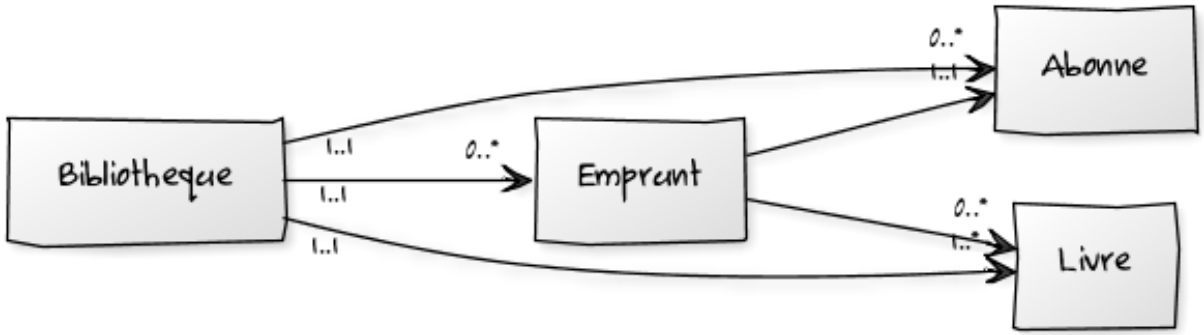
# Un premier exemple

## La gestion des emprunts d'une bibliothèque

Voici un schéma de classes pour la gestion des emprunts d'une bibliotheque :




### Le Modèle

- Dans une bibliothèque on a des livres et des abonnés.
- Une bibliothèque gère un ensemble d'emprunts.
- Un emprunt est une relation entre une bibliothèque, un abonné et un ou plusieurs livres.

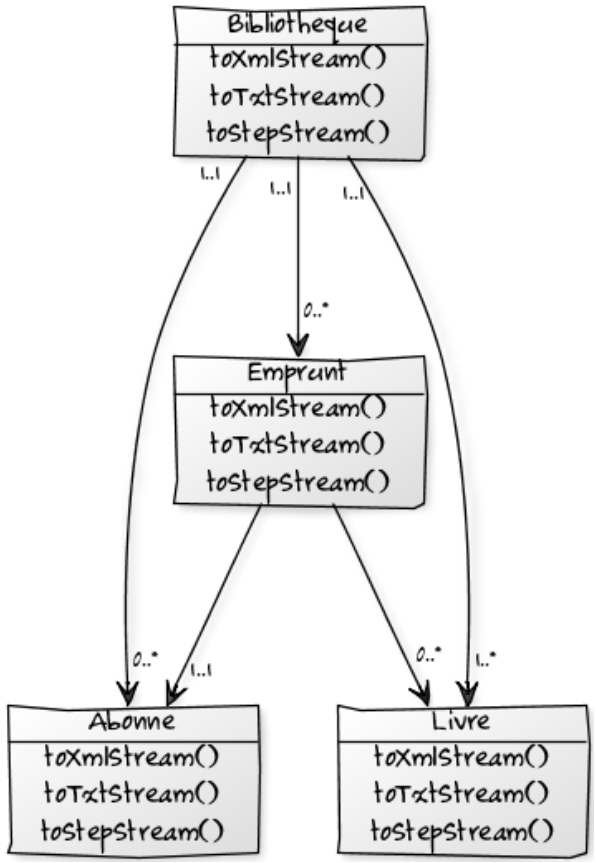
# Le besoin

Il faut pouvoir produire un flux XML à partir de toutes les instances de *Bibliotheque*. Une solution consiste à intégrer la méthode *#toXmlStream :* dans toutes les classes du modèle. Par exemple, dans la classe *Livre* :

```
1 Livre ≫toXmlStream : aStream
2   aStream nextPutAll : '<Livre>', String cr,
3     String tab, '<title>', self title, '</title>', String cr
4 ...
```

On a donc Livre>>toXmlStream:, Abonne>>toXmlStream: et Emprunt>>toXmlStream.
Mais on veut aussi pouvoir produire un flux en text clair et un autre pour le format d'échange STEP.
On aurait donc en plus : Livre>>toTxtStream:, Abonne>>toTxtStream: et Emprunt>>toTxtStream et Livre>>toStepStream:, Abonne>>toStepStream: et Emprunt>>toStepStream.

