Compléments sur l’architecture d’un backend web

Architecture logicielle

Identifier et définir les composants de l'application, leurs interactions et interrelations, afin de

- Décomposer le problème
- Organiser le développement
- Améliorer la qualité

Architecture logicielle

Identifier et définir les composants de l'application,

en visant :

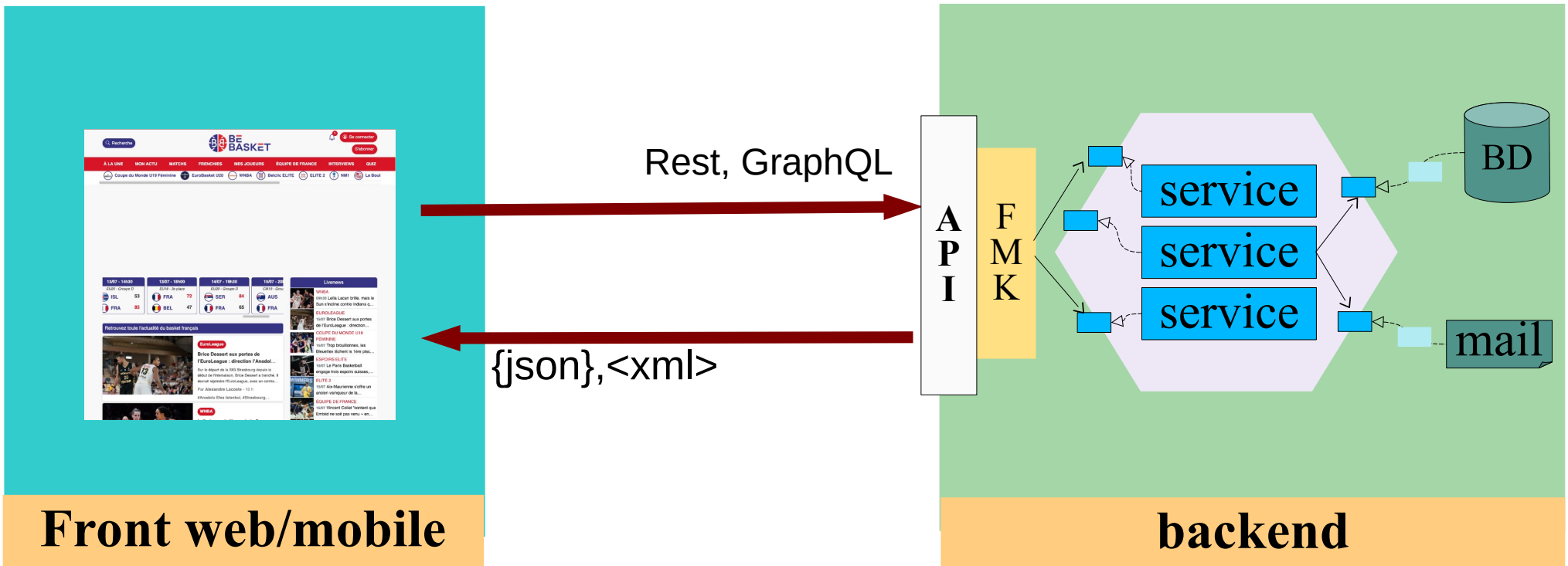
- à minimiser le couplage et les dépendances entre composants
- La meilleure cohésion possible pour chaque composant

de manière à :

- minimiser l'impact d'un remplacement ou d'un changement au sein d'un composant sur les autres composants
- obtenir des composants testables individuellement
- faciliter les changements au sein de chaque composant
- obtenir des composants réutilisables et portables

Contexte : application monopage (SPA)

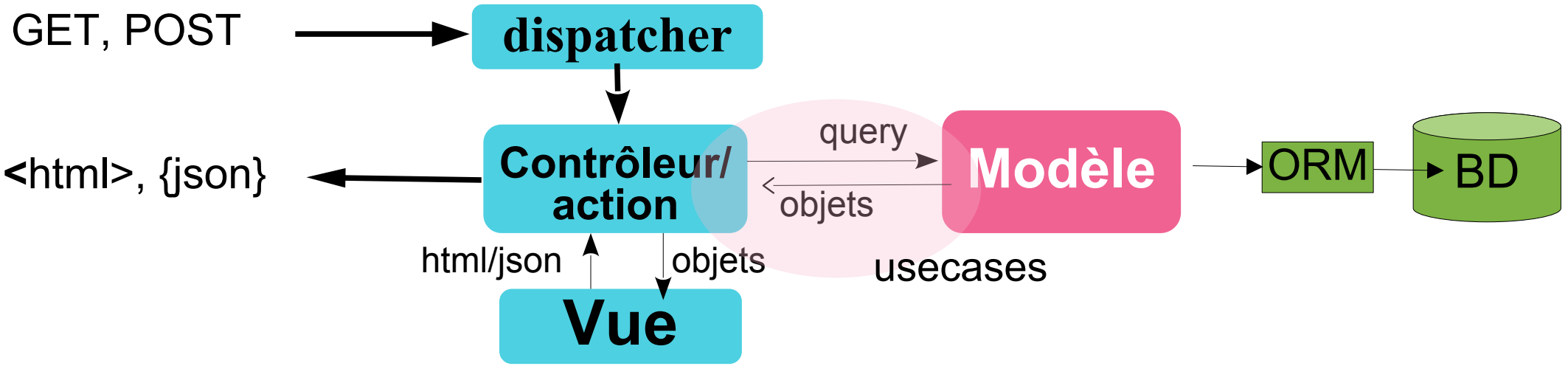
- L'interface est générée et exécutée côté frontend



Intérêt

- Le frontend dépend uniquement de l’API
- Plusieurs frontend peuvent partager la même API
- Frontend et Backend peuvent être développés et testés séparément
- Le backend peut évoluer sans impacter le frontend tant qu’il respecte l’API
- Le frontend peut évoluer sans impacter le backend tant qu’il respecte l’API

Architecture du backend : MVC



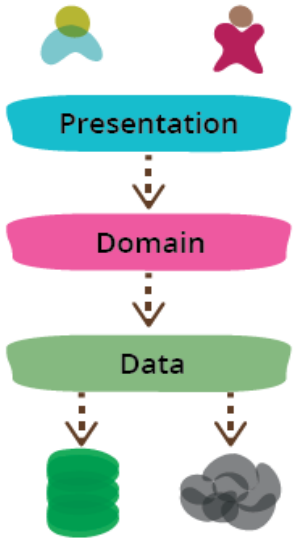
- Intérêt :
  - séparation des préoccupations
  - structuration simple
- Limites :
  - Le métier est réparti dans les contrôleurs (usecases) et les modèles : il dépend du framework et de l'ORM
  - Couplage fort et cohésion faible

conséquences

- Les fonctionnalités métier au coeur de l'application
  - → sont **impossibles à tester** individuellement, et en dehors du framework et de l'ORM
  - → sont **non réutilisables** dans différentes applications
  - → sont **difficiles à maintenir et à faire évoluer**
  - → sont **difficiles à porter** sur une infrastructure différente

- Bien adapté pour des applications de petite taille et/ou à faible durée de vie
- Avec un domaine métier simple et réduit

Architecture 3-tiers
Présentation-métier-infrastructure



- 3 couches de composants
- Chaque couche utilise uniquement des composants de la couche inférieure

