HLIN603 : Programmation Objet Avancée

Contrôle d’accès statique
Visibilité

Sommaire

Contrôler à la compilation le bien fondé d'un envoi de message à un objet

Cadre des langages à objets à typage statique

- *Type statique* vérifie que l'objet saura répondre au message ; si le type statique est une classe il s'agira de vérifier qu'elle possède bien une méthode de signature conforme à l'envoi de message
- `const` *placé en fin de signature d'une méthode* vérifier que l'objet receveur est une constante ;
- *Généricité paramétrique* contrôle les paramètres de types, ex. dans une collection paramétrée par le type de ses éléments, on ne peut ranger que des éléments du bon type.

Contrôle d'accès statique : un regard dual sur l'envoi de message

L'envoi de message implique :

- l'expéditeur (objet courant - receveur de la méthode où se trouve l'envoi de message, dont le type statique est pointeur constant de la classe) ;
- le message (nom de méthode, types de paramètres et retour pour les cas simples) ;
- le receveur (objet sur lequel est appliquée la méthode, cet objet a un type statique, type de la variable qui désigne l'objet).

Contrôle d'accès statique : un exemple

```
class Revendeur{
private:   ....
public:    ....
virtual vector<Produit*> vendre(string ref, int qty);
};

class Personne{
private:
string nom;
vector<Produit *> placard;
public:   .....
virtual void achete(Revendeur* r, string ref,int qty);
};

void Personne::achete(Revendeur* r,string ref,int qty){
vector<Produit*> v = r->vendre(ref,qty);
........ // ajouter v au placard
}
```

Contrôle d'accès statique : un exemple

```
void Personne::achete(Revendeur* r,string ref,int qty){
vector<Produit*> v = r->vendre(ref,qty);
........ // ajouter v au placard
}
```

La méthode `achete` contient (partie droite de l'affectation) un message dont

- l'expéditeur est caché sous l'identité `this`
- le receveur est `r`
- le message est `vendre(string,int)`

Contrôle d'accès statique : un regard dual sur l'envoi de message

La vérification effectuée par le contrôle d'accès consiste à répondre, en s'appuyant sur les types statiques, à la question générale :

> *l'expéditeur a-t-il le droit d'envoyer le message au receveur ?*

Contrôle d'accès statique : un regard dual sur l'envoi de message

La vérification effectuée par le contrôle d'accès dans notre exemple consiste à répondre à la question particulière :

> *une personne peut-elle envoyer le message `vendre` à un revendeur ?*

Objectifs

- masquer l'implémentation, favoriser l'abstraction
  - sur un objet `Pile`, on n'a accès qu'aux opérations légales pour le type abstrait de données
- faire respecter certaines spécifications du problème
  - on pourrait ainsi compléter l'exemple précédent avec des classes `Enfant` et `RevendeurAlcool`, et un objet `Enfant` n'aurait pas accès à la méthode `vendre` d'un objet `RevendeurAlcool`

