M3105 – TD : Un Kata Kebab à notre sauce[1]
Visitons les kébabs !

Il vous est conseillé de faire ce TD en pair-programming 😉

Rappelons le contexte de notre client :

« Bonjour, je suis un vendeur de Kebab.
J'ai besoin de fabriquer des kebabs de toutes sortes.
Les ingrédients sont variés : salade, tomate, oignon, agneau, bœuf, cheddar, etc. »

« Pour le bien-être de mes clients, je dois pouvoir déterminer le régime d'un kébab c-a-d un kebab doit pouvoir me dire s'il est végétarien ou non, s'il est pescétarien ou non… »

Remarques :

- D'après Wikipédia :
  - ➔ Le **végétarisme** est une pratique alimentaire qui exclut la consommation de chair animale.
  - ➔ Le **pescétarisme**, ou pesco-végétarisme, est un néologisme désignant le régime alimentaire d'une personne omnivore qui s'abstient de consommer de la chair animale à l'exception de celle issue des poissons, des crustacés et mollusques aquatiques
    Remarque : **Pescetarien** = végétarien + poissons + **crevettes**

- Nous nous **limitons pour l'instant aux régimes alimentaires** c-a-d que nous n'essaierons pas de modifier le kebab en lui doublant le fromage ou en lui enlevant les oignons… d'ailleurs **ces deux dernières fonctionnalités sont-elles bien de la responsabilité du kebab (quid du principe SRP ?)**

*Pour commencer, vous travaillerez en **mode déconnecté** :*
Laissez vos ordinateurs éteints pour le moment 😉

1. Une « quick design[2] » session pour commencer …

Un Kébab, qu'es acquo ?

Avant de s'intéresser aux différents régimes, commencez par **modéliser à l'aide d'un diagramme de classes** le fait qu'un kébab est composé d'ingrédients et qu'un ingrédient peut-être du pain, de la salade, de la tomate, de l'oignon, de l'agneau, du thon, de la crevette, du fromage, de la sauce, …

Photo extraite de http://www.ledahu.net/dahu/astuces-du-dahu/les-bienfaits-du-kebab/

[1] Cet énoncé a été écrit après avoir participé au Kata Kébab créé par Romeu Mora : **https://github.com/malk/the-kebab-kata**
[2] https://www.agilealliance.org/glossary/quickdesign/ (conception au tableau blanc : http://institut-agile.fr/quickdesign.html)

Quid des régimes végétarien et pescétarien ?

Maintenant que vous êtes en mesure de confectionner un kébab,
vous allez pouvoir vous intéresser aux régimes alimentaires…

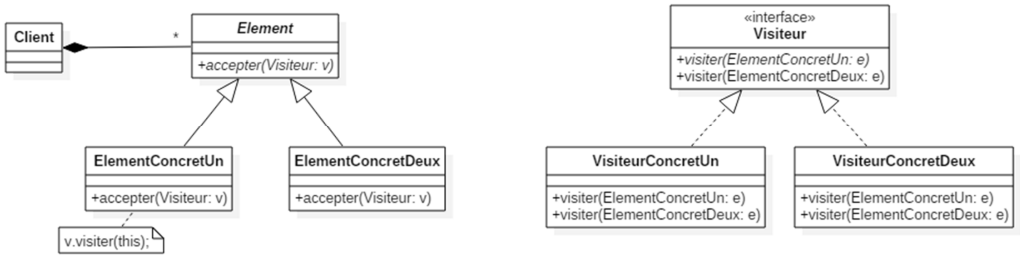
→ **Le kébab est-il (ou pas) végétarien ? Le kébab est-il (ou pas) pescétarien ?** Dans votre diagramme de classes, **quelle classe est susceptible d'être responsable de répondre à ces questions ?** Compléter votre diagramme en faisant apparaître ces deux nouvelles opérations.

