POLYTECHNIQUE MONTRÉAL

# CHAPITRE 14

## Patrons de conception : Composite, Itérateur et Proxy

# Sommaire

- Les patrons comme éléments architecturaux,
- Différents groupes de patrons,
- Concevoir un système à l'aide de patrons:
  - **Composite,**
  - **Itérateur,**
  - **Proxy**,
  - Décorateur,
  - Visiteur,
  - Méthode usine,
  - Commande,
  - Médiateur,
  - Singleton,
  - Méthode patron,
  - Stratégie,
  - État,
  - Façade,
  - Observateur.

# Les patrons comme éléments architecturaux

- En UML, un patron de conception est modélisé comme une collaboration paramétrée
  - Les paramètres de la collaboration sont les classes réelles qui vont participer à la collaboration,
  - La hiérarchie de classes associée à la collaboration décrit la structure de la collaboration en utilisant les noms des paramètres,
  - Les diagrammes d'interaction associés à la collaboration décrivent le comportement des éléments impliqués dans la collaboration en utilisant les noms des paramètres.

# Description d'un patron de conception

- Pour décrire les patrons, Gamma et al. proposent un mode de documentation uniforme dans lequel on retrouve:
  - Le **nom** du patron,
  - Son **intention**,
  - Une **motivation**,
  - Son **applicabilité**,
  - Sa **structure** et les classes/objets participants,
  - Les **collaborations** entre les participants,
  - **Conséquences** d'utilisation,
  - Remarques sur l'implantation,
  - Exemples de code,
  - Utilisations connues,
  - Patrons reliés.

# Classification des patrons

| | | **but** | | |
|---|---|---|---|---|
| | | **créationel** | **structurel** | **comportemental** |
| **portée** | **classe** | Factory method | Adapter | Interpreter |
| | | | | Template Method |
| | **objet** | Abstract Factory | Adapter | Chain of Responsability |
| | | Builder | Bridge | Command |
| | | Prototype | Composite | Iterator |
| | | Singleton | Decorator | Mediator |
| | | | Facade | Memento |
| | | | Flyweight | Observer |
| | | | Proxy | State |
| | | | | Strategy |
| | | | | Visitor |

Les 23 patrons de Gamma et al.

# Objectifs

- Se familiariser avec l'utilisation des patrons de conception,
- Apprendre à identifier les bons patrons,
- Comprendre leur applicabilité,
- Apprendre à adapter un patron à nos besoins,
- Apprendre à évaluer efficacement les compromis durant la conception.
- Éviter de réinventer la roue !!!

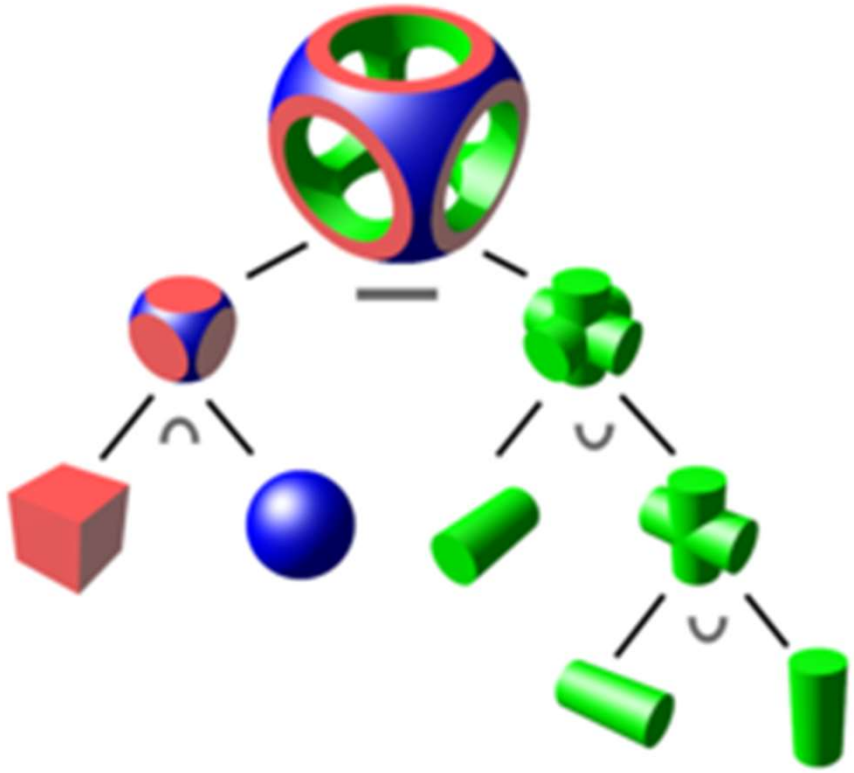
# Conception d'une application d'édition d'icônes 3D

## Éléments à considérer pour la conception de l'application. Comment allons-nous :

1. Représenter les primitives,
2. Itérer sur les primitives,
3. Représenter une sélection de primitives,
4. Appliquer des transformations aux primitives,
5. Ajouter des fonctionnalités aux primitives,
6. Construire les primitives,
7. Contrôler l'affichage des primitives,
8. Être avertis en cas de changement d'une icône,
9. Appliquer des modifications aux primitives,
10. Sauvegarder les primitives en différents formats,
11. Fournir une interface simple d'utilisation des icônes.

