M3105 – TD : Un Kata Kebab à notre sauce
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 : Laitue, roquette, tomate, oignons, 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 le dire d'il est végétarien ou non, s'il est pescétarien ou non… »

Remarques :

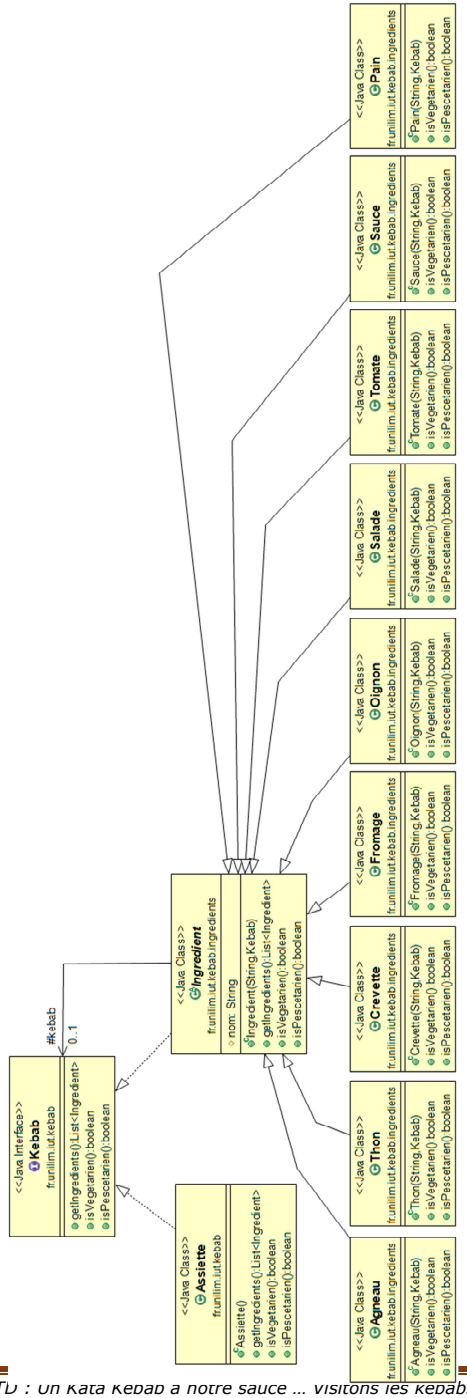
- D'après Wikipédia :
  - ➔ Le **végétarisme** est une pratique alimentaire qui exclut la consommation de chair animale.
  - ➔ Le **pescétarisme**, ou pesco-végétarisme, est un néologisme désignant le régime alimentaire d'une personne omnivore qui s'abstient de consommer de la chair animale à l'exception 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 ?)**

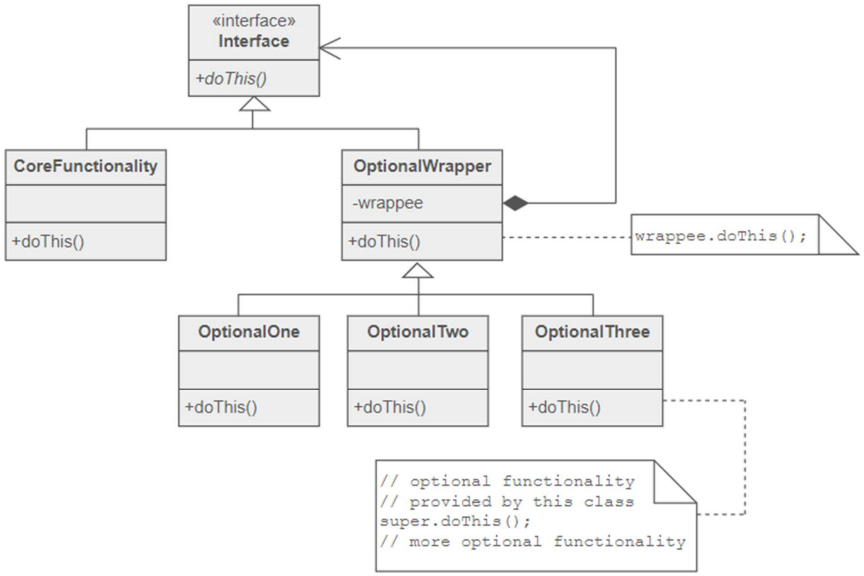
Quid de votre implémentation ?