→ Comment savoir si un kébab est végétarien et/ou pescétarien ?
Comment réaliser une implémentation aussi propre que possible qui sera facilement extensible en terme de régime pour vous permettre d'ajouter facilement un nouveau régime (genre sans gluten, sans lactose,…)? Pour répondre à cette question, vous décidez d'aller de prendre conseil auprès de votre *tech lead*. Ce dernier vous conseille d'implémenter ce projet en utilisant un **pattern Visiteur (Visitor)** car

l'intention du pattern **Visiteur** n'est autre que
séparer un algorithme d'une structure de données
pour définir une nouvelle opération sans modifier
les classes sur lesquelles elle opère.

2. Pattern Visiteur à la rescousse !

Commencez par consulter un peu de documentation sur le pattern **Visiteur**.
Le diagramme de classes du pattern **Visiteur** est le suivant :



Les classes participantes à ce patron sont :
(extrait de http://www.goprod.bouhours.net/)
→ **`Visiteur`**
Déclare une opération **`visiter`** pour chaque classe concrète d'**`Element`** (**`ElementConcretUn`**, **`ElementConcretDeux`**,…) Le nom de l'opération et sa signature identifient la classe émettrice de la requête **`visiter`** vers le visiteur. Cela permet au visiteur de déterminer la classe concrète de l'élément visité. Le visiteur peut ensuite accéder à l'élément directement, au travers de son interface.
→ **`VisiteurConcretUn, VisiteurConcretDeux`**
Concrétise par codage chaque opération déclarée par la classe **`Visiteur`**. Chaque opération code un fragment de l'**algorithme** défini pour les classes d'objets qui lui correspondent dans la structure. Le visiteur concret fournit le contexte pour l'algorithme et mémorise son état local. Cet état représente souvent le cumul des résultats obtenus pendant le parcours de la structure.
→ **`Element`**
Définit une opération **`accepter`** qui prend pour argument un visiteur.
→ **`ElementConcretUn, ElementConcretDeux`**
Réalise le codage d'une opération `accepter` qui prend pour argument un visiteur.
→ **`Client`**
Peut énumérer ses éléments.
Peut fournir une interface de haut niveau permettant au visiteur de visiter ses éléments.
Peut être un composite ou un conteneur tel qu'une liste ou un ensemble.

Il ne reste plus qu'à adapter le diagramme de classes générique du **pattern Visiteur** à votre contexte, celui du vendeur de kébabs qui cherche à savoir si le kébab qu'il confectionne est végétarien et pescétarien.

Faites ci-dessous le **diagramme de classes du pattern Visiteur dans le contexte du vendeur de kébabs** c-a-d identifier les classes participantes à ce pattern en vous aidant des questions suivantes :
Dans le contexte du vendeur de kébabs, quels sont les éléments visités ?
Que recherchent à faire les visiteurs, qui sont-ils ?

→ la classe **`Element`** du pattern Visiteur correspond à la classe _____ dans le contexte du vendeur de kébab.

→ la classe **`ElementConcretUn`** du pattern Visiteur correspond à la classe _____ dans le contexte du vendeur de kébab.

→ la classe **`ElementConcretDeux`** du pattern Visiteur correspond à la classe _____ dans le contexte du vendeur de kébab.

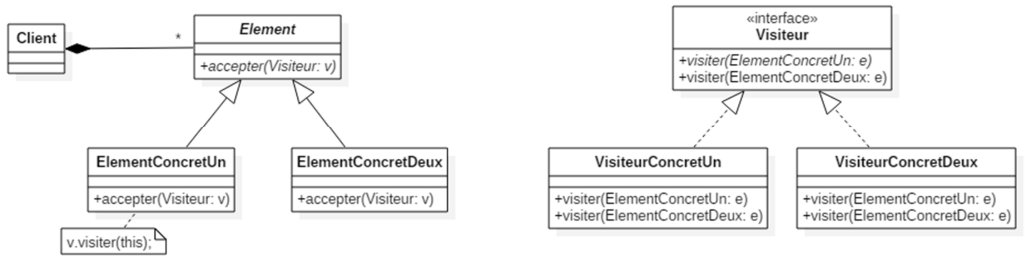
→ l'interface **`Visiteur`** du pattern Visiteur correspond à la classe _____ dans le contexte du vendeur de kébab.

