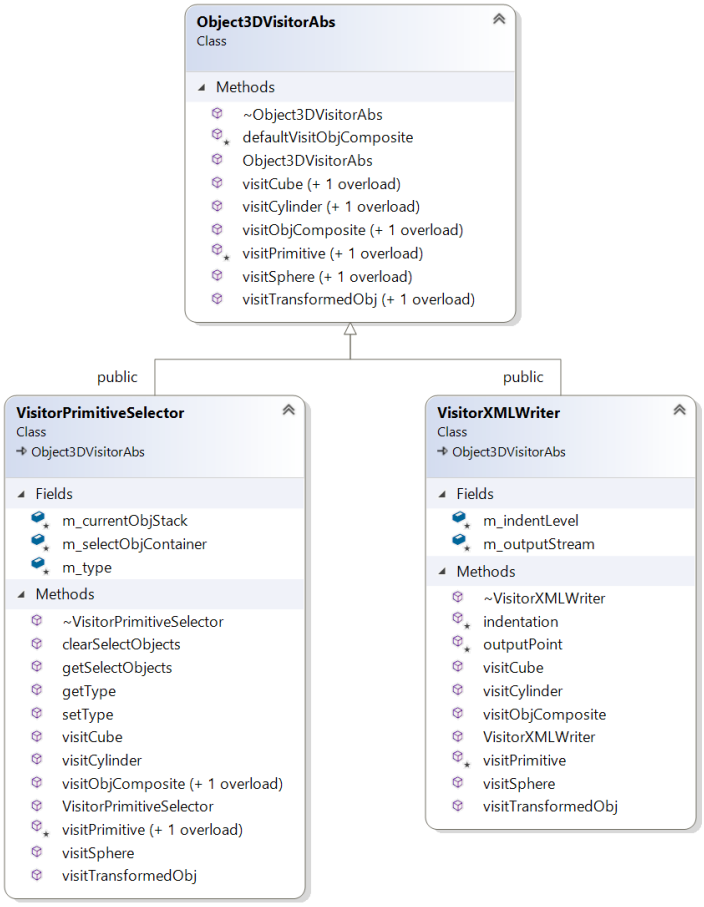
TP 5 – Questions théoriques

1 – Patron Visiteur

1) Identifiez l'intention du patron *Visiteur*.

Le patron *Visiteur* permet d'ajouter de nouvelles fonctionnalités aux classes avec un minimum de changements à ces dernières. En effet, les classes visiteurs définies extérieurement permettent de modifier la logique opérationnelle interne de plusieurs classes avec la possibilité d'adapter les méthodes pour chacun des types de ces dernières.

2) Diagramme de classes des instances du patron *Visiteur* :



3) Identifiez l'intention du patron *Template Method*.

Le patron *Template Method* suggère la création d'une classe abstraite servant de modèle de base duquel pourraient hériter un nombre de sous-classes. Cette classe abstraite force une implémentation de certaines méthodes mais permet aussi aux sous-classes de faire une implémentation d'autres méthodes, spécifiques à ces dernières.

4) Établissez la liste des classes et méthodes impliquées dans l'implémentation de chaque approche et discutez, en 250 mots ou moins, des avantages et inconvénients de chaque approche.

a. *Patron Visiteur* : Avec le patron *Visitor*, on se doit bien évidemment de créer une classe abstraite de Visiteur (dans notre cas *Objet3DVisitorAbs*) ainsi qu'une classe concrète de ce dernier (ici *VisitorXMLWriter*) qui implémenteront les méthodes *visit* pour chacune des classes qu'elles devront visiter. Ces méthodes contiendront la logique opérationnelle du visiteur en fonction de la classe visitée. De plus, il est important de définir les méthodes *accept* pour chacune des classes qui seront visitées (dans notre cas, toutes les classes dérivées de *Objet3Dabs*). On obtient ainsi un modèle où la hiérarchie de classes est préservée et la logique opérationnelle concernant toutes les classes se retrouve au même endroit. Par contre, lorsque la hiérarchie de classes doit être modifiée (ou de nouvelles classes doivent être ajoutées) tous les visiteurs doivent être modifiés et toutes les classes recompilées.
b. *Patron Template Method* : L'implémentation du patron *Template Method* implique la classe *Objet3DAbs* et toutes ses sous-classes avec la surcharge de l'opérateur '<<' et la méthode *toStream*. Ceci nous permet d'obtenir un logique opérationnelle adaptée en fonction de l'appelant, ce qui correspond à une inversion de contrôle, ainsi que de faire une certaine réutilisation de code. Par contre, pour appliquer ce patron, nous devons ajouter un niveau de hiérarchie de classes en faisant des sous-classements. Ceci représente un désavantage minime pour les possibilités qu'il offre.

