POLYTECHNIQUE MONTRÉAL

CHAPITRE 19

Patrons de conception :
Singleton, Observateur et Façade

POLYTECHNIQUE MONTRÉAL

Sommaire

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

POLYTECHNIQUE MONTRÉAL

12 – Permettre de facilement invoquer des commandes sur les icônes

- Problème de conception:
  - Fournir à différentes composantes de l'application d'édition des icônes la possibilité d'invoquer des commandes.
  - Éviter de passer à chaque composante un pointeur sur l'objet **`Invoker`**.
  - Fournir un point d'accès global à l'objet sans utiliser une variable globale.

POLYTECHNIQUE MONTRÉAL

12 – Permettre de facilement invoquer des commandes sur les icônes

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

Patrons de création	**Variabilité fournie**
Abstract Factory	Construire les objets d'une famille de classes
Builder	Construire un objet composite
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 **Singleton**

- Intention

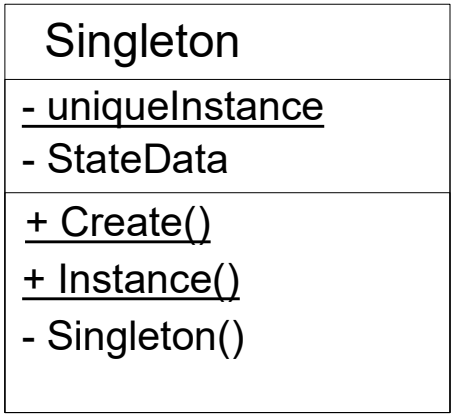
  S'assurer qu'il ne soit possible de créer qu'une seule instance d'une classe, et fournir un point d'accès global à cette instance.

- Applicabilité

  Lorsqu'il doit y avoir exactement une seule instance d'une classe, et que cette instance soit accessible par un mécanisme bien identifié.

  Lorsque la seule instance d'une classe doit pouvoir être étendue en sous-classant et que les clients doivent être en mesure d'utiliser l'instance dérivée sans modification à leur code.

- Structure



Patron **Singleton**

```
class Invoker
{
public:
        virtual ~Invoker() = default;
        static Invoker* getInstance(void);
        virtual void execute(CmdPtr& cmd);
        virtual void undo();
        virtual void redo();
protected:
        Invoker() = default;

        static std::unique_ptr<Invoker> m_instance;
        CmdContainer m_cmdDone;
        CmdContainer m_cmdUndone;
};
```

Patron **Singleton**

Invoker.cpp

```
[...]

std::unique_ptr<Invoker> Invoker::m_instance(nullptr);

Invoker * Invoker::getInstance(void)
{
   if (m_instance == nullptr)
      m_instance = std::unique_ptr<Invoker>(new Invoker);
   return m_instance.get();
}
```

Patron **Singleton**

- Conséquences
  - \+ Réduit la pollution du *namespace* global
  - \+ Permet le rafinement par sous-classification (comparativement à une classe équivalente dont toutes les fonctions seraient statiques)
  - \+ Permet de contrôler l'instanciation d'une classe (on peut limiter le nombre d'instances à 1, 2, n.)
  - \- Implantation peut être légèrement moins efficace qu'une variable globale
- Implémentation
  - ➢Opération getInstance() statique
  - ➢Enregistrement de l'instance du Singleton

Patron **Singleton**

Extrait des conséquences du patron Singleton

- Permet de raffiner les opérations et la représentation

  La classe Singleton peut être sous-classée, et il est facile de configurer une application avec une instance de la classe étendue. On peut configurer l'application avec l'instance de la classe dont on a besoin à l'exécution.

- Permet plus d'une instance de la classe

  « [Singleton] permet un nombre variable d'instances. Ce patron permet facilement de changer d'idée pour permettre plus d'une instance de la classe Singleton. De plus, on peut utiliser la même approche pour contrôler le nombre d'instances qu'une application utilise. Seule la méthode qui donne accès à l'instance du Singleton (getInstance) doit être modifiée pour changer les règles d'accès. »

Patron **Singleton**: classe dérivée

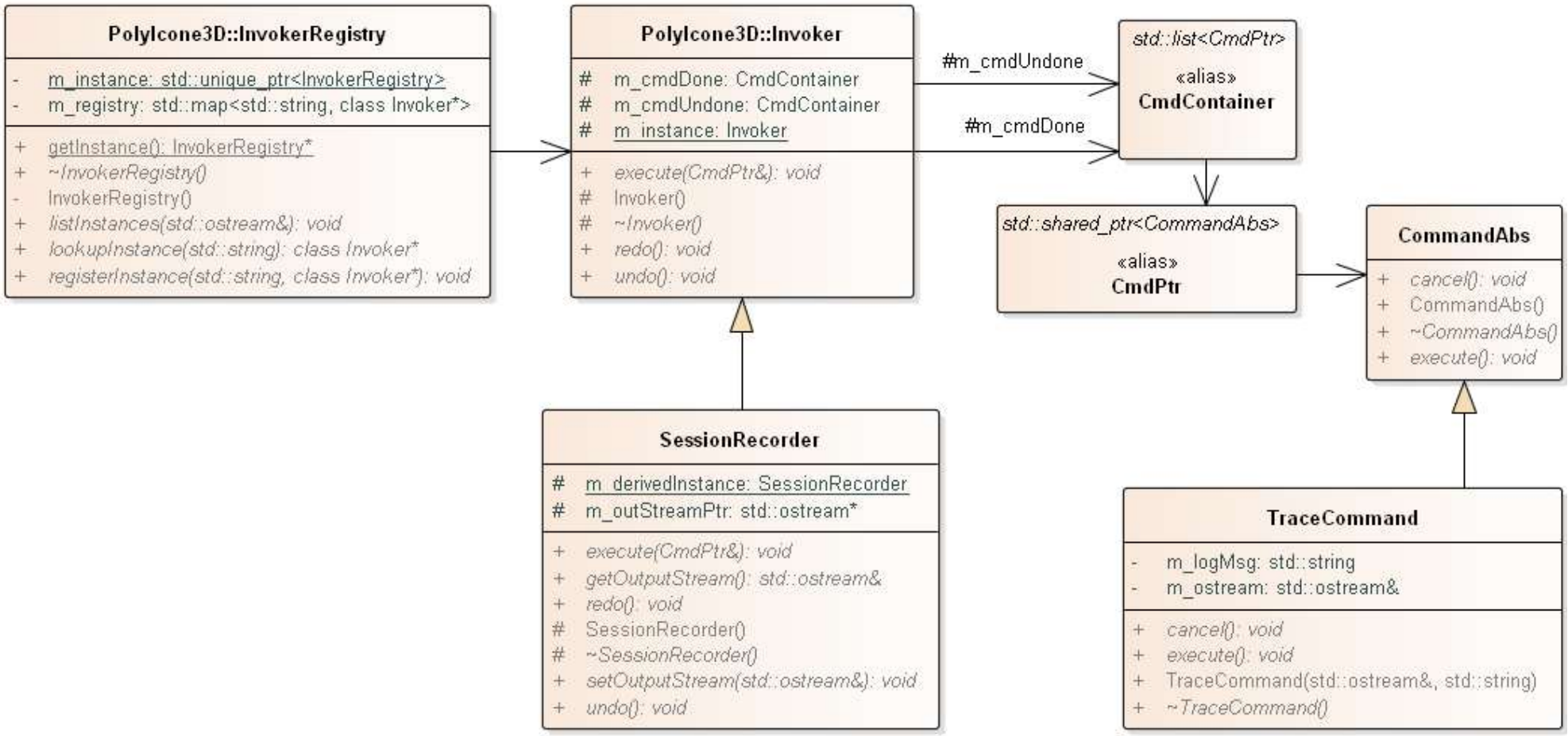
Créer une classe dérivée d'une classe Singleton est facile. Accéder à la fois à la classe de base et à la classe dérivée peut être plus complexe:

