POLYTECHNIQUE MONTRÉAL

CHAPITRE 16

Patrons de conception : Décorateur et Visiteur

POLYTECHNIQUE MONTRÉAL

Sommaire

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

4 – Transformer les primitives

- Problème de conception:
  - Différentes transformations peuvent être appliquées aux primitives (translation, rotation, mise à l'échelle, etc.),
  - Ces transformations ne font pas directement partie des primitives, mais leurs sont étroitement associées,
  - Ces données peuvent être ajoutées et modifiées dynamiquement

    → Sous-classer les primitives n'est pas une option

Transformer les primitives

Patrons de conception structuraux	**Variabilité fournie par le patron**
Adapteur	Interface de l'objet
Bridge	Implantation de l'objet
Composite	Structure et composition de l'objet
Décorateur	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 **Décorateur**

- Intention

  Attacher dynamiquement des responsabilités additionnelles à un objet. Le patron Décorateur fournit une alternative flexible à la dérivation pour étendre la fonctionnalité d'une classe.

- Applicabilité

  Ce patron est utilisé pour ajouter des responsabilités à des objets individuels de façon dynamique et transparente, sans affecter d'autres objets.

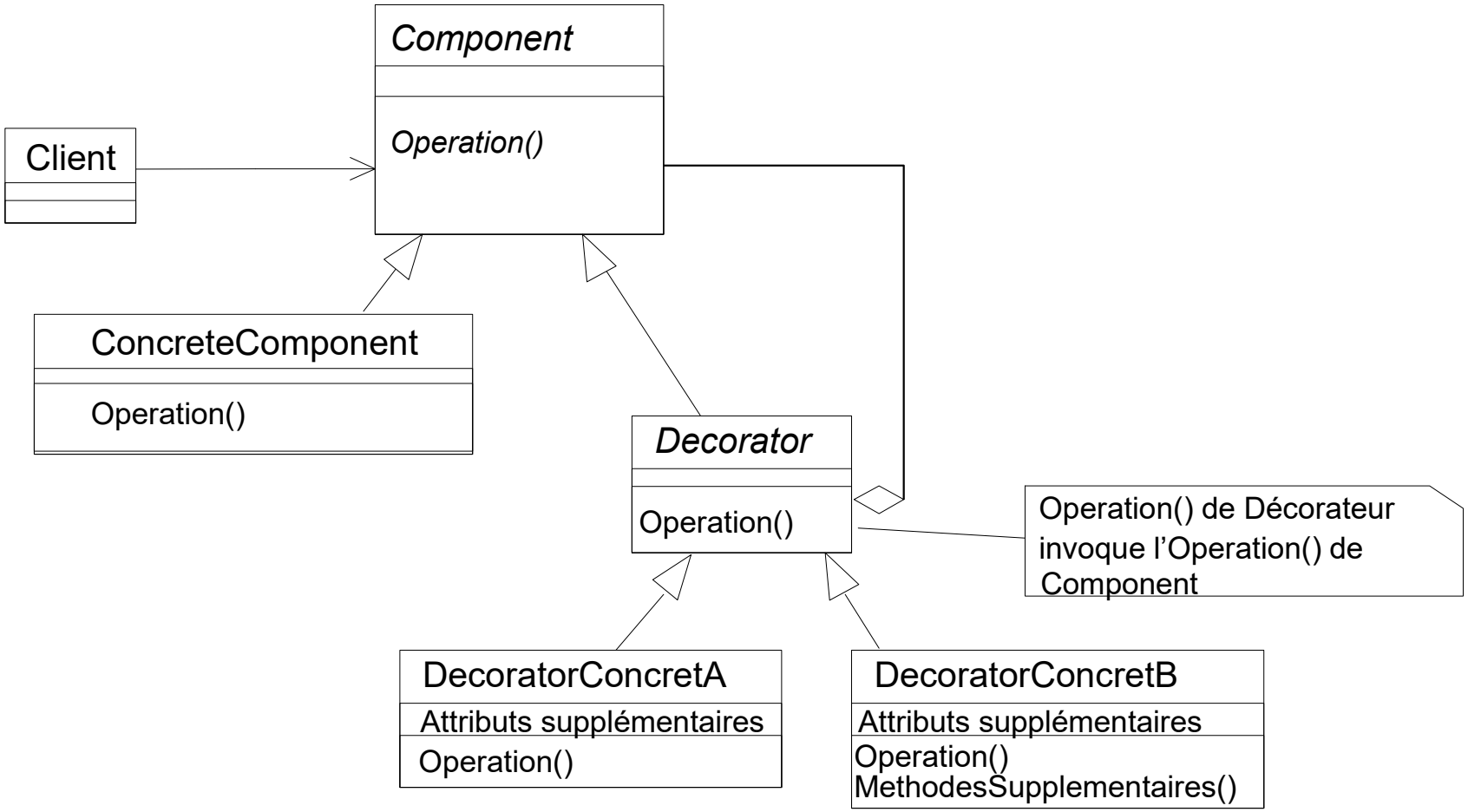
  Permet d'ajouter des responsabilités qui peuvent ensuite être retirées.

  On utilise ce patron lorsqu'il n'est pas pratique de sous-classer. Parfois, un grand nombre d'extensions indépendantes sont possibles, et produiraient une explosion dans le nombre de sous-classes à supporter pour obtenir toutes les combinaisons. Parfois, la définition de la classe à étendre peut être inaccessible, ce qui empêche de sous-classer.

POLYTECHNIQUE MONTRÉAL

Patron **Décorateur**

- Structure



Patron **Décorateur**

- Conséquences
  - + Flexibilité: Plus flexible que l'héritage statique. Permet d'ajouter et de retirer des responsabilités en cours d'exécution en attachant ou détachant des décorateurs. Permet de choisir les responsabilités « à la carte ».
  - + Évite les classes racines complexes: Plutôt que d'essayer de supporter toutes les fonctionnalités dans la classe de base, les décorateurs permettent de définir des responsabilités ciblées qui peuvent être ajoutées au besoin.
  - - Identité des objets: L'ajout d'un décorateur à un objet modifie l'identité de l'objet. L'objet original et l'objet décoré sont deux objets différents, ce qui rend l'accès à l'objet original moins direct.
  - - Coût: peut nécessiter un grand nombre d'objets.

