takima

Progresser en conception avec les design patterns

Mathilde Lorrain & Benjamin Yvernault

Qu’est ce que c’est que ÇA !

UtilsService.java

```java
package fr.takima.fakeProject.utils;

import java.io.InputStream;

public class UtilsService {

    public static int getTableSize(int[] array) {
        int length = 0;
        for (int idx = 0; idx < array.length; idx++) {
            length++;
        }
        return length;
    }

    public static Object toObject(String value) {
        return (Object) value;
```

Commande qu'on connaît tous

“

Je n’avais pas les bons réflexes

”

Wilson, tu n’es pas seul au monde !

Comment progresser en conception ?

ALGORITHMIE

2 choix

Choix numéro 1

On n'est pas le plus XP

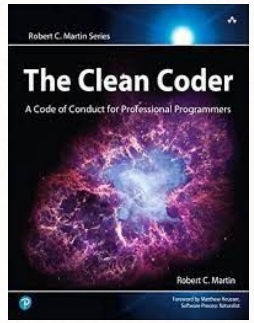
Choix numéro 1

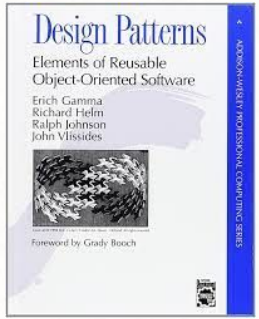
Une chance

Choix numéro 2

Livré à soi-même

- Pas vraiment d'outils pour progresser
- Ta seule option





Troisième option

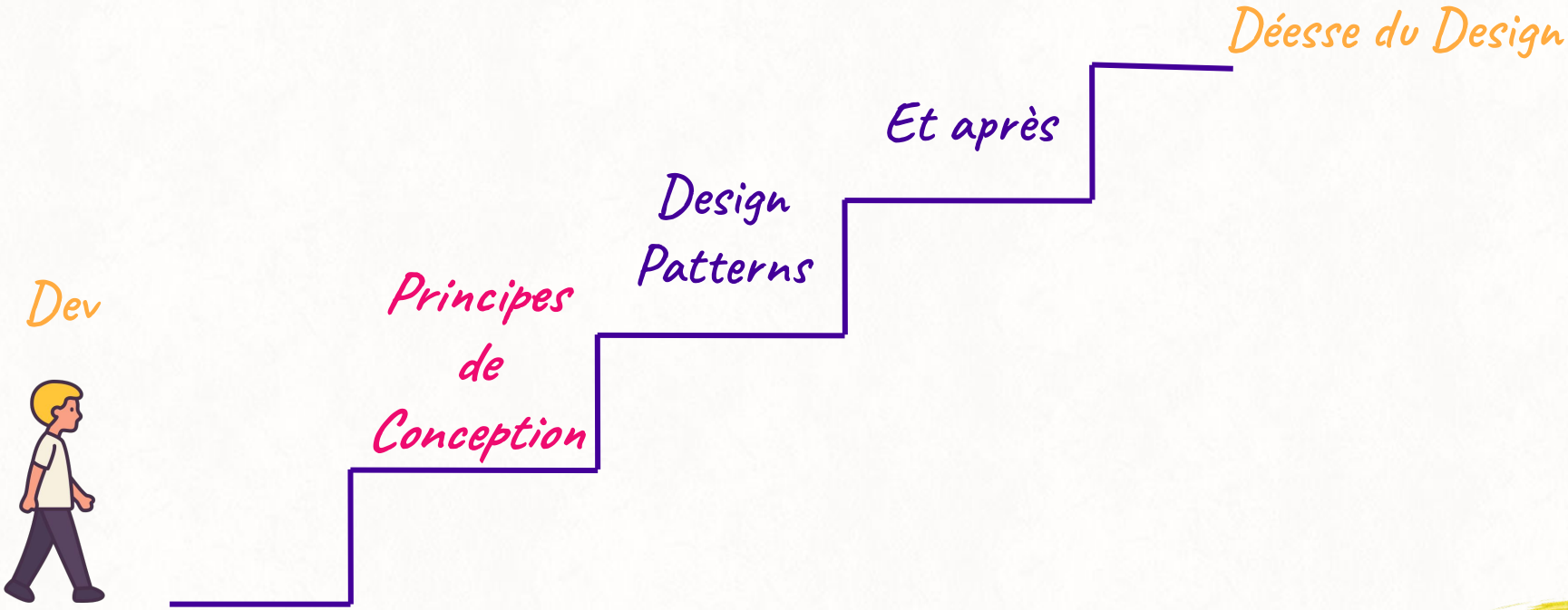
takima
mastering technology

Mathilde Lorrain
@mathildelorrain

Benjamin Yvernault
@BenYvernault

La progression en conception

Les étapes



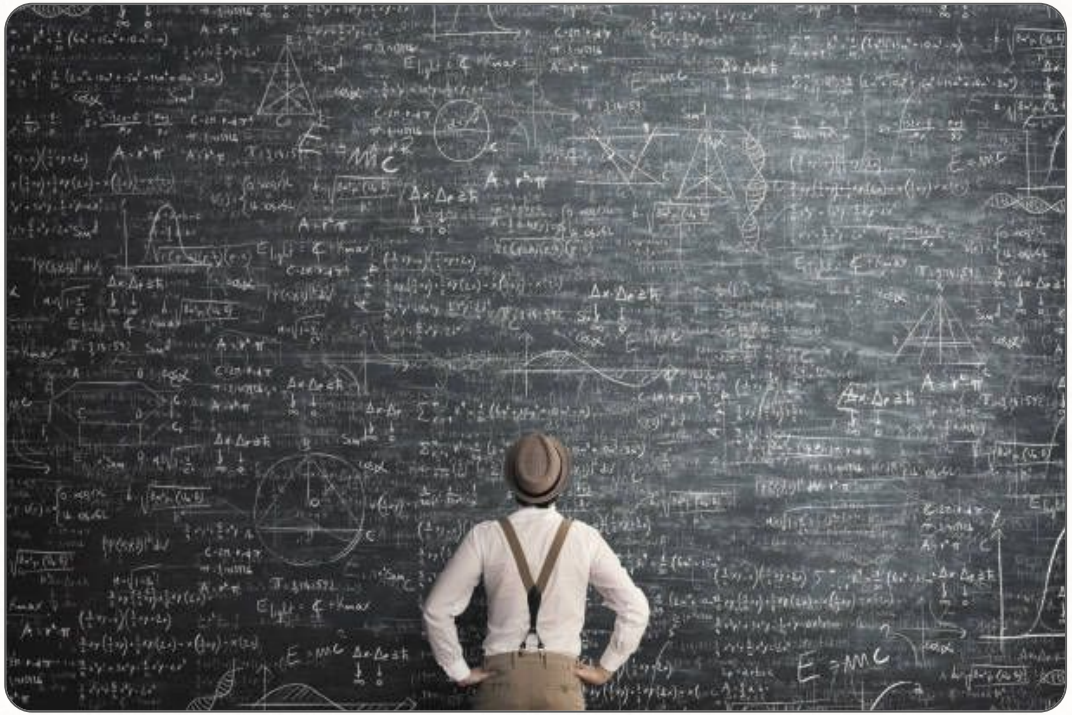
Quand on commence dans le dev

Eviter les mauvais réflexes

Quand on commence dans le dev

Ce qu’on a remarqué

1. La complexité avant tout !
   Avec une pointe **d’over engineering**



Quand on commence dans le dev

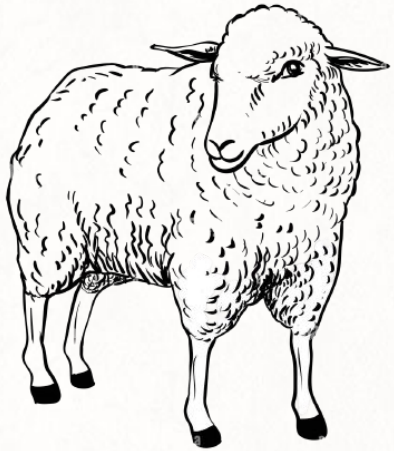
On aime bien la complexité !

Dessine-moi un mouton ?

Quand on commence dans le dev

On aime bien la complexité !

