POLYTECHNIQUE MONTRÉAL

CHAPITRE 18

Patrons de conception : État, Commande et Médiateur

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.

9 – Permettre à l'utilisateur de contrôler l'animations des icônes

- Problème de conception:
  - L'utilisateur doit pouvoir contrôler l'animation des icônes avec des fonctions telles que:
    - démarrer,
    - arrêter,
    - mettre en pause,
    - accélérer,
    - ralentir,
    - jouer en boucle,
    - etc.

9 – Permettre à l'utilisateur de contrôler l'animations des icônes

- L'algorithme d'animation lui-même n'est qu'une des nombreuses opérations qui doivent être contrôlées par l'utilisateur.
- Le comportement de plusieurs opérations dépend de l'état courant du processus d'animation:
  - Certaines opérations n'ont un sens que si l'animation est en cours,
  - D'autres n'ont un sens que si l'animation n'est pas démarrée.
- Le comportement de plusieurs opérations doit changer en bloc selon l'état de l'icône.

Patron État

Intention

Permettre à un objet de changer son comportement en fonction de son état. L'objet se comportera comme s'il avait changé de classe.

Applicabilité

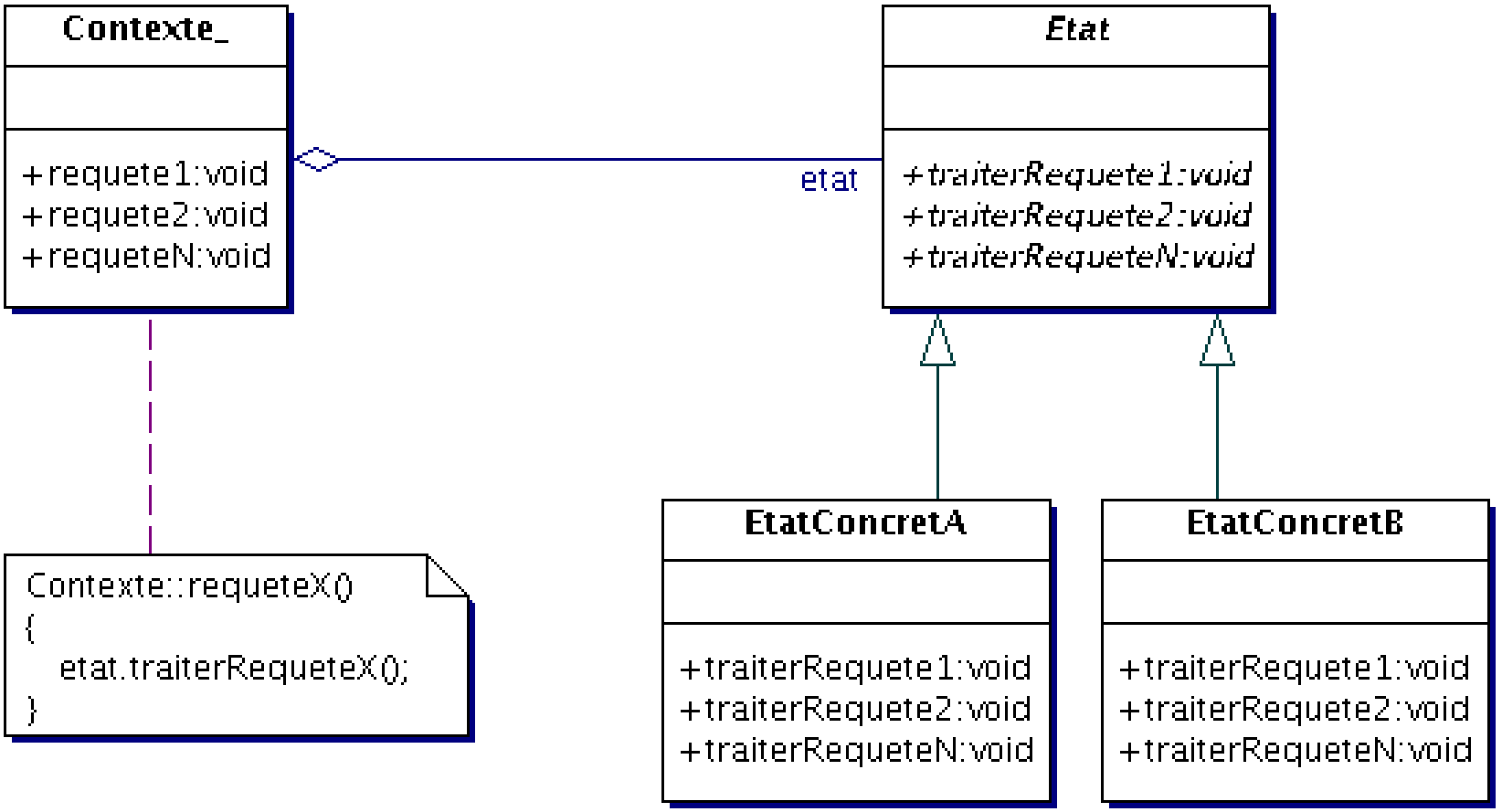
Le comportement d'un objet dépend de son état et cet état varie en cours d'exécution.

Les opérations contiennent de multiples énoncés conditionnels de type *switch-case* (souvent basés sur des type énumérés) qui permettent d'avoir un comportement différent selon l'état de l'objet.

La patron État permettra d'éliminer ce genre d'énoncés conditionnels et de les remplacer par une solution orientée objet.

POLYTECHNIQUE MONTRÉAL

Patron État



Patron État

Conséquences

+ Localisation des comportements propres aux états. Toutes les données et les comportements propres aux états se situent dans les sous-classes. Il est facile d'ajouter de nouveaux états.
+ Rend les transitions entre les états explicites.
+ Permet d'éviter les états inconsistants (le changement d'état est atomique).
+ Élimine les grands énoncés conditionnels.
± Les comportements reliés aux états sont distribués dans des sous-classes, ce qui augmente le nombres de classes et est moins compact. Toutefois, cette avenue est très utile s'il y a beaucoup d'états différents et est largement préférable à l'utilisation de multiples énoncés conditionnels.