- Comment le client peut-il choisir parmi les différentes versions disponibles ?
- Est-ce que toutes les classes sont créées au démarrage de l'application ou seulement au moment où elles sont utilisées ?
- Une approche simple consiste à décider de la classe à instancier lors de l'appel de la méthode `getInstance(type_t)` en spécifiant un type à la méthode.
  - Toutes les classes de Singleton doivent être connues dans la méthode.
- Une approche plus flexible consiste à construire un **registre** de Singletons. Chaque Singleton, lors de sa création, s'enregistre dans le registre. Les clients peuvent choisir le bon Singleton en interrogeant le registre.

POLYTECHNIQUE MONTRÉAL

Patron **Singleton**: classe dérivée

La classe **`SessionRecorder`**, dérivée de la classe **`Invoker`**, permet d'enregistrer dans un fichier de trace les commandes exécutées, annulées et réexécutées.



POLYTECHNIQUE MONTRÉAL

Patron **Singleton**: la classe de registre

La classe **`InvokerRegistry`** sert de registre des Singletons. Chaque Singleton, en se créant, s'enregistre dans le registre. La classe **`InvokerRegistry`** est un Singleton.

```
class InvokerRegistry
{
public:
        virtual ~InvokerRegistry() = default;

        static InvokerRegistry* getInstance();
        virtual void registerInstance(std::string invokerName,
                                      class Invoker* instance);
        virtual class Invoker* lookupInstance(std::string invokerName);
        virtual void listInstances(std::ostream& o);

private:
        InvokerRegistry() = default;
        static std::unique_ptr<InvokerRegistry> m_instance;
        std::map<std::string, class Invoker*> m_registry;
};
```

Patron **Singleton**: la classe de registre

```
std::unique_ptr<InvokerRegistry> InvokerRegistry::m_instance(nullptr);

InvokerRegistry* InvokerRegistry::getInstance() {
   if (m_instance == nullptr)
      m_instance = std::unique_ptr<InvokerRegistry>(new InvokerRegistry);
   return m_instance.get();
}

void InvokerRegistry::registerInstance(std::string invokerName,
                                       Invoker * instance) {
   m_registry[invokerName] = instance;
}

Invoker* InvokerRegistry::lookupInstance(std::string invokerName) {
   return  m_registry[invokerName];
}
```

Patron **Singleton**: la classe dérivée

La classe **`SessionRecorder`** dérive de la classe **`Invoker`**.

```
class SessionRecorder : public Invoker
{
public:
   virtual void setOutputStream(std::ostream& o) { m_outStreamPtr = &o; };
   virtual std::ostream& getOutputStream(void) { return *m_outStreamPtr; };

   virtual void execute(CmdPtr& cmd);
   virtual void redo();
   virtual void undo();

protected:
   SessionRecorder();
   virtual ~SessionRecorder() = default;

   std::ostream* m_outStreamPtr;
   static SessionRecorder m_derivedInstance;
};
```

POLYTECHNIQUE MONTRÉAL

Patron **Singleton**: la classe dérivée

La classe **`SessionRecorder`** dérive de la classe **`Invoker`**.

```
SessionRecorder SessionRecorder::m_derivedInstance;

SessionRecorder::SessionRecorder(void)
        : m_outStreamPtr(&std::cerr)
{
        InvokerRegistry* registry = InvokerRegistry::getInstance();
        registry->registerInstance("SessionRecorder", &m_derivedInstance);
}

void SessionRecorder::execute(CmdPtr & cmd)
{
        (*m_outStreamPtr) << "Avant execute: nombre de commandes faites: "
        << m_cmdDone.size() <<
        ", nombre de commandes defaites:" << m_cmdUndone.size() << std::endl;
        cmd->execute();
        m_cmdDone.push_back(cmd);
        (*m_outStreamPtr) << "Apres execute: nombre de commandes faites: "
        << m_cmdDone.size() <<
        ", nombre de commandes defaites:" << m_cmdUndone.size() << std::endl;
}
```

POLYTECHNIQUE MONTRÉAL

Patron **Singleton**: la classe dérivée

La classe **`SessionRecorder`** dérive de la classe **`Invoker`**.

```
void SessionRecorder::undo()
{
        (*m_outStreamPtr) << "Avant Undo: nombre de commandes faites:"
        << m_cmdDone.size() <<
        ", nombre de commandes defaites:" << m_cmdUndone.size() << std::endl;
        if (!m_cmdDone.empty())
        {
                CmdPtr cmd = m_cmdDone.back();
                cmd->cancel();
                m_cmdDone.pop_back();
                m_cmdUndone.push_back(cmd);
        }
        (*m_outStreamPtr) << "Apres Undo: nombre de commandes faites: "
         << m_cmdDone.size() <<
        ", nombre de commandes defaites:" << m_cmdUndone.size() << std::endl;
}
```