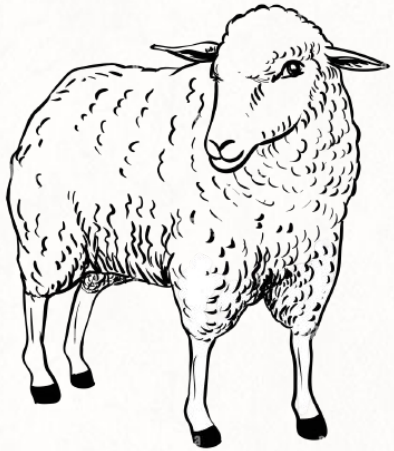
Dessine-moi un mouton ?

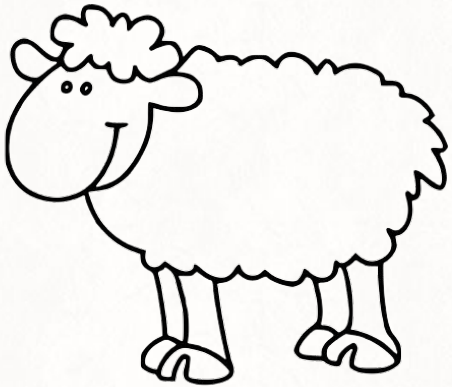


Quand on commence dans le dev

On aime bien la complexité !

Dessine-moi un mouton ?

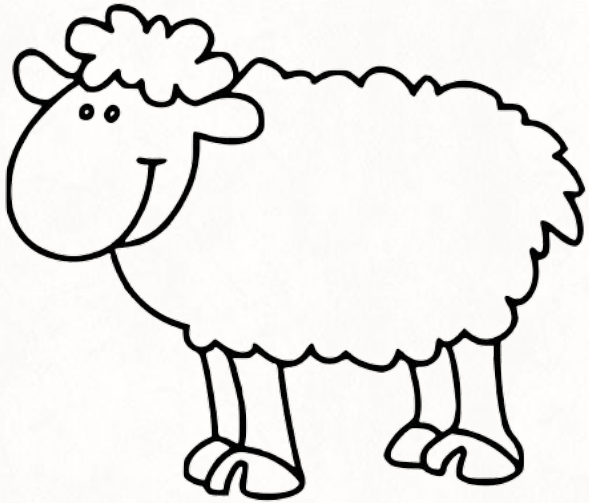




Quand on commence dans le dev

On veut de la simplicité !

KISS : Keep it simple, stupid !



“

Plus vous êtes expérimentés, plus votre code sera facile à lire.

”

Quand on commence dans le dev

Pour qu'il soit facile à lire

- Single responsibility
- Open/Closed
- Liskov Substitution
- Interface segregation
- Dependency Inversion

You Ain't Gonna Need It !