# Une première solution

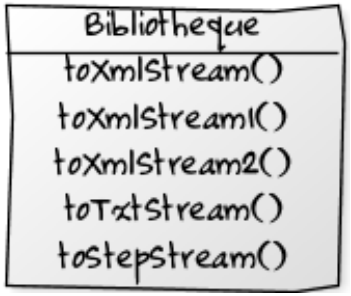



## Le Modèle avec les services de sérialisation

- Pour chaque export, chaque classe met en œuvre une fonction specifique ;

# Pour plusieurs formats XML differents

Si on veut adapter notre systeme pour plusieurs clients avec des exigences differentes concernant le Xml :

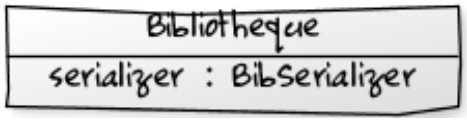



Evolutivité, maintenabilité et intelligibilté faibles :

- tout nouveau besoin implique une revue de toutes les classes qui exportent ;
- le code de la fonctionalité est disséminé dans toutes les classes ;
- il faut recompiler le tout et livrer une nouvelle version du système complet ;
- le code est pollué : les fonctions d'export sont implantées en dur, impossible de ne pas les avoir

# Une meilleure solution (1)

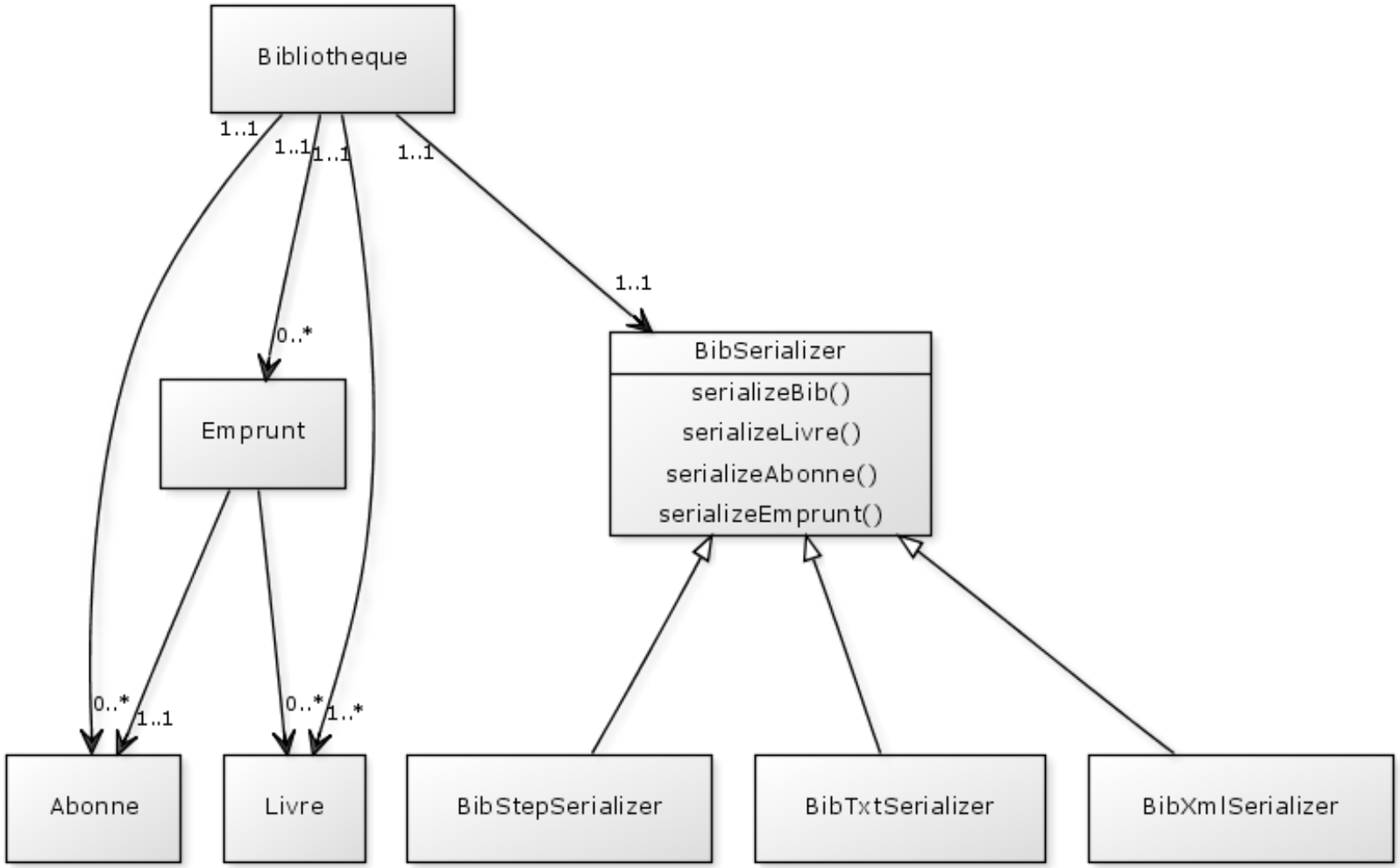
La fonctionalité d'export est bien identifiée mais peut varier. On peut utiliser la délégation pour rendre le service sans encombrer le code du modele : les fonctions d'export sont mises en oeuvre dans une classe dédiée au service d'export.




