1 Contacter la base de données

Affirmation 1. *Nous ne voulons pas faire une relation directe entre notre coeur applicatif et notre base de données, car si on change le SGBD (e.g. Postgres vers MySQL) alors le coeur applicatif pourrait être impacté.*

1.1 Représenter la BDD

Nous souhaitons écrire en Java notre base de données. Ne pas à créer manuellement les tables, ne pas à avoir à écrire des requêtes SQL. Nous souhaitons que Java gère cela.

pour ce faire, on peut utiliser le patron *repository* ou *DAO* (malgré que les deux ne visent pas le même périmètre, vous pouvez les considérer identiques dans un premier temps).

- `FactureRepostitory` permet d'appeler les méthodes `save`, `get`, etc ... qui derrière seront traduites en SQL pour récupérer les informations.
- Notre table en base de données est représentée par une classe Java `FactureJPA`.

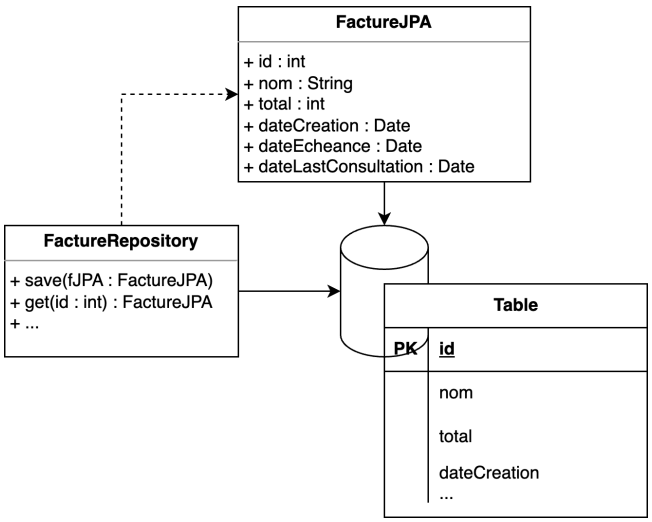

FIGURE 1 – Pattern Repository représentant notre base de données

1.2 Accéder à la base de données

Il est hors de question que le coeur applicatif est connaissance de la base de données. En effet si `FactureJPA` évolue nous n'avons pas forcément envie de modifier le coeur applicatif. Le patron Adaptateur est une solution pour découpler.

- `FactureJPA` et `FactureDTO` n'ont pas les mêmes attributs
- En effet certain seront calculés. Par exemple lorsqu'on va enregistrer une nouvelle facture nous allons dans l'adaptateur récupérer la date du jour.
- C'est pour cette raison que nous avons besoin de convertir notre `FactureDTO` en `FactureJPA` (et inversement pour le retour)

