Comment et Pourquoi refactorer son code
Cahier de TPs

98 avenue du Général Leclerc
92100 Boulogne-Billancourt
Tél : (+33)1 55 60 76 72 Fax : (+33)1 55 60 76 73
formation@oxiane.com
www.oxiane.com

Table des matières

Préparation avant démo

- Ouvrir IntelliJ
- Ouvrir le projet Vistamboire
- Dans le terminal : `. setJava17`
- Dans le terminal : `git co demo1`
- `mvn clean`
- Ouvrir un onglet de Chrome sur `http://localhost:8080/swagger-ui/index.html`

Démo 1

Montrer la classe `Facture` avec la méthode `calculate`, et la classe `FactureController`

Démo 2

```
git sw demo
```

Montrer la classe `FactureController` et la méthode `createFacture` que s'est compléxifiée.
Montrer la classe `PrixUnitCalculateurImpl` qui est nouvelle et qui contient du code de calcul.

Démo 3

Objectif : rendre le code lisible et compréhensible

> Dans `FactureController`, `databaseValuesExtractor` est déclaré de type `DatabaseValuesExtractorImpl`. C'est une faute. On a une interface `DatabaseValuesExtractor` et il est plus prudent d'injecter une interface plutôt qu'une implémentation. Cela simplifiera l'écriture des tests.
  - Changement du type de donnée de l'attribut. On ne prend aucun risque. D'autant qu'on a qu'une seule classe d'implémentation.

> Dans la classe `FactureController`, la méthode `createFacture(...)` contient beaucoup de code, et les actions métier ne sont pas clairement identifiées. Le code n'est pas lisible. On va essayer de rendre ce code lisible.

> Il n'y a aucun test pour la classe `FactureController`, donc on ne peut pas commencer le refactoring. Il faut créer des tests. J'ai déjà avancé le travail, le sujet de ce séminaire n'est pas comment écrire les tests unitaires, mais le refactoring. Donc j'ai les tests qui sont écrits, je copie / colle
  - Créer la classe de test `FactureControllerTest`
  - Exécuter la commande suivante : `./demo6-step1`
  - Sélectionner toute la classe et coller
  - Exécuter les tests unitaire avec le code-coverage et montrer ce qu'on a : toute la méthode `createFacture()` est testée.

> Commenter la ligne `vistamboire.setPrixUnitaire(...)` et relancer les tests
  - Tous les tests passent
  - C'est donc que les tests ne sont pas correctement écrits
  - Décommenter la ligne `vistamboire.setPrixUnitaire...`
  - lancer `pitest.sh` et afficher le rapport directement dans IntelliJ
  - Il y a 3 lignes de code qui posent problème : quand pitest modifie le code, les tests unitaires ne plantent pas.
  - Il faut donc mieux tester ces lignes, en écrivant de nouveaux tests
  - Exécuter la commande suivante : `./demo6-step2`
  - Sélectionner tout le code de `FactureControllerTest` et coller
  - Relancer les tests unitaires
  - Relancer pitest

> Maintenant le code est entièrement couvert par des tests unitaires, on peut se lancer dans le refactoring.
  - Première étape, on va sortir de la méthode `createFacture()` la récupération du client. On utilise pour cela l'opération de refactoring `ExtractMethod`, et on nomme cette nouvelle méthode `findClientById`.
  - Exécution des tests unitaires

> Ensuite, on s'intéresse à la variable locale `qteDejaAchetee`. Celle-ci est utilisée uniquement deux lignes plus bas. On va donc regrouper cette variable de son lieu d'utilisation avec l'opération de refactoring `Slide Statements`. Autant l'extraction d'une méthode, quand elle est réalisée par l'IDE est sans grand risque. Autant `Slide Statements` peut aisément avoir de gros impacts sur le comportement du code, et il est impératif de ré-exécuter les tests unitaires après ce déplacement. Slide Statements s'utilise avec ceinture et bretelles.

