POLYTECHNIQUE MONTRÉAL

CHAPITRE 17

Patrons de conception :

Méthode usine, patron de méthode
Stratégie

POLYTECHNIQUE MONTRÉAL

Sommaire

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

6 – Gérer le cycle de vie des composantes d'une icône

- Problème de conception:
  - S'assurer que les éléments du composite sont créés et détruits correctement,
  - Fournir un mécanisme de création des éléments du composite qui facilite l'ajout de nouveaux types d'éléments,
  - S'assurer que les éléments du composite peuvent être détruits sans qu'il reste de références aux objets qui sont détruits.

6 – Gérer le cycle de vie des composantes d'une icône

Pour s'assurer que le composite gère correctement ses enfants, il faut:

- S'assurer que le composite soit l'unique propriétaire des éléments qui le composent,
- Le composite doit donc faire une copie des éléments qui lui sont passés en argument,
- Le composite doit avoir la capacité de créer tous les types d'éléments qu'il gère,
- Éviter que la classe de base du Composite connaisse toute les classes d'éléments.

POLYTECHNIQUE MONTRÉAL

6 – Gérer le cycle de vie des composantes d'une icône

Contrôle du processus d'instanciation. À la recherche d'un patron de création ?

Patrons de création	**Variabilité fournie**
Fabrique abstraite (*Abstract Factory*)	Construire les objets d'une famille de classes
Monteur (*Builder*)	Construire un objet composite
Méthode Usine (*Factory method*)	Instantier des objets des sous-classes d'une classe
Prototype	Instancier les objets d'une classe par clonage
Singleton	La seule instance d'une classe

Patron **Méthode Usine**

- Intention

  Définir une interface pour créer un objet, mais laisser les sous-classes décider quelle classe concrète instancier. La Méthode Usine permet à une classe de déléguer l'instanciation à ses sous-classes.

- Applicabilité

  Ce patron est utilisé lorsque:

  – Une classe n'est pas en mesure d'anticiper la classe concrète de l'objet qu'elle doit créer,

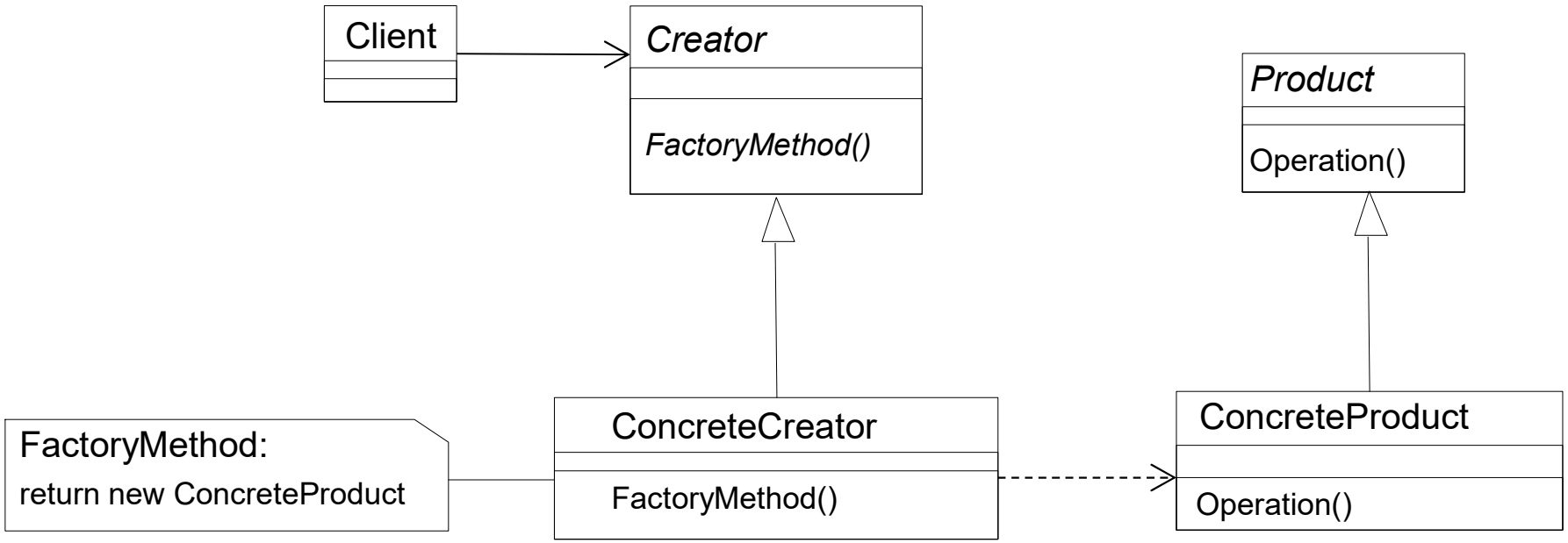
  – Une classe veut laisser à ses sous-classes la spécification de l'objet qu'elle va créer,