# 1 – Représenter les primitives

- Problème de conception:
  - Représenter les différents types de primitives (sphère, cylindre, cube, tore, polyèdre, etc.),
  - *Pour les usagers*: icônes de grosseur et de complexité arbitraires,
  - *Pour les programmeurs*: faciles à manipuler et à étendre



http://commons.wikimedia.org/wiki/Image:Csg_tree.png

# 1 – Représenter les primitives

- Une structure en arbre suggère un patron **Composite**
  - On cherche un patron structurel
  - On veut maximiser l'uniformité et la flexibilité

- Comment appliquer le patron
  - Choisir les participants: **AbsractComponent**, **Feuille** et **Composite**
  - Choisir les opérations à traiter de façon uniforme

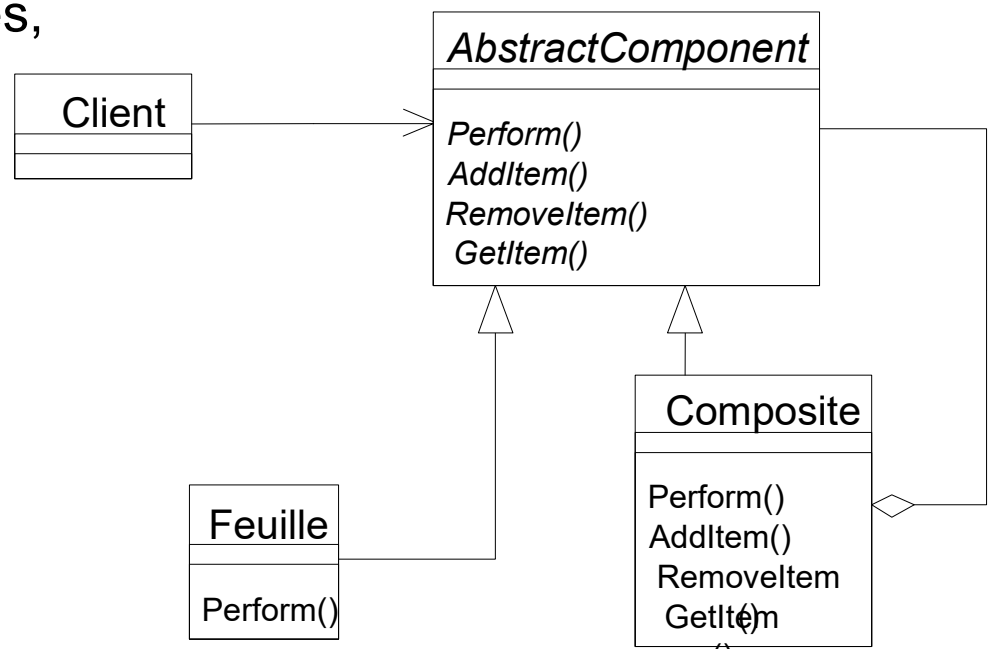
# Patron **Composite**

- Intention

  Traiter les objets individuels et les objets multiples, composés récursivement, de façon uniforme.

- Applicabilité
  - les objets doivent être composés récursivement,
  - les objets dans la structure peuvent être traités uniformément,
  - les clients peuvent ignorer les différences entre les objets individuels et composés,

- Structure



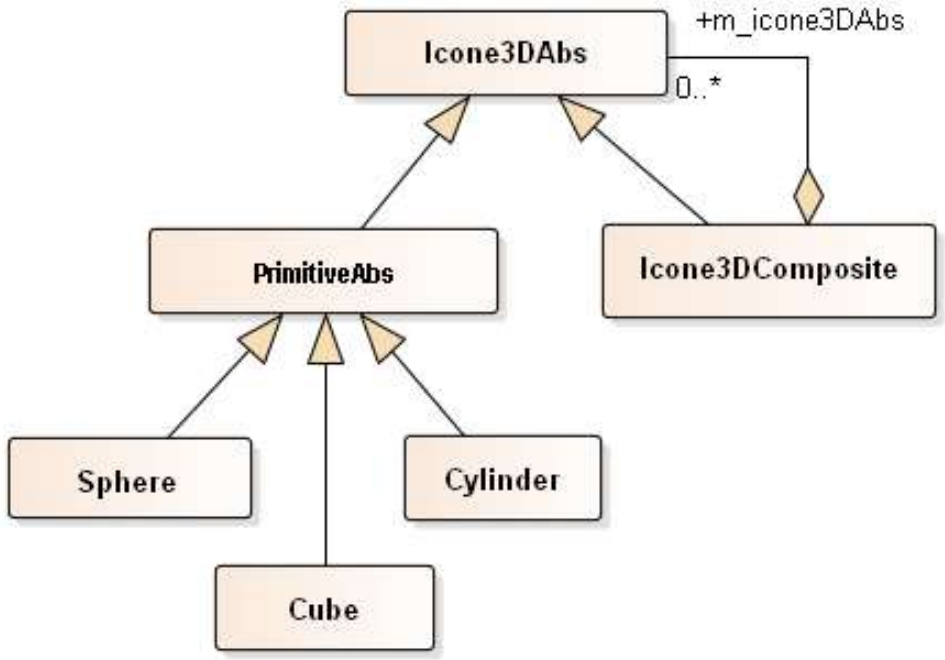
# Patron **Composite**

- Conséquences
  - +Uniformité: traite les composants uniformément sans égard à leur complexité.
  - +Extensibilité: les nouvelles sous-classes de Component fonctionnent partout où les anciennes fonctionnent.
  - - Coût: peut nécessiter un grand nombre d'objets
- Implantation
  - –Les Components connaissent-ils leur parent ?
  - –Quelle interface est-elle uniforme entre les Feuilles et Composites ?
  - –On ne doit pas allouer d'espace de stockage pour les enfants dans la classe de base AbstractComponent
  - –Qui est responsable de détruire les objets enfants ?