Patron État

Implémentation

1. Qui définit les transitions d'état ?
   - Si les critères de transition sont fixes, la classe contexte peut s'occuper des transitions.
   - Sinon, c'est aux sous-classes représentant les états de définir les transitions. Il faut donc que la classe contexte ajoute à son interface une opération permettant aux sous-classes d'état de changer l'état. Cela sous-entend également que la classe de base `Etat` doit maintenir une référence à la classe `Contexte`.
2. Création et destruction des objets états, deux possibilités :
   - Tous les états sont construits une fois pour toutes et jamais détruits,
   - Les états sont créés uniquement au moment où ils sont nécessaires.
3. Héritage dynamique…
   - Changer le type d'un objet à l'exécution… n'est pas supporté par la plupart des langages OO.

POLYTECHNIQUE MONTRÉAL

Contrôler l'animations des icônes

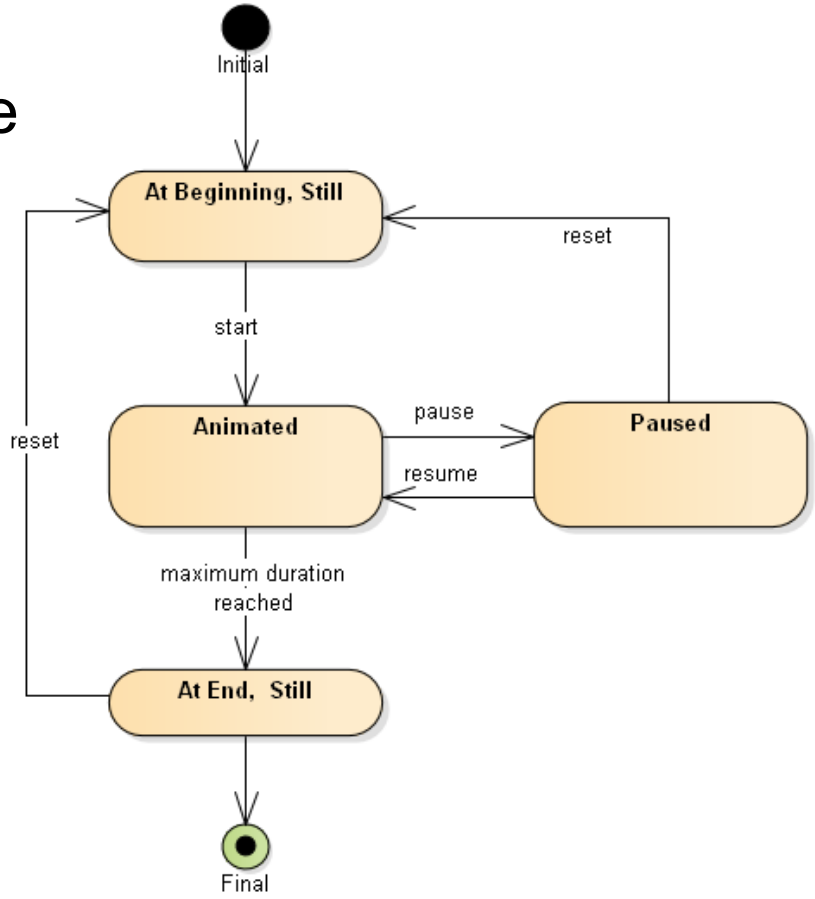
On veut confier le contrôle de l'animation à un objet spécialisé: le contrôleur d'animation.

Le contrôleur va changer d'état état selon les opérations invoquées.

On identifie la liste des états du contrôleur:

- At beginning,
- Animated,
- Paused,
- At end.

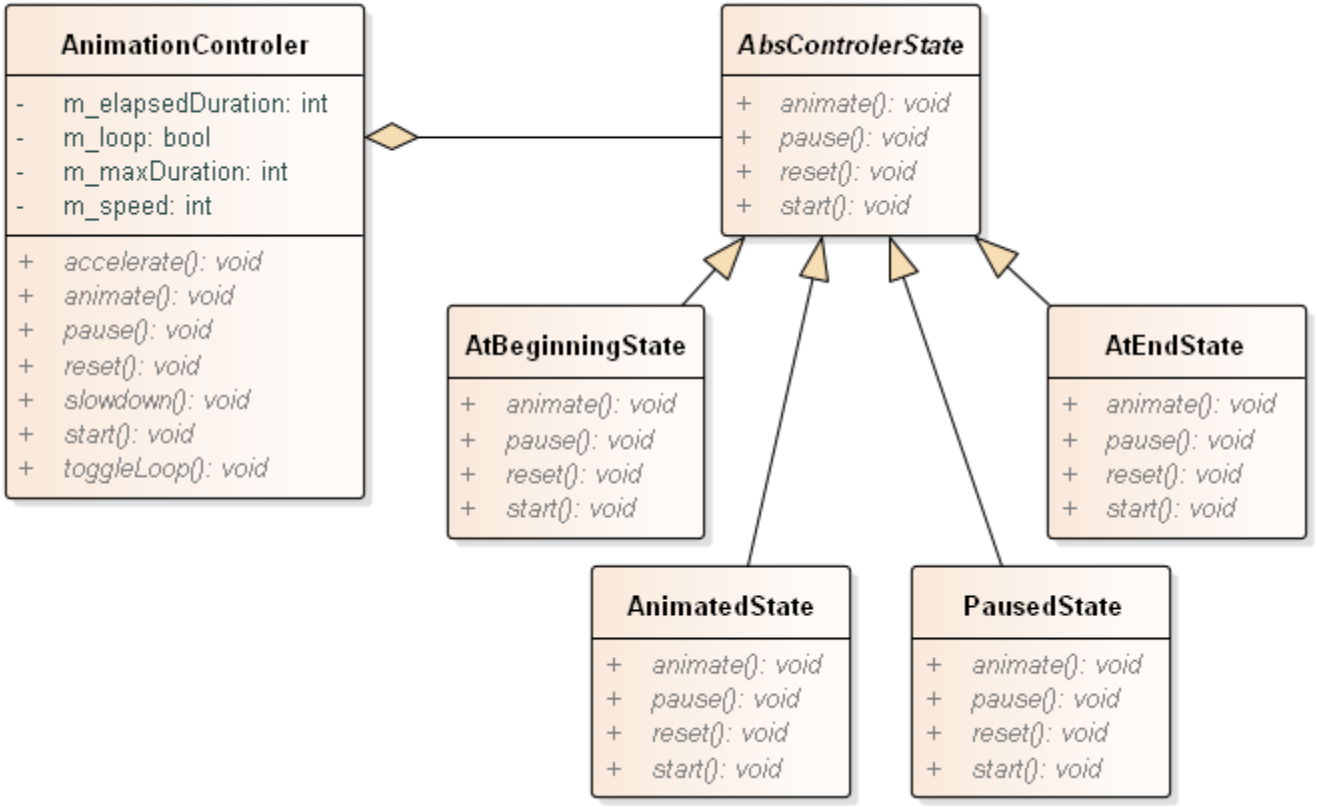
On identifie aussi les transitions possibles.



