# Programmation orientée objet

Composition et agrégation

Samuel Kadoury

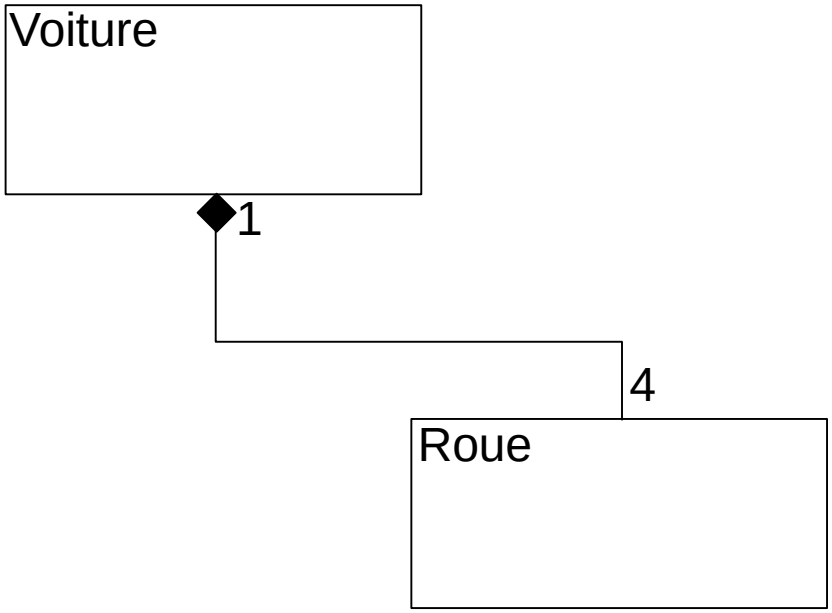
# Composition

- La composition consiste à utiliser un objet comme attribut d'un autre objet
- Par exemple, la classe *Voiture* contiendra quatre objets de la classe *Roue*:

```
class Roue {
public:
    ...
private:
    ...
};
```

```
class Voiture {
public:
    ...
private:
    Roue roueAvantGauche_;
    Roue roueAvantDroite_;
    Roue roueArriereGauche_;
    Roue roueArriereDroite_;
    ...
};
```

# Composition (représentation en UML)



# Composition

- Si un objet A est un attribut de l'objet B, le constructeur de l'objet A sera appelé **avant** celui de l'objet B.
- Si vous réfléchissez bien, ceci est logique: pour construire une voiture, il faut d'abord construire ses composantes, comme le moteur et les roues.
- On dit que A et B sont reliés par une relation de **composition**, c'est-à-dire que B est composé de A
- Il s'agit d'une relation forte: si B est détruit, A disparaît aussi

# Initialisation d'un objet composite

- Supposons que l'on veuille définir un contructeur de company qui prend comme paramètre le nom de la compagnie et le nom du président:
- Par exemple, le nom de la compagnie "Polytechnique" et sa présidente "Michele":

```
Company poly("Polytechnique","Michele");
```

# Initialisation d'un objet composite

- Soit la classe Company suivante:

```
class Company
{
public:
    Company();
                ...
private:
    string name_;
    Employee president_;
    ...
};
```

# Initialisation d'un objet composite (suite)

- Nous pourrions définir le constructeur suivant:

```
Company::Company(string name, string presidentName)
{
   name_ = name;
   president_ = Employee(presidentName);
   ...
}
```

←→ Par contre: Qu'est-ce qui se passe?

Tous les attributs sont déjà initialisés **avant** que le corps d'un constructeur commence à exécuter!

Comment éviter la construction redondante des attributs?

# Initialisation d'un objet composite (suite)

- Pour résoudre ce problème, on aimerait que, lorsqu'un objet de la classe Company est créé, ce soit non pas le constructeur par défaut qui soit appelé pour l'attribut **`president_`**, mais plutôt son constructeur par paramètres auxquel on passerait le nom
- Pour y arriver, il faut utiliser une liste d'initialisation

# Liste d'initialisation

```
Company::Company(string name, string
  presidentName):president_(presidentName)
{
  name_ = name;
  ...
}
```

Au lieu de créer l'attribut **`president_`** en utilisant le constructeur par défaut de la classe Employee, on le construira plutôt en lui passant le paramètre presidentName

# Liste d'initialisation

```
Company::Company(string name, string
  presidentName): name_(name),
  president_(presidentName)...
{
}
```