  – Des classes délèguent des responsabilités à une ou plusieurs sous-classes auxiliaires, et que la connaissance de la sous-classe auxiliaire qui est réellement utilisée doit être encapsulée.

POLYTECHNIQUE MONTRÉAL

Patron **Méthode Usine**

Structure



POLYTECHNIQUE MONTRÉAL

Patron **Méthode Usine**

- Conséquences
  - \+ Réduit le couplage: La *Méthode Usine* évite de lier les sous-classes spécifiques des objets à créer directement dans le code client. Le client peut donc fonctionner avec n'importe quelle classe de « produit concret ».
  - \- Force à créer une classe dérivée: Pour créer un nouveau type de « produit concret », il faut absolument créer une nouvelle sous-classe.
  - \+ Fournit un point d'extension: Créer des objets par une *Méthode Usine* est toujours plus flexible que de créer directement des objets avec un new. La *Méthode Usine* fournit un point d'extension pour permettre à des sous-classes de créer des versions spécialisées des produits concrets.

Patron **Méthode Usine**

- Implantation
  - La classe abstraite *Creator* peut fournir ou non une version par défaut de la *Méthode Usine*.
  - La Méthode Usine pourrait recevoir un paramètre lui indiquant le type concret d'objet à créer.
  - Lorsque les types de produits concrets à créer sont connus à la compilation, on peut utiliser un template plutôt que de sous-classer.
  - Adopter une convention de nom pour les Méthode Usines pour annoncer clairement l'utilisation du patron.

Patron **Méthode Usine** : gérer le cycle de vie des composantes d'une icône

- Les primitives sont ajoutées comme des enfants dans l'arbre: éléments de la classe Icone3DComposite.
  - La classe Icone3DComposite conserve des copies des enfants dont elle est l'unique propriétaire: **`std::unique_ptr<Icone3DAbs>`**
- Chaque classe d'enfant est responsable de se copier :
  - Méthode **`clone()`**
- Rôles:
  - Creator: Icone3DAbs
  - ConcreteCreator: Icone3DComposite
  - Product: Icone3DAbs
  - ConcreteProduct: toutes les classes dérivées de Icone3DAbs
- La méthode **`clone()`** agit comme une Méthode Usine.

POLYTECHNIQUE MONTRÉAL

Patron **Méthode Usine** : gérer le cycle de vie des composantes d'une icône

```
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;
        …
};

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;
        …
};
```

POLYTECHNIQUE MONTRÉAL

Patron **Méthode Usine** : gérer le cycle de vie des composantes d'une icône

Une primitive se copie en construisant un nouvel objet en appelant son constructeur de copie:

```
Sphere::Sphere(Object3DAbs & parent, const Sphere & sph)
         : PrimitiveAbs(parent,sph.m_center), m_radius(sph.m_radius)
{
}

Sphere * Sphere::clone(Icone3DAbs& parent) const
{
   return new Sphere( parent, *this );
}
```