Précédemment, vous avez suivi les conseils d'un architecte qui vous a demandé d'implémenter votre application en vous appuyant sur un **pattern Decorator**, ce qui vous a amené à concevoir votre projet de la manière suivante :



1. A propos du pattern **Decorator** …

Rappelons pour commencer le diagramme de classes du pattern **Decorator** :
(extrait de sourcemaking.com : https://sourcemaking.com/design_patterns/decorator)



1.a Retrouvez-vous facilement le diagramme de classes du pattern **Decorator** dans le contexte de kebabs de notre application ?

1.b ***En mode déconnecté*** (sur *une feuille de papier* 😉), ajoutez sur le diagramme de classes de notre application, un nouvel ingrédient : du **`Boeuf`** par exemple 😉.

1.c Sous votre IDE préféré, créez un nouveau projet maven **kebabvisitor** dans lequel vous allez importer le code lié à cette implémentation. Ce code est disponible sous \\ubox.unilim.fr\pedago-iut\info dans le répertoire M3105.
Faites compiler ce code et exécutez les tests : ils doivent passer AU VERT !!!

Remarque : Les assertions ont été écrites avec le framework AssertJ qui permet d'améliorer la lisibilité des assertions et de faciliter les tests sur les collections. Pour pouvoir utiliser AssertJ dans votre projet maven, n'oubliez pas de rajouter la dépendance suivante dans votre **`pom.xml`** :

```
<dependency>
      <groupId>org.assertj</groupId>
      <artifactId>assertj-core</artifactId>
      <version>3.2.0</version>
      <scope>test</scope>
</dependency>
```

Pour en savoir plus sur AssertJ : http://joel-costigliola.github.io/assertj/

Avant de continuer, jetez un coup d'œil attentif au code …

Aviez-vous remarqué qu'en utilisant un tel pattern,
il est possible d'écrire du code « sans if et ni for »
(réduisant ainsi ici fortement la complexité cyclomatique,
donc améliorant la qualité de code 😉)

1.d Ajoutez dans votre projet le nouvel ingrédient **`Bœuf`**.
Le bœuf est une chair animale : cet ingrédient est donc à bannir aussi bien d'un régime végétarien que d'un régime pescetarien.

1.e Pour vérifier la « bonne » implémentation de votre nouvel ingrédient, vous allez maintenant ajouter quelques tests …
Rendez-vous dans le fichier **`KebaTest.java`** :

- ➔ Déclarez et instanciez au sein de la méthode `setUp` un nouveau kebab du genre **`kebabBoeuf`** qui prendra entre autres comme ingrédient un `Bœuf` qui pourra par exemple porté le `nom` de « **`bœuf du Limousin`** ».
- ➔ Rajoutez dans le test **`un_kebab_contient_bien_les_noms_de_tous_les_ingredients_ajoutes`** un **`assertThat`** qui permet de vérifier ce que contient le **`kebabBoeuf.`**
- ➔ Enfin pour vérifier les régimes ajoutez les deux tests suivants :
  `isVegetarien_devrait_retourner_faux_pour_kebabBoeuf`
  `isPescetarien_devrait_retourner_faux_pour_kebabBoeuf`

1.f L'intérêt de mettre en œuvre un design pattern au sein de votre conception est de faciliter la maintenabilité du code en permettant à ce dernier d'être facilement modifiable et extensible.

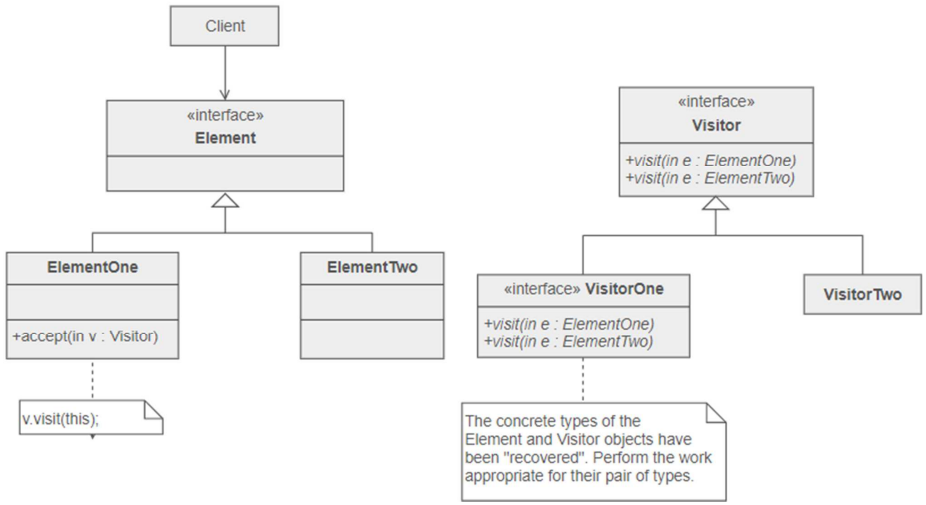
- ➔ L'ajout d'un nouvel ingrédient vous a-t-il semblé facile avec ce pattern ?
- ➔ Le client souhaite maintenant ajouter un nouveau régime et aimerait bien savoir si un kebab est sans gluten.
  Expliquez (ne touchez pas à votre code pour le moment), comment à partir de votre conception, vous devez modifiez votre code pour mettre en place ce nouveau régime.
  Trouvez-vous qu'il serait facile d'ajouter de nombreux autres régimes avec un tel pattern ?...