Voici le code de sérialisation qu'on peut implanter :

```
1  BibXmlSerializer≫serializeBib : aBibliotheque
2    aStream nextPutAll : '<Bibliotheque>', String cr,
3      String tab, '<nom>',aBibliotheque name, '</nom>',String cr
4    ...
5    aBibliotheque livres do : [ :l | self serializeLivre : l ].
6
7  BibXmlSerializer≫serializeLivre : aLivre
8    aStream nextPutAll : '<Livre>' ...
```

# Une meilleure solution (2)



# Mais pas si bonne que ça ...

La methode **serializeBib:** fixe la serialisation. Si une sous classe de Livre est introduite, elle ne peut pas être prise en compte. Une telle solution conduira à un code du style :

```
1  BibSerializer≫serializeBib : aBibliotheque
2    aStream nextPutAll : '<Bibliotheque>', String cr,
3      String tab, '<nom>',aBibliotheque name, '</nom>',String cr
4    ...
5    aBibliotheque livres
6      do : [ :l | (l isKindOf : LivreEnfant)
7        ifTrue : [self serializeLivreEnfant : l]
8        ifFalse : [(l isKindOf : LivreAdulte)
9          ifTrue : [self serializeLivreAdulte : l]
10         ifFalse : [self serializeLivre : l]]]
```

- ce code ne résiste pas à l'introduction de nouvelles sous-classes ;

# Solution avec Visiteur

Le visiteur s'implante en deux étapes :

- Etape 1 : Instrumentation du modèle ; Chaque classe du modèle met en oeuvre une fonction accept: qui prend un sérialiseur en argument.
  - Le code de la fonction accept: est toujours le même (modulo le nom de la classe)
  - Le code est implanté dans **toutes** les classes visitables du modèle.
- Etape 2 : Implementation du sérialiseur : une hiérarchie de classe

# Visiteur : Etape 1

```
1 Biliotheque≫accept : unSerialiseur
2   unSerialiseur visitBibliotheque : self
3
4 Livre≫accept : unSerialiseur
5   unSerialiseur visitLivre : self
6
7 Abonne≫accept : unSerialiseur
8   unSerialiseur visitAbonne : self
```

# Visiteur : Etape 1

```
1  class Biliotheque {
2    void accept(Visitor unSerialiseur) {
3      unSerialiseur.visitBibliotheque(this);
4    }
5  }
6  class Livre {
7    void accept(Visitor unSerialiseur) {
8      unSerialiseur.visitLivre(this);
9    }
10 }
11 class Abonne {
12   void accept(Visitor unSerialiseur) {
13     unSerialiseur.visitAbonne(this);
14   }
15 }
```

# Visiteur : Etape 2

```
1  XmlSerialiseur≫visitBibliotheque : uneBib
2    stream nextPutAll : '<Bibliotheque>', String cr,
3      String tab, '<nom>',uneBib name, '</nom>',String cr
4    ...
5    uneBib livres do : [ :l | l accept : self ].
6
7  XmlSerialiseur≫visitLivre : unLivre
8    stream nextPutAll : '<Livre>' ...
9    ...
```

Le sérialiseur à une variable d'instance stream dans lequel est accumulé le code Xml. On peut avoir un Sérialiseur abstrait dont hérite XmlSerialiseur. On peut ainsi développer autant de version de sérialiseurs que nécessaire.