POLYTECHNIQUE MONTRÉAL

Patron **Méthode Usine** : gérer le cycle de vie des composantes d'une icône

Le composite se copie en construisant des nouveaux objets en appelant la méthode clone de chacun de ses enfants:

```
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);
        …
};
```

Patron **Méthode Usine** : gérer le cycle de vie des composantes d'une icône

```
Icone3DComposite::Icone3DComposite(Icone3DAbs& parent,
                                   const Icone3DComposite & mdd)
   : Icone3DAbs(parent)
{
   for (auto it = mdd.cbegin(); it != mdd.cend(); ++it)
      addChild(*it);
}

Icone3DComposite * Icone3DComposite::clone(Icone3DAbs& parent) const
{
   return new Icone3DComposite( parent, *this );
}

void Icone3DComposite::addChild(const Icone3DAbs& ico3d)
{
   m_Icone3DContainer.push_back(Icone3DPtr(ico3d.clone(*this)));
}
```

Gérer le cycle de vie des composantes d'une icône

La méthode clone:

- S'assure que les éléments du composite sont créés correctement,
- Fournit un mécanisme de création des éléments du composite qui facilite l'ajout de nouveaux types d'éléments,
- Évite que la classe de base du Composite connaisse toutes les classes d'éléments.

7 – Contrôler le processus d'animation d'une icône

- Problème de conception. On veut:
  - Permettre aux développeurs d'applications de réalité augmentée d'animer facilement les icônes dans leurs applications.
  - Fournir un mécanisme d'animation des icônes qui puisse être configuré par les développeurs.
  - Conserver un certain contrôle sur le processus d'animation.

7 – Contrôler le processus d'animation d'une icône

Des questions sur le processus d'animation:

- Comment le processus d'animation se déroule-t-il ?
- Comment modifie-t-il les primitives d'une icône ?
- Y-a-t-il plusieurs algorithmes possibles pour animer une icône ?
- Comment peut-on laisser les développeurs définir leur propre algorithme d'animation ?

POLYTECHNIQUE MONTRÉAL

Contrôler le processus d'animation d'une icône

Quelques observations sur le problème à régler:

- On voudrait fournir un algorithme d'animation par défaut qui puisse être reconfiguré par les développeurs.
- Certaines étapes de l'algorithme pourraient être implémentées par des méthodes par défaut qui ne seront probablement pas modifiées par les développeurs. Voici un algorithme générique d'animation :
  - **`Sélectionner tous les décorateurs de transformation`**
  - **`Itérer sur les pas de temps de l'animation`**
    - `Itérer sur les décorateurs`
      - `Modifier la transformation contenue dans le décorateur`
    - `Afficher l'icône`
  - **`Rénitialiser l'icône après l'animation`**
- Il y a donc des **éléments communs** à toutes les sous-classes et des **éléments variables** selon les sous-classes
  - Ceci nous amène à regarder du côté du patron **Patron de méthode**.

Contrôler le processus d'affichage d'une icône

- Le patron **Patron de méthode**
  - Permet de définir un squelette d'algorithme dont certaines parties seront configurées dans les sous-classes
  - Pas de coût associé à l'ajout d'objet ou à la délégation.
- Pour l'appliquer, il faut déterminer les portions *variables* et *invariables*
  - Invariable: enchainement des opérations de l'algorithme d'animation
  - Variable: opération spécifiques de l'algorithme

Patron **patron de méthode**

- Intention

  Définir le squelette d'un algorithme dans une opération, et laisser les sous-classes définir certaines étapes.

- Applicabilité

  Pour implanter les aspects invariants d'un algorithme une seule fois et laisser les sous-classes définir les portions variables.