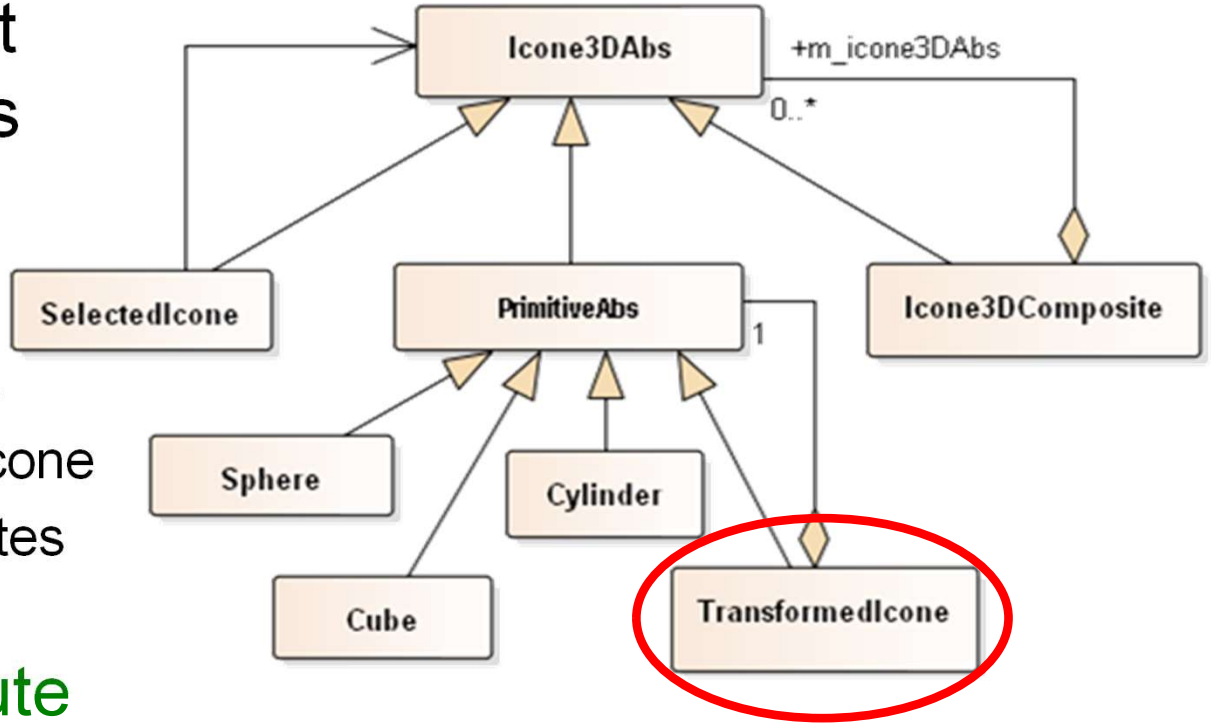
Patron **Décorateur**

- Implantation
  - Uniformité et conformité de l'interface
  - Classe abstraite de Décorateur facultative
  - Garder les classes de Component légères
  - Changer l'enrobage d'un objet plutôt que ses entrailles

POLYTECHNIQUE MONTRÉAL

Patron **Décorateur** : transformer les primitives

- La transformation peut invoquer les méthodes de la primitive.
- Rôles:
  - Component: PrimitiveAbs
  - Decorator: TransformedIcone
  - ConcreteComponent: toutes les classes de primitives
- La transformation ajoute une responsabilité supplémentaire.



Quelle est la différence entre le patron **Proxy** et la patron **Décorateur ?**

La différence fondamentale est l'**intention**:

- Intention de Proxy: Fournir un remplaçant ou une doublure pour un autre objet afin de contrôler l'accès à ce dernier.
- Intention de Décorateur: Attacher dynamiquement des responsabilités additionnelles à un objet.

La structure peut être très semblable mais l'objectif de conception est entièrement différent.

POLYTECHNIQUE MONTRÉAL

Patron **Décorateur** : TransformedIcone

```
class TransformedIcone : public PrimitiveAbs
{
private:
        float m_scale;
        Point3D m_translation;
        std::unique_ptr<PrimitiveAbs> m_Icone;
public:
        TransformedIcone(const PrimitiveAbs& primitiv, const Point3D& translat, float scal);
        TransformedIcone(IconeAbs& i, const PrimitiveAbs& p, const Point3D& t, float s);
        TransformedIcone(IconeAbs& parent, const TransformedIcone& mdd);
        virtual ~TransformedIcone();
        virtual TransformedIcone* clone(IconeAbs& parent) const;

        virtual float getScale() const;
        virtual void setScale(float scal);
        virtual Point3D getTranslation() const;
        virtual void setTranslation(const Point3D& translat);
        virtual PrimitiveAbs& getPrimitive();
        virtual const PrimitiveAbs& getPrimitive() const;

        virtual void addChild(const IconeAbs& obj3d);
        virtual IconeIterator begin();
        virtual IconeIterator_const cbegin() const;
        virtual IconeIterator_const cend() const;
        virtual IconeIterator end();
        virtual void removeChild(IconeIterator_const obj3dIt);

        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);
};
```

POLYTECHNIQUE MONTRÉAL

Patron **Décorateur** : TransformedIcone

Les méthodes définies dans l'interface commune peuvent être ajustées selon les nouvelles responsabilités du Décorateur ou déléguées directement au component.

```
Point3D TransformedIcone::getCenter() const
{
        return m_Icone->getCenter() + m_translation; // Le centre de la primitive est déplacé
}


size_t TransformedIcone::getNbParameters() const
{
        return m_Icone->getNbParameters(); // délégation directe à la primitive
}

PrimitiveParams TransformedIcone::getParameters() const
{
        PrimitiveParams params = m_Icone->getParameters();
        for (int iparam = 0; iparam < params.size(); ++iparam)
                params[iparam] *= m_scale; // Les paramètres de la primitive sont mis à l'échelle
        return params;
}
```

5 – Fonctionnalité ouverte

- Problème:
  - Les clients veulent appliquer un nombre arbitrairement grand d'opérations sur les icônes
    - Ex: calculer le volume, sauvegarder dans un fichier, envoyer à un autre programme, trouver les primitives d'un certain type, etc.
  - Il faut éviter de traiter l'interface de la classe Icone3DAbs comme un fourre-tout qui ramasse toutes les opérations, ce qui résulterait en une classe à faible cohésion.

Fonctionnalité ouverte

- On ne veut pas obtenir des classes comme:

```
class Icone3DAbs {
public:
    virtual void faireCalculVolume();
    virtual void faireSauvergarder();
    virtual void faireEnvoyer();
    virtual void faireTrouver();
};

class Sphere {
public:
    virtual void faireCalculVolume();
    virtual void faireSauvegarder();
    virtual void faireEnvoyer();
    virtual void faireTrouver();
};
```

Fonctionnalité ouverte

- Cette approche a des désavantages significatifs:
  - Dispersion du code: pour trouver le code qui implante le calcul du volume, il faut parcourir toutes les classes
    - On ne peut pas voir et comprendre le code de la fonction en regardant une seule composante,
    - Ça rend le code plus difficile à maintenir, étendre, documenter, etc.
  - L'ajout d'une nouvelle fonctionnalité est difficile
    - Il faut modifier toute la hiérarchie
    - Tous les clients doivent être recompilés même s'ils n'utilisent pas la nouvelle fonctionnalité

Fonctionnalité ouverte

- Il faut:
  - Pouvoir grouper tout le code associé à une fonctionnalité particulière à un seul endroit,
  - Ne pas devoir modifier la hiérarchie de classes des icônes chaque fois qu'une nouvelle fonctionnalité est ajoutée,
  - Pouvoir appliquer ces opérations de l'extérieur.

Fonctionnalité ouverte

- Note: toutes les fonctionnalités dont on parle ici exigent de parcourir un certain nombres de primitives (voire toutes les primitives) de l'icône.
- Si la classe Icone3DAbs n'a pas les fonctions virtuelles appropriées, les clients peuvent être tentés d'utiliser une exécution conditionnelle basée sur RTTI:

POLYTECHNIQUE MONTRÉAL

Fonctionnalité ouverte : RTTI

```
for ( chaque Icone3DAbs i de la structure )
{
   Sphere* s = dynamic_cast<Sphere*>( &i );
   if ( s != NULL )
      // faire qq chose avec la sphere
   else
   {
      Cube* c = dynamic_cast<Cube*>( &i );
      if ( c != NULL )
         // faire qq chose avec le cube
      else
      {
          ...;
      }
   }
}
```

➔ Beurk !

Fonctionnalité ouverte

Lorsque l'on traverse les primitives de l'icône, on doit visiter chaque primitive de la structure (ou une partie des primitives).

- Le nom **Visiteur** est motivé par le besoin commun de traverser des structures de données:
  - Une action est accomplie sur chaque élément de la structure de données
- Le patron Visiteur décrit comment construire une hiérarchie qui facilite ce type de traversée
  - Permet d'ajouter élégamment les nouvelles actions,
  - Évite de recourir à des structures if-then-else basées sur RTTI,
  - Permet d'ajouter une nouvelle fonction qui se comporte comme une fonction virtuelle sans changer la hiérarchie de classes.

