

# Une syntaxe pour l'agrégation

Philippe PRADOS

[pp@philippe.prados.name](mailto:pp@philippe.prados.name)



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

## TABLE DES MATIERES

1.	Relation ou agrégation .....	3
1.1	Relation .....	4
1.2	Agrégation .....	5
2.	Que proposent C++ et Java ? .....	8
2.1	C++ .....	8
2.2	Java .....	9
2.3	Quelle sémantique pour les paramètres ? .....	9
2.4	Tracer les agrégations .....	10
3.	Quand utiliser l'agrégation ? .....	10
3.1	Les variables locales .....	10
3.2	Les variables globales .....	11
3.3	Variables de l'OS .....	11
4.	Comment proposer l'agrégation ? .....	12
4.1	Un pointeur d'agrégation .....	12
4.2	Les affectations .....	12
4.3	La comparaison de référence .....	14
4.4	La construction d'une instance .....	14
4.5	Le passage de paramètres .....	14
4.6	Le retour de pointeur d'agrégation .....	15
4.7	Les tableaux .....	16
4.8	Les conversions .....	16
4.9	Les exceptions .....	17
4.10	Optimisation .....	17
5.	Les erreurs mémoires classiques .....	19
6.	Implantation .....	19
6.1	C++ .....	19
6.1.1	clone .....	19
6.1.2	Traductions des syntaxes .....	20
6.1.3	Méthodes constantes .....	21
6.1.4	Préprocesseur .....	22
6.2	Java .....	22
6.2.1	clone .....	22
6.2.2	Traductions des syntaxes .....	23
6.2.3	Préprocesseur .....	24
7.	Cas d'utilisation .....	24
7.1	Liste chaînée .....	24
7.1.1	C++ .....	24
7.1.2	Java .....	25
7.2	Arbre .....	27
7.2.1	C++ .....	27
7.2.2	Java .....	29
8.	Agrégation partagée .....	30
8.1	C++ .....	30
8.2	Java .....	31
9.	Instances immuables .....	31
9.1	C++ .....	31
9.2	Java .....	31
10.	Améliorations des outils .....	32
10.1	Object Request Broker .....	32
10.2	Base de données objets .....	32
11.	Pourquoi le ramasse-miettes n'est pas la solution ? .....	32
12.	Syntaxe .....	33
13.	Critiques .....	33
13.1	Pourquoi ne pas utiliser les « deepCopy() » et les « shallowCopy() » de Smalltalk ? .....	33
13.2	Cela complexifie le langage .....	33
13.3	Les comités de normalisation n'accepteront jamais cela .....	34
13.4	L'agrégation n'existe pas dans le monde réel .....	34
14.	Conclusion .....	34

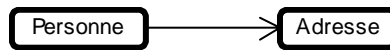
## Avant-propos

L'agrégation est un concept absent des langages de programmation alors qu'il est utilisé depuis des années avec plus ou moins de réussite. La plupart des erreurs mémoire pourraient être corrigée si ce concept était présent dans les langages. Ce document explique pourquoi l'agrégation est nécessaire et propose des implémentations pour C++ et Java.

Nous allons étudier comment et pourquoi utiliser un pointeur d'agrégation. Les sources C++ et Java permettront d'éclairer le discours. Les lecteurs familiers avec l'un de ces langages pourront s'abstenir de lire les chapitres ou les codes spécifiques à l'autre langage. Certains codes sont *compatibles* avec les deux langages. Ils sont alors lisibles par les deux populations.

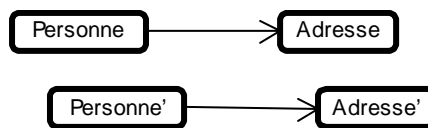
Soit une classe `Personne` dont un des attributs est une `Adresse` (Figure 1).

Figure 1



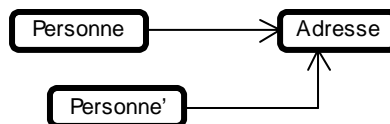
Si je désire dupliquer la personne, faut-il dupliquer son adresse ? (Figure 2)

Figure 2



Faut-il partager l'adresse ? (Figure 3)

Figure 3



Cette question est récurrente en développement objet. L'adresse est-elle une partie de la personne au même titre que son nom ?

Pour y répondre, le formaliste UML (Unified Method Language) propose la notion d'agrégation. L'agrégation permet d'indiquer que l'adresse appartient à la personne. C'est un élément indissociable de la personne au même titre que son nom ou que les autres attributs. L'adresse est référencée par un pointeur pour des raisons techniques imposé par le langage, non pour des raisons sémantiques. Il y a plusieurs façons d'écrire la classe `Personne`.

### C++

```

// V1.0
class Personne
{ string _nom;
  Adresse* _adresse;
};

// V1.1
class Personne
{ string _nom;
  Adresse _adresse;
}

// V2.0
class Personne
{ string _nom;
  string _adr1;
  string _adr2;
  string _adr3;
}
  
```

### Java

```

// V1.0
public class Personne
{ String _nom;
  Adresse _adresse;
}

// V2.0
public class Personne
{ string _nom;
  string _adr1;
  string _adr2;
  string _adr3;
}
  
```

Sémantiquement, la version 2 est équivalente à la version 1. La classe `Adresse` n'est plus nécessaire. Cela n'empêche pas le programme de fonctionner. L'adresse est bien une partie de la personne. Le pointeur de la version 1.0 ne représente pas un lien d'utilisation mais un lien de possession. La version 1.1 en C++ traduit bien cela. La personne agrège l'adresse comme elle agrège le nom de l'individu.

Un lien de possession traduit une « agrégation ».

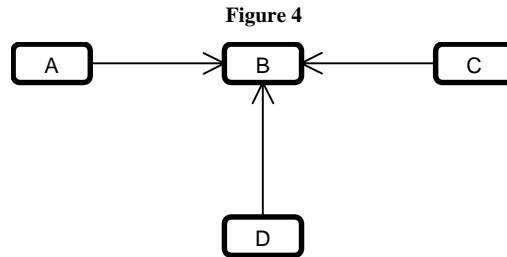
## 1. RELATION OU AGREGATION

Les programmes informatiques utilisent, souvent sans le savoir, deux types principaux de relations. Les relations simples et les agrégations. Regardons ce qui les différencie.

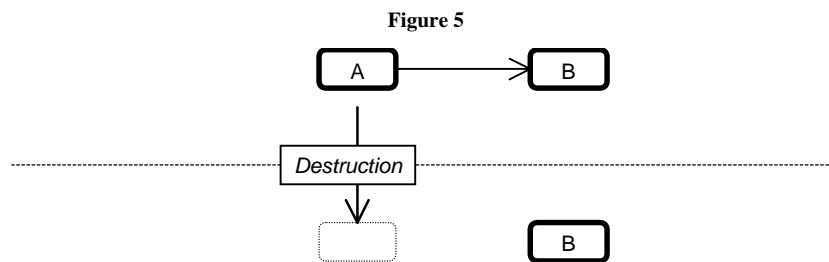
### 1.1 Relation

Une relation entre deux classes indique une dépendance entre deux ou plusieurs instances. En C++, cela se traduit par un pointeur. En Java, cela se traduit par une référence sur un objet.

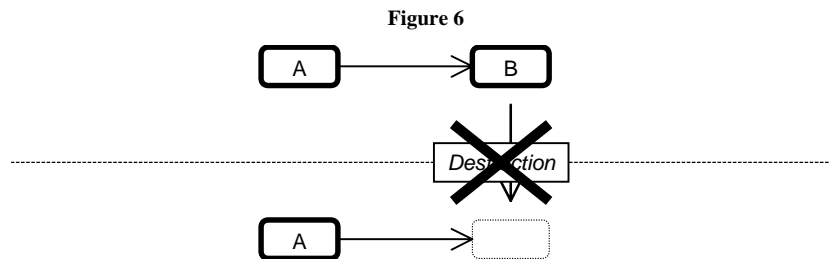
- Un objet pointé peut être partagé par plusieurs instances (Figure 4).



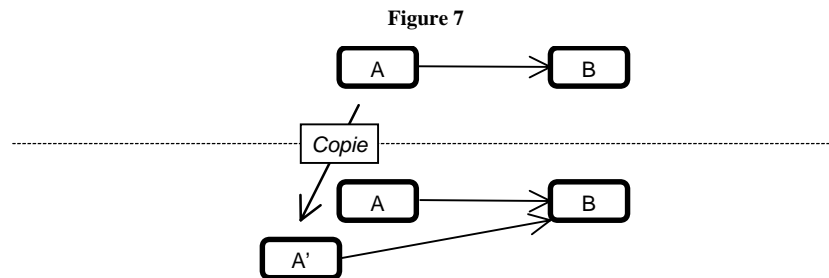
- La destruction de A n'entraîne pas la destruction de B (Figure 5).



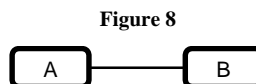
- Il est interdit de détruire l'objet B s'il existe des objets en relation avec lui. Sinon, l'objet A aurait un pointeur dans le vide (Figure 6).



- La copie de A n'entraîne pas la copie de B (Figure 7).



- Une relation est bidirectionnelle si les objets sont en relation entre eux (Figure 8).



En C++, cela se traduit par un pointeur dans chaque classe. En Java, cela se traduit par une référence dans chaque classe.

```

C++
class B;
class A
{ B* _ptb;

```

```

Java
class A
{ B _ptb;
}

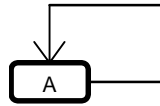
```

```
};
class B
{ A* _pta;
};
```

```
class B
{ A _pta;
}
```

Une classe peut avoir une relation sur un objet du même type, voir lui-même (Figure 9).

Figure 9



C++

```
class A
{ A* _pta;
};
```

Java

```
class A
{ A _pta;
}
```

La relation est également appelée « relation use ».

### 1.2 Agrégation

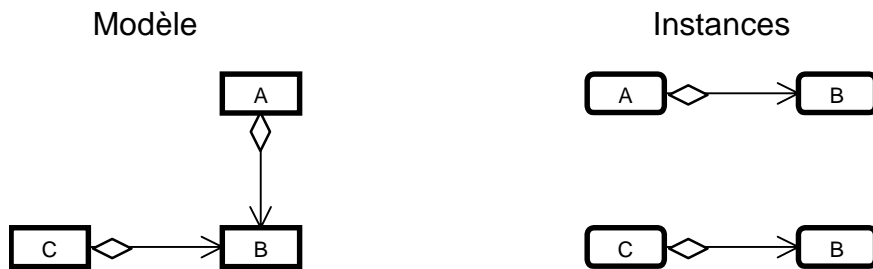
Une agrégation est une relation particulière. Un objet en agrégation n'existe que par la présence de l'objet propriétaire. L'agrégation ajoute sur la relation une information de durée de vie.

L'approche est similaire à l'utilisation des types primitifs. Si un objet possède un attribut de type `int`, celui-ci est détruit lorsque l'instance est détruite. Un pointeur d'agrégation propose une approche similaire, mais par l'intermédiaire d'un pointeur. Le pointeur est le représentant de l'instance pointé. Lorsque l'on perd un pointeur d'agrégation, l'instance référencée est également détruite. Si un objet en relation doit être détruit lors de la perte du pointeur, il s'agit d'une agrégation.

En formaliste UML (Unified Method Language), l'agrégation est représentée à l'aide d'un losange du côté de l'agrégeant.

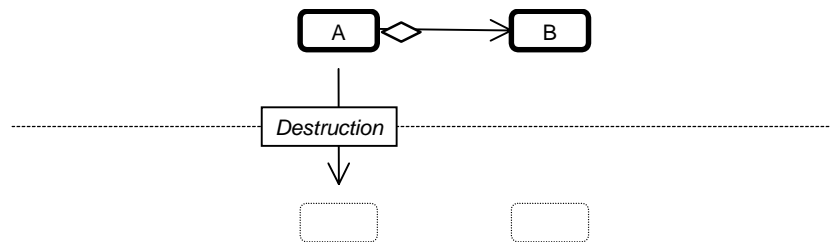
- Si B est agrégé par A et par C, la durée de vie de B dépend de la durée de vie de A ou de C (Figure 10).

Figure 10



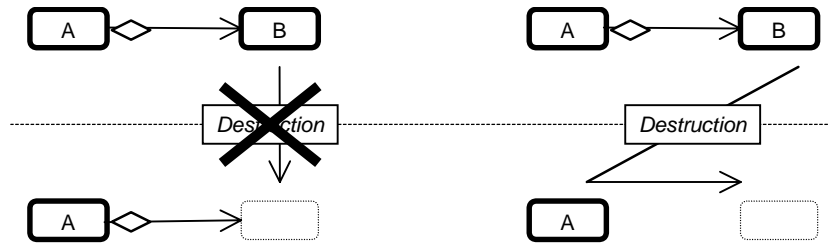
- La destruction de A entraîne la destruction de B (Figure 11).