POLYTECHNIQUE MONTRÉAL

Contrôler l'animations des icônes

Le contrôleur agrège différentes classes d'état qui dérivent toutes d'une classe abstraite d'état.

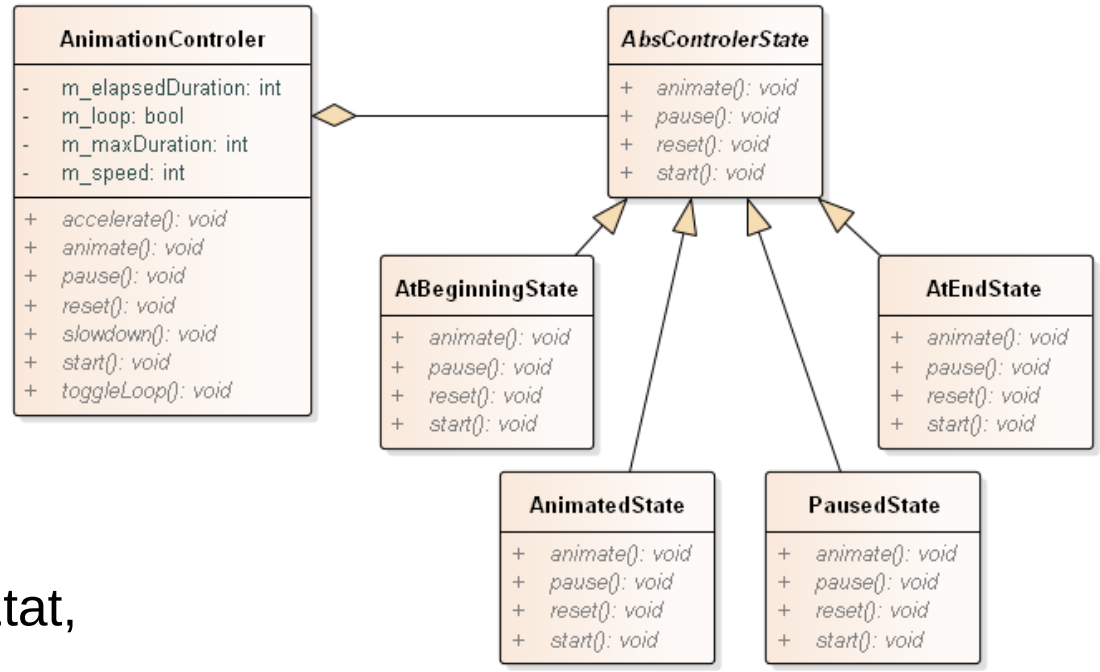


Contrôler l'animations des icônes

Le contrôleur agrège différentes classes d'état qui dérivent toutes d'une classe abstraite d'état.

Rôles:

- Contexte: AnimationControler
- État abstrait: AbsControlerÉtat
- États concrets: AtBeginningÉtat, AnimatedÉtat, PausedÉtat, AtEndÉtat



POLYTECHNIQUE MONTRÉAL

Patrons État vs Stratégie : quelle est la différence ?

Comparaison entre les deux patrons

- **La structure** des deux patrons est similaire.
  - Dans les deux cas, on délègue la responsabilité de l'implantation à une hiérarchie de classes distinctes (les stratégies et les états).
  - Dans le cas des états, **plusieurs fonctions varient** en fonction de l'état, alors que les stratégies encapsulent chacune **un algorithme**.
- **L'intention** derrière les deux patrons n'est pas la même.
  - Dans les stratégies, l'intention est d'encapsuler un algorithme et de possiblement combiner plusieurs stratégies pour utiliser simultanément et indépendamment différentes versions de plusieurs algorithmes.
  - Dans le cas des états, le comportement de **tout l'objet** doit être modifié lorsque l'état change. Le patron État permet de spécialiser le comportement en fonction de l'état de l'objet et les transitions d'états peuvent être déterminées par les états eux-mêmes.

10 – Éditer une icône et permettre à l'utilisateur de changer d'idée

- Problème de conception:
  - Parmi les nombreuses opérations d'édition qui peuvent être appliquées à une icône, plusieurs modifient l'icône ou certains éléments de l'icône.
  - Il faut permettre à l'utilisateur de tester certaines opérations et lui permettre de changer d'idée:
    - ➢ Fournir un mécanisme pour canceller une opération ou de réexécuter une opération cancellée (Undo/Redo).

Patron Commande

Intention

Encapsuler une requête dans un objet de façon à permettre de supporter facilement plusieurs types de requêtes, de définir des queues de requêtes et de permettre des opérations « annuler ».

