TD M3105 : A la découverte du pattern Chaîne de responsabilité (Chain of Responsibility : CoR)

Vous venez d'intégrer une équipe qui est chargée de développer un prototype de distributeur de billets pour un terminal bancaire (appelé aussi ATM *Automated Teller Machines* ou DAB *Distributeur Automatique Bancaire*).
Pour l'instant, cette équipe s'est uniquement focalisée sur le service de **retrait** de billets.
Pour simplifier le problème, l'équipe a choisi, pour sa première implémentation, de considérer que le distributeur disposerait uniquement de **billets de 50 €, 20 €** et de **10 €** pour répondre à la demande du client. Elle a également considéré que le distributeur disposerait pour l'instant de toujours assez de billets pour bien répondre à la demande du client. La gestion de l'état du distributeur et de la balance du compte du client viendront dans un prochain sprint 😉
Pour l'instant, lors d'un retrait, le distributeur doit donc simplement faire en sorte de distribuer le moins de billets possible, en commençant par délivrer des billets de 50€, puis de 20 € et enfin de 10 €.

*Pour commencer, vous travaillerez en **mode déconnecté** :*
Laissez vos ordinateurs éteints pour le moment et prenez une feuille de papier 😉

1. Se familiariser avec le code existant …

L'équipe a commencé à implémenter un bout de code qui répond aux spécifications précédentes. Pour vous intégrer, l'équipe commence par vous briefer sur la terminologie métier utilisée dans ce développement (*ubiquitous language*) :

- **withdraw** est le service de retrait de billets que propose le distributeur
- **amount** est la somme demandée par le client lors d'un retrait
- **bank note** est une billet de banque qui est caractérisé par une valeur et une devise
- **bank note bundle** est un ensemble de billets de banque de même valeurs (une liasse de billets en quelque sorte)

L'équipe est ensuite très fière de vous présenter le fichier de tests unitaires qui sert également de *documentation vivante* puisque ces tests ne sont autres que des spécifications exécutables (ce qui devrait vous rappeler quelques souvenirs du module M3301 😉)

Pour écrire les tests unitaires, l'équipe a décidé d'utiliser le framework **AssertJ** qui permet d'écrire des assertions plus *fluentes* en utilisant *`assertThat`* (au lieu du *`assertEquals`* de JUnit)

Commencez par prendre connaissance de ce fichier de tests pour vous familiariser avec les différents composants et le comportement du code existant :

```
public class ATMTest {

   ATM atm = new ATM();
   BankNote bankNote_of_10Euros = new BankNote(10,"Euros");
   BankNote bankNote_of_20Euros = new BankNote(20,"Euros");
   BankNote bankNote_of_50Euros = new BankNote(50,"Euros");


   @Test
   public void should_dispense_one_10EurosBankNote_when_withdraw_10Euros() {
      int amount = 10;
      Money money = atm.withdraw(amount);
      assertThat(money.getNotes()).hasSize(1)
                                  .containsExactly(new BankNotesBundle(bankNote_of_10Euros,1));
   }
```



```
   @Test
   public void should_dispense_one_20EurosBankNote_when_withdraw_20Euros() {
      int amount = 20;
      Money money = atm.withdraw(amount);
      assertThat(money.getNotes()).hasSize(1)
                                  .containsExactly(new BankNotesBundle(bankNote_of_20Euros,1));
      }

   @Test
public void should_dispense_one_20EurosBankNote_and_one_10EurosBankNote_when_withdraw_30Euros() {
      int amount = 30;
      Money money = atm.withdraw(amount);
      assertThat(money.getNotes()).hasSize(2)
                          .containsExactlyInAnyOrder(new BankNotesBundle(bankNote_of_20Euros,1),
                                                     new BankNotesBundle(bankNote_of_10Euros,1));
      }

   @Test
   public void should_dispense_two_20EurosBankNotes_when_withdraw_40Euros() {
      int amount = 40;
      Money money = atm.withdraw(amount);
      assertThat(money.getNotes()).hasSize(1)
                                  .containsExactly(new BankNotesBundle(bankNote_of_20Euros,2));
      }

   @Test
   public void should_dispense_one_50EurosBankNote_when_withdraw_50Euros() {
      int amount = 50;
      Money money = atm.withdraw(amount);
      assertThat(money.getNotes()).hasSize(1)
                                  .containsExactly(new BankNotesBundle(bankNote_of_50Euros,1));
      }

   @Test
public void should_dispense_one_50EurosBankNote_and_one_10EurosBankNote_when_withdraw_60Euros() {
      int amount = 60;
      Money money = atm.withdraw(amount);
      assertThat(money.getNotes()).hasSize(2)
                          .containsExactlyInAnyOrder(new BankNotesBundle(bankNote_of_50Euros,1),
                                                     new BankNotesBundle(bankNote_of_10Euros,1));
      }

   @Test
   public void should_dispense_one_50EurosBankNote_and_one_20EurosBankNote_when_withdraw_70Euros() {
      int amount = 70;
      Money money = atm.withdraw(amount);
      assertThat(money.getNotes()).hasSize(2)
                          .containsExactlyInAnyOrder(new BankNotesBundle(bankNote_of_50Euros,1),
                                                     new BankNotesBundle(bankNote_of_20Euros,1));
      }

   @Test
   public void should_dispense_two_50EurosBankNote_and_one_10EurosBankNote_when_withdraw_110Euros() {
      int amount = 110;
      Money money = atm.withdraw(amount);
      assertThat(money.getNotes()).hasSize(2)
                          .containsExactlyInAnyOrder(new BankNotesBundle(bankNote_of_50Euros,2),
                                                     new BankNotesBundle(bankNote_of_10Euros,1));
}
```

```
   @Test
   public void
should_dispense_six_50EurosBankNotes_and_one_20EurosBankNote_and_one_10EurosBankNote_when_amount_is_330Euros
() {
      int amount = 330;
      Money money = atm.withdraw(amount);
      assertThat(money.getNotes()).hasSize(3)
                        .containsExactlyInAnyOrder(new BankNotesBundle(bankNote_of_50Euros, 6),
                                                   new BankNotesBundle(bankNote_of_20Euros, 1),
                                                   new BankNotesBundle(bankNote_of_10Euros, 1));
      }
}
```