→ la classe **`VisiteurConcretUn`** du pattern Visiteur correspond à la classe _____ dans le contexte du vendeur de kébab.

→ la classe **`VisiteurConcretDeux`** du pattern Visiteur correspond à la classe _____ dans le contexte du vendeur de kébab.

→ la classe **`Client`** du pattern Visiteur correspond à la classe _____ dans le contexte du vendeur de kébab.

Rappel du diagramme de classes générique pour le pattern Visiteur :



Remarque : si vous souhaitez en savoir plus le pattern Visiteur, rendez-vous dans le dépôt https://github.com/iblasquez/enseignement-iut-m3105-conception-avancee où vous retrouverez les travaux de vos collègues, ainsi que d'autres liens dans references_patterns.md.

Et maintenant, si on implémentait tout ça …
*Vous pouvez maintenant passer en **mode connecté** 😉*

3. Commençons par confectionner des kébabs …

3.1 Dans votre IDE préféré, créez un projet Maven que vous appellerez **kebab**.
N'oubliez pas de versionner votre projet (juste en local pour commencer, vous pousserez sur le distant uniquement en fin de séance)
et de commiter régulièrement au cours de cette séance !!

3.2 Confectionnons un premier kebab …

Pour confectionner un premier Kébab, nous allons commencer par écrire un premier test 😉

→ Dans **src/test/java**, créer une classe **KebabTest** où vous vous contenterez pour l'instant de **confectionner** un **Kebab** par un **Kebabier**, considérant ainsi que le kebabier joue le rôle de **monteur (builder) de Kebab .**

Commencez par écrire le début du test suivant :

```
@Test
public void un_kebab_contient_bien_tous_les_ingredients_ajoutes() {

      Kebab kebabAgneau = new Kebabier()
                  .avec(new Salade())
                  .avec(new Tomate())
                  .avec(new Oignon())
                  .avec(new Agneau())
                  .avec(new Pain())
                  .avec(new Sauce())
                  .confectionnerKebab();
}
```

→ En vous aidant de votre IDE, créer le plus rapidement possible dans **src/main/java** les classes et les méthodes qui permettent de **faire compiler ce code**.

Remarque : Veillez bien à ce qu'il n'y ait **qu'une seule méthode** avec dans la classe **Kebabier** qui reçoive en entrée un _____(revenez voir le diagramme de classes de la première question) et qui retourne un _____ : c-a-d n'oubliez pas de modifier le type `Object` généré automatiquement par votre IDE 😉

→ Une fois que ce code compile, ajouter à la méthode de test précédente l'assertion suivante :

```
      assertThat(kebabAgneau.listerLesIngredients()).containsExactly(
new Salade(),new Tomate(),new Oignon(),new Agneau(),new Pain(),new Sauce());
```

Remarque : Cette assertion est écrite avec **AssertJ**, n'oubliez pas d'ajouter sa dépendance dans le **pom.xml** (https://joel-costigliola.github.io/assertj/assertj-core-quick-start.html)

Faites compiler ce code, puis faire passer le test AU VERT.

Remarque : Veillez bien à ce que la méthode `listerLesIngredients` renvoie le bon type 😉
Pour faciliter la lecture de votre code, vous pouvez utiliser si vous le souhaitez le projet lombok (https://projectlombok.org/)😉

→ Pour améliorer la lisibilité du test, utilisez l'option **Extract Constant …** du menu **Refactor** de votre IDE pour transformer les **new** des ingrédients en constante afin d'obtenir le test suivant :

```
public void un_kebab_contient_bien_tous_les_ingredients_ajoutes() {

      Kebab kebabAgneau = new Kebabier()
                  .avec(SALADE)
                  .avec(TOMATE)
                  .avec(OIGNON)
                  .avec(AGNEAU)
                  .avec(PAIN)
                  .avec(SAUCE)
                  .confectionnerLeKebab();

      assertThat(kebabAgneau.listerLesIngredients()).containsExactly(
                  SALADE, TOMATE, OIGNON, AGNEAU, PAIN, SAUCE);
}
```

→ Complétez le test précédent avec la confection d'un kébab végétarien, par exemple de la forme :

```
    Kebab kebabVegetarien = new Kebabier()
                .avec(SALADE)
                .avec(TOMATE)
                .avec(OIGNON)
                .avec(PAIN)
                .avec(SAUCE)
                .confectionnerLeKebab();

      assertThat(kebabVegetarien.listerLesIngredients()).containsExactly(
                        SALADE, TOMATE, OIGNON, PAIN, SAUCE);
```

Faites passer le test AU VERT avec cette nouvelle assertion !!!

