# Programmation orientée objet

## Composition

Samuel Kadoury & Claudy Picard

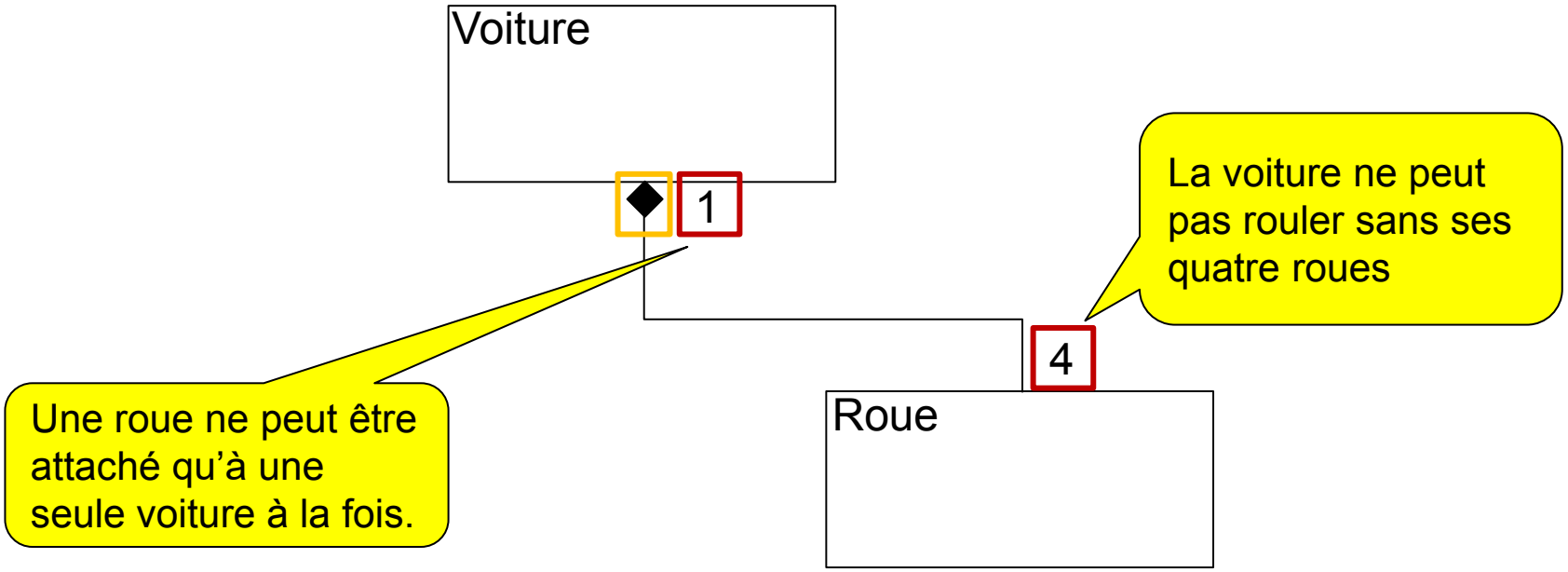
# Composition

- La composition est une relation de possession (relation “**a un**” ou “**est composé de**”) entre un objet composite (objet englobant) et sa ou ses composantes (objet(s) englobé(s))
- Par exemple:
  - Une voiture a quatre roues
  - Un ordinateur a une carte-mère
  - Un train a des sièges
  - Un humain a un coeur

# Composition

- Il s'agit d'une relation forte, l'objet englobant gère la mémoire de ses composantes
- Lorsque l'objet englobant est détruit, ses composantes sont automatiquement détruites, leur **durée de vie est dépendante** de celle de l'objet englobant
- Un objet englobé n'appartient qu'à un seul objet composite à la fois, l'objet composite est propriétaire unique de ses composantes

# Composition par valeurs



# Composition par valeurs

- Il y a composition par valeurs  lorsque l'objet englobant a absolument besoin de l'objet englobé pour exister
- 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.

