Étienne Perron : 2128926

Mounir Lammali : 2141302

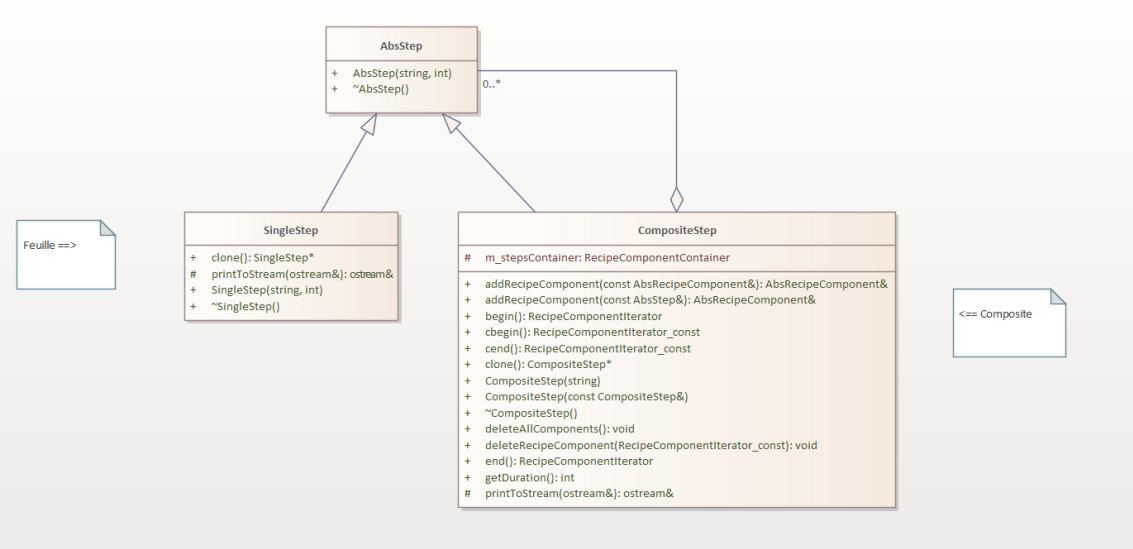
TP5

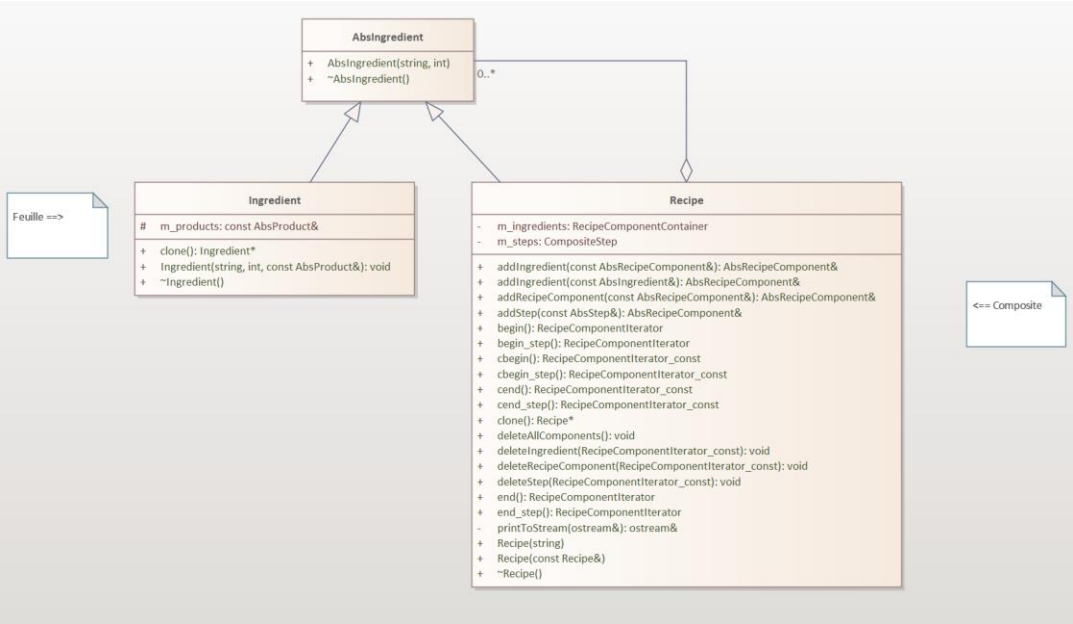
Partie 2 – Patron Composite

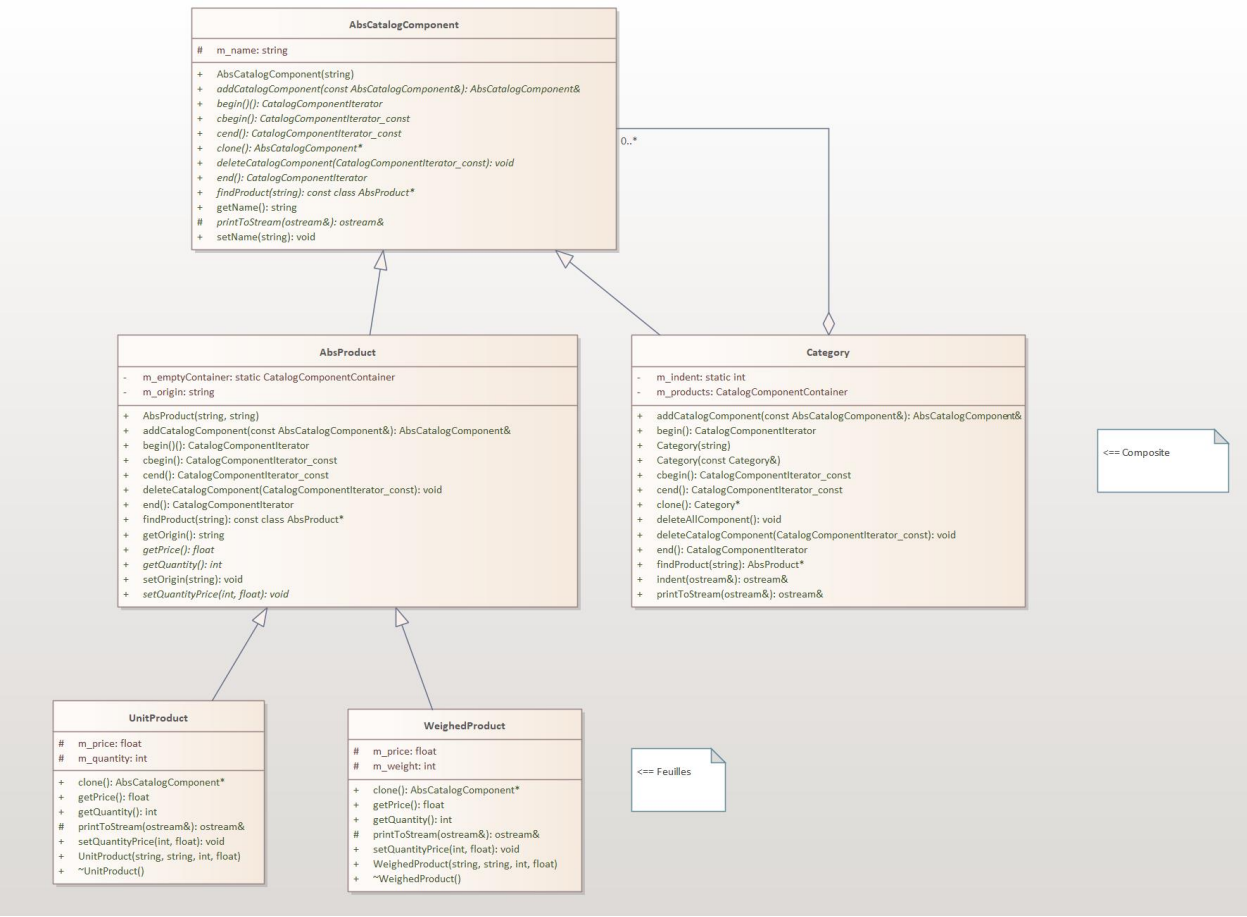
1)

a. L'intention du patron composite est de faire abstraction des différences entre les objets composés et primitifs pour nous permettre de les manipuler indifféremment des différences.

b.







c. Il permet de définir une valeur de m_indent qui sera la même pour toutes les instances de la classe et de ses enfants. Ainsi, si on souhaite modifier m_indent pour l'ajuster, il sera modifié pour tous les objets de la mère et de ses enfants. Ceci permet d'incrémenter et décrémenter m_indent alors que l'on s'enfonce dans