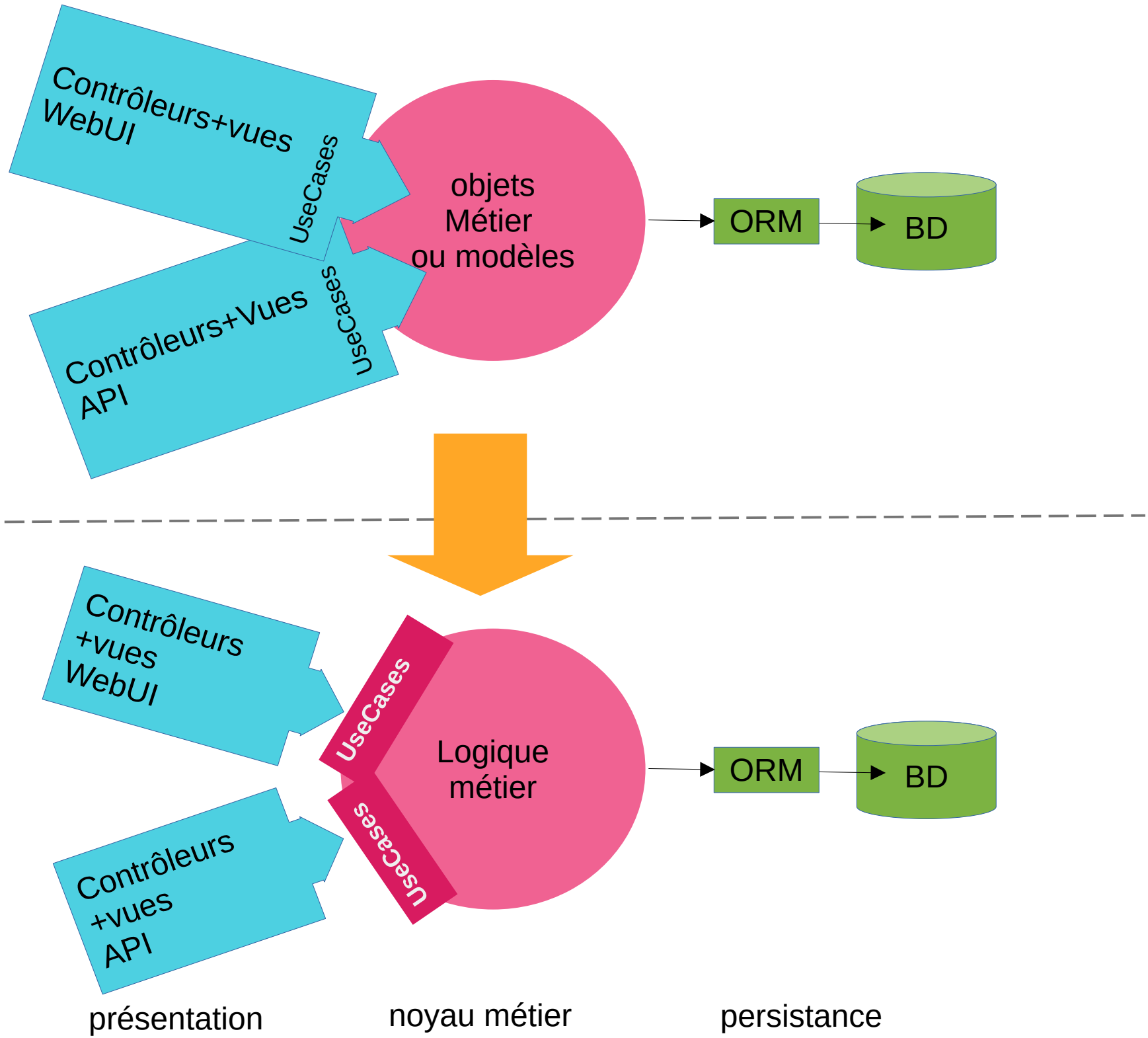
https://martinfowler.com/bliki/PresentationDomainDataLayering.html

- **Présentation** : composants gérant l'interaction avec l'utilisateur
- **Noyau applicatif métier** : composants réalisant l'ensemble des fonctionnalités métiers : **cas d'utilisation**, entités du domaine métiers, règles de gestion
- **Données, persistance ou infrastructure** : gestion de l'infrastructure d'exécution pour la persistance des entités,

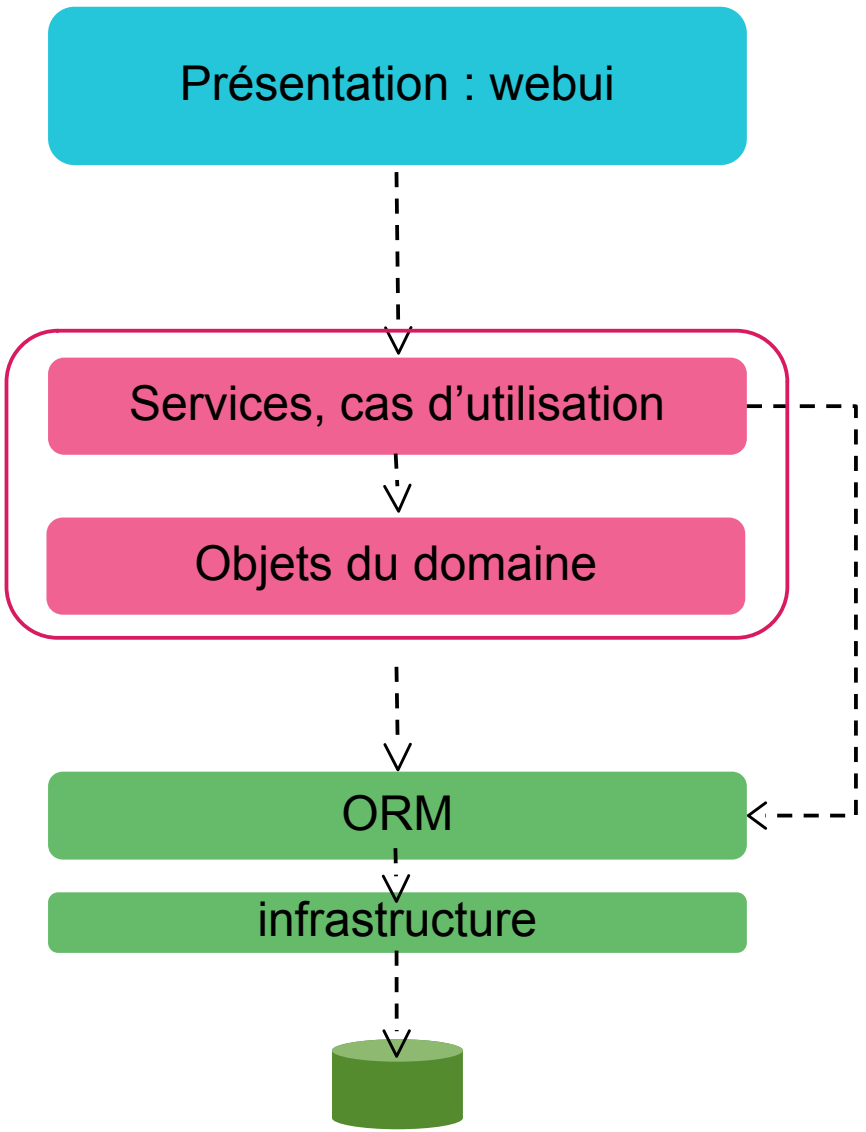
MVC

vs.