Figure 11



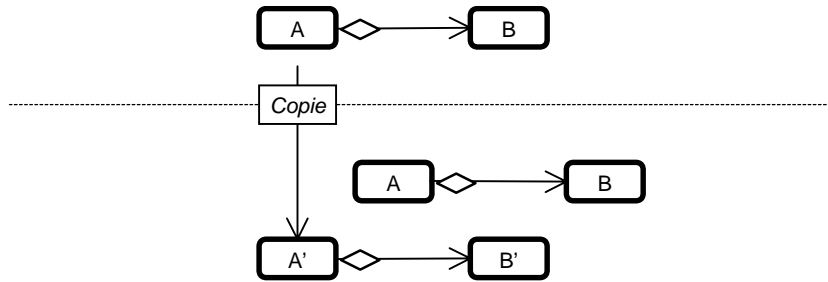
- Il n'est pas possible de détruire directement une agrégation. Il faut le demander à l'agrégeant (Figure 12).

Figure 12



- La copie de A entraîne la copie de B (Figure 13)

Figure 13

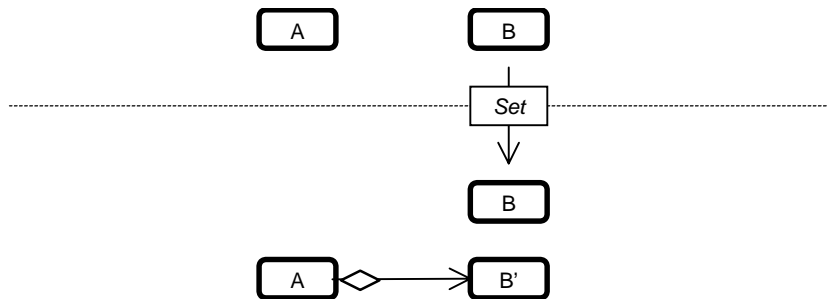


- Lorsque l'on valorise un attribut de type primitif, la valeur précédente est perdue. Elle est remplacée par une copie du paramètre.

```
int a=1;
int b=2;
a=b; // 1 est perdu
```

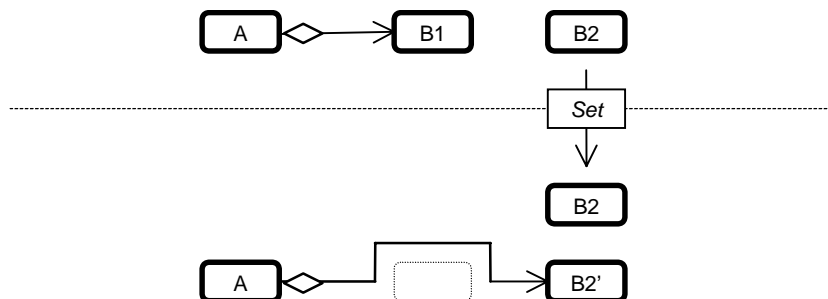
La sémantique est la même lors de la valorisation d'un pointeur d'agrégation. L'instance précédemment référencée est perdue, et l'agrégant garde une copie de l'objet reçu en paramètre (Figure 14).

Figure 14



- La valorisation d'un objet agrégé, détruit l'agrégation précédente (Figure 15).

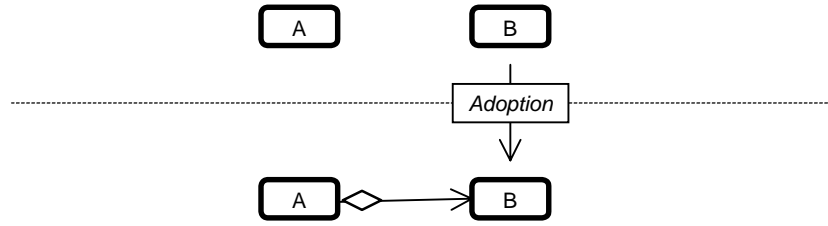
Figure 15



L'adoption d'un pointeur indique qu'un objet en devient responsable et se chargera de sa destruction. Un pointeur peut transiter de propriétaire en propriétaire par adoptions successives. Bien maîtriser l'adoption des pointeurs permet de gérer correctement les allocations mémoires.

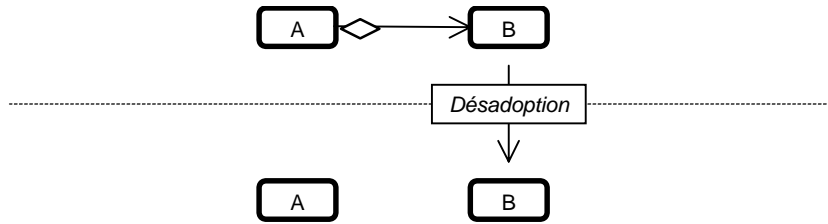
- L'adoption est un transfert de responsabilité sur la durée de vie d'un objet. Il n'est possible d'adopter un objet que s'il est libre, sans propriétaire (Figure 16).

Figure 16



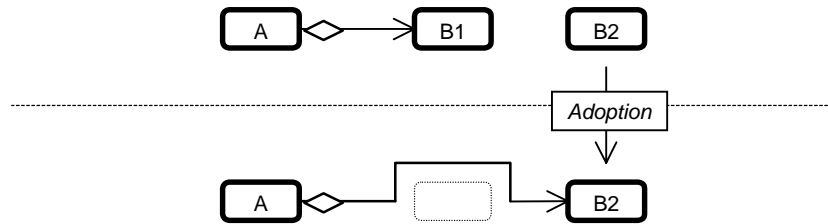
- La désadoption est un abandon de responsabilité (Figure 17)

Figure 17



- L'adoption détruit l'agrégation précédente (Figure 18).

Figure 18



L'agrégation est également appelée « relation *has* ». Un pointeur d'agrégation utilise une sémantique par valeur. L'instance référencée est dupliquée dans les mêmes situations que le serait un type primitif. De plus, un pointeur d'agrégation peut accepter la valeur `null`. Cela est similaire aux champs d'une table relationnelle acceptant la valeur `null`. Le champ peut ou non être valorisé.

Il existe de nombreux exemples d'agrégations dans l'informatique.

- Les composants OLE de Windows™ proposent la notion de « copier » et « copier avec liaison ». L'utilisateur manipule ces commandes dans le menu « Édition ». La première notion est une agrégation ; la deuxième est une relation. Lorsque vous copiez un morceau d'une feuille Excel dans un document Word, c'est le fichier Word qui possède le morceau Excel. La feuille Excel peut être supprimée du disque sans perturber le document Word. Lorsque vous copiez avec liaison, le document Word garde le nom du fichier Excel d'où la section est extraite. Les deux fichiers doivent être présents sur le disque.
- Les bases de données SQL proposent généralement un mécanisme de « *delete propagate* ». Cela permet, lors de l'effacement d'un enregistrement `Personne`, de détruire également l'enregistrement `Adresse` correspondant. La notion de « *delete propagate* » est une traduction de l'agrégation. La durée de vie de l'adresse dépend de la durée de vie de la personne. Si on détruit un enregistrement `Personne`, il faut également détruire la ligne correspondante dans la table `Adresse`.
- La gestion de fichier sous Windows propose également cette notion. Un répertoire possède des fichiers. Si on détruit un répertoire, on détruit également tous ces fichiers. La notion d'adoption est traduite par la commande `move`. On demande à un répertoire d'accueillir un fichier à la place d'un autre répertoire. La gestion de fichier sous Unix est différente. Un même fichier peut appartenir à différents répertoires (commande `ln`).
- Le format XML gère également ces notions. Un tag peut être inclus dans un autre ou être en relation avec un autre.

```
<Personne>
  <Nom>Philippe Prados</Nom>
  <Adresse>
    34, Place de l'église
  </Adresse>
  <Employeur ref="ibm"/>
</Personne>
<Entreprise id="ibm">
</Entreprise>
```

Dans cet exemple, le tag `<Personne>` possède le tag `<Nom>` et `<Adresse>`, et référence le tag `<Entreprise>` appelé `ibm`.

## 2. QUE PROPOSENT C++ ET JAVA ?

Regardons ce que propose C++ et Java pour manipuler les références.

### 2.1 C++

C++ considère que tout pointeur est une *relation* et non une *agrégation*. Cela apparaît lors des méthodes générées automatiquement par le compilateur. Lors de la rédaction d'une classe, le compilateur peut générer automatiquement un constructeur de copie, un opérateur d'affectation et un destructeur.

Si un pointeur est présent comme attribut, le pointeur est dupliqué lors de l'affectation ou du constructeur de copie, mais pas l'objet pointé. Lors de la destruction d'une instance, le pointeur est détruit mais pas l'objet pointé.

Ces choix traduisent une sémantique de relation pour tous les pointeurs. Par défaut, un pointeur est une relation.

La rédaction en C++ d'une agrégation peut se traduire de deux façons différentes. Soit en déclarant un objet dans un objet,

```
class A
{ //...
};
class B
{ A _agregation;
public:
  B(const A& agregation) // Copie de l'objet
  : _agregation(agregation)
  { }
  B(const B& x) // Constructeur de copie
  : _agregation(x._agregation)
  { }
};
```

soit en déclarant un pointeur, et en ajoutant dans le constructeur de copie la duplication de l'objet pointé et dans le destructeur de la classe agrégeant, la destruction de l'objet agrégé.

```
class A
{ //...
};
class B
{ A* _agregation;
public:
  B(A* agregation) // Adoption du pointeur
  : _agregation(agregation)
  { }
  B(const A& agregation) // Copie de l'objet
  : _agregation(agregation.clone())
  { }
  B(const B& x) // Constructeur de copie
  : _agregation(x._agregation->clone())
  { }
  void adopte(B* x)
  { delete _agregation;
    _agregation=x;
  }
  ~B()
  { delete _agregation; }
};
```

L'implantation par un pointeur permet de bénéficier du polymorphisme et de l'adoption.

La surcharge de l'opérateur de comparaison doit également tenir compte de l'agrégation.

```
class B
{ A* _agr;
  A* _rel;
  //...
  bool operator ==(const B& x) const
  { return ((*_agr == *x._agr) // Comparaison de l'agrégation
          && (_rel==obj._rel)); // Comparaison de la relation
  }
}
```

Il faut comparer la valeur des objets agrégés et la valeur des relations.

Une grande partie des erreurs mémoires est due à une mauvaise traduction de ce concept. Par exemple le destructeur détruit l'objet pointé, mais le constructeur de copie ne duplique pas l'objet pointé.

## 2.2 Java

La sémantique par relation apparaît avec Java lorsqu'une méthode implémente l'interface `Cloneable`. Dans ce cas, la méthode `clone()` hérité de la classe `Object`, duplique la valeur des pointeurs mais pas les objets pointés. Il faut réécrire cette méthode pour traduire l'agrégation. Il faut explicitement appeler la méthode `clone()` de l'objet agrégé.

```
class A implements Cloneable
{ public Object clone()
  { try
    { return super.clone();
    } catch (CloneNotSupportedException x)
    { return null; }
  }
}

class B
{ A _agr=new A();
  B(B x) // Constructeur de copie
  { _agr=(A)x._agr.clone();
  }
  public Object clone()
  { return new B(this);
  }
}
```

La destruction de l'agrégat est gérée implicitement par le ramasse-miettes. Lors de la destruction de `B`, l'instance `A` agrégé est également détruite, si et seulement si, il n'existe pas d'autre pointeur sur l'instance. Il ne devrait jamais y avoir d'autres pointeurs référençant l'instance possédée par `B` lorsqu'il est détruit. Ce bogue est très difficile à identifier.

Il faut également modifier la méthode `equals()` de la classe `Object` pour traduire une agrégation. Il faut comparer les valeurs des attributs et comparer la valeur des objets agrégés.

```
class B implements Cloneable
{ A _agr=new A();
  A _rel;
  //...
  boolean equals(Object x)
  { B obj=(B)x;
    return ((_agr.equals(obj._agr)) // Comparaison de l'agrégation
    && (_rel==obj._rel)); // Comparaison de la relation
  }
}
```

Les tableaux Java proposent également une sémantique par référence. La méthode `clone()` des tableaux permet d'obtenir un nouveau tableau dont les références pointent sur les mêmes instances.

```
{ Object[] tab1=new Object[2];
  tab[1]=new Object();
  Object[] tab2=(Object[])tab1.clone();
  // tab1[0]==tab2[0] est vrai
}
```

RMI (Remote Method Invocation) propose une approche particulière de l'agrégation. Toutes les instances d'une classe qui implémente l'interface `java.rmi.Remote` proposent automatiquement un usage par relation. Les instances ne transitent pas sur le réseau. Un client peut alors invoquer l'instance sur le serveur. Pour toutes les autres instances, RMI utilise un passage par valeur. Cela correspond à une agrégation. RMI propose une agrégation par défaut, et une relation pour certaines classes. Il n'est pas possible d'avoir des instances utilisables parfois via une relation, parfois via une agrégation. Une instance proposant l'interface `java.rmi.Remote` reste toujours présente sur la même machine. Il n'est pas possible de l'envoyer sur le client par valeur.