# Visiteur : Etape 2

```
1  class XmlSerialiseur {
2    Stream stream = new ...
3
4    void visitBibliotheque(Bibliotheque b) {
5      stream.nextPutAll('<Bibliotheque nom="'+b.name()+'">\n';
6      for (Livre l  : uneBib.livres()) {
7        l.accept(self);
8      }
9    }
10
11   void visitLivre( Livre unLivre) {
12     stream.nextPutAll('<Livre>') ...
13   }
14 }
```

# Utilisation du sérialiseur-visiteur

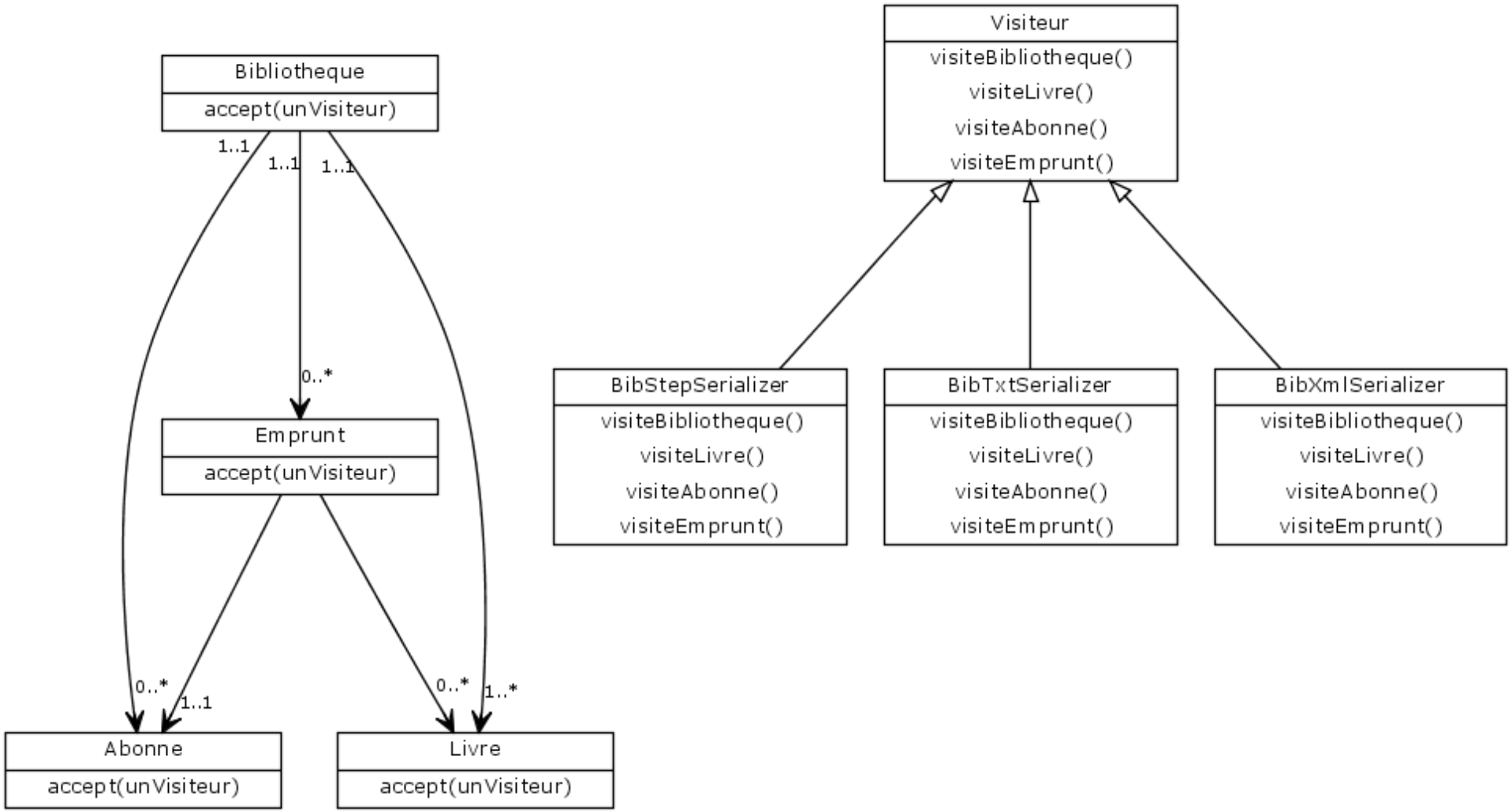
Le code suivant permet d'invoquer la sérialisation :

```
| bib serialiseur |
bib := Bibliotheque fromDataBase. "On recupere une bibliotheque"
serialiseur := XmlSerialiseur new.
bib accept : serialiseur.
" on recupere et affiche le resultat "
Transcript show : serialiseur stream contents.
```

