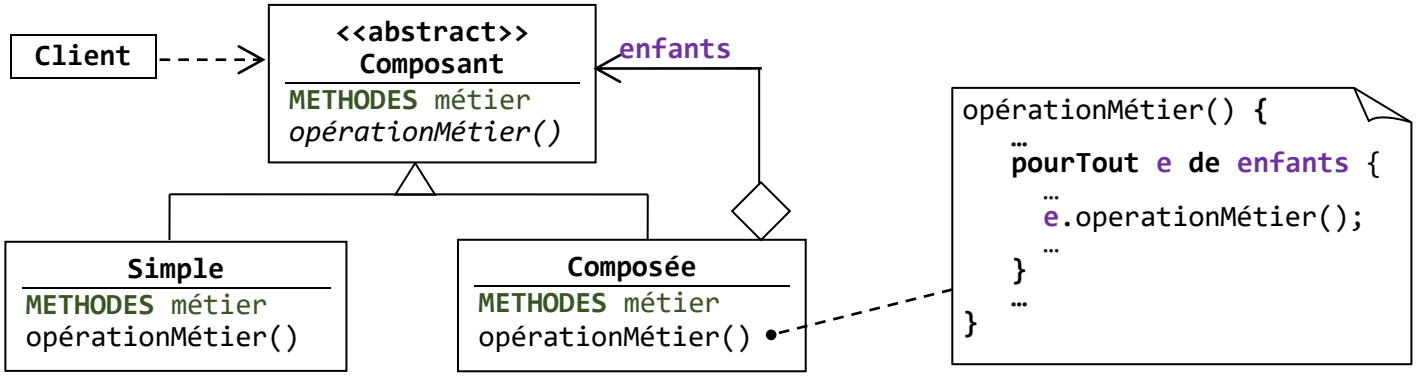
TD-TP : Le Design pattern « Composite »

1. IDEE DE BASE DU DESIGN PATTERN « Composite »

Le patron de conception « Composite » permet de composer des hiérarchies d'objets en structures arborescentes. Il permet au client de traiter de la même et unique façon tous les *`Composant`* de la structure arborescente, qu'ils soient des composants `Simple` (cf. les feuilles de l'arborescence) ou bien des objets `Composés` d'autre composants (cf. les nœuds de l'arborescence).



Constituants :

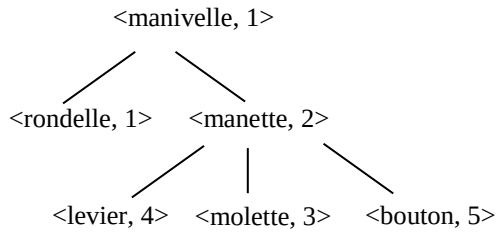
- *`Composant`* : *Classe abstraite* qui déclare l'interface métier (cf. *`opérationMétier()`*) des objets qui entrent dans la composition de l'arborescence, que ce soit en tant que feuille (cf. `Simple`) ou en tant que nœud (cf. `Composé`).
- `Simple` : Représente les objets feuille dans l'arborescence. Une feuille n'a pas d'enfant. Elle implémente le comportement métier (cf. `opérationMétier()`) d'un objet élémentaire de la composition.
- `Composé` : Représente les objets nœud dans l'arborescence. Un `Composé` stocke les composants enfants qui le composent. Un `Composé` sollicite le comportement métier de ses enfants pour réaliser son propre comportement métier (cf. `opérationMétier()`).
- `Client` : Utilise et manipule les *`Composant`* métier de la hiérarchie à l'aide de l'interface définie dans la super-classe *`Composant`*.

Collaboration : Un client utilise l'interface de la classe *`Composant`* pour manipuler indifféremment les objets de la structure composite (i.e. arborescente hiérarchique). Si l'objet manipulé est `Simple`, l'opération métier est traitée directement. Si l'objet manipulé est `Composé`, il sollicite l'opération métier de l'ensemble de ses composants, en effectuant éventuellement des opérations supplémentaires pour combiner les résultats obtenus de ses composants.

2. EXEMPLE DE SITUATION D'USAGE DU DESIGN PATTERN « Composite »

Considérons qu'une pièce de voiture (caractérisée par un `libellé` et un `prix` entier) est soit simple, soit composée. Dans cette intention apparaitront les classes *`PièceComposant`* et les sous-classes `PièceSimple` et `PièceComposée` spécifiques à notre problème. Cette représentation permettra d'assembler des pièces selon une représentation arborescente, typique du patron Composite.