POLYTECHNIQUE MONTRÉAL

Patron **Singleton**: la classe dérivée

La classe **`SessionRecorder`** dérive de la classe **`Invoker`**.

```
void SessionRecorder::redo()
{
        (*m_outStreamPtr) << "Avant Redo: nombre de commandes faites:"
        << m_cmdDone.size() <<
        ", nombre de commandes defaites:" << m_cmdUndone.size() << std::endl;
        if (!m_cmdUndone.empty())
        {
                CmdPtr cmd = m_cmdUndone.back();
                cmd->execute();
                m_cmdUndone.pop_back();
                m_cmdDone.push_back(cmd);
        }
        (*m_outStreamPtr) << "Apres Redo: nombre de commandes faites:"
        << m_cmdDone.size() <<
        ", nombre de commandes defaites:" << m_cmdUndone.size() << std::endl;
}
```

Patron **Singleton**: la classe dérivée

Les clients peuvent sélectionner le bon Singleton grâce à la méthode **`InvokerRegistry::lookupInstance()`**.

```
// Tester l'instance de l'Invoker standard
Invoker* invoker = registry->lookupInstance("Invoker");
if (invoker == nullptr)
{
   std::cerr << "Erreur Fatale: l'Invoker standard n'a pas ete correctement"
             << " enregistre" << std::endl;
}
else
{
   // Creer des commandes
   CmdPtr cmd1 = std::make_shared<TraceCommand>(std::cout, "Une 1ere commande");
   CmdPtr cmd2 = std::make_shared<TraceCommand>(std::cout, "Une 2eme commande");

   invoker->execute(cmd1);
   invoker->execute(cmd2);
}
```

POLYTECHNIQUE MONTRÉAL

Patron **Singleton**: la classe dérivée

```
// Tester l'instance du SessionRecorder
invoker = registry->lookupInstance("SessionRecorder");
if (invoker == nullptr) {
   std::cerr << "Erreur Fatale: le SessionRecorder n'a pas ete correctement "
             << " enregistre" << std::endl;
}
else {
   SessionRecorder* recorder = dynamic_cast<SessionRecorder*>(invoker);
   if (recorder == nullptr) {
      std::cerr << "Erreur Fatale: la conversion de type n'a pas fonctionne"
                <<  std::endl;
   }
   else {
      std::ofstream logFile("session.out");
      // Indiquer au SessionRecorder le fichier d'enregistrement
      recorder->setOutputStream(logFile);
      // Creer des commandes
      CmdPtr cmd1 = std::make_shared<TraceCommand>(logFile, "Une 1ere commande");
      CmdPtr cmd2 = std::make_shared<TraceCommand>(logFile, "Une 2eme commande");
      invoker->execute(cmd1);
      invoker->execute(cmd2);
   }
}
```

13 – Réaffichage de la scène pendant l'animation

- Problème de conception: certains clients doivent être avertis en cas de changements d'état de l'icône.
- Exemple: en cours d'animation, chaque fois qu'une nouvelle position des primitives est calculée, il faut redessiner l'icône:
  - La position de chaque icône fait partie du modèle, et est associée à la couche de logique d'application dans l'architecture du logiciel,
  - Le représentation 3D de l'icône à l'écran fait partie de la vue, qui est associée à la couche de présentation dans l'architecture,
  - Le modèle ne peut pas appeler directement des méthodes de la vue,
  - Un mécanisme de mise à jour automatique doit être mis en place pour avertir la vue lorsque le modèle change.
- On ne veut pas obliger l'utilisateur à rafraîchir manuellement la vue pour voir la nouvelle configuration de l'icône.
- Autres exemples: nouveau courrier, documents imbriqués…

Avertissement de changement

Les notions de « *mise à jour automatique* », d'« *avertissement* » et de « *dépendance* » annoncent le patron **Observateur**, un patron particulièrement riche.

Son utilisation soulève plusieurs questions:

- Quelles classes jouent les rôles de sujets (abstrait et concrets) ?
- Quelles classes jouent les rôles d'observateurs (abstrait et concrets) ?
- Quel modèle de correspondance devrait-on utiliser pour la situation courante ?

Patron **Observateur**

- Intention

  Définit une relation un à plusieurs entre des objets de façon à ce que lorsqu'un objet change d'état, que tous ses dépendants soient avertis et mis à jour automatiquement.

- Applicabilité

  Lorsqu'une abstraction a deux aspects, l'un dépendant de l'autre.

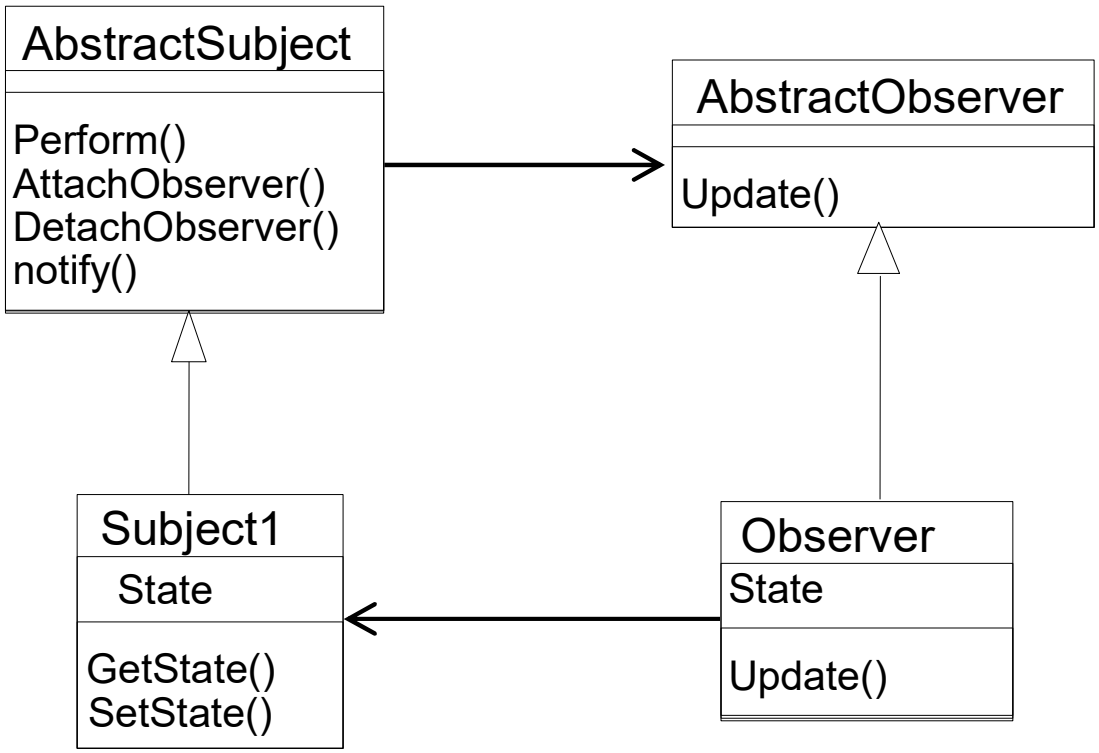
  Lorsque le fait de changer un objet implique d'en changer d'autres, sans que l'on sache combien d'autres objets doivent être changés.

