Patron Composite

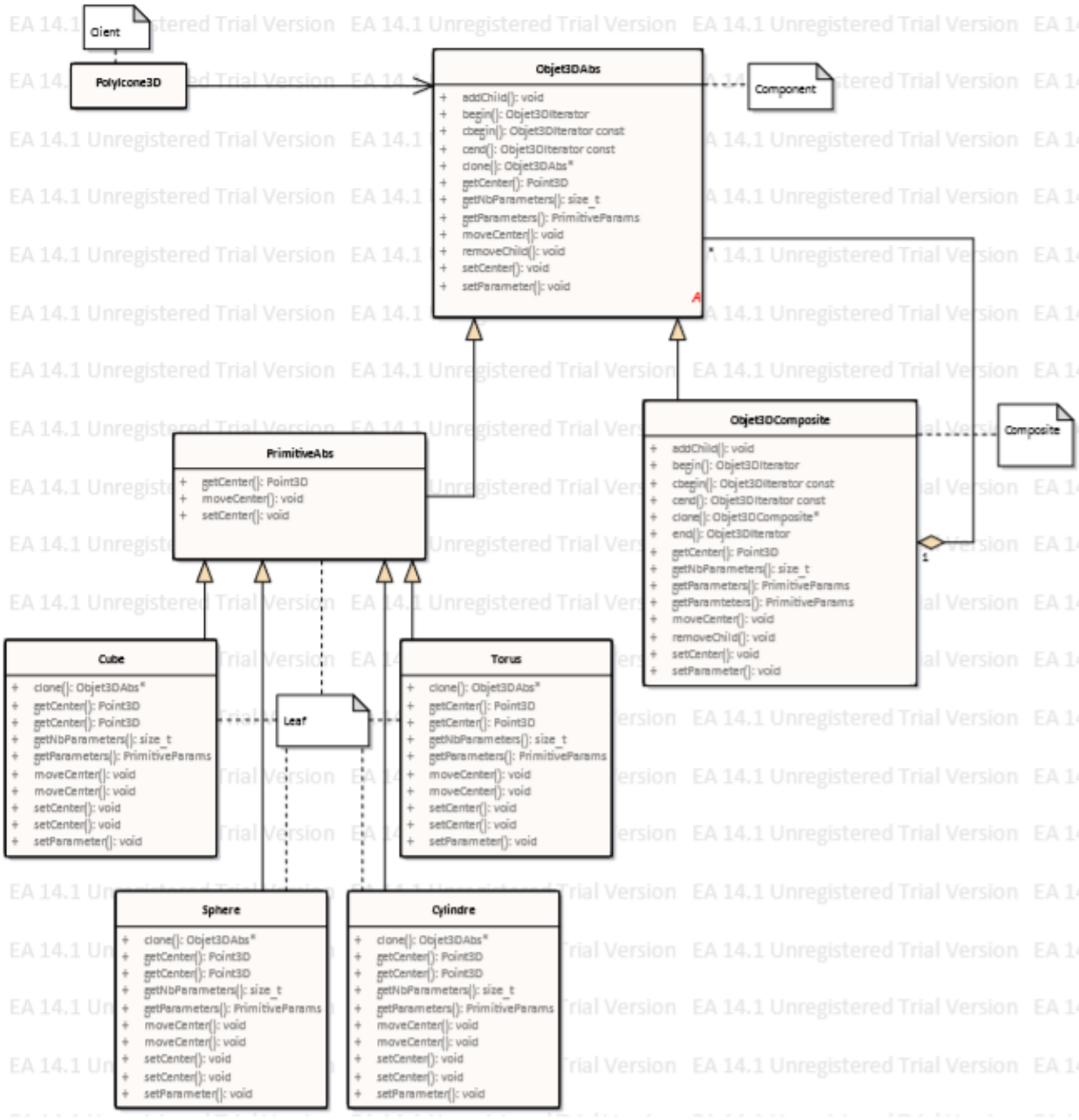
Question 1

a) L'intention du patron Composite

Selon les notes de cours, l'intention d'un patron composite est de « traiter les objets individuels et les objets multiples, composés récursivement, de façon uniforme ». En d'autres mots, un patron composite permet de gérer des objets ayant des fonctions similaires de la même manière en ignorant les différences entre les classes. Ce patron augmente l'uniformité et la flexibilité du code.

b) Structure du patron Composite dans PolyIcone3D

Dans PolyIcone3D, les classes Objet3DAbs, Objet3DComposite ainsi que PrimitiveAbs et ses dérivés forment le patron Composite. Le rôle du *component* est associé à la classe Objet3Dabs étant donné que c'est la classe abstraite dont toutes les autres classes du patron héritent. C'est l'interface uniforme. Le rôle de *leaf* est associé à la classe abstraite PrimitiveAbs et à ses dérivées. Quant au rôle du *composite*, il est associé à la classe Objet3DComposite, car c'est celle-ci qui peut être composée d'un ou de plusieurs objets de classe Objet3DAbs. Voir le diagramme sur la prochaine page.