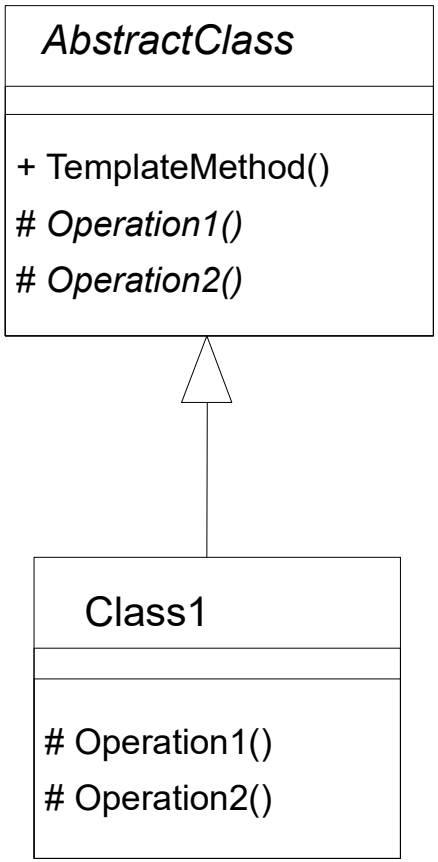
  Pour situer les comportements communs dans une classe afin d'augmenter la réutilisation de code.

  Pour contrôler les extensions des sous-classes.

POLYTECHNIQUE MONTRÉAL

Patron **patron de méthode**

- Structure



Operation1() et Operation2() sont abstraites dans la classe de base.

TemplateMethod() est non-virtuelle et appelle les méthode Operation1() et Operation2()

Patron **patron de méthode**

- Conséquences
  + mène à une inversion de contrôle («Principe Hollywood: ne nous appelez pas, nous vous appellerons »)
  + favorise la réutilisation de code
  + permet d'imposer des règles de surcharge
  - il faut sous-classer pour spécialiser le comportement.
- Implémentation
  – méthode Template virtuelle ou non-virtuelle
  – peu ou beaucoup d'opérations primitives
  – convention de noms (préfix do- ou faire-)

Cadre d'applications (« *framework* »)

- Il s'agit d'un *ensemble de classes en interaction* qui constituent une conception réutilisable pour une problématique spécifique.
- Le cadre va diriger l'architecture de l'application, et le concepteur d'application s'attardera plutôt aux aspects qui requièrent spécifiquement son attention.
- Par exemple, les classes du cadre vont *appeler les classes définies par l'utilisateur de celui-ci* (et non l'inverse)!
  - Exemple: le patron **Patron de méthode**

Contrôler le processus d'animation d'une icône

```
class AnimatorAbs
{
public:
   Animator( Icone3DAbs& icone );
   void animate();      // n'est pas une methode virtuelle
   [...]
protected:
   virtual void doSelectDecorators(Icone3DIteratorContainer& iters);
   virtual bool doIterateTime();
   virtual void doAnimatePrimitive(TransformedIcone3D& decor);
   virtual void doDisplayIcone() = 0;
   virtual void doResetIcone();

   Icone3DAbs& m_icone;
};
```

