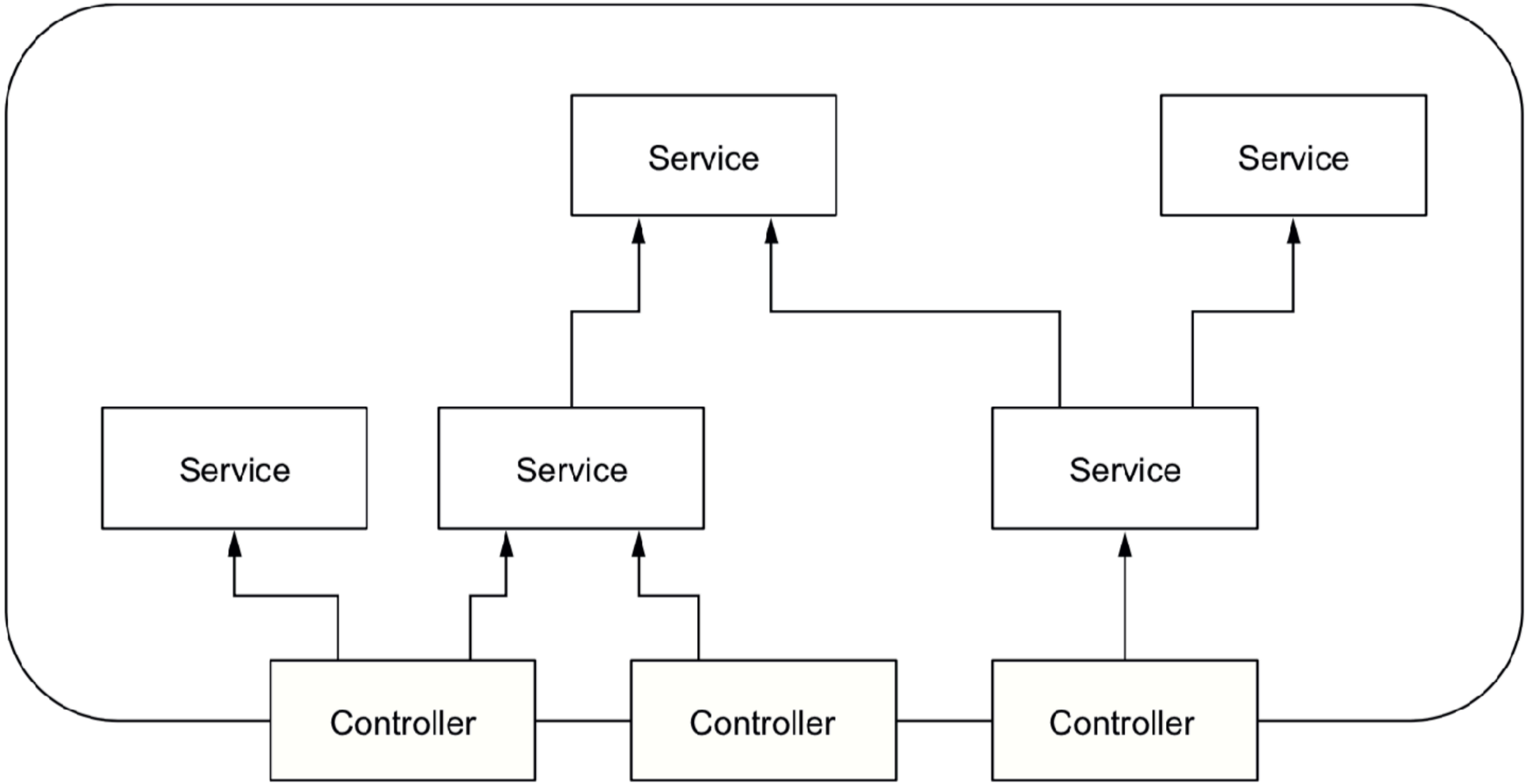
OBJECT DESIGN

ARCHITECTURE LOGICIELLE

OBJECT GRAPH



DAO & REPOSITORY

	DAO	Repository
Fonctionnalités	CRUD Orienté Base de données	Collection augmentée (recherches + fonctionnalités spécifiques) Uniquement métier
Nommage	Orienté technique	Suivant l'Ubiquitous Language. Typiquement, la forme plurielle d'une Entité
Granularité	Souvent mono Table	*Agrégats*
Idempotence	A la sémantique d'une BD : add n'est pas idempotent	A la sémantique d'un **Set** : add/remove sont idempotents
Comportement transactionnel	Participe à (mais n'initie pas) la transaction courante. La démarcation transactionnelle est faite par l'Application Service	

INFRASTRUCTURE SERVICE

- Généralisation d'un Repository
  - Mais avec une sémantique moins précise
- Il contient des éléments techniques, n'ayant pas pour objectif de représenter un concept métier
  - Intégration
  - Technique de mapping
  - Communication extérieure
- Il respecte le DIP
  - L'interface est dans la couche Domain : le besoin est exprimé par le métier
  - L'implémentation est dans la couche Infrastructure