Client
PolyIcone3D
Objet3DAbs
+ addChild(): void
+ begin(): Objet3DIterator
+ cbegin(): Objet3DIterator const
+ cend(): Objet3DIterator const
+ clone(): Objet3DAbs*
+ getCenter(): Point3D
+ getNbParameters(): size_t
+ getParameters(): PrimitiveParams
+ moveCenter(): void
+ removeChild(): void
+ setCenter(): void
+ setParameter(): void
Component
PrimitiveAbs
+ getCenter(): Point3D
+ moveCenter(): void
+ setCenter(): void
Objet3DComposite
+ addChild(): void
+ begin(): Objet3DIterator
+ cbegin(): Objet3DIterator const
+ cend(): Objet3DIterator const
+ clone(): Objet3DComposite*
+ end(): Objet3DIterator
+ getCenter(): Point3D
+ getNbParameters(): size_t
+ getParameters(): PrimitiveParams
+ getParamteters(): PrimitiveParams
+ moveCenter(): void
+ removeChild(): void
+ setCenter(): void
+ setParameter(): void
Composite
Cube
+ clone(): Objet3DAbs*
+ getCenter(): Point3D
+ getCenter(): Point3D
+ getNbParameters(): size_t
+ getParameters(): PrimitiveParams
+ moveCenter(): void
+ moveCenter(): void
+ setCenter(): void
+ setCenter(): void
+ setParameter(): void
Leaf
Torus
+ clone(): Objet3DAbs*
+ getCenter(): Point3D
+ getCenter(): Point3D
+ getNbParameters(): size_t
+ getParameters(): PrimitiveParams
+ moveCenter(): void
+ moveCenter(): void
+ setCenter(): void
+ setCenter(): void
+ setParameter(): void
Sphere
+ clone(): Objet3DAbs*
+ getCenter(): Point3D
+ getCenter(): Point3D
+ getNbParameters(): size_t
+ getParameters(): PrimitiveParams
+ moveCenter(): void
+ moveCenter(): void
+ setCenter(): void
+ setCenter(): void
+ setParameter(): void
Cylindre
+ clone(): Objet3DAbs*
+ getCenter(): Point3D
+ getCenter(): Point3D
+ getNbParameters(): size_t
+ getParameters(): PrimitiveParams
+ moveCenter(): void
+ moveCenter(): void
+ setCenter(): void
+ setCenter(): void
+ setParameter(): void

Question 2

Identification des abstractions

Dans cette structure, il y a deux abstractions, soit la classe Objet3DAbs et la classe PrimitiveAbs. La responsabilité principale d'Objet3DAbs est de regrouper et d'uniformiser les fonctionnalités des classes Objet3DComposite des classes qui héritent de PrimitiveAbs. Cette abstraction regroupe des fonctions telles que addChild() et cbegin() qui permettent la gestion des enfants dans la classe Objet3DComposite, des fonctions telles que getParameters() et getNbParameters() qui permettent la gestion des paramètres d'une primitive (classe qui dérive de PrimitiveAbs) et des fonctions telles que clone(), getCenter() et setCenter() qui sont des fonctionnalités communes aux classes Objet3DComposite, PrimitiveAbs et aux dérivées de PrimitiveAbs. La deuxième abstraction, soit PrimitiveAbs, sert à regrouper les différentes *leaves* du patron composite. Elle contient les fonctions pour effectuer une transformation sur une primitive et elle fait en sorte que les fonctions de gestion des enfants qu'elle hérite d'Objet3DAbs ne fassent rien pour ses dérivées.

