TP 4 – Questions théoriques

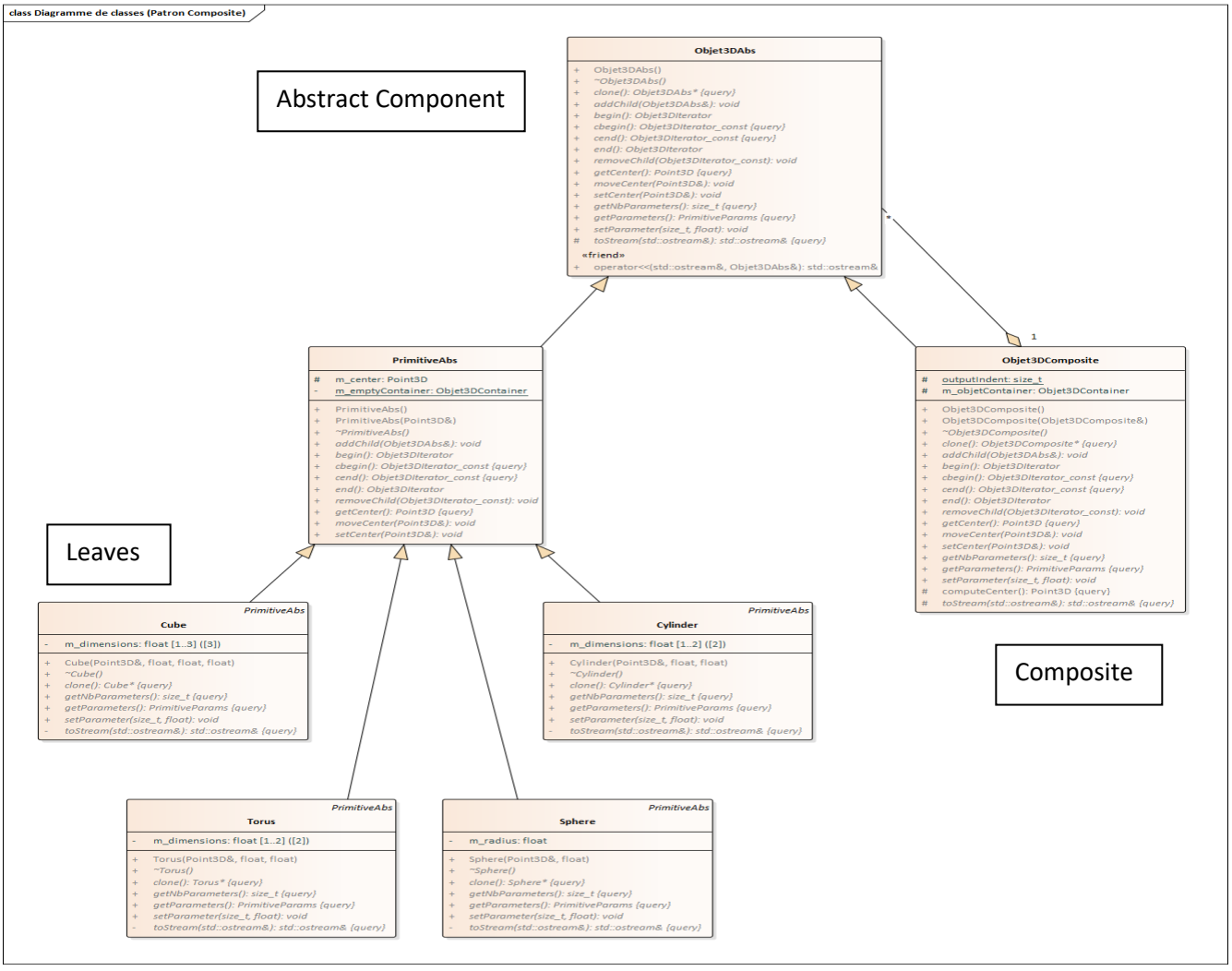
1 – Patron Composite

1) Identifiez les points suivants :

   a. L'intention du patron Composite.

      Le Patron Composite vise la création d'une structure de classes en arbre, permettant ainsi des structures composites qui seront traités de la même manière que les structures non-composites (feuilles). Ceci permet donc d'uniformiser l'utilisation de ces structures pour simplifier l'expérience de l'utilisateur.

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



2) Identifiez la ou les abstractions présentent dans la conception du TP4, et pour chacune, identifiez les responsabilités spécifiques qui lui ont été assignées.

1. *Objet3DAbs* : représente une interface qui permet l'utilisation uniforme de tous les objets, composites ou non. Elle définit donc toutes les méthodes utilisées par les objets et c'est à travers elle que l'utilisateur les manipulera.
2. *PrimitiveAbs* : représente l'interface d'utilisation d'objets singuliers qui permet d'obtenir et manipuler les paramètres et le point central des primitives. Cette abstraction permet d'implémenter les méthodes de gestion du point central, tandis que celles servant aux manipulations de paramètres seront spécifiques à chacune des primitives concrètes.

2-Patron Decorator

1) Identifiez les points suivants :

a. L'intention du patron Décorateur.