INFRASTRUCTURE SERVICE

	Repository	Infrastructure Service
Représente typiquement	Un data store, dans le même SI	Un service technique (intégration, mapping, processing, etc)
Sémantique	Collection augmentée (ajout, suppression)	Pas forcément de notion de collection d'éléments
Participe aux transactions ?	Oui	Pas forcément
Implémentation	Respect de l'inversion de dépendance (DIP)	

Identification des problématiques

DOMAIN INVARIANTS


```
final class User
{
    private string emailAddress;

    public function __construct(string emailAddress)
    {
        if (!is_valid_email_address(emailAddress)) {
            throw new InvalidArgumentException(
                'Invalid email address'
            );
        }
        this.emailAddress = emailAddress;
    }

    // ...

    public function changeEmailAddress(string emailAddress): void
    {
        if (!is_valid_email_address(emailAddress)) {
            throw new InvalidArgumentException(
                'Invalid email address'
            );
        }
        this.emailAddress = emailAddress;
    }
}
```

Validates that the provided email address is valid

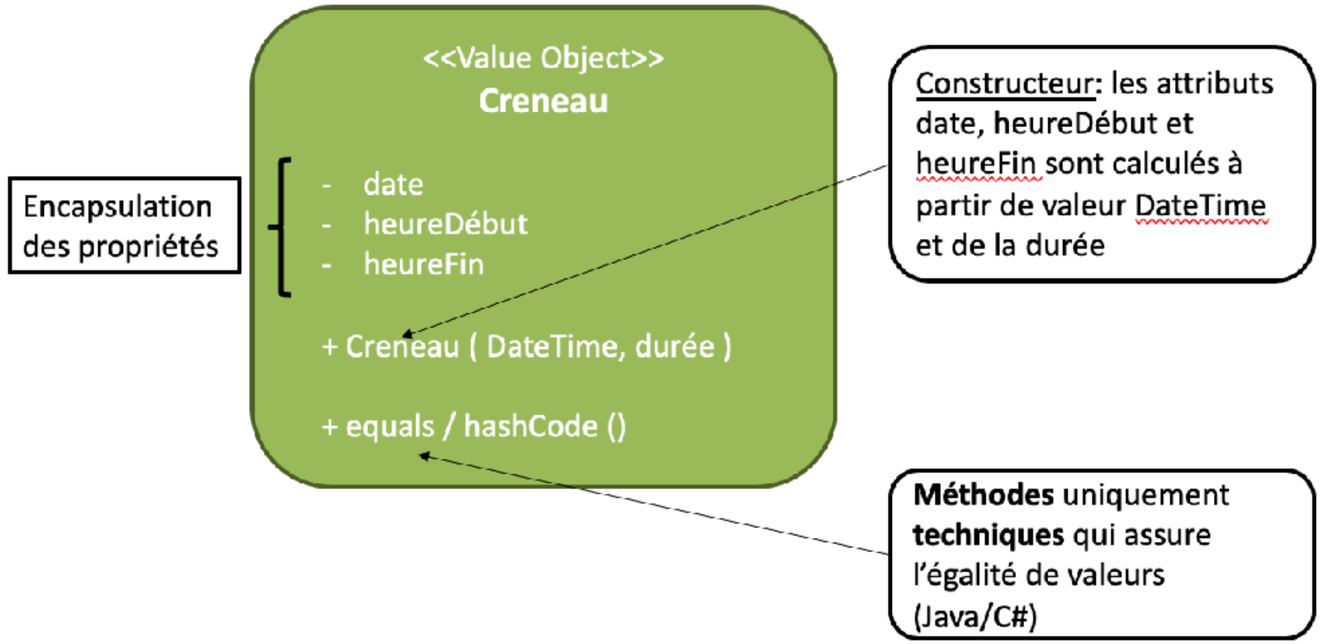
Validates it again, if it's going to be updated

VALUE OBJECTS

```
final class EmailAddress
{
    private string emailAddress;

    public function __construct(string emailAddress)
    {
        if (!is_valid_email_address(emailAddress)) {
            throw new InvalidArgumentException(
                'Invalid email address'
            );
        }
        this.emailAddress = emailAddress;
    }
}

final class User
{
    private EmailAddress emailAddress;

    public function __construct(EmailAddress emailAddress)
    {
        this.emailAddress = emailAddress;
    }

    // ...

    public function changeEmailAddress(EmailAddress emailAddress): void
    {
        this.emailAddress = emailAddress;
    }
}
```