  Lorsqu'un objet devrait en avertir d'autres sans faire de supposition sur la nature de ces autres objets.

POLYTECHNIQUE MONTRÉAL

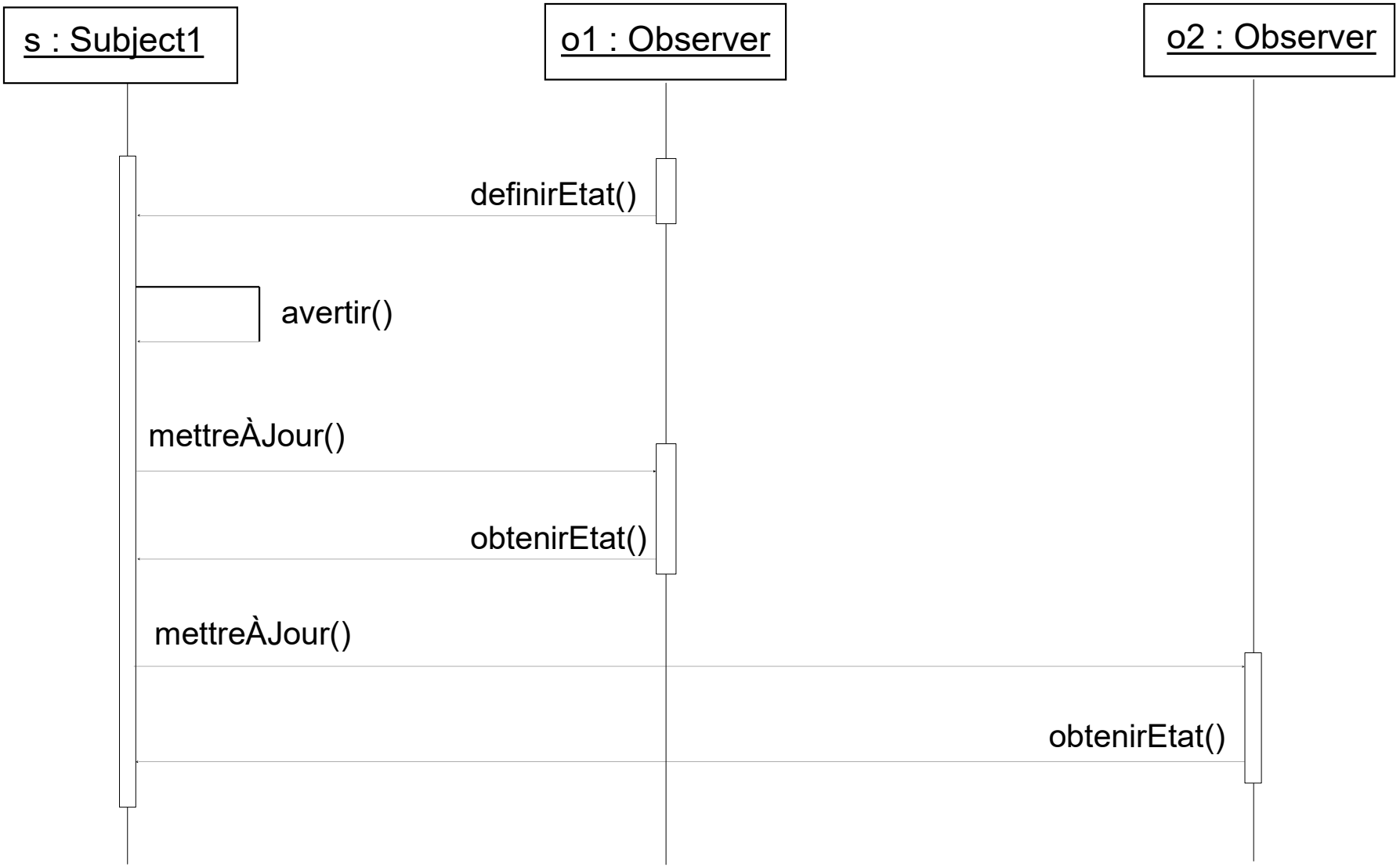
Patron **Observateur**

- Structure



POLYTECHNIQUE MONTRÉAL

Patron **Observateur** – Collaborations



Patron **Observateur**

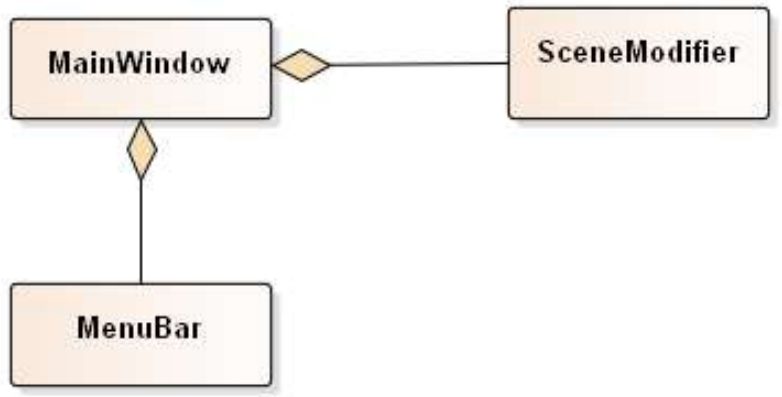
- **Conséquences**
  - \+ Modularité: Le sujet et les observateurs peuvent varier de façon indépendante.
  - \+ Extensibilité: On peut définir et ajouter autant d'observateurs que nécessaire.
  - \+ Adaptabilité: Différents observateurs fournissent différentes vues du Subject
  - \- Mises à jour inattendues: les Observers ne se connaissent pas
  - \- Coût de la mise à jour: certains Observers peuvent avoir besoin d'indices sur ce qui a changé
- **Implémentation**
  - ➢ Correspondance Sujet-Observateur… il y a plusieurs façon de l'implémenter.
  - ➢ Références pendantes.
  - ➢ Éviter des protocoles de mise à jour spécifiques à un Observer (modèle « tirer » vs. « pousser »).
  - ➢ Enregistrement explicite des modifications d'intérêt.