Une fois que vous avez pris connaissance des tests, vous décidez de jeter un coup d'œil au code source existant. A la lecture de ce code, vous constatez que l'équipe a choisi d'utiliser le **projet lombok** pour ne pas avoir à écrire, entre autres, les méthodes **`equals`** et **`hascode`**, et simplifier ainsi la lisibilité du code dans les classes métier. Avec le projet lombok, les méthodes **`equals`** et **`hascode`** sont considérées comme redéfinies dès lors que la classe est annotée par **`@EqualsAndHashCode`**

```
import lombok.EqualsAndHashCode;

@EqualsAndHashCode
public class BankNote {

      int value;
      String currency;

      public BankNote(int value, String currency) {
            this.value = value;
            this.currency = currency;
      }

      public String denomination() {
            return ("" + this.value + " " + this.currency);
      }
}
```

```
import lombok.EqualsAndHashCode;

@EqualsAndHashCode
public class BankNotesBundle {

      BankNote bankNote;
      int numberOfNotes;

      public BankNotesBundle(BankNote bankNote, int numberOfNotes) {
            this.bankNote = bankNote;
            this.numberOfNotes = numberOfNotes;
      }
```



```
@Override
      public String toString() {
            return ("" + this.numberOfNotes + " * " +
                                                this.bankNote.denomination());
      }
}
```

```
import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;
import static java.util.stream.Collectors.joining;

import lombok.EqualsAndHashCode;
import lombok.Getter;

@EqualsAndHashCode
@Getter
public class Money {
      public static final Money NO = new Money(Collections.emptyList());

      private final List<BankNotesBundle> notes;

      public Money add(BankNotesBundle newNotes) {

            List<BankNotesBundle> updatedNotes =
                                    this.notes.stream().collect(Collectors.toList());

            updatedNotes.add(newNotes);
            return new Money(updatedNotes);
      }

      private Money(List<BankNotesBundle> notes) {
            this.notes = notes;
      }


      @Override
      public String toString() {
            return  return notes.stream()
                        .map(BankNotesBundle::toString)
                        .collect(joining());
      }
}
```