Patron **Visiteur**

- Intention

  Représenter une opération qui doit être appliquée sur les éléments d'une structure d'objets. Un Visiteur permet de définir une nouvelle opération sans modification aux classes des objets sur lesquels l'opération va agir.

- Applicabilité

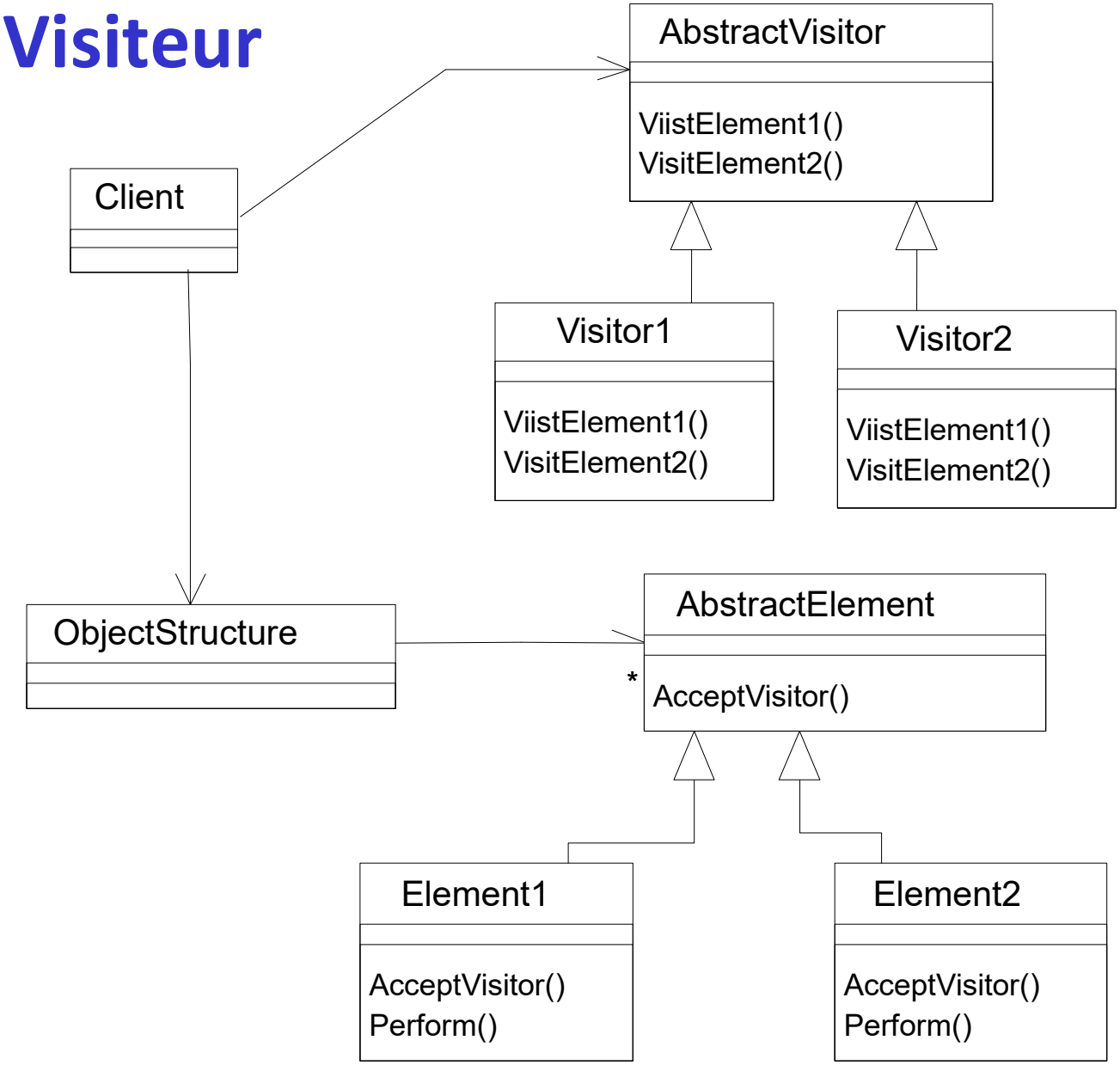
  Lorsqu'une structure d'objets contient plusieurs classes avec des interfaces différentes.

  Lorsque plusieurs opérations distinctes et sans liens entre elles doivent agir sur des objets conservés dans une structure.

  Lorsque les classes définissant la structure des objets changent rarement, mais que l'on veut fréquemment définir de nouvelles opérations sur cette structure.