Réaffichage de la scène pendant l'animation

Parmi les classes servant à afficher l'icône en 3D, une classe est responsable de contrôler le contenu de la scène:

- La classe SceneModifier

Sous-ensemble très partiel des classes composant l'interface utilisateur:



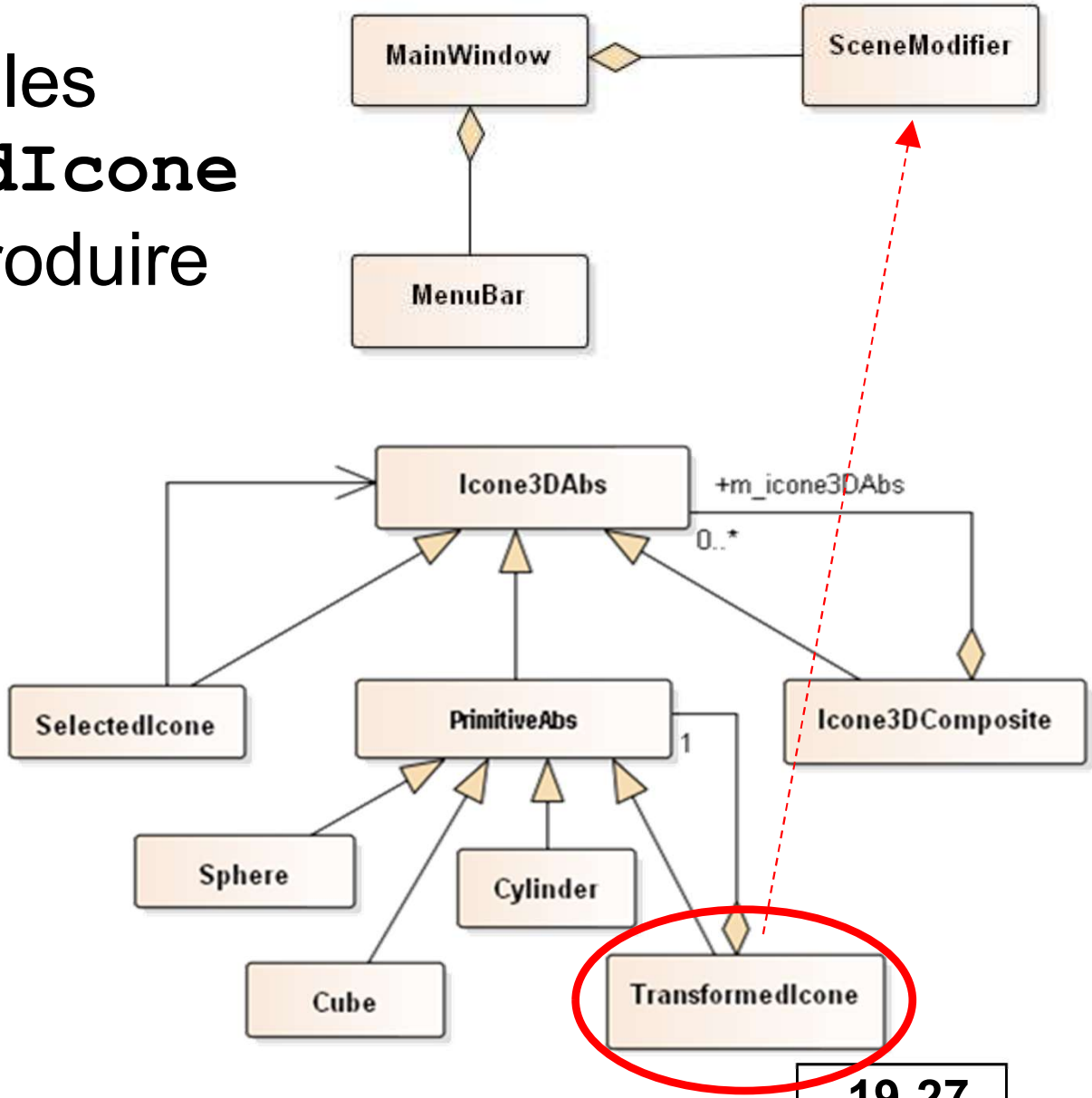
POLYTECHNIQUE MONTRÉAL

Réaffichage de la scène pendant l'animation

Pendant l'animation, les objets **`TransformedIcone`** sont modifiés pour produire l'animation.

Rôles:

- Sujet: chaque objet **`TransformedIcone`**
- Observateur: l'objet **`SceneModifier`**



POLYTECHNIQUE MONTRÉAL

Réaffichage de la scène pendant l'animation

Concevoir des classes de base pour les sujets et les observateurs permet de donner les caractéristiques de Sujet et d'Observateur à n'importe quelle classe.

```
class Sujet
{
public:
   virtual ~Sujet() = 0;
   void attach(Observateur&o);
   void detach(Observateur&o);
   void notify();
private:
   set<Observateur*> observateurs;
};
```

```
class Observateur
{
public:
   virtual ~Observateur();
   virtual update(Sujet&s)=0;
};
```

```
class TransformedIcone : public Sujet { ... };

class SceneModifier : public Observateur { ... };
```

Réaffichage de la scène pendant l'animation

```
void Sujet::attach(Observateur&o)
{
  observateurs.insert( &o );
}

void Sujet::detach(Observateur&o)
{
  observateurs.erase(
         observateurs.find( &o ) );
}

void Sujet::notify()
{
  for ( set<Observateur*>::iterator
        it = observateurs.begin();
        it != observateurs.end();
        ++it )
  {
    (*it)->update(*this);
  }
}
```

```
void SceneModifier::update(Sujet&s)
{
    // Mettre à jour l'affichage
    // de la scene
    [ ... ]
  }
}
```

Réaffichage de la scène pendant l'animation

MAIS !

Les objets **`TransformedIcone`** ne sont pas les seuls objets qui peuvent changer.

- Il faudrait plutôt définir toutes les composantes des icônes 3D comme sujets possibles: faire dériver la classe **Icone3DAbs** de la classe Sujet.

Chaque objet qui peut changer doit être observé:

- L'objet **`SceneModifier`** doit observer un très grand nombre d'objets.

Il peut y avoir d'autres objets de l'application qui veulent être avertis lorsque l'icône change.

Réaffichage de la scène pendant l'animation

Plusieurs sujets et plusieurs observateurs mènent à une relation plusieurs à plusieurs:

- Pour éviter à tous les objets d'assumer le coût des relations plusieurs à plusieurs, il vaut mieux centraliser cette responsabilité dans un objet spécialisé.
- Patron **mediator**: on introduit un gestionnaire de changement.