Applicabilité

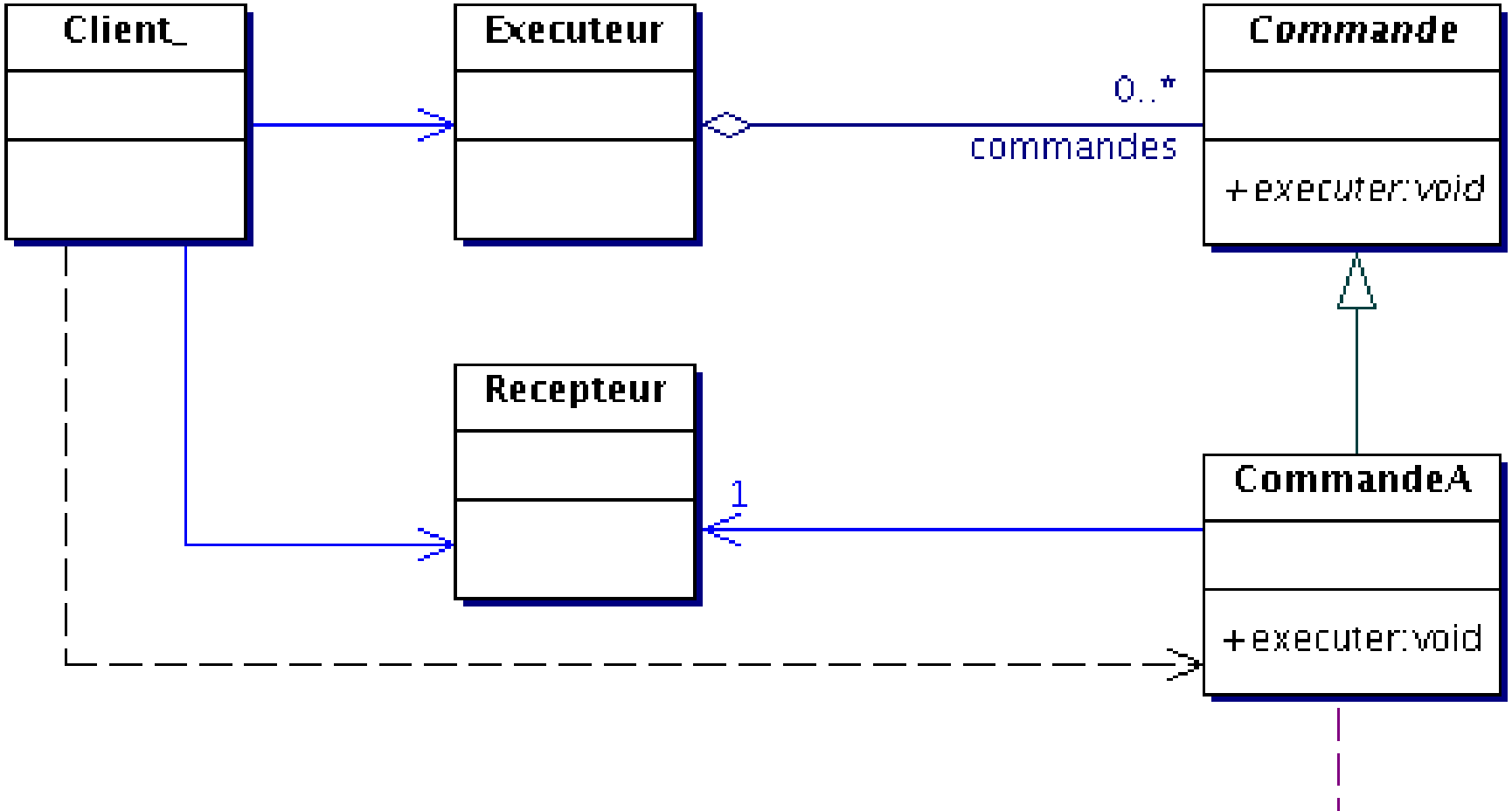
Paramétrer des objets avec une action à effectuer. Commande est une alternative orientée objet aux fonctions de rappel *(callback)*.

Spécifier une queue de requêtes qui seront exécutées ultérieurement.

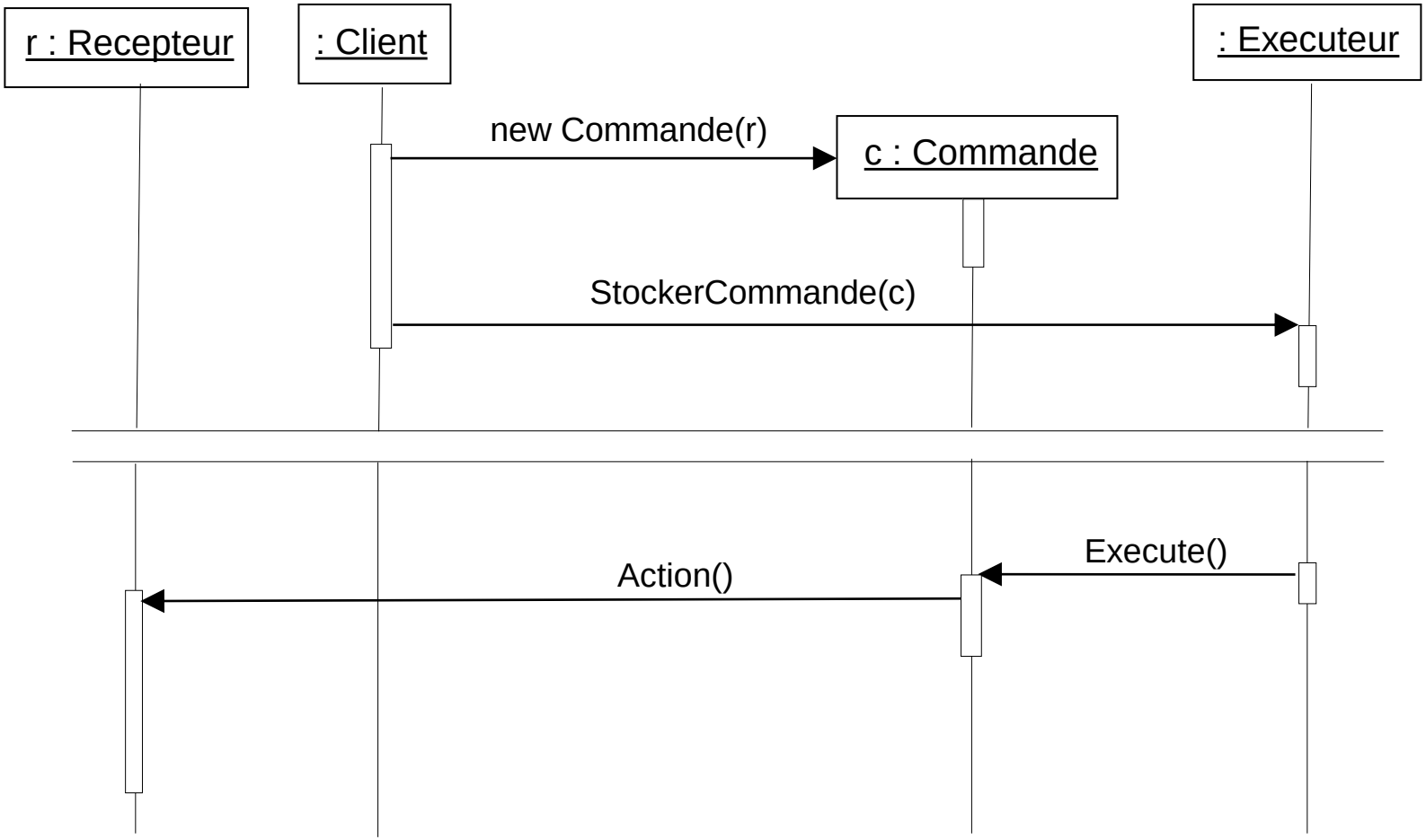
Support naturel pour les *undo*/*redo*.