> On a maintenant 2 lignes qui permettent de vistamboire qui sera utilisé ensuite. Ces deux lignes constituent en fait une unique opération qui consiste à récupérer un vistamboire correctement initialisé. On applique encore `Extract Method` et on introduit une nouvelle méthode `getVistambboireForFacture()`

> Cette nouvelle méthode a deux paramètre, `facture` et `client`. Hors, le client est déjà un membre de la facture il n'est pas nécessaire d'avoir ces deux paramètres.
- dans l'appel de la méthode `calculatePrixUnit`, on remplace `client` par `facture.getClient()`.
- le paramètre client n'est plus utilisé dans la méthode, on peut le supprimer. Cette opération de refactoring s'appelle `Change Method Signature` et mon IDE me propose un assistant que je vais utiliser : ⌘+F6. Cet assistant a le bon goût de modifier aussi tous les appels à cette méthode.
- exécution des tests unitaires

> La variable `qteDejaAchetee` n'étant utilisée qu'à un seul endroit, elle n'est en fait pas nécessaire. Je peux remplacer son utilisation par l'expression qui permet de l'initialiser. Cette opération de refactoring s'appelle `Inline Variable`, et mon IDE me propose un assistant pour le faire : ⌥⌘N. Ca donne un truc pas très beau, mais j'ai une idée derrière la tête...

> Cette longue instruction, son unique fonction, c'est de calculer la remise client pour la facture. J'extraie le contenu avec `Extract Method` ^⌘M pour créer une `calculateRemiseClientForFacture(...)`
- exécution des tests unitaires

> Pour éclaircir la ligne, je peux extraire la variable `quantiteDejaCommandeeCetteAnnee`

> Ensuite, je rationalise les paramètres de la méthode, et il ne me reste plus qu'un seul paramètre : `facture`.

> Enfin, j'indente les paramètres des deux appels de fonction, pour gagner en lisibilité
- Exécution des tests unitaires

> Retour dans la méthode `createFacture()`, le `facture.calculate(vistamboire)` est présent 2 fois, je n'y peux rien, mais le premier est au milieu d'un bloc qui ne concerne que le calcul des promotions. Je le déplace plus faut avec l'opération de refactoring `Slide Statement`.
- Là encore, je vérifie que les tests unitaires passent bien, car j'ai sorti cet appel d'une branche d'un `if` pour l'exécuter systématiquement, donc il y a un changement dans le déroulé du programme.

> Il y a maintenant un bloc de code qui ne concerne que le calcul des promotions, je vais l'isoler dans une méthode `applyPromotionsToFacture(facture)` avec l'opération de refactoring `Extract Method`
- Exécution des tests unitaires

> Le test `exclusivePromotions.isEmpty()`, bien que parfaitement fonctionnel, n'est pas très parlant d'un point de vue métier. Effectivement, écrit tel quel, il est très technique. Je décide d'extraire ce code et d'en faire une méthode avec un nom qui va révéler l'intention métier : `thereIsNoExclusivePromotionsIn(availablePromotions)`.

- Je peux pour cela utiliser la méthode `noneMatch(Promotion::isExclusive)` du stream, qui en plus me simplifie l'écriture et permet d'indiquer clairement la règle technique qui est appliquée dans cette méthode métier.
- Exécution des tests unitiaires.

> Du coup, je n'ai plus besoin, pour savoir si il y a des promotions exclusives, d'avoir constitué la liste des promotions exclusives, et je peux déplacer cette création de liste de promotions exclusives dans la seconde branche du `if`, avec `Slide Statement`.

- Exécution des tests unitaires.

> Ensuite, j'ai un bloc de code dont le rôle est de trouver quelle est la promotion la plus intéressante pour la facture. Ce bloc de code n'est pas très parlant, je l'extrait dans une méthode `getBestExclusivePromotionForFacture()`.

- Exécution des tests unitaires

> Dans cette nouvelle méthode, je construit une nouvelle liste en filtrant la première, puis je parcours cette nouvelle liste dans une boucle `for`. Ai-je réellement besoin de cette liste `exclusivePromotions` ? Non, je peux faire mon filtrage directement dans la boucle `for`. Je crée donc un `if` dans la boucle et je change la liste parcourue, puis je supprime la liste qui est maintenant inutile.

- Ré-exécution des tests unitaires, et particulièrement parce que j'ai tapé du nouveau code, je n'ai pas fait qu'utiliser des opérations de refactoring de mon IDE.

> Cette boucle `for` est laide, très technique, elle utilise des variables pour stocker le meilleur montant de promotion et garder la promotion qui a permis ce calcul. Simplement parce que je n'ai pas de structure de `PromotionCalculée`. Si je définis une structure qui me permet de stocker en même temps la promotion et le montant qui y est associé, alors je pourrais simplifier cette méthode.

