4 – Domain Driven Design

1MODE - Modélisation d'applications

Sommaire

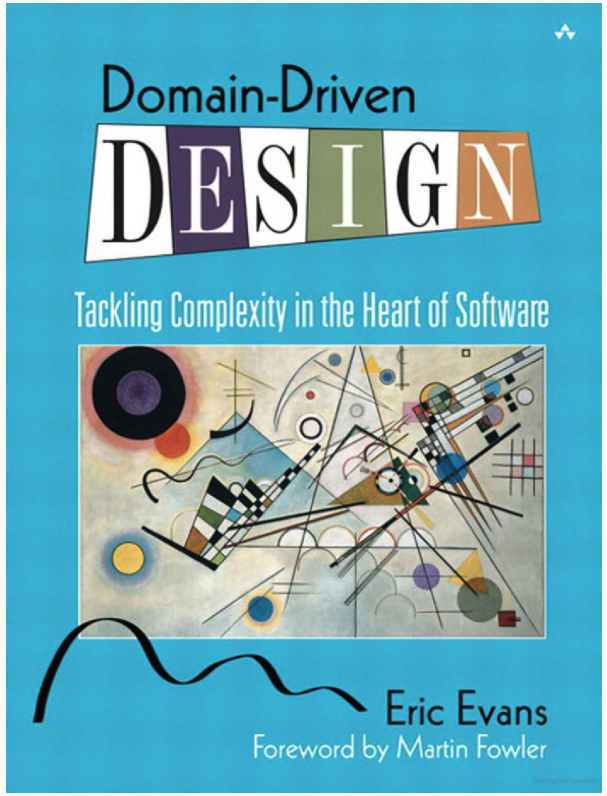
1. Introduction
2. Design Stratégique
3. Design Tactique
4. Architecture en couches

1. Introduction

1. Introduction

Domain Driven Design

- "Conception dirigée par le domaine/le métier"
- Approche qui consiste à résoudre la complexité du développement logiciel en mettant en avant un domaine
- Introduite en 2003 dans le livre du même nom par Eric Evans
- Ensemble d'outils dont une philosophie, une terminologie, et des modèles de conceptions



1. Introduction

Définition d’un domaine

- Le terme "domaine" fait référence au sujet spécifique pour lequel un projet est développé
- Exemple : la comptabilité
- L'objectif du DDD est de limiter la complexité d'une solution en l'adaptant au plus près du domaine avec l'aide d'experts dans ce domaine

1. Introduction

Principes du DDD

- Se concentrer sur le domaine principal et la logique qui en découle
- Utiliser de modèles du domaine pour concevoir des fonctionnalités complexes
- Instaurer une collaboration étroite entre les experts techniques et les experts du domaine pour créer un modèle d'application adapté aux problématiques spécifiques du domaine

1. Introduction

Design stratégique et tactique

- Le modèle stratégique aide à concevoir les domaines, sous-domaines qui sont communiqués par le langage ubiquitaire, puis aide à organiser les équipes de développement

- Le modèle tactique lui guide sur la façon de mettre en œuvre une application de manière évolutive