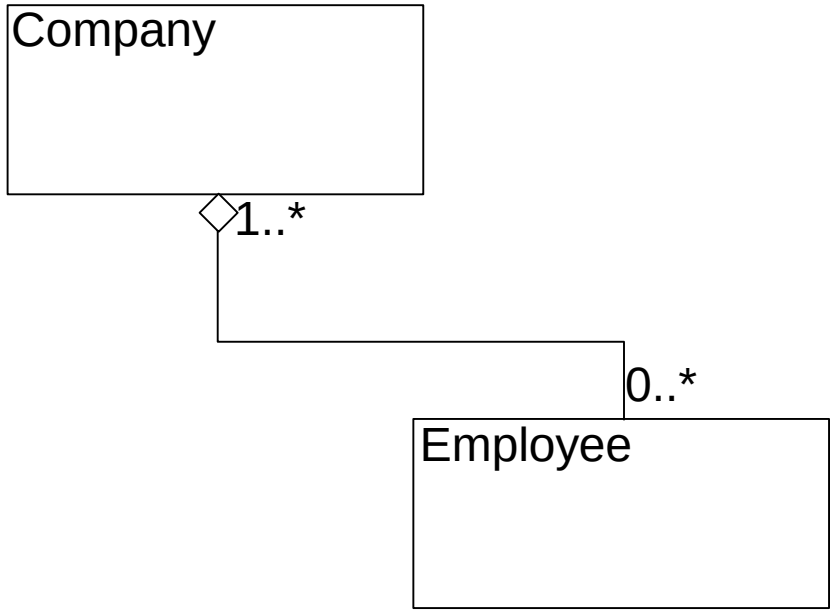
Tant qu'à y être, on peut aussi initialiser les autres attributs.

**Attention**: l'ordre de construction, donc initialisation, des membres est **selon la déclaration dans la classe** (pas l'ordre de la liste d'initialisation). Pour faciliter la compréhension, la liste devrait être dans le même ordre que les déclarations des attributs.

# Agrégation

- L'agrégation consiste essentiellement en une utilisation d'un objet comme faisant partie d'un autre objet
- Contrairement à la composition, où l'objet inclus disparaît si l'objet englobant est détruit, **dans une agrégation cet objet ne disparaît pas lorsque l'objet englobant est détruit**
- La manière habituelle d'implémenter l'agrégation en C++ est par l'utilisation de pointeurs

# Agrégation (représentation en UML)



# Agrégation par pointeur

- Définition de la classe:

```
class Company
{
   ...
private:
   string name_;
   Employe** employees_;
   int nbEmployees_;
}
```

- Constructeur:

```
Company::Company()
  : name_("unknown"),
    employees_(nullptr),
    nbEmployees_(0)
{
}
```

- Méthode pour ajouter un employé:

```
void Company::addEmployee(Employe* employe)
{ ...
  employees_[nbEmployees_++] = employee;
}
```

# Agrégation par pointeur (suite)

- Pour qu'un objet de la classe Company soit réellement intéressant, il nous faut une méthode pour lui associer un employé

- Ceci suppose qu'un objet de la classe Employee a déjà été créé auparavant et que la méthode fera pointer son pointeur sur cet objet

# Agrégation par pointeur (suite)

- Dans la fonction principale:

```
int main()
{
   ...
  Company poly("Polytechnique”, ”Michele");
  Employe* samuel = new Employe(”Samuel Kadoury", 50000.0);
  poly.addEmployee(samuel);
   ...
}
```

# Agrégation par pointeur (suite)

- On pourrait aussi utiliser un paramètre supplémentaire dans le constructeur:

```
Company::Company(string name, Employe* employe)
  : name_(name), president_(employe)
{
}

int main()
{
 ...
 Employe* michele = new Employe("Michele", 50000.0);
 Company poly ("Polytechnique",michele);
 ...
}
```

# Agrégation par référence

- Lorsqu'on fait une agrégation par référence, la **référence doit absolument être initialisée lors de la création de l'objet**
- En d’autres mots: l’objet requiert la référence pour son fonctionnement (≠ par pointeur)
- L'objet doit donc exister déjà lorsqu'on crée une instance de la classe englobante

# Agrégation par référence

- Définition de la classe:

```
class Company
{
   ...
private:
   string name_;
   Employe& president_;
}
```

- Constructeur:

```
Company::Company(string name,
                 Employe& employe)
    : name_(name),
      president_(employe)
{
}
```

Une fois l'objet créé, la référence désignera toujours le même objet.

# Agrégation par référence (suite)

- Dans la fonction principale:

```
int main()
{
   ...
   Employe employe("Michele", 50000.0)
   Company poly("Polytechnique", employe);
   ...
   employe.setSalary(60000.0);
}
```

Comme l'attribut interne de l'objet **poly** réfère au même objet, il verra lui aussi la modification du salaire.

L'attribut interne sera une référence à cet objet.

# Composition vs. agrégation

| | Composition | Agrégation (référence) | Agrégation (pointeur) |
|---|---|---|---|
| B | **Requiert** A | **Requiert** A | Peut survivre sans A |
| A | Partie de B | Partie de ≥1 Bs | Partie de ≥1 Bs |
| Durée de vie d'A | Contrôlée par B | **Indépendente** de B | **Indépendente** de B |