private, protected et *package* pour les classes internes

Lors de leur spécialisation dans une sous-classe de la classe englobante, il n'y a pas de règle d'affaiblissement ou de renforcement du contrôle d'accès de la sous-classe interne.

```
public class TestRedef{
  private class Cpri{}
  protected class Cpro{}
}
class SousClasseTestRedef extends TestRedef{
  // Cpri est inaccessible
  private class Cproprisc extends Cpro{}
  public class Cpropubsc extends Cpro{}
}
```

Redéfinition des droits d'accès

L'accès aux propriétés ne peut être remis en cause par une classe qui en hérite que si cette classe la redéfinit ou si la classe a un niveau de protection différent de celui de la super-classe.

Lors d'une redéfinition, on ne peut qu'affaiblir le contrôle d'accès

- ▶ une méthode `private` ne se redéfinit pas (contrairement à C++); si on écrit une méthode privée de même signature dans une sous-classe, c'en est ... une autre, il n'y aura pas de liaison dynamique;
- ▶ une méthode `package` peut se redéfinir `package`, `protected` ou `public`;
- ▶ une méthode `protected` peut se redéfinir `protected` ou `public`;
- ▶ une méthode `public` ne peut que le rester.

Redéfinition des droits d'accès

```
package P3;
public class TestRedef
{
    private void pri(){System.out.println("pri-testredef");}
    protected void pro(){System.out.println("pro-testredef");}
    void pack(){}
    public void pub(){}
    public void testAppelPri(){pri();}
    public void testAppelPro(){pro();}
    public static void main(String[] a){
        TestRedef inst=new SousClasseTestRedef();
        inst.testAppelPri(); // pri-testredef
        inst.testAppelPro(); // pro-sousclassestestredef
        SousClasseTestRedef insts=new SousClasseTestRedef();
        insts.testAppelPri(); // pri-testredef
        insts.testAppelPro(); // pro-sousclassestestredef
    }
}
```

Redéfinition des droits d'accès

```
class SousClasseTestRedef extends TestRedef
{
    private void pri(){System.out.println("pri-sousclassestestredef");}
    // comme une autre méthode pri,
    // donc on lui met la protection que l'on veut !
    public void pro(){System.out.println("pro-sousclassestestredef");}
    // ne peut devenir package ou private, mais peut devenir public
    public void pack(){}
    // peut rester package, et devenir seulement protected ou public
    public void pub(){}
    // ne peut que rester public
}
```

Principes généraux

- ▶ Les droits de lecture et d'écriture sont dissociés, de même que l'appel d'une méthode dans le contexte de la création d'un objet ou plus tard.
- ▶ On peut interdire la redéfinition d'une propriété.
- ▶ L'accès en écriture n'est autorisé que sur `CURRENT`.
- ▶ L'accès en lecture et l'appel des méthodes est autorisé au niveau de la classe.
- ▶ Une propriété accessible par une classe l'est aussi par ses sous-classes.

Directives

Les directives s'appliquent aux propriétés.

- ▶ `feature{A,B,C}` autorise les classes A, B et C et leurs descendantes à accéder à la propriété, dans le cadre de la définition ou de la redéfinition ;
- ▶ `export{A,B,C}` autorise les classes A, B et C et leurs descendantes à accéder à la propriété, on écrit cette directive pour modifier les droits d'accès sans redéfinir la propriété ;
- ▶ `frozen` empêche les descendants de redéfinir la propriété ;
- ▶ `feature{NONE}` et `export{NONE}` sont des équivalents de `private` ;
- ▶ `feature{ANY}` et `export{ANY}` sont des équivalents de `public`.

Un exemple

```
class C
feature{A,G}
  m is do end -- méthode m
  feature{A,E}
    n is do end -- méthode n
  end -- C

class D
inherit C
  export {} n
end -- D
}
```

Un exemple

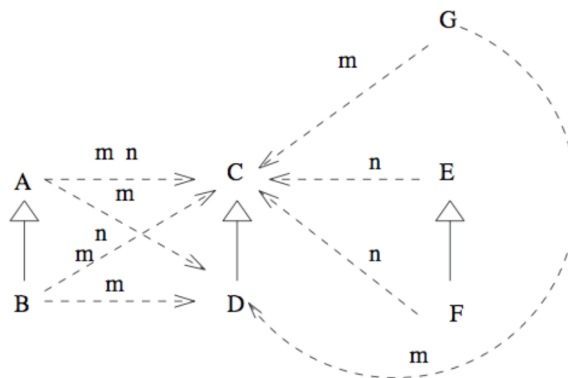


Fig.: Accès en Eiffel

Un bilan sur les mécanismes

- ▶ C++ et Java ont une vision technique, orientée vers le masquage de l'implémentation ;
- ▶ C++ est très complexe, sans doute excessivement ; il est assez difficile de se faire une idée d'une bonne utilisation des mécanismes ;
- ▶ Java est plus simple, mais avec d'étranges comportements comme la modification de certaines interfaces et l'impossibilité de redéfinir des méthodes privées ;
- ▶ Eiffel a une vision des contrôles d'accès plus orientée vers la traduction d'accès liés à la sémantique d'un problème.

Existe-t-il un langage plus riche ?