1. Introduction

2. Design Stratégique

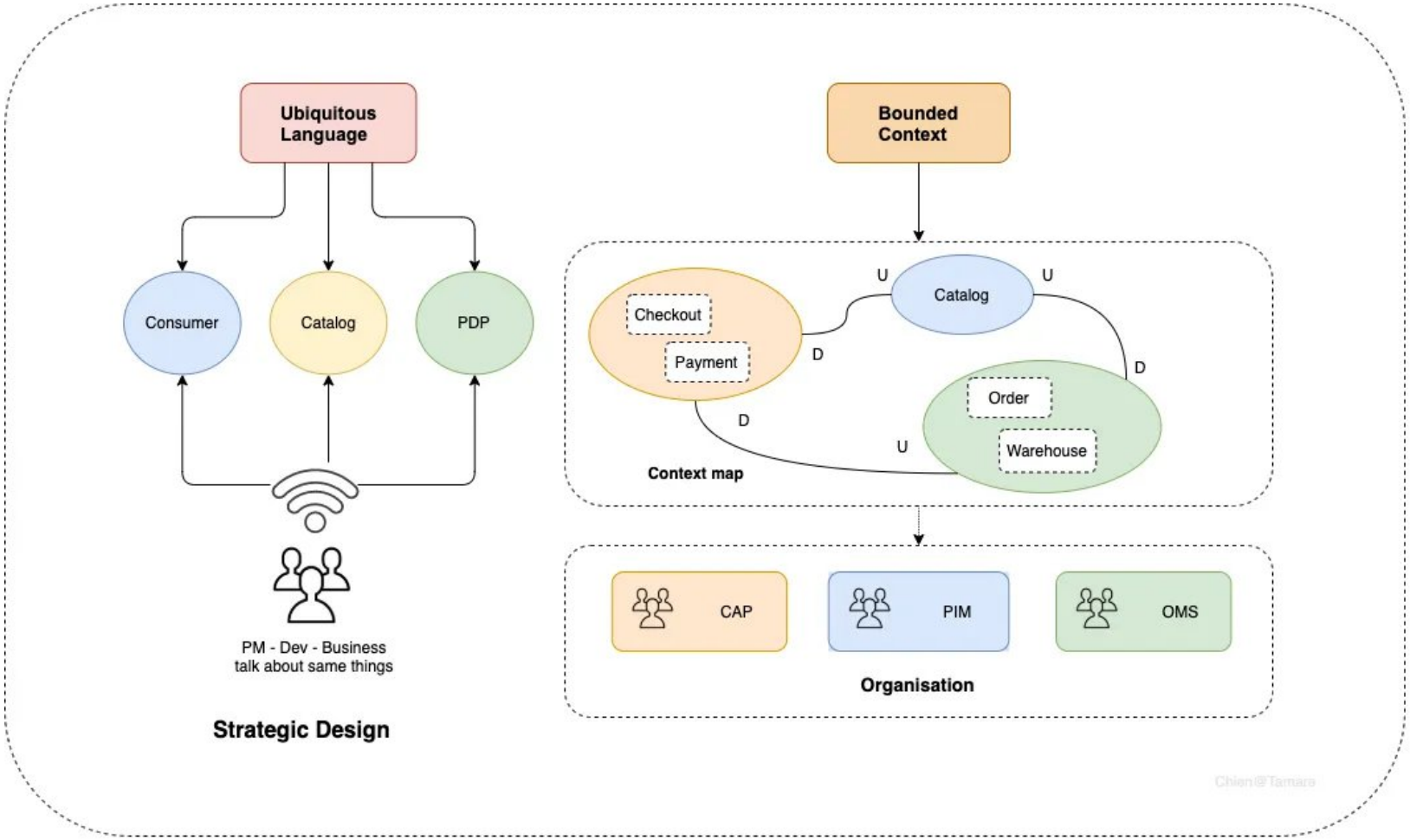
2. Design Stratégique

Design Stratégique

- Les outils de conception stratégique nous aident à résoudre les problèmes liés à la modélisation de logiciels
- Il s'agit d'une approche de conception similaire à la conception orientée objet, où nous sommes contraints de réfléchir en termes d'objets
- Avec la conception stratégique, nous sommes contraints de réfléchir en termes de contexte

2. Design Stratégique

Aperçu



2. Design Stratégique

Langage Ubiquitaire

- Consiste à définir un vocabulaire commun compris par tous les membres de l'équipe de développement, y compris les experts métier
- En TypeScript, nous pouvons utiliser des noms de variables et de fonctions qui correspondent à ce langage ubiquitaire

2. Design Stratégique

Contexte borné (*Bounded context*)

- Un contexte borné est une frontière logique d'un domaine où des termes et des règles particuliers s'appliquent de manière cohérente
- À l'intérieur de cette frontière, tous les termes, définitions et concepts forment le langage ubiquitaire
- En TypeScript, nous pouvons mettre en œuvre le concept de contexte borné en créant des modules ou des packages séparés pour chaque sous-domaine, et en définissant un langage ubiquitaire pour chacun d'entre eux