→ Déplacez la confection du `kebabAgneau` et du `kebabVegetarien` dans une méthode **confectionnerLesKekabs** qui ne sera autre que votre **@Before** afin que la méthode de tests `un_kebab_contient_bien_tous_les_ingredients_ajoutes` ne soit plus composée que par deux assertions…

→ Complétez ce premier test avec :
- ➔ Un **kebabCrevette** qui doit contenir un `Ingredient` de type `Crevette`. Vous avez le choix des autres ingrédients 😉
- ➔ Un **kebabThon** qui doit contenir un `Ingredient` de type `Thon`. Vous avez le choix des autres ingrédients 😉
- ➔ Faites en sorte qu'au moins un de ces deux kébabs ait également du `Fromage`…

Avant de continuer, assurez-vous que les quatre assertions de ce premier test passent bien AU VERT !!!

Commitez avec un message du genre :
« confection des kebabs : Kebab et ingredients »

Intéressons-nous maintenant aux régimes que respectent (ou pas) ces kébabs …

4. Le kébab confectionné suit-il le régime Végétarien ?

En vertu du principe ***petit pas par petit pas***, nous commencerons par nous focaliser uniquement sur le régime végétarien pour mettre en place le pattern Visiteur dans notre projet.

Commençons par rappeler la règle métier à implémenter :

> Le **végétarisme** est une pratique alimentaire
> qui exclut la consommation de chair animale.

Pour ***implémenter*** le pattern Visiteur ***en toute confiance***, rien de tel que de commencer par un fichier de tests qui permettra dans un premier temps de bien comprendre et spécifier la règle métier au travers d'exemples et dans un second temps de valider automatiquement votre implémentation 😉

4.1 Quelques tests pour commencer …

Vous allez donc commencer par implémenter dans votre fichier de tests **`KebabTest`**, **quatre nouveaux tests** portant sur les quatre kébabs confectionnés précédemment :

- ➔ un test devra être en mesure de vérifier que le **`kebabAgneau`** n'est pas végétarien
- ➔ un test devra être en mesure de vérifier que le **`kebabVegetarien`** est végétarien
- ➔ un test devra être en mesure de vérifier que le **`kebabCrevette`** n'est pas végétarien
- ➔ un test devra être en mesure de vérifier que le **`kebabThon`** n'est pas végétarien

A vous d'implémenter ces tests de la manière la plus explicite possible 😉
Faire compiler le code à partir des tests que vous venez d'écrire.
Les tests passent AU ROUGE, c'est normal, c'est votre *bonne* implémentation du pattern Visiteur qui leur permettra passer AU VERT 😉

4.2 Mise en place du pattern Visiteur dans le code de production …

Vous pouvez maintenant vous lancer dans l'implémentation de la branche **`VisiteurdeRegimeVegetarien`** en vous basant sur le diagramme de classes que vous avez réalisé dans le contexte du vendeur de kébabs dans la partie ***Pattern Visiteur*** *à la rescousse !*

Pour vous aider dans votre implémentation, vous pouvez vous aider de la Check List proposée sur le site (sourcemaking.com : extrait de https://sourcemaking.com/design_patterns/visitor) et de toute autre documentation sur le pattern Visitor que vous pourriez trouver en ligne 😉

