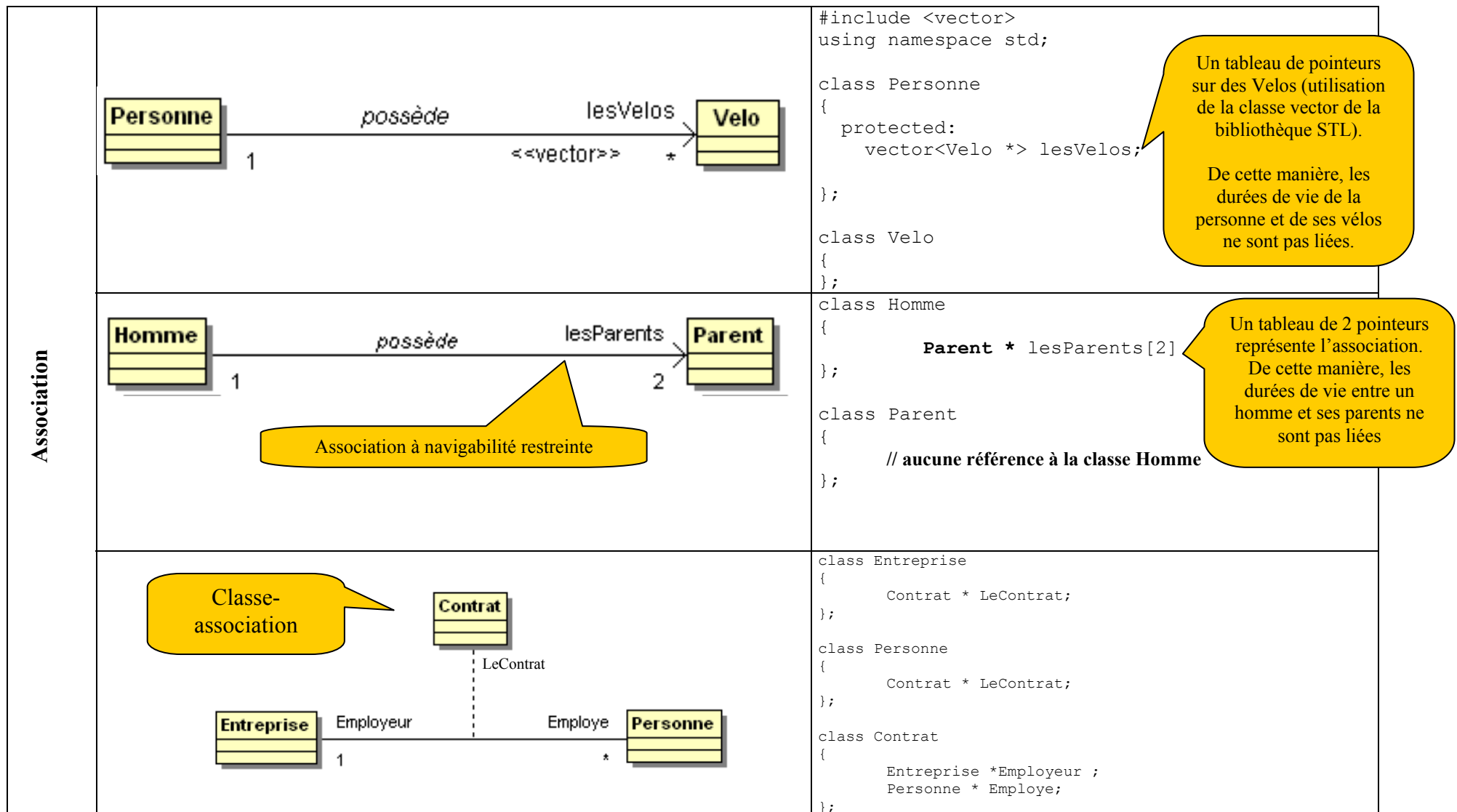


CODAGE DES RELATIONS ENTRE CLASSES EN C++

Association

- On utilise l'association quand deux classes sont liées, sans notion de propriété de l'une par rapport à l'autre.
- Les durées de vie ne sont pas forcément liées (la mort d'une instance de la classe A ne provoque pas forcément la mort des objets associés de la classe B)

	UML	C++
Association		<pre>class Homme { protected: Femme * epouse; }; class Femme { protected: Homme * epoux; };</pre>
		<pre>class Personne { Velo ** LeVelo; }; class Velo { // aucune référence à la classe Personne };</pre> <p>** pour faire référence à un tableau de pointeurs dont le nombre n'est pas connu. Bien sûr, il faut gérer soit même les allocations dynamiques de mémoire.</p>