2. Design Stratégique

Exemple

Supposons que nous construisons un système qui permet aux utilisateurs de créer et de gérer des listes de courses. Nous pouvons identifier deux sous-domaines principaux :

- Listes de courses
- Gestion des utilisateurs

2. Design Stratégique

Exemple

```
// Module "Shopping"
class List {
  private name: string
  private items: Item[]

  constructor(name: string) {
    this.name = name
    this.items = []
  }

  addItem(item: Item) {
    this.items.push(item)
  }
}
```

2. Design Stratégique

Exemple

```
class Item {
  private name: string
  private quantity: number

  constructor(name: string, quantity: number) {
    this.name = name
    this.quantity = quantity
  }
}
```

2. Design Stratégique

Exemple

```
# Module "User Management"
class User {
  private username: string
  private password: string
  private role?: Role

  constructor(username: string, password: string) {
    this.username = username
    this.password = password
    this.role = null
  }

  setRole(role: Role) {
    this.role = role
  }
}
```

2. Design Stratégique

Exemple

```
class Role {
  private name: string
  private permissions: string[]

  constructor(name: string, permissions: string[]) {
    this.name = name
    this.permissions = permissions
  }
}
```

2. Design Stratégique

Exemple

- Nous avons créé deux packages distincts pour chaque sous-domaine : "Shopping" et “User Management"
- Chaque package contient des classes qui correspondent aux termes et concepts définis dans le Langage Ubiquitaire pour chaque sous-domaine
- En utilisant le contexte borné et le langage ubiquitaire, nous créons une séparation claire des préoccupations
- Cette approche rend notre code plus maintenable et plus facile à comprendre