Check list

1. Confirm that the current hierarchy (known as the Element hierarchy) will be fairly stable and that the public interface of these classes is sufficient for the access the Visitor classes will require. If these conditions are not met, then the Visitor pattern is not a good match.
2. Create a Visitor base class with a `visit(ElementXxx)` method for each Element derived type.
3. Add an `accept(Visitor)` method to the Element hierarchy. The implementation in each Element derived class is always the same – `accept( Visitor v ) { v.visit( this ); }`. Because of cyclic dependencies, the declaration of the Element and Visitor classes will need to be interleaved.
4. The Element hierarchy is coupled only to the Visitor base class, but the Visitor hierarchy is coupled to each Element derived class. If the stability of the Element hierarchy is low, and the stability of the Visitor hierarchy is high; consider swapping the 'roles' of the two hierarchies.
5. Create a Visitor derived class for each "operation" to be performed on Element objects. `visit()` implementations will rely on the Element's public interface.
6. The client creates Visitor objects and passes each to Element objects by calling `accept()`.

➔ **Le point 1** est vérifié : le diagramme de classes du Kebab et ses ingrédients a été réalisé à la première question de ce TD.

➔ Le **point 2** est la mise en place de **l'arborescence du Visiteur**.
- Pour répondre à ce point, créez dans un package **`visiteur`** (**`fr.unilim.iut.visiteur`**) l'arborescence du Visiteur limité pour l'instant à une interface **`VisiteurDeRegime`** et une classe **`VisiteurDeRegimeVegetarien`** avec pour commencer une méthode **`boolean visiter(Agneau ingredientAgneau)`** implémentée dans la classe **`VisiteurDeRegimeVegetarien.`** Que doit contenir l'implémentation de cette méthode ?
- Le point 2 indique « ***a visit(ElementXxx) method for each Element*** ». Compléter donc votre arborescence de manière à faire apparaître autant de méthodes **`visiter`** que d' actuelles classes concrètes d'ingrédients.

➔ Le **point 3** indique que le visiteur doit être transmis à chaque élément de la hiérarchie. Autrement dit, chaque élément concret *susceptible* d'être visité doit **accepter de recevoir un visiteur**.
Pour ce faire, chaque élément concret doit implémenter une méthode d'acceptation du visiteur (**`accepter`**) dans laquelle le visiteur va appeler la méthode spécifique qu'il doit **`visiter`** (via **`this`**).
(Rappelons qu'au point précédent, chaque visiteur a implémenté une méthode spécifique pour chaque type d'élément).

En vous aidant du diagramme de classes du **pattern Visiteur** et de ce qui précède, dans notre contexte, quelle devrait être la signature et l'implémentation de la méthode **`accepter`** qui prend en paramètre un `VisiteurDeRegime` ?

Implémentez la méthode `accepter` pour tous les éléments de la hiérarchie c-a-d pour toutes les classes concrètes de la hiérarchie `Ingredient`.
Dans la classe `Ingredient`, n'oubliez pas de déclarer `accepter` comme une méthode abstraite.

➔ Le **point 6** indique que c'est une classe **client** qui, pour implémenter le comportement souhaité, est chargée de **créer le visiteur** en rapport avec ce comportement et **de faire naviguer le visiteur d'éléments en éléments** en appelant **`accepter`**.
L'opération à implémenter est **`estVegetarien`** : il semblerait que le client soit directement le **`Kebab`** qui va être en mesure de guider le visiteur au travers de tous les ingrédients qui le compose.

Il ne vous reste donc plus **qu'à implémenter la méthode `estVegetarien`** dans la classe **`Kebab`** pour permettre à un visiteur de régime végétarien de visiter un à un chaque ingrédient de la liste des ingrédients. Bien évidemment, si le visiteur détecte qu'un ingrédient ne respecte pas le régime végétarien, la méthode **`estVegatarien`** devra renvoyer **`false`**.

Pour vérifier votre implémentation, exécutez les tests : ils doivent passer AU VERT !!!

Commitez avec un message du genre :
« régime végétarien implémenté avec le pattern visiteur » !