AGREGATION

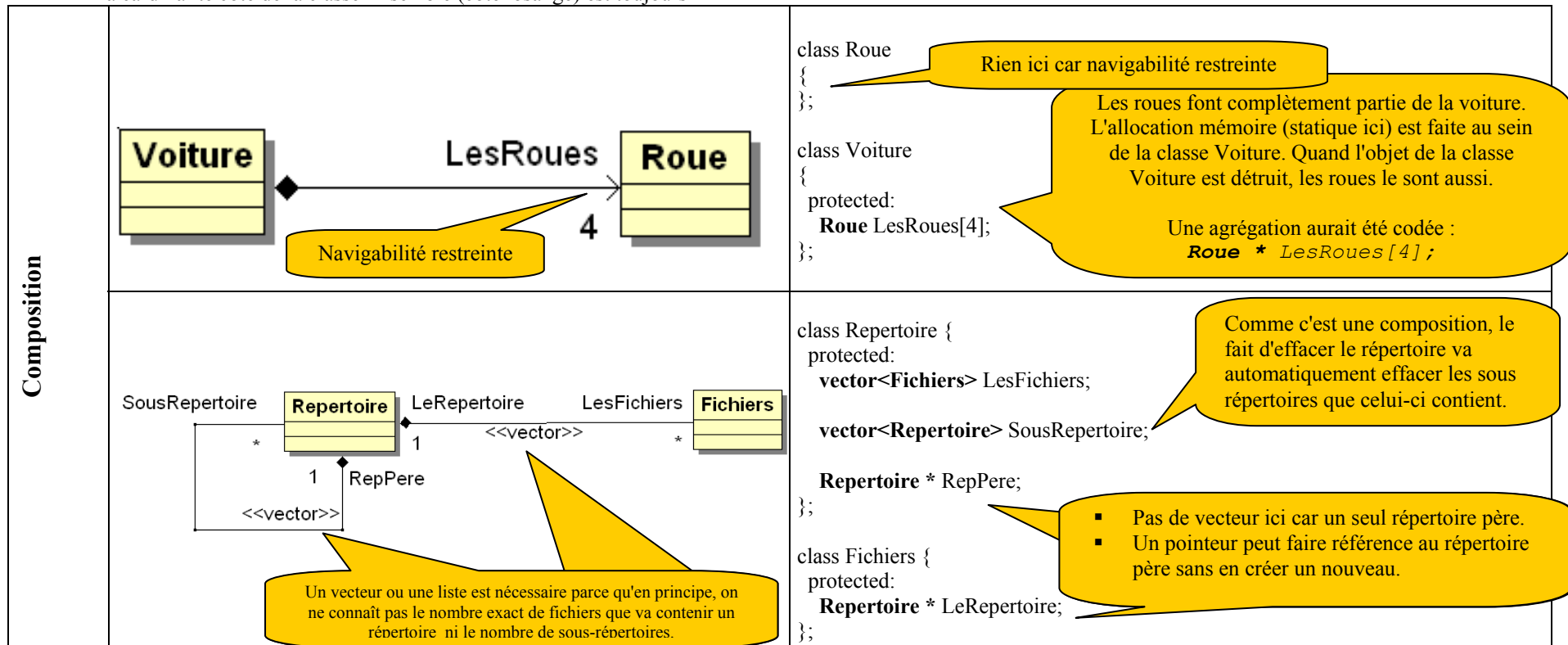
- On utilise une agrégation quand on peut utiliser le verbe « comporte » (Ensemble ---- Sous ensembles)
- L'agrégation est modélisée par un losange vide (du côté de la classe Ensemble)
- Les durées de vie entre les classes associées ne sont pas forcément liées !

Agrégation		<pre>class Immeuble { protected: Proprietaire ** LesProprios; }; class Proprietaire { protected: Immeuble ** L_immeuble; };</pre>	<p>** pour faire référence à un tableau de pointeurs dont le nombre n'est pas connu. Bien sûr, il faut gérer soit même les allocations dynamiques de mémoire.</p>
		<pre>class Immeuble { protected: list<Proprietaire *> LesProprios; }; class Proprietaire { protected: list<Immeuble *> L_immeuble; };</pre>	<p>Deux niveaux de pointeurs car on peut être propriétaire de plusieurs immeubles. Si on avait limité à 1 le nombre d'immeubles, on aurait eu un seul niveau de pointeurs</p> <p>Pointeurs car AGREGATION (idem si ASSOCIATION)</p>

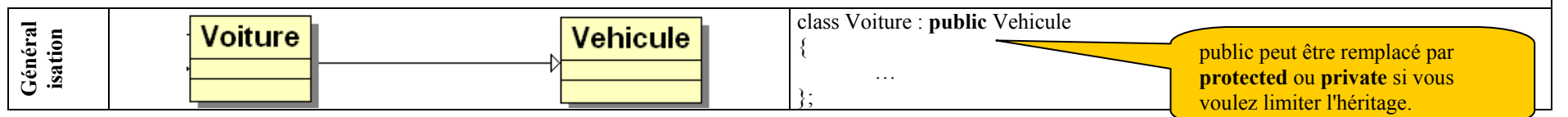
Stockage des propriétaires et des immeubles dans des listes (vous pouvez aussi utiliser des vecteurs si l'ordre de rangement n'a pas d'importance). Cette solution à base de vecteurs ou de listes est à préconiser quand le nombre de propriétaires ou d'immeubles n'est pas connu à priori