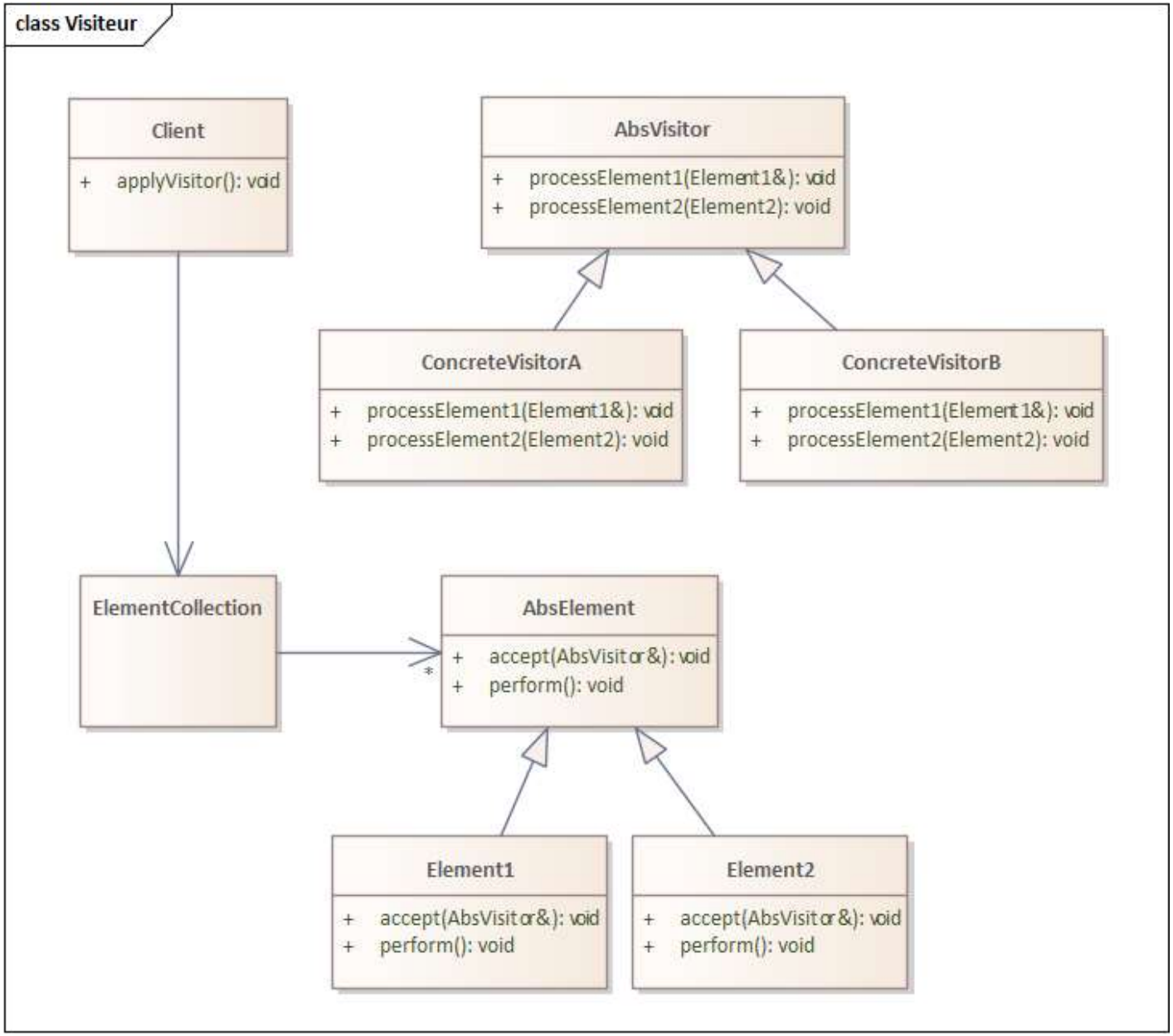
POLYTECHNIQUE MONTRÉAL

Patron **Visiteur**

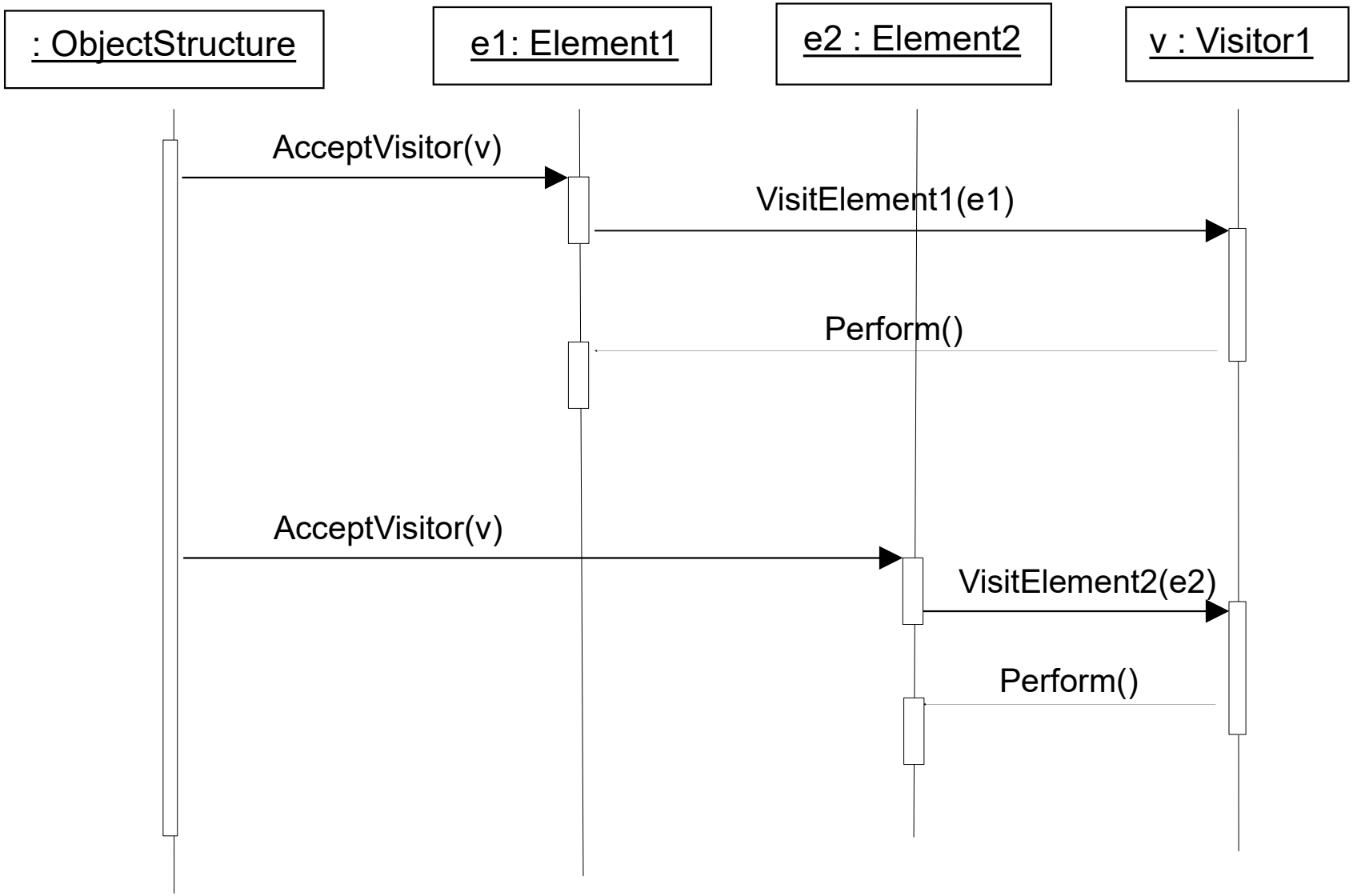


POLYTECHNIQUE MONTRÉAL

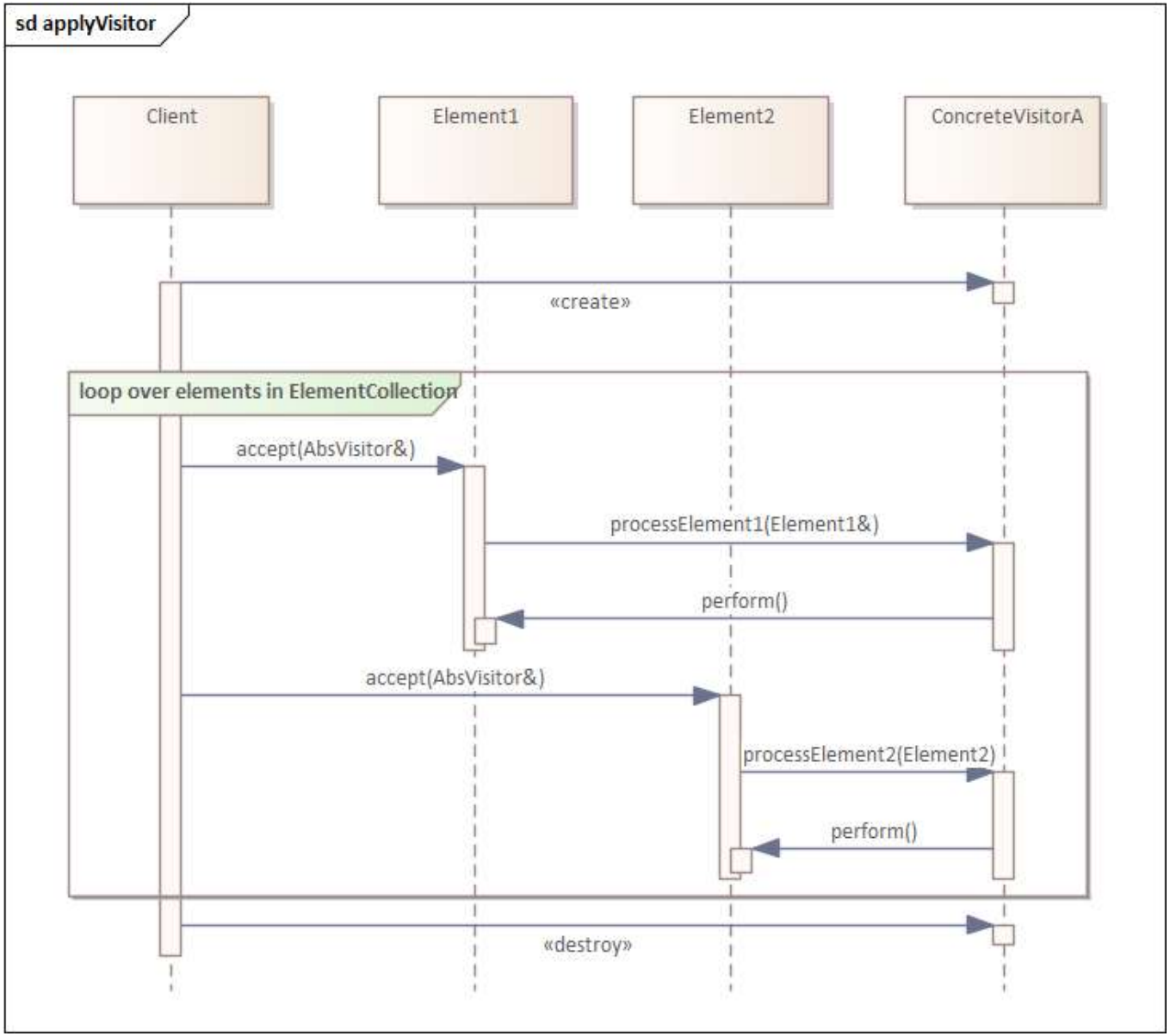
Patron **Visiteur**



Patron **Visiteur** – Collaborations



Patron **Visiteur** – Collaborations



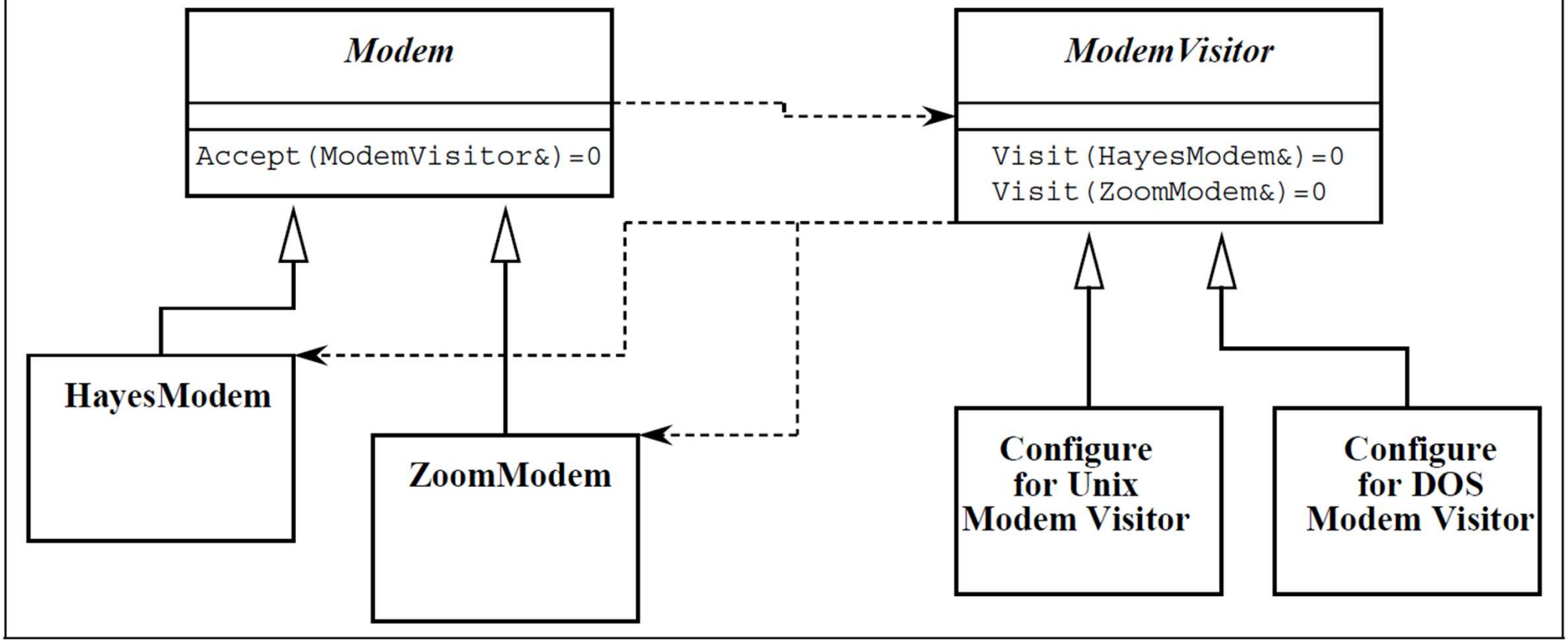
Patron **Visiteur**

- Conséquences
  + Flexibilité: les Visiteurs et la structure d'objets sont indépendants
  + Fonctionnalité localisée: tout le code associé à une fonctionnalité se retrouve à un seul endroit bien identifié.
  - Coût de communication supplémentaire entre les Visiteurs et la structure d'objets.
- Implantation
  – double invocation (double dispatch),
  – interface générale aux éléments de la structure d'objets.

POLYTECHNIQUE MONTRÉAL

Exemple: configuration de modems

Figure 2: Modem Configuration Visitors



Source: Robert C. Martin, *Acyclic Visitor (v1.0)*, https://condor.depaul.edu/dmumaugh/OOT/Design-Principles/acv.pdf

Patron **Visiteur :** Fonctionnalité ouverte

Comment faire pour ajouter de nouvelles fonctionnalité à nos icônes par des visiteurs ?

- Définir une classe **IconeVisitor** qui servira de base à tous les visiteurs qui seront ajoutés dans le futur,
- Ajouter une opération **accept()** dans la classe Icone3DAbs et ses sous-classes qui permet d'accueillir les visiteurs dérivant de la classe VisiteurIcone,
- Ajouter de nouvelles fonctionnalités se résume maintenant à définir de nouvelles sous-classes de **IconeVisitor**.

Patron **Visiteur** : Fonctionnalité ouverte

Définition de la classe IconeVisitor (1):

```
class IconeVisitor
{
public:
   virtual ~IconeVisitor() = 0;

   virtual void visitSphere( Sphere& s ) = 0;
   virtual void visitCube( Cube& c ) = 0;
   virtual void visitCylindre( Cylindre& c ) = 0;