Contrôler le processus d'animation d'une icône

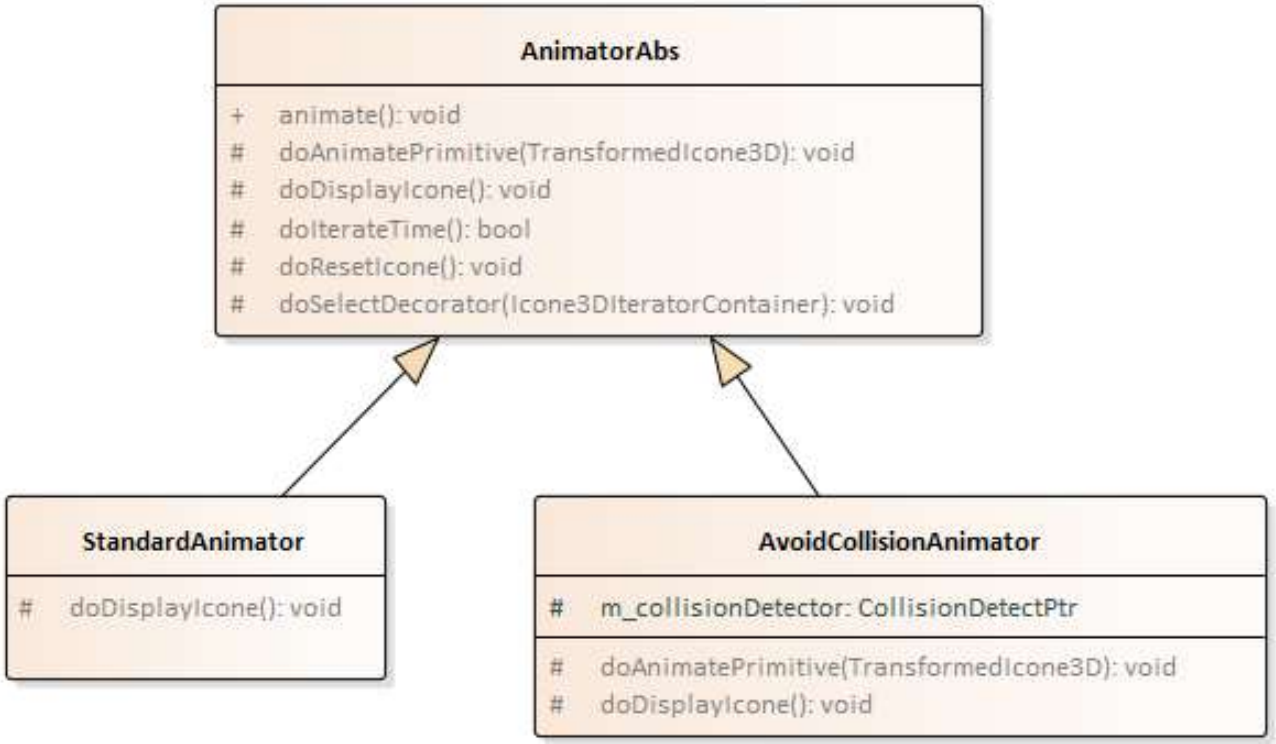
```
AnimatorAbs::Animate()
{
    Icone3DIteratorContainer iters;
    doSelectDecorators(iters);
    while( doIterateTime() )
    {
        for( auto& it : iters )
            doAnimatePrimitive(it);
        doDisplayIcone();
    }
    doResetIcone();
};
```

POLYTECHNIQUE MONTRÉAL

Contrôler le processus d'animation d'une icône

La classe de base peut fournir ou non des implémentations par défaut pour les différentes opérations.

Les classes dérivées ne peuvent pas changer le squelette de l'algorithme, mais peuvent modifier chaque étape selon les besoins spécifiques de la sous-classe.



Contrôler le processus d'animation d'une icône

```
class StandardAnimator : public AnimatorAbs
{
public:
   StandardAnimator( Icone3DAbs& icone );

protected:
   virtual void doDisplayIcone();
};

class AvoidCollisionAnimator : public AnimatorAbs
{
public:
   AvoidCollisionAnimator( Icone3DAbs& icone );

protected:
   virtual void doAnimatePrimitive(TransformedIcone3D& decor);
   virtual void doDisplayIcone();
};
```

8 – Choix de l'algorithme de détection de collision

- Problème de conception: choix de l'algorithme d'animation. On veut :
  - Proposer différentes versions de l'algorithme de détection de collision.
  - Fournir un mécanisme pour choisir dynamiquement l'algorithme qui sera utilisé.
  - Ne pas redéfinir un objet dérivé de la classe **`AnimatorAbs`** pour chaque algorithme de détection de collision.