2. Design Stratégique

3. Design Tactique

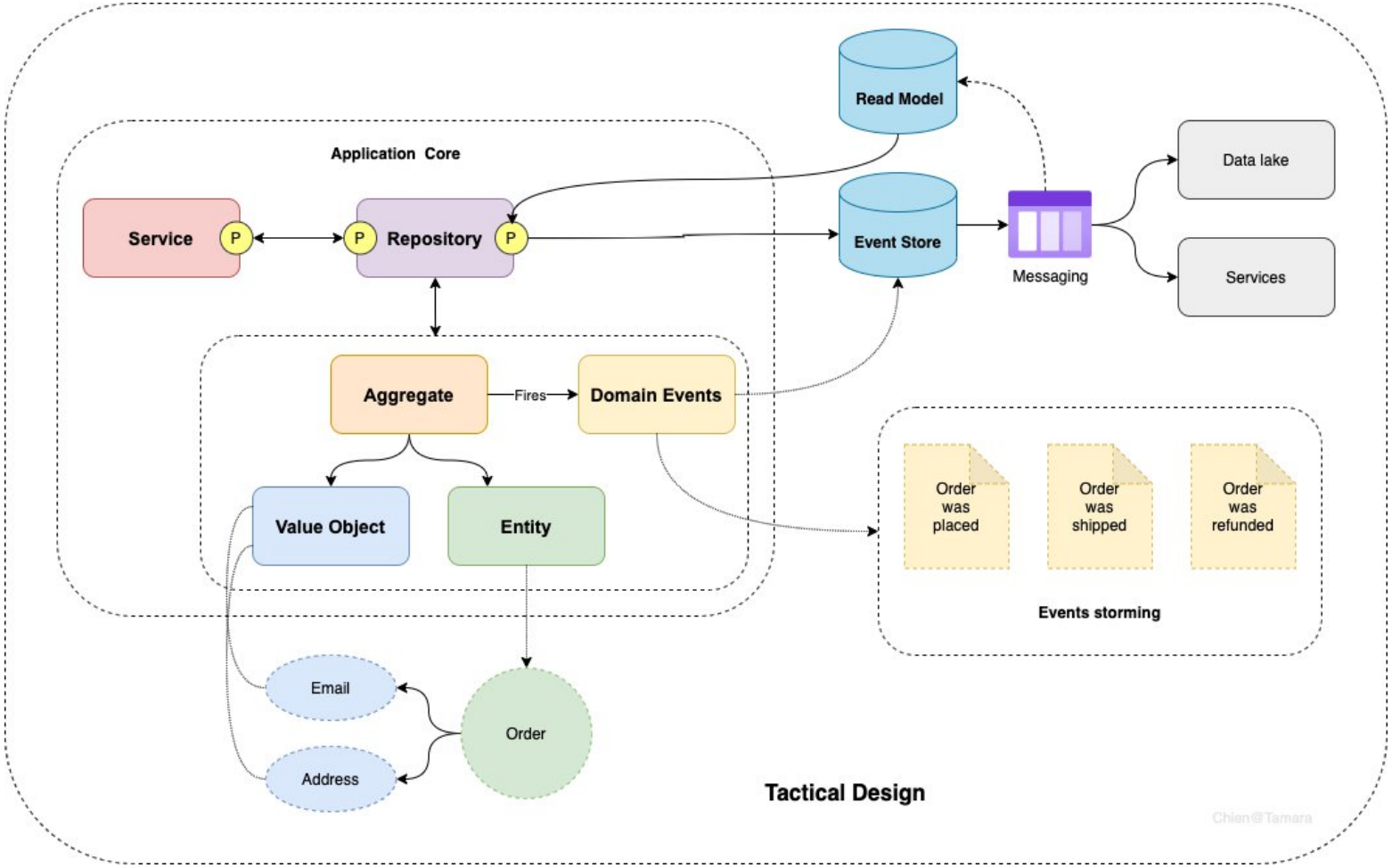
3. Design Tactique

Design Tactique

- Le design tactique traite des détails de mise en œuvre, c'est-à-dire de la modélisation de domaine
- Il prend généralement en charge les éléments à l'intérieur d'un contexte borné
- Le design tactique prend place pendant la phase de développement du produit

3. Design Tactique

Aperçu



3. Design Tactique

Entité

- Une entité est une classe avec des attributs liés au domaine
- Une instance de ces classes ont une identité propre (*i.e.*, ont un ID unique)
- Cet ID est immuable même si l'état de l'objet change
- Cet ID est généralement un UUID, ou un nombre entier incrémenté de 1 automatiquement

3. Design Tactique

Entité – Exemple

```
import { v4 as uuid } from 'uuid'

class Customer {
  private customerId: string
  private name: string
  private email: string

  constructor(customerId: uuid, name: string, email: string) {
    this.customerId = customerId
    this.name = name
    this.email = email
  }
  setName(name: string) {
    this.name = name
  }
  setEmail(email: string) {
    this.email = email
  }
}
```

3. Design Tactique

Objets valeur

- Les objets valeur sont des objets immuables et légers qui n'ont pas d'identité
- Les objets valeur réduisent la complexité en effectuant des calculs complexes, en isolant la logique lourde des entités

Exemple

- User est une entité et Address est un objet valeur
- L'adresse peut changer plusieurs fois, mais l'identité de User ne change jamais
- Chaque fois qu'une adresse est modifiée, une nouvelle adresse est instanciée et assignée à User

3. Design Tactique

Objets valeur – Exemple

```
class Address {
  private street: string
  private city: string
  private zipCode: string

  constructor(street: string, city: string, zipCode: string) {
    this.street = street
    this.city = city
    this.zipCode = zipCode
  }

  setStreet(newStreet: string): Address {
    return new Address(newStreet, this.city, this.zipCode)
  }
  setCity(newCity: string): Address {
    return new Address(this.street, newCity, this.zipCode)
  }
  setZipCode(newZipCode: string): Address {
    return new Address(this.street, this.city, newZipCode)
  }
}
```

3. Design Tactique

Objets valeur – Exemple

```
class User {
  private userId: string
  private name: string
  private email: string
  private address: Address

  constructor(userId: uuid, name: string, email: string, address: Address) {
    this.userId = userId
    this.name = name
    this.email = email
    this.address = address
  }
  // ...
}
```

3. Design Tactique

Service

- Un service est une classe sans état qui convient à un endroit autre qu'une entité ou un objet valeur
- Fonctionnalité qui se situe quelque part entre les entités et les objets valeur, mais qui n'est liée ni à une entité ni à un objet valeur