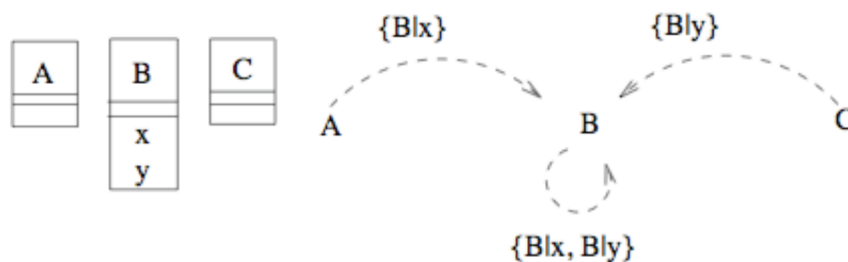


Fig.: Accès exprimables en Eiffel mais pas en Java ni en C++

Existe-t-il un langage plus riche ?

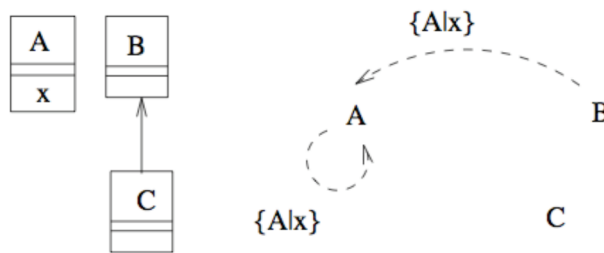


Fig.: Accès exprimables en Java et C++ mais pas en Eiffel

Existe-t-il un langage plus riche ?

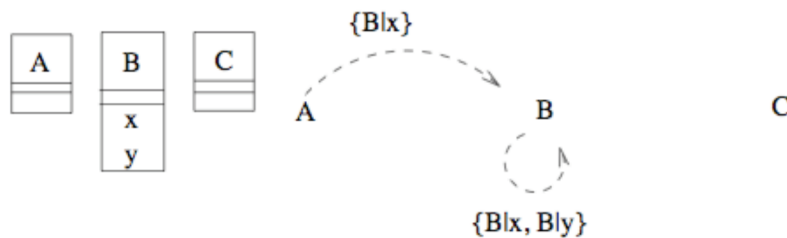


Fig.: Accès exprimables en Java mais pas en C++

Les revendeurs

- ▶ la classe Fournisseur offre deux méthodes :
 - ▶ fournir est destinée aux revendeurs exclusivement pour obtenir des produits d'une certaine référence et dans une quantité donnée ;
 - ▶ listeDeFournisseurs est destinée aux personnes qui désireraient connaître les revendeurs agréés par le fournisseur.
- ▶ la classe Revendeur offre la méthode vendre. Cette méthode est destinée aux personnes sauf dans un cas particulier, les enfants ne doivent y avoir accès que sur la classe RevendeurDeBonbons.

Les revendeurs

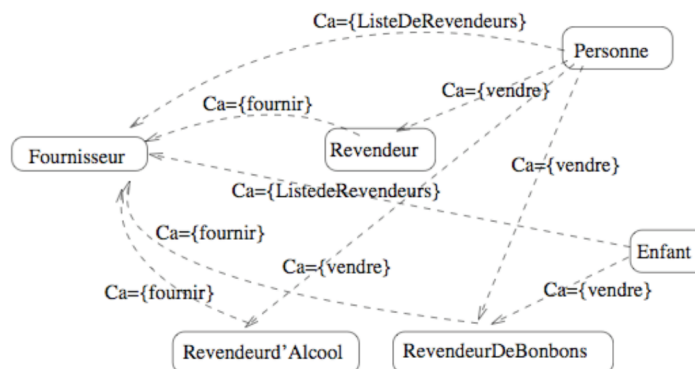


Fig.: Les accès réclamés

Les revendeurs

Exercice : Traiter l'exemple en Java et en Eiffel

La bibliothèque

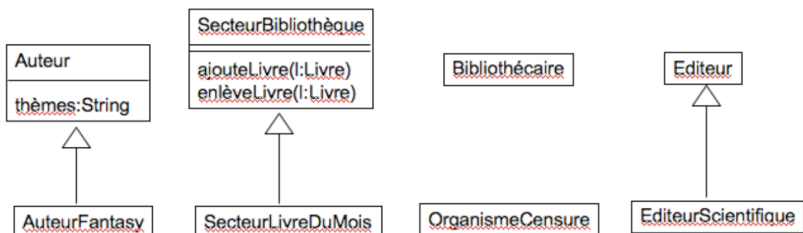


Fig.: Diagramme de classes pour une bibliothèque

La bibliothèque

- ▶ Les bibliothécaires peuvent ajouter et enlever des livres dans tous les secteurs de bibliothèque.
- ▶ Seuls les bibliothécaires ont le droit d'ajouter des livres dans les secteurs de type « le livre du mois » car le choix des livres de ces secteurs, correspondant à l'actualité du moment, leur appartient.
- ▶ Tous les éditeurs (incluant les éditeurs scientifiques) peuvent ajouter des livres dans les secteurs de bibliothèque (à l'exception des secteurs de type « le livre du mois »).
- ▶ Tous les auteurs (incluant les auteurs de fantasy) peuvent ajouter et enlever des livres (généralement leurs propres livres) dans les secteurs de bibliothèque, mais sur les secteurs « le livre du mois » ils ne peuvent qu'enlever des livres.
- ▶ Les organismes de censure peuvent seulement enlever des livres, et ceci dans tous les secteurs de bibliothèque, y compris les secteurs « le livre du mois ».

La bibliothèque

Exercice : Modéliser les accès puis Traiter l'exemple en C++