Composition : agrégation particulière où les parties sont complètement intégrées à l'ensemble

- On utilise une composition quand on peut utiliser le verbe « est composé de »
- La composition est modélisée par un losange noirci.
- Les durées de vie entre les classes associées sont liées (la mort d'un objet de la classe Ensemble provoque la mort des objets composites)
- La cardinalité côté de la classe Ensemble (côté losange) est toujours 1

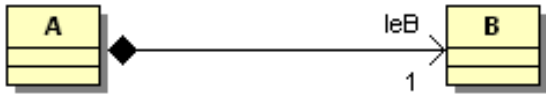


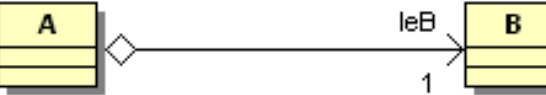
Généralisation

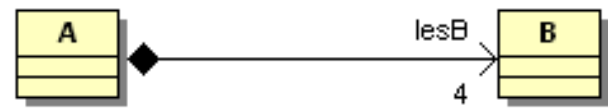
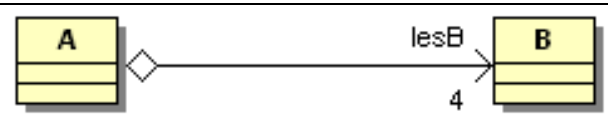


REVENONS SUR LES DIFFERENCES ENTRE AGREGATION ET COMPOSITION

Le fait de dire : « Une agrégation est codée par un ou plusieurs pointeurs d'objets » et « une composition est codée par un ou plusieurs objets » n'est pas systématique. Tout dépend en fait de l'endroit où sont créés les objets composites. Dans le cas où ceux-ci sont créés dans le fichier de déclaration(.h) , on parlera de composition forte, sinon on parlera de composition applicative.

	Un objet de la classe A est « composé » d'un objet de la classe B nommé « leB » ⇒ L'objet « leB » doit être créé et détruit dans A (soit dans le fichier de déclaration, soit dans le fichier de définition).	
<u>Solution de codage 1 : COMPOSITION FORTE</u> Cette solution est valable si le constructeur de B n'a pas d'argument ou s'il a des arguments par défaut.	A.h <pre>class A { B leB ; };</pre>	A.cpp
<u>Solution de codage 2 : COMPOSITION APPLICATIVE</u> Cette solution est toujours valable. « leB » est déclaré en tant que pointeur, mais il est créé dans le constructeur de A et la destruction d'un objet de la classe A provoque la destruction de « leB »	A.h <pre>class A { B *leB ; };</pre>	A.cpp <pre>A :: A(.....) // le constructeur { leB = new B(....) ; } A::~~A() // le destructeur { delete leB ; }</pre>

	Un objet de la classe A « comporte » un objet de la classe B nommé « leB » ⇒ L'objet « leB » ne doit pas être créé dans A.	
« leB » est déclaré en tant que pointeur, et le constructeur de A fait référence à un pointeur défini ailleurs. La destruction d'un objet de la classe A ne provoquera pas la destruction de « leB ».	A.h <pre>class A { B *leB ; }</pre>	A.cpp <pre>A :: A(B * refB) // le constructeur { leB = refB ; }</pre>

	Un objet de la classe A est « composé » de 4 objets de la classe B nommés « lesB » ⇒ Les objets « lesB » doivent être créés et détruits dans A (soit dans le fichier de déclaration, soit dans le fichier de définition)	
<u>Solution de codage 1 : COMPOSITION FORTE</u> Cette solution est valable si le constructeur de B n'a pas d'argument ou s'il a des arguments par défaut.	A.h <pre>class A { B lesB[4]; };</pre>	A.cpp
<u>Solution de codage 2 : COMPOSITION APPLICATIVE</u> Cette solution est toujours valable. « lesB » sont déclarés en tant que pointeur, mais il sont créés dans le constructeur de A et la destruction d'un objet de la classe A provoque la destruction de « lesB »	A.h <pre>class A { B *lesB[4]; };</pre>	A.cpp <pre>A :: A(.....) // le constructeur { lesB[0] = new B(...); lesB[1] = new B(...); lesB[2] = new B(...); lesB[3] = new B(...); } A :: ~A() // le destructeur { for (int i=0 ; i<4 ; i++) delete lesB[i]; }</pre>
	Un objet de la classe A « comporte » 4 objets de la classe B nommés « lesB » ⇒ Les objets « lesB » ne doivent pas être créés et effacés dans A.	
« lesB » sont déclarés en tant que pointeur, et le constructeur de A fait référence à 4 pointeurs définis ailleurs. La destruction d'un objet de la classe A ne provoquera pas la destruction de « lesB ».	A.h <pre>class A { B *lesB[4]; }</pre>	A.cpp <pre>A :: A(B * refB[4]) // le constructeur { for (int i=0 ; i<4 ; i++) lesB[i] = refB[i]; }</pre>