Conséquences importantes sur les qualités du logiciel

- 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

Sommaire

Accès aux propriétés

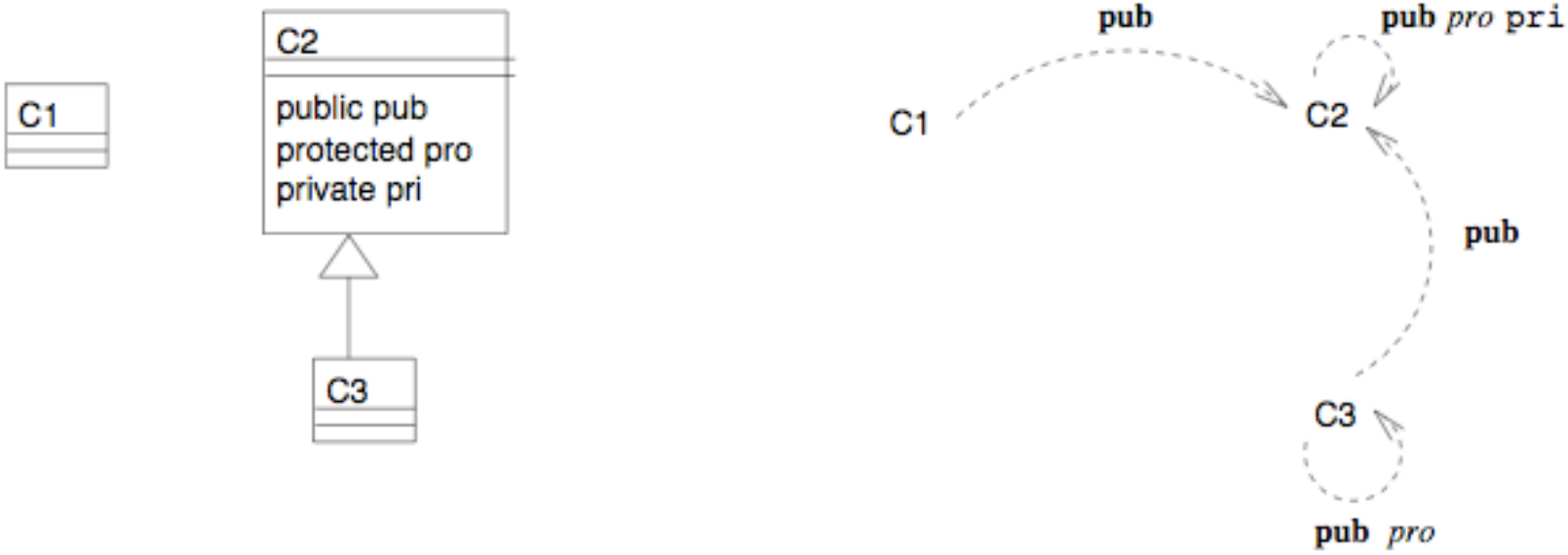

Figure : Accès aux propriétés en dehors d'autres mécanismes. Une flèche de A vers B étiquetée m se lit "dans une méthode de A, on peut envoyer le message m à un objet de type statique B"

Accès aux propriétés

En première approximation

- les propriétés `public:` sont accessibles 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` sur un objet dont le type statique est la classe `C2` ;
- les propriétés `private:` ne sont accessibles que dans les autres méthodes de la même classe ;
- les propriétes `protected:` sont accessibles par une classe et ses sous-classes mais seulement sur leurs propres instances ; `C3` n'a pas accès à `pro` sur les objets de type statique `C2`.

D'autres accès se rajouteront en cas d'héritage déclaré `public` (voir plus loin)

Sommaire

friend

Permet à une classe de donner le droit *d'accéder à sa partie privée (ou protégée)*

- à une autre classe
- à une méthode d'une autre classe
- à une fonction

Voitures et personnes

```
class Garage{....};

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

Voitures et personnes

on autorise l'accès à la partie privée de Personne de

- l'opérateur <<,
- la méthode vendre de la classe Voiture
- la fonction main
- la classe Garage

```
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 Garage;};

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

Limites de `friend`

- 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

Conditions d'utilisation :

- 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
- ne pas généraliser son usage

Sommaire

Clause de déclaration des super-classes

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

Exemple :

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

Ces déclarations permettent à la sous-classe de restreindre, sur ses propres objets, l'accès aux propriétés héritées. Ainsi une propriété `public` héritée dans une sous-classe par un lien d'héritage `protected` devient `protected`.

