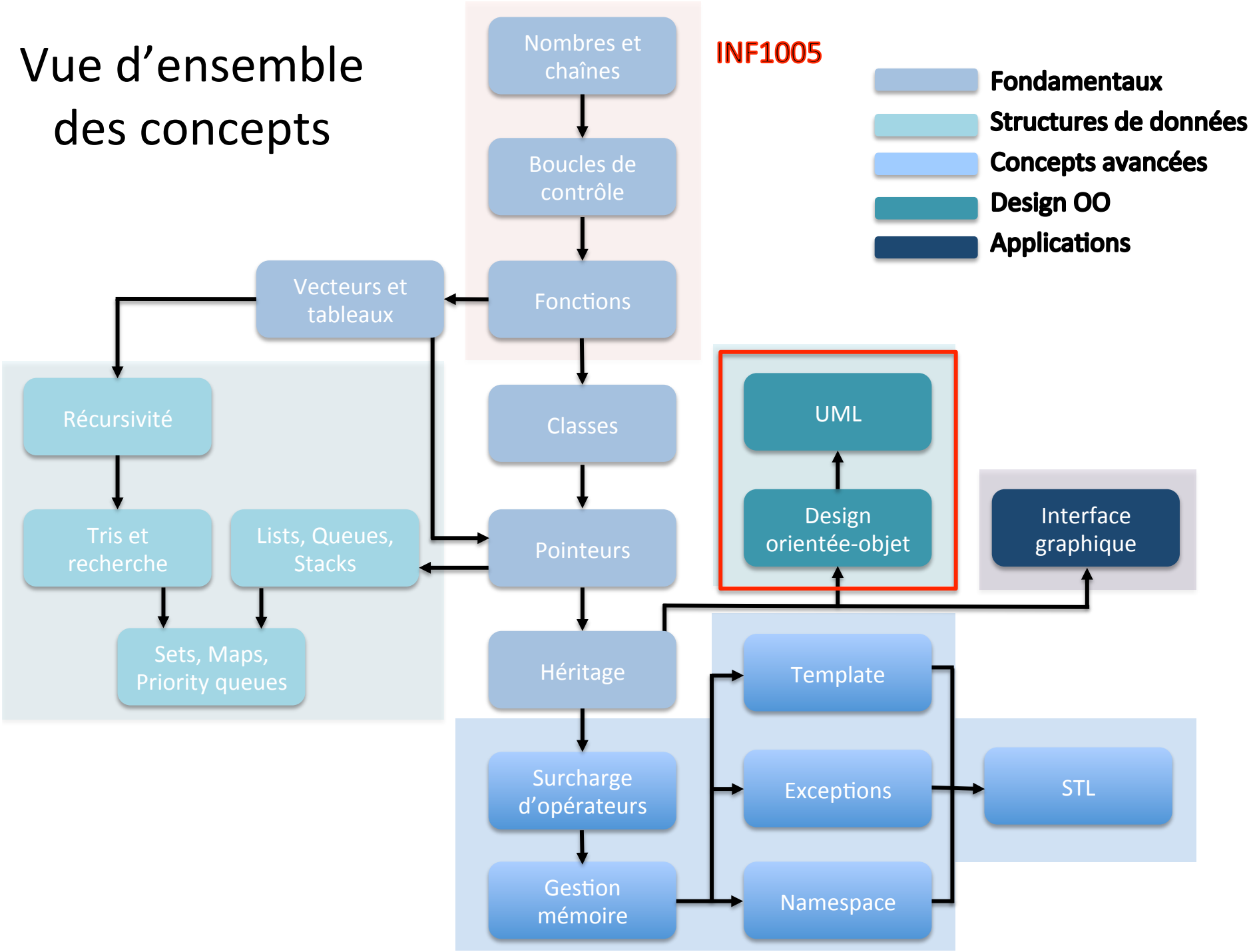
Programmation orientée objet

Composition et agrégation

Michel Gagnon

Vue d'ensemble des concepts



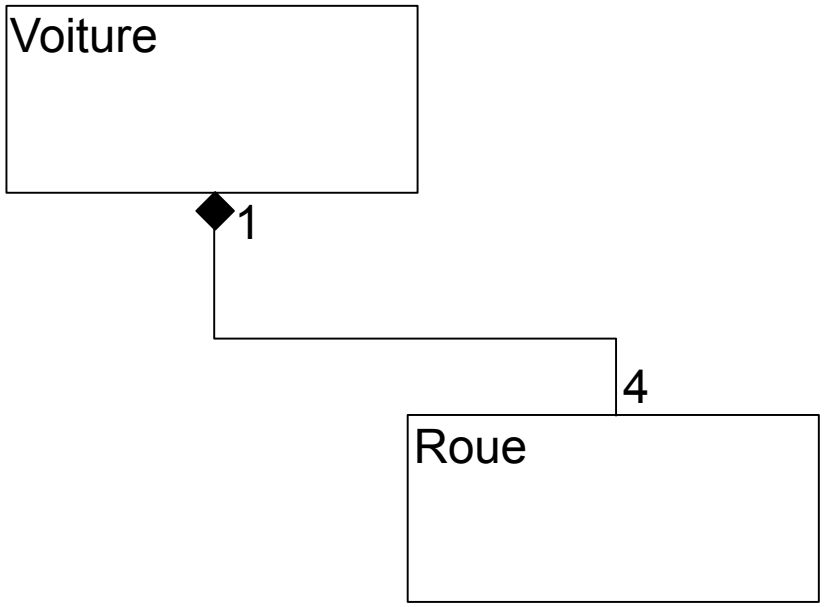
Composition

- 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 cercle qui prend comme paramètre les coordonnées *x* et *y* de son centre et son rayon:
- Par exemple, un cercle de rayon 2 dont le centre est le point (3,4):

```
Cercle unCercle(3,4,2);
```