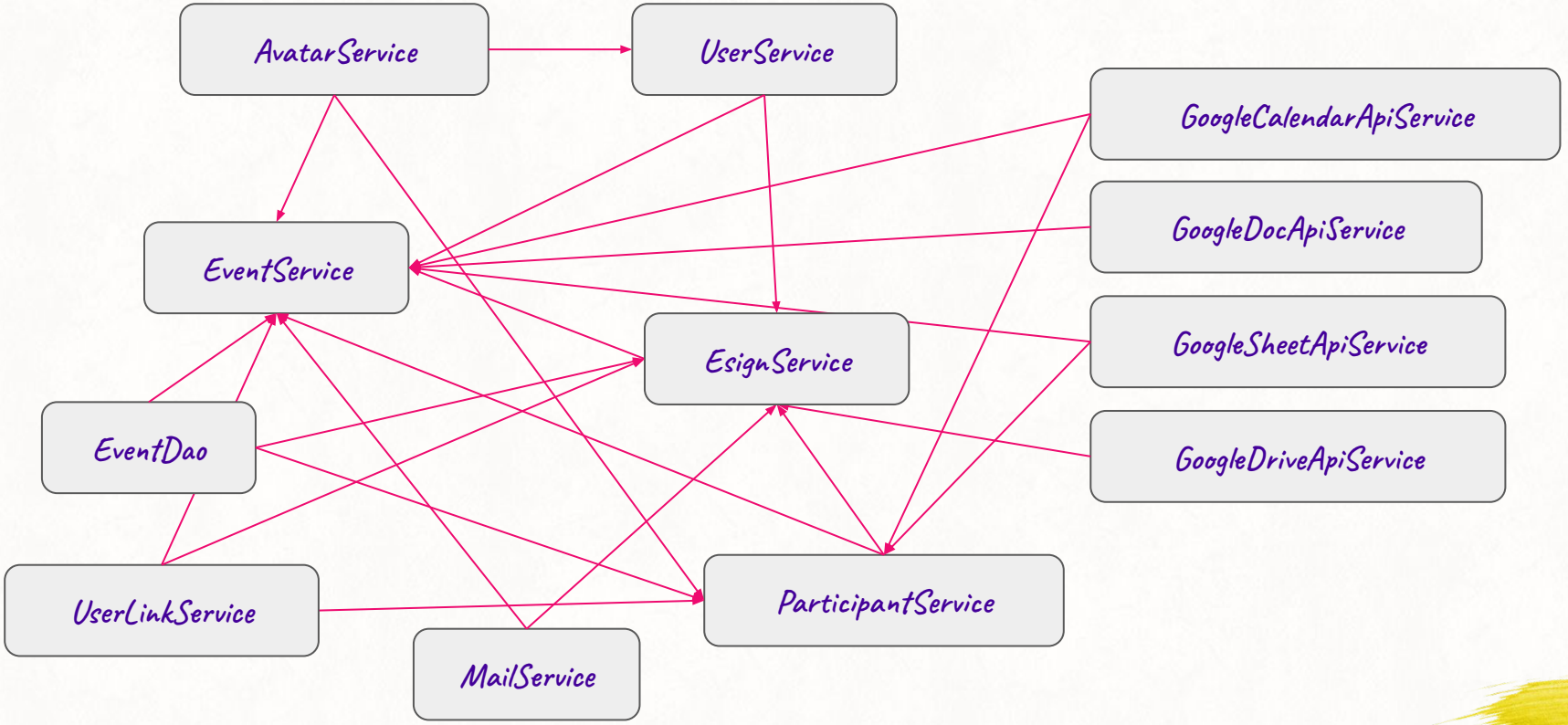
Keep It Simple, Stupid !

Clean Code

Code...

Couplage démentiel

Single responsibility



Quand on commence dans le dev

Ce qu’on a remarqué

2

L’héritage à toutes les sauces

“

L'héritage n'est pas la réponse à tout !

”

Quand on commence dans le dev

L’héritage

“

Comment tester mes moteurs ?

”

L’héritage ou pas

switchOn()
switchOff()

Quand on commence dans le dev

L’héritage

switchOn()
switchOff()

