Architecture hexagonale

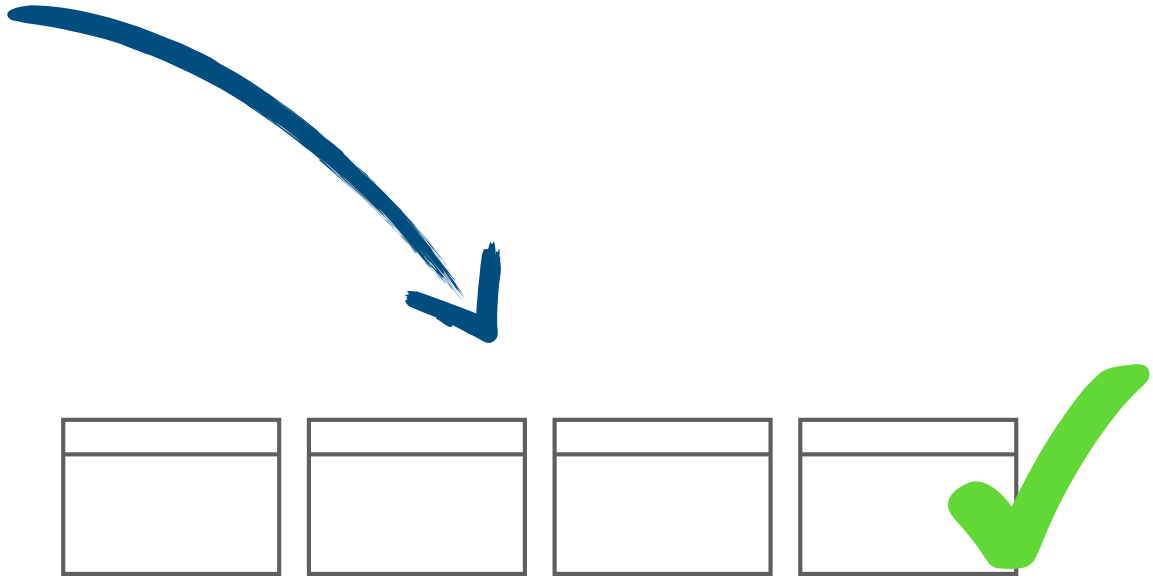
ou Clean, ou Onion, ou Ports and Adapters...

Olivier Perrin – 2023

Principes de base

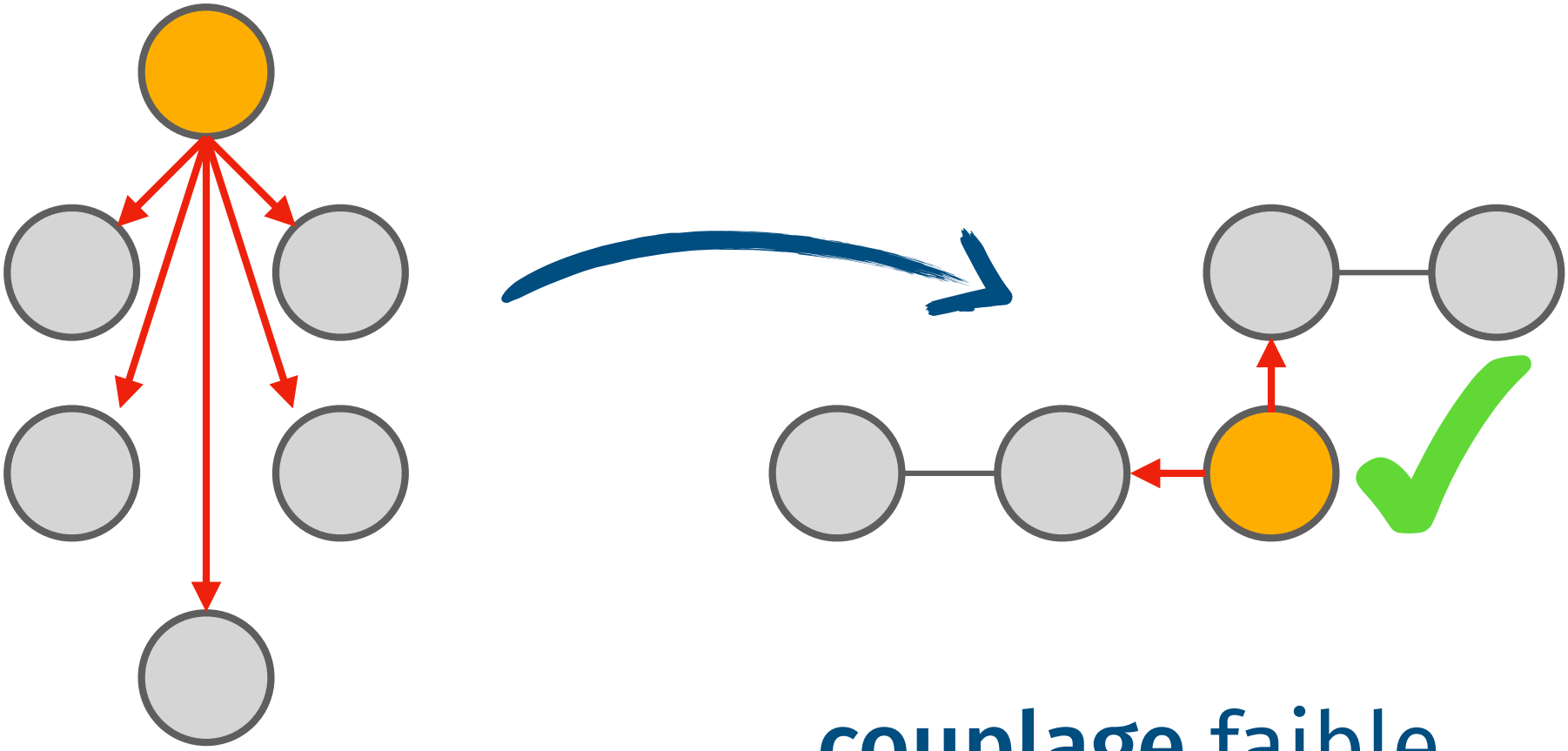
Single **R**esponsibility **P**rinciple

EmployeManager
- read/persist - compute payroll - generate PDF reports - manage projects



cohésion forte

Couplage



couplage faible

Don't **R**epeat **Y**ourself 

or **DIE**

Duplication is evil !

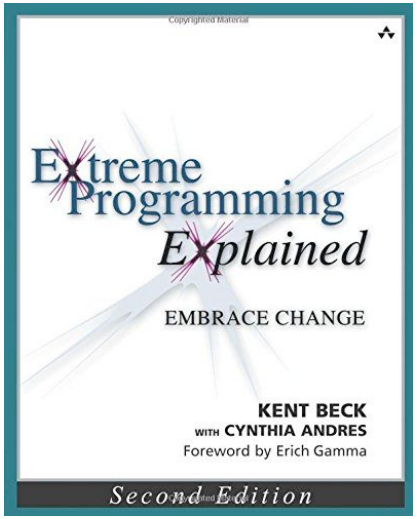
Keep **I**t **S**hort & **S**imple

La **suringénierie** est le début des ennuis

Moins de code, code plus simple →
bonheur des développeurs

Règles d'une conception simple

1. passer les tests
2. révéler l'intention
3. éviter la duplication
4. conserver le moins d'éléments possible



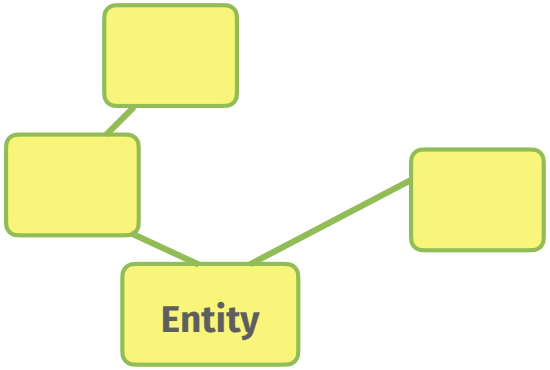
Peut-on prédire le futur ?

lutter continuellement pour
une **conception simple**

Modélisation des données

Les **entités** contiennent **vos** données persistantes

Vous les controllez !