# Utilisation du sérialiseur-visiteur

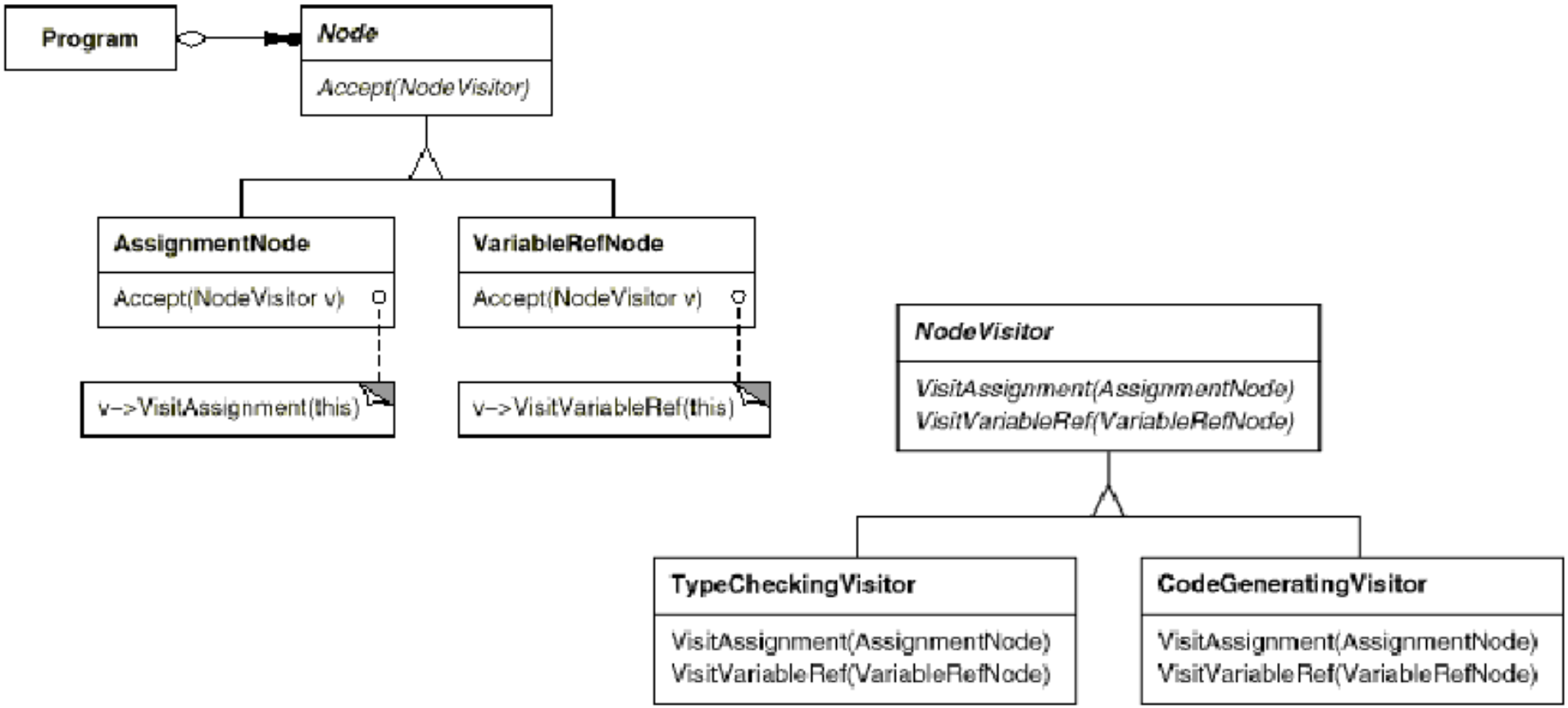
```
1  testSerialiseur () {
2    Bibliotheque bib;
3    Serialiseur serialiseur;
4    bib = new Bibliotheque();
5    bib.fromDataBase()."On recupere une bibliotheque"
6    serialiseur = new XmlSerialiseur();
7    bib.accept(serialiseur);
8    " on recupere et affiche le resultat "
9    System.out.println(serialiseur.contents());
10 }
```