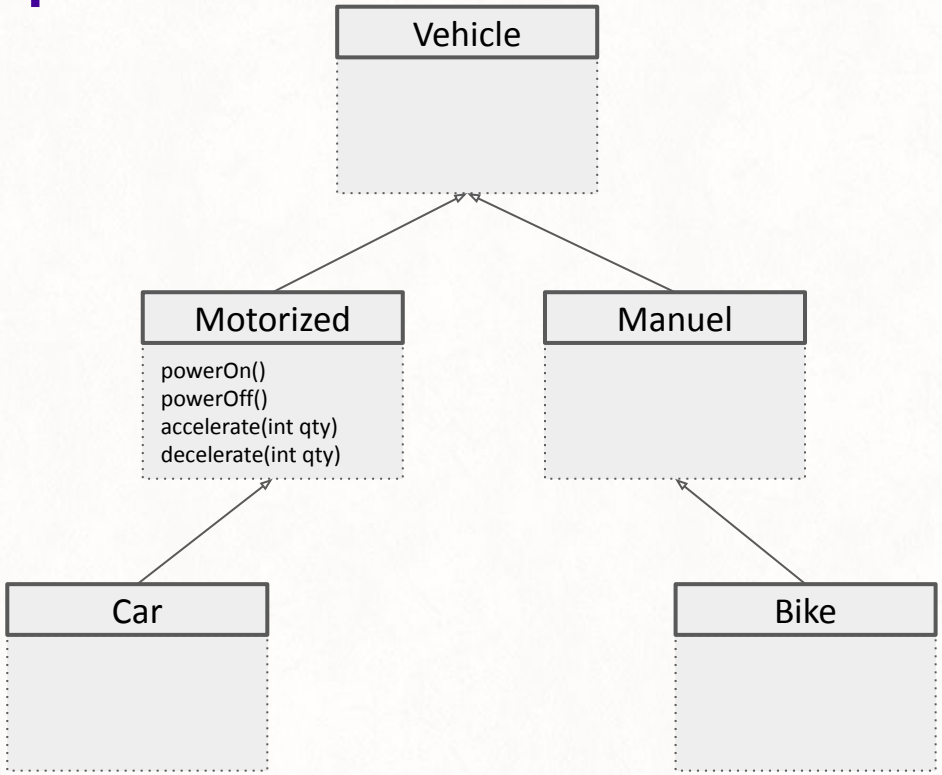
“

On aime tester un comportement

”