Héritage

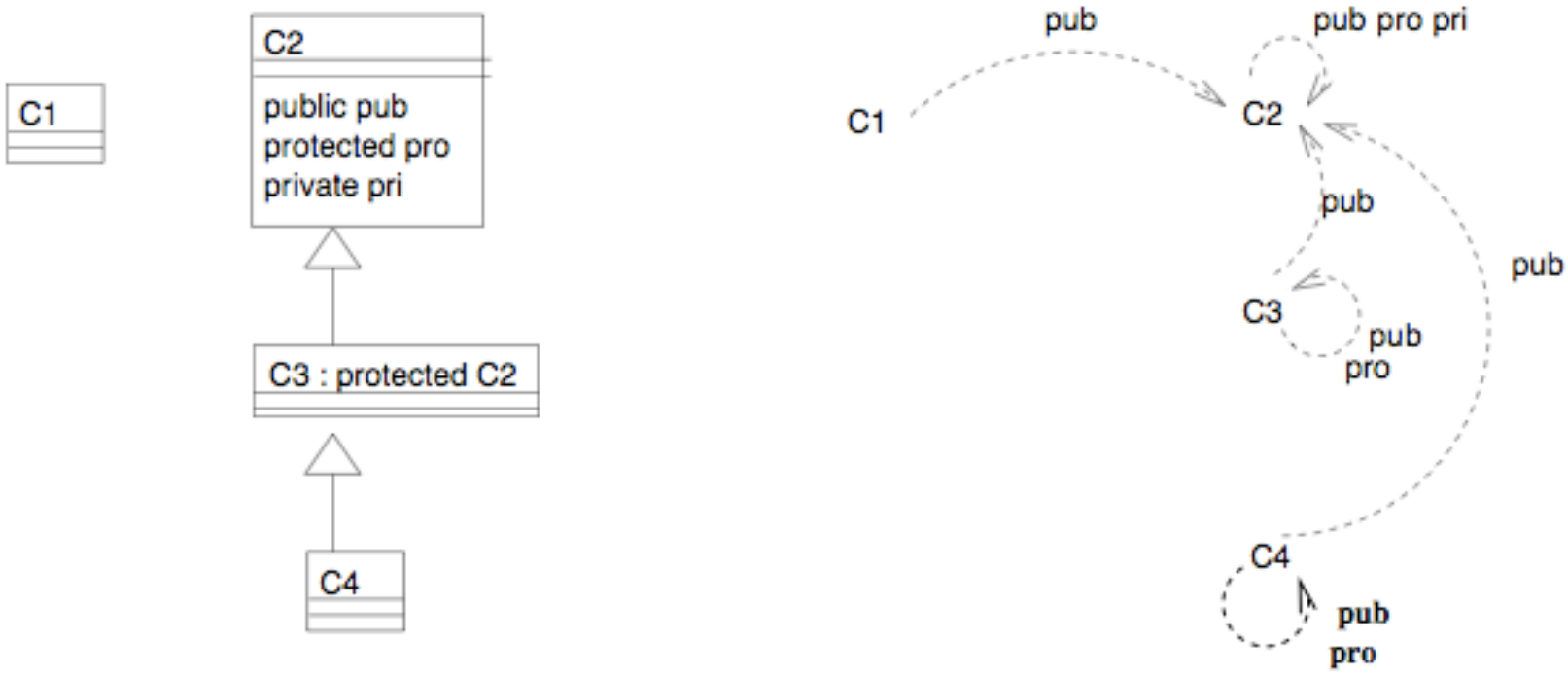

Figure : Accès aux propriétés avec protected sur un lien d'héritage

Nota. Rien n'est précisé pour la classe `C4`, les accès dans les méthodes de `C3` vers les objets de la classe `C4` ne sont pas précisés