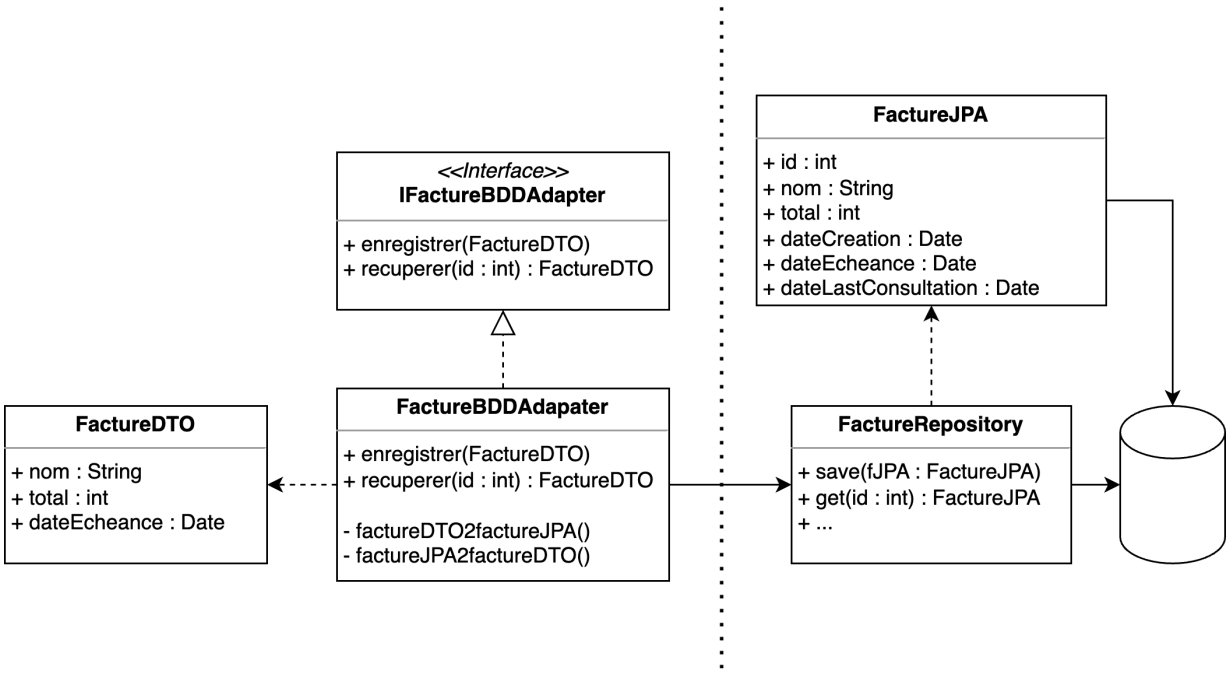

FIGURE 2 – Pattern Adaptateur et Repostiory - communication

```
public class FactureBDDAdapter {
   private FactureRepository factureRepository;

   void enregistrer(FactureDTO factureDTO) {
      FactureJPA factureJPA = factureDTO2factureJPA(factureDTO);

      factureRepository.save(factureJPA);
   }

   FactureDTO get(int id) {
      FactureJPA factureJPA = factureRepository.get(id);

      return factureJPA2factureDTO(factureJPA);
   }

   FactureJPA factureDTO2factureJPA(FactureDTO fDTO) {
      // dateCreation et dateLastConsultation sont crees ici en
         interne dans le systeme via Date.now()
      return new FactureJPA(fDTO.getNom(), fDTO.getTotal(),
         Date.now(), fDTO.getEcheance(), fDTO.now() ))
   }
}
```

1.3 Contacter la base de données

Donc maintenant mon coeur applicatif n'a qu'à appeler l'adaptateur nous communiquer avec la base de données. Ainsi si je change la base de données, j'aurais à modifier le code contenu dans l'adaptateur, mais pas celui de mon coeur applicatif.

— `Facture` est très proche de `FactureDTO` mais ils peuvent avoir des attributs qui divergent.
— `Facture` représente une *entité métier*. C'est la représentation d'une facture dans notre logique métier. Cette classe n'est pas forcément la même qu'en base de données (e.g. on s'en

moque de savoir qu'elle était la dernière consultation ; cela n'apporte pas de valeur)