VALUE OBJECTS

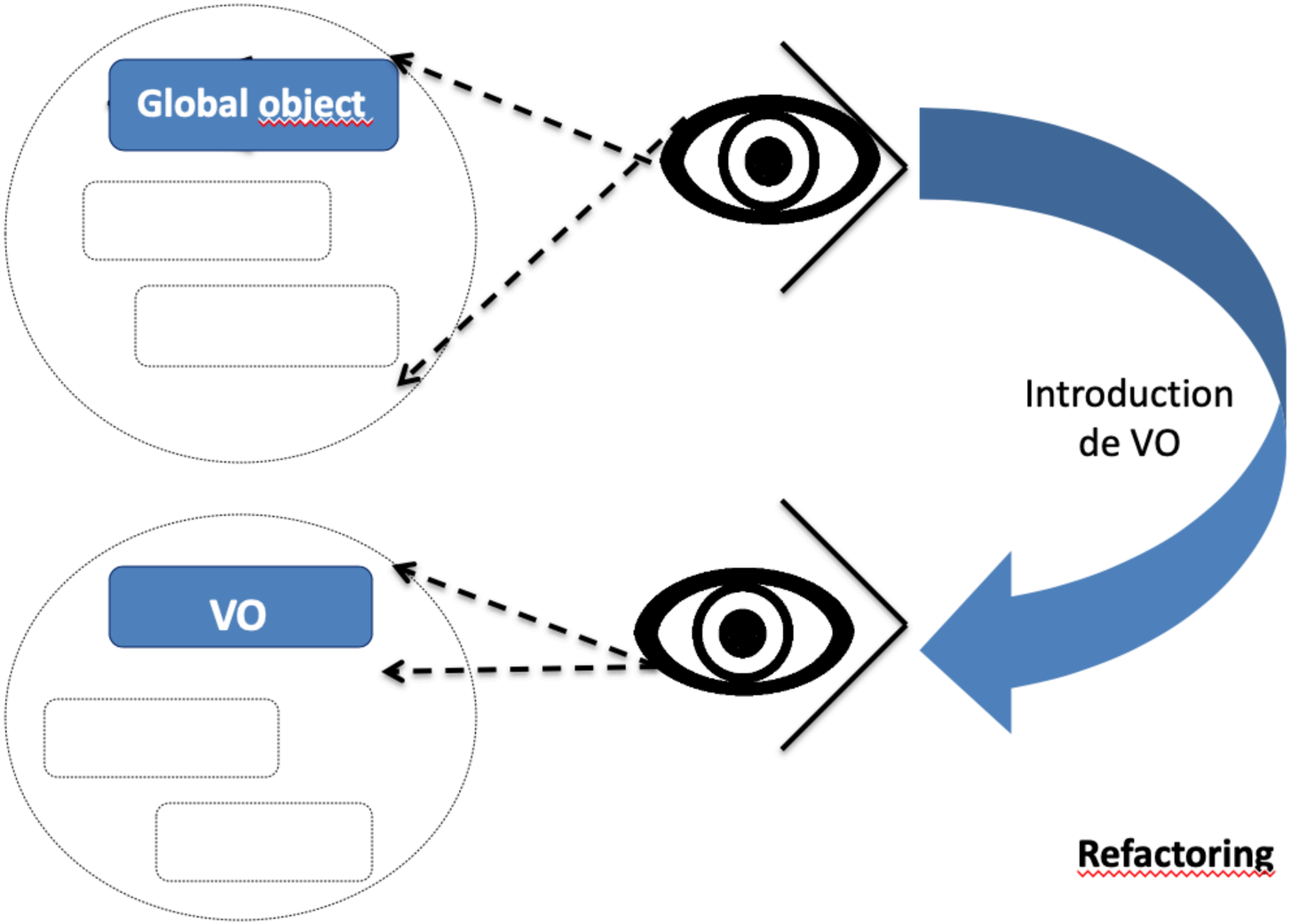
- Concept du domaine
- Des objets avec des attributs et du comportement
  - sans identité
  - sans cycle de vie
- Immuable
- Intangible : Existence indépendante du temps et de l'espace
- La comparaison des Value Objects se fait en comparant les attributs



CONCEPTUAL WHOLE

- Les Value Objects ne signifie pas qu'ils doivent contenir une longue liste d'attributs
- Les attributs d'un Value Object doivent former un tout insécable
  - ex: 5000 $