# Composition par valeurs – Exemple avec Voiture

```
class Voiture {
public:
    Voiture(): roues_(make_unique<Roue[]>(4)) {}

private:
    unique_ptr<Roue[]> roues_;
};

int main() {
    Voiture voiture;
}
```

Les quatres roues sont construites par défaut avant même que l'objet voiture soit construit

La classe voiture est responsable d'allouer la mémoire pour les roues

Lorsque l'objet voiture est détruit, les quatre roues sont aussi détruites

# Composition par valeurs – Exemple avec Company

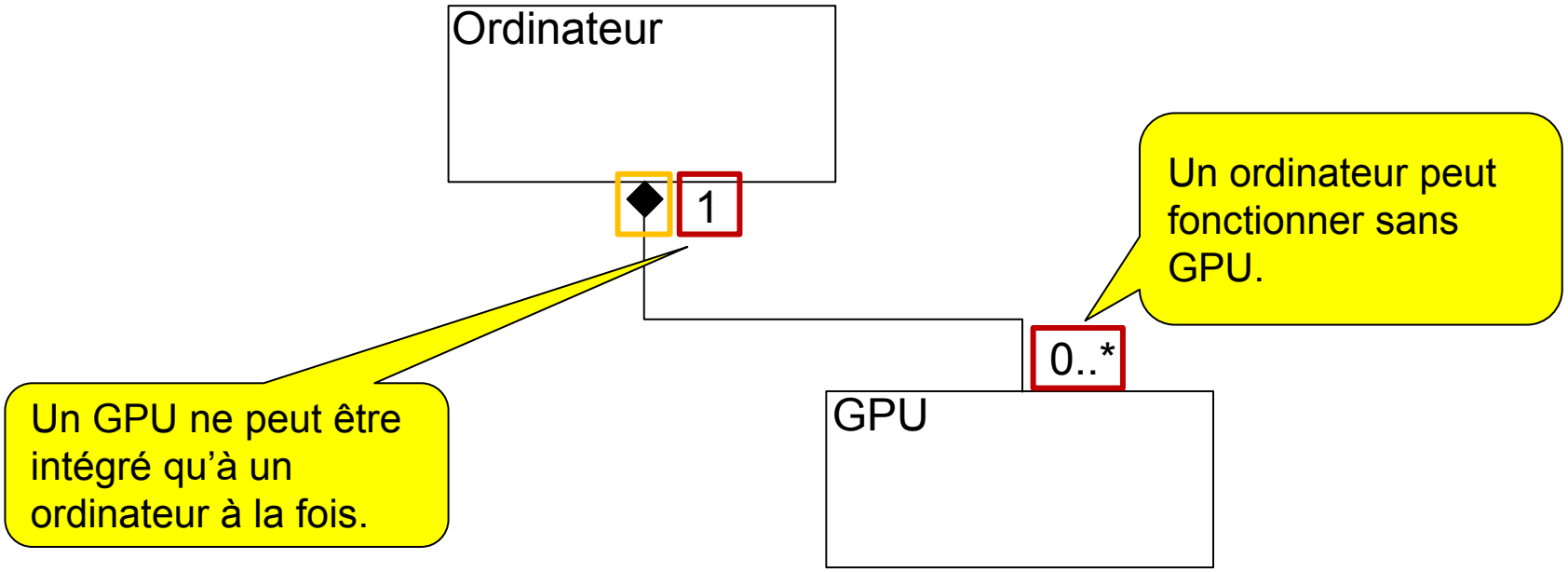
```
class Company {
public:
    Company() {}
    Company(string nomPresident): president_(nomPresident) {}

private:
    Employee president_;
};

int main() {
    Company comp;
    Company poly("Bob");
}
```

L'objet président_ est construit en utilisant le constructeur par défaut de Employee

L'objet président_ est construit en utilisant le constructeur par paramètres de Employee

Lorsque les objets de la classe Company sont détruits, leur président est aussi détruit