3. Design Tactique

Service – Exemple

```
class PaymentService {
  private paymentGateway: PaymentGateway

  constructor(paymentGateway: PaymentGateway) {
    this.paymentGateway = paymentGateway
  }

  processPayment(userId: string, amount: number): boolean {
    // Vérification du solde de l'utilisateur, etc.
    return this.paymentGateway.processPayment(userId, amount)
  }
}
```

3. Design Tactique

Aggrégat

- Un agrégat est un groupe d'objets qui sont traités comme une unité cohérente dans le modèle de domaine
- Il est composé d'une racine d'agrégat et d'autres entités ou objets de valeur associés qui sont considérés comme faisant partie de cet agrégat
- La racine d'agrégat est une entité qui a une identité globale unique au sein du contexte borné
- Les autres entités ou objets valeur associés à l'agrégat ne peuvent être accédés que via la racine d'agrégat

3. Design Tactique

Aggrégat

- L'agrégat est responsable de maintenir la cohérence des objets qui le composent, en garantissant que les règles métier sont respectées à chaque modification
- Les modifications apportées à un agrégat doivent être effectuées en une seule opération atomique pour garantir la cohérence de l'ensemble de l'agrégat
- Les agrégats peuvent être utilisés pour gérer la complexité du modèle de domaine en réduisant le nombre d'objets avec lesquels l'application doit interagir
- Les agrégats peuvent être de tailles variables, allant d'une seule entité à des collections complexes d'entités et d'objets de valeur associés

3. Design Tactique

Aggrégat – Exemple

```
class OrderItem {
  private productId: uuid
  private quantity: number

  constructor(productId: uuid, quantity: number) {
    this.productId = productId
    this.quantity = quantity
  }
}
```

3. Design Tactique

Aggrégat – Exemple

```
class Order {
  private orderId: uuid
  private customerId: uuid
  private items: OrderItem[]
  private shippingAddress?: Address

  constructor(orderId: uuid, customerId: uuid) {
    this.orderId = orderId
    this.customerId = customerId
    this.items = []
    this.shippingAddress = null
  }

  addItem(productId: string, quantity: number) {
    this.items.push(new OrderItem(productId, quantity))
  }

  setShippingAddress(street: string, city: string, zipCode: string) {
    this.shippingAddress = new Address(street, city, zipCode)
  }
}
```

3. Design Tactique

Aggrégat – Exemple

```
class OrderAggregate {
  private order: Order

  constructor(order: Order) {
    this.order = order
  }

  addItem(productId: string, quantity: number) {
    this.order.addItem(productId, quantity)
  }

  setShippingAddress(street: string, city: string, zipCode: string) {
    this.order.setShippingAddress(street, city, zipCode)
  }
}
```

3. Design Tactique

Factories et Repositories

- Les factories (usines) et les repositories (dépôts) sont utilisés pour gérer les agrégats
- Les factories aident à gérer le début du cycle de vie des agrégats, tandis que les repositories aident à gérer le milieu et la fin du cycle de vie d'un agrégat
- Les factories aident à créer des agrégats, tandis que les repositories aident à persister les agrégats
- On devrait toujours créer un repository par racine d'agrégat, mais pas pour toutes les entités
- Les repositories et les factories sont des mécanismes importants pour maintenir la cohérence des agrégats et des entités dans un modèle de domaine complexe

3. Design Tactique

Factories et Repositories – Exemple

```
class User {
  private username: string
  private email: string
  private passwordHash: string
  private isActive: boolean

  constructor(username: string, email: string, passwordHash: string) {
    this.username = username
    this.email = email
    this.passwordHash = passwordHash
    this.isActive = false
  }
}
```

3. Design Tactique

Factories et Repositories – Exemple

```
class UserRepository {
  private db: Database

  constructor(db: Database) {
    this.db = db
  }

  save(user: User): void {
    // Persister l'utilisateur en base de données
  }

  getByUsername(username: string): User {
    // Récupérer l'utilisateur en base de données à partir de son nom
  }
}
```

3. Design Tactique

Factories et Repositories – Exemple

```
class UserFactory {
  create(username: string, email: string, password: string): User {
    const passwordHash = this.hashPassword(password)
    const user = new User(username, email, passwordHash)
    user.isActive = true

    return user
  }
}
```