5) Si en cours de conception vous constatiez que vous voudriez ajouter une nouvelle sous-classe dérivée de *PrimitiveAbs*, établissez la liste de toutes les classes qui doivent être modifiées à cause de l'utilisation du patron *Visitor*.

L'ajout d'une nouvelle classe dérivée de *PrimitiveAbs* nous obligerait d'ajouter de nouvelles fonctions de traitement (*visit*) dans la classe abstraite de visiteur (*Object3DVisitorAbs*) ainsi que dans toutes ses sous-classes (*VisitorPrimitiveSelector* et

VisitorXMLWriter). Ce nouvel ajout nous forcera aussi à procéder à une recompilation de toutes les classes implémentant les visiteurs.

6) **Selon vous, l'application des transformations aux primitives pourrait-elle être implémentée comme un visiteur ? Si oui, discutez en 250 mots ou moins, des avantages et inconvénients d'utiliser le patron visiteur pour cette fonction et sinon expliquez pourquoi le patron n'est pas applicable.**

L'application des transformations aux primitives pourrait, en effet, se faire à l'aide d'un visiteur qui viendrait modifier les attributs des primitives (centre et paramètres). Bien que cette approche éviterait une restructuration du point de vue de la hiérarchie (comme dans le cas du patron *Decorator*), elle rendrait l'ajout de nouvelles primitives difficile, comme mentionné ci-haut. Ceci rendrait donc l'ajout de nouveaux éléments au système plus fastidieux. Nous pensons donc qu'il serait mieux d'appliquer ces transformations à l'aide du patron décoateur.