Remarque : Cette classe utilise des `stream` qui sont disponibles dans le JDK depuis Java 8. **Vous ne devez pas ici comprendre comment fonctionne un stream**, mais juste être conscient que cette notation est couramment utilisée dans le code Java d'aujourd'hui et que vous pouvez y faire face. Vous n'aurez pas à modifier ce bout de code Java 8 pendant le TD, cependant pour faciliter votre compréhension, il vous est indiqué en Annexe les instructions équivalentes qui auraient pu être écrites de manière « plus classique » dans les versions antérieures de Java.

```java
public class ATM {

   private static final String CURRENCY = "Euros";

   public Money withdraw(int amount) {

      Money money = Money.NO;

      // Dispense 50 Euros Notes
      if (amount >= 50) {
         int numberof50BankNotes = amount / 50;
         money = money.add(new BankNotesBundle(new BankNote (50, CURRENCY),
                                                numberof50BankNotes));
      }
      amount = amount % 50;

      // Dispense 20 Euros Notes
      if (amount >= 20) {
          int numberof20BankNotes = amount / 20;
          money = money.add(new BankNotesBundle(new BankNote (20, CURRENCY),
                                                numberof20BankNotes));
      }

      amount = amount % 20;

      // Dispense 10 Euros Notes
      if (amount >= 10) {
         int numberof10BankNotes = amount / 10;
         money = money.add(new BankNotesBundle(new BankNote (10, CURRENCY),
                                                numberof10BankNotes));
      }

     return money;
   }

}
```

1.1 Modéliser le **diagramme de classes** correspondant au code source existant.

1.2 Représenter sur ce diagramme de classes les dépendances de la classe **`ATM`** avec les autres classes. De combien de classes, la classe ATM dépend-elle ?

2. Revue de code…

2.a Zoom sur les classes `BankNote`, `BankNotesBundle` et `Money` :

Que pouvez-vous dire sur ces classes ? Quel pattern respectent-elles ?
Pourquoi est-ce intéressant de mettre en place un tel pattern dans votre code ?

2.b Zoom sur la classe `ATM`

→ Ce code ne suit pas certains principes SOLID.
Identifiez ces principes, rappelez leur définition et indiquez en quoi ces principes ne sont pas respectés ici.

→ Identifiez-vous d'autres mauvaises odeurs (*code smell*) dans ce code ?

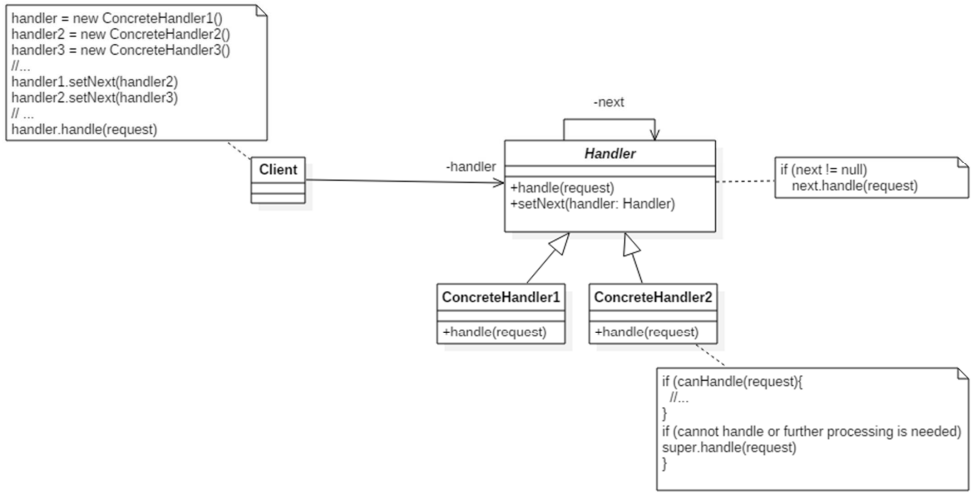
Pour rendre ce code plus SOLIDe, des développeurs expérimentés de votre équipe suggèrent de le refactorer avec un pattern chaîne de responsabilité et cette tâche vous est confiée 😉

3. Présentation du design pattern chaîne de responsabilité

Le design pattern **chaîne de responsabilité** (**Chain of Responsibility : CoR**) est un **pattern de comportement**.

Pour vous familiariser avec ce pattern voici un peu de documentation inspirée de **http://www.goprod.bouhours.net**,

Le diagramme de classes de ce pattern est :



Participants au patron :

→ **`Handler (Gestionnaire)`**
Classe abstraite qui définit une interface pour gérer les requêtes.
Comporte éventuellement le code de liaison avec le successeur.