Choisir l'algorithme de détection des collisions

Patrons de conception comportemental	**Variabilité fournie par le patron**
Chaîne de Responsabilité	L'objet qui répond à une requête
Commande	Quand et comment une requête est traitée
Interpréteur	Grammaire et interprétation d'un langage
Itérateur	Comment les éléments d'un agrégat sont accédés
Médiateur	Comment et quels objets interagissent ensemble
Memento	Quelle information privée d'un objet est stockée
Observateur	Quels objets dépendent d'un autre et comment
État	L'état d'un objet
Stratégie	Un algorithme
Patron de méthode	Les étapes d'un algorithme
Visiteur	Les opérations applicables à des objets

Patron **Stratégie**

Intention

Encapsuler un algorithme dans une classe de façon à le rendre **interchangeable**. Le patron Stratégie permet de faire varier l'algorithme indépendamment du client qui l'utilise.

Applicabilité

Plusieurs classes ne diffèrent que par leur comportement. On peut alors configurer une classe avec plusieurs comportements indépendants sans multiplier le nombre de sous-classes.

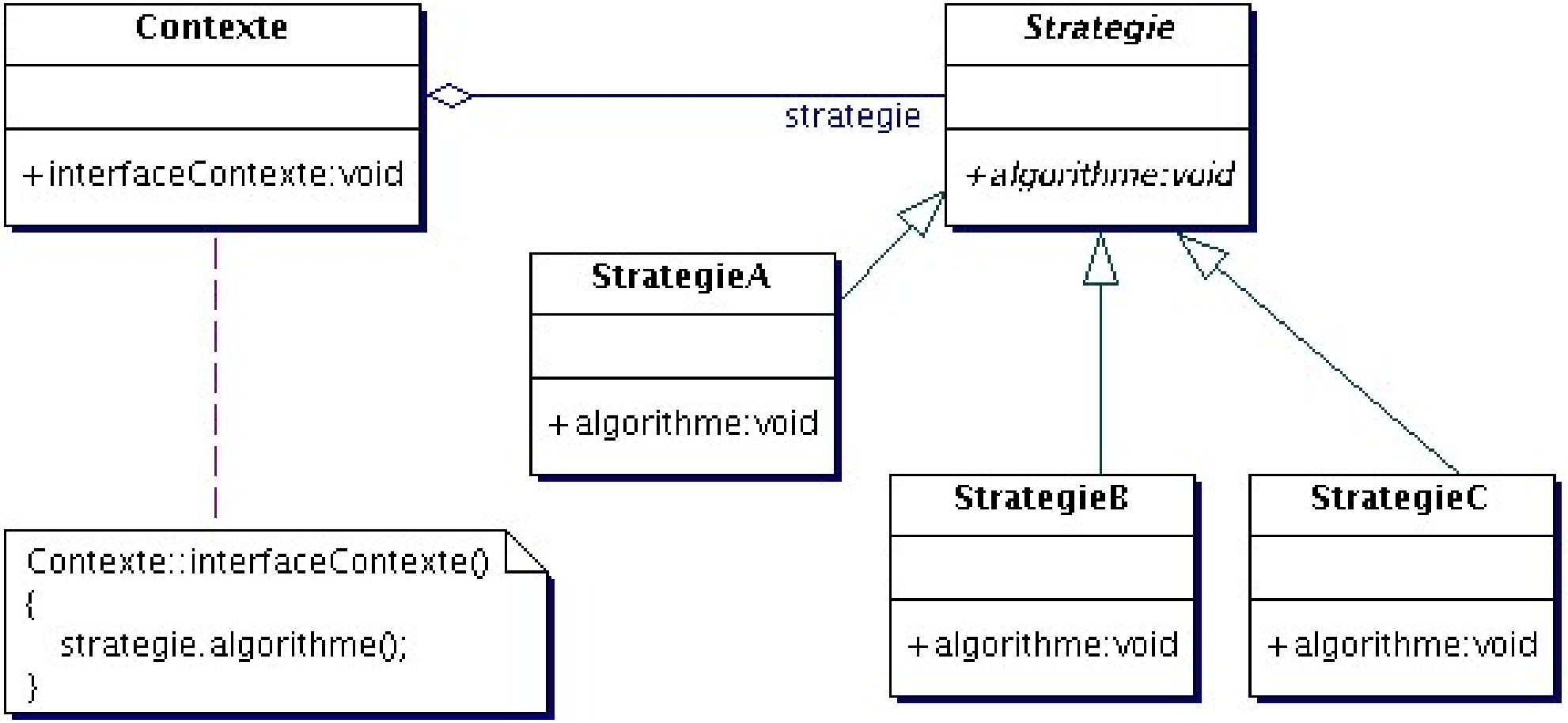
Plusieurs variantes d'un algorithme sont nécessaires.