# Patron **Composite** : Primitives

Correspondance entre les participants au patron composite et les classes de primitives:

- AbstractComponent, l'interface uniforme
  - Icone3DAbs
- Composite, pour les objets qui ont des enfants
  - Icone3DComposite
- Feuilles, pour les objets qui n'ont pas d'enfants
  - sphère, cylindre, cube, etc.



# Patron **Composite** : Primitives

- Que peuvent faire les objets Primitive ?
  - retourner le centre, le nombre de paramètres, la valeurs des paramètres
  - modifier le centre, la valeur des paramètres

- Que peuvent faire les objets Objet3DComposite ?
  - retourner leur centre
  - énumérer leurs enfants
  - ajouter et retirer des enfants

# Patron **Composite** : Primitives

- Quelle interface uniforme la classe abstraite Icone3DAbs devrait-elle définir ?
  - –Retourner le centre
    - évidemment commun,
  - –Retourner/modifier les paramètres
    - moins évidemment commun,
  - –Énumérer les enfants
    - Nécessaire pour la récursion, il serait bon de cacher la structure interne,
    - Le patron Iterator pourra être utilisé,
  - –Adopter et abandonner des enfants
    - Compromis entre la sécurité du typage et l'uniformité.

# Patron **Composite** : Primitives

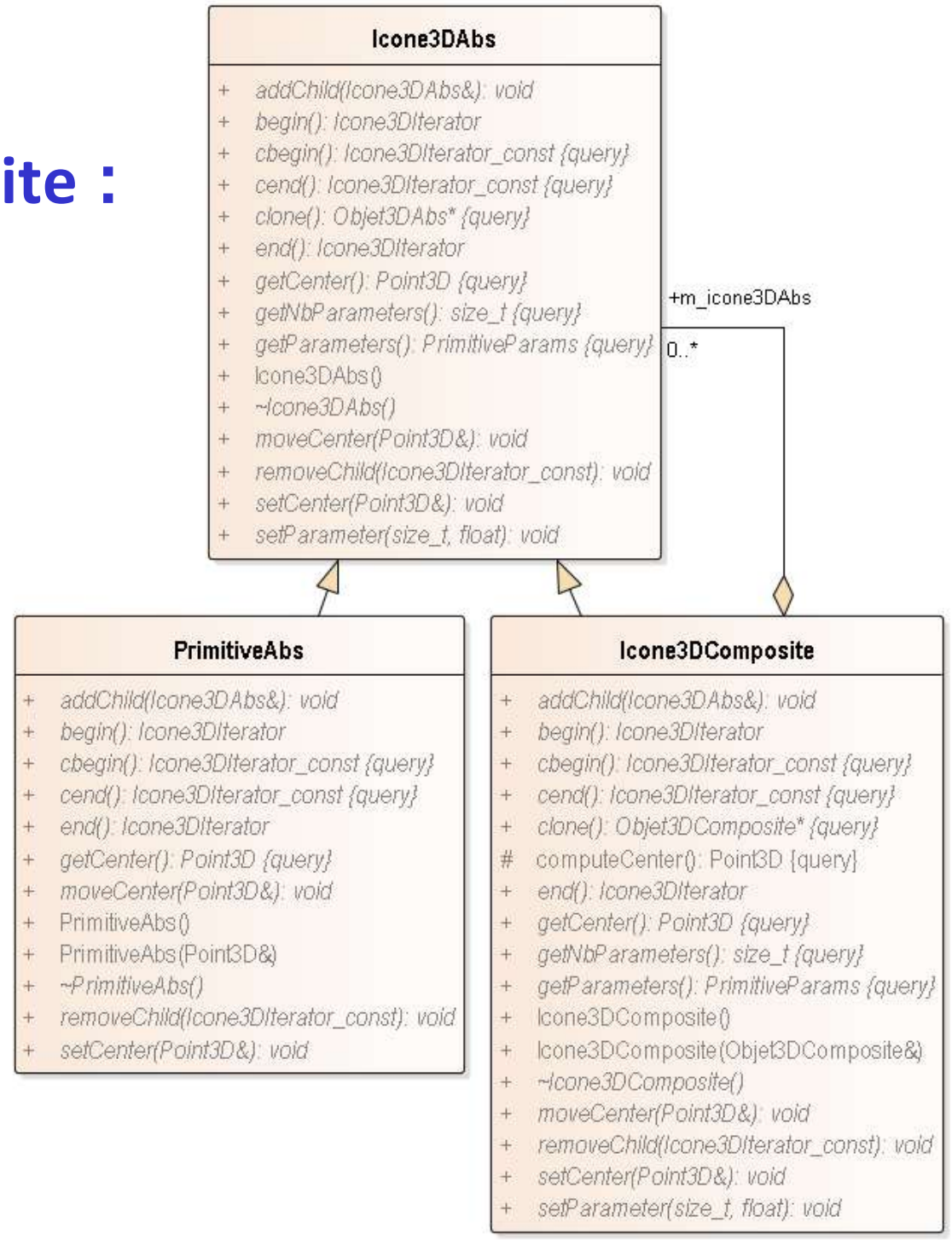
- Une interface uniforme pour ajouter et retirer des enfants simplifie les clients:

  Dans la mesure où les objets Feuille peuvent traiter ces requêtes élégamment.