- Les `record` sont faits pour cela, j'introduis donc un Record `PromotionCalculee`, local à la méthode, qui contiendra une `Promotion` et un montant de type `BigDecimal`

```
  record PromotionCalculee(Promotion promotion, BigDecimal montant)
{};
```

- Ensuite, il n'y a plus qu'à parcourir le stream :

```
return availablePromotions.stream()
          .filter(Promotion::isExclusive)
          .map(promotion -> new PromotionCalculee(promotion,
getRemiseAmountOfPromotionAppliedTo(promotion, facture)))
          .max((pc1, pc2) ->
pc1.montant.substract(pc2.montant).signum())
          .get().promotion;
```

- Bon, d'accord, ce n'est peut-être pas tellement plus métier, mais cela me permet de n'avoir plus qu'une seule instruction, plus de variables temporaires, et plus d'algorithme codé sur plusieurs instructions qui risque d'être modifié par indvertance dans le temps. Ce sont des préoccupations dont on reparlera plus tard.

- Maintenant, je peux revenir dans la méthode `applyPromotionsToFacture`. Le commentaire ne précise plus rien puisque le code en dessous est parfaitement compréhensible d'un point de vue métier. Je supprime donc ce commentaire.
  - Et là, exceptionnellement, je me permet de ne pas relancer les tests unitaires !

- La méthode `createFacture` maintenant instancie une facture, la calcule, l'enregistre en base, et la renvoie. On peut là encore extraire tout ce contenu dans une unique méthode `createAndSaveFacture(clientId, qte)`

- Dans la méthode `createAndSaveFacture`, j'ai un appel à `new GregorianCalendar`. Ce n'est qu'un appel technique pour récupérer la date courante. Je l'extrait dans une méthode `getCurrentDate()` ; cela me permettra, si besoin, de la surcharger dans des tests unitaires si cela était nécessaire.

- J'ai créé plein de méthodes privées dans ma classe. Ces méthodes privées ne sont que le détail des méthodes publiques imposées par l'interface. Je déplace donc ces méthodes plus bas en essayant de les mettre dans un ordre cohérent pour le sens de la lecture. ⌘⇧ flêche.

On a maintenant un code lisible, clair, dans lequel on comprend immédiatement les intentions métier. N'importe quel développeur peut venir dans ce code et comprendre rapidement ce que ce code fait.

Démo 4

Ici, l'objectif va être d'appliquer le Single Responsability Principle. Si on regarde la classe `FactureController`, combien de raisons a-t-on de la modifier ?

- changement de l'API REST (URLS, méthodes, etc...)
- changement de façon de calculer la facture

Soit 2 raisons de modifer cette classe. Clairement, cela fait une raison en trop. Il va falloir résoudre ce problème.

Cette classe étant annotée `@RestController`, je décide d'en faire la classe qui implémentera la partie REST de la facture. Tout ce qui n'est pas du REST doit donc sortir de cette classe et être positionné ailleurs. Je vais procéder étape par étape pour faire cela, comme fait lors du refactoring **de compréhension**.

Tests unitaires

Je commence par regarder les méthodes sur lesquelles je ne suis pas encore intervenu : `getFacture(id)`, `getFactures()` et `printFacture(id)`. Ces méthodes accèdent aux classes `repository`, qui n'ont rien à voir avec le REST, il faudra donc les modifier. Mais elles ne sont pas testées. Je dois donc couvrir l'ensemble de la classe avec des tests unitaires, jusqu'à ce que tout soit testé. Je procède comme tout à l'heure, nous ne sommes pas ici pour écrire des tests unitaires, mais pour refactorer.

- lancer la commande `demo7-step-01`, ouvrir `FactureControllerTest`, selectionner tout et coller.

Création inner class

L'idée générale est de déplacer tout le code qui ne concerne pas les services REST dans une autre classe. A terme, cette classe s'appellera `FactureBusinessImpl` et elle implémentera `FactureBusiness` pour respecter les standards de Spring.

Mais je veux éviter d'écrire du code manuellement ; c'est le principe du refactoring, on évite d'écrire du code, on préfère utiliser les capacités de l'outil de développement.

Ici, je vais extraire ce qui ne m'intéresse pas dans une inner class, puis j'extrairai cette inner class, puis j'extrairai l'interface, puis je m'occuperai des tests unitaires ensuite.

> Je crée une inner classe non statique dans `FactureController`. Ensuite, je vais déclarer un attribut `factureBusiness` dans `FactureController`, que je vais initialiser manuellement, c'est à dire sans Spring, pour le moment. Comme je dois initialiser cet attribut, mais que je ne peux pas le faire dans le constructeur, j'ajoute une méthode `postInit()` que j'annote avec `@PostConstruct`.
>
- Je ré-exécute les tests unitaires.