POLYTECHNIQUE MONTRÉAL

Patron Commande



Patron **Commande** – Collaborations



POLYTECHNIQUE MONTRÉAL

Patron Commande

Conséquences

+ Découple l'objet qui invoque la requête de celui qui sait comment la satisfaire.
+ Les commandes sont encapsulées dans des objets qui peuvent être manipulées comme tout objet. L'utilisation d'objets amène également plus de flexibilité.
+ On peut facilement créer de nouvelles commandes.
+ Les commandes peuvent être assemblées en des commandes composites si nécessaires. Le patron Command peut être combiné au patron Composite pour représenter des commandes qui sont un ensemble d'autres commandes.

Patron Commande

Implantation

1. Quel est le niveau d'intelligence d'une commande ?
   - Une commande peut ne garder qu'un lien entre un receveur et une action à faire pour répondre à une requête.
   - À l'autre extrême, une commande pourrait implanter tout elle-même.
     - Cette solution est souhaitable si on veut que les commandes soient indépendantes des autres classes du système, quand aucun receveur n'existe ou quand le receveur est connu implicitement.
     - Il faut, bien entendu, que la commande possède toutes les données nécessaires pour exécuter elle-même la requête.
   - Le choix entre ces deux solutions dépend de la situation.

Patron Commande

Implantation

2. Support pour les *undo*/*redo*.
   - Le patron Command permet de gérer naturellement les opérations de type *undo*/*redo*. Pour cela, il faut que les commandes puissent être réversibles (possèdent une opération « `annuler()` »).
   - Pour ce faire, les commandes peuvent avoir besoin d'informations supplémentaires:
     - L'objet receveur.
     - Les arguments de l'opération qui a été appelée.
     - L'état original du receveur.
   - Pour supporter les *undo*, il faut conserver **un historique (une liste) des commandes** qui ont été faites afin de pouvoir invoquer leur opération d'annulation lorsque nécessaire.
   - Pour supporter les *redo*, il faut, de plus, **conserver un historique des commandes** qui ont été annulées afin de pouvoir les ré-exécuter lorsque le *redo* est invoqué.

Patron Commande

Implantation

3. Conservation d'état pour les *undo*/*redo*.
   - Il se peut que les commandes ne soient pas réversibles ou que la succession de plusieurs undo et redo en alternance finissent par faire diverger l'état de l'application par accumulation d'erreurs.
   - Il peut donc être nécessaire de conserver l'état de l'objet receveur (ainsi que de tout autre objet pouvant être affecté par l'exécution de la commande). Les opérations *undo*/*redo* deviennent alors des opérations de restauration et de sauvegarde d'états sur des objets.
     - Le patron **Memento** propose une solution à ce problème de sauvegarde et de restauration d'état qui peut être mise à profit dans le patron Command.

POLYTECHNIQUE MONTRÉAL

Patron Commande

Implantation

4. Des commandes simples et non annulables peuvent être implantées dans une classe template.

```
template <class Receveur>
class CommandeSimple : public Commande {
public:
   typedef void (Receveur::*Action)();

   CommandeSimple( Receveur* r, Action a )
      : receveur( r ), action( a ) {};

   virtual void executer()
         {(receveur->*action)();};
private:
   Action action;
   Receveur receveur;
};

class UnRcv{
public:
   void fct_( void );
};
```

```
UnRcvr* r = new UnRcvr;

[...]

Commande* c = new
    CommandeSimple<UnRcvr>
         ( r, &UnRcvr::fct_);

[...]

c->executer();
```