# Le modele complet avec le visiteur



# Un autre exemple de visiteur

Voici celui du GOF. Les classes d'un arbre syntaxique peuvent utiliser un visiteur pour la compilation, la vérification, la génération de code ...



# Quelques remarques

- Le visiteur est utilisable pour tout ce qui nécessite la visite d'un graphe ou d'un arbre d'objet.
- Sérialisation donc mais aussi, vérification, calcul, transformation etc.
- La méthode accept: ou accept(Visitor) de chaque classe du modèle prend en argument un visiteur (et non un sérialiser).

# Quelques remarques

- Dans le visiteur, on a des méthodes visitXXXX : pour les classes concrètes du modèle ;
- On peut éventuellement ajouter des méthodes pour les classes abstraites pour factoriser du code ;
  - mais on doit les invoquer explicitement (on ne bénéficie plus de l'héritage)

```
1  XmlSerialiser≫visitLivre : unLivre
2    stream nextPutAll : '<Livre title="', unLivre title, '"/>'
3
4  XmlSerialiser≫visitLivreEnfant : unLivreEnfant
5    | age |
6    stream nextPutAll : '<LivreEnfant title="', unLivre title, '"'.
7    age := unLivreEnfant age printString.
8    stream nextPutAll : ' ageConseille="', age, '"/>'
```

**Pour la mise en oeuvre de *visitLivreEnfant* : il est difficile de réutiliser le code de *visitLivre* alors que la classe *LivreEnfant* est une sous-classe de *Livre*.**

# Avantages du visiteur

- L'ajout de nouvelles fonctionnalités sans toucher au code du modèle de base est facilité,
  - on peut ignorer certaines fonctionnalités et en ajouter en respectant l'OCP ;
- Un visiteur regroupe toutes les opérations d'une fonctionnalité ;
  - Les classes du modèles ne sont pas "encrassées" avec du code et des états propres à des fonctionnalités particulières.

# Inconvénients du visiteur

- Le visiteur convient quand le modèle est stable : ajouter une nouvelle classe au modèle peut être pénible
- Le visiteur casse le lien d'héritage : il faut exécuter explicitement les méthodes
- Le visiteur nécessite de disposer des accesseurs en publique du coté modèle ce qui n'est pas toujours souhaitable.
- Le fonctionnement est quelques fois difficile à suivre.
- La réutilisation est quelque fois plus lourde car on ne bénéficie plus de l'héritage pour le modèle.