## 2.3 Quelle sémantique pour les paramètres ?

Que se soit avec le C++ ou Java, comment connaître la sémantique d'un paramètre de type pointeur ?

C++

```
class A
{ void m(B* pt)
  { ... }
};
```

Que fera la méthode `m()` du paramètre `pt` ? Elle peut

- Uniquement manipuler l'objet référencé,
- dupliquer l'objet référencé
- adopter l'objet référencé

Suivant les cas, l'appelant devra respecter certains principes. Par exemple, si la méthode `m()` ne fait que consulter l'objet référencé ou si elle duplique le paramètre, l'appelant peut continuer à utiliser le paramètre.

Java

```
class A
{ void m(B pt)
  { ... }
}
```

**C++**

```
pta->m(ptb);
ptb->autre_methode();
```

Si la méthode `m()` duplique le paramètre, l'appelant est garanti qu'il n'y aura pas d'effet de bord sur l'instance pointée par `ptb`. La méthode `m()` ne modifiera pas cette instance.

Si la méthode `m()` adopte le paramètre, l'appelant ne peut plus utiliser le pointeur.

**C++**

```
pta->m(ptb);
// ptb ne doit plus être utilisé
```

L'usage qu'un client d'une méthode peut faire d'un objet référencé dépend de choix de conception qui ne sont généralement pas documentés. Cette information n'est pas présente dans la syntaxe des langages. De nombreux bogues sont dus à une méconnaissance de la sémantique d'un pointeur. Comment contrôler le code du client ?

De même, si une méthode retourne un pointeur, que doit-on en faire ? L'appelant devient responsable de l'instance retournée (cas d'une méthode de construction) ? Il ne peut que consulter l'objet ?

L'expressivité de C++ ou de Java est insuffisante. Il faut offrir une syntaxe enrichie pour documenter correctement les protocoles d'appels et vérifier cela lors de la compilation.

## 2.4 Tracer les agrégations

Lorsque l'on désire tracer une instance, on affiche la valeur de chaque attribut. Si l'attribut est une agrégation, on demande à l'instance référencée de s'afficher.

Si par contre, l'attribut est une relation, il ne faut pas demander à l'attribut de s'afficher car on risque une récursivité. Il faut afficher une information identifiant la relation, mais pas l'instance en relation. Par exemple, la classe `Employe` agrège une `Adresse` et possède une relation avec son employeur.

**C++**

```
class Employe
{ Adresse* _agr;
  Entreprise* _rel;
  void trace(ostream& o)
  { o << "_agr=" << *_agr << endl
    << "_rel=" << _rel->getNom()
    << endl;
  }
}
```

Il faut afficher l'adresse de l'employé, et le nom de l'entreprise. En effet, si l'entreprise agrège ses employés, elle va demander l'affichage de chacun. Il ne faut pas que l'employé demande l'affichage de l'entreprise sinon le cercle infernal commence. L'employé affiche l'entreprise, qui affiche l'employé, qui affiche l'entreprise,...

**Java**

```
pta.m(ptb);
ptb.autre_methode();
```

**Java**

```
pta.m(ptb);
// ptb ne doit plus être utilisé
```

**Java**

```
class Employe
{ Adresse _agr;
  Entreprise _rel;
  String toString()
  { return "_agr=" + _agr + '\n'
    + "_rel=" + _rel.getNom();
  }
}
```

## 3. QUAND UTILISER L'AGREGATION ?

Au risque de surprendre, tout objet est agrégé par un autre. L'agrégation est utilisée tous les temps et pourtant, il n'existe pas de langage proposant une traduction de ce concept. Pour démontrer cela, regardons les différentes situations présentes dans un programme.

### 3.1 Les variables locales

Une variable locale, déclarée dans une fonction, sera détruite lorsque celle-ci sera terminée. On peut considérer les variables créées dans les fonctions comme des agrégations de celles-ci. Par exemple, la fonction suivante :

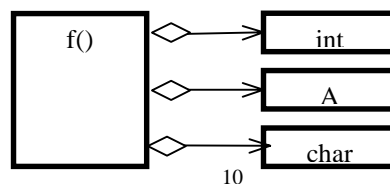
**C++**

```
static void f()
{ int i;
  A a;
  char buf[10];
  // Traitement...
}
```

**Java**

```
static void f()
{ int i;
  A a=new A();
  char[] buf=new char[10];
  // Traitement...
}
```

peut-être schématisée comme ci-dessous :



Sortir de la fonction permet de détruire les variables locales de celle-ci. En quelque sorte, appeler une fonction équivaut à créer une *instance d'exécution*. Lors de la destruction d'une instance d'exécution, les attributs (les variables locales) sont détruits. Il est possible de traduire cette fonction par une classe équivalente :

<pre>C++ class f { int i;   A a;   char buf[10]; public:   void f()   { // Traitement     // sans variable locale   } };</pre>	<pre>Java class f { int i;   A a=new A();   char[] buf=new char[10]; public void f() { // Traitement   // sans variable locale };</pre>
--	---

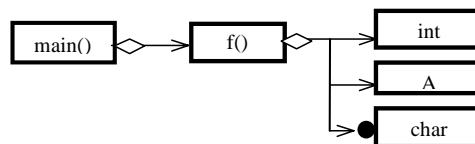
et l'appel de celle-ci par :

<pre>C++ { delete new f(); // Oupps !!! }</pre>	<pre>Java { new f(); }</pre>
---	------------------------------

L'appel d'une fonction crée une instance d'exécution de la classe `f`, et le retour de la fonction détruit cette instance (`delete` ou `apèle` du ramasse-miettes). Il n'est pas nécessaire d'avoir des variables déclarées dans le constructeur de la classe `f` car chaque instance possède les attributs nécessaires. Un appel récursif peut être vu comme la création de plusieurs instances d'exécution de la même fonction.

Lorsqu'une fonction appelle une autre fonction, elle agrège l'instance d'exécution. Par exemple, si la fonction `main()` appelle la fonction `f()`, l'instance d'exécution de `f()` est agrégée par l'instance d'exécution de `main()` (Figure 19).

Figure 19



La machine virtuelle de Java utilise en interne cette notion d'instance d'exécution. Lors de l'appel d'une méthode, une instance d'une *stack frame* est ajoutée au sommet de la pile. Les paramètres et les variables locales sont mémorisés dans cette instance. Lors de la sortie de la méthode, cette instance est supprimée de la pile.

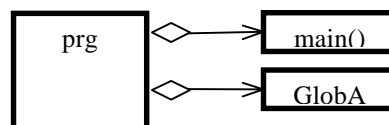
### 3.2 Les variables globales

Les objets globaux à l'application sont créés par le C++ avant l'appel du `main` et détruits après celui-ci. Pour Java, les objets globaux sont créés lors du chargement des classes les possédants.

<pre>C++ A GlobA; void main(int argc,           const char*[] argv) { //... }</pre>	<pre>Java class A { static A GlobA=new A();   public static void main(     String[] args)   { //...   } }</pre>
---	---

Les variables globales et l'instance d'exécution de `main()` sont agrégées par l'instance du programme. Le code précédent se schématise comme Figure 20.

Figure 20



L'instance du programme sera détruite après que celui-ci aura libéré toutes les ressources nécessaires à son exécution. En quelque sorte, les variables globales sont les attributs de l'*instance du programme*. Les fonctions et les méthodes statiques étant les méthodes de l'instance du programme. Lancer simultanément plusieurs fois le même programme permet d'avoir plusieurs fois l'instance du programme avec ces attributs (ces variables globales).

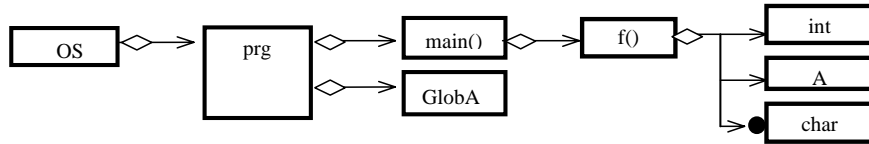
### 3.3 Variables de l'OS

Les paramètres passés à la fonction `main` ont également une durée de vie gérée par le compilateur. En effet, avant d'appeler la fonction `main`, le programme exécute une fonction de démarrage souvent appelée `crtd`. Celle-ci se chargera de traduire les informations fournies par

le système d'exploitation et de les convertir dans un formalisme compréhensible par le langage. Ainsi, la ligne de commande et les variables d'environnement sont traduites par la fonction de démarrage pour pouvoir servir de paramètres à la fonction `main`.

Qui s'occupe de la durée de vie des paramètres que fournit le système d'exploitations ? Eh bien, il s'agit du système lui-même. En général, ces informations sont stockées dans la zone mémoire réservée au programme exécuté. Celui-ci informera le système d'exploitation de sa fin, qui s'occupera alors de libérer les zones mémoires qui étaient nécessaires à l'application. Les paramètres complémentaires seront ainsi détruits (Figure 21).

Figure 21



Comme on le voit avec cette description détaillée des rôles de chacun, chaque objet possède un responsable qui s'occupera de le détruire. L'ultime responsable de la gestion des ressources étant le système d'exploitation.

Toutes les zones mémoires appartenant à un objet doivent être détruites par celui-ci. Une zone mémoire appartenant à un objet est une *agrégation*. Une agrégation est une relation particulière qui lie la durée de vie de l'objet en relation à celle de l'objet possédant cette relation. Si un objet est créé dans une fonction, à la fin de celle-ci l'objet sera détruit et les zones mémoires nécessaires à l'objet également.

Le ramasse-miettes de java parcourt toutes les références des objets de la mémoire pour identifier les instances inaccessibles. Pour cela, l'algorithme doit choisir un premier objet servant de racine à tous les autres. Cet objet est le représentant de l'instance `prg` ci-dessus. Tous les objets de java sont agrégés, directement ou indirectement par cet objet racine. Détruire cet objet c'est détruire tous les objets de la machine virtuelle, c'est donc sortir du programme.

## 4. COMMENT PROPOSER L'AGREGATION ?

Il semble important de proposer une syntaxe particulière traduisant les spécificités des agrégations. Cela à plusieurs avantages.

- Les informations sémantiques de modélisation restent présentes dans le source. Avec les langages actuels, la traduction d'une agrégation du modèle n'apparaît plus dans le langage.
- Les comportements par défaut des compilateurs peuvent générer automatiquement un code correct.
- Des écritures invalides peuvent être détectées par le compilateur avant même la phase d'exécution.
- La gestion de la mémoire peut être fortement optimisée. Un programme C++ pourra n'avoir aucune instruction `delete` et pourtant, sans avoir de ramasse-miettes.

De plus, des outils spécifiques peuvent tirer parti de ces informations :

- ORB type Corba, RMI ou COM
- Base de données objet ou non
- Outils d'ingénieries inverses pour déduire un schéma UML
- ...

Regardons les différents traitements nécessaires à l'utilisation d'une agrégation et proposons une extension de la syntaxe pour cela.

### 4.1 Un pointeur d'agrégation

Pour différencier les pointeurs de relation des pointeurs d'agrégation, proposons la syntaxe suivante :

<pre> <b>C++</b> { Object* rel; // relation   Object@ agr; // agrégation }           </pre>	<pre> <b>Java</b> { Object rel; // relation   Object@ agr; // agrégation }           </pre>
---	---

Le caractère '@' permet d'indiquer que la référence est une agrégation. Lors de la destruction d'un pointeur d'agrégation, il est possible de détruire immédiatement l'objet référencé.

Cela décharge énormément le ramasse-miettes de Java. La destruction d'une instance est parfaitement déterministe. En fait, le ramasse miette n'a alors à s'occuper que de très peu d'objets.

Le choix d'utiliser le caractère '@' est parfaitement arbitraire. D'autres syntaxes peuvent être proposées. Comme il en faut une pour éclairer le discours, j'ai choisi celle-là.

En C++, le pointeur d'agrégation permet d'écrire des programmes sans jamais utiliser l'opérateur `delete`. Les instances sont détruites précisément lorsqu'il le faut.

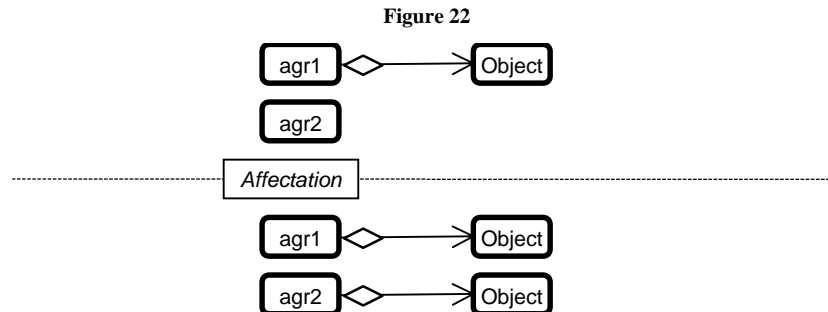
### 4.2 Les affectations

Les affectations entrent pointeurs d'agrégations sont de deux types. Par définition, un objet ne peut être agrégé que par un seul pointeur d'agrégation. Il faut donc, soit dupliqué l'objet agrégé, soit l'adopter.