l'arborescence de la structure que l'on souhaite afficher pour fournir un affichage en arbre. Autrement, il serait beaucoup plus difficile de connaître le nombre d'indentation courant lorsqu'on affiche un *SingleStep* qui est inclus dans le un *CompositeStep*.

d. La méthode printToStream permet d'afficher le contenu des différentes classes sans connaître les détails de leur implémentation, et ce, en appelant simplement l'opérateur << sur l'objet. En effet, l'opérateur << appelle la méthode printToStream qui s'occupe de retourner les informations à afficher selon un formatage précis que l'utilisateur de l'opérateur << n'a pas besoin de connaître.

Partie 3 – Conteneurs et Patron Itérateur

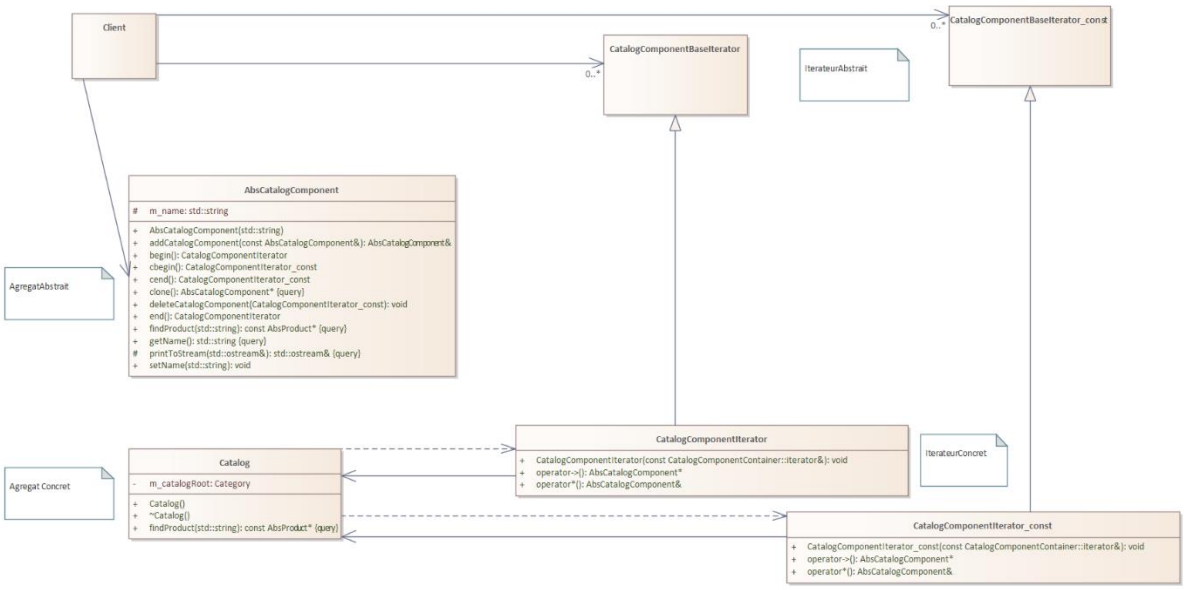
2)