→ **`ConcreteHandler`**
Traite les requêtes (**`handle(request)`**) dont il est responsable
Sait accéder à son successeur dans la chaîne
Si un handler (gestionnaire) sait traiter une requête, il le fait ; dans le cas contraire, il la transmet à son successeur en utilisant la chaîne de responsabilité.

→ **`Client`**
Est l'émetteur de la requête, qu'il passe à la chaîne c-a-d qu'il propose initialement la requête à un objet **`ConcreteHandler`** de la chaîne.

Collaborations : Le client effectue la requête initiale auprès d'un gestionnaire. Cette requête est propagée le long de la chaîne de responsabilité jusqu'au moment où l'un des gestionnaires la traite.

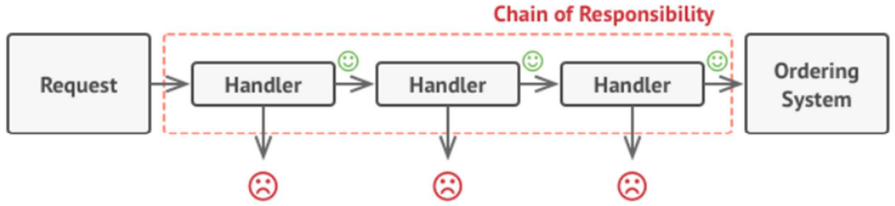
L'intention de ce pattern est donc d'éviter de coupler l'émetteur d'une requête à son récepteur en permettant à plus d'un objet d'y répondre.

Le **patron chaîne de responsabilité** permet de chainer les objets récepteurs et de faire passer la requête tout au long de la chaîne jusqu'à ce qu'un objet la traite.

… et pour illustrer tout ça, un peu de documentation en anglais
extraite de https://refactoring.guru/design-patterns/chain-of-responsibility :

Like many other behavioral design patterns, the Chain of Responsibility relies on transforming particular behaviors into stand-alone objects called ***handlers***[1].

The pattern suggests that you link these handlers into a chain. Each linked handler has a field for storing a reference to the next handler in the chain. In addition to processing a request, handlers pass the request further along the chain. The request travels along the chain until all handlers have had a chance to process it.

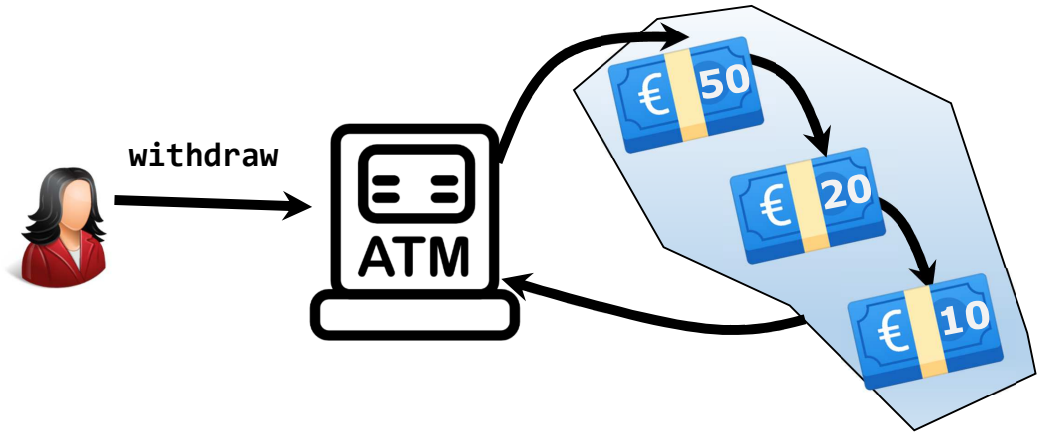


Handlers are lined up one by one, forming a chain.

Here's the best part: a handler can decide not to pass the request further down the chain and effectively stop any further processing.

However, there's a slightly different approach (and it's a bit more canonical) in which, upon receiving a request, a handler decides whether it can process it. If it can, it doesn't pass the request any further. So it's either only one handler that processes the request or none at all.

Autrement dit, chaque objet de la chaîne joue le rôle de gestionnaire et a un objet successeur.
Le client initie la requête,
si un *handler concret (gestionnaire concret)* peut gérer la requête il le fait,
sinon il la transmet à son successeur (prochain handler de la chaîne).