Elles doivent simplifier votre code métier

Codez des petits fragments réutilisables de logique métier dans vos **entités de domaine**

(vs. classes anémiques)

```
public String toExportString() {
    return String.format ("%s;%s;%d",
        firstName, lastName, isActive()?1:0);
}
```

```
public class Customer {
[...]
    public String getFullName () {
        return firstName + " " +
    }
```

getters nécessaires

validation

audit

encapsulation

immutabilité

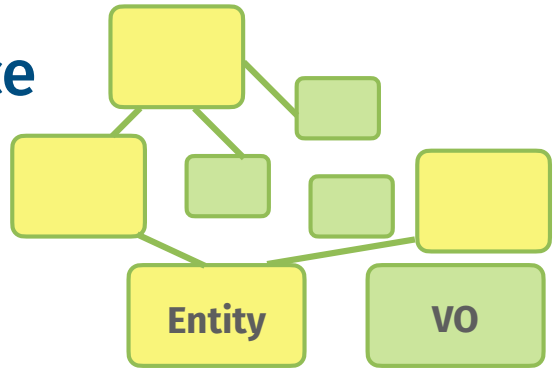
Value object: pour décrire des aspects du domaine métier, sans ID

```
public class Money {
    private final Currency currency;
    private final BigDecimal amount;
    public Money(Currency currency,
        BigDecimal amount) {
        this.currency = currency;
        this.amount = amount;
        validate();
    }
    public Currency getCurrency() {
        return currency;
    }
    public BigDecimal getAmount () {
        return amount;
    }
    public boolean equals(Object other)
    { ... }
}
```

Petits

Immuables !

Pas d'ID de persistence
(vs. entités)



Égaux par valeur
de tous les attributs

Intégrables
dans des entités

Ne **jamais** exposer vos entités
dans vos APIs !

Les consommateurs (SPA, service, app)
veulent des données
compatibles avec leurs UX/UI !

mais... UX/UI et métier ont des objectifs différents !

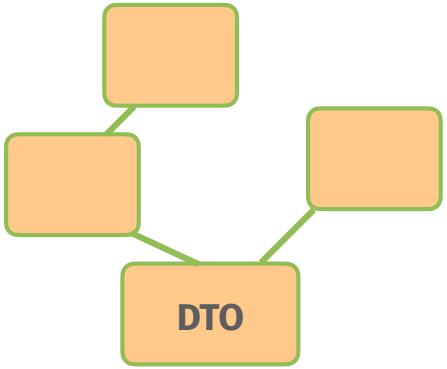
Pour exposer les données dans votre API

Data transfer object (DTO)

Forms/request ⇒

Views/response ⇐

DTO ⇔

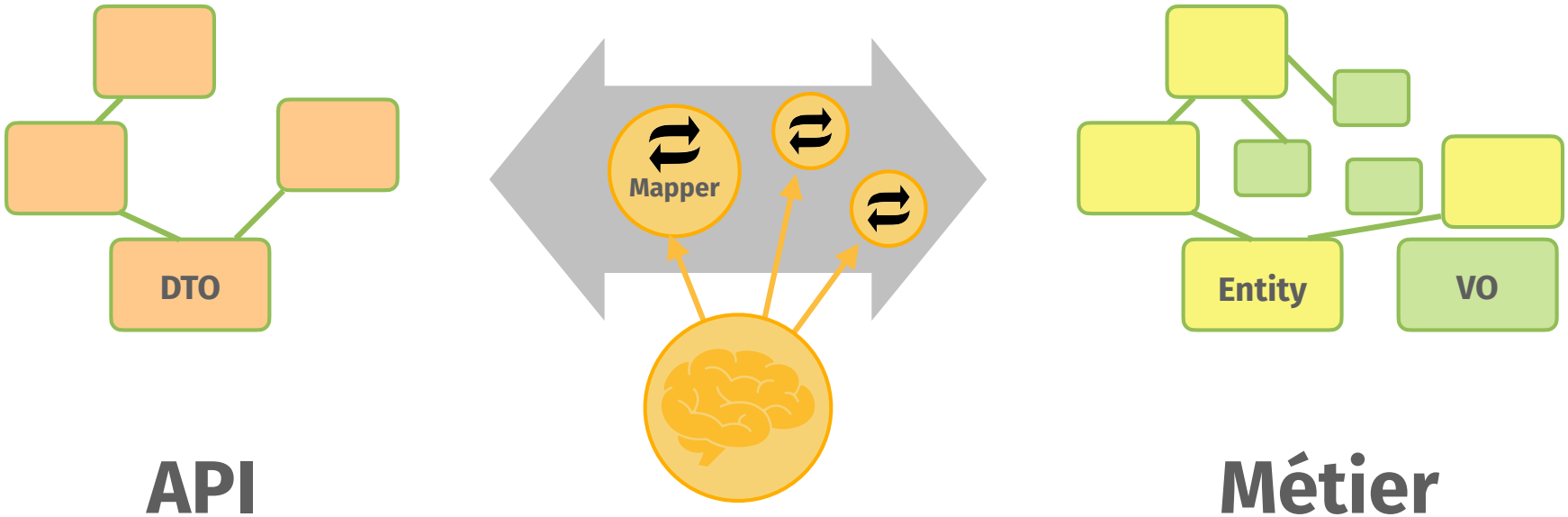


DTOs aussi **minimaux** que possible !

```
public class CustomerDto {
    private String fullName;
    private String phoneNumber;
    private Date birthDate;
    public final String getFullName(
        return fullName;
    }
    public final void setFullName()
        this.fullName = fullName;
    }
    public final String getPhoneNuml
        return phoneNumber;
    }
    public final void setPhoneNumbe
        this.phoneNumber = phoneNumb
    }
    public final Date getBirthDate(
        return birthDate;
    }
    public final void setBirthDate(
        this.birthDate = birthDate;
    }
}
```

Logique métier

Première étape: faire le lien




```
CustomerDto dto = new CustomerDto(customer);
```

```
CustomerDto dto = new CustomerDto();
dto.fullname = customer.getFullName();
dto.birthDate = customer.getBirthDate();
dto.phoneNumber = customer.getPhoneNumber();
```

On peut avoir besoin
de plusieurs DTOs
(selon les usecases)


```
public class CustomerDTO {
    public String fullName;
    public Date birthDate;

    public Date createdDate;
    public Date modifiedDate;
}
```

null dans le usecase *create*

DRY !

```
public class CustomerCommonDTO {
    public String fullName;
    public Date birthDate;
}
```

```
public class CustomerView extends CustomerCommonDto {
    public CustomerCommonDto common;
    public Date createdDate;
    public Date modifiedDate;
}
```

Composition over Inheritance [GoF]

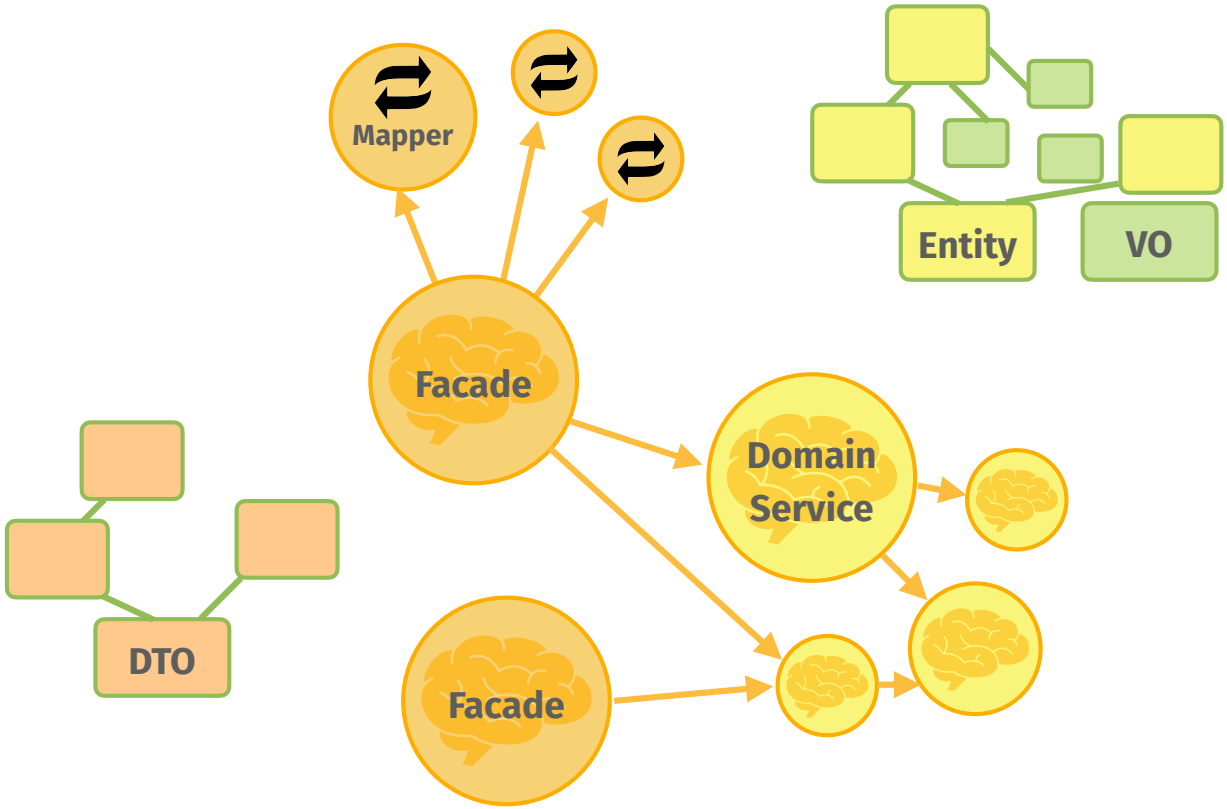
SRP

Est-ce que deux usecases métiers doivent partager du code ?