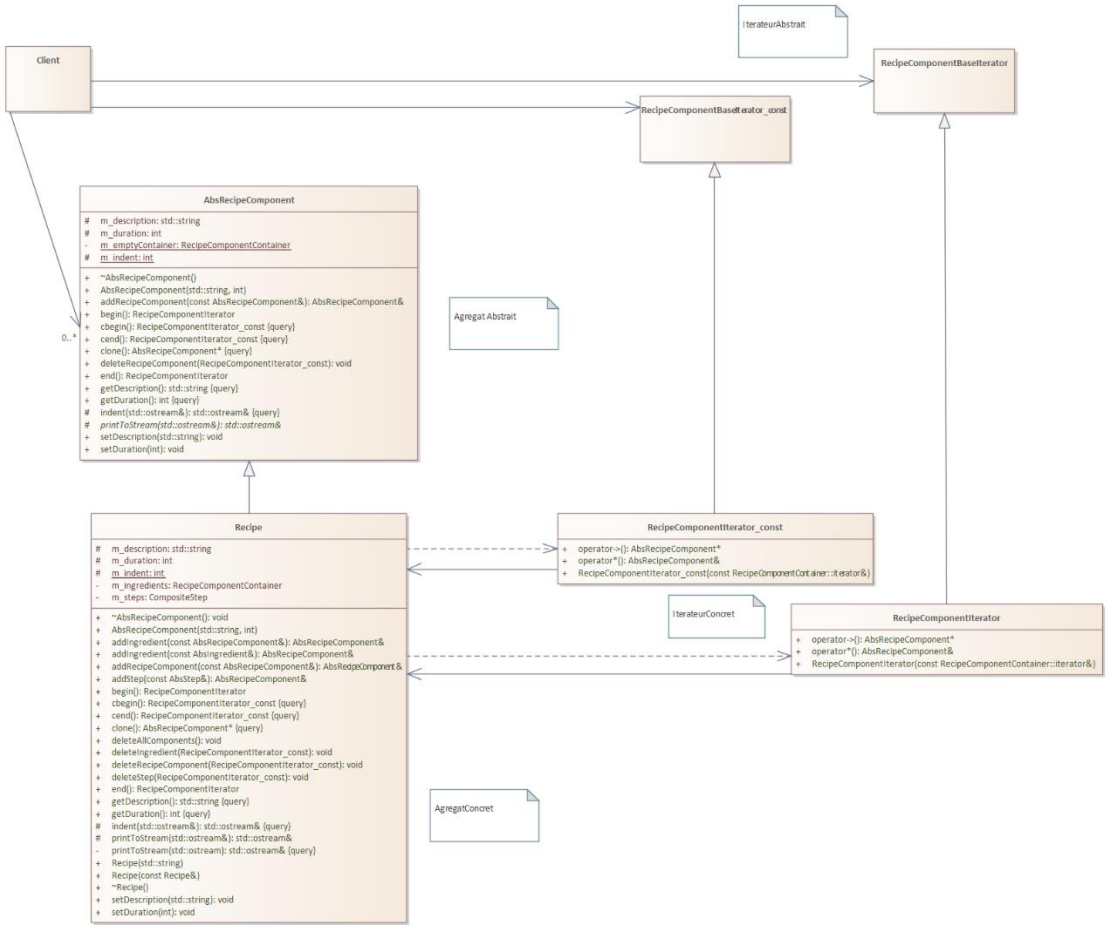
a. L'intention du patron itérateur est de parcourir les éléments agrégés d'un objet les uns à la suite des autres sans briser l'encapsulation.

b. Pour les classes Recipe et CompositeStep, le conteneur de la STL utilisé est le std::vector<std::unique_ptr<AbsRecipeComponent>> et les itérateurs sont std::vector<std::unique_ptr<AbsRecipeComponent>>::iterator,

std::vector<std::unique_ptr<AbsRecipeComponent>>::const_iterator. Pour la classe Category, le conteneur de la STL utilisé est le std::list<std::unique_ptr<AbsCatalogComponent>> et les itérateurs utilisés sont std::list<std::unique_ptr<AbsCatalogComponent>>::iterator, std::list<std::unique_ptr<AbsCatalogComponent>>::const_iterator

c.