➔ Revenons maintenant sur le **point 4** qui était une remarque indiquant que :
- Chaque classe de la *hiérarchie des éléments* est seulement ***couplée avec une seule classe*** de la *hiérarchie visiteurs,* à savoir la classe mère (`VisiteurDeRegime`)

… alors que …

- Chaque classe de la *hiérarchie des visiteurs* ***est couplée à toutes les classes*** de la *hiérarchie des éléments* (11 méthodes `visiter`, chacune étant couplée à un élément de la *hiérarchie des éléments*)

➔ Le **point 5** indiquait que chaque nouvelle « opération » (service) nécessite l'implémentation d'une nouvelle classe dans la hiérarchie de visiteur.
Vous allez traiter ce point maintenant en mettant en place le service relatif au régime pescétarien dans le pattern Visiteur !

5. Le kébab confectionné suit-il le régime Pescétarien ?

Avant de continuer, deux petites questions :
➔ Pour une meilleure cohérence de votre code, avez-vous pensé à regrouper tout ce qui concerne les ingrédients dans un package `ingredients` (`fr.unilim.iut.kebab.ingredients`) ?
➔ Pour une meilleure cohérence de votre code, avez-vous pensé à regrouper tout ce qui concerne les visiteurs dans un package `visiteur`?

Commençons par rappeler la nouvelle règle métier à implémenter :

> Le **pescétarisme**, ou pesco-végétarisme, est un néologisme désignant le régime alimentaire d'une personne omnivore qui s'abstient de consommer de la chair animale à l'exception de celle issue des poissons, des crustacés et mollusques aquatiques
> **Remarque** : **Pescétarien** = végétarien + poissons + **crevettes**

5.1 Quelques tests pour implémenter le pattern en toute confiance …

Vous allez donc commencer par implémenter dans votre fichier de tests **`KebabTest`**, **quatre nouveaux tests** portant sur les quatre kébabs confectionnés précédemment :

➔ un test devra être en mesure de vérifier que le **`kebabAgneau`** n'est pas pescétarien
➔ un test devra être en mesure de vérifier que le **`kebabVegetarien`** est pescétarien
➔ un test devra être en mesure de vérifier que le **`kebabCrevette`** est pescétarien
➔ un test devra être en mesure de vérifier que le **`kebabThon`** est pescétarien

A vous d'implémenter ces tests de la manière la plus explicite possible 😉
Faire compiler le code à partir des tests que vous venez d'écrire.
Les tests passent AU ROUGE, c'est normal, c'est votre *bonne* implémentation du pattern Visiteur qui leur permettra passer AU VERT 😉

5.2 Mise en place du pattern Visiteur dans le code de production …

En vous inspirant de ce qui a été fait précédemment et du diagramme de classes, vous pouvez maintenant vous lancer dans l'implémentation du régime pescetarien en faisant en sorte que le `Kebab` propose un service relatif au respect du régime pescétarien (**`estPescetarien`**) via un visiteur.

Pour vérifier votre implémentation, exécutez les tests : ils doivent passer AU VERT !!!

Commitez avec un message du genre :
« régime pescétarien implémenté avec le pattern visiteur » !

6. Diagramme de classes du pattern Visiteur dans notre contexte

Utilisez votre IDE pour générer automatiquement un diagramme de classes à partir du code que vous venez d'écrire. Vérifiez que ce diagramme de classes est bien cohérent avec le diagramme de classes du pattern **Visitor** de la question 2.1.

7. Mise en place d'un nouveau régime…

Il est désormais facile de satisfaire le nouveau besoin du vendeur de kébabs qui souhaiterait disposer d'une nouvelle fonctionnalité pour savoir si un kebab respecte un nouveau régime, comme par exemple le **régime sans gluten**.
Implémentez au sein de votre architecture, le **nouveau service permettant de savoir si un Kebab est sans gluten ou pas…**
Remarque : L'**ajout d'un nouvel ingrédient** **`GaletteDeSarrasin`** (par exemple) vous est indispensable pour créer un kébab sans Gluten 😉
Le comportement du nouveau régime doit bien sûr être **couvert par les tests !!!**

Remarque : nous vous avons suggéré d'implémenter le régime sans gluten, mais si vous avez envie d'implémenter un autre régime, sans lactose ou autre, faites-vous plaisir 😉

Commitez avec un message du genre :
« Ajout du nouveau régime sans gluten » !