Heureusement, le pattern **Visitor** va vous faciliter la tâche !

Avant de continuer, placez votre projet sous git (si ce n'est déjà fait) et
commitez avec un message du genre :
« decorator 2 régimes et 10 ingredients 1 assiette» !

2. **Pattern Visitor** à la rescousse !

2.1 Commencez par consulter un peu de documentation sur le pattern **Visitor**.
Le diagramme de classes du pattern **Visitor** est le suivant :
(extrait de sourcemaking.com : https://sourcemaking.com/design_patterns/visitor)



Rendez-vous ensuite 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)

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

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

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

2.2 ***En mode déconnecté*** (sur *une feuille de papier* 😉), commencez par proposer une ébauche du diagramme de classes mettant en œuvre le pattern **Visitor** sur le diagramme de classes actuel (limité pour le moment aux fonctionnalités `isVegetarien` et `isPescetarien`).

Pour vous aider à construire ce diagramme, vous pouvez vous aider de la Check List proposée sur le site (sourcemaking.com : extrait de https://sourcemaking.com/design_patterns/visitor)

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()`.

2.**3 Il est temps de refactorer !** Pour cela, ***passez en mode connecté*** sous votre IDE préféré 😉
Vous n'allez pas tout refactorer d'un coup, mais vous allez refactorer petit pas par petit pas, un régime après l'autre, en exécutant les tests aussi souvent que possible (vive Infinitest !)
Pour refactorer, nous allons reprendre les différents point de la check list précédente …

➔ **Le point 1** est vérifié, d'autant plus que le diagramme de classes actuel est un décorator (un composite dégénéré en quelques sortes…)

Pour commencer, le refactor ne portera que sur le régime végétarien !

➔ Le **point 2** est la mise en place de **l'arborescence du Visiteur**.
- Pour répondre à ce point, créer dans un package **`visitor`** (**`fr.unilim.iut.visitor`**) l'arborescence du Visiteur limité pour l'instant à une interface **`VisiteurDeRegime`** et une classe **`VisiteurDeRegimeVegetarien`** avec pour commencer une méthode `boolean visit(Agneau ingredientAgneau)` implémentée dans la classe **`VisiteurDeRegimeVegetarien`.** Que doit contenir l'implémentation de cette méthode ?
- Exécutez les tests qui doivent toujours passer au VERT puisque nous n'avons pas modifié le comportement existant, mais juste ajouter un peu de code…
- Le point 2 indique « ***a visit(ElementXxx) method for each Element*** ». Compléter donc votre arborescence de manière à faire apparaître autant de méthodes **`visit`** que de classes concrètes actuelles.
  Au total, vous devrez donc avoir implémenté 11 méthodes **`visit`** (10 ingrédients et 1 assiette)
- Exécutez les tests qui doivent toujours passer au VERT puisque rien n'a été modifié …

Commitez avec un message du genre :
« mise en place de l'arborescence du visiteur de régime végétarien » !

➔ 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 (**`accept`**) dans laquelle le visiteur va appeler la méthode spécifique qu'il doit **`visit`**er (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 Visitor** et de ce qui précède, dans notre contexte, quelle devrait être la signature et l'implémentation de la méthode **`accept`** qui prend en paramètre un `VisiteurDeRegime` ?

Implémentez la méthode `accept` pour tous les éléments de la hiérarchie c-a-d pour toutes les classes concrètes de la hiérarchie `Ingredient` et pour l'`Assiette`.
Dans la classe `Ingredient`, déclarer `accept` comme une méthode abstraite.

