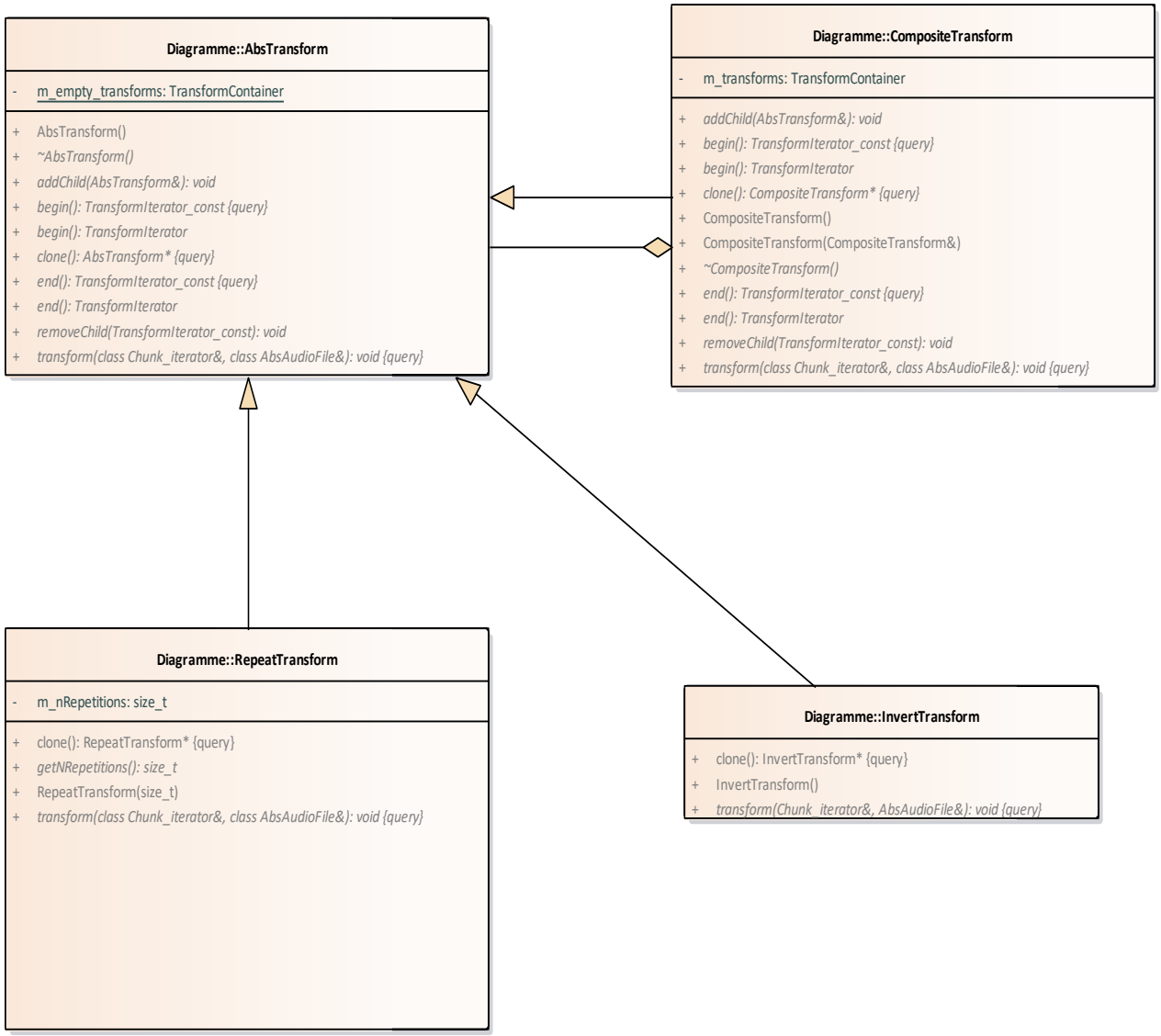
2 - Patron Composite

1)

a) L'intention du patron composite est essentiellement de manipuler des objets similaires (repeatTransform, invertTransform) par l'intermédiaire d'une interface(AbsTransform), qui ont des méthodes communes et aux implémentations adaptées à chaque objet.

b)



2)

Feuilles : invertTransform, repeatTransform

Responsabilités :

invertTransform : inverser le segment audio

repeatTransform : dupliquer le segment audio un nombre de fois donné.

Composite : compositeTransform

Responsabilitees :

Inverser le segment audio

Dupliquer le segment audio un nombre de fois donné.

Ajouter et supprimer des enfants.

Cloner la transformation composite et ses commandes enfant.

Retourne le debut et la fin du conteneur.