8. Remarques

8.1 A propos du pattern Visiteur ….

Le visiteur permet de parcourir une série d'éléments et d'invoquer, pour chacun, des comportements déterminés.

D'après Tête la Première :
Utilisez le Visiteur quand vous voulez ajouter des capacités à un ensemble composite d'objets et que l'encapsulation n'est pas importante…

Le visiteur doit parcourir chaque élément du composite : cette fonctionnalité se trouve dans un objet navigateur. Le Visiteur est guidé par le navigateur et recueille l'état de tous les objets du composite.
Une fois l'état recueilli le client peut demander au visiteur d'exécuter différentes opérations sur celui-ci. Quand une nouvelle fonctionnalité est requise seul le visiteur doit être modifié.

Avantages :
- ➔ Permet **d'ajouter des opérations à la structure d'un Composite** sans modifier la structure elle-même.
- ➔ L'ajout de nouvelles opérations est relativement facile
- ➔ Le code des opérations exécutées par le visiteur est centralisé

Inconvénients :
- ➔ L'encapsulation des classes du composite est brisée (distribution des traitements dans la hiérarchie des visiteurs)
- ➔ Comme une fonction de navigation est impliquée, **les modifications de la structure du composite sont plus difficiles** (ajout de types)

8.2 Dans le contexte de notre vendeur de kebabs …

Les **patterns Decorator** (et Composite) sont des patterns de structuration qui seraient adapté si de nombreux ingrédients devraient être ajoutés.
En effet, ajouter une nouvelle classe à de tels patterns a un coût négligeable.
Or dans un magasin de kebabs, la liste des ingrédients est assez figée : une fois définie elle restera sensiblement la même. Ce sont les régimes qui vont être amenés à évoluer c-a-d qu'une extension de l'application portera plutôt sur l'ajout d'une fonctionnalité (`estxXX`) que sur l'ajout d'un type d'un nouvel `Ingredient`.
Ajouter un comportement (fonctionnalité) est couteux dans un pattern de structuration, ce type de pattern n'était donc pas adapté à notre problème.
Le pattern Visiteur est adapté car il permet d'ajouter facilement du comportement et ce n'est pas un hasard s'il fait partie des **patterns de comportement**.
Cet exemple montre qu'un pattern s'adresse à un problème en particulier et choisir un pattern demande de bien s'interroger sur son contexte…
Hésiter entre un **pattern Decorator/Composite et un pattern Visiteur** revient à faire une choix entre **ajouter *facilement* un type** (structure) **vs ajouter *facilement* une fonctionnalité** (comportement)…

Avant de se lancer dans une quelconque implémentation,
il faut toujours prendre le temps d'une réflexion sur le design
et le design va dépendre du contexte 😉 !

8.3 De manière plus générale :
Pattern de structuration vs Pattern de comportement (composition vs delegation) :

➔ **L'objectif des patterns de structuration** est de **faciliter l'indépendance de l'interface d'un objet ou d'un ensemble d'objet vis-à-vis de son implémentation**.[…]
En fournissant des **interfaces**, les patterns de structuration **encapsulent la composition des objets** et augmentent le niveau d'abstraction du système à l'image des patterns de créations qui encapsulent la création des objets.[…]
L'encapsulation de la composition est réalisée non pas en structurant l'objet lui-même mais en transférant cette structuration à un second objet (lié au premier).
(Extraits de Design Pattern pour Java de Laurent DEBRAUWER)

➔ **L'objectif des patterns de comportement** est de fournir des solutions pour **distribuer les traitements et les algorithmes entre les objets.**
La distribution peut se faire :
- Soit **par délégation** (comme dans le cas du pattern Visitor) où les traitements sont distribués dans des classes indépendantes
- Soit **par héritage** (comme dans le cas du pattern Template Method) où un traitement est réparti dans des sous-classes.

(Extraits de Design Pattern pour Java de Laurent DEBRAUWER)