VALUE OBJECTS, UNE AIDE AU REFACTORING

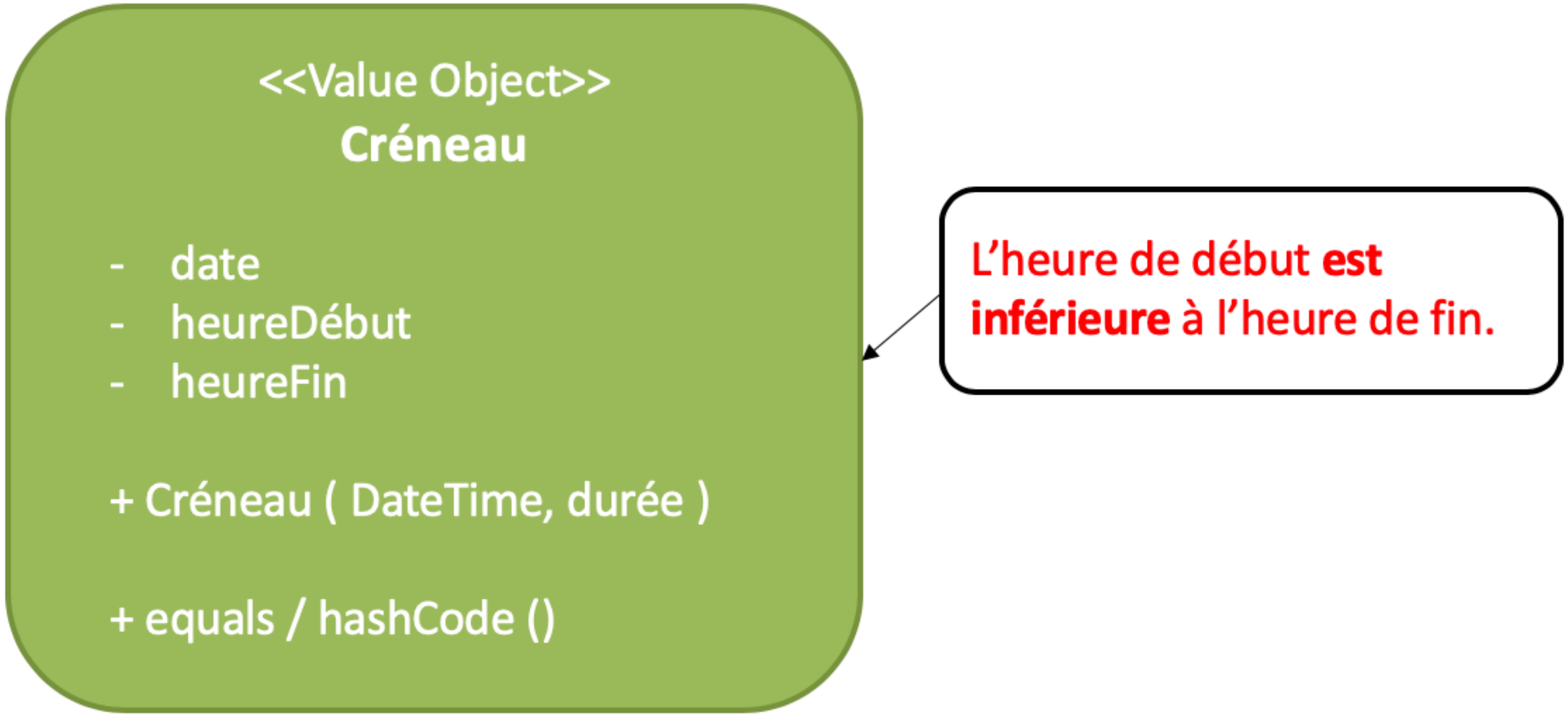


VALUE OBJECTS

Donner des exemples de Value Object

VALUE OBJECTS

NOTION DE INVARIANTS



VALUE OBJECT

Implémenter l'objet Créneau

ENTITIES

```
final class SalesInvoice
{
    /**
     * @var Line[]
     */
    private array lines = [];

    private bool finalized = false;

    public static function create(/* ... */): SalesInvoice
    {
        // ...
    }

    public function addLine(/* ... */): void
    {
        if (this.finalized) {
            throw new RuntimeException(/* ... */);
        }

        this.lines[] = Line.create(/* ... */);
    }

    public function finalize(): void
    {
        this.finalized = true;
        // ...
    }

    public function totalNetAmount(): Money
    {
        // ...
    }

    public function totalAmountIncludingTaxes(): Money
    {
        // ...
    }
}
```

You can create a sales invoice.

You can manipulate its state, such as by adding lines to it.

You can finalize it.

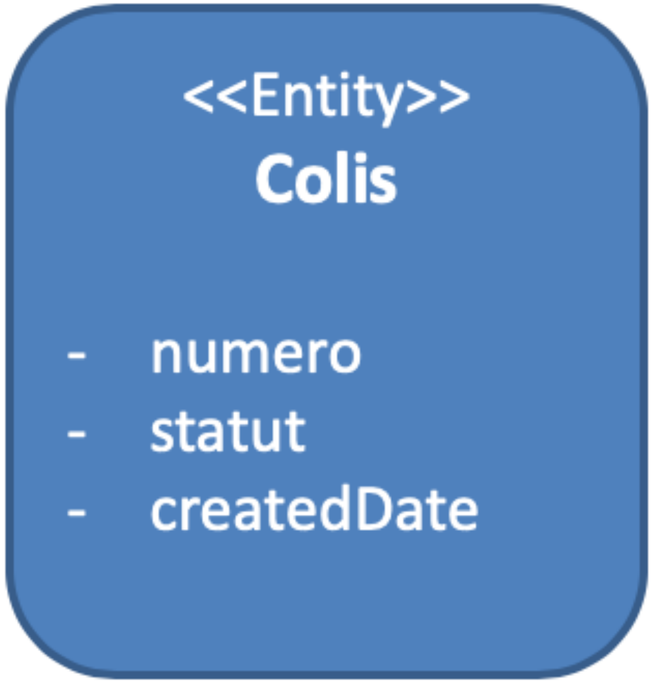
It exposes some useful information about itself.