3-tiers



Application au cas des applications web



Traitement des requêtes/réponses HTTP
Gestions des données, construction des vues,
Gestion des erreurs
Basée sur un framework web

Cas d'utilisation utilisables dans les actions, enchaînement des fonctionnalités de base, création et sauvegarde des entités/objets du domaine

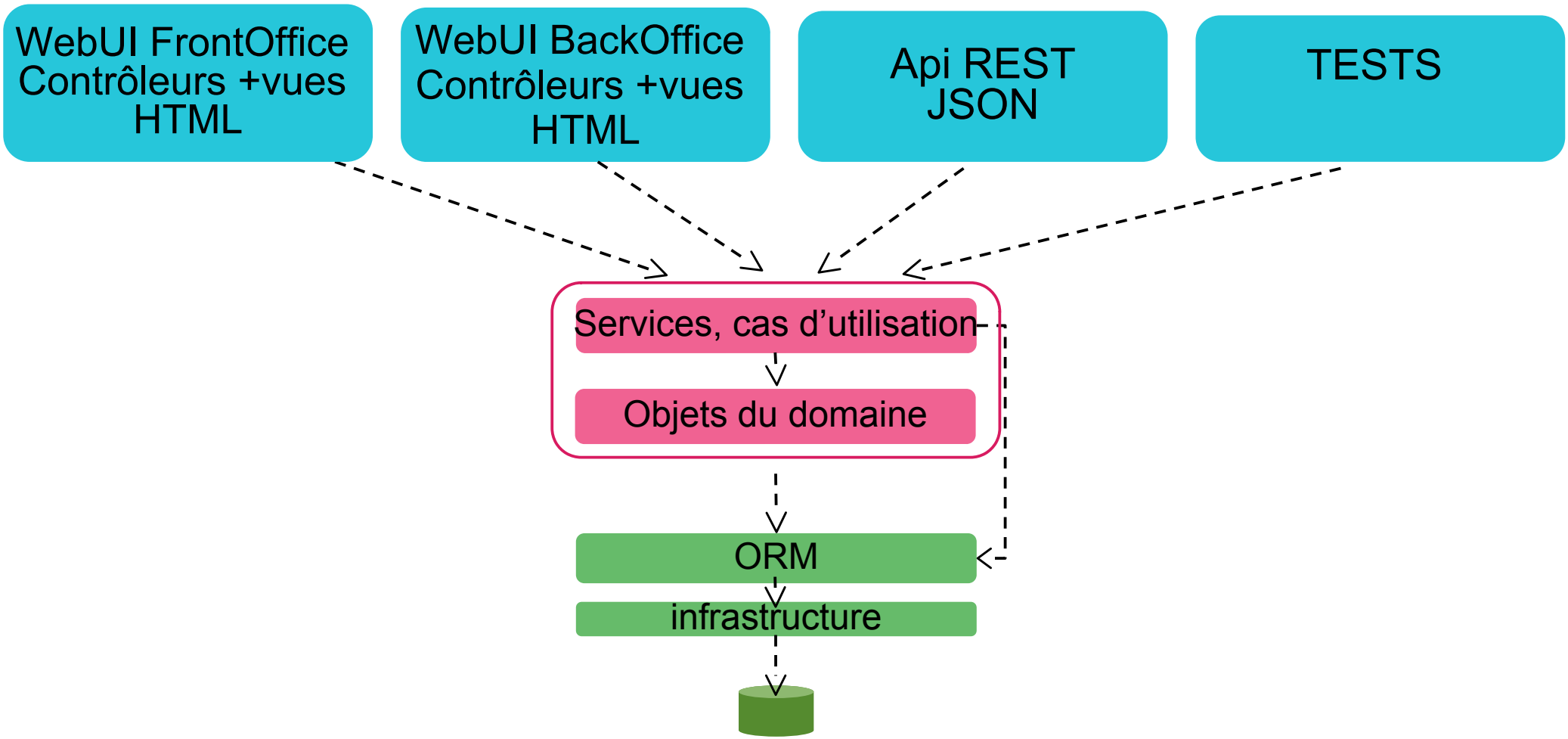
Entités et fonctionnalités métier, règles de gestion

ORM : mapping des entités vers le système de persistance

Intérêt

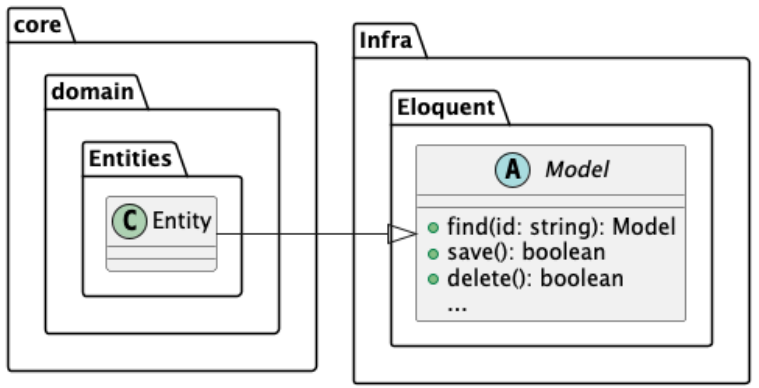
Les **fonctionnalités métiers** sont isolées et non dépendantes du framework web qui interagit avec le client, et donc :

- Elles sont réutilisables pour construire différentes applications
- Elles sont plus facilement **testables** individuellement, en dehors de l'interface



Limites

- Le métier, et notamment les entités, sont dépendantes de l'infrastructure, en particulier lorsque l'on utilise un ORM type Eloquent

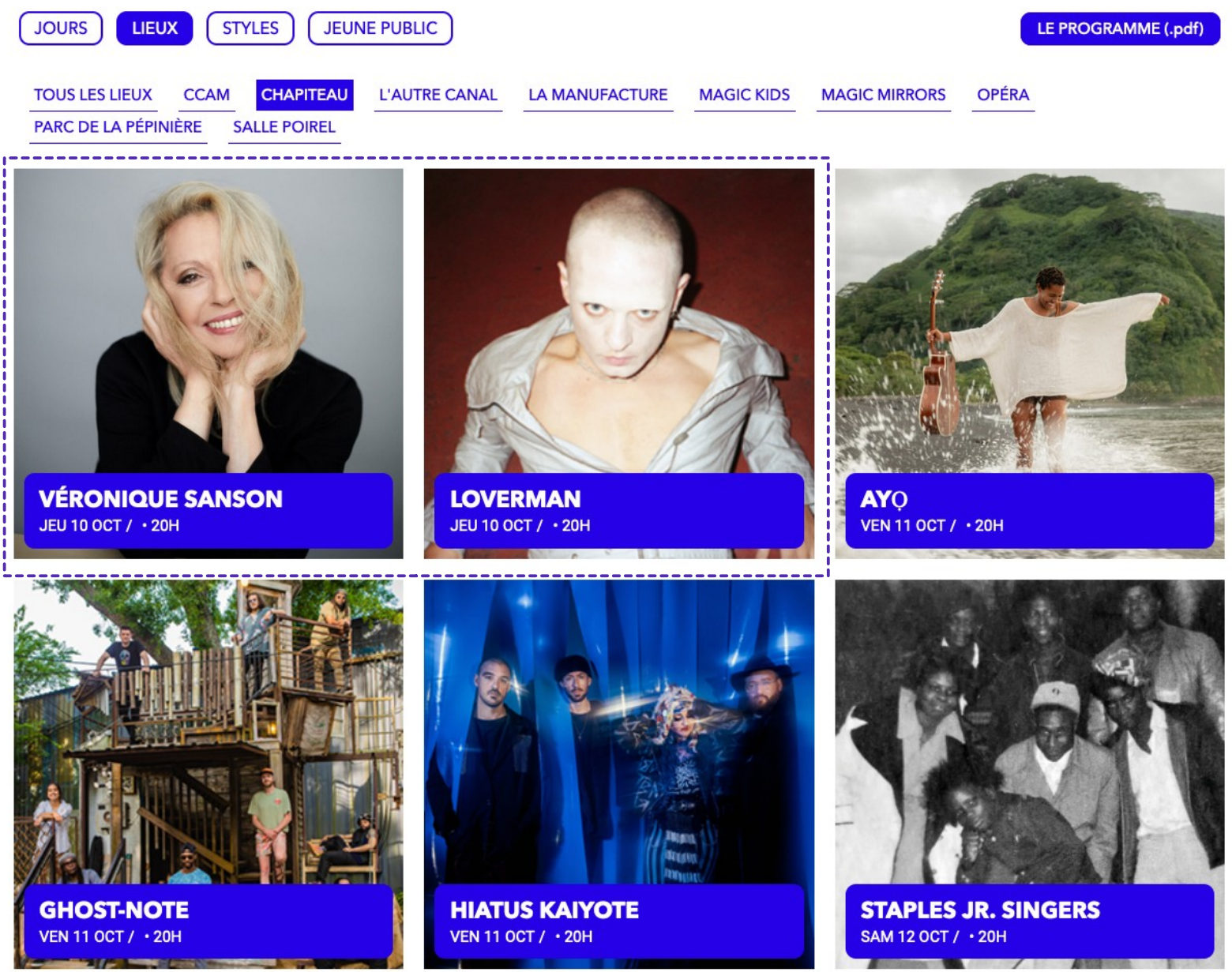


- Le métier difficilement testable en dehors de l'infrastructure
  - Impossible de tester les entités sans Eloquent et sans BD
- L'infrastructure n'est pas interchangeable, le métier n'est pas portable
  - Très difficile de changer d'ORM ou de système de persistance (par exemple SQL → mongoDB)