- Solution: traiter **`addChild`** et **`removeChild`** de façon uniforme
  - On déclare les méthodes dans l'interface d'**`Icone3DAbs`**
  - On définit un comportement par défaut

```
class Icone3DAbs{
public:
  virtual void addChild( Icone3DAbs& )
  {
    cerr << "L'icône n'est pas un composite." << endl;
  }

  virtual void removeChild(Icone3DIterator_const it)
  {
    cerr << "L'icône n'a pas d'enfant." << endl;
  }
  [...]
};
```

# Patron **Composite** : Primitives



# Patron **Composite** : Icone3DAbs

```
class Icone3DAbs
{
public:
          Icone3DAbs() : m_parent(*this) {};
          Icone3DAbs( Icone3DAbs& parent) : m_parent(parent) {};
          virtual ~Icone3DAbs() = default;
          virtual Icone3DAbs* clone(Icone3DAbs& parent) const =0;

          virtual void addChild(const Icone3DAbs& obj3d) =0;
          virtual Icone3DIterator begin() =0;
          virtual Icone3DIterator_const cbegin() const =0;
          virtual Icone3DIterator_const cend() const =0;
          virtual Icone3DIterator end() =0;
          virtual void removeChild(Icone3DIterator_const obj3dIt) = 0;
          virtual void replaceChild(Icone3DIterator obj3dIt, const Icone3DAbs& newObj) = 0;

          virtual Point3D getCenter() const =0;
          virtual void moveCenter(const Point3D& delta) =0;
          virtual void setCenter(const Point3D& center) = 0;

          virtual size_t getNbParameters() const = 0;
          virtual PrimitiveParams getParameters() const = 0;
          virtual void setParameter(size_t pIndex, float pValue) =0;

          virtual bool isRoot() const { return (&m_parent == this); }
          virtual const Icone3DAbs& getParent() const { return m_parent; }
          virtual Icone3DAbs& getParent() { return m_parent; }

protected:
          Icone3DAbs& m_parent;
};
```

# Patron **Composite** : PrimitiveAbs

```
class PrimitiveAbs : public Icone3DAbs
{
public:
          PrimitiveAbs(const Point3D& pt);
          PrimitiveAbs(Icone3DAbs& parent);
          PrimitiveAbs(Icone3DAbs& parent, const Point3D& pt);
          virtual ~PrimitiveAbs();
          virtual PrimitiveAbs* clone(Icone3DAbs& parent) const = 0;

          // Toutes les methodes de gestion des enfants ne font rien
          virtual void addChild(const Icone3DAbs& obj3d);
          virtual Icone3DIterator begin();
          virtual Icone3DIterator_const cbegin() const;
          virtual Icone3DIterator_const cend() const;
          virtual Icone3DIterator end();
          virtual void removeChild(Icone3DIterator_const obj3dIt);
          virtual void replaceChild(Icone3DIterator obj3dIt, const Icone3DAbs& newObj);

          // Toutes les primitives ont un centre qui est gere dans la classe abstraite
          virtual Point3D getCenter() const;
          virtual void moveCenter(const Point3D& delta);
          virtual void setCenter(const Point3D& center);

protected:
          Point3D m_center;

private:
          static Icone3DContainer m_emptyContainer;
};
```

# Patron **Composite** : Sphere

```
class Sphere : public PrimitiveAbs
{

public:
          Sphere(const Point3D& pt, float r);
          Sphere(Icone3DAbs& parent, const Point3D& pt, float r);
          Sphere(Icone3DAbs& parent, const Sphere& sph);
          virtual ~Sphere();
          virtual Sphere* clone(Icone3DAbs& parent) const;

          virtual size_t getNbParameters() const;
          virtual PrimitiveParams getParameters() const;
          virtual void setParameter(size_t pIndex, float pValue);

private:
          float m_radius;
};
```