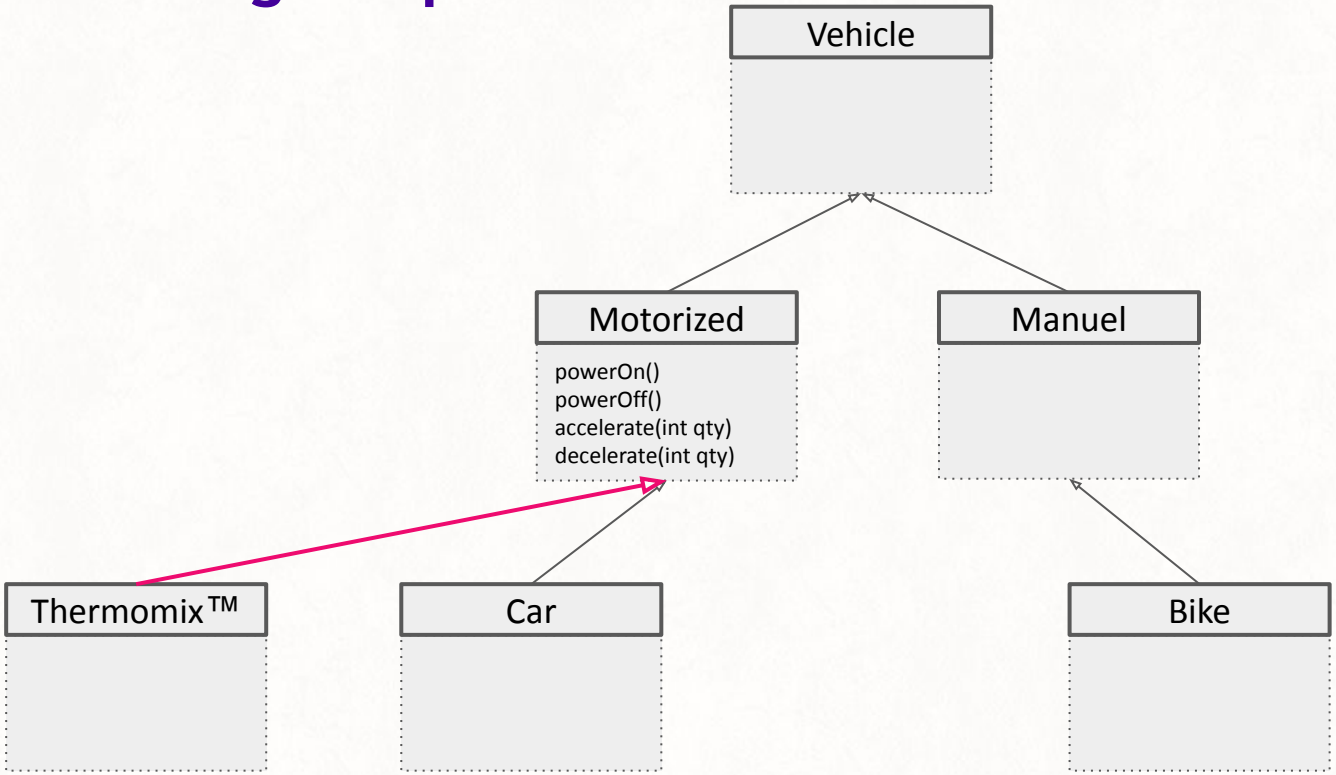
Quand on commence dans le dev

L’héritage ou pas



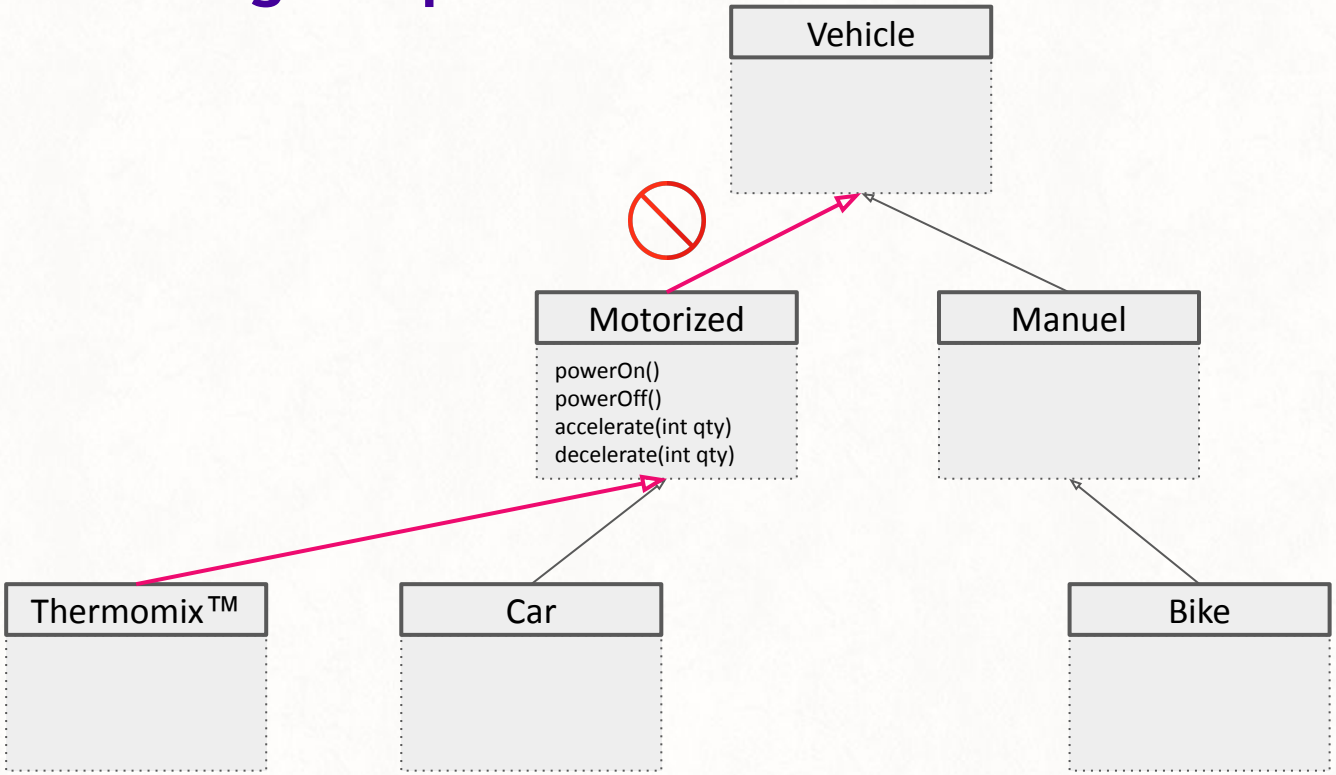
Quand on commence dans le dev