Client
0..*
CatalogComponentBaseIterator_const
0..*
CatalogComponentBaseIterator
IterateurAbstrait
AbsCatalogComponent
m_name: std::string
+ AbsCatalogComponent(std::string)
+ addCatalogComponent(const AbsCatalogComponent&): AbsCatalogComponent&
+ begin(): CatalogComponentIterator
+ cbegin(): CatalogComponentIterator_const
+ cend(): CatalogComponentIterator_const
+ clone(): AbsCatalogComponent* {query}
+ deleteCatalogComponent(CatalogComponentIterator_const): void
+ end(): CatalogComponentIterator
+ findProduct(std::string): const AbsProduct* {query}
+ getName(): std::string {query}
# printToStream(std::ostream&): std::ostream& {query}
+ setName(std::string): void
AgregatAbstrait
CatalogComponentIterator
+ CatalogComponentIterator(const CatalogComponentContainer::iterator&): void
+ operator->(): AbsCatalogComponent*
+ operator*(): AbsCatalogComponent&
IterateurConcret
Agregat Concret
Catalog
- m_catalogRoot: Category
+ Catalog()
+ ~Catalog()
+ findProduct(std::string): const AbsProduct* {query}
CatalogComponentIterator_const
+ CatalogComponentIterator_const(const CatalogComponentContainer::iterator&): void
+ operator->(): AbsCatalogComponent*
+ operator*(): AbsCatalogComponent&

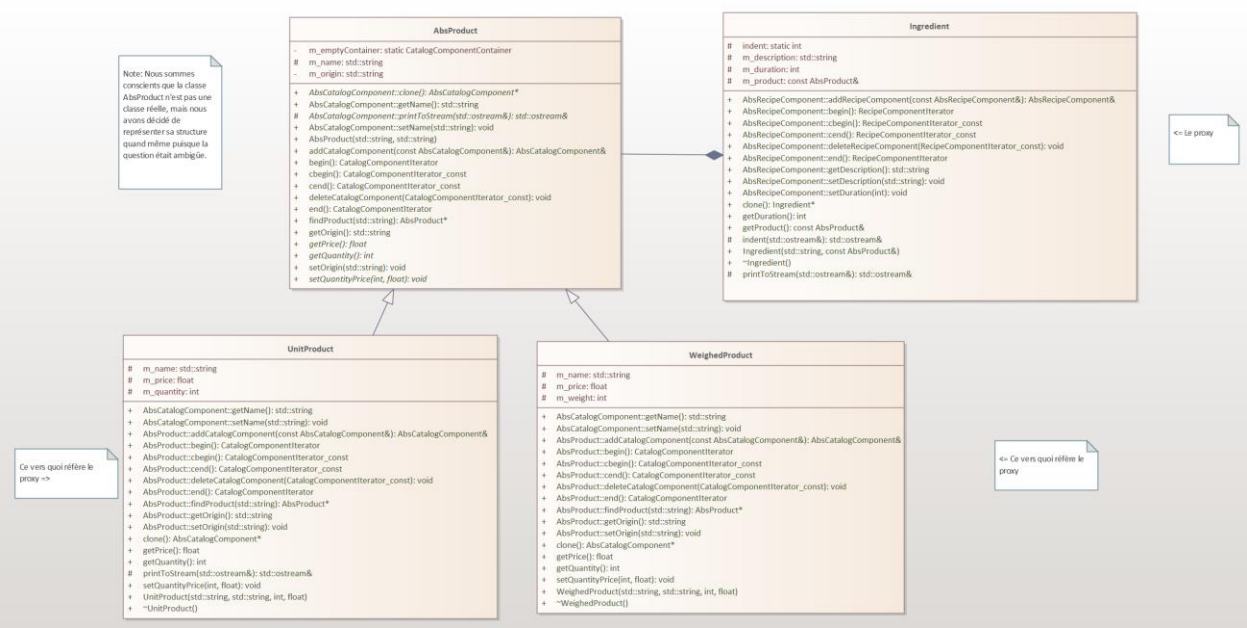

IterateurAbstrait
Client
RecipeComponentBaseIterator
RecipeComponentBaseIterator_const
AbsRecipeComponent
m_description: std::string
m_duration: int
- m_emptyContainer: RecipeComponentContainer
m_indent: int
+ ~AbsRecipeComponent()
+ AbsRecipeComponent(std::string, int)
+ addRecipeComponent(const AbsRecipeComponent&): AbsRecipeComponent&
+ begin(): RecipeComponentIterator
+ cbegin(): RecipeComponentIterator_const {query}
+ cend(): RecipeComponentIterator_const {query}
+ clone(): AbsRecipeComponent* {query}
+ deleteRecipeComponent(RecipeComponentIterator_const): void
+ end(): RecipeComponentIterator
+ getDescription(): std::string {query}
+ getDuration(): int {query}
# indent(std::ostream&): std::ostream& {query}
# printToStream(std::ostream&): std::ostream&
+ setDescription(std::string): void
+ setDuration(int): void
0..*
Agregat Abstrait
Recipe
m_description: std::string
m_duration: int
m_indent: int
- m_ingredients: RecipeComponentContainer
- m_steps: CompositeStep
+ ~AbsRecipeComponent(): void
+ AbsRecipeComponent(std::string, int)
+ addIngredient(const AbsRecipeComponent&): AbsRecipeComponent&
+ addIngredient(const AbsIngredient&): AbsRecipeComponent&
+ addRecipeComponent(const AbsRecipeComponent&): AbsRecipeComponent&
+ addStep(const AbsStep&): AbsRecipeComponent&
+ begin(): RecipeComponentIterator
+ cbegin(): RecipeComponentIterator_const {query}
+ cend(): RecipeComponentIterator_const {query}
+ clone(): AbsRecipeComponent* {query}
+ deleteAllComponents(): void
+ deleteIngredient(RecipeComponentIterator_const): void
+ deleteRecipeComponent(RecipeComponentIterator_const): void
+ deleteStep(RecipeComponentIterator_const): void
+ end(): RecipeComponentIterator
+ getDescription(): std::string {query}
+ getDuration(): int {query}
# indent(std::ostream&): std::ostream& {query}
# printToStream(std::ostream&): std::ostream&
- printToStream(std::ostream): std::ostream& {query}
+ Recipe(std::string)
+ Recipe(const Recipe&)
+ ~Recipe()
+ setDescription(std::string): void
+ setDuration(int): void
AgregatConcret
RecipeComponentIterator_const
+ operator->(): AbsRecipeComponent*
+ operator*(): AbsRecipeComponent&
+ RecipeComponentIterator_const(const RecipeComponentContainer::iterator&)
IterateurConcret
RecipeComponentIterator
+ operator->(): AbsRecipeComponent*
+ operator*(): AbsRecipeComponent&
+ RecipeComponentIterator(const RecipeComponentContainer::iterator&)