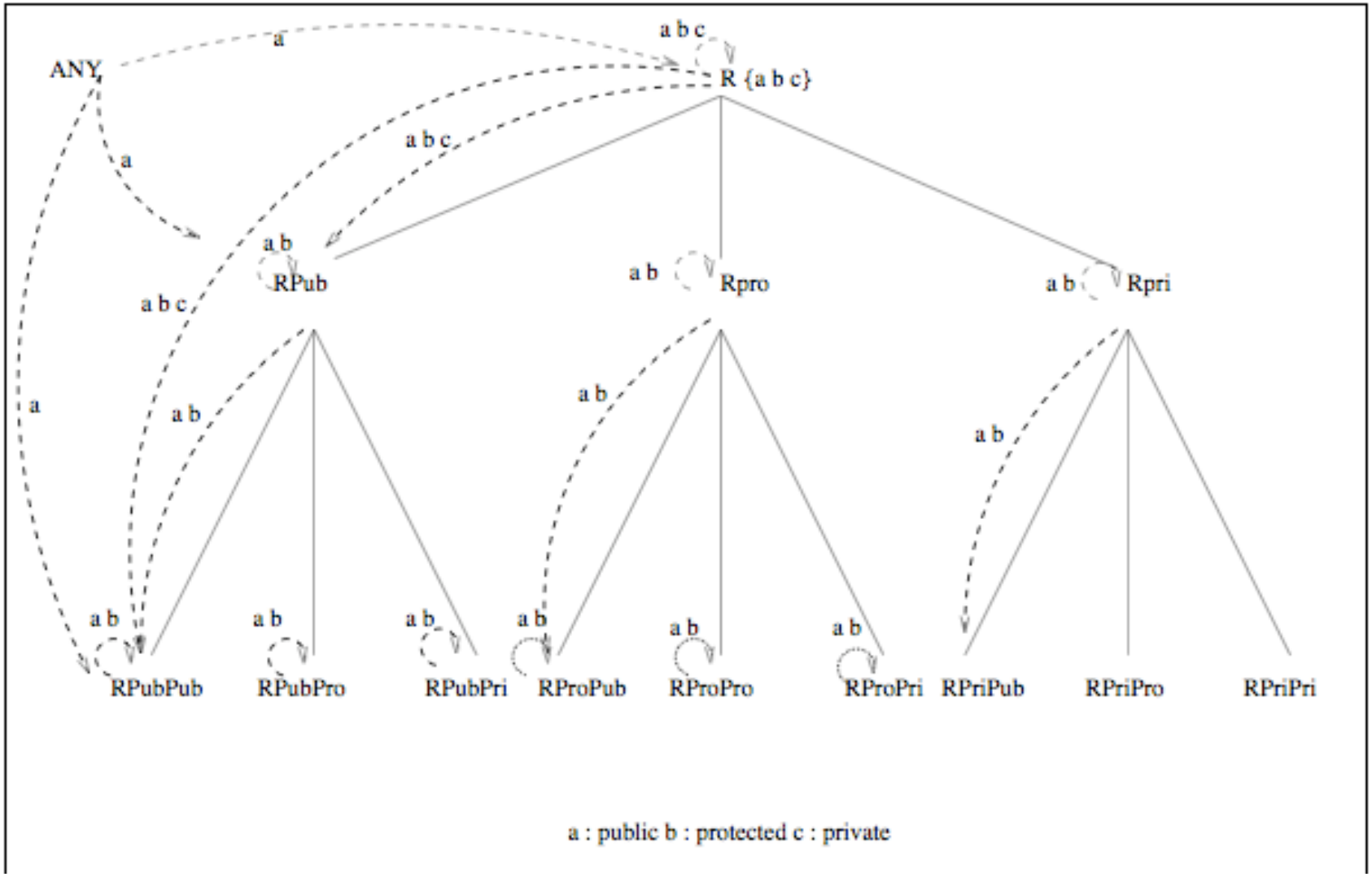

Figure : C_x est dérivée de C avec la protection x

Contrôle d'accès et conversion

De plus, la conversion implicite (affectation polymorphe) devient impossible lorsqu'on restreint la visibilité du lien d'héritage.

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

int main()
{
PersonnePro* pp .....;
Personne *p=pp;  // maintenant impossible
}
```

Choix d'héritage privé ou protégé

Par exemple lorsqu'on utilise l'héritage pour des besoins d'implémentation, mais qu'il n'y a pas de spécialisation entre les types abstraits correspondants

```
template<typename T>
class Pile : private vector<T>
{
........
public:
.........
  virtual void empiler(T t);
};
.........
int main()
{
Pile<int> p;
p.empiler(2);
// (erreur : méthode d'une super-classe privée) p.size();
// (erreur : méthode d'une super-classe privée) p.push_back();
}
```

Sommaire

using

Lever la restriction sur une propriété héritée par un lien d'héritage portant une restriction

Exemple : réhabiliter dans une pile certaines méthodes du vecteur

```
template<typename T>
class Pile : private vector<T>
{
public:
  virtual void empiler(T t);
  using vector<T>::size;
};

int main()
{
Pile<int> p;
p.empiler(2);
cout << p.size() << endl; // maintenant possible
// (toujours erroné : méthode d'une super-classe privée) p.push_back();
}
```

Sommaire

Contrôles d'accès lors de la redéfinition

- Pas de contraintes particulières
- Il est utile de déclarer `virtual` même les méthodes privées !

Contrôles d'accès lors de la redéfinition

```
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();}
};
```

Contrôles d'accès lors de la redéfinition

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

>> cest pour A
>> f de A
>> h de A f de A
```

Contrôles d'accès lors de la redéfinition

```
int main(){
A *pb=new B(); cout << "cest pour B" << endl;
pb->g();
pb->h();
}

>> cest pour B
>> f de B
>> h de B f de B
```