Component : absTransform

Responsabilitees :

Definie l'interface uniforme qui :

effectue la transformation

Ajoute et supprime un enfant

Retourne le debut et la fin du conteneur.

3) L'attribut de type transformContainer du compositeTransform est responsable de la creation de l'arbre des composantes.

3 - Patron Proxy

a) L'intention du parton proxy est de créer un conteneur pour un objet avec lequel le programme doit interagir. Dans ce cas les objets en question sont des fichiers « audio ». MemAudioFile.cpp implémente une façon de stocker les données nécessaires dans la mémoire sans avoir besoin de relire les fichiers binaires à chaque fois que les données en question doivent être utilisés. Un des avantages les plus importants est aussi la facilitation d'interaction et manipulation des données grâce au patron proxy. MemAudioFile.cpp implémente justement un itérateur pour parcourir les « chunks ». Tout ceci non-seulement rends les interactions moins couteuses mais aussi plus simples à faire.

b)

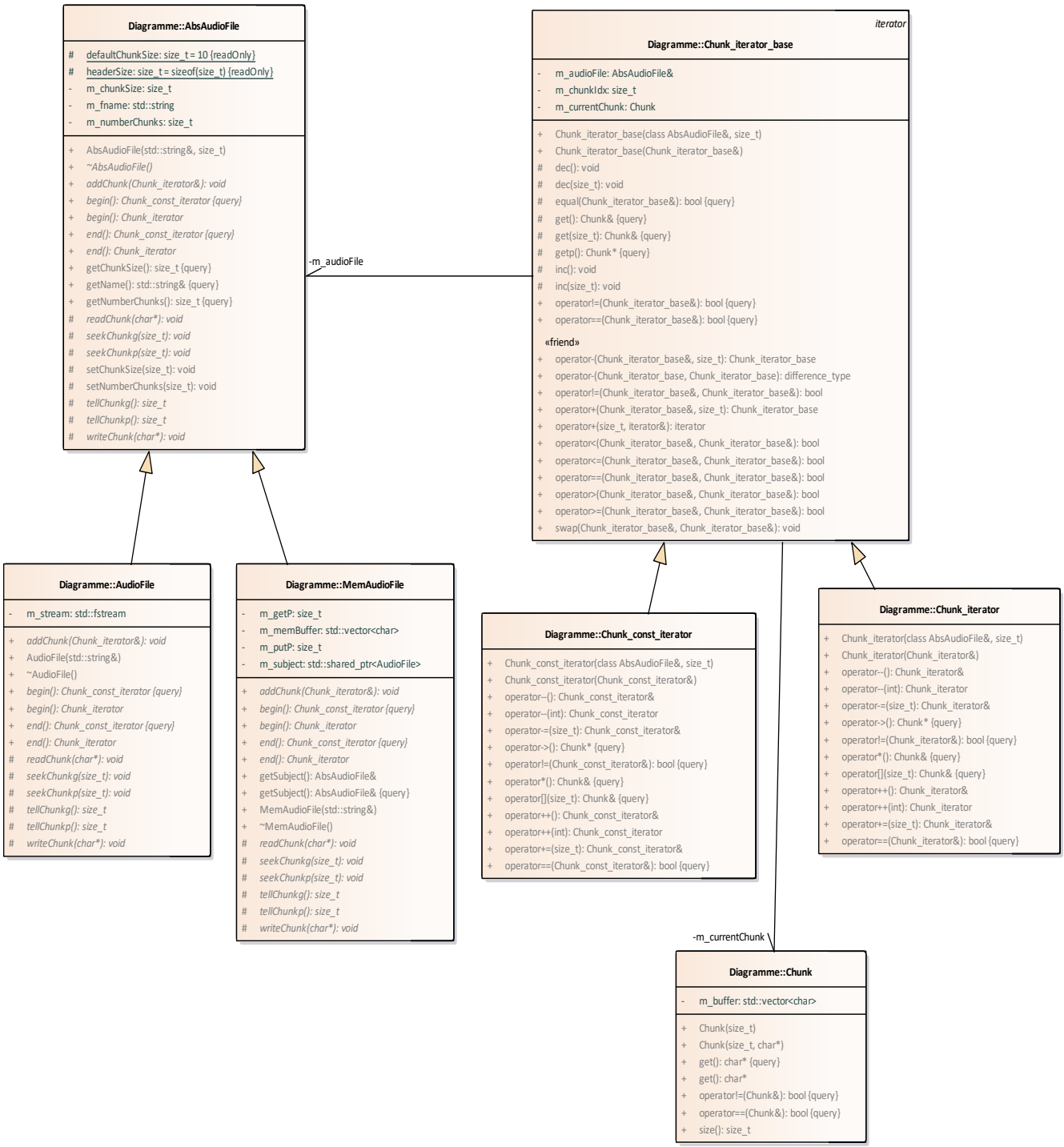


Diagramme::AbsAudioFile
defaultChunkSize: size_t = 10 {readOnly}
headerSize: size_t = sizeof(size_t) {readOnly}
- m_chunkSize: size_t
- m_fname: std::string
- m_numberChunks: size_t
+ AbsAudioFile(std::string&, size_t)
+ ~AbsAudioFile()
+ addChunk(Chunk_iterator&): void
+ begin(): Chunk_const_iterator {query}
+ begin(): Chunk_iterator
+ end(): Chunk_const_iterator {query}
+ end(): Chunk_iterator
+ getChunkSize(): size_t {query}
+ getName(): std::string& {query}
+ getNumberChunks(): size_t {query}
# readChunk(char*): void
# seekChunkg(size_t): void
# seekChunkp(size_t): void
# setChunkSize(size_t): void
# setNumberChunks(size_t): void
# tellChunkg(): size_t
# tellChunkp(): size_t
# writeChunk(char*): void
-m_audioFile
iterator
Diagramme::Chunk_iterator_base
- m_audioFile: AbsAudioFile&
- m_chunkIdx: size_t
- m_currentChunk: Chunk
+ Chunk_iterator_base(class AbsAudioFile&, size_t)
+ Chunk_iterator_base(Chunk_iterator_base&)
# dec(): void
# dec(size_t): void
# equal(Chunk_iterator_base&): bool {query}
# get(): Chunk& {query}
# get(size_t): Chunk& {query}
# getp(): Chunk* {query}
# inc(): void
# inc(size_t): void
+ operator!=(Chunk_iterator_base&): bool {query}
+ operator==(Chunk_iterator_base&): bool {query}
«friend»
+ operator-(Chunk_iterator_base&, size_t): Chunk_iterator_base
+ operator-(Chunk_iterator_base, Chunk_iterator_base): difference_type
+ operator!=(Chunk_iterator_base&, Chunk_iterator_base&): bool
+ operator+(Chunk_iterator_base&, size_t): Chunk_iterator_base
+ operator+(size_t, iterator&): iterator