Un algorithme requiert et stocke plusieurs données. La stratégie permet d'encapsuler ces données.

Une classe a plusieurs comportements qui impliquent de multiples énoncés conditionnels.

POLYTECHNIQUE MONTRÉAL

Patron Stratégie



Patron Stratégie

Conséquences

+ Support de familles d'algorithmes reliés.
+ Alternative au sous-classement. Permet d'obtenir des combinaisons d'algorithmes en limitant le sous-classement.
+ Élimination d'énoncés conditionnels.
+ Choix d'implantations libre.
- Les clients doivent être conscients de l'existence des stratégies et comprendre les nuances entre chacune afin de choisir la bonne. Les clients sont donc exposés à l'implémentation.
- Coût de communication supplémentaire pour l'appel à la fonction virtuelle de la stratégie.
- Augmente le nombre d'objets dans le système. Certaines stratégies peuvent être implémentées dans des objets partagés pour réduire ce problème.

Patron Stratégie

Implémentation

1. Définition de l'interface des classes **`Contexte`** et **`Strategie`**
   - Les stratégies concrètes doivent généralement avoir un accès efficace aux données du contexte.
   - Toutes ces informations pourraient être passées en paramètres à l'algorithme dans la stratégie (modèle « *pousser* »).
     - **Garde le découplage** entre les stratégies et le contexte.
     - **Risque de passer de l'information inutile**.
   - Le contexte pourrait se passer lui-même en paramètre à l'algorithme de la stratégie et la stratégie pourrait alors interroger le contexte pour obtenir seulement les informations nécessaires (modèle « *tirer* »).
     - **Permet d'éviter le passage d'information inutile**.
     - **Augmente le couplage** entre les stratégies et le contexte.
   - Une alternative intéressante serait de passer les informations nécessaires **au moment de la construction de la stratégie concrète**, de façon à *passer uniquement les informations nécessaires sans augmenter le couplage*.