ENTITIES

Donner des exemples de Entities

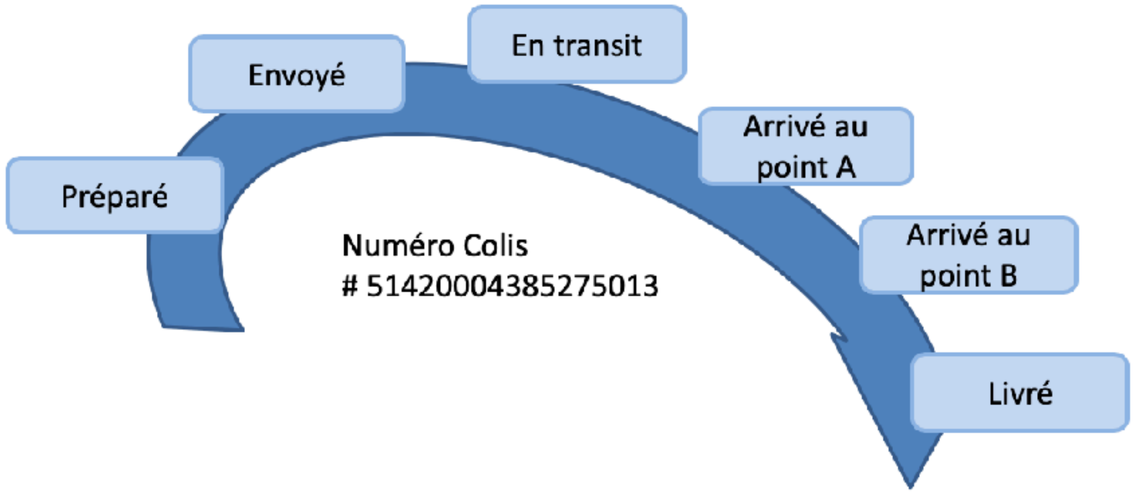
ENTITY

- Certains objets sont intrinsèquement définis par la continuité de leur identité et non par leurs attributs (qui eux peuvent varier dans le temps).
- Identifiable
  - propriété d'unicité
  - point d'ancrage qui permet le suivi des objets dans le temps à travers un cycle de vie
- Peut contenir des Value Objects



IDENTITÉ DE L'ENTITY

- Chaque Entity doit établir de manière précise son identité afin de se distinguer
- D'autres entities peuvent avoir les mêmes valeurs des attributs
- L'unicité de l'attribut d'identité doit être garantie dans tout le système
- L'identité doit être immuable

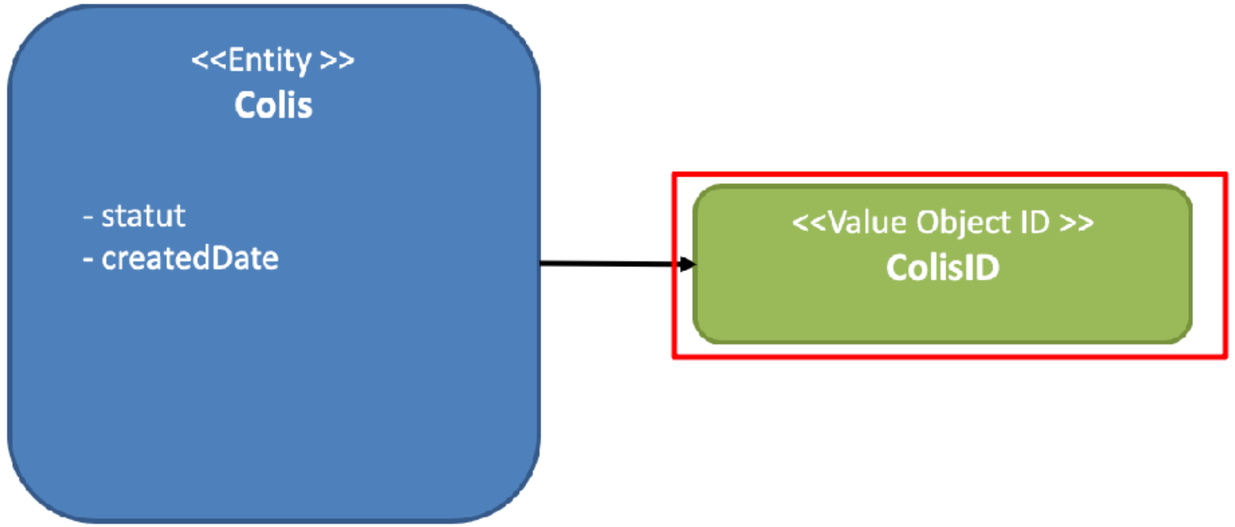


VALUE OBJECTS

Proposer une implémentation
de l'Entity "Colis".