Avertissement de changement : Patron Observateur – Gestionnaire de changements

- On peut consolider les opérations du sujet dans un gestionnaire de changement.

```
class GestionnaireChangement
{
private:
    GestionnaireChangement();
    static GestionnaireChangement* instance;

    map< Sujet*, set<Observateur*>* > registre;

public:
    static GestionnaireChangement* getInstance();

    void enregistrer( Sujet& s, Observateur& o );
    void demobiliser( Sujet& s, Observateur& o );
    void avertir( Sujet& s );
};
```

Avertissement de changement : Patron Observateur – Gestionnaire de changements

```
GestionnaireChangement::enregistrer( Sujet& s,
                                     Observateur& o )
{
   set<Observateur*>* vo = registre[&s];
   if ( vo == NULL )
   {
      vo = new set<Observateur*>;
      registre[&s] = vo;
   }
   vo->insert( &o );
}
```

Avertissement de changement : Patron Observateur – Gestionnaire de changements

```
GestionnaireChangement::demobiliser( Sujet& s,
                                     Observateur& o )
{
   set<Observateur*>* vo = registre[&s];
   if ( vo != NULL )
   {
      vo->erase( vo->find( &o ) );
   }
}
```

Avertissement de changement : Patron Observateur – Gestionnaire de changements

```
GestionnaireChangement::avertir( Sujet& s )
{
   set<Observateur*>* vo = registre[&s];
   if ( vo != NULL )
   {
      for ( set<Observateur*>::iterator it = vo->begin();
            it != vo->end();
            ++it )
      {
         (*it)->mettreAJour( s );
      }
   }
}
```

Avertissement de changement : Patron Observateur – Gestionnaire de changements

```
class Sujet
{
Public:
   virtual ~Sujet() = 0;
};

class Observateur
{
Public:
   virtual ~Observateur();
   virtual void mettreAJour( Sujet& s ) = 0;
}
```

POLYTECHNIQUE MONTRÉAL

Avertissement de changement : Patron Observateur – Gestionnaire de changements

```
class Primitive : public Sujet
{
public:
   virtual ~Primitive(){};
   virtual const string& getNom() const;

private:
   string nom;
};

class Afficheur : public Observateur
{
public:
   virtual ~Observateur();
   virtual void mettreAJour( Sujet& s )
   {
      Primitive& n = dynamic_cast<Primitive&> ( s );
      cerr << "L'observateur " << nom
           << " est mis a jour par le sujet " << n.getNom();
   };

private:
   string nom;
}
```

POLYTECHNIQUE MONTRÉAL

Avertissement de changement : Patron Observateur – Gestionnaire de changements

Fonction main() :

```
void main( void )
{
   Primitive n1( "n1" );
   Primitive n2( "n2" );
   Afficheur e1( "e1" );
   Afficheur e2( "e2" );

   GestionnaireChangement::getInstance().enregistrer( n1, e1 );
   GestionnaireChangement::getInstance().enregistrer( n1, e2 );
   GestionnaireChangement::getInstance().enregistrer( n2, e2 );

   GestionnaireChangement::getInstance().avertir( n1 );
   GestionnaireChangement::getInstance().avertir( n2 );
}
```

Output obtenu :

```
L'observateur e2 est mis a jour par le sujet n1
L'observateur e1 est mis a jour par le sujet n1
L'observateur e2 est mis a jour par le sujet n2
```

Avertissement de changement : Patron Observateur

- Qui invoque la méthode avertir() ?
  - Le **sujet** peut l'invoquer lui-même lorsqu'il est modifié.
    - Rend le processus d'avertissement **automatique**.
    - Par contre, c'est **inefficace** dans les cas où plusieurs **modifications consécutives** sont faites sur le sujet.
  - Le **client** peut être chargé de déclencher explicitement le processus d'avertissement après avoir fait ses modifications sur un sujet donné.
    - **Efficace pour les modifications consécutives**.
    - Par contre, cela force les clients à démarrer le processus d'avertissement eux-mêmes (**on perd l'automatisme**).

Patron Observateur : Résumé

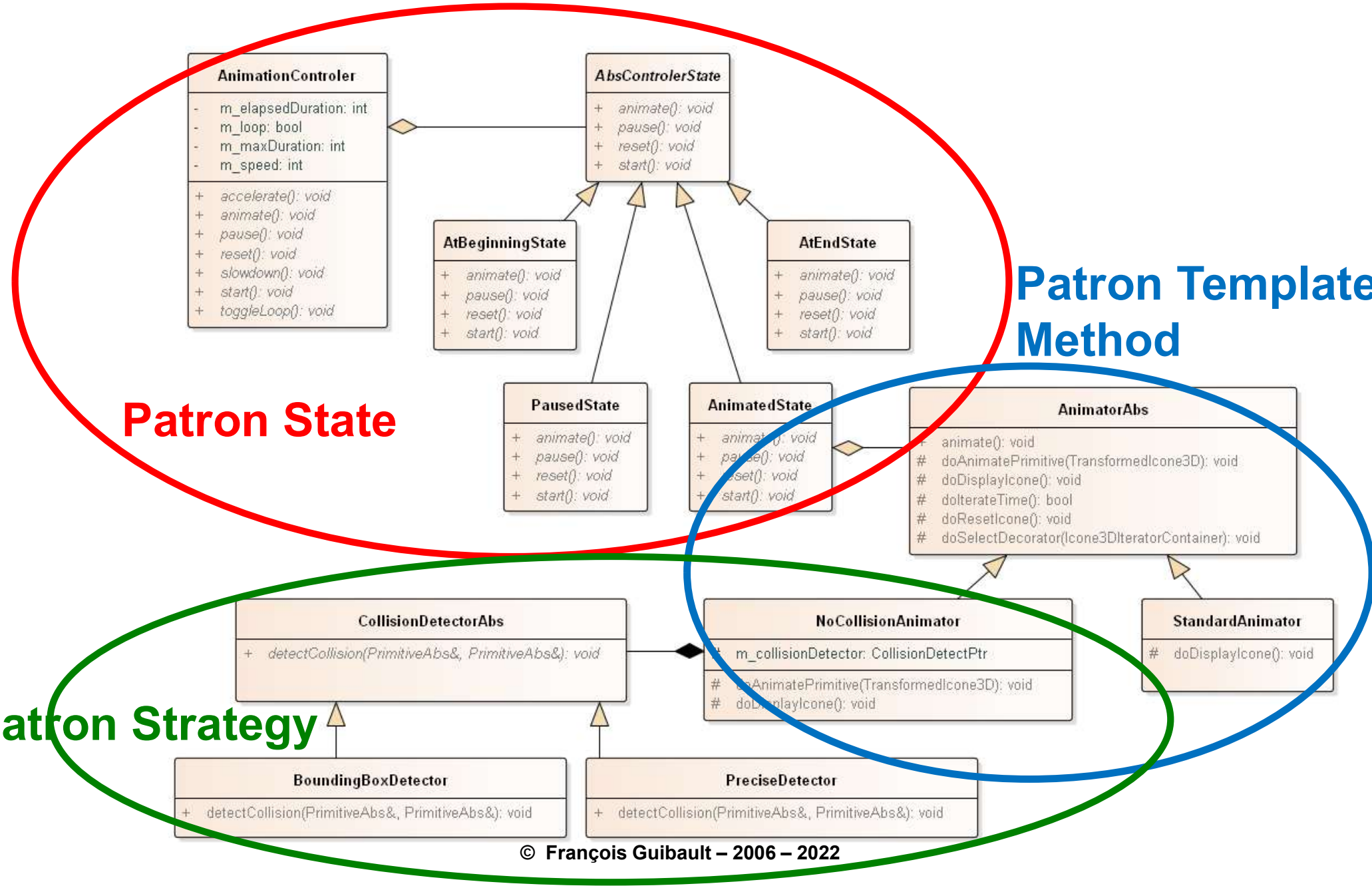
- Le patron **Observer** permet à des sujets d'avertir des entités (observateurs) qu'une autre entité (sujet) a changé, sans que les observateurs n'aient à connaître les sujets en détail.
- À sa plus simple expression, le patron **Observer** permet aux sujets d'enregistrer des observateurs qui seront avertis d'éventuelles modifications au sujet.
- On peut également utiliser un gestionnaire de changements qui permet de centraliser la gestion des associations sujets-observateurs. Ce gestionnaire de changements est en fait un **Médiateur** qui peut être implémenté sous forme de **Singleton**.
- Plusieurs bibliothèques et systèmes de classes sont déjà disponibles qui fournissent des mécanismes d'enregistrement et de rappel, dont la bibliothèque `signals2` de Boost.
- Un mécanisme de signaux et de réception de signaux peut être mis en place pour traiter différents évènements non connus à l'avance.