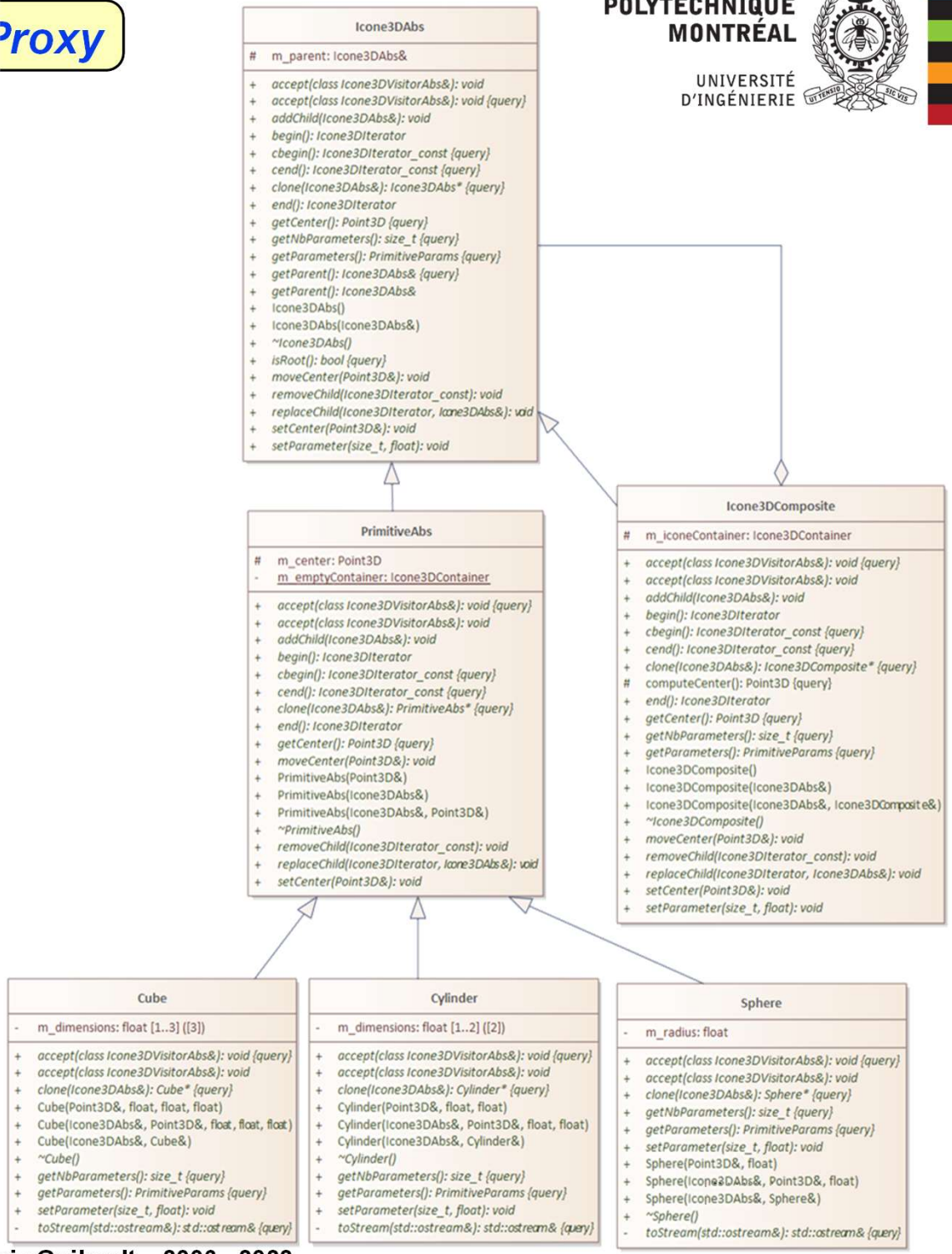
# Patron **Composite** : Icone3DComposite

```
class Icone3DComposite : public Icone3DAbs
{
public:
          Icone3DComposite();
          Icone3DComposite(Icone3DAbs& parent);
          Icone3DComposite(Icone3DAbs& parent, const Icone3DComposite& mdd);
          virtual ~Icone3DComposite();
          virtual Icone3DComposite* clone(Icone3DAbs& parent) const;

          virtual void addChild(const Icone3DAbs& obj3d);
          virtual Icone3DIterator begin();
          virtual Icone3DIterator_const cbegin() const;
          virtual Icone3DIterator_const cend() const;
          virtual Icone3DIterator end();
          virtual void removeChild(Icone3DIterator_const obj3dIt);
          virtual void replaceChild(Icone3DIterator obj3dIt, const Icone3DAbs& newObj);

          virtual Point3D getCenter() const;
          virtual void moveCenter(const Point3D& delta);
          virtual void setCenter(const Point3D& center);

          virtual size_t getNbParameters() const;
          virtual PrimitiveParams getParameters() const;
          virtual void setParameter(size_t pIndex, float pValue);

protected:
          Point3D computeCenter() const;
          Icone3DContainer m_iconeContainer;
};
```

# Patron **Composite** : Primitives

Correspondance entre les participants au patron composite et les classes de primitives:

- AbstractComponent, l'interface uniforme
  - Icone3DAbs
- Composite, pour les objets qui ont des enfants
  - Icone3DComposite
- Feuilles, pour les objets qui n'ont pas d'enfants
  - sphère, cylindre, cube, etc.



# 2 – Accéder aux enfants du composite sans briser l'encapsulation

Comment donner accès aux enfants du composite sans révéler la structure de données utilisée pour les stocker ?

- Permettre à la classe composite de « changer d'idée »,
- Éviter de changer l'interface de la classe en cas de modification de la représentation des enfants,
- Éviter les changements dans les clients.

# Patron Iterator

## Intention

Fournir une méthode d'accès séquentielle aux éléments d'un objet agrégat (liste, vecteur, …) sans exposer sa structure interne.

## Applicabilité

Pour accéder au contenu d'un objet agrégat sans révéler sa structure interne.

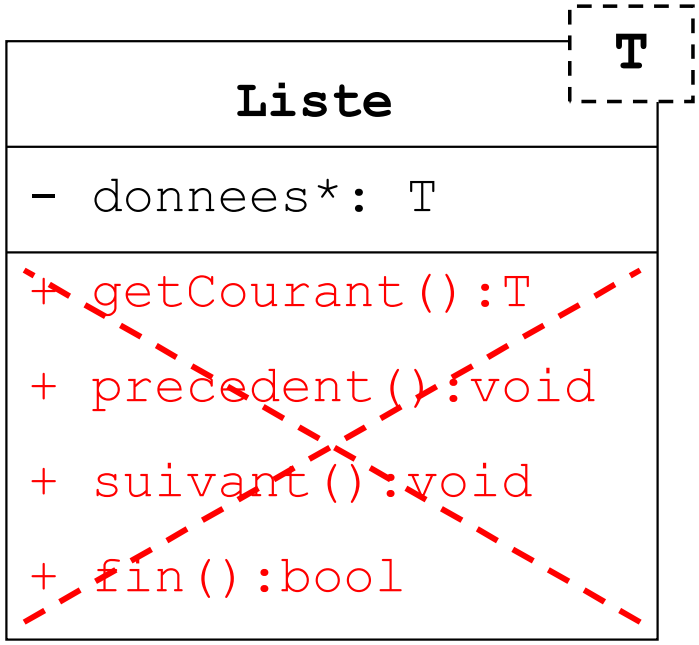
Pour supporter les traversées multiples et simultanées d'un objet agrégat.

Pour fournir une interface uniforme de traversée pour différents types de structures agrégat.