DRY

Avons-nous écrit deux fois le même code ?

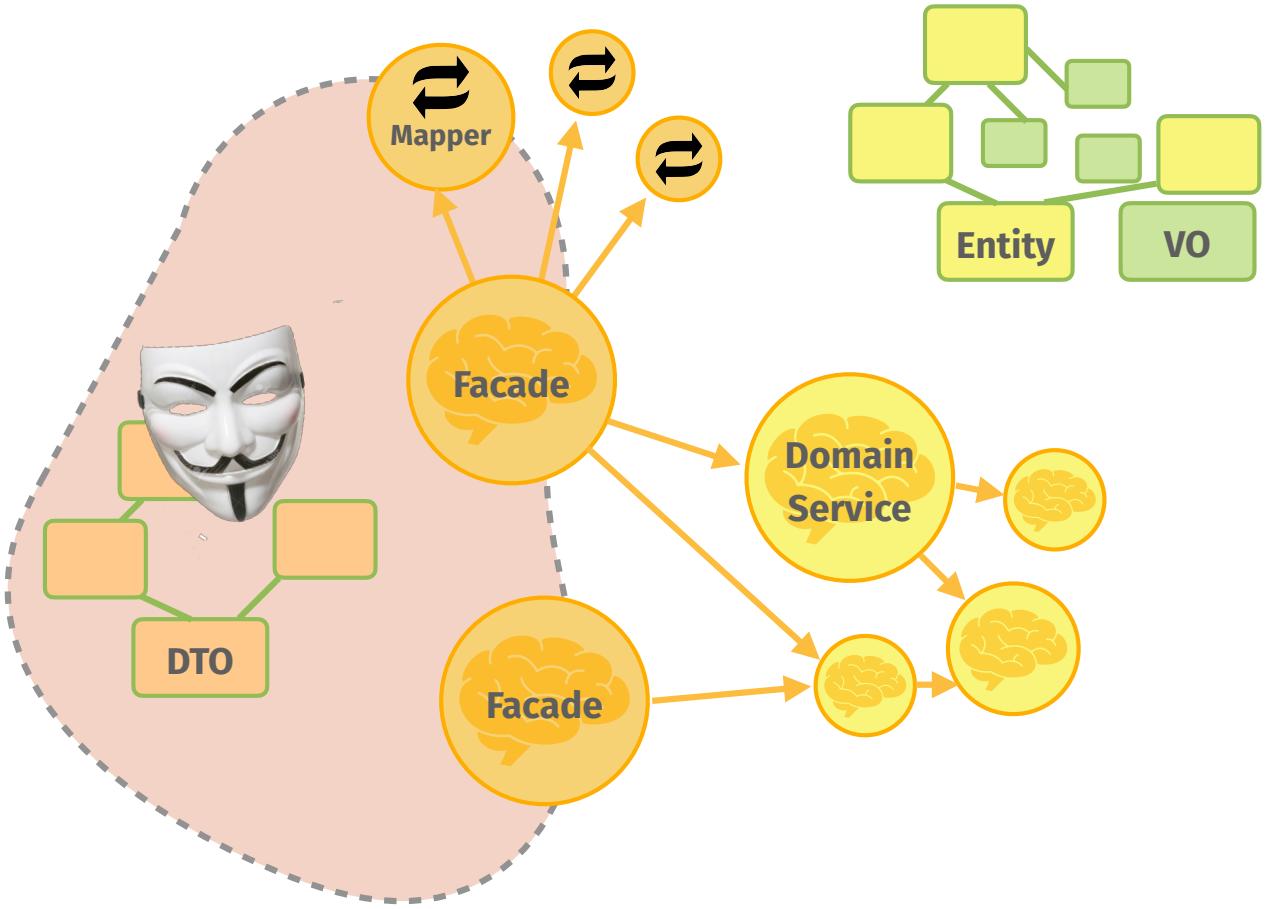
On implante ensuite le domaine métier
dans une **Facade**



... puis on extrait la logique dans des **services métier**

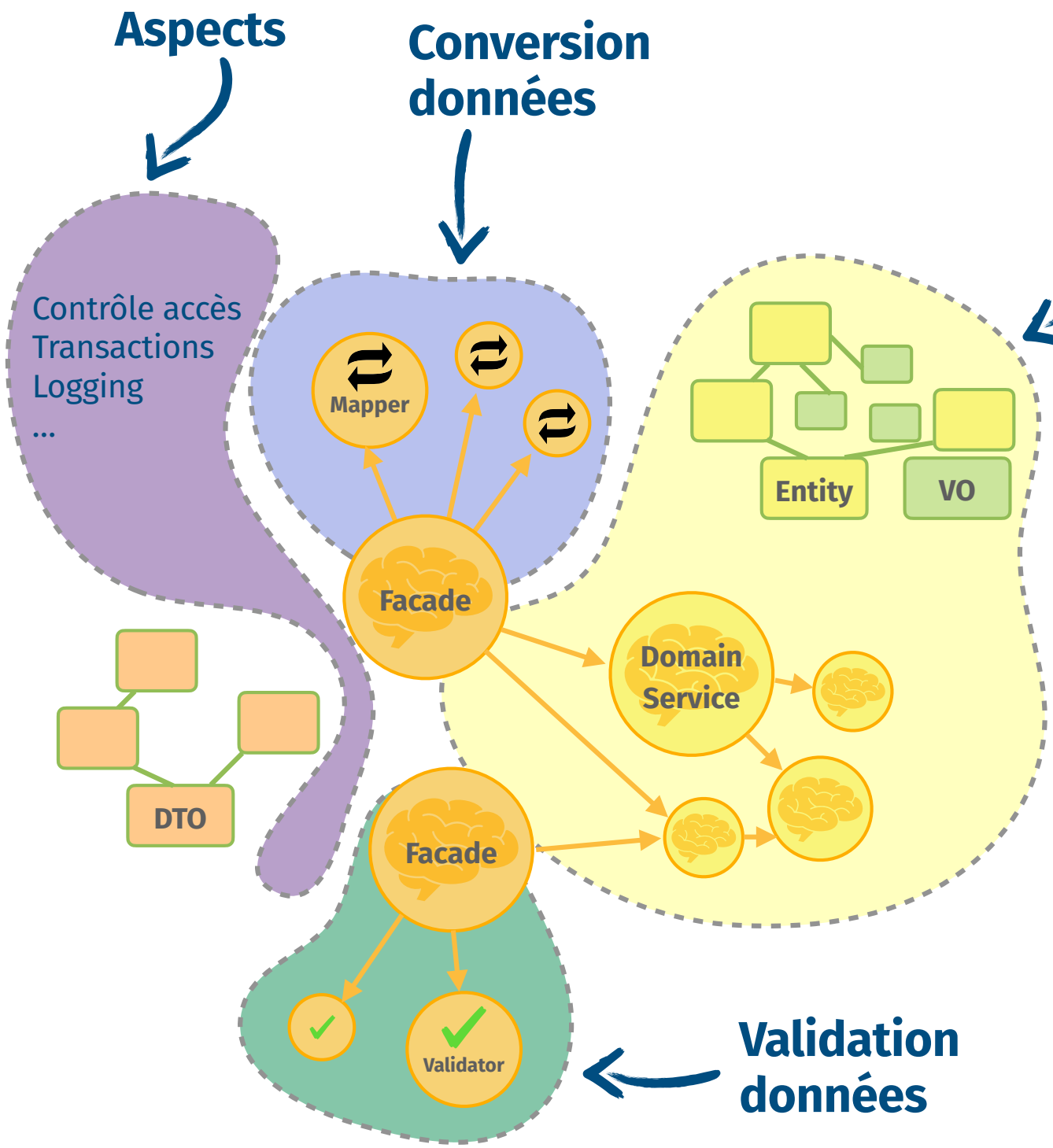
on applique **SRP** et **DRY**

Les services métier **réalisent le métier** et rien que le métier



Les DTOs sont **fragiles**, donc on les simplifie au maximum (pas de méthodes)

et on les conserve **hors** du domaine métier (on les convertit ASAP vers des objets du domaine)



Différentes **responsabilités** des façades

Dès qu'une classe devient trop *grosse*...