Le patron décorateur permet d'ajouter des attributs ou des méthodes (et donc d'étendre les fonctionnalités) à des classes déjà existantes, et ce sans modifier la classe de base. Cet ajout dynamique peut tout aussi bien être retiré, ce qui augmente grandement la flexibilité des classes et évite la création d'un réseau d'héritage complexe.

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

Voir page suivante!

2) Identifiez les responsabilités des classes primitives qui sont réinterprétées lorsque le Décorateur est utilisé.

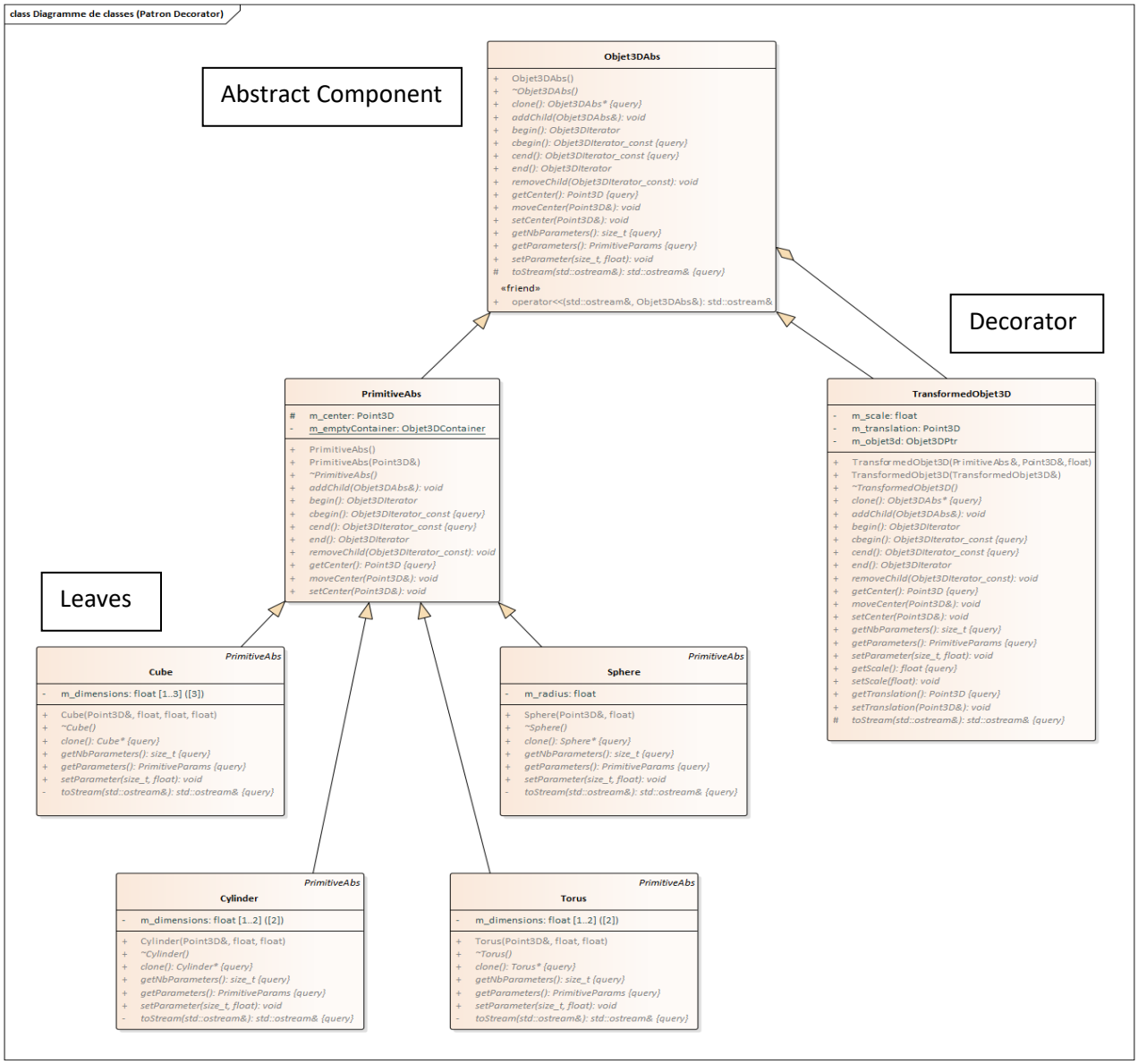
L'utilisation du décorateur dans cet exemple implémente l'ajout de deux paramètres de transformation dans les classes primitives. Ces dernières auront maintenant la possibilité d'être déplacées (translation) et agrandies (mises à l'échelle). Les classes primitives pourront donc effectuer les mêmes actions que précédemment, en plus de retourner le nouveau centre après translation et les nouveaux paramètres après mise à l'échelle.

3) Selon vous, pourquoi dans la conception actuelle, un Décorateur s'applique aux primitives (classe *PrimitiveAbs*) et non à tous les objets 3D (*Objet3Dabs*) ? Serait-il possible d'appliquer le Décorateur à tous les objets et quelle en serait les conséquences ?