+ operator<(Chunk_iterator_base&, Chunk_iterator_base&): bool
+ operator<=(Chunk_iterator_base&, Chunk_iterator_base&): bool
+ operator==(Chunk_iterator_base&, Chunk_iterator_base&): bool
+ operator>(Chunk_iterator_base&, Chunk_iterator_base&): bool
+ operator>=(Chunk_iterator_base&, Chunk_iterator_base&): bool
+ swap(Chunk_iterator_base&, Chunk_iterator_base&): void
Diagramme::AudioFile
- m_stream: std::fstream
+ addChunk(Chunk_iterator&): void
+ AudioFile(std::string&)
+ ~AudioFile()
+ begin(): Chunk_const_iterator {query}
+ begin(): Chunk_iterator
+ end(): Chunk_const_iterator {query}
+ end(): Chunk_iterator
# readChunk(char*): void
# seekChunkg(size_t): void
# seekChunkp(size_t): void
# tellChunkg(): size_t
# tellChunkp(): size_t
# writeChunk(char*): void
Diagramme::MemAudioFile
- m_getP: size_t
- m_memBuffer: std::vector<char>
- m_putP: size_t
- m_subject: std::shared_ptr<AudioFile>
+ addChunk(Chunk_iterator&): void
+ begin(): Chunk_const_iterator {query}
+ begin(): Chunk_iterator
+ end(): Chunk_const_iterator {query}
+ end(): Chunk_iterator
+ getSubject(): AbsAudioFile&
+ getSubject(): AbsAudioFile& {query}
+ MemAudioFile(std::string&)
+ ~MemAudioFile()
# readChunk(char*): void
# seekChunkg(size_t): void
# seekChunkp(size_t): void
# tellChunkg(): size_t
# tellChunkp(): size_t
# writeChunk(char*): void
Diagramme::Chunk_const_iterator
+ Chunk_const_iterator(class AbsAudioFile&, size_t)
+ Chunk_const_iterator(Chunk_const_iterator&)
+ operator--(): Chunk_const_iterator&
+ operator--(int): Chunk_const_iterator
+ operator-=(size_t): Chunk_const_iterator&
+ operator->(): Chunk* {query}
+ operator!=(Chunk_const_iterator&): bool {query}
+ operator*(): Chunk& {query}
+ operator[](size_t): Chunk& {query}
+ operator++(): Chunk_const_iterator&
+ operator++(int): Chunk_const_iterator
+ operator+=(size_t): Chunk_const_iterator&
+ operator==(Chunk_const_iterator&): bool {query}
Diagramme::Chunk_iterator
+ Chunk_iterator(class AbsAudioFile&, size_t)
+ Chunk_iterator(Chunk_iterator&)
+ operator--(): Chunk_iterator&
+ operator--(int): Chunk_iterator
+ operator-=(size_t): Chunk_iterator&
+ operator->(): Chunk* {query}
+ operator!=(Chunk_iterator&): bool {query}
+ operator*(): Chunk& {query}
+ operator[](size_t): Chunk& {query}
+ operator++(): Chunk_iterator&
+ operator++(int): Chunk_iterator
+ operator+=(size_t): Chunk_iterator&
+ operator==(Chunk_iterator&): bool {query}
-m_currentChunk
Diagramme::Chunk
- m_buffer: std::vector<char>
+ Chunk(size_t)
+ Chunk(size_t, char*)
+ get(): char* {query}
+ get(): char*
+ operator!=(Chunk&): bool {query}
+ operator==(Chunk&): bool {query}
+ size(): size_t