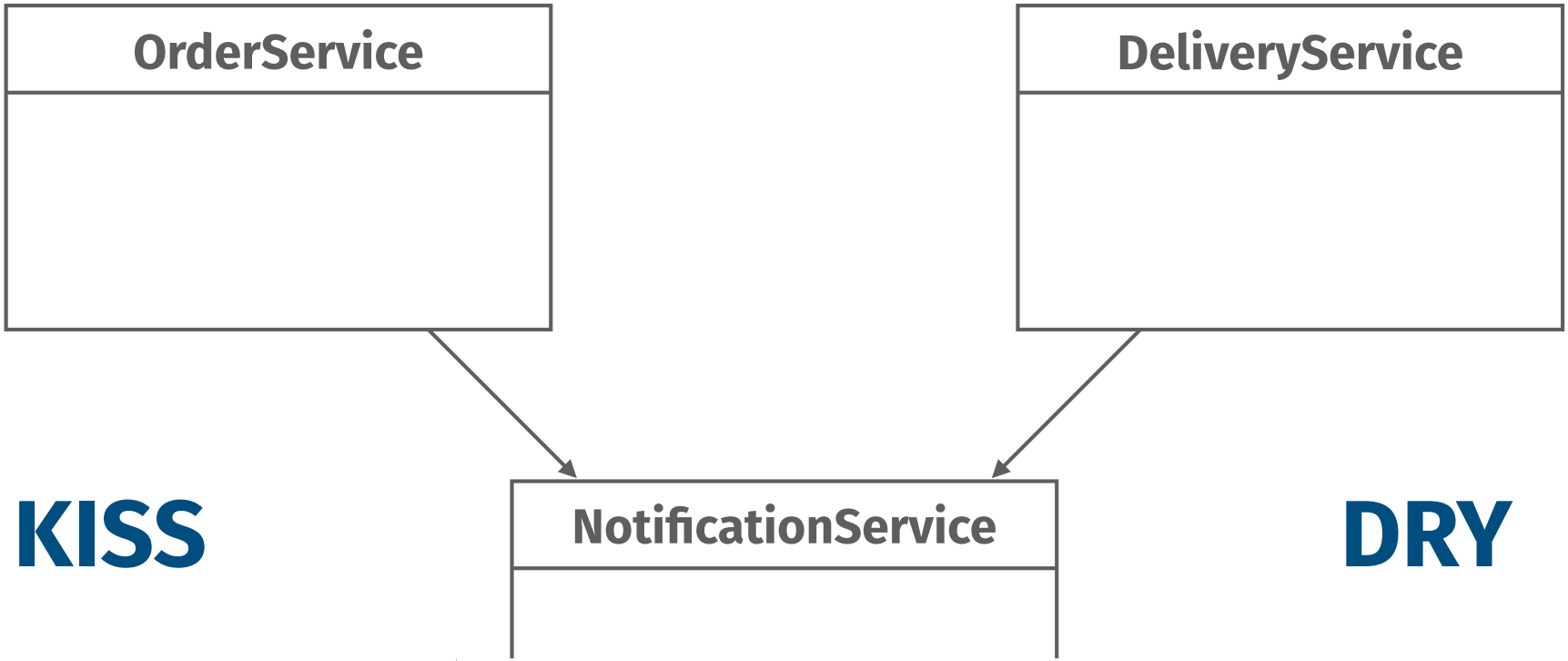
...on **extrait**

soit **horizontalement**, soit **verticalement**

Extraction horizontale

Même niveau

Extraction verticale



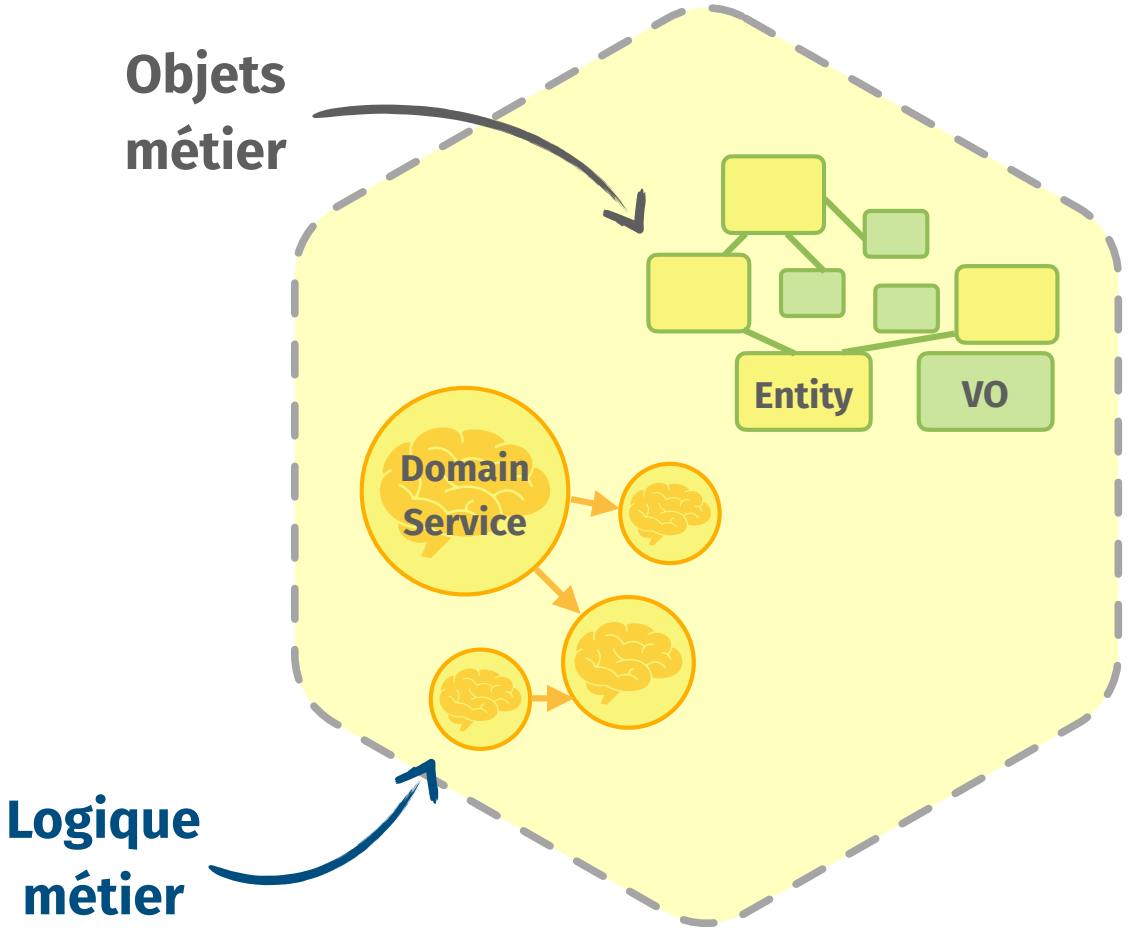
KISS

DRY

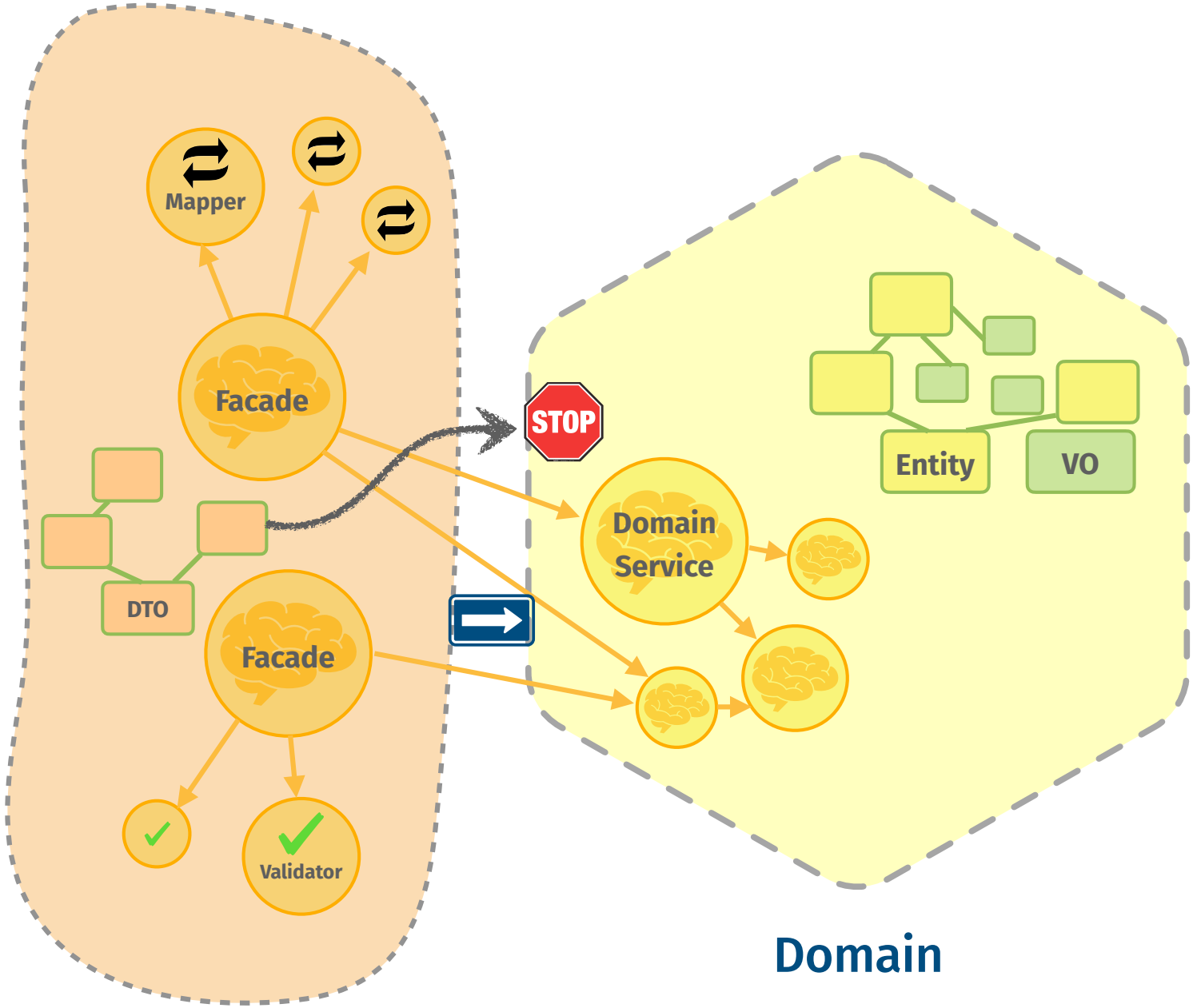
Séparation par couches d'abstraction