d. L'attribut m_emptyContainer permet de fournir une implémentation par défaut des méthodes begin(), cbegin(), cend() et end() qui retournent un itérateur. Comme les classes AbsRecipeComponent et AbsProduct sont vides, alors elles contiennent un conteneur vide m_emptyContainer sur lequel il est possible d'appeler les méthodes mentionnées plus tôt. Ces méthodes correspondent alors à l'implémentation par défaut dont les classes enfants héritent ce qui leur permet de redéfinir un comportement propre à leur classe ou d'utiliser celui de la classe parent. L'attribut m_emptyContainer est privé puisqu'on ne veut pas que les classes enfants le possèdent aussi étant donné que celles-ci devront être concrètes et posséder un conteneur qui leur est propre. Le mot-clé static permet d'allouer une simple case mémoire à m_emptyContainer. Ainsi, ce sera toujours celui-ci qui sera appelé si on invoque la méthode AbsProduct::begin() ou AbsRecipeComponent::begin() de même que pour les méthodes cbegin(), cend() et end() des mêmes classes.
e. Nous avons modifié la classe de conteneur pour un std::vector. Nous remarquons alors que l'encapsulation est parfaitement respectée puisque les classes qui héritent de la classe composite n'ont pas à se soucier des modifications apportées à la classe parent. En effet, les méthodes de cette classe s'utilisent toujours de la même façon et ont toujours le même comportement qu'auparavant. De plus, les classes enfants ne seront jamais au courant que le conteneur utilisé pour stocker les composites a été modifié, ce qui correspond à la définition de l'encapsulation.

Partie 4 – Patron Proxy

1)

a. Le patron Proxy permet de manipuler une référence à un objet au lieu de l'objet lui-même pour contrôler l'accès à ce dernier. Ceci permet de travailler avec une référence plus sophistiquée qu'un pointeur sans devoir manipuler l'objet lui-même.
b. Dans l'application *Recipes*, le proxy *Ingredient* est de type référence intelligente puisqu'il ajoute des fonctionnalités à l'objet original. En effet, lorsqu'on manipule un *AbsProduct* au moyen d'un *Ingredient*, il est possible d'afficher le *AbsProduct* dans un *stream* de sortie alors que cette fonctionnalité n'est pas présente si on accède directement au *AbsProduct*.
c.

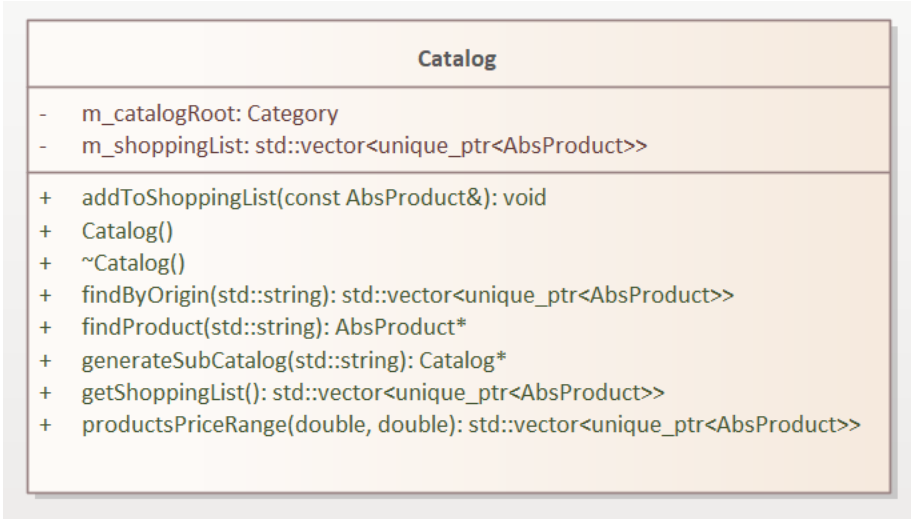


Partie 5 – Architecture 3 niveaux

3)

a. L'essentiel du code fourni dans l'application Recipes correspond à la logique d'application puisque c'est le code qui s'occupe de la gestion des informations des recettes. Par exemple, c'est là que l'on retrouve les méthodes pour trouver une recette ou un produit et pour ajouter un ingrédient. De plus, nous pouvons constater que le code ne comporte aucun module de sauvegarde ; il ne se trouve donc pas à ce niveau. Par ailleurs, l'affichage et interaction avec l'utilisateur est très minimale, voire nulle. Le seul segment de présentation est dû au besoin de confirmer la validité de notre code et n'est pas nécessaire à l'application. Ainsi, nous pouvons déduire que le code fourni ne se situe pas au niveau de la couche de présentation

b. Interface de la classe Catalog :