Pour demander une duplication implicite de l'objet agrégé, il faut utiliser l'opérateur d'affectation classique.

```
Object@ agr1;
Object@ agr2;
agr1 = new Object();
agr2 = agr1; // equivalent à agr2=agr1.clone()
```

Lors de l'affectation, la méthode `clone()` de l'objet agrégé par `agr1` est appelé (Figure 22).



Au même titre que pour les types primitifs, l'affectation d'un pointeur d'agrégation propose une sémantique par valeur de l'instance référencée.

```
int a=1;
int b=2;
a=b;
```

Est similaire à

**C++**

```
int@ a=new int(1);
int@ b=new int(2);
a=b;
```

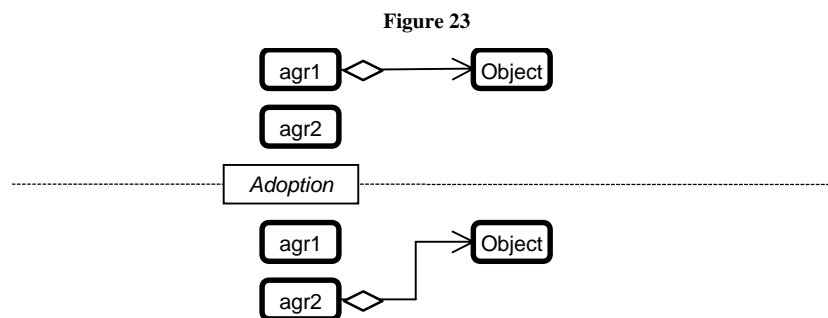
**Java**

```
Integer@ a=new Integer(1);
Integer@ b=new Integer(2);
a=b;
```

Pour demander l'adoption de l'objet il faut utiliser un nouvel opérateur `@=`.

```
Object@ agr1;
Object@ agr2;
agr1 = new Object();
agr2 @= agr1; // agr1=null
```

`agr1` possède alors la valeur `null`. L'instance est référencée par un seul pointeur d'agrégation (Figure 23).



Pour détruire l'instance référencée par un pointeur d'agrégation, il suffit de le valoriser avec la valeur `null`. C'est l'équivalent à appeler l'opérateur `delete` du C++.

**C++**

```
Object@ agr=new Object();
agr=NULL;
```

**Java**

```
Object@ agr=new Object();
agr=null;
```

Les affectations entrent relations et agrégations respectent des règles strictes. Une relation peut être initialisée à l'aide d'un pointeur d'agrégation, mais pas l'inverse.

**C++**

```
{ Object@ agr=new Object();
  Object* rel;
  rel=agr; // valide
  agr=rel; // invalide
}
```

Pourquoi ces contraintes ? Il existe deux affectations possibles d'un pointeur d'agrégation. Il faut savoir s'il faut dupliquer l'instance référencée par la relation, ou s'il faut l'adopter. L'adoption n'est pas possible à partir d'une relation. Pour éviter les duplications d'instances involontaires, l'affectation d'une agrégation à partir d'une relation n'est pas possible. Il faut explicitement indiquer comment cloner l'objet référencé.

**C++**

```
agr=rel->clone();
```

Ainsi, le développeur doit prendre la responsabilité de la duplication d'une instance à partir d'une relation. Il ne peut pas faire d'adoption à partir d'une relation.

**Java**

```
{ Object@ agr=new Object();
  Object rel;
  rel=agr; // valide
  agr=rel; // invalide
}
```

**Java**

```
agr=rel.clone();
```

### 4.3 La comparaison de référence

Comparer deux pointeurs d'agrégation n'a pas de sens. En effet, par définition, chaque pointeur d'agrégation est le seul à pointer sur l'instance. Donc la comparaison de deux pointeurs d'agrégation est toujours fautive.

```
Object@ p=...;
Object@ q=...;
p==q; // Toujours false
```

Une erreur ou un message d'avertissement peut signaler cela. Par contre, comparer une relation avec un pointeur d'agrégation est valide. Pour les variables `agr` et `rel` suivante :

**C++**

```
Object@ agr=new Object();
Object* rel;
```

Les comparaisons autorisées sont :

**Java**

```
Object@ agr=new Object();
Object rel;
```

<code>agr==agr</code>	Erreur
<code>agr==rel</code>	Valide
<code>rel==agr</code>	Valide
<code>rel==rel</code>	Valide

Pour respecter la sémantique par valeur de l'agrégation, il faut utiliser une écriture du type :

**C++**

```
*agr1==*agr2
```

Cela permet de comparer la valeur des instances pointées. Attention, `agr1` ou `agr2` peuvent être à `null`. On peut envisager un appel implicite de ce type pour autoriser la comparaison de deux pointeurs d'agrégations.

**Java**

```
agr1.equals(agr2)
```

### 4.4 La construction d'une instance

Lors de la construction d'une instance, la valeur du pointeur retourné par l'opérateur `new` est l'unique pointeur référençant l'objet qui vient d'être créé. Cela correspond exactement à la sémantique d'un pointeur d'agrégation. Si on perd le pointeur, en sortant de la méthode par exemple, l'objet référencé peut être immédiatement détruit. Il n'est pas nécessaire d'attendre l'intervention d'un ramasse-miettes éventuelle. L'opérateur `new` ne doit plus retourner une relation, mais une agrégation. La signature de l'opérateur `new` doit être modifiée. Il n'est pas possible de valoriser une relation avec la valeur retournée par l'opérateur `new`.

**C++**

```
{ Object@ agr=new Object();
  Object* rel=new Object(); // Non !
}
```

Pour des raisons de compatibilité ascendante, la valorisation d'une relation au retour de l'opérateur `new` peut être conservée.

**Java**

```
{ Object@ agr=new Object();
  Object rel=new Object(); // Non !
}
```

### 4.5 Le passage de paramètres

Recevoir une agrégation en paramètre peut avoir deux traductions possibles. Est-ce que l'instance pointée doit être dupliquée ou doit-elle être adoptée par le paramètre ?

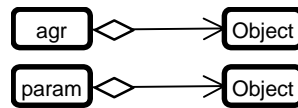
Pour demander la duplication de l'instance agrégée lors de la réception d'un paramètre, il faut indiquer que le paramètre est une agrégation.

```
void f(Object@ param)
{ ...
}
...
void g()
{ Object@ agr=new Object();
```

```
f(agr);
}
```

Le paramètre `param` reçoit une copie de l'instance référencée par `agr` (Figure 24).

Figure 24



Ce comportement est similaire au passage de paramètre des types primitifs.

```
void f(int param)
{ ...
}
...
void g()
{ int agr=0;
  f(agr);
}
```

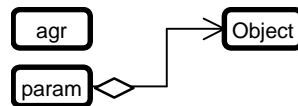
`param` reçoit une copie de la variable `agr`.

Pour demander l'adoption de l'objet référencé par le pointeur d'agrégation, il faut utiliser une syntaxe particulière.

```
void f(Obj@= param)
{ ...
}
...
void g()
{ Obj@ agr=new Object();
  f(agr); // agr=null
}
```

`@=` permet d'indiquer que le paramètre est un pointeur d'agrégation qui adopte l'instance référencée par l'appelant. La variable `agr` prend la valeur `null` juste avant d'appeler `f()`. Le paramètre `param` pointe sur l'instance précédemment référencée par `agr` (Figure 25).

Figure 25



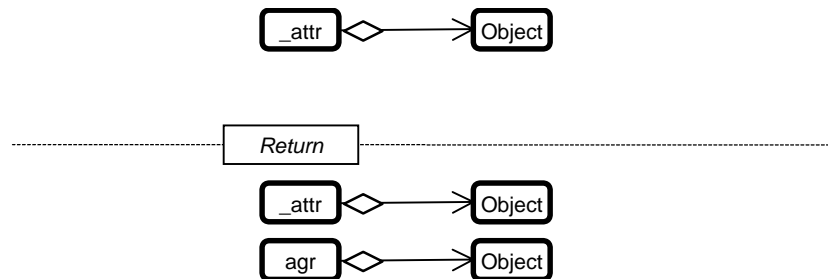
#### 4.6 Le retour de pointeur d'agrégation

Retourner un pointeur d'agrégation permet d'indiquer qu'il s'agit d'une méthode ou une fonction de construction (« factory method »).

```
Object@ getCopy()
{ return _attr;
}
...
Object@ agr=getCopy();
```

La méthode `getCopy()` retourne une agrégation référençant une copie de l'instance référencée par `_attr` (Figure 26).

Figure 26



Ce comportement est similaire à l'utilisation d'un type primitif.

```
int getCopy()
{ return _attr;
}
```

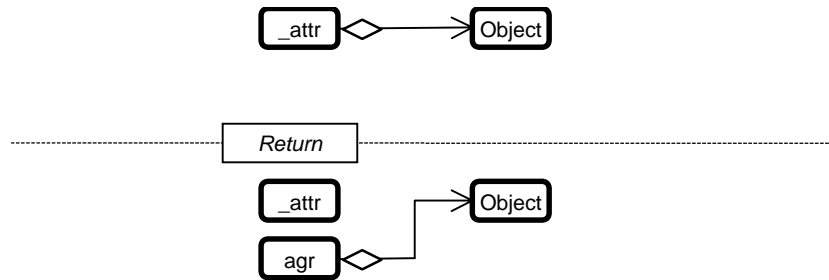
```
...
int agr=getCopy();
```

agr reçoit une copie de l'attribut `_attr`.

Pour signaler que l'on désire faire adopter une instance par l'appelant, il faut utiliser `@=` (Figure 27).

```
Object@= clone()
{ return new A(this);
}
Object@= unadopte()
{ return _attr;
}
...
Object@ agr=unadopte();
```

Figure 27



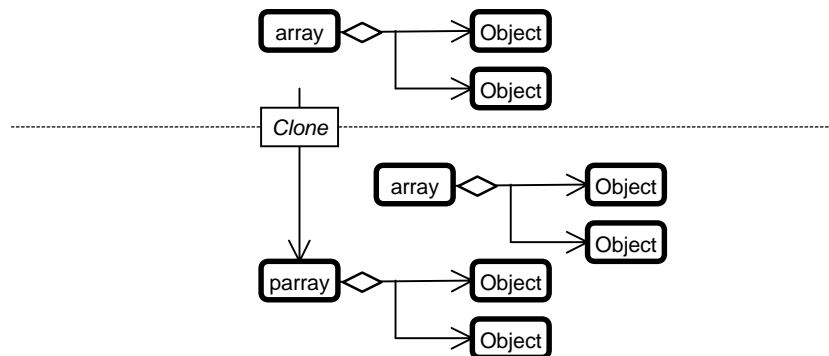
#### 4.7 Les tableaux

Il est possible d'avoir des tableaux de pointeur d'agrégation. Si on duplique le tableau, les instances référencées sont également dupliquées (Figure 28).

```
C++
{ Object@ array[10];
  array[0]=new Object();
  Object@[ ]@ parray=clone(array);
}
```

```
Java
{ Object@[ ]@ array=new Object@[10];
  array[0]=new Object();
  Object@[ ]@ parray=array.clone();
}
```

Figure 28



`parray` agrège un tableau de pointeur d'agrégation. Perdre `parray` entraîne la destruction du tableau et de toutes les instances référencées dans le tableau.

#### 4.8 Les conversions

Comment convertir un pointeur d'agrégation ? La conversion d'une agrégation entraîne toujours une adoption. Sinon, il pourrait y avoir deux pointeurs d'agrégation référençant la même instance.

```
C++
String@ s=new String();
Object@ o @=s;
s @= static_cast<String@>(o);
```

```
Java
String@ s=new String();
Object@ o @=s;
s @=(String@)o;
```

Il est possible d'utiliser une conversion pour obtenir une relation sur l'instance agrégée.

```
C++
String* rel=static_cast<String*>(o);
```

```
Java
String rel=(String)o;
```

La variable `rel` référence l'instance agrégée par `o`, avec le type `String`. Les conversions implicites ne sont pas autorisées avec les agrégations.

#### 4.9 Les exceptions

Comment se comportent les agrégations lors d'une exception ? Si un pointeur d'agrégation est détruit, l'objet référencé doit être automatiquement détruit. Lors d'une exception, les objets agrégés seront immédiatement détruits.

<b>C++</b> <pre>{ try   { Object@ p=new Object();     //...   } catch (...)   { } }</pre>	<b>Java</b> <pre>{ try   { Object@ p=new Object();     //...   } catch (Throwable x)   { } }</pre>
--	---

L'instance référencée par `p` est automatiquement détruite lors de la capture d'une exception. Ce comportement est similaire aux types primitifs.