L’héritage ou pas



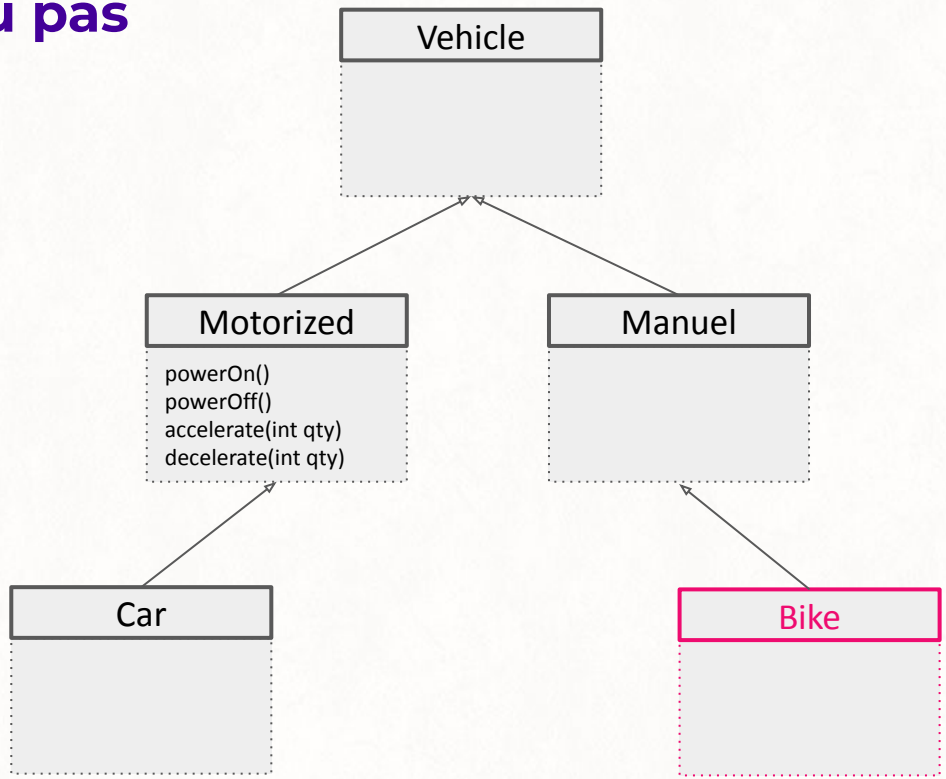
Quand on commence dans le dev

L’héritage ou pas



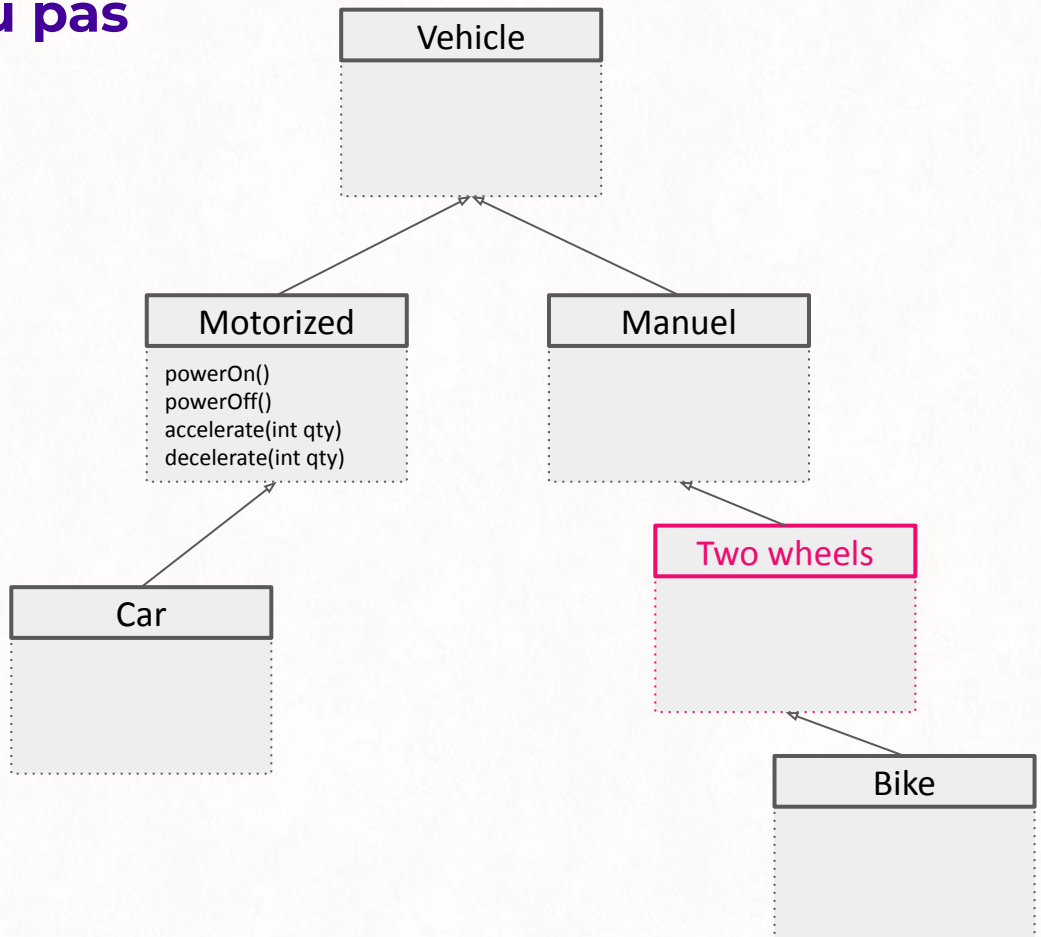