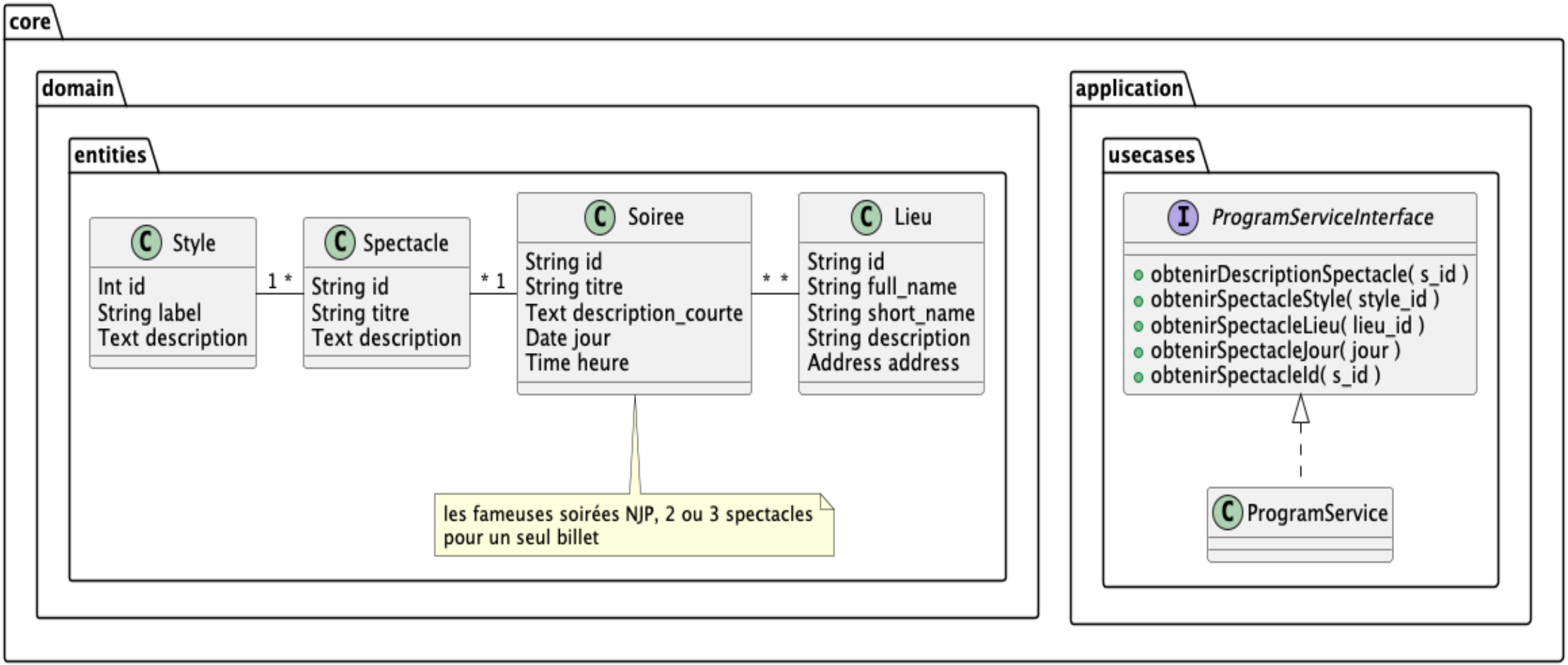
Exemple : la programmation du NJP

1 soirée

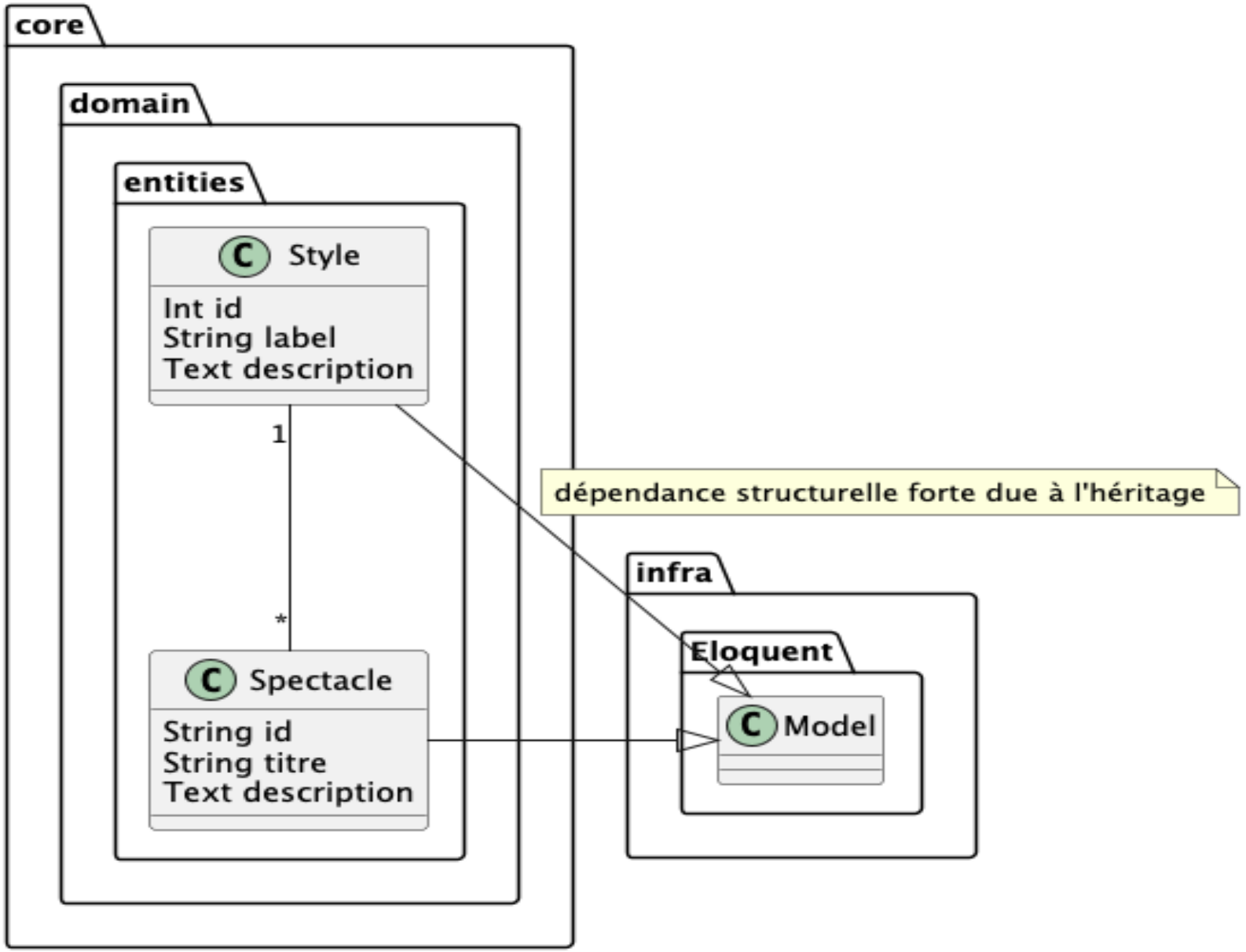
+ spectacles



La programmation du NJP : noyau métier pour la gestion du programme



Noyau NJP : la dépendance métier->infra



Rappel de conception objet

- Faites appel à vos souvenirs de conception objet 1ère et 2ème année

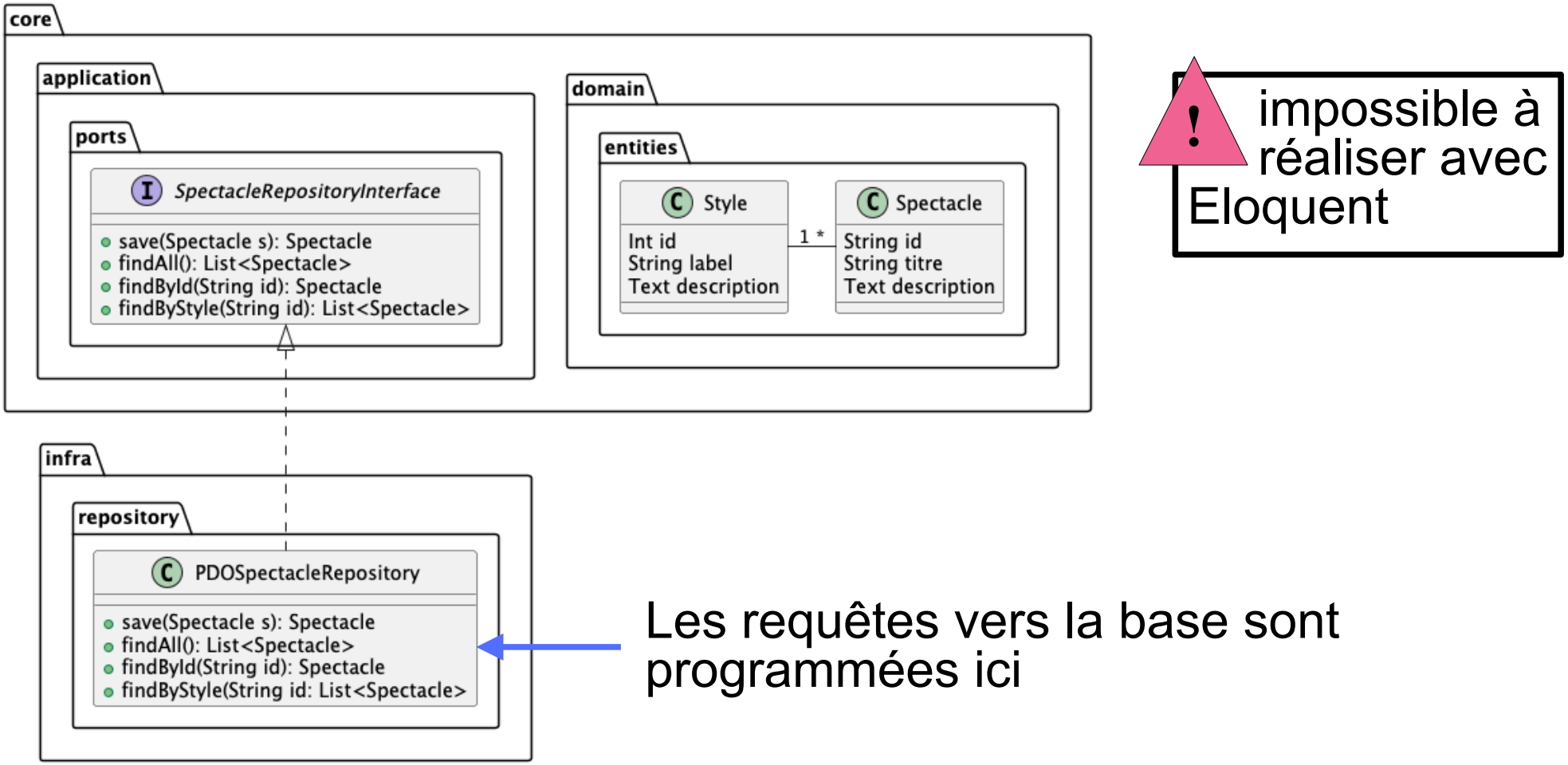
- **Comment peut-on faire pour inverser cette dépendance ?**

Réponse