Par exemple, dans l'arborescence de combinaison de pièces représenté ci-contre, `manette` et `manivelle` sont des `PièceComposée` alors que `rondelle`, `levier`, `molette` et `bouton` sont des `PièceSimple`.



Le prix total de `manette` est égal à `14`, et celui de `manivelle` est égal à `16`, à savoir la somme des prix de ses composants (ses `enfants`) + le prix de l'assemblage défini au niveau de la `manette` (cf. `2`) et de la `manivelle` (cf. `1`).

A la lueur de cet exemple, l'idée forte est de bien comprendre, ce que signifie la propriété rappelée ci-contre pour les opérationMétier(), mise en évidence dans le DesignPattern « Composite » de la section précédente, et de bien comprendre comment cette propriété serait implémentée pour cet exemple.

```
opérationMétier() {
   …
   pourTout e de enfants {
     …
     e.operationMétier();
     …
   }
   …
}
```

Dans cet exemple, `int getPrix()` est une `opérationMétier()` du Composite.

Travail à faire

1. Donnez la représentation UML des classes nommées dans cet exemple.
2. Ajoutez à votre schéma les objets `levier`, `molette` et `bouton` avec leur attribut `prix`, puis ajoutez l'objet `manette` avec son `prix` d'assemblage, de même que l'attribut `enfants` en charge de mémoriser les adresses de ses *`PièceComposant`*.
3. Donnez le code de l'opération métier `int getPrix()`, pour chacune des 3 classes de cet exemple.

3. LE DESIGN PATTERN AU COMPLET : 2 approches

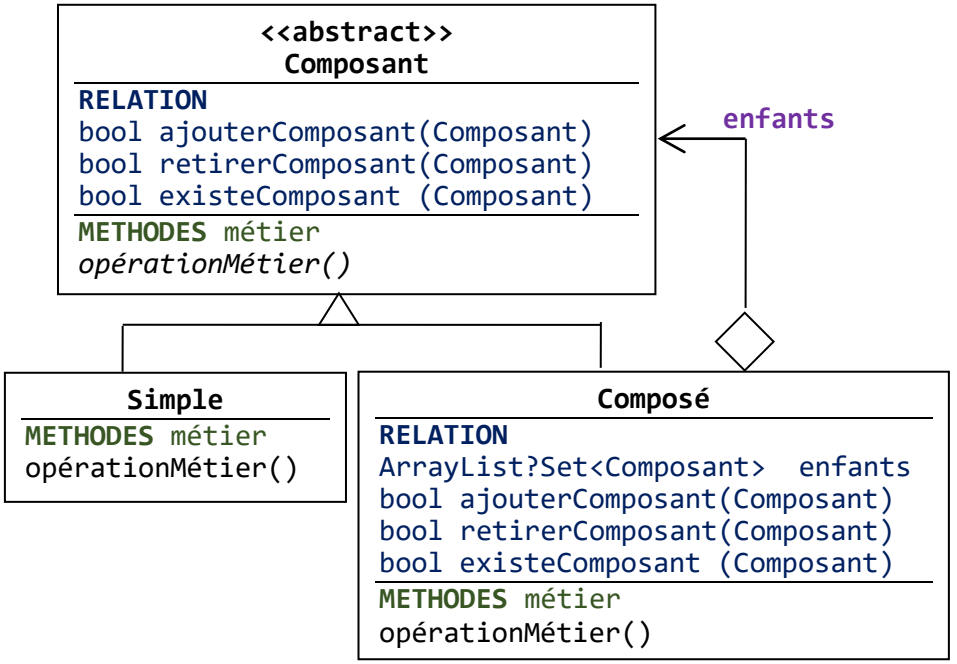
Selon les auteurs, deux approches sont proposées pour implémenter la relation `enfants` au choix avec `Set`, ou `ArrayList`, ou …

La première approche, préconise de définir les méthodes `existe/ajouter/retirerComposant` dans la super-classe *`Composant`*, par exemple avec un comportement naïf (cf. retourner faux), et de les spécialiser dans la classe `Composé`.

Ainsi, tous les objets des sous-classe de *`Composant`* ont la même interface.