Déplacement de getFacture(Long)

Maintenant que la structure est prête, je peux m'occuper de la méthode `getFacture(Long)`. Je ne peux pas changer la signature de cette méthode, mais elle utilise le repository.

> Donc je vais extraire `repository.findById(id)` dans une autre méthode que j'appelle `findFactureById`.

- Je ré-exécute les tests unitaires.

> Cette nouvelle méthode utilise `repository`. Quand je vais la déplacer, j'aurais besoin d'un repository dans la `FactureBusiness`.
- Je crée un variable privée et `final FactureRepository factureRepository` dans `FactureBusiness`, et j'initialise dans le constructeur. Le refactoring change aussi l'appel au constructeur.
- Je ré-exécute les tests untiaires.

> Ensuite je déplace la méthode extraite avec `Move Instance Method` **F6** et je choisis `FactureBusiness` comme destination. IntelliJ passe un `factureController` comme paramètre,
- je modifie pour utiliser celui de `FactureBusiness`
- je change la signature de la méthode pour supprimer le paramètre inutile et passer le paramètre en `long`
- je renomme la méthode en `getFacture` qui était le nom original
- je ré-exécute les tests unitaires

Extraction de getFactures

Je procède de la même façon avec `getFactures`.

> J'extraie `repository.findAll` dans une méthode que j'appelle `getAllFactures`

> J'utilise `Move instance method` et je choisis `FactureBusiness` comme cible
- Cette fois, je n'ai pas de problème avec `repository`

> Je renomme la méthode `getFactures`

> Je ré-exécute les tests unitaires

Extraction de printFacture

`printFacture` utilise `vistamboireRepository` et `printer`. Je déclare ces deux attributs dans `FactureBusiness` et les initialise dans le constructeur.

> Dans la méthode `printFacture` je change l'appel à `getFacture` par un appel à `factureBusiness.getFacture`

> J'extraie les 3 lignes de code dans une nouvelle méthode `printInnerFacture`

> Je déplace la méthode dans `FactureBusiness`
- je change l'utilisation de `vistamboireRepository` pour utiliser le local
- pareil pour le `printer`
- je change la signature de la méthode pour supprimer le paramètre `factureController` inutile et passer id en `long`
- enfin je renomme la méthode en `printFacture`

> Je ré-exécute les tests unitaires

Extraction de createAndSaveFacture

Reste maintenant toutes les méthodes qui sont utilisées par `createAndSaveFacture`. Je vais procéder de la même façon en faisant attention à l'accès aux différents repository et autres classes externes.

> Je commence par `findClientyId`
- Je crée le field `ClientRepository` dans `FactureBusiness`
- Ensuite, la méthode jette une exception `ResponseStatusException` qui est un truc HTTP, dont je ne veux pas dans `FactureBusiness`.
- Je crée une nouvelle exception `ValueNotFoundException`.
- Je change l'exception changée par une `new ValueNotFoundException("Client inconnu : "+clientId);`
- Je propage l'exception dans `createAndSaveFacture`
- Dans `createFacture`, je catch et je relance l'exception originale
- Je ré-exécute les tests unitaires
- Je déplace `findClientById` dans `FactureBusiness`
- Je change l'usage des paramètres et je change la signature
- Je ré-exécute les tests unitaires

> Je déplace ensuite `getVistamboireForFacture` de la même façon
- création du `prixUnitCalculateur`
- déplacement de la méthode
- modification de la signature
- tests unitiaires

> Ensuite pareil avec `getRemiseAmountOfPromotion`

> Puis avec `thereIsNoExclusivePromotionIn`

> Puis avec `calculateRemiseClientForFacture
- déclaration du field `databaseValuesExtractor`
- déplacement de la méthode
- changement de la signature
- tests unitiares

> Puis avec `getBestExclusivePromotionForFacture`
- avec changement de signature
- tests unitaires

> Puis avec `applyPromotionsToFacture`
- field `PromotionRepository`
- déplacement
- changement signature

> Puis avec `getCurrentDate`

> Puis avec `createAndSaveFacture`

- paramètre, signature

Maintenant, tout le code qu'on souhaitait isoler est dans une inner class de `FactureController`, il reste à remonter cette inner class d'un niveau. JE fais ça avec une opération de refactoring appellée `Move inner class to upper level`.

> Je la met dans le package `business.impl`

> Je change la visibilitié des méthodes

> Je ré-exécute les test unitaires.

Il reste à rendre cette classe conforme à Spring.