<b>C++</b> <pre>{ try   { int p;     //...   } catch (...)   { } }</pre>	<b>Java</b> <pre>{ try   { int p     //...   } catch (Throwable x)   { } }</pre>
---	---

`p` est détruit lors de l'exception.

#### 4.10 Optimisation

Certaines situations peuvent être automatiquement optimisées. Par exemple, cloner une instance pointée par un pointeur d'agrégation temporaire n'a pas d'intérêt. Dans ce cas, l'adoption peut être choisie implicitement par le compilateur.

<b>C++</b> <pre>class Object { Object@ methode(bool tst)   { if (tst==true) return _attr;     else return new Object(*this);   } };</pre>	<b>Java</b> <pre>Class Object { Object@ methode(boolean tst)   { if (tst==true) return _attr;     else return new Object(*this);   } };</pre>
--	--

La `methode()` retourne un pointeur d'agrégation avec clonage de l'instance agrégé. Si le paramètre `tst` est à `true`, l'instance pointée par `_attr` est clonée avant d'être retournée. Si par contre, le paramètre `tst` est à `false`, il faudrait normalement construire un pointeur d'agrégation temporaire pour récupérer l'instance créée par `new`, et dupliquer cette instance temporaire pour le retour de la méthode. Le code est normalement équivalent à :

<b>C++</b> <pre>class Object { Object@ methode(bool tst)   { if (tst==true) return _attr;     else       { Object@ rc=new Object(*this);         return rc; // rc.clone();       }   } };</pre>	<b>Java</b> <pre>Class Object { Object@ methode(boolean tst)   { if (tst==true) return _attr;     else       { Object@ rc=new Object(*this);         return rc; // rc.clone();       }   } };</pre>
--	--

L'instance référencée par `rc` n'est pas utilisée. Pour supprimer la création de cette instance temporaire, le compilateur peut décider de générer un code demandant la désadoption par `rc` de son instance. Cette traduction est équivalente à avoir `Object@ methode(bool tst)` si `tst` est à `true` et `Object@= methode(bool tst)` si `tst` est à `false`.

L'optimisation permet d'améliorer également les situations de conversion d'agrégation.

<b>C++</b> <pre>String@ s=new String(); Object@ o @=s; s = static_cast&lt;String@&gt;(o);</pre>	<b>Java</b> <pre>String@ s=new String(); Object@ o @=s; s =(String@)o;</pre>
--	---

La conversion de l'agrégation `o` entraîne normalement la création d'une référence d'agrégation temporaire qui adopte l'instance référencée par `o`. La variable `o` possède alors la valeur `null`. Ensuite, l'opérateur égale entraîne normalement une copie de l'instance référencée par le pointeur temporaire.

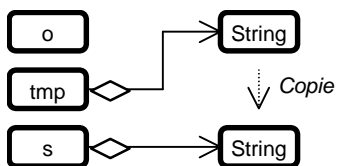
<b>C++</b> <pre>String@ s=new String(); Object@ o @= s; String@ _tmp @= o;</pre>	<b>Java</b> <pre>String@ s=new String(); Object@ o @= s; String@ _tmp @= o;</pre>
---	--

`s = tmp;`

`s = _tmp;`

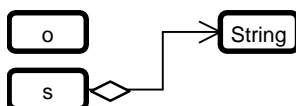
Sans optimisation, la mémoire se retrouve comme ceci après l'affectation (Figure 29).

Figure 29



Avec l'optimisation, le code évite l'utilisation de la référence temporaire et ainsi la duplication de l'instance référencée par `o`. La mémoire se retrouve comme Figure 30.

Figure 30



Le principe de l'optimisation se résume en une phrase : L'affectation d'une référence d'agrégation par une référence d'agrégation temporaire entraîne toujours une adoption.

## 5. LES ERREURS MEMOIRES CLASSIQUES

Le concept d'agrégation apporte indirectement une meilleure gestion de la mémoire et corrige partiellement les erreurs classiques. Regardons comment le concept d'agrégation permet de réduire considérablement les erreurs mémoires.

Type d'erreur	Solutions
<p><b>Fuite mémoire.</b></p> <p>Si un programme demande la création d'un objet dans la mémoire, il ne doit pas oublier de le détruire. Sinon, le programme fonctionne, mais à la longue, il ralentit avant de s'arrêter par manque de ressource.</p>	<p>Java répond à cela automatiquement grâce au ramasse-miettes.</p> <p>Le C++ impose au développeur de gérer précisément les allocations et les libérations mémoires. L'agrégation permet d'exprimer à l'avance quand un objet devra être supprimé de la mémoire. En choisissant d'utiliser un pointeur d'agrégation, le développeur indique que l'objet pointé devra être détruit lorsque le pointeur disparaîtra. Il n'est plus nécessaire d'appeler explicitement l'opérateur <code>delete</code>, alors même que le C++ ne possède pas de ramasse-miettes.</p>
<p><b>Modification involontaire d'un objet agrégé.</b></p> <p>Il arrive que l'on garde par erreur un pointeur sur un objet adopté. Cela viole le principe d'encapsulation. En effet, il est ainsi possible de modifier une instance, à l'insu de l'objet propriétaire.</p>	<p>Lors de l'adoption d'une instance, le pointeur utilisé comme paramètre prend immédiatement la valeur <code>null</code>. Ainsi, il n'est plus possible au client de manipuler l'instance à l'insu de l'instance propriétaire.</p>
<p><b>Partage involontaire d'instances.</b></p> <p>Il arrive souvent qu'une instance soit par erreur partagée. Cela entraîne qu'une modification de cette instance par un chemin ait un impact involontaire par un autre chemin.</p> <p>Par exemple, modifiant l'<code>Adresse</code> de Paul, Pierre à son <code>Adresse</code> modifié. Ces erreurs sont très difficiles à identifier car l'erreur sera détectée en consultant Pierre, souvent bien après la modification de Paul.</p>	<p>La duplication automatique des instances agrégées permet d'éviter ces erreurs. La déclaration d'un pointeur d'agrégation entraîne l'interdiction de partager une instance. Cette contrainte est garantie par le compilateur.</p>
<p><b>Destructions partielles d'une instance.</b></p> <p>Si une voiture agrège un moteur, il est possible de détruire la voiture sans détruire le moteur. Cela ne devrait normalement pas être possible. S'il existe un pointeur référençant le moteur, le ramasse-miettes peut décider de détruire la voiture et de garder le moteur.</p>	<p>La sémantique des pointeurs étant améliorée, il est possible d'ajouter un compteur de référence sur chaque instance. Lors de la destruction d'une agrégation, une exception peut être générée pour signaler qu'il existe encore, par erreur, des références sur l'instance.</p>
<p><b>Modification d'une agrégation dans une méthode constante.</b></p> <p>Le C++ propose le concept de méthode constante. Une méthode constante garantie qu'elle ne modifie pas l'instance courante. Un objet agrégé est une partie de l'agrégant. Il ne devrait pas être possible de modifier un objet agrégé dans une méthode constante. Les pointeurs représentant une relation, cela n'est pas vérifié par les compilateurs.</p>	<p>Le compilateur peut vérifier dans une méthode constante que seules les méthodes constantes sont appelées par l'intermédiaire d'un pointeur d'agrégation.</p>

Parmi les erreurs non gérées, citons la possibilité de construire par erreur des instances s'agrégant mutuellement. L'adoption permet cela.

**C++**

```
struct A
{ A@ _agr;
}
A@ a=new A();
a->_agr@=a; // a==null
```

Dans cette situation, le C++ aura une perte mémoire, Java nettoiera cela avec l'aide du ramasse-miettes.

**Java**

```
class A
{ public A@ _agr;
}
A@ a=new A();
a._agr@=a; // a==null
```

## 6. IMPLANTATION

Nous allons voir quelques propositions pour l'implantation de ce concept dans le C++ et dans Java.

### 6.1 C++

Il est raisonnable de rédiger un préprocesseur faisant une traduction de ce concept en code C++ classique. Cette approche permet de facilement modifier le code et garantir une compatibilité avec tous les compilateurs C++ du marché. Par la suite, lorsque ce concept sera stabilisé, les compilateurs pourront l'intégrer. Le code bénéficiera alors d'optimisations spécifiques.

#### 6.1.1 clone

Comme nous l'avons constaté, il est souvent nécessaire de dupliquer les instances agrégées. Le C++ propose d'utiliser un constructeur de copie pour cela. Cette approche n'est pas polymorphique. Un objet pointé doit être capable de produire une copie de lui-même, quel que soit son type. Pour cela, nous déclarons une fonction `template` permettant d'obtenir une copie d'une instance.

```
template <class T>
inline T@= clone(const T& x)
{ return new T(x); }
```

Le comportement par défaut de cette fonction appelle le constructeur de copie.

```
int a=1;
int@ p=clone(a);
```

Pour bénéficier du polymorphisme, une classe doit surcharger la fonction `clone()`.

```
class Object
{ public:
  Object(const Object& x) // cctr
  { }
  virtual Object@= clone() const
  { return new Object(*this);
  }
};
inline Object@= clone(const Object & x)
{ return x.clone(); }

class SubObject : public Object
{ public:
  virtual SubObject@= clone() const
  { return new SubObject(*this);
  }
}
```

S'il n'existe pas de constructeur de copie ou de spécialisation de la méthode `clone()` pour un type d'objet agrégé, seules les adoptions sont autorisées.

### 6.1.2 Traductions des syntaxes

La classe `template _agr<>` représente un pointeur d'agrégation. Elle permet de détruire automatiquement l'instance référencée lors de la destruction du pointeur et propose des services d'adoptions et de désadoptions. La sémantique est différente de `auto_ptr<>` car il y a clonage automatique lors du constructeur de copie et de l'opérateur d'affectation.

```
class _agr_base {};
template <class T>
class _agr : public _agr_base
{ T* _pt;
  public:
  _agr (T* pt=NULL)
  : _pt(pt) {}

  _agr(const _agr<T>& x)
  { _pt=(x._pt!=NULL) ? (T*)clone(*x._pt) : NULL;
  }

  ~_agr()
  { delete _pt; }

  T* operator ->() const
  { assert(_pt!=NULL);
    return _pt;
  }

  T& operator *() const
  { return *operator->(); }

  operator T* () const
  { return _pt; }

  _agr<T>& adopte(T* x)
  { delete _pt;
    _pt=x;
    return *this;
  }
  _agr<T>& adopte(_agr<T>& x)
  { T* old=_pt;
    _pt=x._pt;
    x._pt=NULL;
    delete old;
    return *this;
  }
  _agr<T>& operator =(const _agr<T>& x)
  { _pt=(x._pt!=NULL) ? clone(*x._pt) : NULL;
    return *this;
  }
}
```

```

}

T* release()
{ T* rc=_pt;
  _pt=NULL;
  return rc;
}

protected:
void operator[](size_t); // Non implémenté
void operator =(const _agr_base&); // Non implémenté
bool operator ==(const T* p); // Non implémenté
bool operator !=(const T* p); // Non implémenté
_agr(const _agr_base &); // Non implémenté
};

```

L'extension C++ génère les traductions suivantes :

Nouvelle syntaxe	Traduction en C++ classique
Obj@ p;	_agr<Obj> p;
Obj@ p = new Obj(); Obj@ q = p;	_agr<Obj> p = new Obj(); _agr<Obj> q = p;
Obj@ p = new Obj(); Obj@ q @= p; p @= q;	_agr<Obj> p =new Obj(); _agr<Obj> q = p.release(); p.adopte(q);
Void f(Obj@ para) { } ... Obj@ p = new Obj(); f(p);	void f(_agr<Obj> para) { } ... _agr<Obj> p = new Obj(); f(p);
Void f(Obj@= para) { } ... Obj@ p = new Obj(); f(p);	void f(_agr<Obj> para) { } ... _agr<Obj> p = new Obj(); f(p.release());
Obj@ f() { return x; } ... Obj@ p=f();	_agr<Obj> f() { return x; } ... _agr<Obj> p=f();
Obj@ x; Obj@= f() { return x; } ... Obj@ p=f();	_agr<Obj> x; _agr<Obj> f() { return x.release(); } ... _agr<Obj> p=f();
Obj@ arr[10]; Obj@ arr2[10]=arr;	_agr<Obj> arr[10]; _agr<Obj> arr2[10]=arr;
Obj@[ ] arr= new Obj@[10]; Obj@[ ] arr2=clone(arr);	_agr<Obj>[ ] arr= new _agr<Obj>[10]; _agr<Obj>[ ] arr2=clone(arr);
Derived@ d=static_cast<Derived@>(b);	_agr<Derived> d= static_cast<_agr<Derived> >(b.release());

Le C++ utilisant un code particulier pour l'effacement des tableaux (`delete[]`) il est nécessaire d'offrir des traductions similaires utilisant le `template _agr_array<>` pour les agrégations de tableaux.