- En ajoutant une **interface** et en utilisant le pattern **Adaptateur**
  - → l'interface est définie par le métier
  - → l'adaptateur est implanté par l'infrastructure

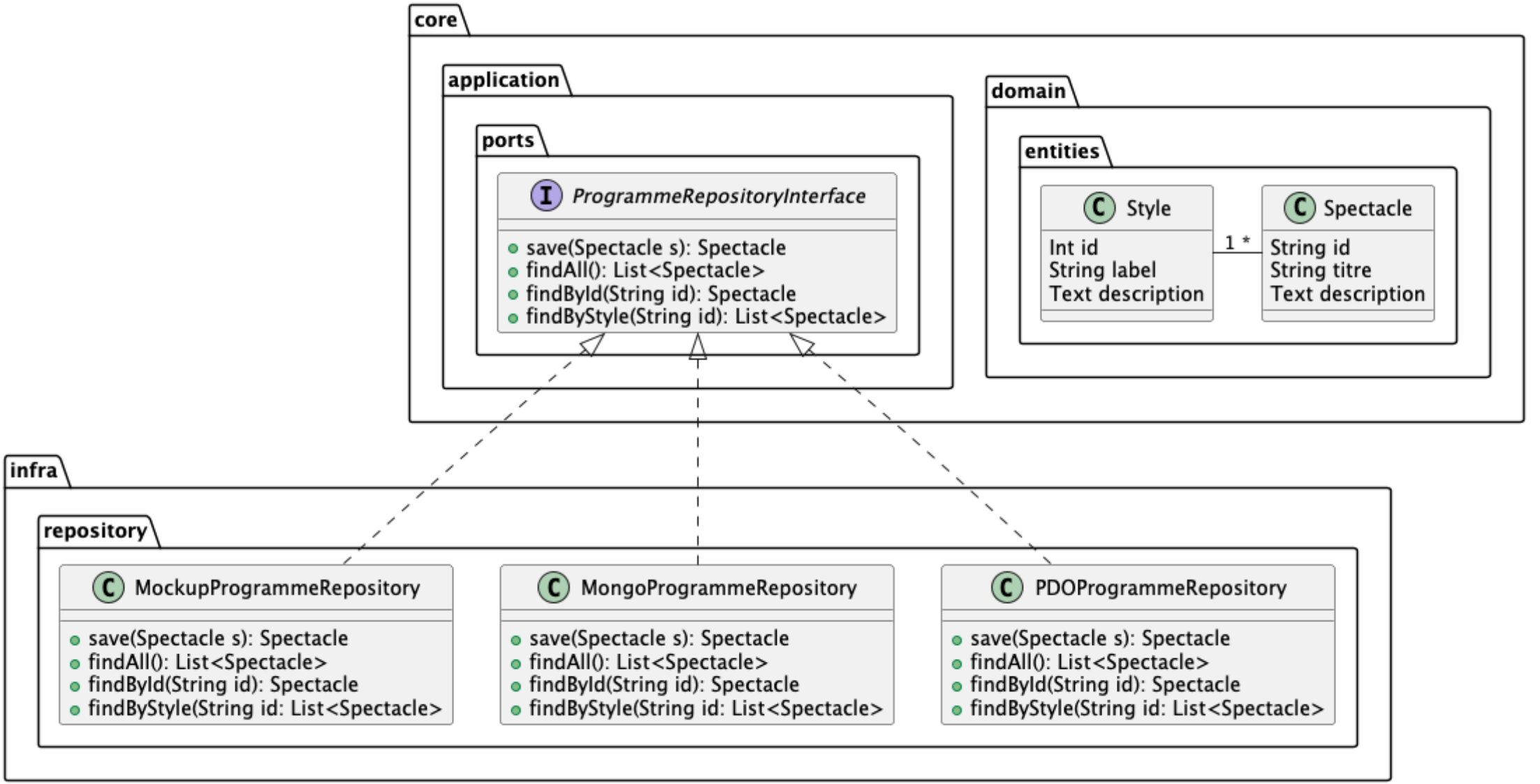
- **La dépendance est inversée : l'infrastructure implante une interface métier**
- L'adaptateur est un objet intermédiaire entre les entités et la persistance, capable de
  - – obtenir des entités métier dans la base
  - – insérer/sauvegarder des entités dans la base