Initialisation d'un objet composite

- Soit la classe Cercle suivante:

```
class Cercle
{
public:
   Cercle();
            ...
private:
   Point centre_;
   double rayon_;
};
```

Initialisation d'un objet composite (suite)

- Nous pourrions définir le constructeur suivant:

```
Cercle::Cercle(double x, double y, double rayon)
{
  centre_ = Point(x,y);
  rayon_ = rayon;
}
```

←→ 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?

Liste d'initialisation

```
Cercle::Cercle(double x,
               double y,
               double rayon)
    : centre_(x,y)
{
    rayon_ = rayon;
}
```

Au lieu de créer l'attribut **centre_** en utilisant le constructeur par défaut de la classe Point, on le construira plutôt en lui passant les paramètres x et y.

Liste d'initialisation

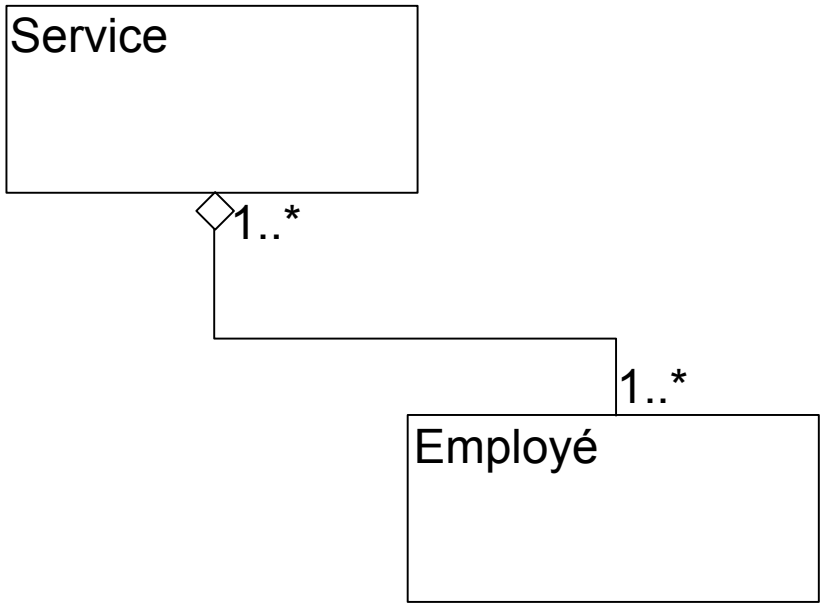
```
Cercle::Cercle(double x,
               double y,
               double rayon)
    : centre_(x,y), rayon_(rayon)
{
}
```

Tant qu'à y être, on peut aussi initialiser les autres 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 Service
{
   ...
private:
   string nom_;
   Employe*
    receptionniste_;
}
```

- Constructeur:

```
Service::Service(string nom)
  : nom_(nom),
    receptionniste_(0)
{
}
```

- Méthode pour associer le réceptionniste:

```
Service::setReceptionniste(Employe* employe)
{
  receptionniste_ = employe;
}
```

Agrégation par pointeur (suite)

- Pour qu'un objet de la classe Service soit réellement intéressant, il nous faut une méthode pour lui associer un employé comme réceptionniste
- Ceci suppose qu'un objet dynamique de la classe Employe a déjà été créé auparavant et que la méthode fera pointer son pointeur sur cet objet

Agrégation par pointeur (suite)

- Méthode pour associer un réceptionniste à un objet de la classe Service:

```
void Service::setReceptionniste(Employe* employe)
{
  receptionniste_ = employe;
}
```

Agrégation par pointeur (suite)

- Dans la fonction principale:

```
int main()
{
   ...
  Service expedition("Expeditions");
  Employe* michel = new Employe("Michel Gagnon", 50000.0);
  expedition.setReceptionniste(michel);
  ...
}
```

Agrégation par pointeur (suite)

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

```
Service::Service(string nom, Employe* employe)
  : nom_(nom), receptionniste_(employe)
{
}

int main()
{
 ...
 Employe* michel = new Employe("Michel Gagnon", 50000.0);
 Service expedition("Expeditions",michel);
 ...
}
```

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 Service
{
  ...
private:
  string nom_;
  Employe& receptionniste_;
}
```

- Constructeur:

```
Service::Service(string nom,
                 Employe& employe)
   : nom_(nom),
     receptionniste_(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("Michel Gagnon", 50000.0)
    Service expedition("Expeditions", employe);
    ...
     employe.setSalaire(60000.0);
}
```

Comme l'attribut interne de l'objet **expedition** 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