> Je supprime le constructeur

> Je déclare les filed `@Autowired`

> J'extraie l'interface avec `Extract Interface`

> Je déclare le Bean dans `VistamboireConfig`

> Je passe le champ dans `FactureController` à `@Autowired`

> Dans `FactureControllerTest` je déclare le bean dans la config

> Je lance les tests unitaires

> Je lance l'application

Nettoyage et tests

Je nettoye `FactrueController` pour supprimer tout ce qui n'est pas utile. Je lance les tests avec couverture.

> La ligne de gestion d'exception n'est pas couverte, c'est logique, elle est nouvelle

> Toute la nouvelle classe est couverte par les tests

Par contre, les tests qui sont dans `FactureControllerTest` concernent maintenant les tests de `FactureBusiness` mais pas la classe `FactureController`. Je vais donc changer cette classe et la spécialiser pour qu'elle ne teste que `FactureBusinessImpl`.

> Renommage et déplacement de la classe

> Suppression de l'attribut FactureController

> Compilation

> Exécution des TU
- `when_getFacture_repository_getFacture_should_be_call_once` plante
- on a changé son fonctionnement, elle renvoie maintenant un `Optional`
- il faut réécrire le test

> Exécution des TU

Si on a le temps, on réécrit les TU pour la classe FactureController qui n'est plus testée

En conclusion

- si on regarde la classe `FactureController`, la seule raison qu'on peut avoir de modifier cette

classe est un changement dans les API REST de la facture. Changement d'URL, de syntaxe, etc...

- si on regarde la classe `FactureBusinessImpl`, la seule raison qu'on peut avoir de modifier cette

classe, c'est un changement dans les règles d'accès à la facture, création et recherche. Chaque méthode est soit un **Middle Man**, soit un orchestrateur.

On a réussi à faire en sorte que si on doit écrire un service SOAP pour la gestion des factures, il n'y ait aucune raison de modifier la classe `FactureBusinessImpl`, et mieux, on pourra la réutiliser directement sans rien avoir à adapter.

Démo 5 : OCP

Open Closed Principle nous dit que quand on a un nouveau cas de gestion, on ne doit pas modifier de code existant. Mais on doit pouvoir enrichir nos cas de gestion simplement en ajoutant du code.

Si je regarde la classe `PrixUnitCalculateurImpl`, il y a deux méthodes qui ne respectent pas l'OOP :

> `calculatePrixUnit` devra être modifiée si il y a un nouveau type de client

> `calculateRemiseClient` devra être modifiée si il y a un nouveau secteur géographique, ou si on change les seuils et/ou les pourcentages de remise, ou si on décide d'avoir des remises par quantité aussi pour d'autres types de clients que le professionnels. C'est aussi un non respect du SRP.

Step 1

Commençons par `calculatePrixUnit(vistamboire, client)`

Le premier bloc de if / else if ressemble bougrement à un switch. Je commence par le convertir en switch.

- sur le if ⌥ + ENTER, Replace if by switch

Ensuite, un switch peut très souvent être remplacé par un pattern Strategy. Celui-ci permet de ne plus avoir à traiter les cas possibles en les connaissant à l'avance, mais en les découvrant lorsqu'on en a besoin.

Dans l'idée, j'aimerai pouvoir écrire, pour le calcul de `prixUnitaireTypeClient` l'expression suivante :

```
ClientType.of(client.getType()).calculatePrixUnitaire(vistamboire)
```

Evidemment, il y a un peu de travail, mais on va réussir à faire ça.

> Création d'une inner-class pour `ClientType`

> Changement de `class`en `enum`

> Retour sur ligne de code, génération de `of`

```
return Arrays.stream(values())
  .filter(clientType -> clientType.code.equals(type))
  .findFirst()
  .orElseThrow();
```

> Retour sur le code, génération de la méthode `calculatePrixUnitaire(vistamboire)`

Pour faire simple, à ce jour, on multiplie le prix unitaire du vistamboire par un coefficient multiplicateur spécifique au type de client sur lequel on est. On va écrire cela dans la méthode :

```
return vistamboire.getPrixUnitaireHT().multiply(coefMultiplicateur);
```

- génération de field `coefMultiplicateur`
- passage en final

- ajout en paramètre de constructeur

> Il reste à déclarer dans cette enum nos deux types de client, avec leurs caractéristiques :

```
PARTICULIER(
    Client.TYPE_PARTICULIER,
    BigDecimal.ONE),
PROFESSIONNEL(
    Client.TYPE_PROFESSIONNEL,
    BigDecimal.valueOf(0.7));
```

> Suppression du code mort

> Exécution des tests unitaires.