→ Pattern Repository



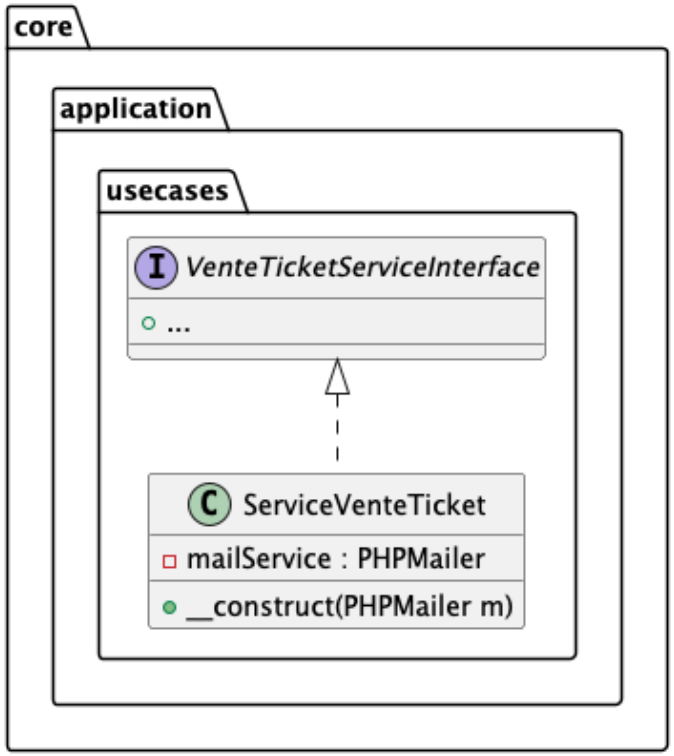
- L'objet Repository implante une interface définie par le métier : il dépend de cette interface
- L'objet Repository est un intermédiaire entre les entités et la base, il est remplaçable

Intérêt


core
application
ports
ProgrammeRepositoryInterface
save(Spectacle s): Spectacle
findAll(): List<Spectacle>
findById(String id): Spectacle
findByStyle(String id): List<Spectacle>
domain
entities
Style
Int id
String label
Text description
1 *
Spectacle
String id
String titre
Text description
infra
repository
MockupProgrammeRepository
save(Spectacle s): Spectacle
findAll(): List<Spectacle>
findById(String id): Spectacle
findByStyle(String id: List<Spectacle>
MongoProgrammeRepository
save(Spectacle s): Spectacle
findAll(): List<Spectacle>
findById(String id): Spectacle
findByStyle(String id: List<Spectacle>
PDOProgrammeRepository
save(Spectacle s): Spectacle
findAll(): List<Spectacle>
findById(String id): Spectacle
findByStyle(String id: List<Spectacle>

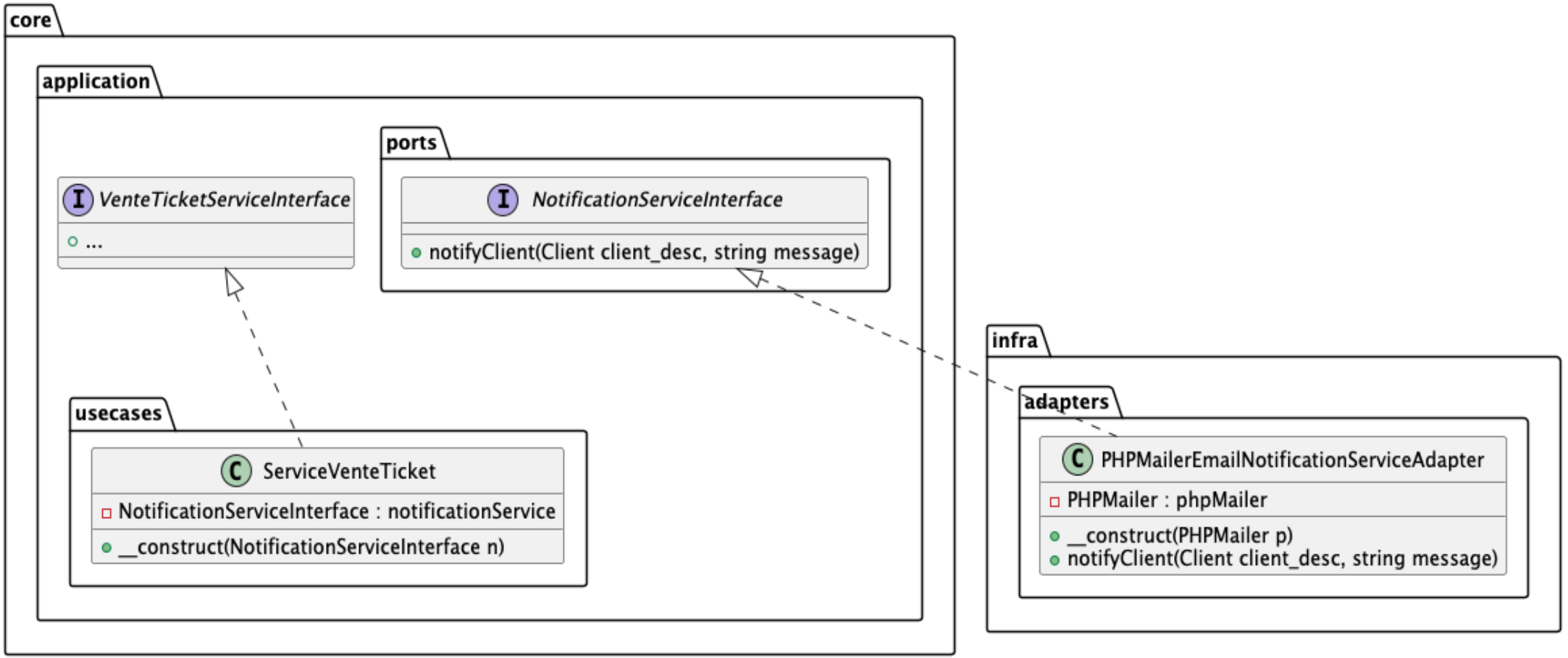
Autres dépendances métier -> infrastructure

- Exemple : le service de vente de tickets NJP souhaite envoyer des notification de confirmation par mail aux acheteurs
  - Service de vente de tickets : **noyau métier**
  - Service d'envoi de mail : **infrastructure**
- Si on ne fait pas attention, on crée une dépendance dans le sens service de vente → service mail
- ➔ Le service métier dépend de l'infrastructure :-(