# Patron Iterator

## Motivation

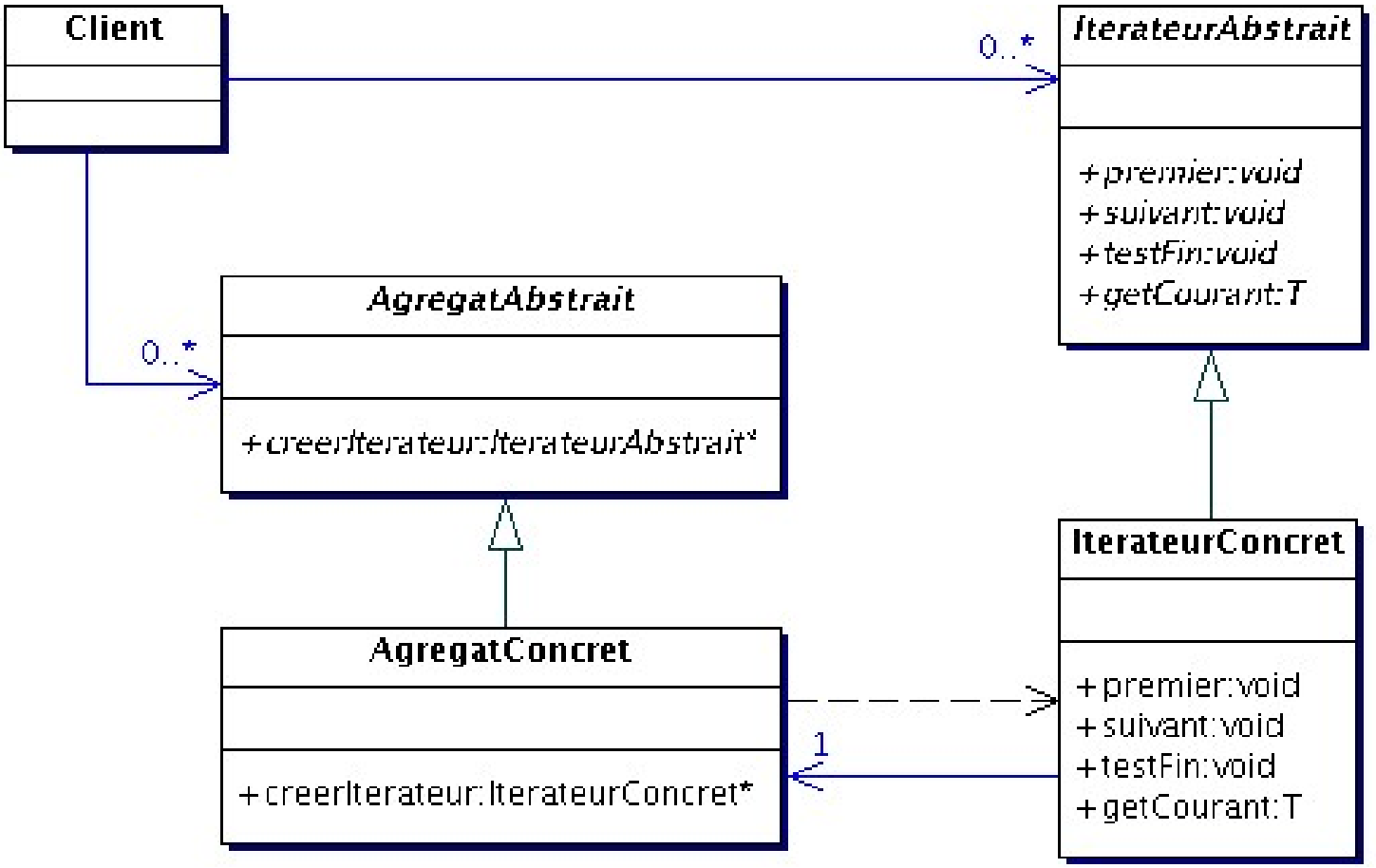
On veut éviter de polluer les objets agrégats avec des fonctions de parcours des éléments. On voudrait éviter d'avoir ceci:




- Pollution de l'interface de l'objet agrégat
- Impossibilité de maintenir plusieurs traversées simultanées
- Impossible d'écrire des algorithmes génériques valables pour tout type d'agrégat

# Patron Iterator

## Structure



# Patron Iterator

## Conséquences

+ Variabilité : L'itérateur permet de supporter plusieurs variations dans le mode de traversée de l'agrégat.
+ Simplification : En définissant un itérateur, on peut simplifier l'interface de l'objet agrégat.
+ Traversées multiples : On peut faire **plusieurs traversées simultanées** sur un même objet agrégat.
+ Réutilisation : On peut écrire des algorithmes génériques valables pour tous les types d'agrégats en passant par les itérateurs (STL).

# Patron **Iterator :** Icone3DContainer

```
// Configuration du stockage des enfants des Icones composites
using Icone3DPtr = std::unique_ptr<class Icone3DAbs>;
using Icone3DContainer = std::vector<Icone3DPtr>;
using Icone3DBaseIterator = Icone3DContainer::iterator;
using Icone3DBaseIterator_const = Icone3DContainer::const_iterator;

class Icone3DIterator : public Icone3DBaseIterator
{
public:
        Icone3DIterator(const Icone3DContainer::iterator& it) : Icone3DBaseIterator(it) {}

        // Operateurs simplifiant l'acces a l'Icone 3D sur lequel pointe l'iterateur
        // pour Icone3DIterator it;
        //    *it est l'Icone 3D
        //    it-> permet d'invoquer une methode sur l'Icone 3D
        class Icone3DAbs& operator*() { return *((*(Icone3DBaseIterator(*this))).get()); }
        class Icone3DAbs* operator->() { return (*(Icone3DBaseIterator(*this))).get(); }
};

class Icone3DIterator_const : public Icone3DBaseIterator_const
{
public:
        Icone3DIterator_const(const Icone3DContainer::const_iterator& it) : Icone3DBaseIterator_const(it) {}

        // Operateurs simplifiant l'acces a l'Icone 3D sur lequel pointe l'iterateur
        // pour Icone3DIterator_const it;
        //    *it est l'Icone 3D constant
        //    it-> permet d'invoquer une methode const sur l'Icone 3D
        const class Icone3DAbs& operator*() { return *((*(Icone3DBaseIterator_const(*this))).get()); }
        const class Icone3DAbs* operator->() { return (*(Icone3DBaseIterator_const(*this))).get(); }
};
```

# 3 – Sélection d'une primitive

- Problème de conception:
  - Comment représenter la sélection d'une ou de plusieurs primitives,
  - Doit permettre d'accéder aux données des primitives et d'appliquer des opérations aux primitives sélectionnées.