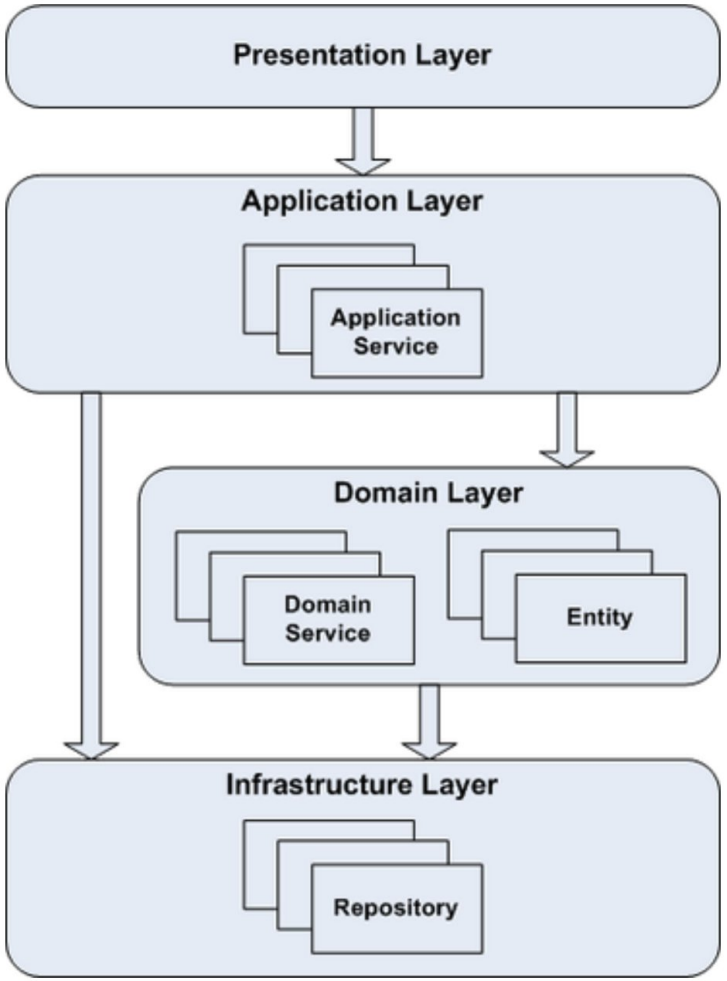
3. Design Tactique

4. Architecture en couches

4. Architecture en couches

Introduction

L'architecture en couches est une approche courante de conception logicielle, qui sépare les différents aspects de l'application en couches logiques distinctes



4. Architecture en couches

Couche Présentation (Interface Utilisateur)

- Responsable de l'affichage des **informations** à l'utilisateur et de la collecte de nouvelles données
- Conçue pour minimiser l'impact sur le reste du système en cas de changement radical de l'interface utilisateur
- Afficher les informations à l'utilisateur, tout en permettant la collecte de nouvelles **données** pour le reste de l'application

4. Architecture en couches

Couche Application

- Cette couche ne contient aucune **règle** métier ni **connaissance** du domaine
- Aucun état métier ne se trouve dans cette couche
- La couche d'application délègue toutes les actions du domaine au domaine lui-même
- Cette couche peut **coordonner** de nombreuses actions, éventuellement dans plusieurs domaines
- Elle peut préparer l'infrastructure à travailler avec le domaine pour une action spécifique, par exemple en préparant des "scopes" de transaction

4. Architecture en couches

Couche Domaine

- Les **règles métier** et la **logique** de l'application résident dans cette couche
- L'état et le comportement des entités du domaine sont définis et utilisés dans cette couche
- L’essentiel des modèles du design tactiques sont utilisés dans cette couche
- La communication avec d'autres systèmes et les détails de persistance sont transférés à la couche d'**infrastructure**

4. Architecture en couches

Couche Infrastructure

- Utilise en général une **base de données** pour stocker les données de l'application
- Séparée de la couche de **domaine** pour maintenir la cohérence et la maintenabilité de l'ensemble du système

4. Architecture en couches