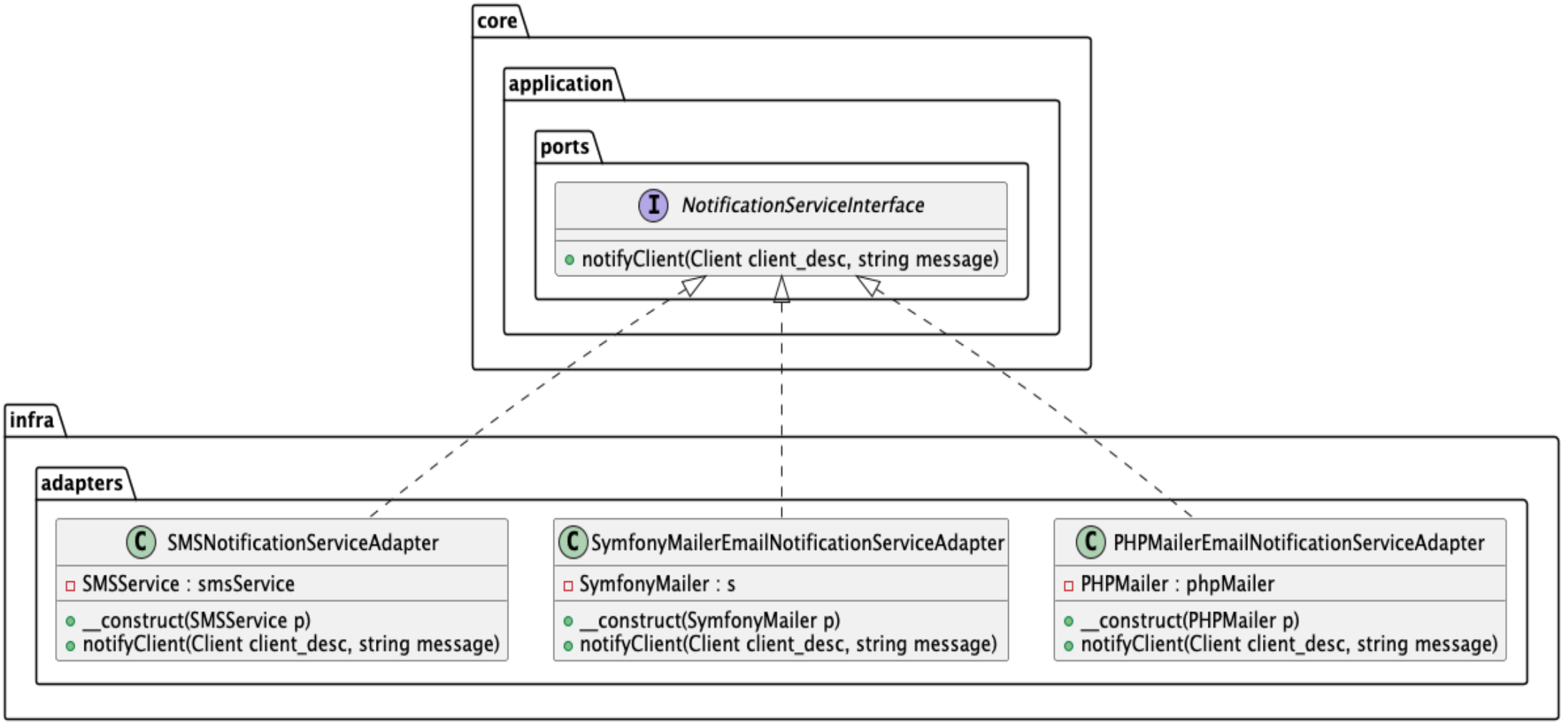
Inversion de la dépendance

- On applique le même principe : interface + pattern adapter



Intérêt

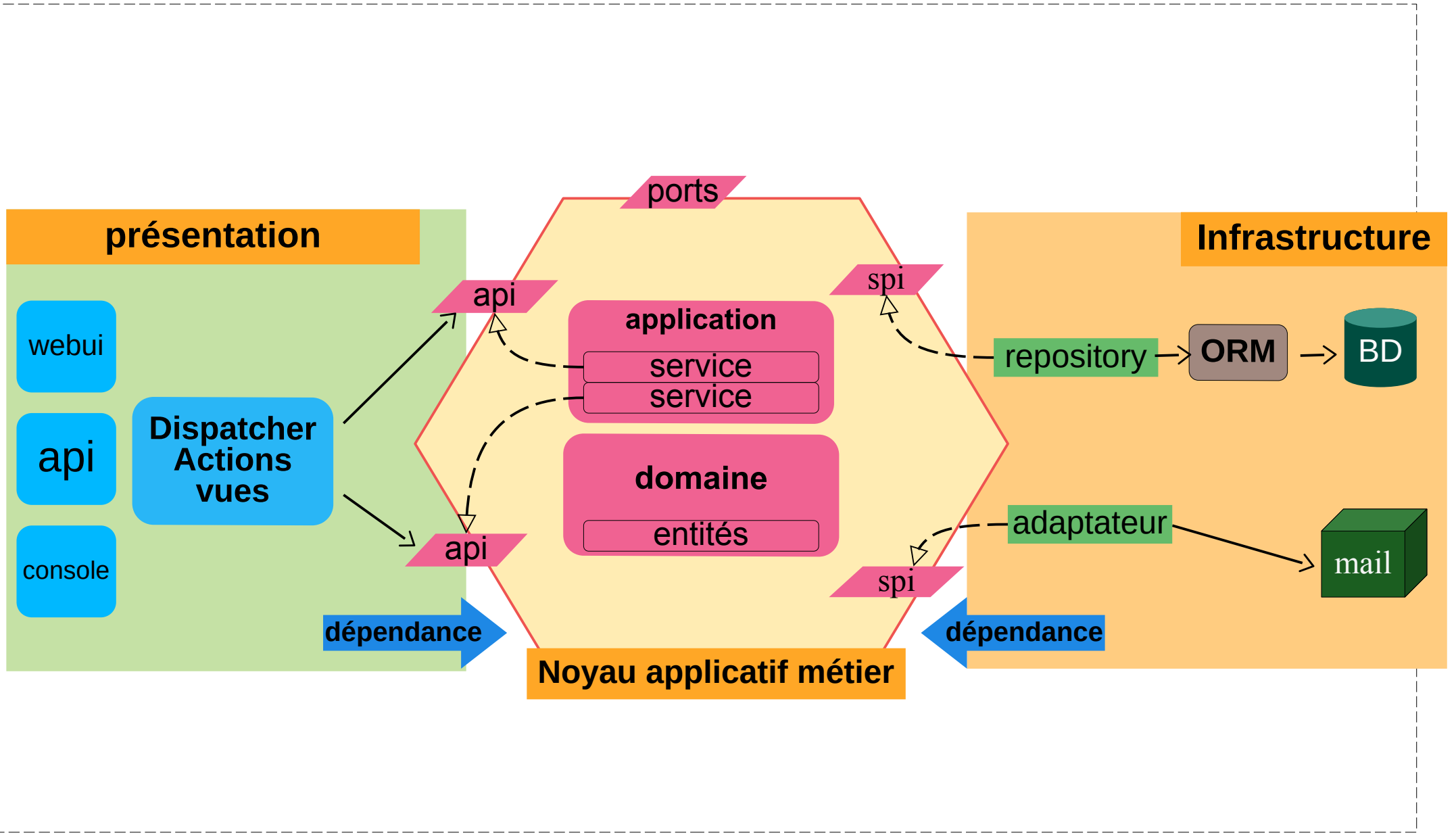
- Le service de notification devient remplaçable :



Architecture Hexagonale

- L'architecture hexagonale est un principe d'architecture logicielle basé sur l'architecture en couche dans laquelle on réalise l'inversion de dépendance avec l'infrastructure
- Le noyau applicatif métier est au coeur de l'architecture
- Il échange avec la couche présentation et l'infrastructure au travers de *ports* exposant des interfaces
  - Ports d'entrée : interfaces utilisées par les actions, il s'agit de l'API du noyau métier
  - Ports de sortie : interfaces implantées par l'infrastructure vue comme un provider pour les services métiers (SPI)

Architecture Hexagonale pour le backend



Architecture hexagonale : intérêt

- Le noyau métier **ne dépend de rien** et donc :
- Il est indépendant des choix techniques : framework, SGBD, communication ...
- Il est **testable seul**
  - notamment, pas besoin du framework web ou de requêtes HTTP pour le tester
  - pas besoin de base de données ou d'ORM pour le tester
- Il est **réutilisable** pour construire différentes interfaces
  - appli html, API json web et/ou mobile, console, standalone
- Il est **portable** d'une infrastructure à une autre
  - BD interchangeable, messagerie interchangeable ...