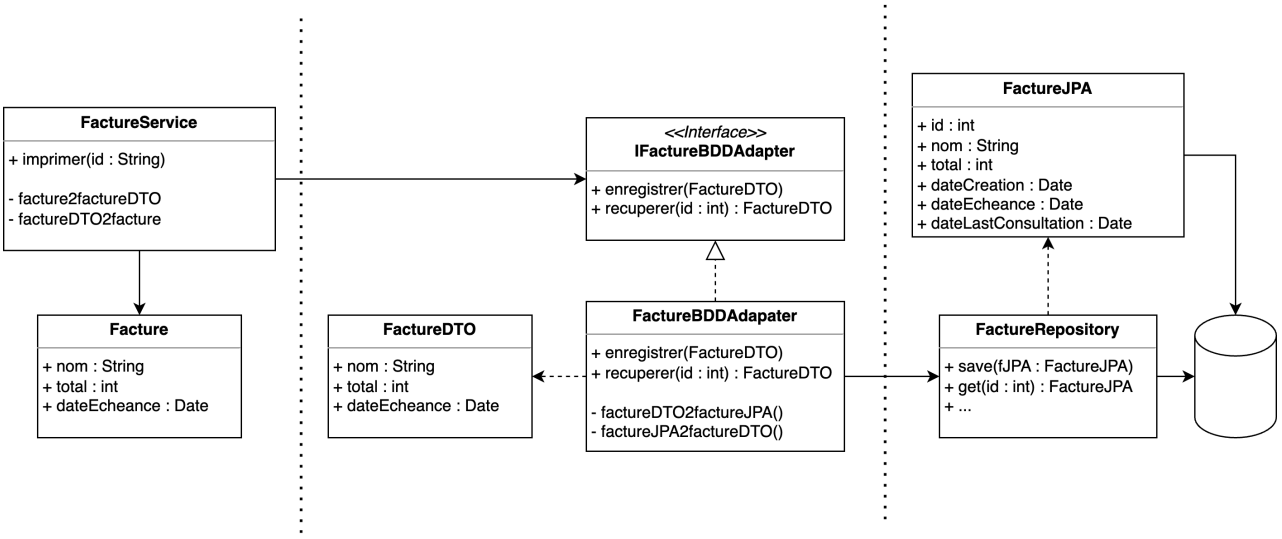

FIGURE 3 – Communication complète

```
public class FactureService {
   private Facture facture

   private IFactureBDDAdapter iFactureBDDAdapter

   void imprimer(int id) {
      FactureDTO fDTO = iFactureBDDAdapter.get(id);

      Facture facture = factureDTO2facture(fDTO)

      // imprimer la facture
   }

   FactureDTO factureDTO2facture(FactureDTO factureDTO) {
      return new Facture(...)
   }
}
```

1.4 Conclusion

— `Facture` représente une *entité métier*
— `FactureJPA` représente une *entité base de données*
— Les deux ne sont pas forcément identiques.

— Les DTO permettent de traverser les couches. On envoie/reçois que les informations strictement nécessaires.
— `FactureDTO` représente un DTO entre la couche métier et la couche persistance
— `FactureJPA` représente un DTO entre la couche persistance et la couche base de données

2 Modularité et interfaces

Dans cette section, nous revenons également sur le rôle des interfaces, mais en prenant de la hauteur. Le fait qu'elles permettent de découpler notre code cela améliorer la modularité (module / service) de notre application.

2.1 Conception initiale

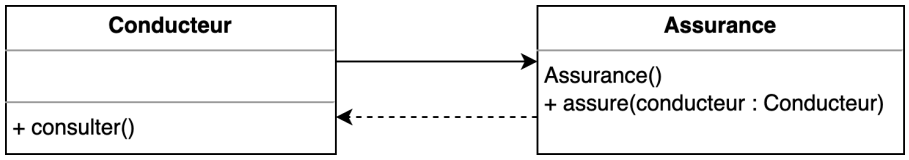

FIGURE 4 – Relation bidirectionnelle

```
class Conducteur {
   public void consulter() {
      Assurance assurance = new Assurance();
      assurance.assure(this)
   }
}
```

— `Conducteur` est fortement liée à `Assurance`
— `Assurance` est fortement liée à `Conducteur`
— Puis trois mois après on veut changer le système d'assurance. Etant fortement dépendant il va falloir re-tester, re-packager et re-deployer tout notre système

2.2 Ajout d'abstraction

Pour répondre à cette problématique nous devons rajouter de l'abstraction entre nos deux classes.