Patron Stratégie

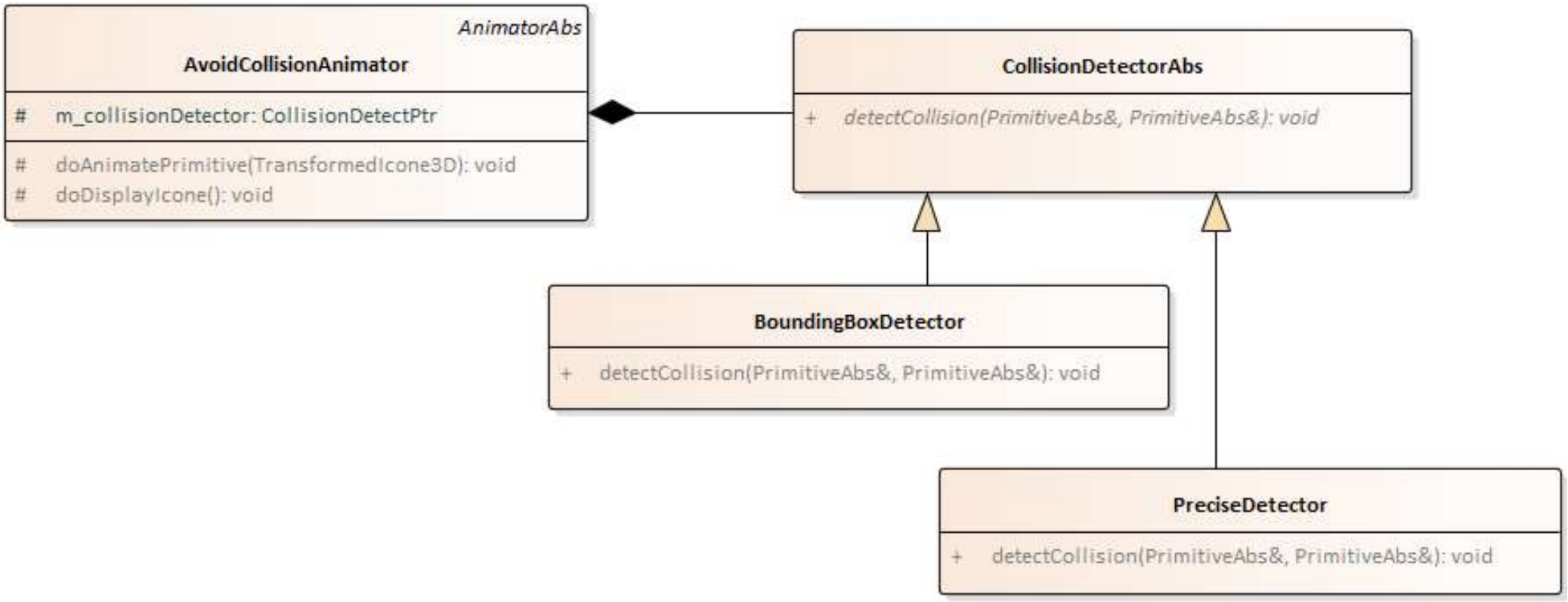
Implantation

2. La stratégie comme un paramètre template du contexte.
   - Une option serait de définir la classe contexte comme une classe template dont le paramètre est la stratégie à utiliser.
   - Possible seulement si on peut définir la stratégie à utiliser au moment de la compilation (si on doit changer dynamiquement de stratégie, c'est impossible).
   - Élimine le besoin d'une classe abstraite pour les stratégies.
   - Permet de lier une stratégie à son contexte plus efficacement.
3. Rendre les objets stratégies optionnels.
   - Dans les cas où il est envisageable de ne pas avoir de stratégie, le contexte peut définir un comportement par défaut qui est utilisé si aucune stratégie n'est spécifiée.
   - Les clients n'ont pas à travailler avec les stratégies si le comportement par défaut leur convient.

POLYTECHNIQUE MONTRÉAL

Choix de l'algorithme de détection de collision

Permettre de choisir l'algorithme de collision dans la classe **`AvoidCollisionAnimator`**



Choix de l'algorithme de détection de collision

Permettre de choisir l'algorithme de collision dans la classe **`AvoidCollisionAnimator`**

```
using CollisionDetectPtr =
std::unique_ptr<CollisionDetectStrategieAbs>

class AvoidCollisionAnimator : public AnimatorAbs
{
public:
   AvoidCollisionAnimator( Icone3DAbs& icone );

protected:
   virtual void doAnimatePrimitive(TransformedIcone3D& decor);
   virtual void doDisplayIcone();

   CollisionDetectPtr m_collisionDetector;
};
```

POLYTECHNIQUE MONTRÉAL

Choix de l'algorithme de détection de collision

La classe **`AvoidCollisionAnimator`** délègue à la stratégie la détection des collisions

```
AvoidCollisionAnimator::doAnimatePrimitive(TransformedIcone3D& decor)
{
   Icone3DIteratorContainer iters;
   selectPrimitiveBoundingBox(decor, iters);
   for( auto& it : iters )
      if( m_collisionDetector->detectCollision(decor, it) )
         [… traiter la collision]
};
```