Le problème est que ces méthodes sont non pertinentes (cf. sans intérêt) pour les objets de la classe `Simple`.

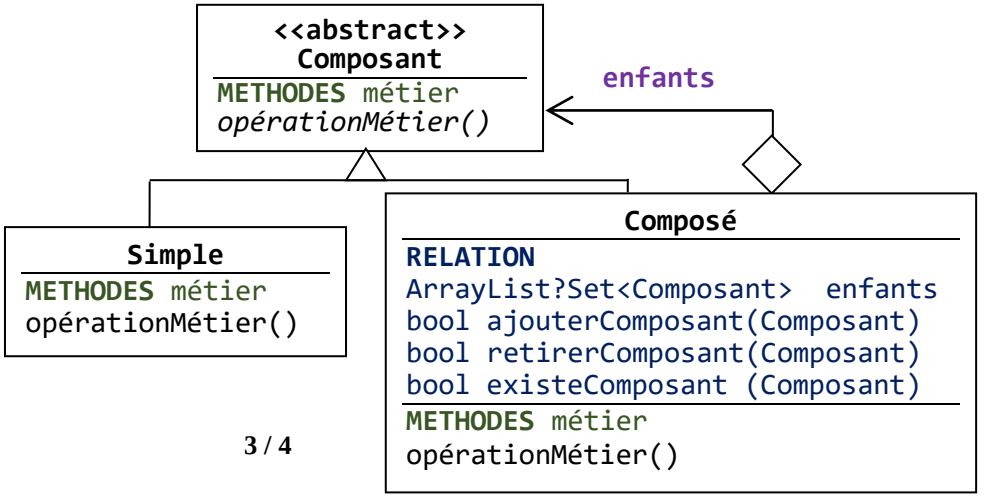
Dans cette approche, certains développeurs prévoient une méthode abstraite *`bool estComposé()`* dans *`Composant`*, à spécialiser dans les sous-classes. Retourne `false` dans `Simple` et `true` dans `Composé`.



La seconde approche, préconise de définir les méthodes `existe/ajouter/retirerComposant` uniquement dans la classe `Composé`.

Ainsi, l'interface pour tous les objets de la hiérarchie de classes de racine *`Composant`* est limitée aux comportements métier.

Le problème est que le développeur ne peut invoquer les méthodes `existe/ajouter/retirerComposant`, indifféremment, pour tout objet des sous-classes de *`Composant`*.



4. APPLIQUER LE DESIGN PATTERN « Composite »

Vous allez produire un exemple concret. Dans cet exemple, les composants sont *PréparationDeCuisine* avec des `PrépartionSimple` et des `PréparationComposée`. Toute *PréparationDeCuisine* est caractérisée par un `libellé` et un `prix`.

Le `prix` représente :

- le prix de revient, pour une `PrépartionSimple`,
- alors qu'ils ne représente que le cout de préparation, pour une `PréparationComposée`, le prix de revient devant intégrer en plus, le prix de revient de chacun de ses composants.

La méthode `getPrix()` retourne le prix de revient de la préparation considérée.

Travail à faire

Il s'agit d'implémenter les deux approches énoncées dans la section **3. LE DESIGN PATTERN AU COMPLET**. Pour cela :

4. Chaque version sera développée dans un projet Eclipse différent. Le projet `5.Composite_1_PlatCuisine` pour la première approche et le projet `5.Composite_2_PlatCuisine` pour la seconde approche.

5. Chacun des deux projets comportera une classe `TesteComposite` dans laquelle le `main()` crée une `PréparationComposée` nommée `patesBolo`, qui résulte d'une `PrépartionSimple` nommée `pate` et une `PréparationComposée` nommée `sauceBolo`, elle-même composée d'une `PrépartionSimple` nommée `viandeHachée` et une `PréparationComposée` nommée `sauceTomate`, composée à son tour de deux `PrépartionSimple` que sont `tomateCuite` d'une part et d'un `assaisonnement` d'autre part.

6. Vous attribuerez un prix et un libellé à chaque objet lors de sa création.

7. Les méthodes `getPrix` et `toString()`seront les méthodes métier et vous demanderez non seulement le prix de revient de `patesBolo` mais également de s'afficher via `toString()`.

8. Vous utiliserez le conteneur `Set` pour implémenter la relation `lesEnfants`, histoire de vous frotter à une autre structure générique de `Java`, qui soit différente de `ArrayList`.