# Exercice 1

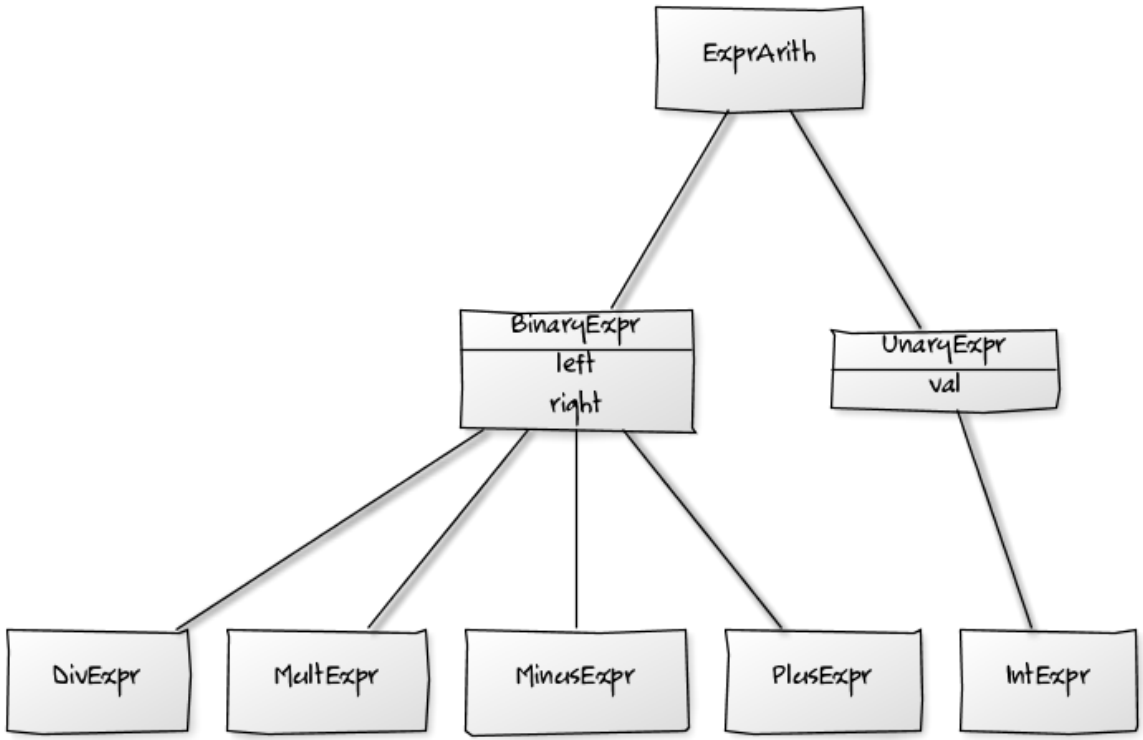
- Programmez l'exemple du cours, pour un carnet de contacts simple :
  - un export vers HTML et
  - un export vers XML ;
- Ecrivez des tests pour *sUnit* :

```
1  CarnetWriterTest≫setUp
2    carn  ≔ Carnet new.
3    carn addContact : (Contact name : 'Yolo' tel : '999999').
4
5  CarnetWriterTest≫testXmlWriter
6    | xmlWriter |
7    xmlWriter  ≔ CarnetXmlWriter new.
8    carn accept : xmlWriter.
9    self assert : (carn result =
10 '<Carnet>
11   <contact name="yolo" tel="999999"/>
12 </Carnet>')
```

# Exercice 2

Programmation d'une calculatrice : développement du modèle

- Programmez l'exemple de la calculatrice suivant :
  - on ne s'occupe que des opérations +, -, * et / ;
  - programmez un modèle pour représenter ces opérations ;
  - à l'aide d'un visiteur, développer un pretty-printer et un évaluateur d'opération arithématique.



# Exercice 2

## Développement du modèle

une instance de ce modèle représente une opération ; par exemple :

```
| e1 |
e1 := PlusExpr left: 5 right: 6. "Pour 5 + 6"
e2 := MultExpr left: e1 right: 4. "Pour (5 + 6) * 4"
e3 := DivExpr left: e2 right: 2. "Pour ((5 + 6) * 4) /2"
```

# Exercice 2

## Le pretty-printer

- programmez le visiteur *ExprPrettyPrinter* qui écrit la représentation d'une expression dans un String :

```
| expr |
expr := PlusExpr left: 5 right: 6.
calc := ExprPrettyPrinter new.
expr accept: calc.
Transcript show: calc result printString " '5 + 6' "
```

- n'oubliez pas vos tests.

# Exercice 2

## L'évaluateur

- programmez le visiteur *ExprCalculator* qui calcule le résultat d'une opération :

```
| expr printer calc |
expr  := PlusExpr left : 5 right : 6.
calc  := ExprCalculator new.
expr accept : calc.
printer  := ExprPrettyPrinter new.
expr result accept : printer.
Transcript show : printer result printString " '11' "
```

- n'oubliez pas vos tests.