4 - Conteneurs et Patrons Iterator

a) Identifiez l'intention du patron Iterator

En théorie, l'intention du patron Iterator est de << Fournir une méthode d'accès séquentielle aux éléments d'un objet agrégat (liste, vecteur, …) sans exposer sa structure interne.>> [1]

b) Identifiez 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.

Le conteneur de la STL utilisé dans la classe Composite est *vector.* En effet, l'utilisation du *pushback* dans la méthode *addChild* nous permet de confirmer l'utilisation du conteneur *vector*. L'une des classes d'*Iterators* utilisées est *TransformIterator* et hérite de *TransformBaseIterator.* Aussi, *Chunk_iterator_base* qui herite de *std::iterator<std::random_access_iterator_tag, size_t>* qui offre un accès séquentiel est un autre *Iterator* utilisé dans la conception. Nous avons aussi *Chunk_const_iterator* et *Chunk_iterator* qui sont des *Iterators* qui agissent sur des *Chunk* constant et non-constant respectivement.

c) Expliquez le rôle de l'attribut statique *m_empty_transforms* défini dans la classe AbsTransform. Expliquez pourquoi, selon vous, cet attribut est déclaré comme un attribut statique et privé.

En pratique, un attribut *static* a pour but de permettre son partage avec tous les objets de la même classe et ainsi éviter sa copie. Le fait que l'on rende un attribut *static* privé fait en sorte qu'il est partagé par tous les objets de la même classe et que sa modification ne puisse se faire en dehors de sa classe. C'est exactement le comportement souhaité pour l'attribut *static m_empty_transforms.* Cet attribut est un vecteur vide de transformation ayant comme rôle de

retourner des itérateurs valides. Il est donc normal que son partage soit exclusif à sa classe et sa copie limitée à une seule (*static)*.

d) <u>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 suite au 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 ?</u>

Si l'on venait à changer les conteneurs, il faudrait appliquer certains changements pour s'assurer que nous respections le principe d'encapsulation, que le comportement de l'héritage soit bel et bien celui souhaité et que l'on puisse accéder aux objets dans les conteneurs grâce à différentes méthodes puisque le risque est qu'en changeant le conteneur, nous n'ayons plus un *Random Access Iterator.* Donc, peu importe le conteneur choisi, il sera important de déterminer son type d'itérateur. Ensuite, il faudra déterminer le classement des méthodes et attributs selon le résultats souhaité (*public, private, protected)*. La notion de *protected* sera importante car elle permet aux classes enfants d'avoir accès aux attributs/méthodes *protected* contrairement au *private.* Cela permettra le respect du principe d'encapsulation qui était déjà dans la conception proposé pour le TP4 puisque l'accès aux vecteur était limité à la classe propriétaire de l'attribut.

e) <u>Les classes dérivées TransformIterator et TransformIterator_const surchargent les opérateur « * » et « -> ». Cette décision de conception a des avantages et des inconvénients. Identifiez un avantage et un inconvénient de cette décision.</u>

La surcharge de ses opérateurs permet à l'iterateur d'être de type *Random Access Iterator* qui utilise les pointeurs. Cette pratique est avantageuse car elle nous évite toutes sortes de méthode tel que *previous, next, hasNext, etc* pour pouvoir parcourir le conteneur qui est un vecteur dans notre cas. Par contre, l'utilisation de ces pointeurs nous force à maintenir une cohérence dans leur gestion. Effectivement, nous ne voudrions pas que ceux pointent vers des objets non existants. De plus, le fait de surcharger ces opérateurs implique que nous devons

nous assurer que tous changements au fonctionnement basique des pointeurs en C++ doivent être cohérent dans tout le code.

1) Notes de cours LOG2410 Chapitre 11 p.29