Patron Commande

Implantation

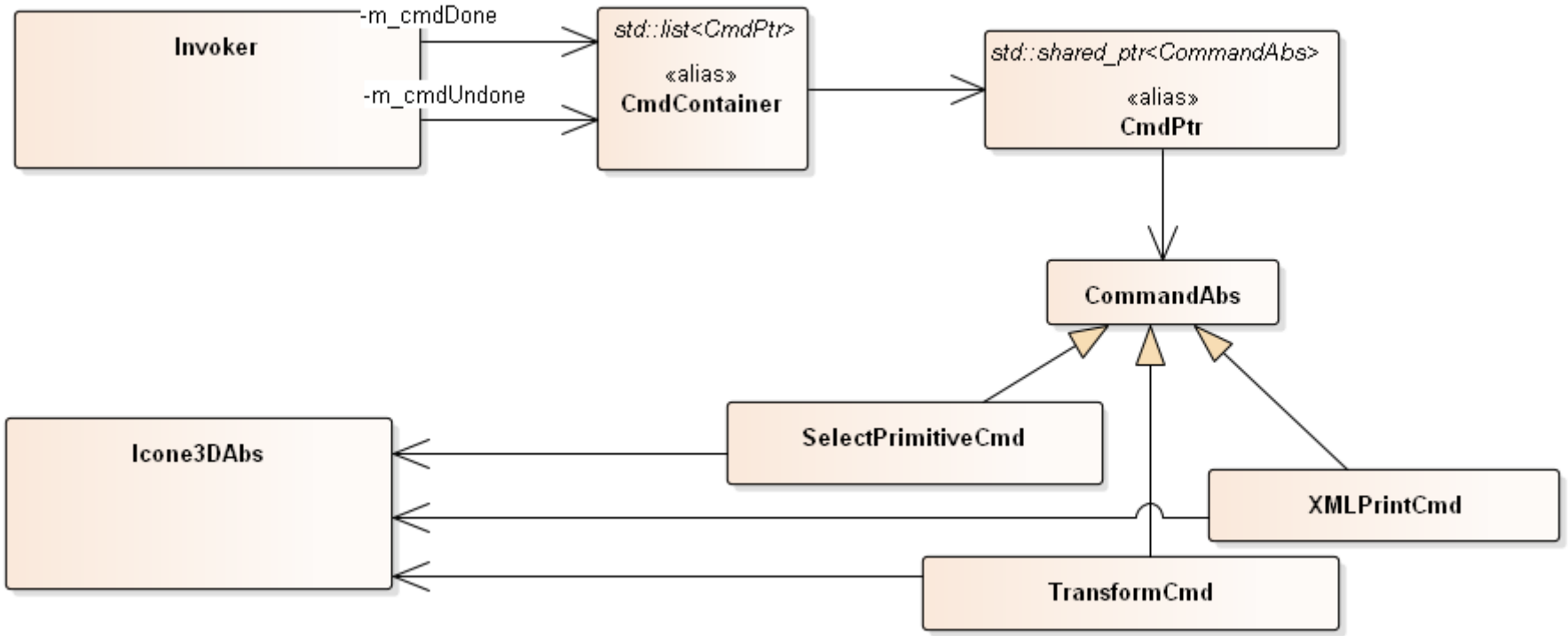
5. Commandes composites.

```
class CommandeComposite : public Commande {
public:
   CommandeComposite();
   virtual ~CommandeComposite();

   virtual void ajouter( Commande* c );
   virtual void enlever( Commande* c );

   virtual void executer();

private:
   list<CommandPtr> listeDeCommandes;
};

CommandeComposite::executer(){
   for ( list<Command*>::iterator it = listesDeCommandes.begin();
         it != listesDeCommandes.end(); ++it )
      (*it)->executer();
}
```

Éditer une icône et permettre à l'utilisateur de changer d'idée

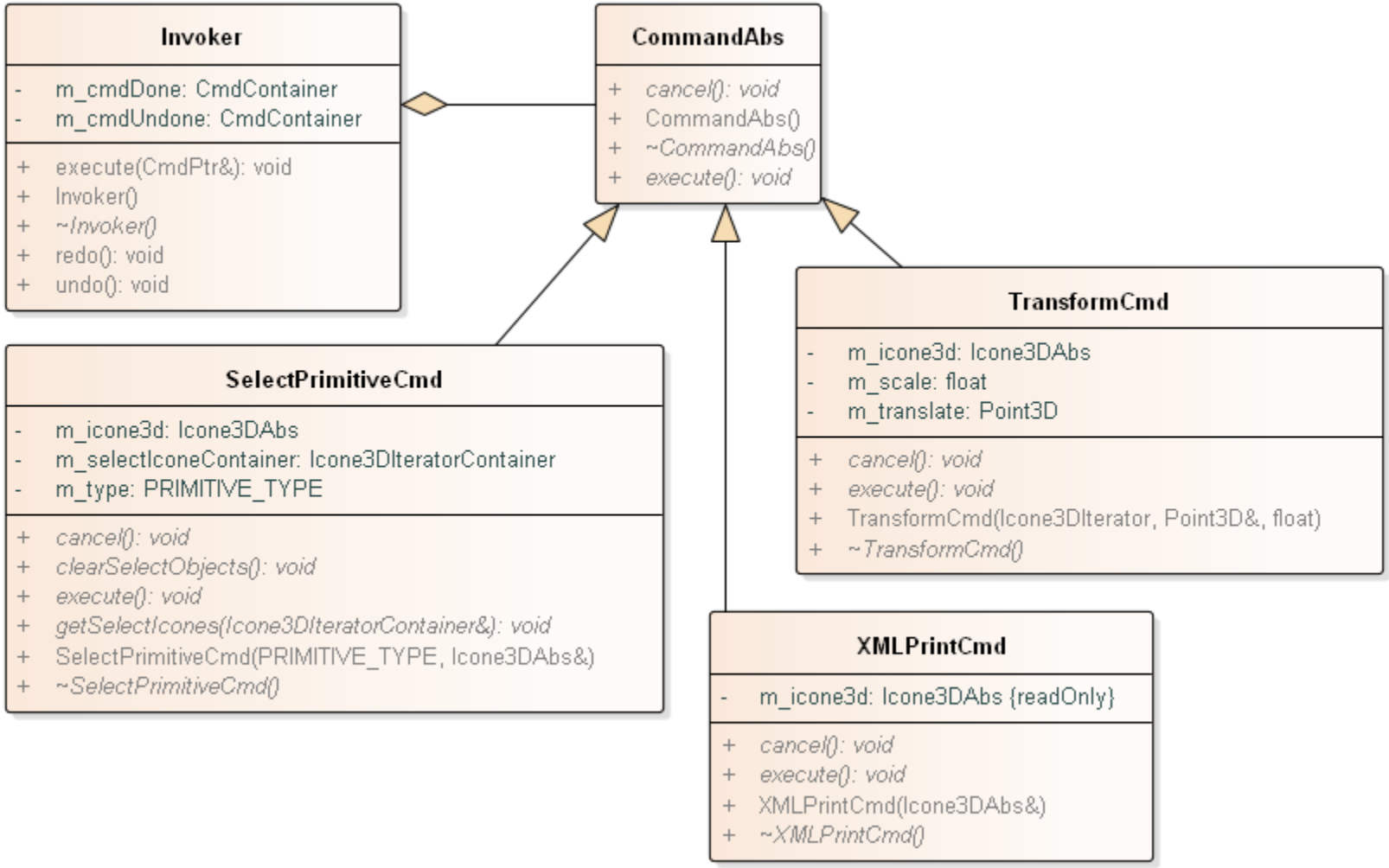
- Les opérations d'édition sont encapsulées dans des objets Commandes qui peuvent être exécutées et annulées
- Rôles:
  - Executeur: Invoker
  - Commade: CommandeAbs
  - CommandeConcrete: SelectPrimitiveCmd, TransformCmd, XMLPrintCmd
  - Recepteur: Icone3DAbs