3.1 Le code de la classe `ATM` traite actuellement dans **`withdraw`** une seule requête (demande de retrait d'un certain montant) qui est la succession de 3 comportements implémentés dans 3 *si* (`if`) linéaires.

➔ Quel est le premier comportement c-a-d qu'est censé gérer la première partie de la requête ?
……..

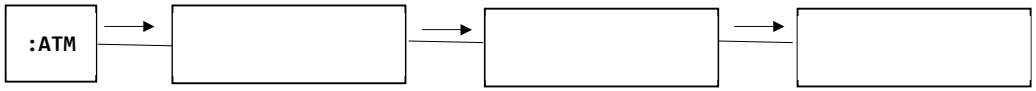
➔ Qu'est censé gérer la deuxième partie de la requête ?
……..

➔ Qu'est censé gérer la troisième partie de la requête ?
……..

3.2 A partir de votre analyse précédente, vous devriez être capable de **compléter le diagramme de communication suivant.**
Pour rappel, le diagramme de communication est un **diagramme d'interactions UML** qui se concentre sur les **échanges de message entre les objets** : souvenir du module M2104 😉…

➔ Compléter le nom de la classe des **objets** : Gestionnaire de ….
➔ et indiquer quel **message** passe d'objet en objet

Refaire le diagramme précédent en utilisant des termes anglais respectant *l'ubiquitous language* du code existant :



3.3 Transformez ce ***diagramme*** d'***objets*** en ***diagramme de classes*** de manière à retrouver un diagramme de classes qui s'apparente au diagramme de classes générique du pattern Chaine de responsabilité présenté dans la partie *3. Présentation du design pattern chaîne de responsabilité* de ce TD.

Bravo, vous venez d'adapter le pattern Chaîne de responsabilité à votre contexte (celui de l'ATM). A l'aide du diagramme précédent, identifiez les classes participantes à ce pattern :

→ la classe **`Client`** du pattern Chaine de responsabilité correspond à la classe _____ dans le contexte de l'ATM.

→ la classe **`Handler`** du pattern Chaine de responsabilité correspond à la classe _____ dans le contexte de l'ATM.

→ la classe **`ConcreteHandler`** du pattern Chaine de responsabilité correspond à la classe _____ dans le contexte de l'ATM.

⇨ la méthode **`handle(request)`** du pattern Chaine de responsabilité correspond à la méthode _____ dans le contexte de l'ATM.
Que contiendra cette méthode ?

Et si on implémentait tout ça maintenant 😉 ...

*Vous pouvez maintenant passer en **mode connecté** 😉*

4. Remaniement du code avec une chaîne de responsabilité ...

4.1 Dans votre IDE préféré, créez un projet Maven **`atm`**.
Importez le code de disponible dans le répertoire **ressources** `atm` du dépôt
https ://github.com/iblasquez/enseignement-iut-m3105-conception-avancee

Sous Eclipse, `File | Import... | Maven | Existing Maven Projects`, puis **Next,** puis sélectionner le répertoire `atm` où vous avez récupérer les fichiers du TD, puis **Finish**.

Faites en sorte que ce code compile et que les tests passent AU VERT !
N'oubliez pas de versionner votre projet et de commiter régulièrement !!

4.2 Mise en place du pattern Chaine de responsabilité :

En vous inspirant du diagramme de classes que vous avez proposé à la question 3.3 et en vous appuyant sur vos tests (comme garde-fou du comportement), vous pouvez procéder en toute confiance au refactoring du code existant c-a-d faire
évoluer ce code pour **faire émerger le pattern Chaine de responsabilité** 😉

Une fois, le pattern Chaine de responsabilité correctement implémenté, vos tests devront continuer de passer AU VERT !

(Bien sûr au cours de ce refactoring, vous ne devez pas en aucun cas toucher aux tests existants… Rappelez-vous la définition du refactoring qui consiste à **modifier le code sans en changer son comportement** 😉)

4.3 Nouveau diagramme de classes du projet :

A l'aide de votre IDE, générez le diagramme de classes de votre projet, et vérifiez bien que vous retrouvez le diagramme de classes du pattern Chaîne de responsabilité adapté à votre contexte.