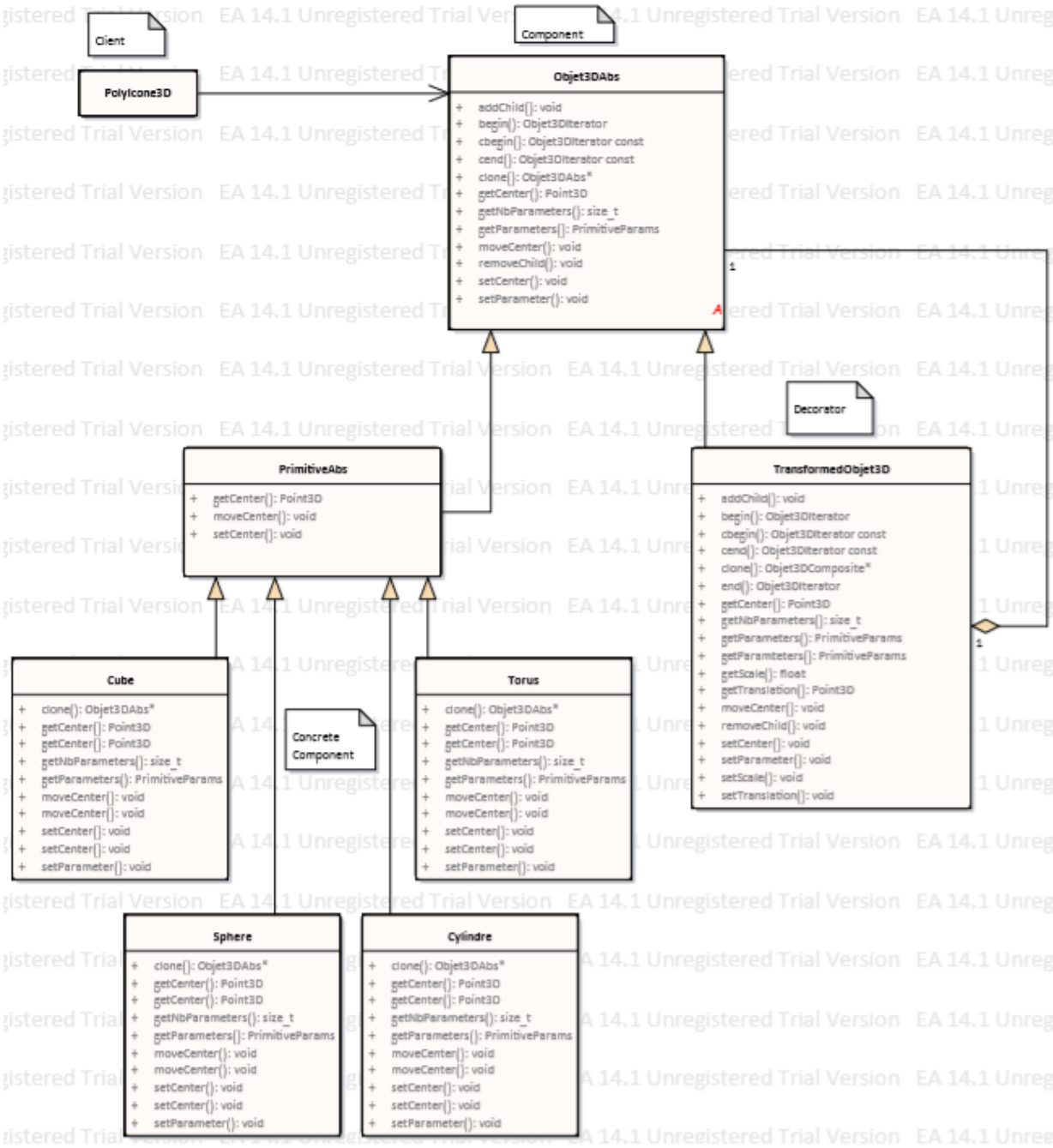
Patron Decorator

Question 1

a) L'intention du patron Decorator

Le patron Decorator donne des responsabilités a la classe TransformedObjet3D, cela donne plus de flexibilité et de fonctionnalités à la classe. Il a aussi pour intention de baisser la complexité des objets composites en se donnant des responsabilité précises.

b) Structure du patron Decorator dans PolyIcone3D



Question 2

Identification des responsabilités des classes primitives

Ces responsabilités sont celles des transformations géométriques sur les objets 3D. Ceux-ci sont la translation dans l'espace 3D et la modification de l'échelle des objets.

Question 3

Pourquoi le Decorator s'applique aux primitives et non à tout objet 3D

Parce ce que selon la conception actuelle, seul les PrimitivesAbs sont suceptibles d'être modifiées par une transformation. Il serait sans doute possible d'appliquer le patron decorator à tous les objets Objet3DAbs, mais nous devrions trouver une façon d'appliquer les transformations sur des objets de type Objet3DComposite sans que la forme de l'objet soit modifiée, car une modification de l'échelle sur toutes les primitives pourrait occasionner des chevauchements entre les formes.

Conteneur et Patron Iterator

Question 1

a) L'intention du patron Iterator

Le patron Iterator a pour intention de rassembler tout le code qui concerne l'itération des objets en un seul patron pour ne pas nuire à la lisibilité et compréhensibilité. Il a aussi comme but de donner la possibilité de faire plusieurs traverses simultanées et de généraliser pour plusieurs types d'objets.

b) Identifier la classe de conteneur et les classes des Iterators

La classe du conteneur est std::vector<Objet3DPtr> et la classe des iterators est Objet3DContainer::const_iterator. Donc la classe des iterateurs est la classe de base d'un itérateur pour un vector.

Question 2

Expliquez le rôle de l'attribut statique m_emptyContainer

C'est pour pouvoir redéfinir les méthodes des itérateurs tels que begin() sans que les méthodes retournent des erreurs. il est mis prive pour pas que l'une classe change son contenu et statique puisqu'il est vide peu importe la primitive.

Question 3

Conséquence du changement de la classe de conteneur

Il n'y a pas de conséquence lors du changement de conteneur étant donné que les types sont redéfinis dans la classe Objet3DContainer. Il suffit de changer `using Objet3DContainer = std::vector<Objet3DPtr>;` et de mettre un autre conteneur pour que cela fonctionne.

Question 4

Surcharge des opérateurs * et ->

Cela a comme avantage de pouvoir utiliser ces opérations de manière que ces opérations fonctionnent comme prévu afin d'améliorer efficacité du programmeur. Cependant, un programmeur n'ayant pas vu l'implémentation ne s'attendra pas à cela.