Éditer une icône et permettre à l'utilisateur de changer d'idée



POLYTECHNIQUE MONTRÉAL

Éditer une icône et permettre à l'utilisateur de changer d'idée



Éditer une icône et permettre à l'utilisateur de changer d'idée

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

        virtual void cancel() =0;
        virtual void execute() =0;
};

class XMLPrintCmd : public CommandAbs
{
public:
        XMLPrintCmd(const Icone3DAbs& obj3d);
        virtual ~XMLPrintCmd() = default;

        virtual void cancel();
        virtual void execute();

private:
        const Icone3DAbs& m_obj3d;
};
```

Éditer une icône et permettre à l'utilisateur de changer d'idée

```
class TransformCmd : public CommandAbs
{

public:
        TransformCmd(Icone3DIterator iter, const Point3D& translat, float scal);
        virtual ~TransformCmd();

        virtual void cancel();
        virtual void execute();

private:
        Icone3DIterator m_iter;
        Point3D m_translate;
        float m_scale;
};
```

Éditer une icône et permettre à l'utilisateur de changer d'idée

```
void TransformCmd::execute()
{
   try
   {
         // Verifier si l'objet sur lequel pointe l'iterateur est une primitive
         PrimitiveAbs& primitive = dynamic_cast<PrimitiveAbs&> (*m_iter);

         // Construire un nouvel objet transforme en ajoutant un decorateur
         TransformedIcone3D icoTransformed(primitive.getParent(), primitive,
                                           m_translate, m_scale);

         // Remplacer l'objet sur lequel pointe l'iterateur par l'objet transforme
         primitive.getParent().replaceChild(m_objIter, icoTransformed);
   }

   catch (std::bad_cast& err)
   {
         std::cerr << "Erreur en executant une commande de transformation: "
                   << " l'objet a transformer n'est pas une primitive"
                   << std::endl << err.what() << std::endl;
         }
}
```

Éditer une icône et permettre à l'utilisateur de changer d'idée

```
void TransformCmd::cancel()
{
   try
   {
         // Verifier si l'objet sur lequel pointe m_iter est une icone transforme
         TransformedIcone3D& ico = dynamic_cast<TransformedIcone3D&> (*m_iter);

         // Recuperer la primitive transformee par le decorateur
         PrimitiveAbs& primitive = icoTransformed.getIcone3D();

         // Remplacer l'objet sur lequel pointe l'iterateur par l'objet transforme
         primitive.getParent().replaceChild(m_iter, primitive);
   }

   catch (std::bad_cast& err)
   {
         std::cerr << "Erreur en annulant une commande de transformation: "
                   << "l'iterateur ne pointe pas sur un objet transforme"
                   << std::endl << err.what() << std::endl;
   }
}
```

Éditer une icône et permettre à l'utilisateur de changer d'idée

```
using CmdPtr = std::shared_ptr<CommandAbs>;
using CmdContainer = std::list<CmdPtr>;

class Invoker
{
public:
        Invoker();
        virtual ~Invoker();

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

private:
        CmdContainer m_cmdDone;
        CmdContainer m_cmdUndone;
};
```

Éditer une icône et permettre à l'utilisateur de changer d'idée