14 – Fournir une interface simple au module d'animation

- Comment pourrait-on rendre les classes d'animation plus faciles à utiliser par les clients ?
  - On voudrait rendre facile l'accès aux fonctionnalités les plus usuelles.
  - On voudrait présenter une interface claire et simple aux classes d'animation.
  - La plupart des clients ne sont pas intéressés à connaître tous les détails de la structure interne du module d'animation. Ils veulent juste l'utiliser !

POLYTECHNIQUE MONTRÉAL

Le module d'animation des icônes



Patron State

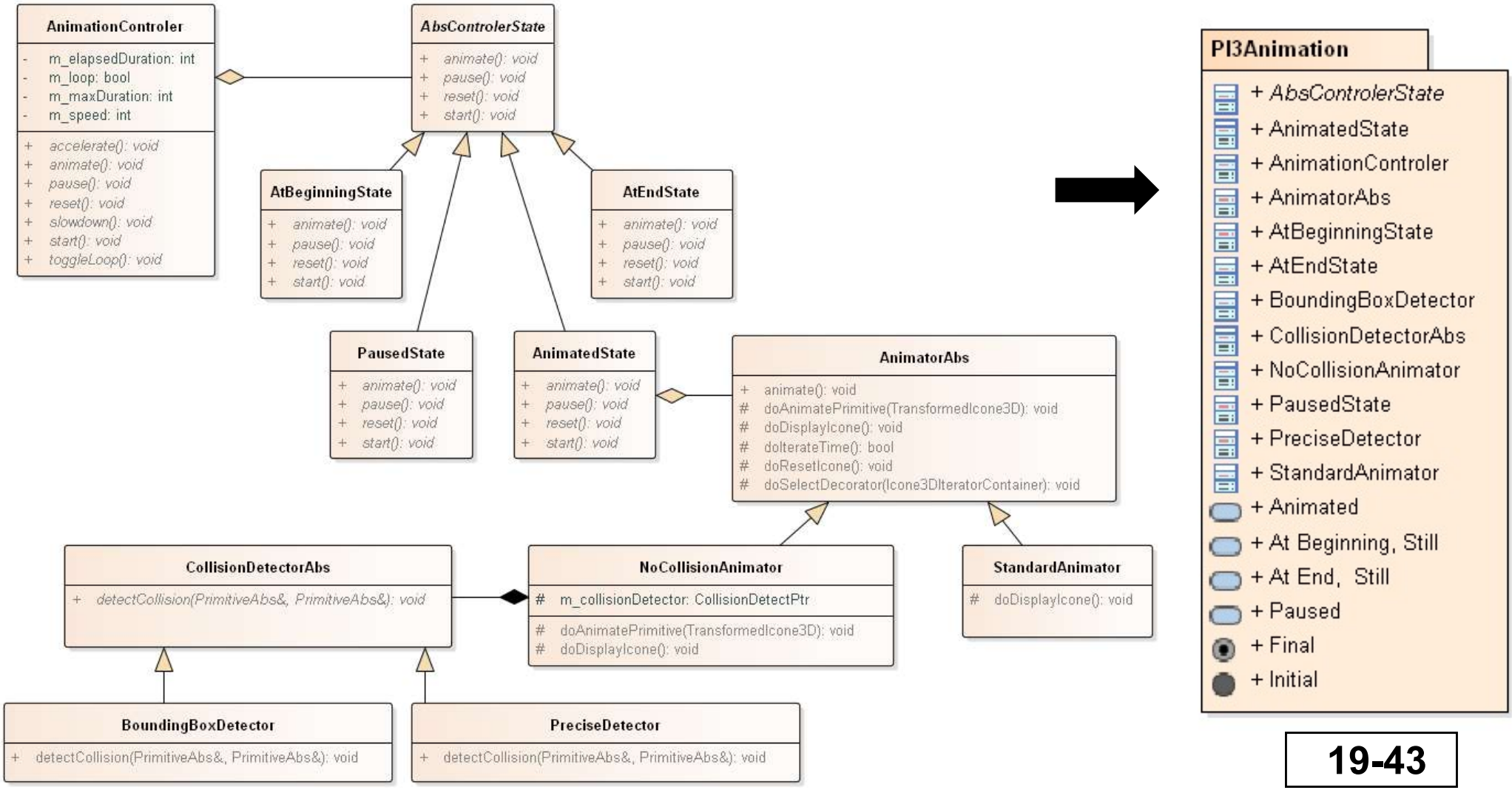
Patron Template Method

Patron Strategy

POLYTECHNIQUE MONTRÉAL

Le module d'animation des icônes

Considérer le module d'animation comme une abstraction avec laquelle on peut interagir sans connaitre les détails.