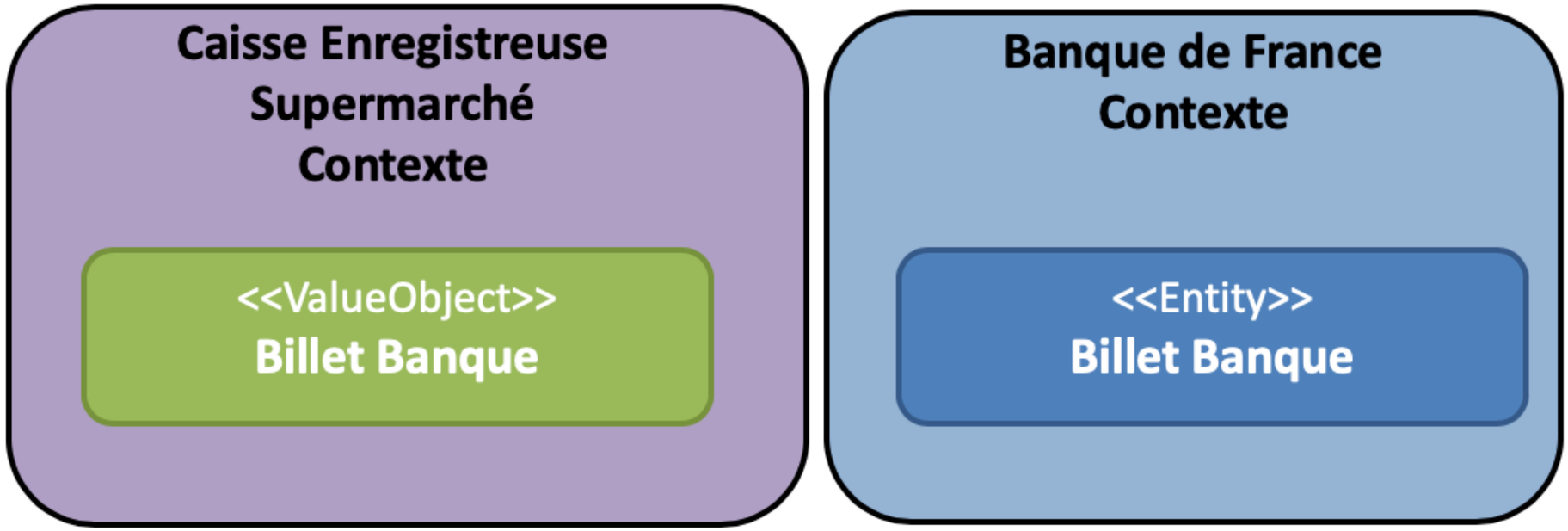
VO ID

- Critère unique d'égalité des entités
- Unique dans un Bounded Context
- Immuable
- Fortement typé
- Peut être sérialisé sous forme de champs int/ String/… (persistance, réseau)
- Typiquement moins de comportement que un VO générique mais peut inclure de la validation métier sur la valeur d'identité (ex: ISBN)