```
void Invoker::execute(CmdPtr & cmd)
{
        cmd->execute();
        m_cmdDone.push_back(cmd);
}
void Invoker::undo()
{
        if(!m_cmdDone.empty())
        {
                CmdPtr cmd = m_cmdDone.back();
                cmd->cancel();
                m_cmdDone.pop_back();
                m_cmdUndone.push_back(cmd);
        }
}
void Invoker::redo()
{
        if (!m_cmdUndone.empty())
        {
                CmdPtr cmd = m_cmdUndone.back();
                cmd->execute();
                m_cmdUndone.pop_back();
                m_cmdDone.push_back(cmd);
        }
}
```

Patron Commande : exemple de undo/redo

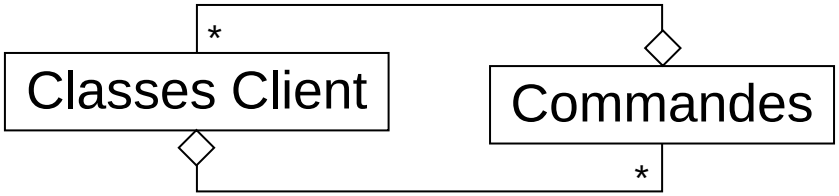
- Avec le patron Commande, le support du *undo*/*redo* devient une simple gestion de listes de commandes faites et défaites.
- Dans les cas où une opération n'est pas réversible, on peut mémoriser les états (patron Memento).
- Il faut rendre le tout robuste en prévoyant les cas où l'usager tente de faire un undo alors qu'aucune commande n'a été faite ou un redo sans avoir fait de undo préalable.
- Il faut également prévoir ce qui se passe lorsque les listes ne sont pas vides et que l'usager exécute une nouvelle commande. Doit-on vider la liste ?, l'état des listes doit-il être modifié ?, etc.
- Le patron peut également être utilisé simplement pour conserver un historique des commandes faites.

11 – Gestionnaire des commandes

- La classe *Invoker* joue un rôle particulier dans la conception du patron Commande:
  - Évite au client de devoir gérer directement les commandes.
- Sans la classe *Invoker*, plusieurs classes du client pourraient être responsables de créer et d'exécuter des commandes.

11 – Gestionnaire des commandes

- Une solution:

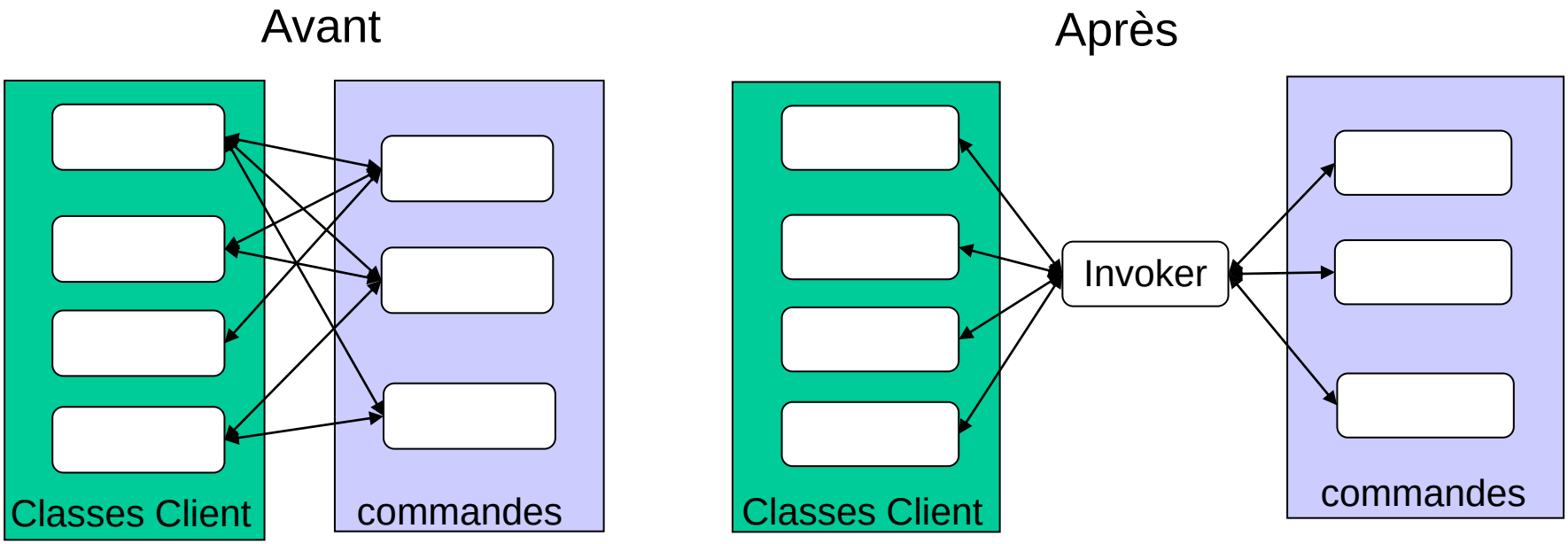


- Désavantages:
  - –Les références sont difficiles à changer de façon non invasive,
  - –Tous les objets doivent payer le prix de références à un ensemble d'objets.
  - –Enchevêtrement de références entre les objets Usagers et les objets Groupes.
    - Diagramme « spaghetti » !

POLYTECHNIQUE MONTRÉAL

11 – Gestionnaire des commandes

- Patron **Mediator**



POLYTECHNIQUE MONTRÉAL

Patron **Médiateur**

- Intention

  Définir un objet qui encapsule comment un ensemble d'objets interagissent afin de promouvoir un couplage faible et de laisser varier l'interaction entre les objets de façon indépendante.

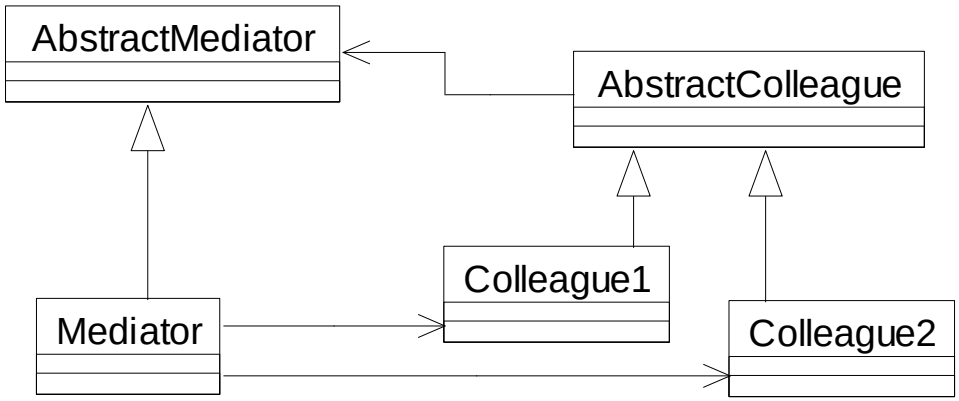
- Applicabilité

  Il existe un comportement de coopération entre des objets, qui ne peut être assigné à un objet individuel,

  Un ensemble d'objets communiquent entre eux de façon bien définie mais complexe,

  L'ordre des opérations peut changer à mesure que le système évolue.

- Structure



Patron **Médiateur**

- Conséquences
  - + Encapsule les communications
  - + Simplifie le protocole entre les objets
  - + Évite de forcer un ou plusieurs collègues à assumer les responsabilités de médiation
  - - Le Médiateur peut devenir complexe et monolithique
- Implantation
  - Utilisation de membres statiques plutôt que de classes séparées.
  - Omission de la classe abstraite **AbstractMediator**.
  - Nécessité de la classe abstraite **AbstractColleague** ?
  - Le Médiateur en Singleton.