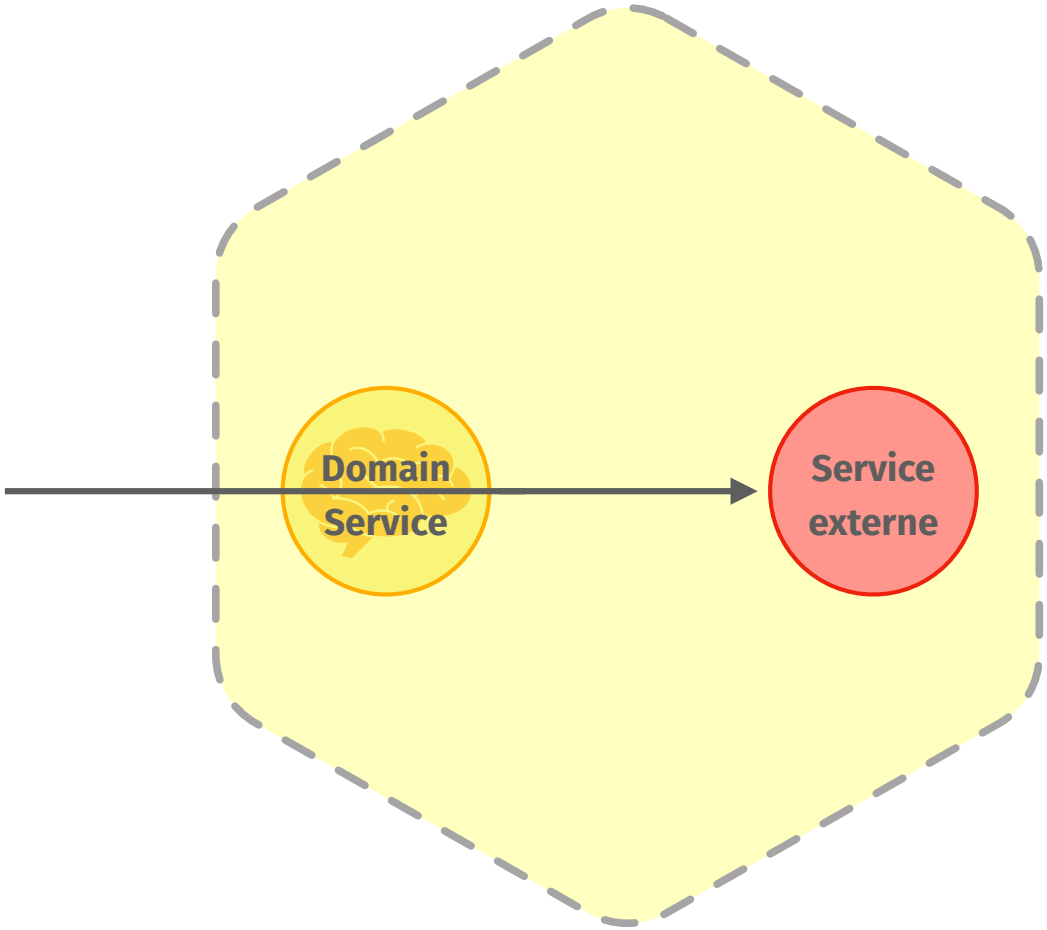
Architecture hexagonale

Quelle partie de votre code est importante ?



Le code **métier** !


Mapper
Facade
DTO
Facade
Validator
STOP
Domain Service
Entity
VO
Application
(dépend du domaine)
Domain



Houston, we'got a problem !