   virtual void visitIconeComposite( Icone3DComposite& i ) = 0;
   virtual void visitSelectedIcone( SelectedIcone& i ) = 0;
   virtual void visitTransformedIcone( TransformedIcone& i ) = 0;
};
```

(1) Version utilisable dans la majorité des langages

Patron **Visiteur** : Fonctionnalité ouverte

Définition de la classe IconeVisitor (2):

```
class IconeVisitor
{
Public:
   virtual ~IconeVisitor() = 0; // classe abstraite

   virtual void visitSphere( Sphere& s ) { defaultVisitPrimitive(s);};
   virtual void visitCube( Cube& c ) { defaultVisitPrimitive(c);};
   virtual void visitCylindre( Cylindre& c ) { defaultVisitPrimitive(c);};
   virtual void visitIconeComposite( Icone3DComposite& i ) {
     defaultVisitIconeComposite(i);};
   virtual void visitSelectedIcone( SelectedIcone& i ) {
     defaultVisitSelectedIcone(i);};
   virtual void visitTransformedIcone( TransformedIcone& i ) {
     defaultVisitTransformedIcone(i);};

protected:
   defaultVisitPrimitive( PrimitiveAbs& p);
   defaultVisitIconeComposite( Icone3DComposite& c );
   defaultVisitSelectedIcone( SelectedIcone& s );
   defaultVisitTransformedIcone( TransformedIcone& t );
};
```

(2) Version utile en C++ pour faciliter le développement

Patron **Visiteur** : implémentations par défaut

```
IconeVisitor::defaultVisitPrimitive( PrimitiveAbs& p)
{
   std::cerr << "Visite une primitive ayant "
             << p.getNbParameters() << " parametre(s)" << endl;
}

IconeVisitor::defaultVisitIconeComposite( Icone3DComposite& c )
{
   for (auto it = c.begin(); it != c.end(); ++it)
      it->accept(*this);
}

IconeVisitor::defaultVisitSelectedIcone( SelectedIcone& s )
{
       s.getSubject().accept(*this);
}

IconeVisitor::defaultVisitTransformedIcone( TransformedIcone& t )
{
       s.getPrimitive().accept(*this);
}
```