# Sélection d'une primitive

- Identifier le bon patron de conception:
  - Considérer de quelle façon les patrons de conception résolvent les problèmes de conception
    - C'est-à-dire lire le manuel: pas le temps !
  - Parcourir les sections Intention (Intent) de chaque patron
    - La force brute
  - Étudier comment les patrons sont inter-reliés (diagramme spaghetti, etc.)
    - Encore trop long, mais on se rapproche…

# Sélection d'une primitive

- Identifier le bon patron de conception:
  - Considérer les patrons visant le bon but (créationel, structural ou comportemental)
    - La représentation d'une sélection suggère un but structurel
  - Examiner les causes de reconception (p. 24)
    - On n'est pas encore rendu là, on veut juste concevoir
  - Considérer ce que l'on veut rendre variable dans notre conception (tableau 1.2, p. 30)

# Sélection d'une primitive

| **Patrons de conception structural** | **Variabilité fournie par le patron** |
|---|---|
| Adapter | Interface de l'objet |
| Bridge | Implantation de l'objet |
| Composite | Structure et composition de l'objet |
| Decorator | Responsabilités sans sous-classer |
| Facade | Interface à un sous-système |
| Flyweight | Coût de stockage des objets |
| Proxy | Mode d'accès à un objet ou sa localisation |

# Patron **Proxy**

- Intention

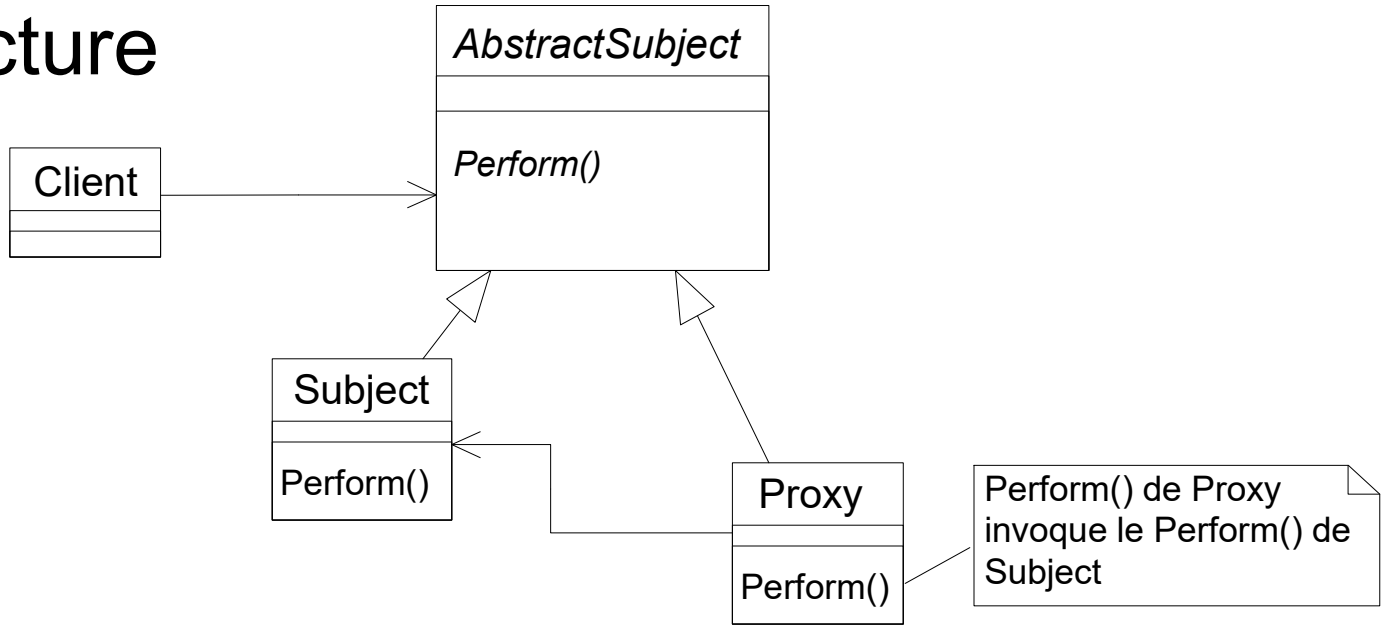
  Fournir un remplaçant ou une doublure pour un autre objet afin de contrôler l'accès à ce dernier.

- Applicabilité

  Ce patron est applicable dès que le besoin d'une référence plus versatile ou plus sophistiquée qu'un simple pointeur se fait sentir.

  L'interface du Proxy doit correspondre à l'interface du sujet.

- Structure



# Patron **Proxy**

- Conséquences
  + Niveau supplémentaire d'indirection. Différents types de proxy (différents usages):
    - Proxy distant: permet de cacher qu'un objet réside dans un autre espace d'adresses.
    - Proxy virtuel: permet des optimisations telles que créer un objet coûteux sur demande.
    - Proxy de protection: permet de contrôler l'accès à l'objet original et modifier les droits d'accès.
    - Références intelligentes: sert de remplacement à un pointeur et exécute des opérations supplémentaires lorsque l'objet est accédé (compter le nombre de références, charger dynamique l'objet en mémoire, verrouiller l'objet pour permettre un accès unique, etc.).

