**\*\*\*** Pour une raison qui nous échappe, nos fichiers de code produisent deux erreurs dans le fichier TP5\_Test.cpp qui empêchent le programme de compiler. Cependant, il est possible de vérifier la bonne fonctionnalité de notre programme en remplaçant les lignes 314 à 316 ainsi que les lignes 454 à 456 du fichier TP5\_Test.cpp par les lignes suivantes:

CmdPtr selecCmdp = std::make\_shared<SelectPrimitiveCmd>(primType /\*remplacer primType par type pour la ligne 454\*/, m\_icone3D);

m\_invoker.execute(selecCmdp);

dynamic\_cast<SelectPrimitiveCmd\*>(&(\*selecCmdp))->getSelectObjects(m\_selectObjContainer);

Merci de votre compréhension. **\*\*\***

**Réponses aux questions**

**Patron Visiteur**

#1- L’intention de ce patron est de représenter une opération qui doit être appliquée sur les éléments d’une structure d’objets. Cette opération peut donc être définie sans qu’il ne soit nécessaire de modifier les classes des objets sur lesquels l’opération va agir.

#2- 

#3- L’intention du patron Template method est de définir le squelette de base d’un algorithme dans une opération, puis de laisser les sous-classes définir le reste des étapes et des paramètres.

#4- Les classes impliquées dans le patron Visiteur sont visibles dans le diagramme de classe présenté plus tôt. Les méthodes impliquées sont les méthodes visitObjComposite(const class Object3DComposite& comp), visitCube(class Cube& cub), visitCylinder(class Cylinder& cyl), visitSphere(class Sphere& sph), visitTransformedObj(class TransformedObj& tobj) et visitPrimitive(class PrimitiveAbs& prim) pour les deux classes (VisitorPrimitiveSelector et VisitorXMLWriter).

Les classes impliquées dans le patron Template Method sont la classe Obj3Dabs et la classe PrimitiveAbs. Les méthodes impliquées sont std ::ostream operator<<(std ::ostream o, const Object3DAbs& obj3d).

L’avantage de l’approche Visitor est qu’on ajoute la méthode qui affiche toutes les composantes d’un type de primitive donné, sans avoir à implémenter de nouvelles méthodes dans les classes des primitives. Le désavantage est qu’il faut implémenter une nouvelle classe complète, avec ses méthodes et attributs, afin de simplement implémenter l’opération qui sélectionne des primitives d’un type donné.

L’avantage de l’approche Template Method est qu’elle implémente la méthode d’affichage de différentes primitives ou ObjetComposite en ajoutant simplement la méthode Object3DAbs ::toStream, cependant cette approche nécessite d’implémenter une méthode différente pour toutes les primitives et ObjetComposite.

#5- Aucune classe ne serait modifiée. C’est là l’intérêt du patron Visitor; l’opération de sélectionner des primitives est indépendante des classes et donc, ajouter une nouvelle classe de primitive n’affecte en rien les autres classes.

#6- Oui, le patron Visiteur pourrait être utilisé afin d’appliquer des transformations aux primitives. Cela permettrait d’appliquer les transformations aux primitives via l’objet Visiteur, sans devoir ajouter des méthodes de transformations aux objets Primitives. Il faudrait donc créer une classe Visiteur, au même titre qu’une classe Decorator, avec des attributs et des méthodes de transformations.

**Patron Command**

#1-

* Encapsuler une requête dans un objet afin que des requêtes de types différents puissent être supportées par la méthode, que des queues de requêtes puissent être définies et que l’opération « annuler » soit supportée.

b)



#2-

* L’intention du patron Mediator est de définir un objet qui encapsule la façon dont un ensemble d’objets interagissent ensemble, dans le but de promouvoir un couplage faible et que l’interaction entre les objets varient de manière indépendante.

L’intention du patron Singleton est de s’assurer qu’une seule instance d’une certaine classe ne puisse être créée et qu’un point d’accès global à cette instance soit fourni.

* Le fait que la classe Invoker agisse comme Mediator permet d’encapsuler toutes les communications liées aux commandes dans une seule classe. Ainsi, on réduit le nombre de méthodes à implanter afin d’appeler les différentes commandes qui, pourtant, agissent différemment sur des composantes différentes. En s’assurant que tous les appels aux commandes et autres interactions avec les commandes soient gérés par la classe Invoker seulement, on réduit le couplage entre les ObjetsComposite et les commandes et ainsi, il est plus facile d’ajouter de nouvelles primitives ou commandes.
* Il serait utile de définir la classe Invoker comme un Singleton, puisqu’une seule instance de cette classe est nécessaire, on ne veut donc pas qu’il soit possible d’en créer d’autres, afin de limiter l’espace mémoire utilisée et la pollution général du code. De plus, on s’assure que cette instance ne puisse pas être modifiée par toutes les autres classes, comme une variable globale pourrait l’être, en définissant des méthodes spécifiques à la modification de ses attributs et en définissant les attributs comme privés. L’inconvénient est que l’implémentation de cette classe demande un peu plus de travail qu’une simple variable globale.

#3- En ajoutant d’autres classes de commandes, il ne serait pas nécessaire du tout de modifier d’autres classes, puisque ces nouvelles classes seraient appelées à être exécutées par la méthode *execute()* de la classe Invoker. Même la classe Invoker n’aurait pas à être changée. C’est là une des forces du patron Mediator.