POLYTECHNIQUE MONTRÉAL

Patron **Visiteur** : Fonctionnalité ouverte

```
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 void accept( IconeVisitor& v) = 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 **Visiteur** : Fonctionnalité ouverte

Toutes les méthodes **accept()** pour les sous-classes d'Icone3DAbs sont implantées de la même façon:

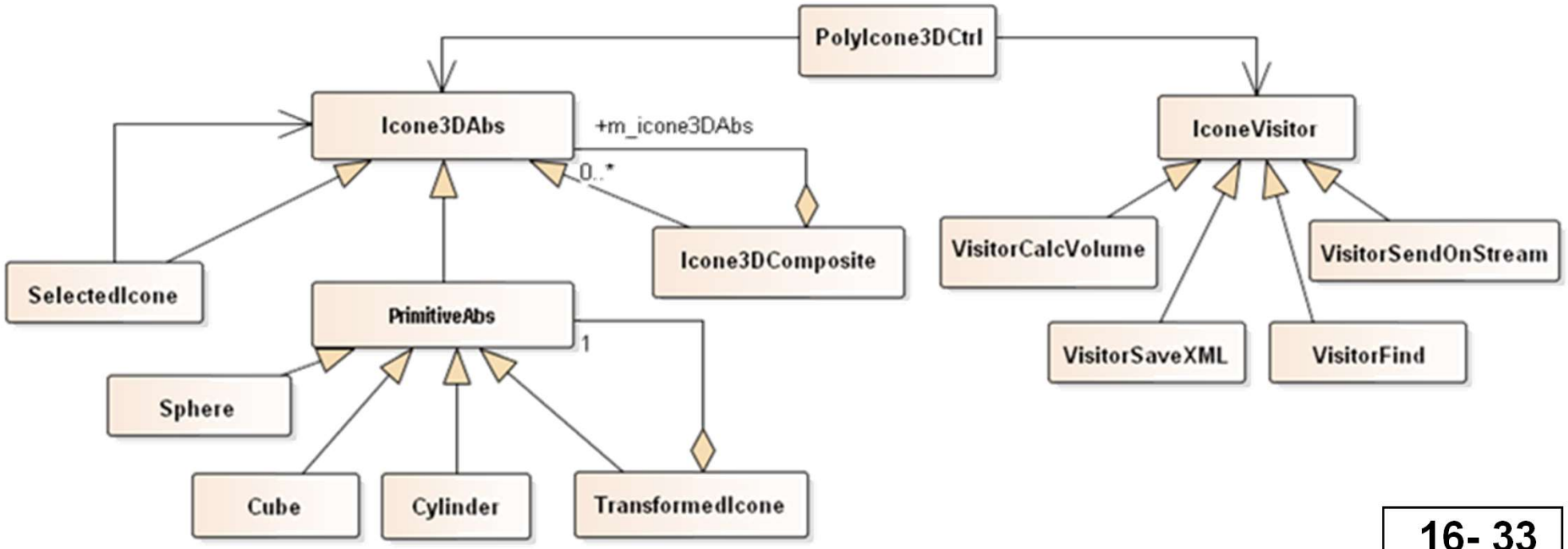
```
Sphere::accept( IconeVisitor& v )
        { v.visitSphere(*this); }
Cube::accept( IconeVisitor& v )
        { v.visitCube(*this); }
Cylindre::accept( IconeVisitor& v )
        { v.visitCylindre(*this); }
IconeComposite::accept( IconeVisitor& v )
        { v.visitIcone3DComposite(*this); }
SelectedIcone::accept( IconeVisitor& v )
        { v.visitSelectedIcone (*this); }
TransformedIcone::accept( IconeVisitor& v )
        { v.visitTransformedIcone(*this); }
```

POLYTECHNIQUE MONTRÉAL

Patron **Visiteur** : Fonctionnalité ouverte

Rôles:

- L'élément abstrait: Icone3DAbs
- Les éléments concrets: toutes les classes concrètes dérivées de Icone3DAbs.
- Le visiteur abstrait: IconeVisitor.
- Les visiteurs concrets: toutes les classes dérivées de IconeVisitor



Patron **Visiteur :** Fonctionnalité ouverte

- On définit une sous-classe de **IconeVisitor** pour chaque fonctionnalité qu'on veut ajouter.
- Dans cette sous-classe, on implante chacune des fonctions de traitement pour accomplir la tâche appropriée pour chaque type d'objet.
- Par exemple, la classe **VisitorCalcVolume** implante la fonctionnalité de calcul du volume.

Patron **Visiteur** : VisitorCalcVolume

```
class VisitorCalcVolume : public IconeVisitor
{
public:
   VisitorCalcVolume(void) : m_accumulatedVolume(0.0) {}

   virtual void visitSphere( Sphere& s );
   virtual void visitCube( Cube& c );
   virtual void visitCylindre( Cylindre& c );
   virtual void visitIconeComposite( Icone3DComposite& i );
   virtual void visitSelectedIcone( SelectedIcone& i );
   virtual void visitTransformedIcone( TransformedIcone& i );

   float getVolume(void) const { return m_accumulatedVolume; }
   void setVolume( float vInit ) {accumulatedVolume = vInit; }

protected:
   float m_accumulatedVolume;
}
```

Patron **Visiteur** : VisitorCalcVolume

```
#include <boost/math/constants.hpp>

VisitorCalcVolume::visitSphere( Sphere& s)
{
   float pi = boost::math::constants::pi<float>();
   PrimitiveParameters params = s.getParameters();
   float r3 = params[0]*params[0]*params[0];
   m_accumulatedVolume += 4.0*pi*r3/3.0;
}

VisitorCalcVolume::visitCube( Sphere& c)
{
   PrimitiveParameters params = s.getParameters();
   m_accumulatedVolume += params[0]*params[1]*params[2];
}

void VisitorCalcVolume::visitIconeComposite(Icone3DComposite& c)
{
       defaultVisitObjComposite(c);
}
```

Patron **Visiteur :** Fonctionnalité ouverte

Pour utiliser la fonctionnalité:

- Les clients créent une instance du visiteur et passent cette instance à chacune des composantes de l'icône.
- Dans le cas d'un objet composite, il suffit de passer le visiteur à l'objet racine.

```
Icone3DComposite uneIcone;
uneIcone.addChild(…);
uneIcone.addChild(…);
[...]
VisitorCalcVolume vis;
uneIcone.accept( vis );