Pour limiter le couplage, veillez bien à ce que vous ayez qu'une seule association entre la classe **`ATM`** et le **`Handler`** 😉
Si tel n'est pas le cas, reprenez votre code et refaites passer les tests AU VERT 😉

5. Du code plus SOLIDe ?

5.1 Comment le pattern **Chaîne de responsabilité** a-t-il permis à votre code de respecter les principes SOLID qui ne l'était pas à la question 2.b

Le pattern Chaine de responsabilité a permis de respecter le principe .....
parce que désormais dans le code .....
.....
.....
.....

Le pattern Chaine de responsabilité a permis de respecter le principe .....
parce que désormais dans le code .....
.....
.....
.....

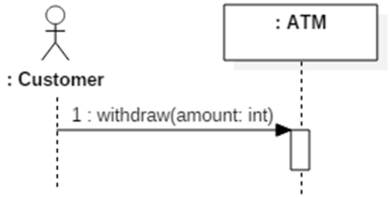
5.2 Pour vous convaincre que votre code est plus SOLID et facilement extensible, ajoutez, en mode TDD, le traitement des **billets de 5 Euros** 😉

Au final, vérifiez bien :

- qu'un retrait d'un montant de 5 Euros renvoie bien un seul billet de 5 euros
- qu'un retrait d'un montant de 85 Euros renvoie bien les billets attendus 😉

5.3 Diagramme de séquence

Avant le refactoring, le diagramme de séquences ressemblait au diagramme ci-dessous (si en considérant que nous avons volontairement pas représenté pas les classes **`BankNote, BankNotesBundle et Money`** sur ce diagramme).



Modéliser le diagramme de séquences après le refactoring (toujours avec la même hypothèse de ne pas représenter **`BankNote, BankNotesBundle et Money`** sur ce diagramme).

6. Capitaliser l'expérience acquise …

Lors de vos prochaines revues/relectures de code, qu'est-ce qui pourrez-vous inciter à proposer un refactoring vers un pattern chaine de responsabilité, d'après votre expérience durant ce TD, quel code est propice à ce genre de refactoring ? Compléter le petit mémo suivant pour que la prochaine fois que vous verrez un tel code, vous sachiez réutiliser ou conseiller à bon escient le pattern chaine de responsabilité pour rendre vos futurs développements plus SOLID…

Quand je vois dans un code ……
……
……
……
……
et que le principes ……. et …… ne semblent pas respectés, je me dis que le pattern chaîne de responsabilité pourrait être utilisé 😉

7. Un petit challenge pour aller plus loin …

Amusez-vous avec votre code et refactorez le de manière à ce que votre projet ne contienne plus aucun **`if`**
Petit indice : vous aurez peut-être besoin de revisiter un peu les tests pour implémenter cette petite contrainte 😉

Annexe : Equivalent en Java 7 (et avant) des instructions à base de stream dans la classe `Money`

➔ Le code suivant consiste à cloner une liste avec un stream (Java 8)

```
List<BankNotesBundle> updatedNotes =
          this.notes.stream().collect(Collectors.toList());
```

aurait pu être écrit en Java 7 (et avant) comme cela :

```
List<Notes> updatedNotes = new ArrayList<>();
updatedNotes.addAll(this.notes);
```

Remarque : En principe, on ne se limite pas au **`collect`**, ceci est juste un exemple pour mettre un peu de Java 8 dans ce code, cela n'a pas grand intérêt de juste collecter 😉 .
Habituellement, un **`stream`** sert à faire du traitement de données comme du filtre données (**`filter`**), de la transformation de données (**`map`**), de la réduction de données (**`reduce`**),…
Une fois, le traitement effectué, on finit par **collecter** dans un **Collector** tout ce traitement comme comme on le montre dans l'exemple suivant 😉

➔ Ce code permet d'illustrer les notions Java 8 : **stream**, `map` et **`Collector`**.
map permet de transformer des données ainsi
`map(BankNotesBundle::toString)` permet de transformer chaque élément de `BankNotesBundle` en `String` grâce à la méthode `toString`.
Pour cela une **méthode de référence** (notation **::**) est utilisée.
La notation **`BankNotesBundle::toString`** indique que la méthode toString va être appliqué sur chaque élément de type **`BankNotesBundle`**.
A la fin, tout les éléments sont collectés et concaténés via `joining`.

```
@Override
public String toString() {
      return  return notes.stream()
                  .map(BankNotesBundle::toString)
                  .collect(joining());
```

A propos de Assert J

En savoir plus sur AssertJ :
https://joel-costigliola.github.io/assertj/ et https://assertj.github.io/doc/