Il n'y a pas d'arithmétique sur les pointeurs d'agrégation.

```

Obj@ p=...
Obj@ q=...
++p; // Erreur
q-p; // Erreur

```

L'arithmétique ne fonctionne qu'avec les relations.

### 6.1.3 Méthodes constantes

Une méthode constante ne peut pas modifier d'attribut. Ce concept peut être étendu pour les agrégations.

```

class A
{ int _x;
  void set(int x)
  { _x=x;
  }
};

class B
{ A _attr;
  void m () const
  { _attr.set(100); // BUG
  }
};

```

La méthode `m()` ne peut pas appeler la méthode `set()` sur l'attribut `_attr`. En effet, la méthode `set()` modifie l'instance `_attr` donc modifie indirectement l'instance `B` courante. `set()` n'est pas une méthode constante.

En utilisant un pointeur d'agrégation, le compilateur limitera l'utilisation des instances agrégées.

```

class A
{ int _x;
  void set(int x)
  { _x=x;
  }
};

class B
{ A@ _attr;
  void m () const
  { _attr->set(100); // Message du compilateur
  }
};

```

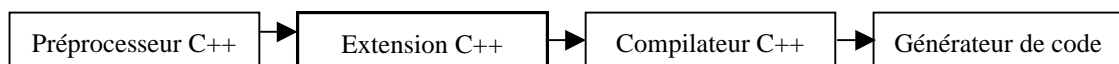
Cela est un nouvel avantage de ce concept. Une méthode constante ne peut pas modifier l'instance courante, ni les instances agrégées. Cela n'est pas vérifié par les compilateurs C++ classique.

#### 6.1.4 Préprocesseur

Pour rédiger un préprocesseur permettant la traduction du concept d'agrégation en C++ classique, il faut procéder par étape. Dans un premier temps, il faut rédiger un analyseur étant capable de comprendre un source C++, d'en avoir une vision mémoire arborescente, et d'être capable de retraduire cette vision mémoire en source C++ classique. Cette première étape n'introduit aucun élément supplémentaire.

Ensuite, il sera possible d'ajouter des extensions à la syntaxe du C++. Ces extensions enrichiront l'arbre syntaxique mémoire. L'algorithme de production du code C++ devra être adapté pour proposer une traduction des nouveaux concepts. Le cheminement de production du code sera comme sur la Figure 31.

Figure 31



L'extension C++ s'intercale entre l'analyse par le préprocesseur et le compilateur C++. Ensuite, il sera préférable d'apporter des modifications directement au compilateur. Cela afin de tenir compte des templates paramétré par un pointeur d'agrégation (Ex. `list<Object@>`).

## 6.2 Java

Il est raisonnable de rédiger un préprocesseur faisant une traduction de ce concept en code Java classique. Cette approche permet de facilement modifier le code et garantie une compatibilité avec tous les compilateurs Java du marché. Par la suite, lorsque ce concept sera stabilisé, les compilateurs pourront l'intégrer. Le code bénéficiera alors d'optimisations spécifiques. La machine virtuelle pourra également être mise à jour.

### 6.2.1 clone

Java propose une méthode `clone()` dans la classe `Object`. Cette méthode sera utilisée à chaque fois qu'une instance agrégée devra être dupliquée. Le préprocesseur pourra vérifier que le type de l'objet pointé implémente publiquement la méthode `clone()`. Une erreur ou un avertissement pourra être affiché lors de la compilation si une instance n'est pas capable de répondre correctement à cette invocation. Dans ce cas, seules les adoptions sont autorisées.

La méthode `clone()` de la classe `Object` n'est pas rédigée correctement pour tenir compte de l'agrégation. Il faut alors ajouter une version `_clone()` lorsqu'une classe implémente l'interface `Cloneable` et remplacer l'appel à `Object.clone()` par un appel à `_clone()`.

S'il n'existe pas de méthode `clone()` de déclarée, il faut en ajouter une.

```

class MaClass implements Cloneable
{ Object@ _agr;
  public Object clone() throws CloneNotSupportedException
  { return super._clone();
  }
};

```

```

}
}

```

Deviens :

```

class MaClass implements Cloneable
{ Object _agr;
  private Object _clone() throws CloneNotSupportedException
  { MaClass x=(MaClass)super.clone();
    // Utilise clone pour les agrégations
    x._agr=(_agr!=null) ? (Object)_agr.clone() : null;
    return x;
  }
  public Object clone() throws CloneNotSupportedException
  { return _clone();
  }
}

```

## 6.2.2 Traductions des syntaxes

L'extension Java génère les traductions suivantes.

Nouvelle syntaxe	Traduction en Java classique
Obj@ p;	Obj p;
Obj@ p = new Obj(); Obj@ q = p;	Obj p = new Obj(); Obj q = (Obj)p.clone();
Obj@ p = new Obj(); Obj@ q @= p; q @= p;	Obj p = new Obj(); Obj q = p; p=null; q = p; p = null;
void f(Obj@ para) { } ... Obj@ p = new Obj(); f(p);	Void f(Obj para) { } ... Obj p = new Obj(); f((Obj)p.clone());
void f(Obj@= para) { } ... Obj@ p = new Obj();  f(p);	Void f(Obj para) { } ... Obj p = new Obj();  Obj _tmp1; _tmp1=p; p=null; f(_tmp1);
Obj@ f() { return x; } ... Obj@ p=f();	Obj f() { return (Obj)x.clone(); } ... Obj p=f();
Obj@ x; Obj@= f() { return x; } ... Obj@ p=f();	Obj x; Obj f() { Obj _tmp=x; x=null; return _tmp; } ... Obj p=f();
Obj@[ ] @ arr=new Obj@[10]; ... Obj@[ ] @ arr2=arr;	Obj[] arr=new Obj[10]; ... Obj[] arr2=_cloneAgrObject(arr); ... private static Obj[] _cloneAgrObject(Obj[] array) { Obj[] rc=new Obj[array.length]; For (int i=array.length-1;i>=0;--i) { rc[i]= (Obj)((array[i]==null) ? null : array[i].clone()); } }

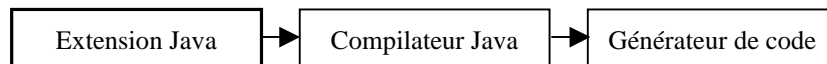
	<pre>return rc; }</pre>
<pre>Derived@ d=(Derived@)b;</pre>	<pre>Derived d=(Derived)b; b=null;</pre>

### 6.2.3 Préprocesseur

Pour rédiger un préprocesseur Java permettant la traduction du concept d'agrégation en Java classique, il faut procéder par étape. Dans un premier temps, il faut rédiger un analyseur étant capable de comprendre un source Java, d'en avoir une vision mémoire arborescente, et d'être capable de retraduire cette vision mémoire en source Java classique. Cette première étape n'introduit aucun élément supplémentaire.

Ensuite, il sera possible d'ajouter des extensions à la syntaxe de Java. Ces extensions enrichiront l'arbre syntaxique mémoire. L'algorithme de production du code Java devra être adapté pour proposer une traduction des nouveaux concepts. Le cheminement de production du code sera comme décrit Figure 32.

Figure 32



L'extension Java s'effectue avant l'intervention du compilateur. Par la suite, il faudra modifier la machine virtuelle et le compilateur Java pour ajouter un nouveau drapeau dans la description des instances. Cela permettra à la méthode `Object.clone()` de proposer une duplication tenant compte des agrégations.

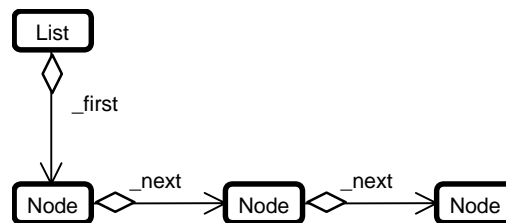
## 7. CAS D'UTILISATION

Regardons quelques exemples de sources utilisant ce nouveau concept.

### 7.1 Liste chaînée

Une liste chaînée est un exemple classique de manipulation complexe de mémoire. La Figure 33 indique un modèle d'instance de cet exemple.

Figure 33



#### 7.1.1 C++

Voici le code en C++ enrichi.

```

template <class T>
class List
{
    struct Node
    {
        Node@ _next;
        T _data;
    public:
        Node(const T& data,
             Node@= next
             )
        {
            _next @= next;
            _data = data;
        }
    };

    Node@ _first;

    public:
    void add(const T& data)
    {
        _first@=new Node(data,_first);
    }
}
  
```

```

void remove(const T& data)
{
    Node* prev = NULL;
    for (Node* cur = _first;
         cur != NULL;
         prev = cur, cur=cur->_next)
    { if (cur->_data == data)
      {
          if (prev==NULL)
              _first @= _first->_next;
          else
              prev->_next @= cur->_next;
          break;
      }
    }
};

```

On remarque que les traitements par défaut sont automatiquement générés. Il n'est pas nécessaire de rédiger le constructeur de copie, le destructeur ou l'opérateur d'affectation. Si une instance `List<T>` est cloné, tous les nœuds sont automatiquement dupliqués. Si une instance `List<T>` est détruite, tous les nœuds le sont automatiquement. Il n'y a pas d'instruction `delete`, pourtant la mémoire est correctement gérée.

La traduction en C++ classique peut ressembler à ceci :

```

template <class T>
class List
{
    struct Node
    {
        _agr<Node> _next;
        T _data;
    public:

        Node(const T& data,
              _agr<Node> next
              )
        {
            _next.adopte(next);
            _data = data;
        }
    };

    _agr<Node> _first;

    public:
    void add(const T& data)
    {
        _first.adopte(new Node(data, _first.release()));
    }

    void remove(const T& data)
    {
        Node* prev = NULL;
        for (Node* cur = _first;
             cur != NULL;
             prev = cur, cur=cur->_next)
        { if (cur->_data == data)
          {
              if (prev==NULL)
                  _first.adopte(_first->_next);
              else
                  prev->_next.adopte(cur->_next);
              break;
          }
        }
    }
};

```

La méthode template `clone` et la classe `_agr<>` doivent également être ajoutées.

## 7.1.2 Java

Voici le code en Java enrichie.

```

interface Clone extends Cloneable
{
    public Object clone() throws CloneNotSupportedException;
}

```

```

class List implements Cloneable
{ static class Node implements Clone
  {
    public Object clone() throws CloneNotSupportedException
    {
      return super.clone();
    }

    private Node@ _next;
    private Clone@ _data;
    public Node(Clone@= data,Node@= next)
    {
      _next @= next;
      _data @= data;
    }
  }
  public Object clone() throws CloneNotSupportedException
  {
    return super.clone();
  }

  private Node@ _first;

  public void add(Clone@= data)
  {
    _first=new Node(data,_first);
  }

  public void remove(Clone data)
  { Node prev = null;
    for (Node cur=_first;
      cur != null;
      prev=cur,cur=cur._next)
    { if (cur._data.equals(data)==true)
      {
        if (prev==null)
          _first @= cur._next;
        else
          prev._next @= cur._next;
        break;
      }
    }
  }
}

```

L'interface `Clone` permet d'indiquer qu'une classe propose la méthode publique `clone()` (`Object` propose cette méthode mais en `protected`). Une instance `List` peut agréger tous objets proposant cette interface. Dupliquer une instance `List` permet de dupliquer les instances agrégées.

La traduction en Java classique peut ressembler à ceci :

```

interface Clone extends Cloneable
{
  public Object clone() throws CloneNotSupportedException;
}

class List implements Cloneable
{ static class Node implements Clone
  {
    protected Object _clone() throws CloneNotSupportedException
    {
      Node x=(Node)super.clone();
      x._next=(x._next!=null) ? (Node)x._next.clone() : null;
      x._data=(x._data!=null) ? (Clone)x._data.clone() : null;
      return x;
    }
    public Object clone() throws CloneNotSupportedException
    {
      return _clone();
    }

    private Node _next;
    private Clone _data;
    public Node(Clone data,Node next)
    {
      { _next = next; next=null; }
    }
  }
}

```

```

    { _data = data; data=null; }
}
protected Object _clone() throws CloneNotSupportedException
{
    Tree x=(Tree)super.clone();
    x._first=( _first!=null) ? (Node)_first.clone() : null;
    return x;
}
public Object clone() throws CloneNotSupportedException
{
    return _clone();
}

private Node _first;

public void add(Clone data)
{
    Clone _tmp1;
    Node _tmp2;
    { _tmp1=data;data=null;
      _tmp2=_first;_first=null;
      _first=new Node(_tmp1,_tmp2);
    }
}

public void remove(Clone data)
{
    Node prev = null;
    for (Node cur=_first;
        cur != null;
        prev=cur,cur=cur._next)
    {
        if (cur._data.equals(data)==true)
        {
            if (prev==null)
            { _first = cur._next; cur._next = null; }
            else
            { prev._next = cur._next; cur._next = null; }
            break;
        }
    }
}
}
}
}
}

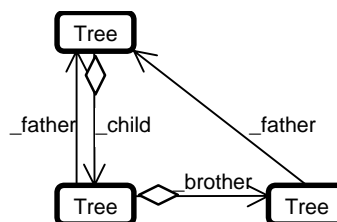
```

La méthode `_clone()` remplace la méthode `Object.clone()`. Cela permet de tenir compte des agrégations.