# Composition par pointeurs



# Composition par pointeurs

- Il y a composition par pointeurs  lorsque l'objet englobant **n'a pas besoin** de l'objet englobé pour exister
- Cette relation est généralement implémentée en utilisant des **pointeurs intelligents uniques**
- L'objet englobant est donc le propriétaire unique de ses composantes et est responsable de l'allocation et de la désallocation de leur mémoire

# Composition par pointeurs – Exemple avec Ordinateur

```
class Ordinateur {
public:
    void ajouterGPU(string fabriquant) {
        gpus_.push_back(make_unique<Gpu>(fabriquant));
    }
private:
    vector<unique_ptr<Gpu>> gpus_;
};
int main() {
    Ordinateur ordi;
    ordi.ajouterGPU("Nvidia");
}
```

L'ordinateur alloue l'espace mémoire et créé le GPU en utilisant le constructeur par paramètres

L'ordinateur est créé sans GPU.

Lorsque l'objet ordi est détruit, tous ses GPUs sont aussi détruits

# Programmation orientée objet

## Agrégation

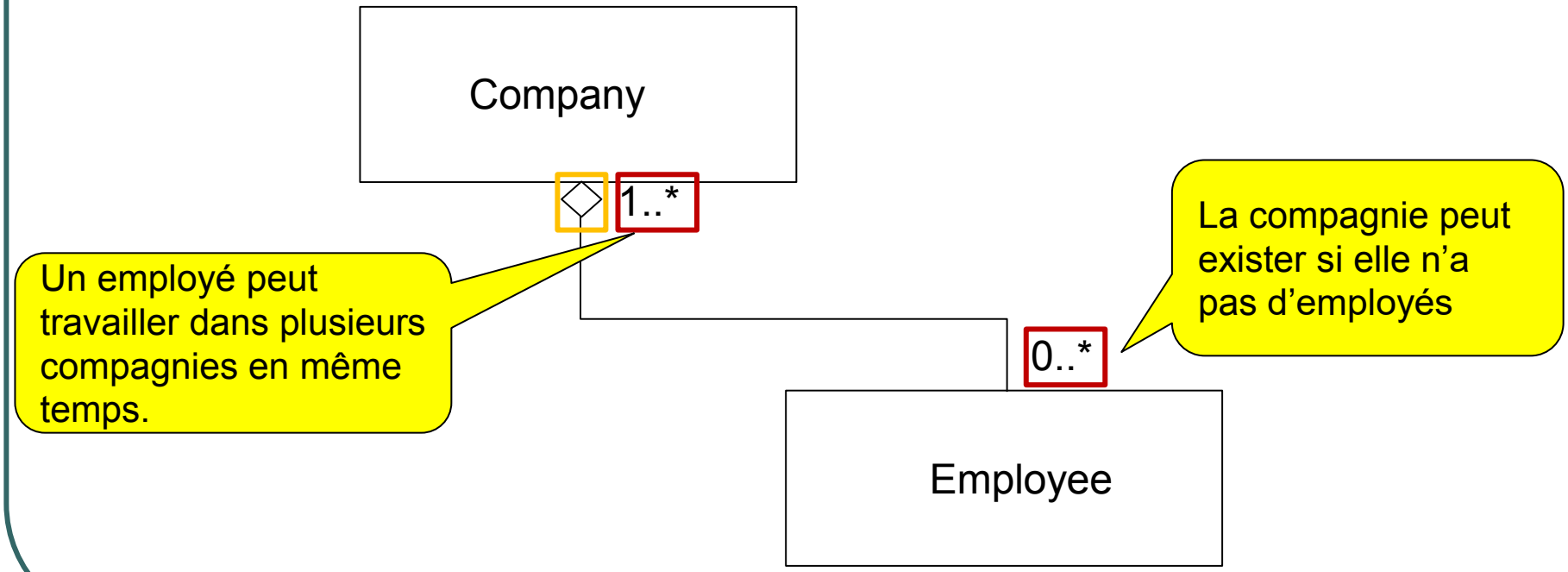
Samuel Kadoury & Claudy Picard

# Agrégation

- L'agrégation consiste essentiellement en une utilisation (relation "**utilise un**") d'un objet comme faisant partie d'un autre objet
- L'objet englobé par agrégation n'est pas détruit lorsqu'il n'est plus utilisé par un agrégat, sa **durée de vie est indépendante** de celle de l'objet englobant
- Un objet peut être utilisé par plusieurs agrégats en même temps, il peut donc y avoir partage de mémoire entre les objets englobants