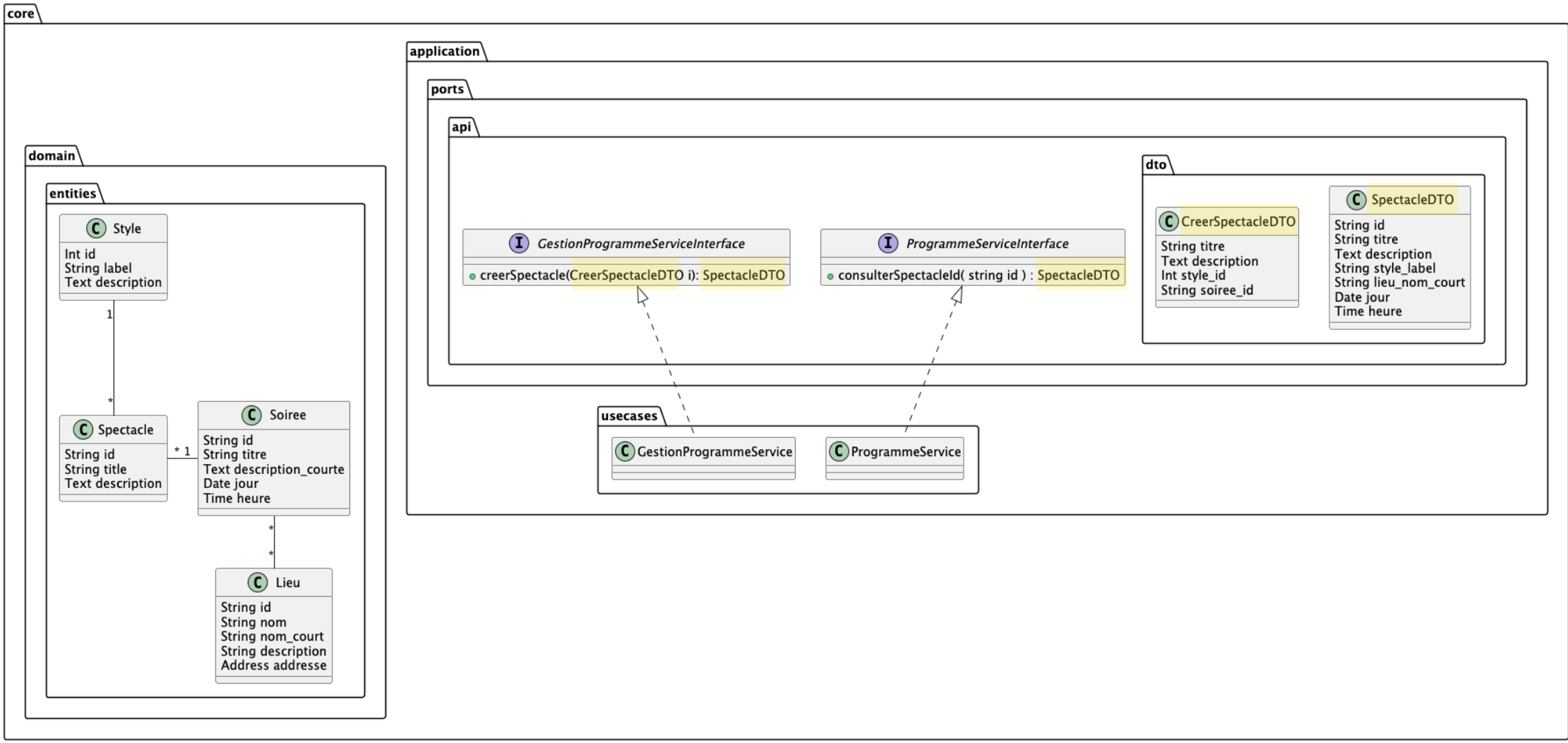
Échanges de données avec les services métier

- Les services métiers sont appelés par la couche présentation ou par d'autres services au travers de leur api
- Les méthodes de ces api reçoivent des données et retournent des données
- **On doit garantir que les données transmises aux services métiers sont bien formées sans utiliser les entités du domaine métier pour limiter le couplage**
- On utilise le pattern DTO : Data Transfer Object

La notion de DTO

- Un **DTO** est un objet neutre servant uniquement à transférer des données d'un composant à un autre
  - des actions vers les services (données issues d'un formulaire)
  - des services vers les actions (données résultat)
  - d'un service métier vers un autre service métier
- Les DTO sont utilisés pour typer les paramètres et résultats des méthodes listées dans les api métier des services
- **Ils sont définis par le métier et lui permettent d'imposer la structure et le type des données qu'il échange avec l'extérieur**
- On peut créer autant de DTO que nécessaire
  - DTO en entrée, en sortie, partiels ...

Exemple : DTO Spectacle



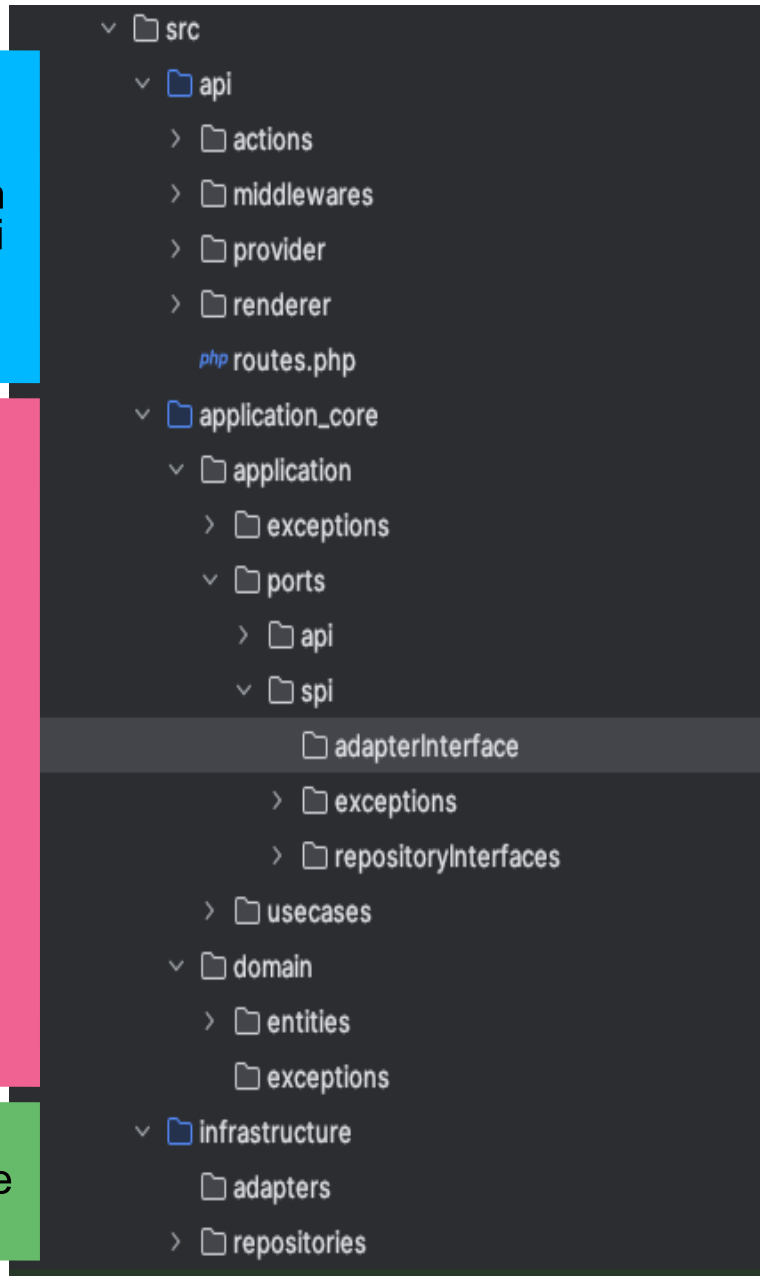
- Un DTO en entrée pour créer un spectacle
- Un DTO en sortie pour décrire un spectacle

Organisation et namespaces

présentation
api/webui/cli

noyau
applicatif
métier

infrastructure




```
njp\api
njp\api\actions
```

```
njp\core
njp\core\application
  njp\core\application\ports
  njp\core\application\ports\api
  njp\core\application\ports\spi
  njp\core\application\usecases
njp\core\domain
  njp\core\domain\entities
```

```
njp\infra
  njp\infra\adapters
  njp\infra\repositories
```