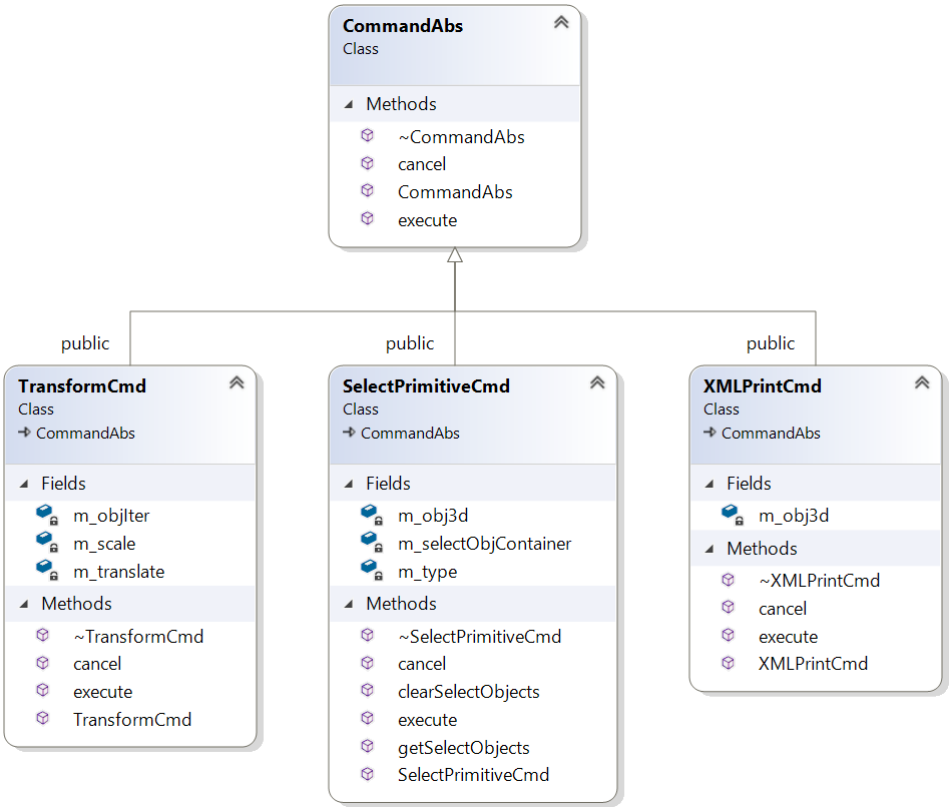
2-Patron Commande

1) **Identifiez les points suivants :**

   a. **L'intention du patron *Command*.**

   En somme, le patron *Command* a comme but de définir des objets qui vont représenter une action (ou une commande) et vont encapsuler toute l'information nécessaire pour l'effectuer. Ces derniers vont contenir, entre autres, la méthode qu'on veut exécuter, l'objet abritant cette méthode, ainsi que les paramètres nécessaires à son exécution. En plus de permettre l'assemblage de plusieurs actions sous une même commande, ce patron permet de la création d'une méthode *annuler* (*cancel* ou *undo*).

   b. **La structure des classes réelles qui participent au patron Command ainsi que leurs rôles :**



2) **Observez attentivement la classe Invoker qui permet de gérer la relation entre les commandes et les icônes 3D. En plus de participer au patron Commande, cette classe agit comme Mediator entre les commandes et les icônes. Puisque la classe Invoker agit comme Mediator, il pourrait sembler logique d'utiliser le patron Singleton lors de la définition de cette classe.**

   a. **Quel sont les intentions des patrons de conception Mediator et Singleton?**

      Patron Mediator : Le but du patron *Mediator* est de créer une classe intermédiaire (*médiateur*) qui va encapsuler l'ensemble de la communication entre plusieurs objets (qu'on appellera de *collègues*) assurant ainsi un couplage faible.

      Patron Singleton : Ce patron consiste tout simplement à concevoir une classe qu'il ne sera possible d'instancier qu'une seule fois tout en fournissant un point d'accès global pour cette dernière.

   b. **Quels sont les avantages de définir la classe *Invoker* comme *Mediator* ?**

      La définition de la classe *Invoker* comme *Mediator* permet d'y regrouper l'ensemble de la communication entre les commandes et les icônes. En effet, elle regroupe 3 méthodes (*execute, undo* et *redo*) qui permettent d'exécuter la commande désirée sur l'icône désirée. On obtient ainsi un couplage très faible entre ces classes et une communication indépendante des objets concernés.

   c. **Discuter en 250 mots ou moins, des avantages et des inconvénients de définir la classe *Invoker* comme *Singleton* ?**

      Le fait de définir la classe *Invoker* comme *Singleton* admet des avantages et des inconvénients. D'un côté, un avantage qui en découle est le fait qu'il n'est nécessaire d'avoir qu'une seule instance de la classe *Invoker* par utilisateur. Pour éviter de créer plusieurs instances de cette classe et ainsi réduire la pollution du code, il est avantageux de définir la classe *Invoker* comme *Singleton*. De plus, un autre avantage est la possibilité à cette classe d'être générée par sous-classification. En d'autres mots, il serait possible d'éviter l'implémentation d'une classe équivalente dont toutes les fonctions sont statiques. D'un autre côté, implanter *Invoker* comme *Singleton* rend équivalente une instance de cette classe à une variable globale ce qui est un inconvénient.

3) **Pour compléter la fonctionnalité de PolyIcone3D, il faudrait ajouter de nouvelles sous-classes de la classe *CommandAbs*. Selon vous, est-ce que d'autres classes doivent être modifiées pour ajouter les nouvelles commandes ? Justifiez votre réponse.**

Selon nous, l'ajout de nouvelles sous-classes de la classe *CommandAbs* ne nécessiterait pas la modification d'autres classes. En effet, toute l'interface d'exécution des commandes étant des sous-type de *CommandAbs* est déjà en place, ce qui nous permet de créer de nouvelles commandes qui seront automatiquement exécutables dans le système, à condition que ces dernières sont valides dans le contexte du logiciel. Autrement dit, le découplage obtenu grâce au patron *Command* fait en sorte qu'aucune autre modification n'est nécessaire. Il devient ainsi facile de rajouter de nouvelles commandes au besoin.