# Agrégation par pointeurs

- Un objet de la classe Company **utilise** des objets de la classe Employee **au besoin**



# Agrégation par pointeurs

- Il y a agrégation par pointeurs lorsque l'objet englobant n'a pas besoin de l'objet englobé pour exister.
- Cette relation est généralement implémentée en utilisant des **pointeurs intelligents partagés**
- Il n'y a aucune allocation et désallocation de mémoire qui se fait lors d'une agrégation par pointeurs

# Agrégation par pointeurs – Exemple implémentation

```
class Company {
public:
    Company(): president_(nullptr) {}
    Company(const shared_ptr<Employee>& emp):
    president_(emp) {}

    void ajouterEmployee(const shared_ptr<Employee>& emp){
        employees_.push_back(emp);
    }

private:
    vector<shared_ptr<Employee>> employees_;
    shared_ptr<Employee> president_;
};
```

# Agrégation par pointeurs – Exemple implémentation

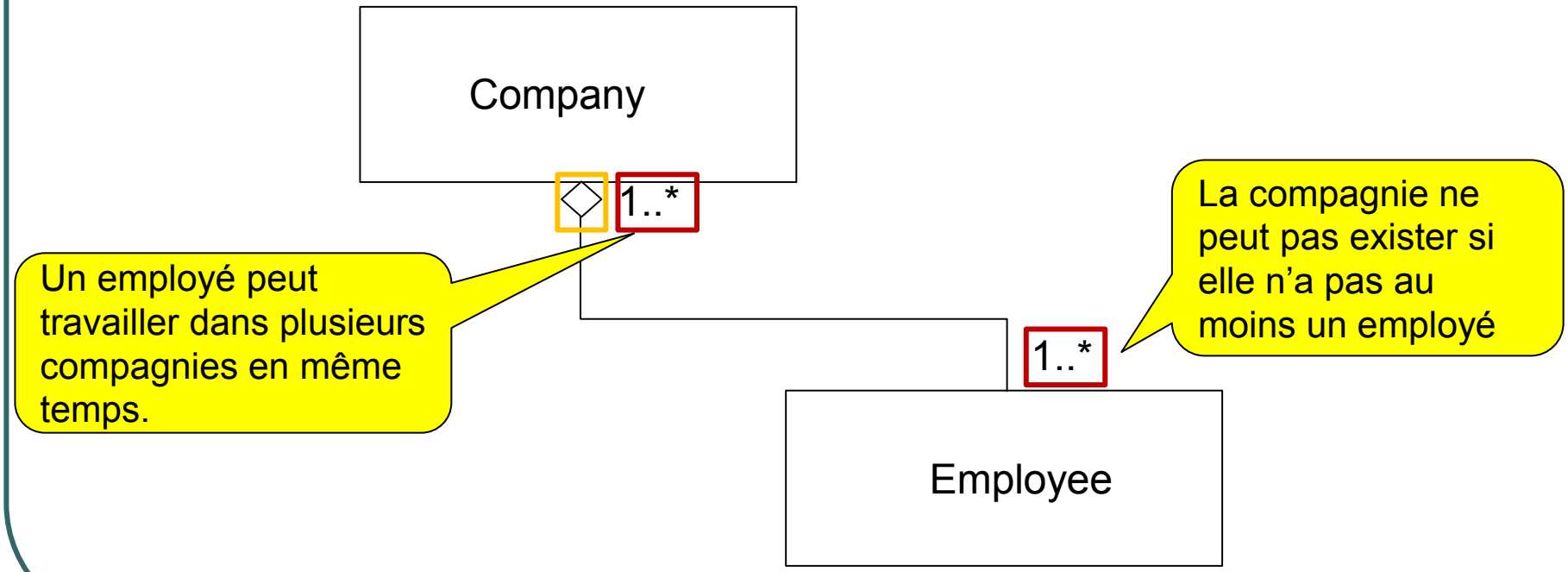
```
int main() {
    shared_ptr<Employee> bob = make_shared<Employee>("Bob", 10000);
    Company poly;

    poly.ajouterEmployee(bob);
}
```