Exécutez les tests qui doivent toujours passer au VERT
puisque le comportement existant n'a été modifié …

Commitez avec un message du genre :
« acceptation du visiteur de régime végétarien auprès des éléments » !

Suite au point 3, l'opération qui consiste à savoir si un ingrédient suit ou non le régime végétarien est désormais déléguée au visiteur (Rappelons que l'objectif du pattern **`Visitor`** est de séparer un algorithme d'une structure de données).
Les méthodes **`isVegetarien`** n'ont donc plus lieu d'exister dans les éléments.

Supprimez donc toutes les méthodes `isVegetarien`
des classes qui `accept`ent de recevoir un visiteur 😉
(hormis la méthode **`isVegetarien`** de la classe **`Assiette`** pour pouvoir compiler !)

Seule la classe **`Ingredient`** dispose désormais d'une méthode **`isVegetarien`**…
… et cette méthode n'a plus lieu d'être puisque le ***polymorphisme*** n'est plus d'actualité dans ce pattern : les fonctionnalités étant désormais ***déléguées au visiteur***.

Vous pouvez donc également supprimer la méthode `isVegetarien` de la classe `Ingredient`

➔ 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 **`accept`**.
L'opération à implémenter est **`isVegetarien`** : 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.
L'implémentation de **`isVegetarien`** (pour ce contexte) se fera donc dans le **`Kebab`**.

Commencez donc par transformer l'interface `Kebab` en classe abstraite
… et corriger toutes les erreurs de compilation qui peuvent apparaître 😉 …

Il ne vous reste plus **qu'à implémenter la méthode `isVegetarien`** de la classe **`Kebab`**
pour permettre à un visiteur de régime végétarien
de visiter un à un chaque ingrédient de la liste des ingrédients
(utiliser la méthode `getIngredients` pour récupérer la liste d'ingrédients
que vous pourrez ensuite parcourir à l'aide d'un *for each*)
Bien évidemment, si le visiteur détecte qu'un ingrédient ne respecte pas le régime végétarien,
la méthode **`isVegatarien`** devra renvoyer **`false`**.

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

Remarques sur la classe **`Assiette`** :
- En réalité seuls les ingrédients peuvent avoir une influence sur le régime, pas l'assiette ! Il n'y a donc aucun intérêt à faire visiter l'**`Assiette`**.
  Vous pouvez donc supprimer maintenant la méthode `accept` de la classe `Assiette` et la méthode `visit(Assiette assiette)` de `VisiteurDeRegime` et de `VisiteurDeRegimeVegetarien`

  Pour vérifier que cette suppression n'a pas eu d'impact sur votre comportement,
  exécutez les tests : ils doivent passer AU VERT !!!

- La méthode **`isVegetarien`** étant directement implémentée dans la classe mère **`Kebab`** et ne concernant les ingrédients peut également être supprimée de la classe **`Assiette`** !

  Pour vérifier que cette suppression n'a pas eu d'impact sur votre comportement,
  exécutez les tests : ils doivent passer AU VERT !!!

Commitez avec un message du genre :
« isVegetarien desormais traité avec un 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 `visit`, 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 pescetarien dans le pattern Visitor !

2.3 Mise en place du régime pescétarien.

Le refactor porte donc maintenant sur le régime pescétarien !...
A vous de jouer !

En vous inspirant de ce qui a été fait précédemment, faites apparaître le régime pescetarien dans votre hiérarchie de visiteur et implémenter le service relatif au respect du régime pescétarien (**`isPescetarien`**) via un visiteur.

Pour vérifier votre implémentation, exécutez les tests : ils doivent passer AU VERT !!!
(N'oubliez pas de supprimer toute méthode inutile 😉)

Commitez avec un message du genre :
« isPescetarien desormais traité avec un visiteur » !

2.4 Diagramme de classes du pattern Visitor appliqué à notre contexte de kebab :
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.

2.5 Mise en place d'un nouveau régime…
Après ce refactoring, il est facile de satisfaire le nouveau besoin du client 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 suggérer d'implémenter le régime sans gluten, mais si vous avez envie d'implémenter un autre régime, faites-vous plaisir 😉

Commitez avec un message du genre :
« Ajout du nouveau régime Sans Gluten » !

2.6 Remarques :

➔ **2.6.1 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)

➔ **2.6.2 Dans le contexte de notre magasin 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 (`isxXX`) 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 (l'architecte avait tort 😉)

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 et un pattern Visiteur** revient à faire une chois 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 😉 !

➔ **2.6.3 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)