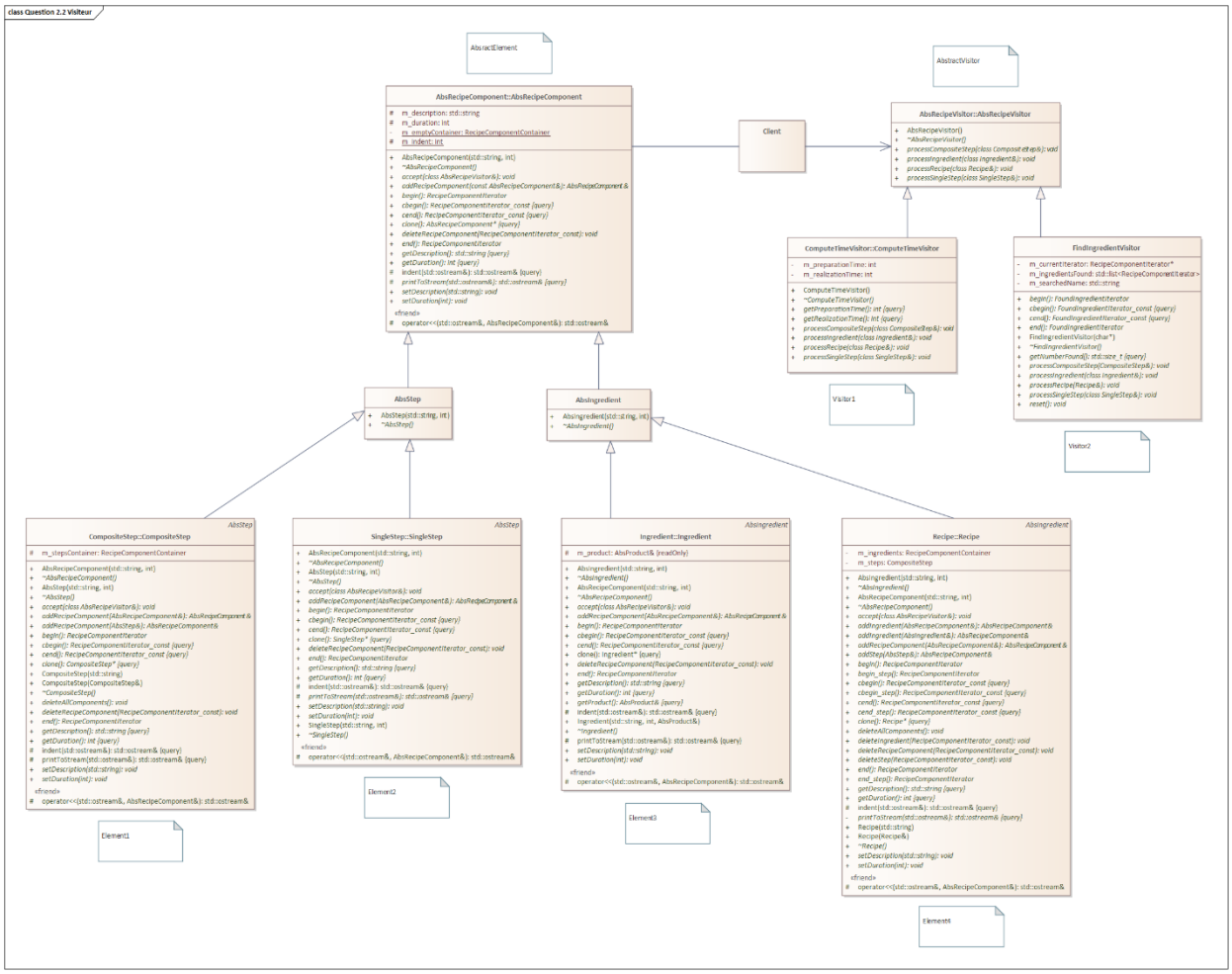


TP6

Partie 2 – Patron Visiteur

2.1 Le patron visiteur permet d'effectuer une opération sur un objet en limitant les modifications sur la structure de sa classe. Il est utilisé pour éviter de faire diminuer la cohérence d'une classe en y ajoutant plusieurs méthodes qui ont peu de lien entre elles ou lorsqu'on souhaite réaliser une opération sur plusieurs objets munis d'une interface différente. Avec le patron visiteur, la structure de la classe demeure inchangée une fois en production alors que l'on peut constamment ajouter de nouvelles opérations à appliquer sur une instance de la classe sans risquer de causer des problèmes sur celle-ci.