float volume_total = vis.getVolume();
```

Patron **Visiteur** : mécanisme des appels

- Que se passe-t-il alors ?
  - Supposons que le premier enfant du composite est une Sphere. Alors, `it->accept(vis)` invoque la méthode `Sphere::accept(vis)` (par polymorphisme).
  - Cette méthode invoque à son tour la méthode `VisitorCalcVolume::visitSphere(Sphere& s)` en passant comme sphère (this) en paramètre.
  - L'instance VisitorCalcVolume est donc maintenant en mesure de traiter la primitive en question.

Note: Le raisonnement est analogue pour les toutes les classes dérivées de Icone3DAbs.

Patron **Visiteur** : évaluation

- Avantages:
  - –La dispersion du code est évitée:
    - Tout le code pour faire calculer le volume se trouve dans la classe VisitorCalcVolume.
  - –Ajouter de nouvelles fonctionnalités ne requiert aucun changement à la hiérarchie de classes des icônes
    - On ajoute plutôt des nouvelles sous-classes de la classe IconeVisitor,
    - Les fonctions membres de ces sous-classes **agissent comme des fonctions virtuelles des classes des icônes**.

Patron **Visiteur** : évaluation

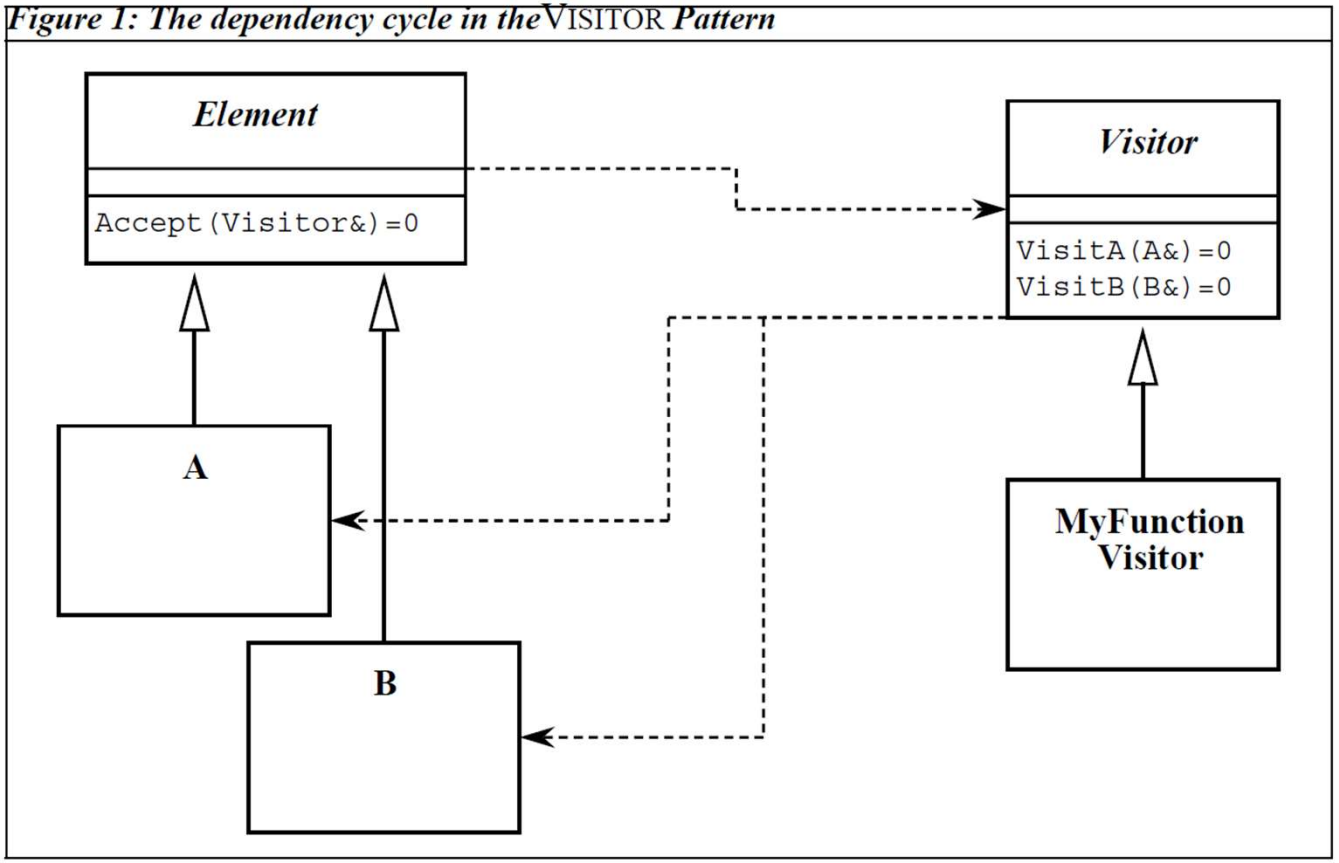
- Désavantages:
  - Difficile de faire des changements à la hiérarchie des éléments
    - **Toutes** les classes de Visiteur doivent être modifiées. Donc tous les clients de l'une ou l'autre de ces classes doivent être recompilés.
    - **Toutes** les implantations des classes Éléments doivent être recompilées,
    - **Tous** les clients des classes Éléments affectés par le changement doivent être recompilés,
  - Tous les types de Visiteur doivent avoir un sens pour tous les types d'Éléments:
    - Comment construire un Visiteur qui n'agisse que sur les icônes sélectionnées ?
  - Le code des Visiteur peut être dupliqué
    - Si plusieurs types de nœuds sont traités de la même façon, il faut quand même écrire une fonction pour chaque type.

Patron **Visiteur** : évaluation

- On peut réduire la duplication de code en utilisant les méthodes de traitement acceptant un pointeur ou une référence à la classe du haut de la hiérarchie d'Éléments:
  - Cette méthode pourrait être déclarée dans la classe de base des Visiteurs:
    - Le code indépendant du type d'élément peut aller là,
    - Pour les icônes, on pourrait ajouter une méthode IconeVisitor::visitIcone3DAbs
  - Les sous-classes dans la hiérarchie des Visiteurs peuvent redéfinir la méthode au besoin.

Patron **Visiteur** : une autre approche

- Dans le Visiteur classique, la classe de base du Visiteur dépend de chaque classe concrète dans la hiérarchie des éléments:



- Il faut une fonction de traitement pour chaque type d'Élément concret.
- On obtient un *cycle de dépendances*.

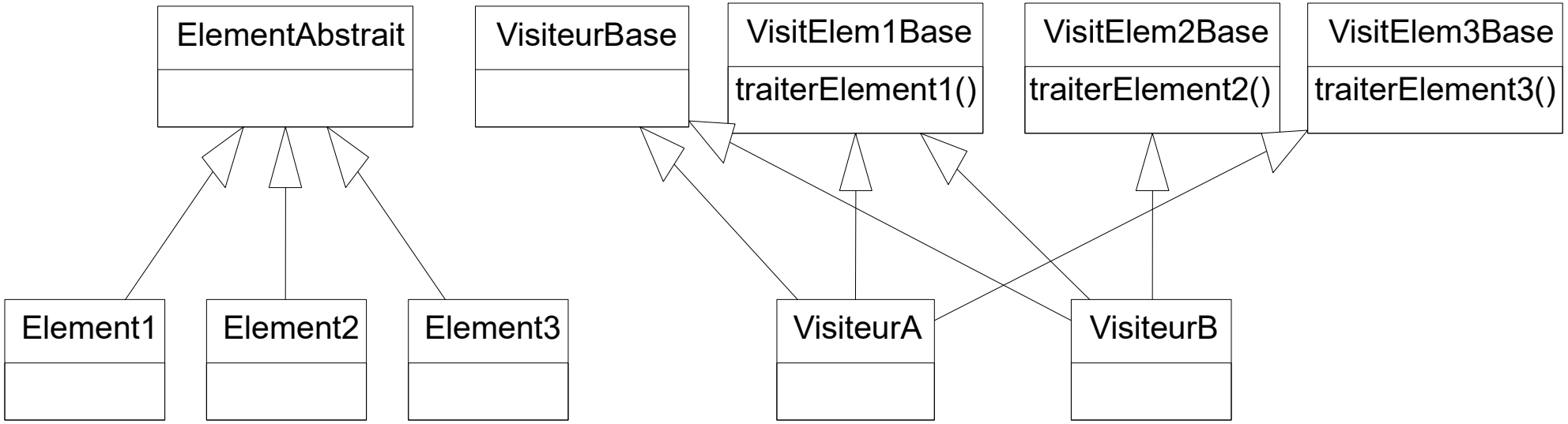
Source: Robert C. Martin, *Acyclic Visitor (v1.0)*, https://condor.depaul.edu/dmumaugh/OOT/Design-Principles/acv.pdf

Patron **Visiteur** : une autre approche

- Le Visiteur Acyclique minimise les dépendances en:
  - Retirant les fonctions de traitement de la classe de base du Visiteur:
    - Pas de fonction de traitement, pas de dépendance
  - Créant une nouvelle classe de base pour chaque type de fonction de traitement:
  - Utilisant l'héritage multiple d'interfaces (mixin) pour ajouter les fonctions de traitement aux classes de Visiteur.

Patron **Visiteur** : Visiteur Acyclique

- Structure générale du patron



Patron **Visiteur** : Visiteur Acyclique

- Structure générale du patron
  - L'ancienne classe de base du Visiteur ne contient plus de fonction de traitement,
  - Chaque autre classe de base contient exactement une fonction de traitement:
    - La fonction de traitement pour la classe d'Élément concret à laquelle la classe de base du Visiteur correspond
    - VisitElem1Base déclare une fonction de traitement pour Element1
    - VisitElem2Base déclare une fonction de traitement pour Element2

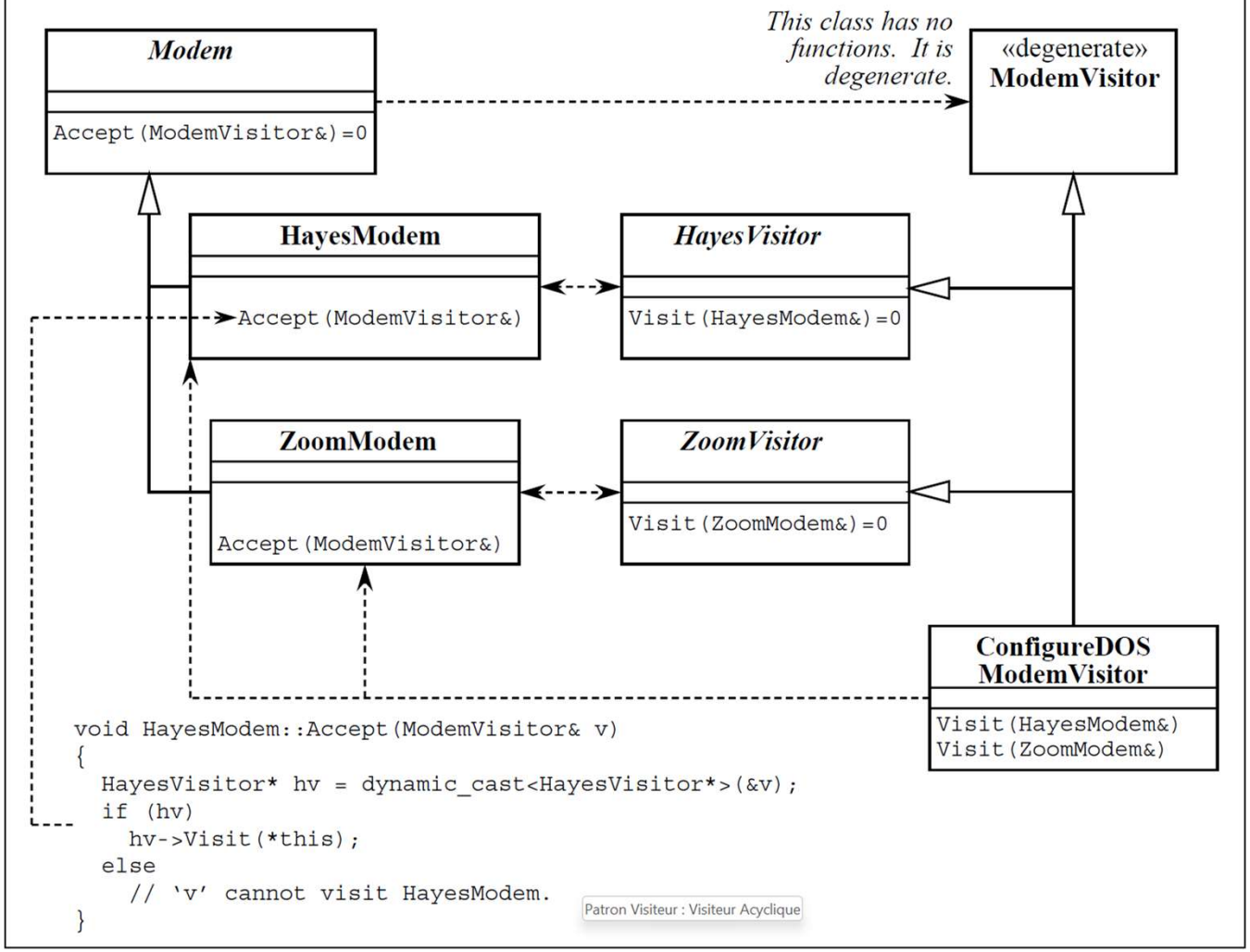
Patron **Visiteur** : Visiteur Acyclique

- Lorsqu'une nouvelle classe d'Élément est ajoutée:
  - Une nouvelle classe de base est ajoutée à la hiérarchie des classes de Visiteur,
  - Un Visiteur concret qui veut traiter le nouveau type d'élément doit dériver de la nouvelle classe de base,
  - Les Visiteurs concrets qui ne sont pas intéressés à traiter le nouvel élément ne sont pas affectés.

Patron **Visiteur** : Visiteur Acyclique

- Retour sur l'exemple de configuration de modems

Figure 3: Acyclic Modem Visitor



Source: Robert C. Martin, *Acyclic Visitor (v1.0)*, https://condor.depaul.edu/dmumaugh/OOT/Design-Principles/acv.pdf

Patron **Visiteur** : Visiteur Acyclique

- Implantation des classes de base des Visiteurs pour les icônes:

```
class IconeVisitor {
public:
    virtual ~IconeVisitor()=0; // rend la classe abstraite
};
VisiteurIcone::~IconeVisitor() {} // pour éviter les arrêts
                                  // brusques…
class VisitorSphere {
public:
    virtual void visitSphere( class Sphere& s )=0;
};

class VisitorCompositeIcone {
public:
    virtual void visitIconeComposite( class Icone3Dcomposite& r )=0;
};

...
```

Fonctionnalité ouverte: le Visiteur Acyclique

- L'implantation des fonctions `accueillir()` dans la hiérarchie des Éléments doit être modifiée afin d'utiliser RTTI pour déterminer si le Visiteur est du bon type:
  - Si c'est le cas, `accueillir` appelle la fonction habituelle de traitement,
  - Sinon, il faut traiter l'erreur
- Par exemple, pour la classe Sphere:

```
void Sphere::accept( IconeVisitor& v )
{
    VisitorSphere* vs = dynamic_cast<VisitorSphere*>(&v);
    if( vs )
       vs->visitSphere(*this);
    else
    {
        // ce type de Visitor ne traite pas les Sphere...
        // il faut faire quelque chose d'autre.
    }
}
```

- On utilise RTTI pour convertir un type latéralement dans la hiérarchie, et non vers le bas de la hiérarchie.

Fonctionnalité ouverte: le Visiteur Acyclique

- Avantages:
  - Les dépendances sont réduites par rapport au Visiteur
    - Lorsqu'un nouveau type d'Élément est ajouté, seules les classes de Visiteur qui sont intéressées par le nouvel Élément doivent être modifiées,
    - Les implantations des Éléments existants n'ont pas besoin d'être recompilées,
    - On peut fournir une bibliothèque précompilée de classes et permettre d'étendre à la fois la hiérarchie des Visiteurs et la hiérarchie des Éléments.
  - Les cas où tous les Visiteurs ne sont pas applicables à tous les éléments sont modélisés naturellement:
    - Les Visiteurs héritent seulement des classes interfaces qui ont un sens pour eux.

Fonctionnalité ouverte: le Visiteur Acyclique

- Désavantages:
  - –Des tests faits à la compilation sont remplacés par des tests faits durant l'exécution:
    - Plus difficile d'éliminer les erreurs de typage,
    - Les erreurs de type doivent être traitées,
    - Plus lent.
  - –Une grande hiérarchie d'Élément mène à une prolifération de classes interfaces de base