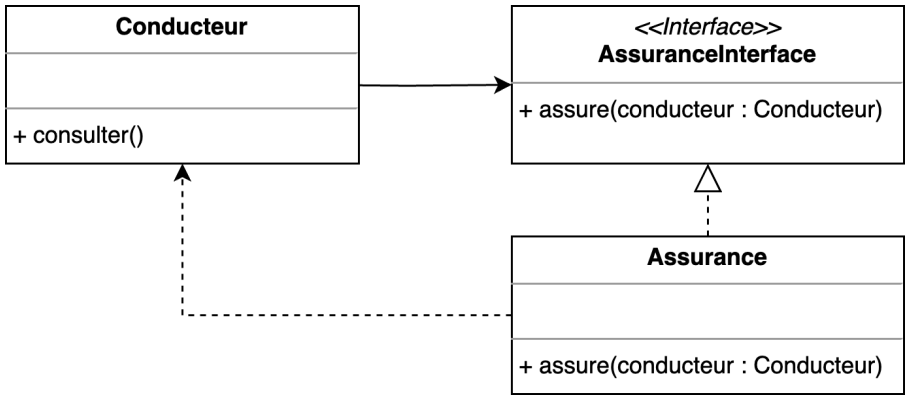

FIGURE 5 – Ajout d'abstraction, pour avoir un couplage plus faible

```
class Conducteur {
   public void consulter(AssuranceInterface assurance) {
      assurance.assure(this)
   }
}
```

— `Conducteur` est liée à `Assurance`

— `Assurance` liée à `Conducteur`
— Mais si on change le code de `assure()` ou on crée une nouvelle implémentation alors `Conducteur` n'aura pas besoin d'être re-testé, ni re-packagé ni re-déployé

Nous venons de faire de l'injection de dépendance

2.3 Mais relation cyclique

Néanmoins, si on prend un peu de hauteur (niveau module) nous remarquons une relation cyclique.

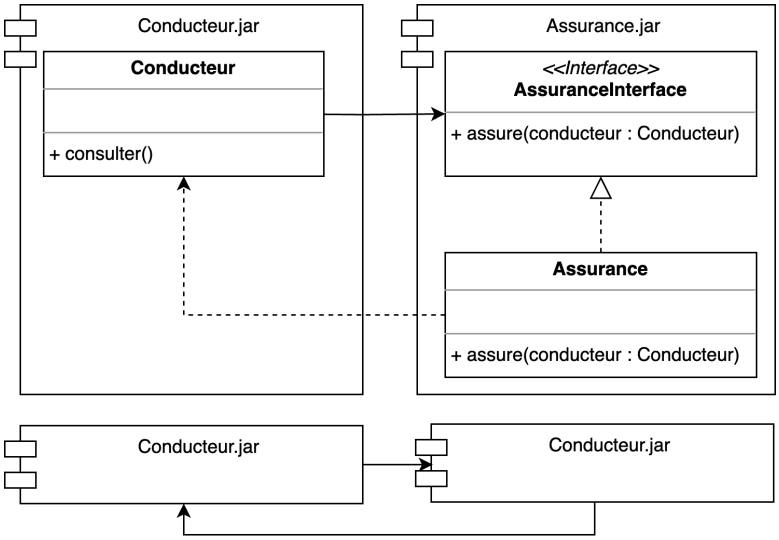

FIGURE 6 – Relation cyclique niveau module

2.3.1 Solution 1 : réagencer nos classes

Si vous avez la main sur toutes les classes (i.e. c'est vous qui les codé) alors vous pouvez modifier qu'elle classe va dans quel package.

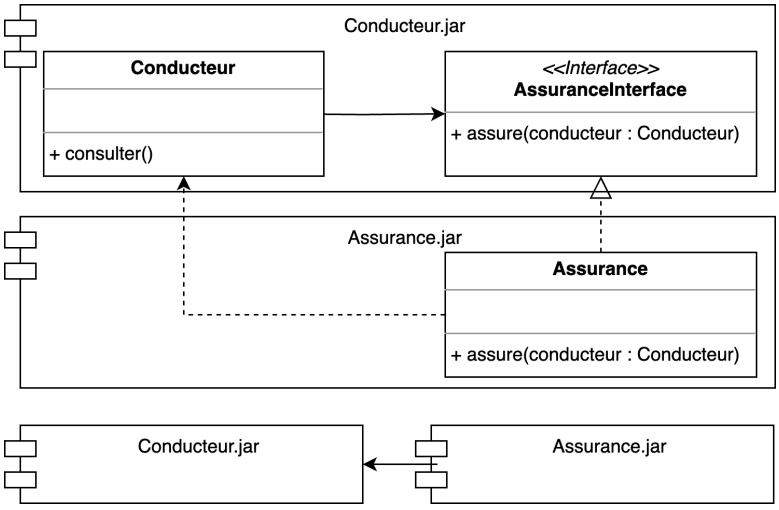

FIGURE 7 – Casser la relation cyclique en réagençant les classes

Maintenant nous avons uniquement `Assurance.jar` qui dépend de `Conducteur.jar`. **Nous venons de faire de l'inversion de dépendance**