# Patron **Proxy** : Sélection de primitives

- Établir la correspondance entre les participants au patron **Proxy** et les classes de primitives:
  - **AbstractSubject**, l'interface à laquelle il faut correspondre
    - Il s'agit de **`Icone3DAbs`**
  - **Proxy**, la classe servant de remplaçant
    - **`SelectedIcone`**, la sélection
  - **Subject**, l'objet auquel le **`Proxy`** réfère
    - L'**`Icone3DComposite`**, la **`PrimitiveAbs`**, la **`Sphere`**, le **`Cylindre`** ou le **`Cube`** ?
- Problème: on ne veut pas lier **`Subject`** à aucune de ces classes.

# Patron **Proxy** : Sélection de primitives

- Solution: lire la description du participant Proxy dans le patron

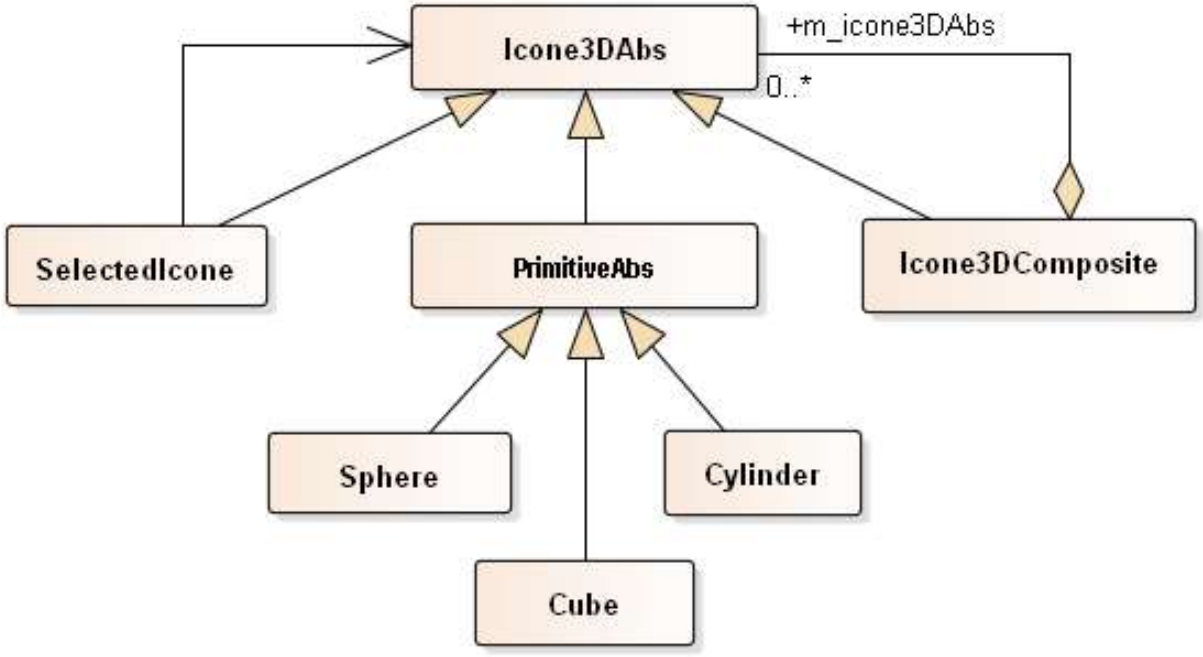
  Le **`Proxy`** maintient une référence qui laisse le **`Proxy`** accéder au Subject. Le **`Proxy`** peut référencer un **`AbstractSubject`** si l'interface du **`Subject`** et de l'**`AbstractSubject`** sont compatibles.

  ➢Le **`Subject`** sera donc l'objet **`Icone3DAbs`**

  ➢Ce choix **ne serait pas possible sans l'interface uniforme** choisie pour le patron **Composite**

# Patron **Proxy** : Sélection de primitives

- L'icône sélectionnée délègue toutes les opérations à son sujet.
- L'icône sélectionnée peut ajouter une fonctionnalité supplémentaire permettant à certains clients (qui savent qu'ils sont en train de manipuler un lien) d'accéder directement au sujet.



# Patron **Proxy** : SelectedIcone

```
class SelectedIcone : public Icone3DAbs
{
public:
          SelectedIcone();
          SelectedIcone(Icone3DAbs& parent);
          SelectedIcone(Icone3DAbs& parent, const SelectedIcone& mdd);
          virtual SelectedIcone* clone(Icone3DAbs& parent) const;
          virtual ~SelectedIcone();

          virtual void addChild(const Icone3DAbs& obj3d);
          virtual Icone3DIterator begin();
          virtual Icone3DIterator_const cbegin() const;
          virtual Icone3DIterator_const cend() const;
          virtual Objet3DAbs* clone() const;
          virtual Icone3DIterator end();
          virtual void removeChild(Icone3DIterator_const obj3dIt);
          virtual void replaceChild(Icone3DIterator obj3dIt, const Icone3DAbs& newObj);

          virtual Point3D getCenter() const;
          virtual void moveCenter(const Point3D& delta);
          virtual void setCenter(const Point3D& center);

          virtual size_t getNbParameters() const;
          virtual PrimitiveParams getParameters() const;
          virtual void setParameter(size_t pIndex, float pValue);

          virtual Icone3DAbs& getSubject();
          virtual const Icone3DAbs& getSubject() const;

protected:
          Icone3DAbs& m_sujet;
};
```