## 7.2 Arbre

Une structure en arbre est un autre exemple de manipulation complexe de mémoire. La Figure 34 présente un modèle d'instance d'un arbre.

Figure 34



Une instance `Tree` agrège un objet implémentant l'interface `Clone` et un de ces fils. Le fils agrège un de ses propres fils et un de ses frères. Toutes les instances `Tree` possèdent une référence sur leur père.

### 7.2.1 C++

Voici le code en C++ enrichie.

```

template <class T>
class Tree
{
private:
    Tree* _father;
    Node@ _brother;
    Node@ _child;
}

```

```

T    _data;

public:
    Tree(const T& data)
    {
        _data = data;
        _father=NULL;
    }
    Tree(const Tree<T>& x) // cctr
    {
        _father=x._father;
        _brother=x._brother;
        _child=x._child;
        _data=x._data;
        // Update _father link
        for (Tree* p=_child;p!=NULL;p=p->_brother)
            p->_father=this;
    }

private:
    Tree(Tree* father,Tree @= next,const T& data)
    {
        _father = father;
        _brother @= next;
        _data = data;
    }

public:
    void addChild(const T& data)
    {
        _child @= new Tree(this,_child,data);
    }
};

```

Malgré la complexité du modèle mémoire, le code est trivial, et il n'est pas nécessaire de gérer explicitement la mémoire. Détruire un arbre, c'est détruire automatiquement toutes ses branches.

La traduction en C++ classique peut ressembler à ceci :

```

template <class T>
class Tree
{
private:
    Tree*    _father;
    _agr<Tree> _brother;
    _agr<Tree> _child;
    T        _data;

public:
    Tree(const T& data)
    {
        _data = data;
        _father=NULL;
    }
    Tree(const Tree<T>& x) // cctr
    {
        _father=x._father;
        _brother=x._brother;
        _child=x._child;
        _data=x._data;
        for (Tree* p=_child;p!=NULL;p=p->_brother)
            p->_father=this;
    }

private:
    Tree(Tree* father,_agr<Tree> next,const T& data)
    {
        _father = father;
        _brother.adopte(next);
        _data = data;
    }

public:
    void addChild(const T& data)
    {
        _child.adopte(new Tree(this,_child.release(),data));
    }
};

```

Il n'est pas nécessaire de rédiger le constructeur de copie, le destructeur ou l'opérateur d'affectation. Si une instance `Tree<T>` est cloné, tous les nœuds sont automatiquement dupliqués. Si une instance `Tree<T>` est détruite, tous les nœuds le sont automatiquement. Il n'y a pas d'instruction `delete`, pourtant la mémoire est correctement gérée.

## 7.2.2 Java

Voici le code en Java enrichie.

```
interface Clone extends Cloneable
{ public Object clone() throws CloneNotSupportedException;
}

class Tree implements Cloneable
{
    private Tree    _father;
    private Node@   _brother;
    private Node@   _child;
    private Clone@   _data;

    public Object clone() throws CloneNotSupportedException
    {
        Tree@ x=(Tree)super.clone();
        // Update _father link
        for (Tree p=x._child;p!=null;p=p._brother)
            p._father=x;
        return x;
    }

    public Tree(Clone@= data)
    {
        _data @= data;
    }

    private Tree(Tree father,Tree @= next,Clone@= data)
    {
        _father = father;
        _brother @= next;
        _data @= data;
    }

    public void addChild(Clone@= data)
    {
        _child @= new Tree(this,_child,data);
    }
}
```

L'interface `Clone` permet d'indiquer qu'une classe propose la méthode publique `clone()`. Une instance `Tree` peut agréger tous objets proposant cette interface. Dupliquer une instance `Tree` permet de dupliquer les instances agrégées.

La traduction en Java classique peut ressembler à ceci :

```
interface Clone extends Cloneable
{ public Object clone() throws CloneNotSupportedException;
}

class Tree implements Cloneable
{
    private Tree    _father;
    private Tree    _brother;
    private Tree    _child;
    private Clone    _data;

    private Object _clone() throws CloneNotSupportedException
    {
        Tree x=(Tree)super.clone();
        x._brother=(_brother!=null) ? (Tree)_brother.clone() : null;
        x._child=(_child!=null) ? (Tree)_child.clone() : null;
        x._data=(_data!=null) ? (Clone)_data.clone() : null;
        return x;
    }

    public Object clone() throws CloneNotSupportedException
    {
        Tree x=(Tree)_clone();
        // Update _father link
        for (Tree p=x._child;p!=null;p=p._brother)
            p._father=x;
        return x;
    }
}
```

```

public Tree(Clone data)
{
    { _data=data; data=null; }
}

private Tree(Tree father,Tree next,Clone data)
{
    { _father = father;
      { _brother = next; next=null; }
      { _data=data; data=null; }
    }
}

public void addChild(Clone data)
{
    {
        { Tree _tmp1=_child;
          _child=null;
          Clone _tmp2=data;
          data=null;
          _child = new Tree(this,_tmp1,_tmp2);
        }
    }
}
}

```

Le code est plus simple en Java enrichie. La méthode `clone()` est automatiquement générée et les protocoles d'appels de méthodes sont parfaitement respectés.

## 8. AGREGATION PARTAGEE

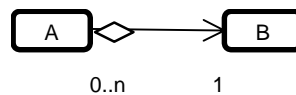
Les agrégations partagées sont des instances devant être détruites lorsque le dernier pointeur référençant l'instance est perdue.

La connexion par modem à l'Internet est un exemple d'agrégation partagée. Lorsqu'une application a besoin de se connecter à l'Internet, une instance de la connexion est ouverte. Il peut y avoir d'autres programmes venant partager cette instance. C'est seulement lorsque le dernier programme libérera la connexion que l'utilisateur pourra indiquer qu'il désire couper la ligne téléphonique. Le programme ayant ouvert la première connexion n'est pas nécessairement le programme entraînant la fermeture de la connexion.

La gestion de fichier sous Unix est également une agrégation partagée. Le même fichier peut être présent dans différents répertoires (commande `ln`). Il est réellement détruit que lorsqu'il n'existe dans aucun répertoire.

Une agrégation partagée est représentée par une cardinalité supérieure à un du côté du losange (Figure 35).

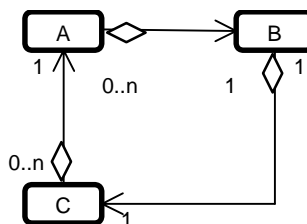
Figure 35



L'instance **B** sera détruite lorsque la dernière instance **A** qui l'agrège sera détruite.

Un compteur de référence peut résoudre les agrégations partagées. Un compteur est associé à l'instance agrégée. Ce compteur est incrémenté lorsqu'un pointeur référence l'instance. Il est décrémenté lorsqu'un pointeur ne la référence plus. Si le compteur est à zéro, l'instance est détruite. C'est le choix fait par la technologie COM de Microsoft. Cela ne résout pas tous les cas.

Figure 36



Dans la Figure 36, les compteurs de références des instances **A**, **B** ou **C** peuvent ne pas être à zéro, alors qu'il n'existe aucun autre pointeur pour les référencer. S'il existe des cycles dans les agrégations partagées, un compteur de référence est insuffisant. Il peut exister des flots d'instances se référençant, mais non rattaché à l'application. Les compteurs de référence de ces instances ne sont pas à zéro.

### 8.1 C++

Pour gérer les instances partagées, avec le C++, il faut utiliser un "smart pointer". Un "smart pointer" est un pointeur gérant un compteur de référence de l'objet pointé. Lorsque ce compteur tombe à zéro, l'instance pointée est détruite. Les cycles ne sont pas gérés avec cet outil. Il faudrait avoir recours à un ramasse-miettes. Cela n'est pas présent dans le langage.

## 8.2 Java

Le ramasse miette est une technologie permettant de résoudre élégamment le problème des cycles des agrégations partagées. La destruction des instances est effectuée automatiquement par le ramasse-miettes. L'inconvénient de cette technique, et qu'il n'est pas possible de savoir exactement à quel moment les instances sont détruites. L'agrégation simple est déterministe, l'agrégation partagée, utilisant le ramasse miette ne l'est pas. On pourrait imaginer un ramasse miette se déclenchant à chaque modification d'un pointeur d'agrégation partagé. Cela permettrait de rendre prédictif les destructions. Le coût en termes de performance est trop élevé pour envisager sérieusement cette solution.

## 9. INSTANCES IMMUABLES

Une instance immuable est une instance initialisée lors de sa construction, mais qui n'évolue pas lors de sa durée de vie. Tous les attributs de l'instance sont valorisés à l'aide du constructeur et il n'existe pas de méthode permettant de les modifier.

La classe `String` de Java est un exemple d'instance immuable. Pour des raisons d'optimisation, il n'est pas nécessaire de dupliquer une instance immuable car il n'y a jamais d'effet de bord. Une méthode `clone()` d'une instance immuable peut retourner `this` à la place de construire un nouvel objet. Une instance peut être partagée sans risque. La sémantique est équivalente à une duplication. Dans ce cas, le ramasse-miettes devra identifier les instances immuables n'étant plus nécessaire à l'application.

Attention, deux instances immuables peuvent avoir les mêmes valeurs. Il faut continuer à utiliser les instances immuables comme si elles ne l'étaient pas.

C++

```
bool compare(String* s1,String* s2)
{ return (s1==s2); // Erreur
}
```

Java

```
boolean compare(String s1,String s2)
{ return (s1==s2); // Erreur
}
```

Il faut comparer la valeur des instances et non si les références pointent sur la même instance.

C++

```
bool compare(String* s1,String* s2)
{ return (*s1==*s2); // Ok
}
```

Java

```
boolean compare(String s1,String s2)
{ return (s1.equals(s2)); // Ok
}
```

Comparer les pointeurs indiquent que les paramètres représentent des relations et non des agrégations. Cela est généralement faux. C'est une erreur classique des instances immuables.

### 9.1 C++

Le C++ ne propose pas de ramasse-miettes. Pour réduire l'espace mémoire utilisée par les instances, il faut construire des instances très petites qui référencent le corps de l'objet, partagé à l'aide d'un compteur de référence.

```
class String
{
    class String_body
    { public:
        int _ref;
        char[] _str;
        ...
    }
    String_body* _body;
    public:
    String()
    { _body=new String_body();
      ++_body->_ref;
    }
    String(const String& x)
    { _body=x._body;
      ++_body->_ref;
    }
    ~String()
    { if (--_body->_ref==0) delete _body;
    }
};
```

Les méthodes de consultations manipuleront directement l'instance `_body`. Si une instance doit modifier son corps et que le compteur de référence est différent de un, une duplication peut être faite avant le traitement. Cette technique permet d'optimiser l'utilisation de la mémoire. L'instance référencée par `String_body` n'est pas une agrégation classique mais une agrégation partagée.

### 9.2 Java

La classe `java.lang.String` est un exemple d'instance immuable. Il est possible d'avoir une relation ou une agrégation sur une instance immuable. La différence est très subtile. Le constructeur de copie peut dupliquer un pointeur sur une `String` (approche équivalente à une relation), tous en voyant le paramètre comme une agrégation.

```
class MaClass
{ String _rel;
  String _agr;
  ...
}
```

```

public MaClass(MaClass x)
{
    _rel=x._rel;
    _agr=x._agr;
}
public Object clone()
{
    return new MaClass(this);
}
public boolean equals(Object x)
{
    MaClass y=(MaClass)x;
    return ((_rel==y._rel) &&
            (_agr.equals(y._agr)))
}
};

```

Par contre, la méthode `equals` est différente. Il faut comparer la valeur des pointeurs pour une relation et la valeur des instances référencées pour l'agrégation. Une erreur classique de Java consiste à utiliser `_agr==y._agr`. Cela peut fonctionner lors des tests unitaires, mais pas en production.

Gérer la durée de vie des instances immuables n'est pas simple. Une instance immuable doit être détruite lorsqu'elle n'est plus utilisée. Le ramasse-miettes fait cela très bien.

Il ne faut pas oublier que les instances immuables sont une technique d'optimisation. Sémantiquement, cela correspond soit à une relation, soit à une agrégation. La classe pourrait être codée comme ceci :

```

class MaClass
{
    String _rel;
    String _agr;
    ...
    public MaClass(MaClass x)
    {
        _rel=x._rel;
        _agr=new String(x._agr);
    }
    ...
};

```

Il n'est pas possible d'utiliser un pointeur d'agrégation pour référencer une instance immuable. Cela nuirait aux optimisations de ce concept. En fait, les références sur des instances immuables représentent des agrégations partagées.