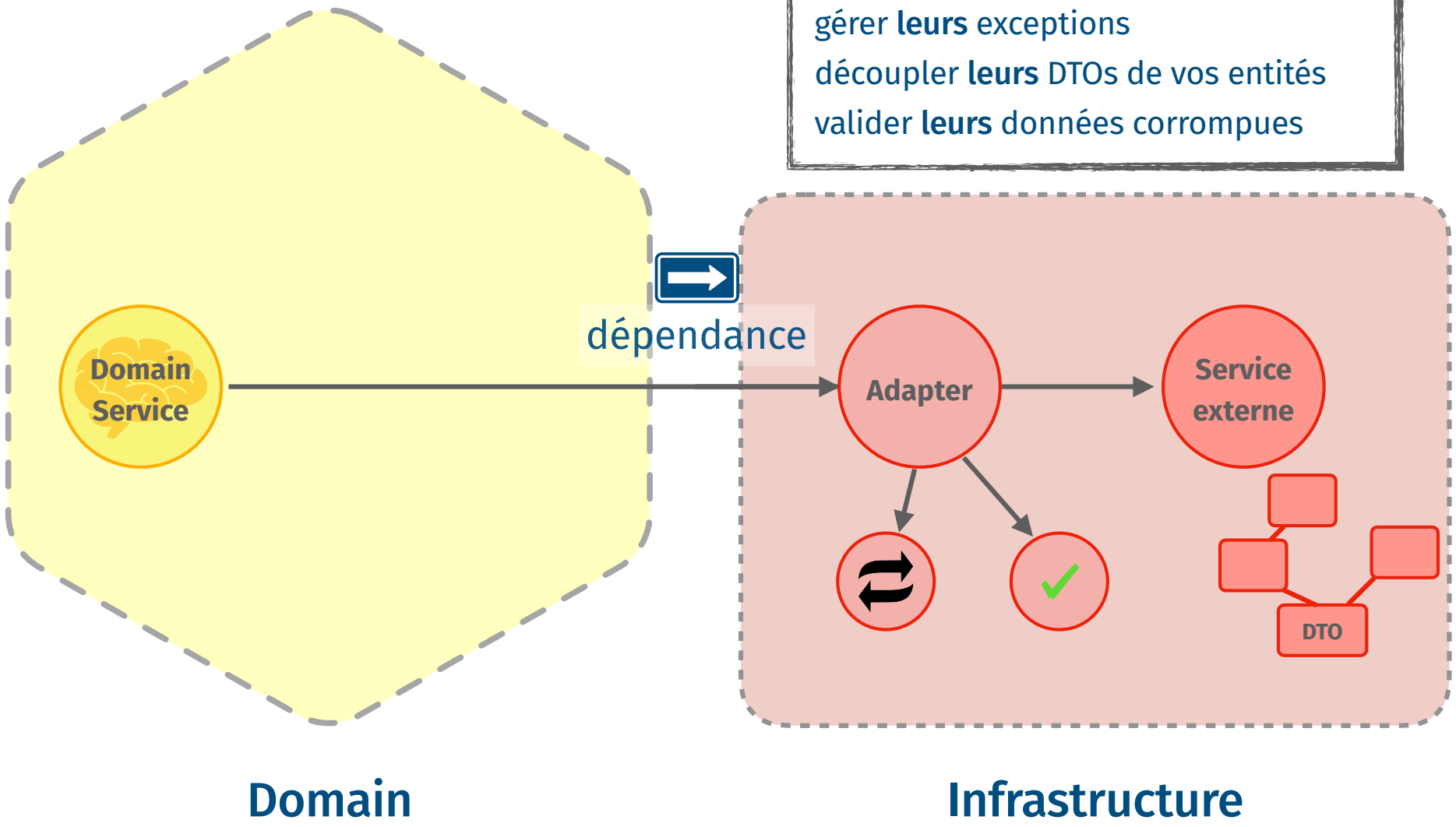
Adapter pattern

cacher **leur** horrible API

gérer **leurs** exceptions

découpler **leurs** DTOs de vos entités

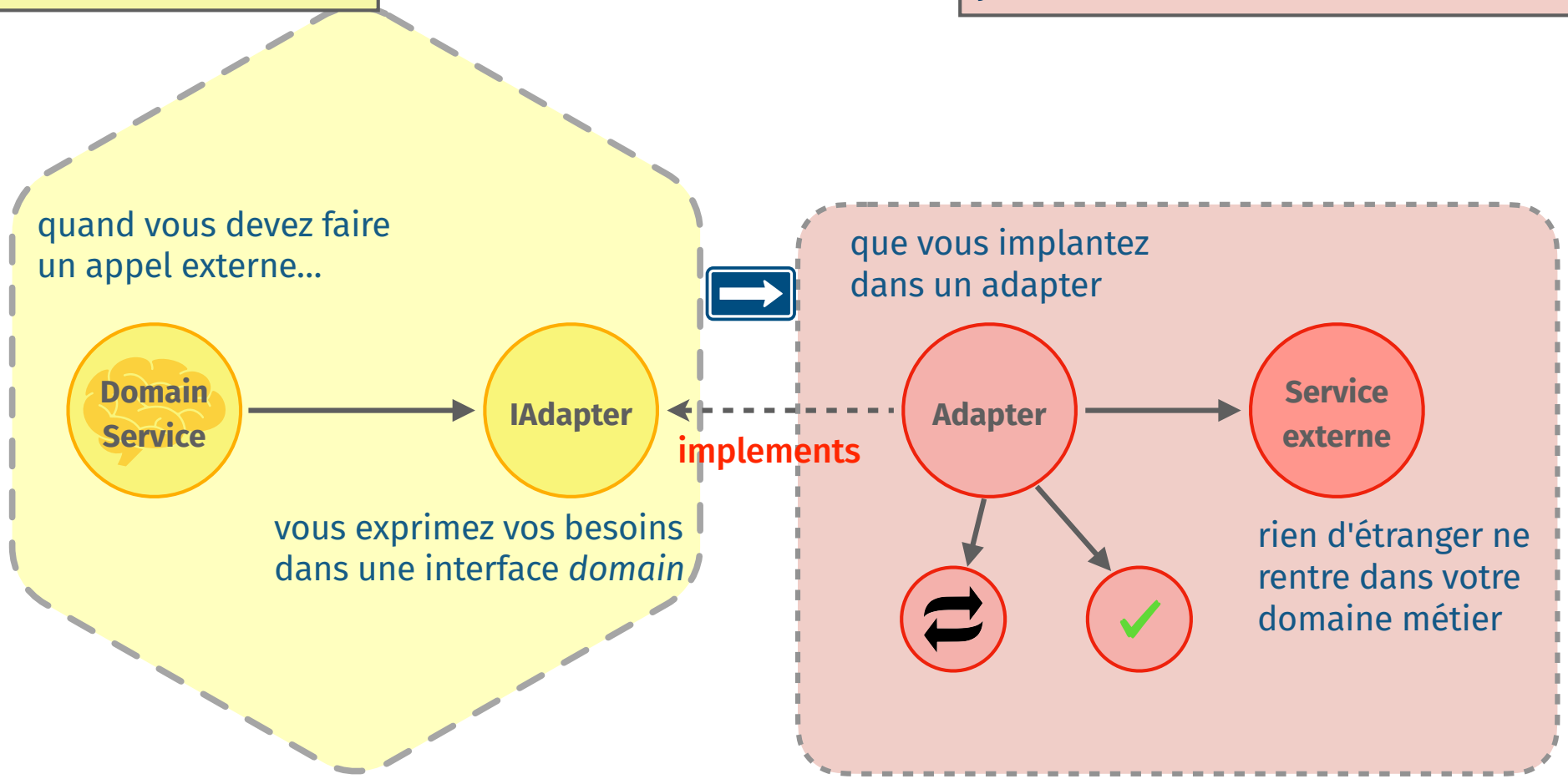
valider **leurs** données corrompues



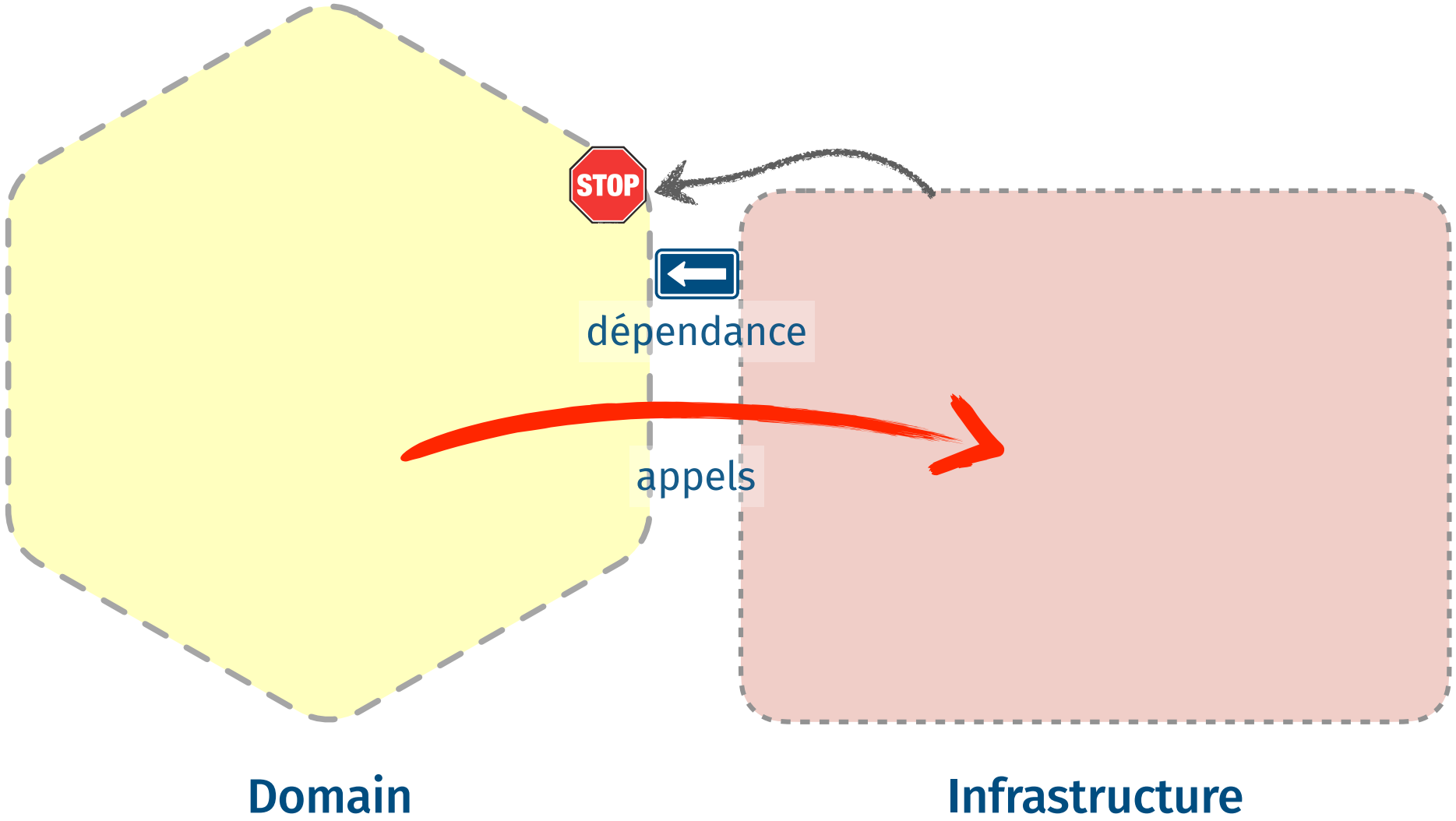
```
class OrderService {
    IOrderRepository repo;
    ... {
            repo.getById(id);
    }
}
```

```
interface IOrderRepository {
    Order getById(id);
}
```

```
class OrderRepository {
    implements IOrderRepository {
    public Order getById(id) {
    ...
    }
}
```