2.2



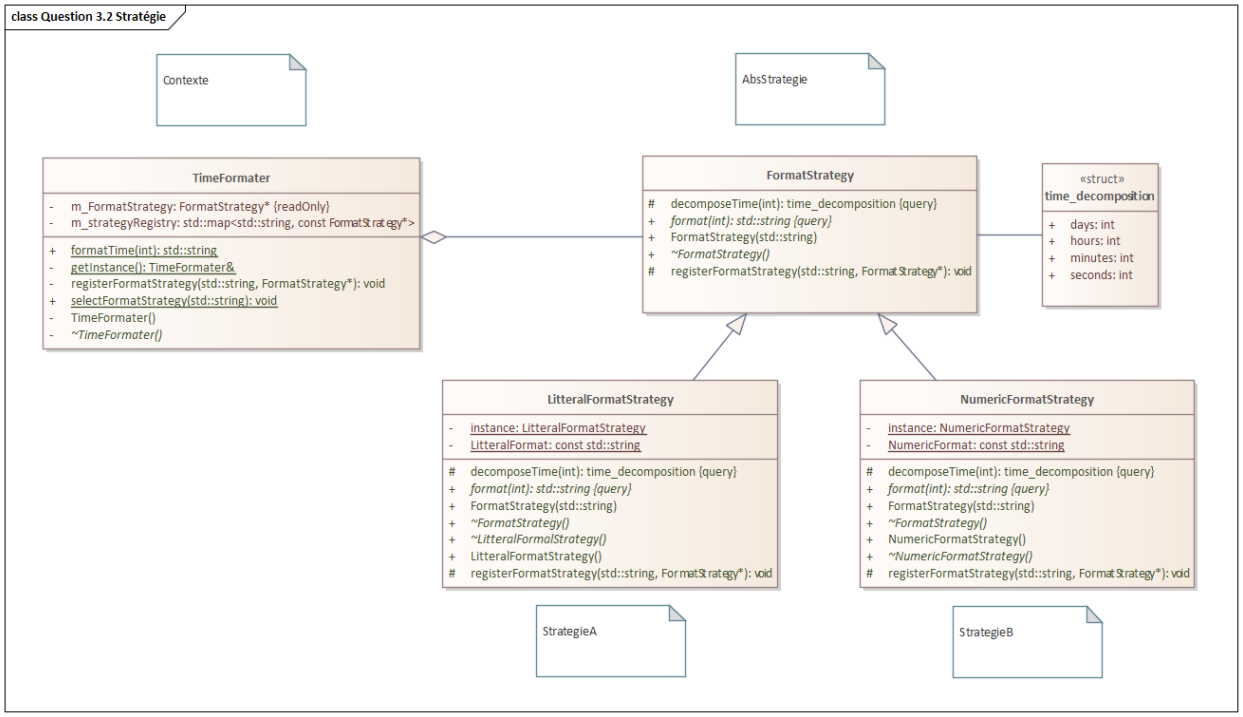
2.3 La méthode *printToStream* aurait pu être implémentée à l'aide d'un visiteur puisqu'elle est appliquée sur plusieurs objets dont l'interface est différente. De plus, l'intérieur de la fonction est différent d'une méthode à l'autre, alors un visiteur pourrait contenir une méthode *printToStream* pour chaque classe affichable. Aussi, les informations que l'on souhaite accéder par la méthode *printToStream* sont accessible par l'interface des classes affichables, donc un visiteur y aurait accès.

2.4 Si nous souhaitions ajouter une sous-classe dérivée de AbsRecipeComponent, il serait alors nécessaire de modifier les classes AbsRecipeVisitor, ComputeTimeVisitor et FindIngredientVisitor pour y intégrer les méthodes permettant d'effectuer les opérations sur la nouvelle classe.

Partie 3 – Patron Stratégie

3.1 Le patron Stratégie permet de proposer à un contexte plusieurs algorithmes différents pour réaliser une même tâche. Ceci permet à l'utilisateur de sélectionner l'algorithme le plus adapté à la situation et de l'utiliser.