> Dans `calculatePrixUnitaire`, remplacement du if / else par un ternaire

> Suppression des commentaires.

On a mantenant un code qui n'évoluera plus lorsqu'on aura un nouveau type de client, respect du OOP.

Méthode calculateRemiseClient

Là encore, on a une structure qui ressemble à un switch. On ne va pas faire apparaitre le switch, maintenant vous savez où je veux en venir. Je sais que le nouveau du marketing veut ajouter un nouveau secteur géographique, il faut donc que je mette le code en capacité d'être étendu sans être modifié. Là encore, utilisation d'un pattern Strategy.

> Je commence par écrire le code que j'aimerai avoir au final :

```
... } else {
  SecteurGeographique secteur...
  int qteFinale = ...
  return
RemiseSecteurGeo.of(secteurGeographique).calculateRemise(qteFinale);
}
```

> Création d'une inner class RemiseSecteurGeo

- conversion en enum
- génération et implémentation de la méthode of

```
  public static RemiseSecteurGeo of(SecteurGeographique
secteurGeographique) {
    return Arrays.stream(values())
      .filter(remiseSecteurGeo ->
remiseSecteurGeo.code.equals(secteurGeographique.getNom()))
      .findFirst()
      .orElseThrow();
  }
```

- génération de la méthode `calculateRemise(int qte)` Là, j'aimerai bien parcourir les seuils, sans pour autant les avoir codés dans un if / else if... Je peux faire ça avec une liste de `Seuil`, où un seuil est un couple (qte, pourcentage)

```
  public BigDecimal calculateRemise(int qteFinale) {
```

```
    return seuils.stream()
      .filter(seuil -> qteFinale > seuil.qte())
      .findFirst()
      .map(seuil -> seuil.percent())
      .orElse(BigDecimal.ZERO);
  }
```

Et ce seuil, grace à Java 17, je peux le déclarer comme étant un inner record de l'enum `RemiseSecteurGeo`

```
record Seuil(int qte, BigDecimal percent) { }
```

- déclaration de `seuils` sous forme de field `List<Seuil>`

> Il reste à déclarer les deux secteurs géographiques :

```
AUTRE(
  SecteurGeographique.NOM_AUTRE,
  Arrays.asList(
    new Seuil(50, BigDecimal.valueOf(0.2)),
    new Seuil(20, BigDecimal.valueOf(0.15)),
    new Seuil(10, BigDecimal.valueOf(0.1))
  )),
MARITIME(
  SecteurGeographique.NOM_MARITIME,
  Arrays.asList(
    new Seuil(40, BigDecimal.valueOf(0.2)),
    new Seuil(25, BigDecimal.valueOf(0.17)),
    new Seuil(15, BigDecimal.valueOf(0.1))
  ));
```

> Suppression du code mort

> Exécution des tests unitaires

Finalisation

On voit maintenant que dans la méthode `calculateRemiseClient`, on a deux blocs de code différents pour chaque type de client. Cela signifie que si demain, il faut un nouveau type de client, cette méthode devra être modifiée.

Mais on a déjà fait des choses génériques avec les type de client, en utilisant `ClientType`. On va réutilliser cela, et on écrit alors le code qu'on souhaiterait obtenir :

```
return
ClientType.of(client.getType()).calculateRemiseClient(secteurGeograp
hique, qteFinale);
```

> Il reste à implémenter la méthode `calculateRemiseClient` dans l'enum `ClientType`

- Oui, mais les 2 types de client ont des fonctionnements très différents
- Il faut donc stocker pour chaque instance de l'enum du code différent
- On peut faire cela avec une interface fonctionnelle, et une lambda :

```
return remiseCalculator.apply(secteurGeographique, qte);
```

> `RemiseCalculator` est une `BiFunction<SecteurGeographique, Integer, BigDecimal>, qu'on initialise.

> On enrichit les constructeurs :
- dans le PARTICULIER, `(secteurGeographique, qte) -> BigDecimal.ZERO)`
- dans le PROFESSIONNEL, on réutilise l'autre énum :

```
(secteurGeographique, qte) ->
RemiseSecteurGeo.of(secteurGeographique).calculateRemise(qte)
```

- Suppression du code mort
- Exécution des tests unitaires

> Il reste l'extraction du secteur géographique, c'est du code dupliqué, on l'extrait dans une méthode