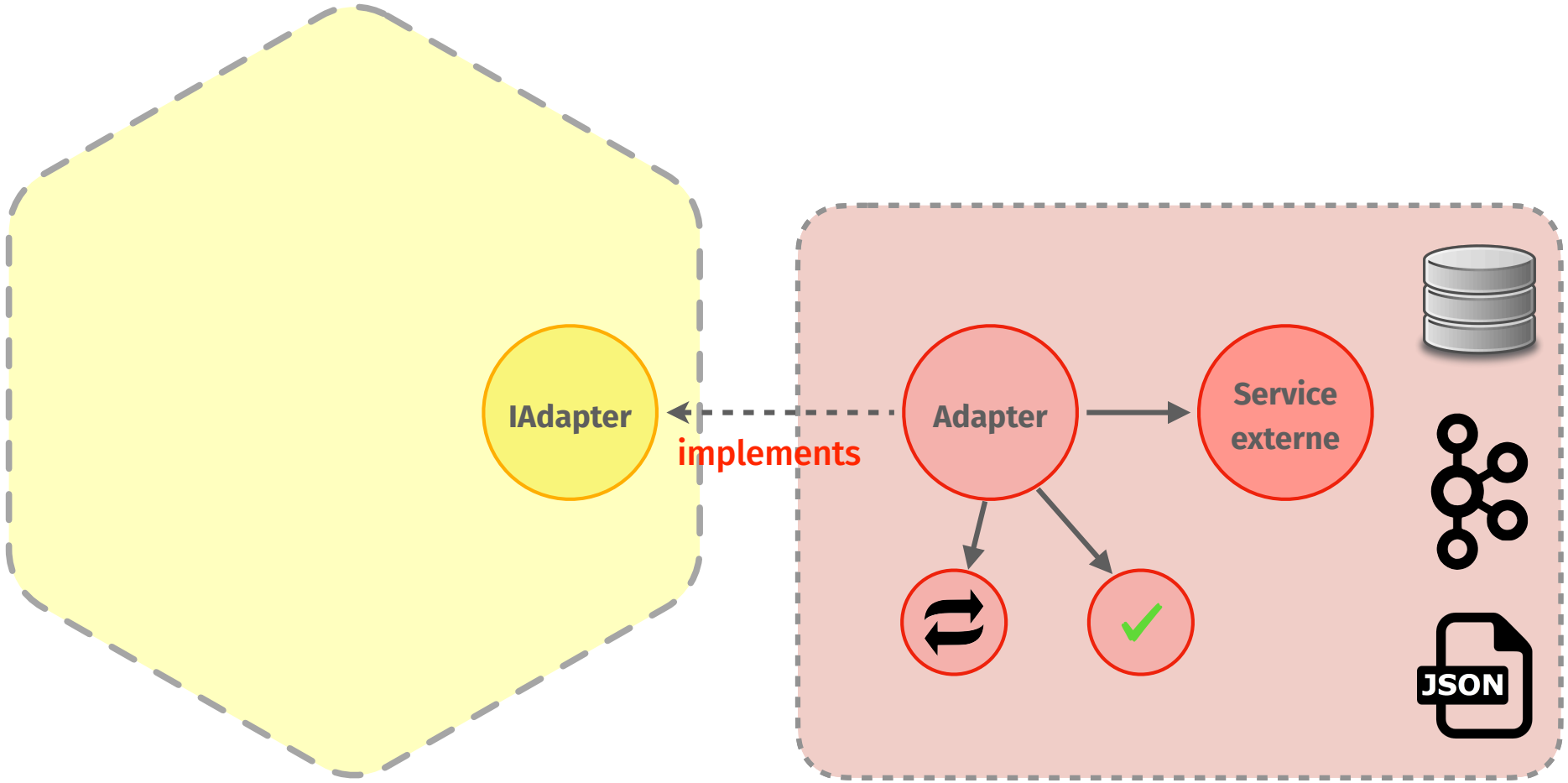


Dependency Inversion Principle



Les abstractions ne dépendent pas des aspects techniques

Choix techniques retardés



Domain

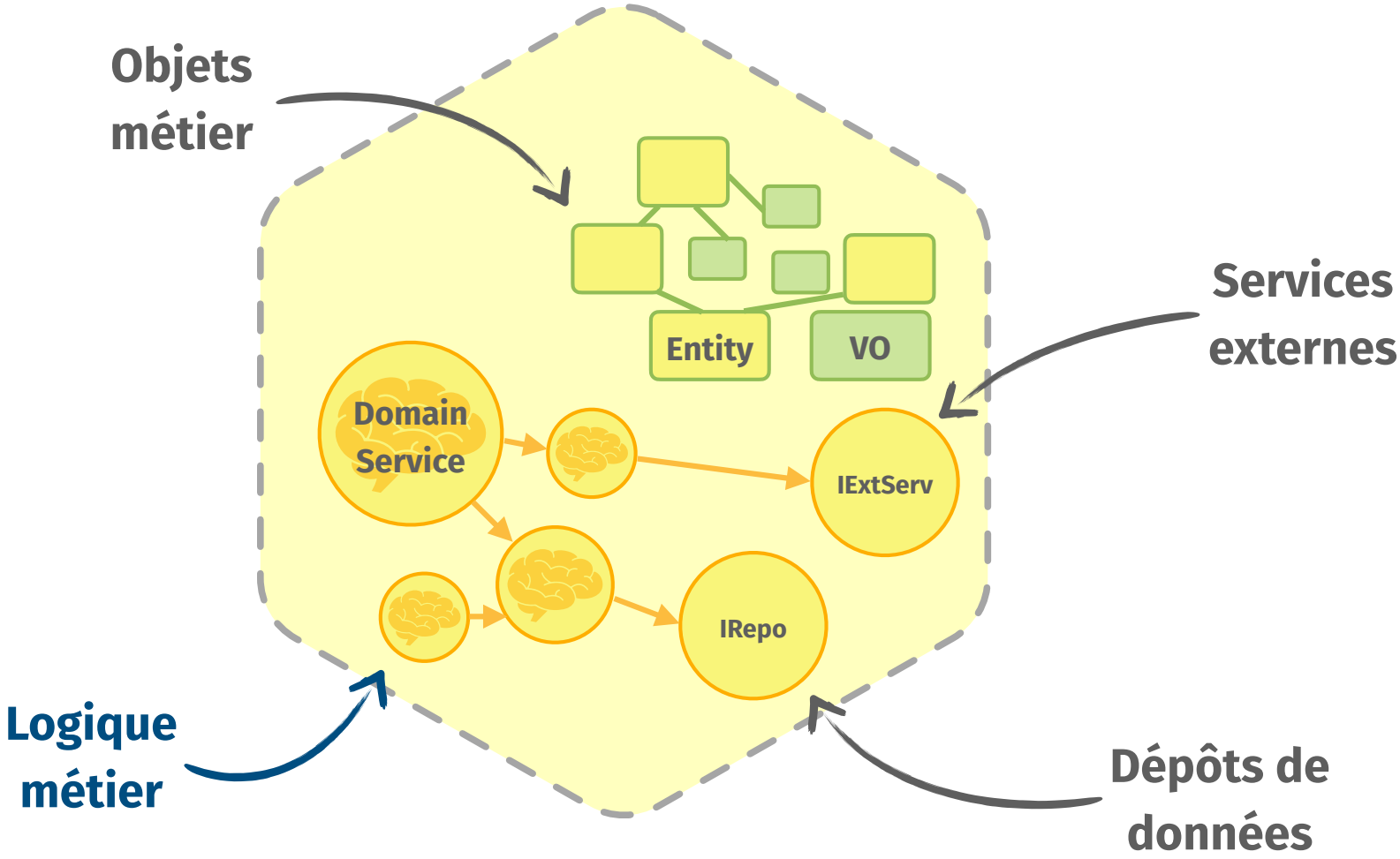
Infrastructure

Un domaine agnostique

vous permet de vous concentrer

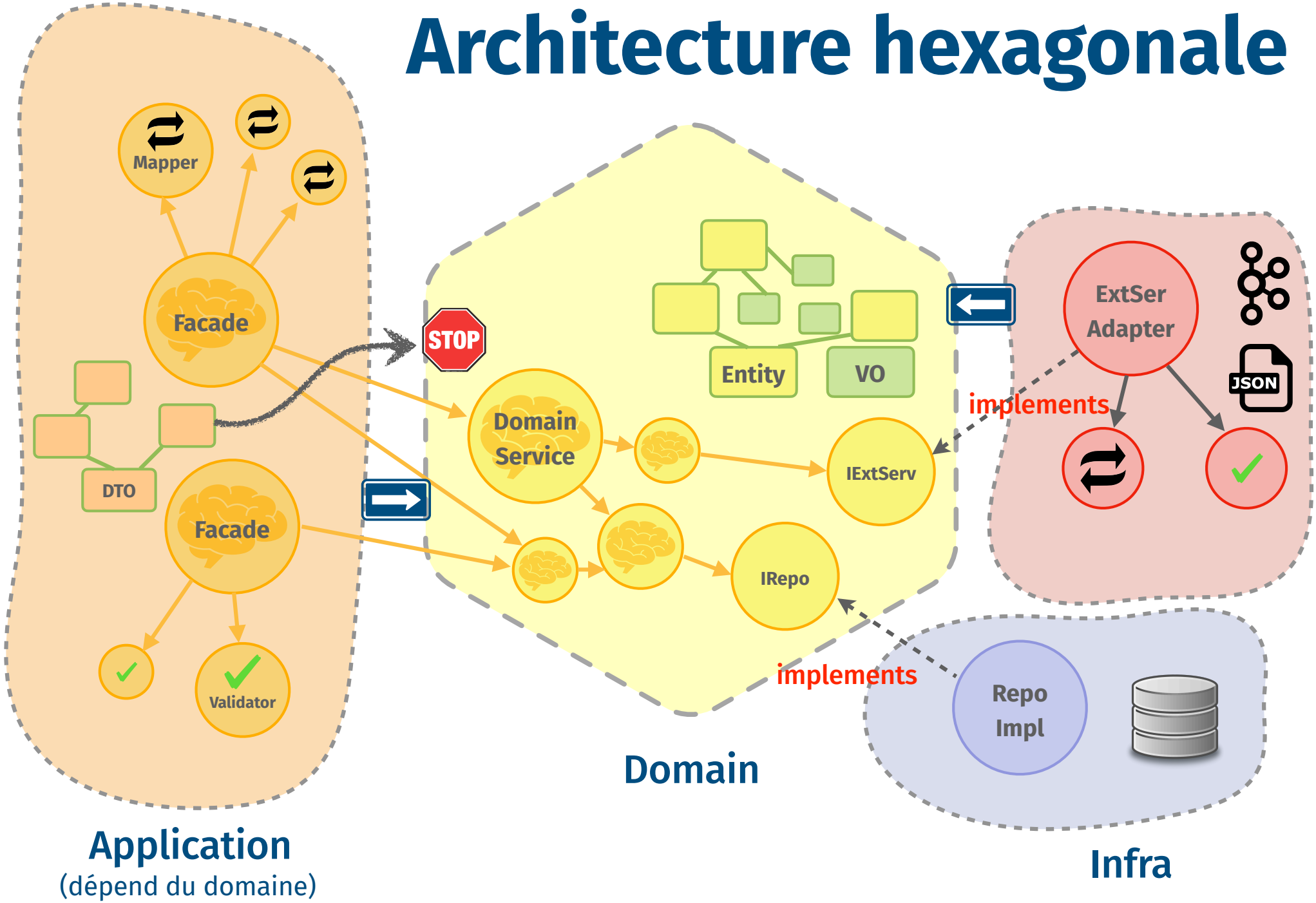
sur VOTRE logique métier

Quelle partie de votre code est importante ?



Le code **métier** !

Architecture hexagonale



On n'évolue que

sous la contrainte !

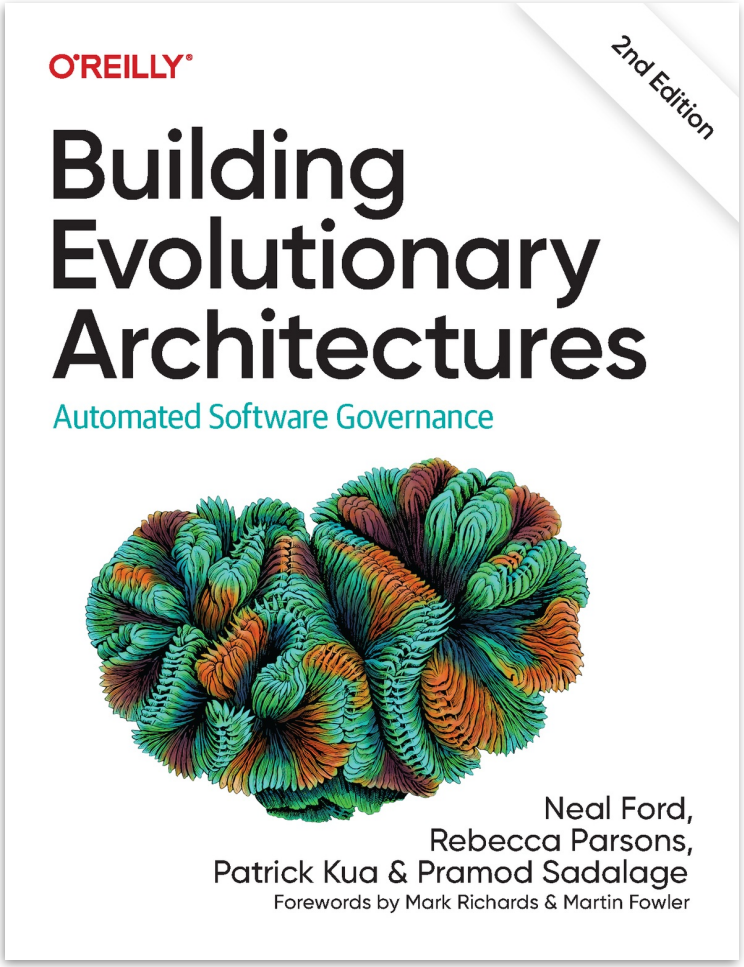
quels critères ?

Couplage

Performance

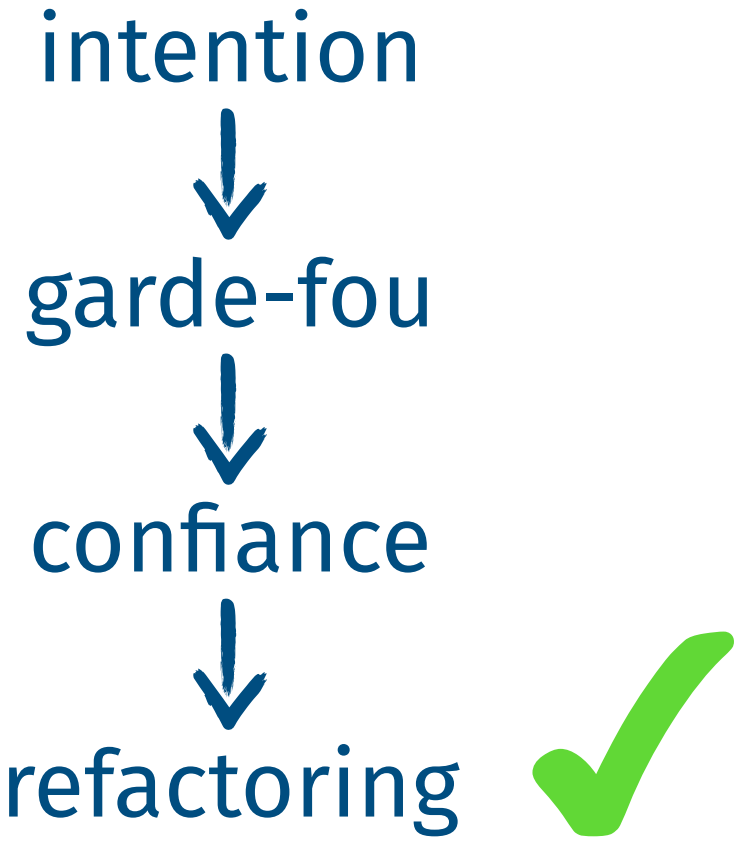
Passage à l'échelle

Sécurité



Tests

Il faut avoir de nombreux tests



Code testable == code plutôt bon

simple

découplé

documenté

Les tests les plus faciles à maintenir

fonctions pures (domaine)

mocks (pas toujours simples)

émulations (H2)

systèmes de production (chaos monkey)

Synthèse

Synthèse

Principes de base	**KISS**	pas de suringénierie
Modélisation des données	**DTOs**	hors de votre domaine métier
Logique métier	**SRP, DRY**	Facades, puis extraction
Architecture hexagonale	**DIP**	le domaine est agnostique !
Tests		Ils vous aident pour améliorer votre code et votre architecture