POLYTECHNIQUE MONTRÉAL

Patron **Façade**

Intention

Fournir une interface unifiée à un groupe d'interfaces d'un sous-système. Une façade définit une interface de haut niveau rendant l'utilisation d'un sous-système plus faciles.

Applicabilité

Lorsqu'on veut **fournir une interface simple à un sous-système complexe** (en particulier, l'utilisation des patrons de conception résulte fréquemment en un grand nombre de petites classes rendant le sous-système plus réutilisable, plus configurable, mais parfois plus difficile à utiliser).

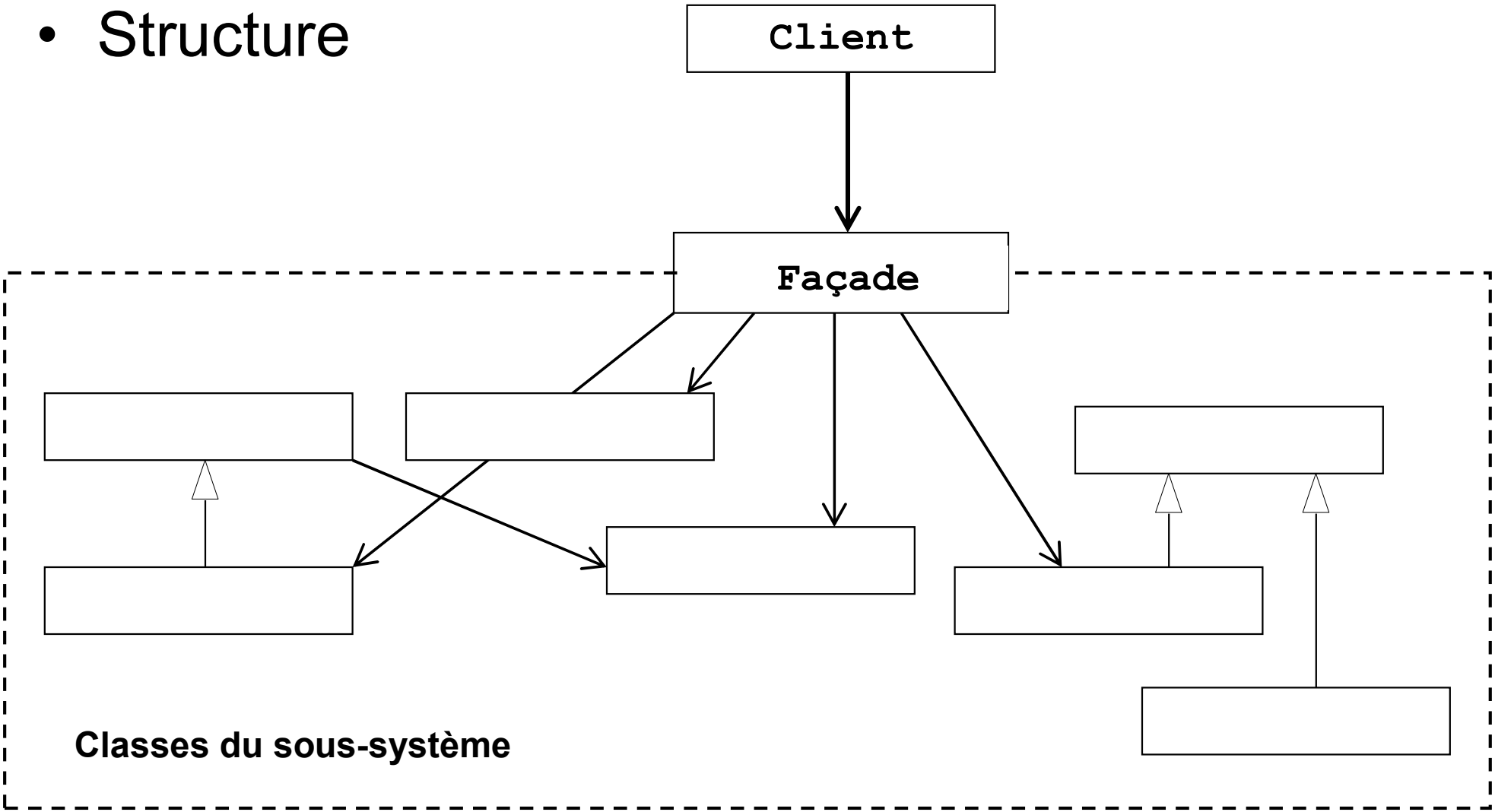
On veut **éviter un trop fort couplage** entre les **clients** et les **classes implantant une abstraction**.

On veut implanter une **architecture du système en plusieurs couches**. La classe Façade fournit un point d'entrée à chaque sous-système.

POLYTECHNIQUE MONTRÉAL

Patron **Façade**

- Structure



Patron **Façade**

- Conséquences
  - + Une classe façade **fournit une vue simplifiée du sous-système, suffisante** pour la plupart des clients.
  - + Le classe façade **découple le sous-système de ses clients et des autres sous-systèmes**, ce qui favorise la modularité et la portabilité.
  - + Lorsque les sous-systèmes communiquent entre eux uniquement par des façades, **les communications sont simplifiées**.
  - + Il reste **possible** pour les clients qui doivent configurer le sous-système **d'aller au-delà de la façade** si nécessaire.

Patron **Façade**

- Implémentation
  - Réduire le couplage entre les clients et le sous-système peut amener à faire de la classe façade une classe abstraite. Différentes sous-classes de la façade peuvent alors implémenter différentes versions du sous-système.
  - Classes publiques et privées. De même que pour les membres d'une classe, on peut parler de classes publiques et privées dans un sous-système.
    - Dans un langage comme C++, ceci peut être difficile à implémenter.
    - Dans un langage comme Java, la notion de classes privées et publiques dans un paquetage existe.

POLYTECHNIQUE MONTRÉAL

Patron **Façade**

Pour le module d'animation, il y a 2 responsabilités principales:

- Définir les paramètres d'animation,
- Contrôler le processus d'animation.

Il vaut mieux séparer ces responsabilités dans des interfaces distinctes.