3.2



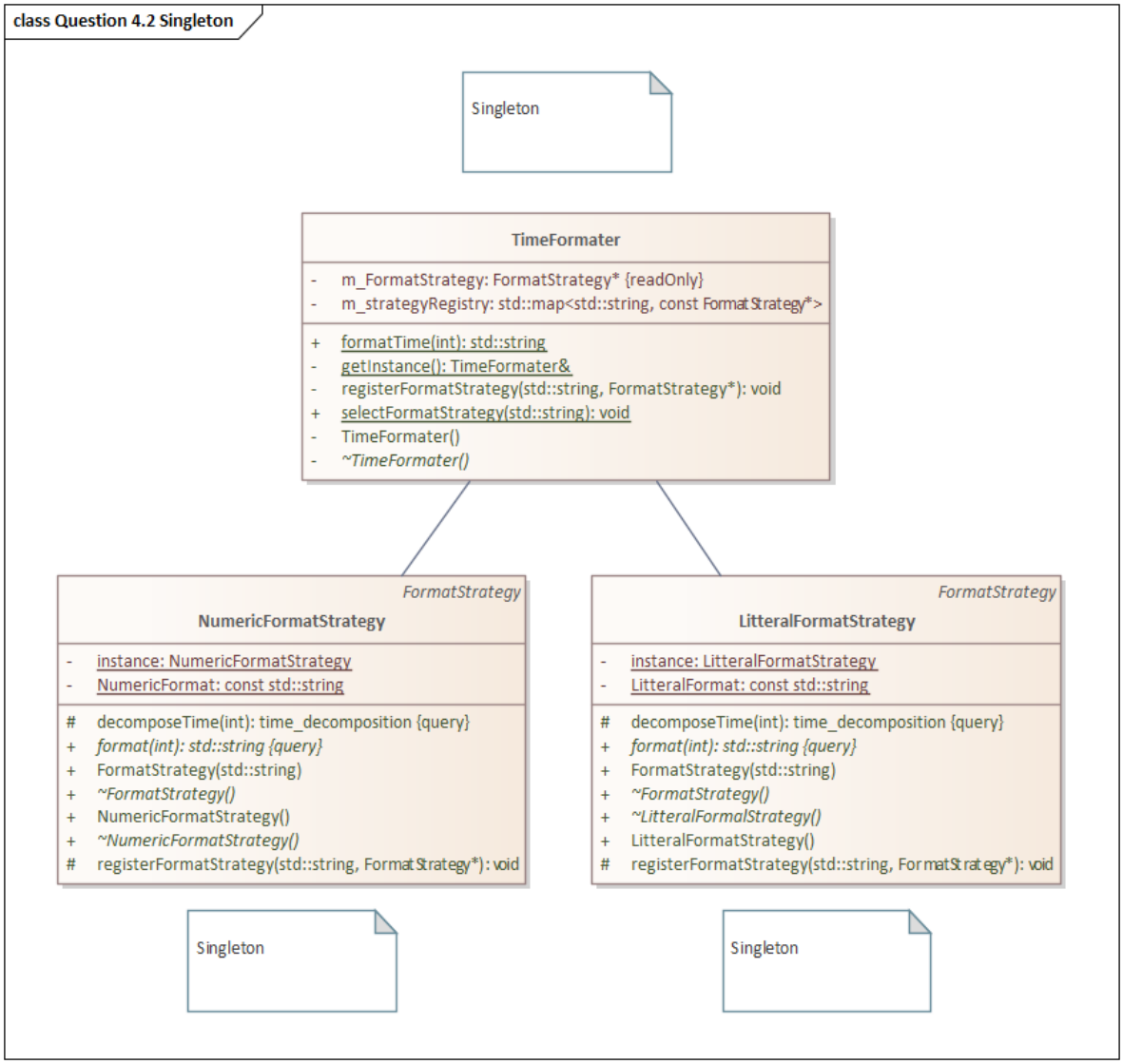
3.3 Nous n'aurions pas de changements à apporter aux classes implémentant le patron Stratégie. En effet, l'ajout d'un nouvel algorithme n'affecte pas la classe Stratégie abstraite, ni ses classes concrètes. De plus, le contexte n'a pas besoin d'être modifié pour considérer le nouvel algorithme puisque, à ses yeux, tous les algorithmes ne sont que des *FormatStrategy*. La nouvelle classe à ajouter serait celle encapsulant le nouvel algorithme que nous souhaitons ajouter à la stratégie.

3.4 Dans le code actuel, si aucune stratégie de formatage n'est choisie par le client, l'attribut *m_FormatStrategy* de la classe *TimeFormater* sera un *nullptr*, ce qui aura pour effet de retourner les secondes sous la forme d'une *string* sans modification quant au format si on tente d'appeler la méthode de formatage. Une solution pour proposer une stratégie de formatage dans le cas où l'utilisateur n'en sélectionne aucune serait de simplement incorporer une stratégie de formatage par défaut dans le constructeur dans la liste d'initialisation.

Partie 4 – Patron Singleton

4.1 Le patron Singleton permet de s'assurer qu'une classe ne possède pas plus d'un certain nombre d'instance d'elle-même. Il permet aussi d'accéder à la classe à n'importe quel endroit dans le programme tout en protégeant l'instance de la classe contre des modifications qui peuvent survenir ailleurs dans le code. Ainsi, le patron Singleton permet d'offrir l'avantage du point d'accès global des variables globales sans permettre le désavantage qui est que celles-ci peuvent être écrasée à n'importe quel moment dans le programme.

4.2



4.3 La première approche implique l'appelle d'une méthode getInstance() qui retourne l'instance en question. Comme celle-ci est indiquée "static", son instanciation n'a lieu qu'une fois. Ainsi, si on tente d'appeler la méthode une seconde fois, le programme ignorera la ligne d'instanciation et

retournera simplement l'objet déjà créé. Cette approche est observée dans la classe TimeFormatter. Dans le second cas observable dans les classes LitteralFormatStrategy et NumericFormatStrategy, on a simplement l'instance en attribut static. Ainsi, celui-ci sera instancié lorsque le constructeur de la classe sera évalué. Nous ne croyons pas qu'une approche soit meilleure que l'autre, mais plutôt que cela dépend du contexte. En effet, si on souhaite accéder directement à l'instance, il est préférable d'utiliser une méthode getInstance() qui nous retourne l'instance en question. Par ailleurs, si l'accès à l'instance n'est pas nécessaire, alors la méthode getInstance() est à éviter puisqu'elle diminue l'encapsulation.

4.4 Lorsque le programme s'arrêtera, les objets singleton ne seront pas explicitement détruits puisque l'on n'appelle pas explicitement le destructeur ou le mot-clé delete (si alloué dynamiquement) sur les objets singletons.