bob est créé par le main et donc sa durée de vie est indépendante des compagnies qui l'utilisent

Lorsqu'on ajoute bob à poly, son compteur de référence est incrémenté

# Agrégation par référence

- Un objet de la classe Company **utilise au moins un** objet de la classe Employee



# Agrégation par référence

- Il y a agrégation par référence lorsque l'objet englobant **a absolument besoin** de l'objet englobé pour exister.
- La durée de vie de l'objet englobé reste indépendante de celle de l'objet englobant, mais elle doit être plus grande que celle de l'objet englobant

# Agrégation par référence

- Lorsqu'on fait une agrégation par référence, la **référence doit absolument être initialisée avant même que l'objet englobant soit construit**
- La référence doit donc absolument être **initialisée dans la liste d'initialisation** sinon il y aura erreur de compilation

# Agrégation par référence

```
class Company {
public:
    Company(Employee& emp): president_(emp) {}

private:
    Employee& president_;
};

int main() {
    Employee bob("Bob", 10000);
    Company poly(bob);
}
```

La classe Company ne peut pas avoir de constructeur par défaut

L'attribut interne sera une référence à cet objet.

Comme l'attribut interne de l'objet **poly** réfère au même objet, toute modification sur bob affectera l'attribut interne et vice versa

*Programmation orientée objet*

# Composition vs. Agrégation

Raphaël Beamonte, 2014

# Composition ou agrégation?

```
class MaClasse {
    public:
        MaClasse();
        ~MaClasse();
        ...
    private:
        Point attribut_;
}
```

# Composition ou agrégation?

```
class MaClasse {
    public:
        MaClasse();
        ~MaClasse();
        ...
    private:
        Point attribut_;
}
```

Composition

# Composition ou agrégation?

```
class MaClasse {
    public:
        MaClasse();
        ~MaClasse();
        ...
    private:
        Point &attribut_;
}
```

# Composition ou agrégation?

```
class MaClasse {
    public:
        MaClasse();
        ~MaClasse();
        ...
    private:
        Point &attribut_;
}
```

Agrégation par référence

# Composition ou agrégation?

```
class MaClasse {
    public:
        MaClasse();
        ~MaClasse();
        ...
    private:
        Point *attribut_;
}
```

# Composition ou agrégation?

```
class MaClasse {
    public:
        MaClasse();
        ~MaClasse();
        ...
    private:
        Point *attribut_;
}
```

Agrégation par pointeur?

# Composition ou agrégation?

```
class MaClasse {
    public:
        MaClasse();
        ~MaClasse();
        ...
    private:
        Point *attribut_;
}
```

```
MaClasse::MaClasse() {
    attribut_ = new Point[2];
}

MaClasse::~MaClasse() {
    delete [] attribut_;
}
```

Composition!

# Composition ou agrégation?

```
class MaClasse {
    public:
        MaClasse();
        ~MaClasse();
        ...
    private:
        Point *attribut_;
}
```

```
MaClasse::setAttribut(
                Point *attribut) {
    attribut_ = attribut;
}

MaClasse::~MaClasse() {
}
```

Agrégation par pointeur!

# Composition ou agrégation?

| Attribut | Objet | Référence | Pointeur |
|---|---|---|---|
| **Type de relation** | Composition | Agrégation par référence | Agrégation par pointeur **OU** Composition |
| **Que regarder?** | Définition | Définition | Définition **ET** Implémentation |