Dans la conception actuelle, le Décorateur se charge d'ajouter des fonctionnalités à un objet singulier (dans ce cas-là les, aux primitives). Par contre, grâce au patron composite, la classe *Objet3Dabs* nous permet de manipuler tous les objets à travers une même interface. Pour cette

raison, il serait possible d'appliquer le décorateur à tous les objets 3D. Par contre, comme l'implémentation de la méthode *getParameters()* de la classe *Objet3DComposite* retourne un conteneur vide (en raison de la complexité de la structure des données de retour et puisque son implémentation n'était pas demandée), le changement apporté par le décorateur pour cette méthode passera inaperçu.

Diagramme de la question 1b du patron decorator.



3-Conteneurs et Patron Iterator

1) Identifiez les points suivants :

    a. L'intention du patron Iterator.

        Le patron iterator fournit une méthode d'accès séquentielle aux éléments d'un objet agrégat que ce soit un vecteur, une liste, etc. Il prend alors la responsabilité de garder l'état d'itération sans exposer sa structure interne.

    b. La classe de conteneur de la STL utilisée pour stocker les enfants dans la classe Composite et les classes des Iterators utilisés dans la conception qui vous a été fournie.

        La classe de conteneur de la STL utilisée pour stocker les enfants dans la classe Composite est celle de *Vector*. De plus, les classes des Iterators *Objet3DBaseIterator* et Objet3DBaseIterator_const (qui représentent respectivement les classe *iterator* et *const_iterator* du vecteur) nous ont été fournies.

2) Expliquez le rôle de l'attribut statique m_emptyContainer défini dans la classe PrimitiveAbs. Expliquez pourquoi, selon vous, cet attribut est déclaré comme un attribut statique et privé.

    Le rôle de m_emptyContainer est de permettre l'implémentation des méthodes *begin()*, *cbegin()*, *cend()* et *end()* hébergées dans la classe abstraite *Objet3DAbs* et dont dérive la classe *PrimitiveAbs*. Ces méthodes doivent obligatoirement être implémentées au sein de cette dernière classe. Cependant, l'attribut m_emptyContainer a pour bout de faire échouer silencieusement ces méthodes en retournant un objet Iterator valide d'un conteneur vide. Cet attribut est déclaré privé puisqu'il serait inutile, voire un ennui, de permettre aux classes dérivées de *PrimitiveAbs* d'hériter d'un attribut qui ne sert qu'à échouer les méthodes héritées de la classe *Objet3DAbs* non nécessaires. De plus, puisque chaque primitive n'a pas besoin de container et donc n'a pas besoin d'itérateur, cet attribut est déclaré statique pour n'en avoir qu'une seule copie qui sera utilisé par toutes les instances dérivées de *PrimitiveAbs*.

3) Quelles seraient les conséquences sur l'ensemble du code si vous décidiez de changer la classe de conteneur utilisée pour stocker les enfants dans la classe Composite? On vous demande de faire ce changement et d'indiquer toutes les modifications qui doivent être faites à l'ensemble du code à la suite de ce changement. Reliez la liste des changements à effectuer à la notion d'encapsulation mise de l'avant par la programmation orientée-objet. À votre avis, la conception proposée dans le TP4 respecte-t-elle le principe d'encapsulation ?

    Pour les conteneurs « *list* » et « *deque* », aucun changement ne sera apporté puisque ceux-ci implémentent les mêmes méthodes utilisées dans le code. Cependant, pour des conteneurs comme « *map* », « *multimap* », « *set* » et « *multiset* », parmi plein d'autres, il faudrait modifier les méthodes qui permettent d'implémenter diverses fonctionnalités nécessaires au sein du code

telles que « *erase()* » et « *push_back()* ». Autrement dit, il faudrait implémenter les méthodes spécifiques à chacun des conteneurs pour manipuler les données de la manière désirée. De plus, un changement très important à effectuer serait la modification du type des itérateurs utilisés pour parcourir les conteneurs. Cependant, vu la structure du code, une seule modification sera nécessaire pour Objet3DContainer. La liste des changements étant minimes, il est possible de dire que la conception du code respecte le principe de l'encapsulation.

Modification au code :

- *using Objet3DContainer = std::list<Objet3DPtr>* (au lieu du *vector*)

4) Les classes dérivées Objet3DIterator et Objet3DIterator_const surchargent les opérateur « * » et « -> ». Cette décision de conception a des avantages et des inconvénients. Identifiez un avantage et un inconvénient de cette décision.

Un avantage provenant de la surcharge des opérateurs « * » et « -> » est celui de simplifier l'accès à l'objet sur lequel pointe l'itérateur. En effet, comme le conteneur abrite des pointeurs intelligents (objets agrégats), la surcharge de l'opérateur « * » permet de déréférencer le pointeur vers l'objet 3D et simplifier l'accès à ce dernier. Ensuite, pour la même raison, la surcharge de l'opérateur « -> » permet de simplifier l'accès aux méthodes de l'objet.

Un inconvénient est celui d'introduire des confusions si les surcharges ne sont pas conçues de façon optimale. De plus, pour la même raison, elles peuvent augmenter la probabilité de bogues potentiels.