2.3.2 Solution 2 : faire des librairies

La solution 1 fonctionne si nous avons la main sur nos classes. Cependant, il se peut que ces `.jar` soient des dépendances maven par exemple. Dans ce cas il faut que les concepteurs aient bien conçu leur module.

Cette deuxième solution est une "amélioration de l'inversion de dépendance" afin de créer des modules pouvant être réutilisés par tout le monde. Ici nous souhaitons pouvoir réutiliser le `Assurance.jar` dans n'importe quel contexte.

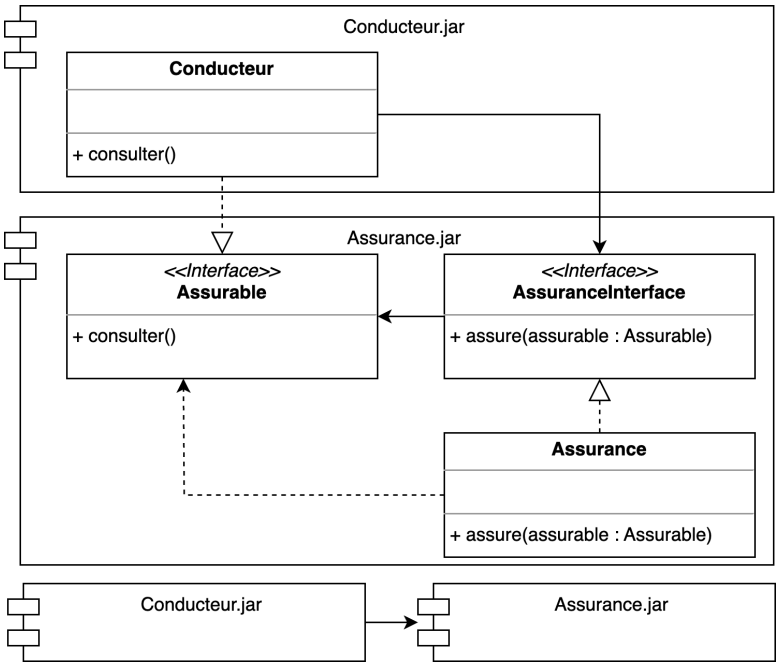

FIGURE 8 – Relation cyclique niveau module

Dans ce cas, les concepteurs de `Assurance.jar` doivent passer à

- Fournir une interface pouvant représenter un conducteur, ici `Assurable`
- Maintenant `Assurance` ne parle plus avec `Conducteur` mais avec l'interface `Assurable`.
- `Conducteur` étant un `Assurable` on peut appeler sans problème `assure()`

```
interface AssuranceInterface {
   public void assure(Assurable assurable);
}

/* Meme code quand 2.2 */
class Conducteur implement Assurable {
   public void consulter(AssuranceInterface assurance) {
      assurance.assure(this);
   }
}
```

2.4 Conclusion

Nous avons vu le rôle des interfaces afin de bien segmenter (via modules) notre application

- En partant d'une relation bidirectionnelle
- Nous avons ajouté une abstraction pour briser cette relation

— , Mais, au niveau module nous avions toujours une relation cyclique
  — Si vous avez la main, vous pouvez changer les classes de place
  — Si vous créez une librairie alors il faut bien penser à fournir des interfaces afin que cette librairie puisse être utilisée n'importe où.

Règle découpage

Une règle pour un bon découpage

Les interfaces devraient être proche des classes qu'elles utilisent et assez loin de leur implémentation

Note :
— Appliqué dans 2.3.1
— Non appliqué dans, 2.3.2 car `assurance.jar` est une dépendance déployable (sur maven par exemple) donc elle doit contenir ces interfaces pour qu'on puisse communiquer avec elle.