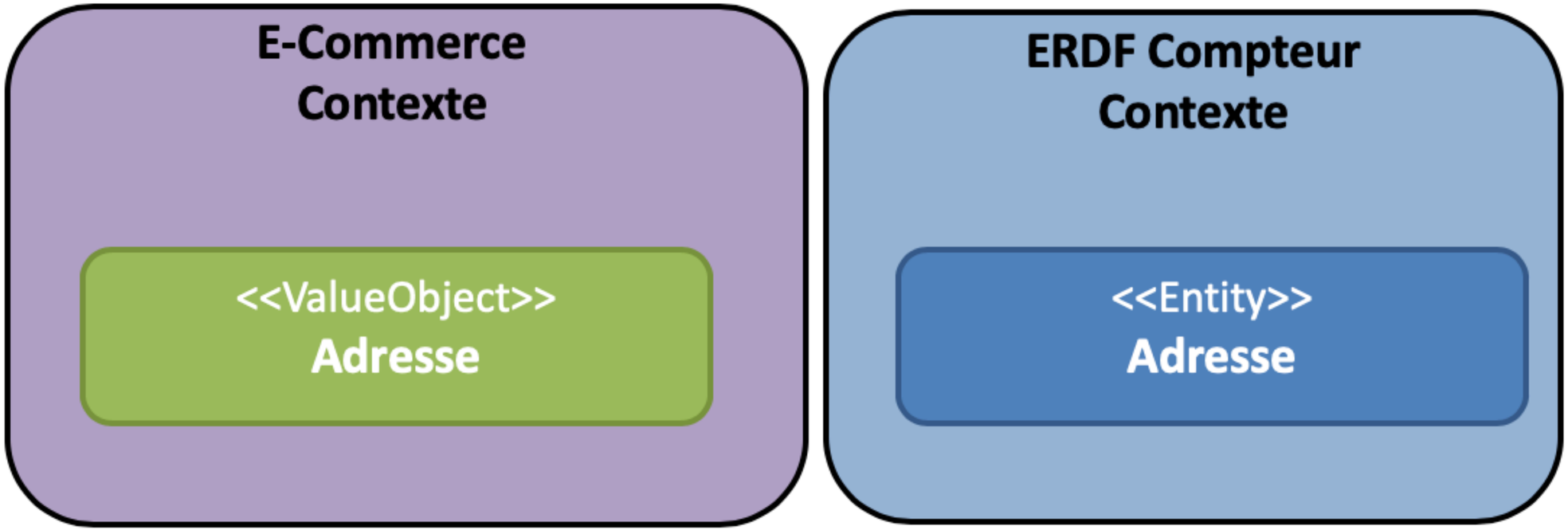


	Value Object	Entity
Caractéristique principale	Ses attributs	Son identité
Comportement lors d'un changement d'état	Production d'un nouveau VO	Mutation de la même Entité
Critère d'égalité	Suivant sélection métier d'un ensemble d'attributs	Suivant référence (en mémoire) ou VO ID
Comportement temporel	Intemporel	Variation de l'état dans le temps
Autonomie (vis à vis d'un cas d'utilisation)	Ne peut pas être utilisé seul, caractérise une Entité.	Peut être utilisée seule
Composition	Ne peut agréger que d'autres VO	Peut agréger d'autres Entités et VO

TOUT DÉPEND DU CONTEXTE (1/2)



TOUT DÉPEND DU CONTEXTE (2/2)



DATA TRANSFER OBJECT (DTO)

```
final class ScheduleMeetup
{
    public string title;
    public string date;
}
```

DATA TRANSFER OBJECT (DTO)

```
final class CreateUser
{
    public string username;
    public string password;
}
```

DATA TRANSFER OBJECT (DTO)

- A DTO can be created using a regular constructor.
- Its properties can be set one by one.
- All of its properties are exposed.
- Its properties contain only primitive-type values.
- Properties can optionally contain other DTOs, or simple arrays of DTOs.

DTO

Reprendre le service
de création d'un utilisateur
en introduisant
un objet DTO createUser

	Data Transfer Object (DTO)	Value Object (VO)
Intention	Transfert de données	Représentation d'un concept du Domain Model
Contenu	Ensemble des données à faire transiter (réseau, passage entre couches, ..). Peut agréger des données hétérogènes pour un cas d'utilisation	Des données fortement cohérentes
Comportement	Pas de comportement	Comportement riche
Immuable	Généralement non	Oui

VALUE OBECTS - IMMUTABILITY

```
final class Integer
{
    private int integer;

    public function __construct(int integer)
    {
        this.integer = integer;
    }

    public function plus(Integer other): Integer
    {
        return new Integer(this.integer + other.integer);
    }
}
```

VALUE OBECTS - IMMUTABILITY

```
final class Position
{
    // ...

    public function toTheLeft(int steps): Position
    {
        copy = clone this;

        copy.x = copy.x - steps;

        return copy;
    }
}

position = new Position(10, 20);

nextPosition = position.toTheLeft(4);
assertEquals(new Position(6, 20), nextPosition);

assertEquals(new Position(10, 20), position);
```

The next position will be (6, 20).

The original object should not have been modified.

MUTABILITY & IMMUTABILITY

```
final class Player
{
    private Position position;

    public function __construct(Position initialPosition)
    {
        this.position = initialPosition;
    }

    public function moveLeft(int steps): void
    {
        this.position = this.position.toTheLeft(steps);
    }

    public function currentPosition(): Position
    {
        return this.position;
    }
}
```

COMMAND METHOD

```
final class Player
{
    private Position position;

    public function __construct(Position initialPosition)
    {
        this.position = initialPosition;
    }

    public function moveLeft(int steps): void
    {
        this.position = this.position.toTheLeft(steps);
    }

    public function currentPosition(): Position
    {
        return this.position;
    }
}
```

```
final class Position
{
    // ...

    public function toTheLeft(int steps): Position
    {
        // ...
    }
}
```