Contrôles d'accès lors de la redéfinition

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

>> cest pour BB
>> f de B
```

Contrôles d'accès lors de la redéfinition

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

Contrôles d'accès lors de la redéfinition

```
int main(){
Bpro *pbo=new Bpro(); cout << "cest pour Bpro" << endl;
pbo->f();
// on ne peut pas accéder à g
}

>> cest pour Bpro
>> f de B
```

Contrôles d'accès lors de la redéfinition

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

Contrôles d'accès lors de la redéfinition

```
int main(){
Bpri *pbi=new Bpri(); cout << "cest pour Bpri" << endl;
pbi->f();
// on ne peut pas accéder à g
}

>> cest pour Bpri
>> f de B
```

Sommaire

Questions sur la modélisation

- Tous ces mécanismes sont-ils réellement utiles et expressifs ?
- oui pour les contrôles d'accès simples pour la protection de l'implémentation d'un type abstrait de données
- pour des problèmes de modélisation, même simples, la réponse est moins évidente
  malgré l'abondance et la complexité de l'imbrication des mécanismes proposés, on est assez vite limité.

Questions sur la modélisation

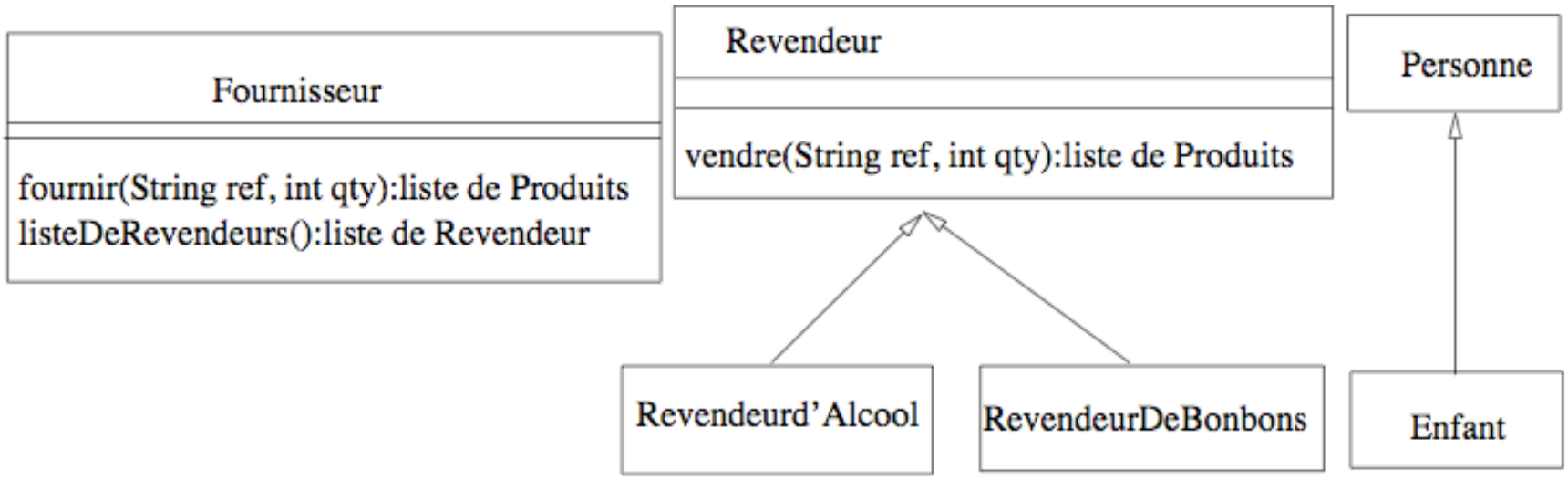

Figure : Un diagramme de classes

Questions sur la modélisation

- 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`.

Questions sur la modélisation

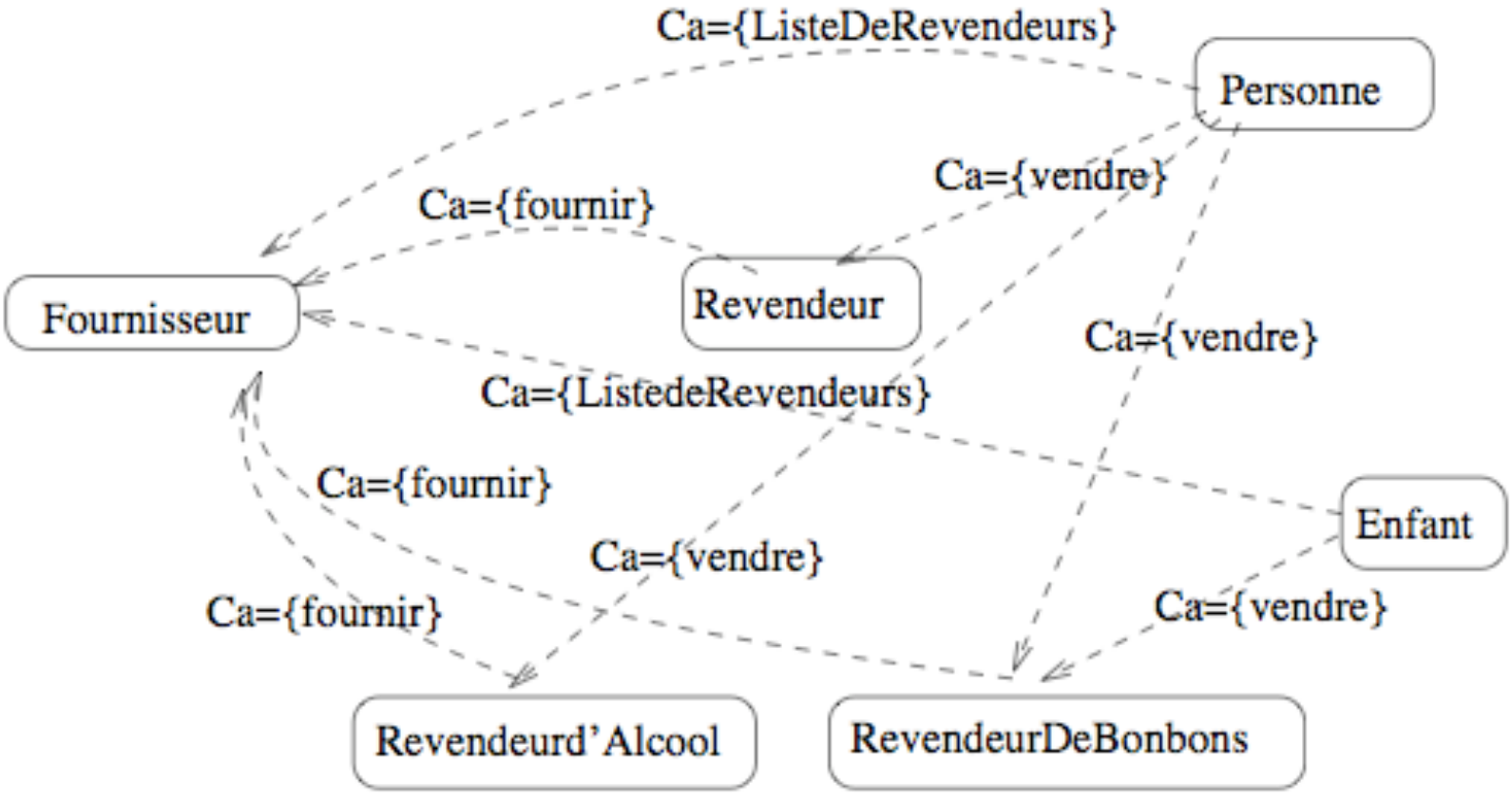

Figure : Les accès réclamés

Accès à la méthode vendre

Les accès peuvent être mis en place sans accès superflu comme suit :

- `vendre` est `protected`
- `Personne` est `friend` des trois classes de revendeur
- `Enfant` est `friend` seulement de `RevendeurDeBonbon`
- mais ce procédé laisse accès à toutes les autres méthodes et attributs des revendeurs s'il y en avait : cela ouvre trop !
- et ce n'est pas extensible

Accès aux méthodes `fournir` et `listeDeRevendeurs`

- ces méthodes ne sont pas accessibles à toutes les classes, elles ne peuvent être publiques
- si elles sont privées ou protégées (moins approprié), il faut que les classes qui ont besoin d'y accéder soient `friend` mais cette fois cela permet par exemple à `Personne` d'accéder à `fournir`, ce qui n'est pas prévu dans la spécification.