## 10. AMELIORATIONS DES OUTILS

Différents outils de développement peuvent tirer avantage de l'introduction de ce concept dans Java ou le C++.

### 10.1 Object Request Broker

Un ORB est une couche logicielle permettant d'invoquer des services objets entre plusieurs machines. Les paramètres des méthodes peuvent transiter sur le réseau par valeur ou par référence. Les ORBs ont des difficultés à savoir comment traduire une référence. Ils ne savent pas s'ils doivent dupliquer l'instance référencée ou s'ils doivent simplement transmettre le pointeur. COM/DCOM et CORBA sont deux technologies pouvant bénéficier du concept d'agrégation. Avec les pointeurs d'agrégation, il est possible de déduire automatiquement la sémantique des pointeurs. Cela permet de transporter sur le réseau les instances agrégées, mais pas les instances en relations.

### 10.2 Base de données objets

Les bases de données objets peuvent bénéficier du concept d'agrégation afin de gérer la durée de vie des instances. Certaines implémentations proposent un ramasse-miettes dans la base de données. Cela n'est plus vraiment nécessaire si on utilise l'agrégation.

## 11. POURQUOI LE RAMASSE-MIETTES N'EST PAS LA SOLUTION ?

Le ramasse-miettes est un mécanisme permettant de gérer automatiquement les ressources mémoires. La mémoire n'est pas la seule ressource utilisée par les programmes. Il n'existe pas de ramasse-miettes pour fermer les fichiers qui ne sont plus utilisés, pour fermer les connexions réseaux ou autres transactions.

Le programmeur doit gérer correctement toutes les ressources; la mémoire comme les autres. L'agrégation est une technique permettant de gérer facilement les ressources mémoires, et peut être utilisée également pour gérer les autres ressources. L'agrégation permet d'avoir un représentant de la durée d'utilisation d'une ressource. Lorsque le représentant est détruit, la ressource associée est libérée.

Le ramasse-miettes est très utile pour les débutants car ils n'ont plus à s'occuper de la mémoire. Par contre, il va cacher de nombreuses erreurs mémoires très difficiles à localiser. Un programmeur sérieux doit être rigoureux vis-à-vis de toutes les ressources, mémoire comprise.

Si une instance `Voiture` possédant une instance `Moteur`, est détruite car il n'existe plus de pointeur pour la référencer, il est possible que la `Voiture` disparaisse et que le `Moteur` reste vivant. Il suffit qu'il existe un pointeur qui référence par erreur le `Moteur`. Le programme peut croire qu'il référence le `Moteur` d'une `Voiture` existante, alors que la `Voiture` n'existe plus depuis longtemps. Lorsque la référence sur le `Moteur` est détruite, celui-ci disparaît à son tour. Ce type d'erreur aura des conséquences différentes en C++ ou en Java. Le C++ générera une violation mémoire ; Java ne dira rien.

D'autre part, si par erreur une instance est partagée par deux autres instances, il sera très difficile de localiser le problème. Par exemple, si une `Adresse` se retrouve par erreur partagée par deux personnes, lorsque l'une des deux la modifie, l'autre aura son `Adresse` modifiée. Le même programme erroné rédigé en C++ ou en Java sera toujours erroné, mais les symptômes de l'erreur seront différents. En C++, il y aura une violation mémoire, en Java, il n'y aura pas de message. L'erreur sera cachée par le ramasse-miettes.

Les situations justifiant un ramasse-miettes existent, mais sont rares. J'en ai identifié deux :

- Les agrégations partagées
- et les instances immuables

Nous avons vu que les instances immuables sont indirectement des agrégations partagées. Le ramasse miette ne devrait s'occuper que des agrégations partagées. Tous les autres pointeurs peuvent utiliser l'agrégation pour détruire au meilleur moment les instances et libérer ainsi les ressources.

Essayez de trouver dans quelles proportions vous avez besoin d'agrégations partagées dans vos projets ? Parmi les classes que vous avez écrites dans votre vie, très peu présentent cette caractéristique. Parmi ces classes, y a-t-il des cycles dans les agrégations partagées ? Le ramasse-miettes ne se justifie que dans ce cas particulier.

## 12. SYNTAXE

Les différentes syntaxes nécessaires à ce concept sont au nombre de sept. Des alternatives peuvent être proposées.

Déclaration	<code>Object@</code>	<code>agregation Object</code>
Affectation	<code>x = y</code>	<code>x = y</code>
Adoption	<code>X @= y</code>	<code>x adopte y</code>
Paramètre	<code>void f(Object@ x)</code>	<code>void f(agregation Object x)</code>
Adoption de paramètre	<code>Void f(Object@= x)</code>	<code>void f(adopte Object x)</code>
Retour d'agrégation	<code>Object@ g()</code>	<code>agregation Object g()</code>
Adoption lors du retour	<code>Object@= g()</code>	<code>adopte Object g()</code>

L'essentiel dans l'immédiat est de valider le concept.

## 13. CRITIQUES

Plusieurs arguments peuvent être opposés à ces idées.

### 13.1 Pourquoi ne pas utiliser les « `deepCopy()` » et les « `shallowCopy()` » de Smalltalk ?

Smalltalk permet de dupliquer une instance suivant deux stratégies. La première, appelé « `shallowCopy()` » est similaire à la méthode `clone()` de Java. Les attributs de l'instance sont dupliqués, mais pas les instances pointées. La deuxième stratégie (`deepCopy()`) au contraire, duplique une instance et toutes les instances qu'elle référence.

Ces approches sont absurdes. En effet, elle impose que pour un objet, il n'y a que des relations (`shallowCopy()`) ou que des agrégations (`deepCopy()`). Ce cas est rarissime. Une instance est composée de relation et d'agrégation. Si je construis un jumeau, il a sa propre adresse, et le même père que son frère, pas un père jumeau !

### 13.2 Cela complexifie le langage

Ajouter les trois boucles (`for` ; `while` ; `do while`) a-t-il complexifié les langages ? Avant il n'y avait que des `goto`...

Un langage est difficile à utiliser si on n'arrive pas à exprimer clairement ce que l'on désire. Par exemple, le C ou le C++ proposent une seule syntaxe pour quatre concepts différents. Un pointeur peut représenter

- une relation sur une seule instance (l'arithmétique est alors interdite),
- une relation sur plusieurs instances (l'arithmétique est autorisée),
- une agrégation sur une instance (pas d'arithmétique, notion d'adoption, de duplication, gestion mémoire déterministe) ou
- une agrégation sur plusieurs instances (idem que pour une seule instance, sauf qu'il faut supprimer le tableau à l'aide de `delete []`).

S'il y avait des syntaxes radicalement différentes pour ces quatre notions, la qualité des programmes s'améliorerait.

Ajouter des contraintes à un langage permet des vérifications impossibles autrement. Pourquoi le C++ et Java sont-ils des langages typés ? Pour éviter d'avoir une règle du type « il ne faut pas mélanger les types des variables ». En effet, même s'il est possible d'écrire un programme sans indiquer de type (cas de VB, Smalltalk, ECMAScript,...), à l'usage, il y a de nombreuses erreurs à cause de cela.

Pourquoi y a-t-il des droits d'accès aux attributs et aux méthodes ? Pourquoi toutes les méthodes ne sont-elles pas publiques ? Cela n'empêche pas d'écrire un programme correct. Les droits sont présents pour aider le développeur. Cela permet au compilateur de vérifier la règle : « Certaines méthodes ne peuvent pas être utilisées dans certaines situations ».

L'être humain n'est pas infallible, il lui faut des outils pour l'aider. En lui permettant d'exprimer clairement ces besoins, le développeur se protège de ses propres erreurs. La loi de Murphy est ainsi faite, que s'il est possible de violer une règle, elle le sera.

Il est vrai que cela ajoute de nouveaux concepts aux langages. L'informatique n'est pas une science simple. Ce n'est pas parce qu'un langage propose peu de concept qu'il est plus simple à utiliser. Dans mon livre sur les trois langages objets « C++, Java et Smalltalk », je décris comment contourner l'absence de concept dans certains langages. Par exemple, Smalltalk ne possède pas de syntaxe pour la gestion du flux de traitement (`for`, `while`, ...). Il résout cela à l'aide d'une méthode récursive. En réalité, comme cette approche ne peut pas fonctionner en réalité, le compilateur est obligé d'utiliser un code caché interne utilisant une boucle classique. Il n'est pas possible d'écrire la méthode `whileTrue` : avec Smalltalk. Il faut donc un concept supplémentaire au langage, même s'il n'apparaît généralement pas pour l'utilisateur.

Les langages ont des caractéristiques qui les rendent pertinents pour certains développements mais pas pour d'autre. Pour écrire un moteur expert, le [Lisp](#) ou [Prolog](#) sont de bon candidat. Pour manipuler des chaînes de caractères, [awk](#) et [perl](#) sont préférables. Il faut utiliser le meilleur outil suivant l'objectif de développement. Les langages généralistes comme Java ou C++ structure les développements en proposant le paradigme objet. Celui-ci apporte avec lui la notion d'agrégation. Il semble alors envisageable d'ajouter ce concept aux langages.

### **13.3 Les comités de normalisation n'accepteront jamais cela**

Ce n'est pas l'objectif pour le moment. Il faut simplement vérifier la validité de ces idées. C++ et Java sont de bon candidat. Une autre approche consiste à concevoir un nouveau langage s'appuyant sur ce concept. Dans un premier temps, il s'agit d'une *recherche* en science informatique.

J'ai déjà pu apprécier ces idées lors de projets C++. Je ne pouvais pas bénéficier de tous les concepts décrits ici, mais j'ai écrit des programmes sans jamais utiliser le mot-clef [delete](#).

### **13.4 L'agrégation n'existe pas dans le monde réel**

En effet, la notion d'agrégation n'existe pas dans le monde réel au même titre que la notion d'objet. Le paradigme objet est une traduction informatique de comment l'être humain appréhende le monde. Nous créons des concepts qui n'existent pas dans la nature. Deux voitures ne sont pas des objets semblables. L'une est composée d'atome agencé différemment de la deuxième. Ils ne sont pas à la même place, n'ont pas les mêmes interactions. Une voiture démarre sans problème, l'autre demande une accélération plus poussée.

C'est nous, les humains, qui forgeons la notion de voiture pour faciliter notre compréhension du monde. Le monde n'a pas besoin de nous pour exister. Les concepts que nous manipulons ne sont pas nécessaires. Ils sont variables d'un lieu à l'autre, d'une époque à une autre. Ils sont différents suivants l'objectif que nous nous fixons. Pour un médecin ORL, le visage appartient à un nez et non l'inverse ! Doit-on dire « on lui a coupé la tête » ou « on lui a coupé le corps » ? Le paradigme objet ne modélise pas le monde, mais une façon de l'appréhender. Il est en contradiction avec l'axiome « rien ne se crée, tous se transforment ».

L'agrégation comme le paradigme objet n'a pas besoin d'exister dans le monde réel pour avoir une application pratique. La notion de possession correspond à notre appréhension du monde. En réalité, rien n'appartient à personne. Un moteur n'appartient pas plus à une voiture qu'une voiture appartient à un moteur. Par contre, il est aisé et efficace de considéré que le moteur appartient à la voiture. Cela permet de simplifier le monde et nous permet de l'exploiter.

## **14. CONCLUSION**

Ces nouvelles syntaxes permettent d'indiquer précisément l'utilisation qui sera faite d'un attribut, d'un paramètre ou d'un retour de méthode. Ces informations sont à une très large majorité absente des documentations des APIs. L'expressivité du langage est fortement améliorée, mais de plus, l'utilisation du concept d'agrégation est vérifiée par le compilateur. Une très grande partie des erreurs mémoires sont dues à une mauvaise traduction de ce concept. Ces additifs permettent d'améliorer la qualité des programmes.

La gestion de la mémoire est grandement améliorée. Un programme C++ peut ne pas avoir de [delete](#) et une machine virtuelle Java peut réduire considérablement les candidats aux ramasses miettes.

Les traductions proposées n'ont jamais été codées pour le moment. Peut-être y a-t-il des situations particulières devant remettre en cause certains des principes évoqués dans ce document. Lors de la rédaction des préprocesseurs pour C++ ou Java il faudra vérifier la validité de ces idées.

Il faut maintenant trouver un laboratoire acceptant de financer ce projet. Si vous connaissez une entreprise ou une université intéressée, contactez-moi ([pp@philippe.prados.name](mailto:pp@philippe.prados.name)) !

Dans l'immédiat, pour améliorer la qualité de vos programmes, il est nécessaire de documenter chaque pointeur pour signaler si celui-ci est une relation ou une agrégation. De même, les commentaires des paramètres d'une méthode doivent indiquer si la méthode utilisera, adoptera ou dupliquera le paramètre.