VALUE OBECTS - IMMUTABILITY

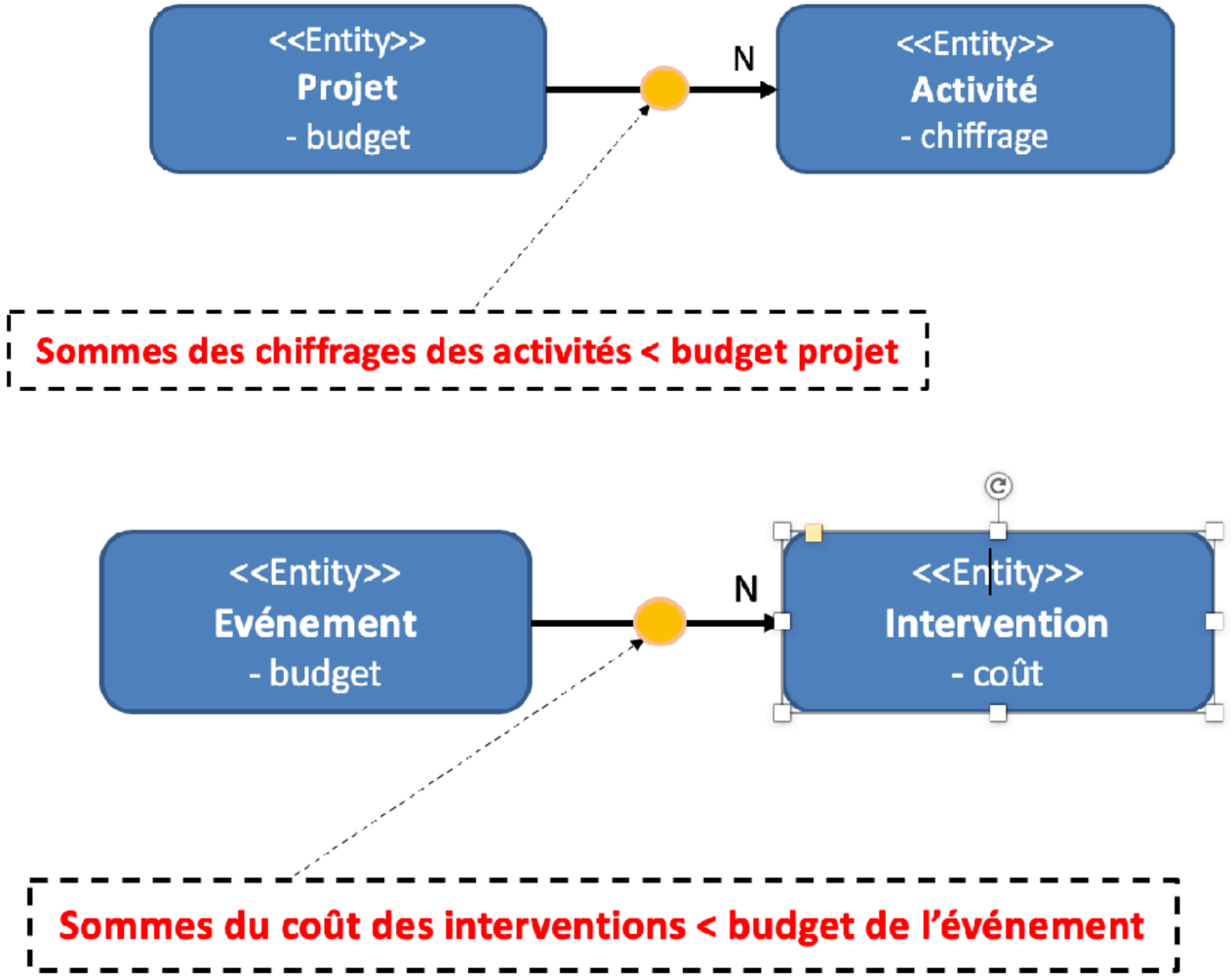
Matérialiser un objet représentant
l'historique du cycle de vie
des états d'un objet,
le tout en respectant l'immutabilité

MUTABILITY & IMMUTABILITY

Reprendre la gestion du Colis
et introduire le VO Etat
et son objet History

LES INVARIANTS

- Un invariant est une règle métier à laquelle l'état du système doit se conformer
- Permet de garantir la cohérence des changements apportés aux éléments du modèle



LES INVARIANTS

Pour le trajet d'un voyage, la date de retour est toujours postérieure à la date de départ.

Pour une carte bancaire, la somme de toutes les opérations effectuées avec cette carte ne doit pas excéder le plafond autorisé

Pour une thermostat, la température minimum est inférieure à la température maximum.

LES INVARIANTS

- Un système peut contenir plusieurs invariants
- Une implémentation cohérente préserve les invariants en rejetant les changements qui ne les respectent pas
- Un invariant défini par une règle précise est un invariant strict
- La modification des objets (qui font intervenir l'invariant) est transactionnelle (atomicité, durabilité)