Quand on commence dans le dev

L’héritage ou pas



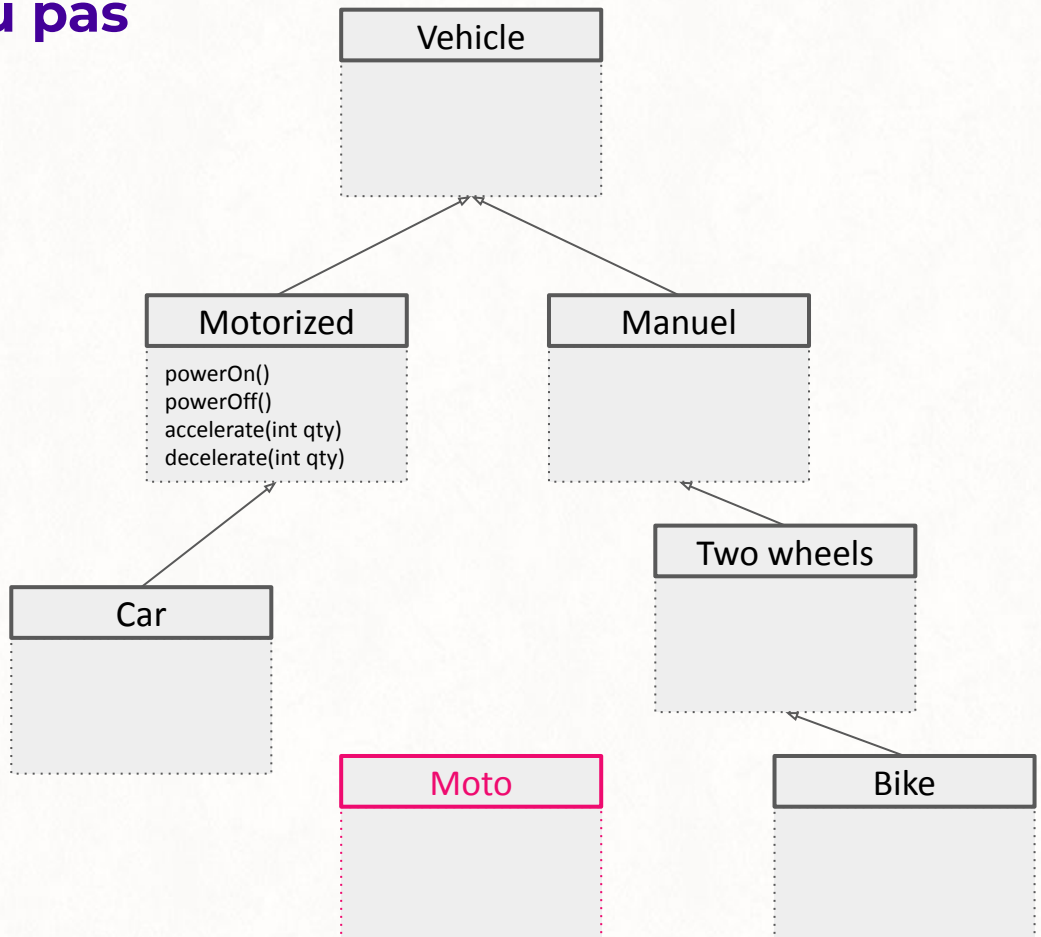
Quand on commence dans le dev

L’héritage ou pas



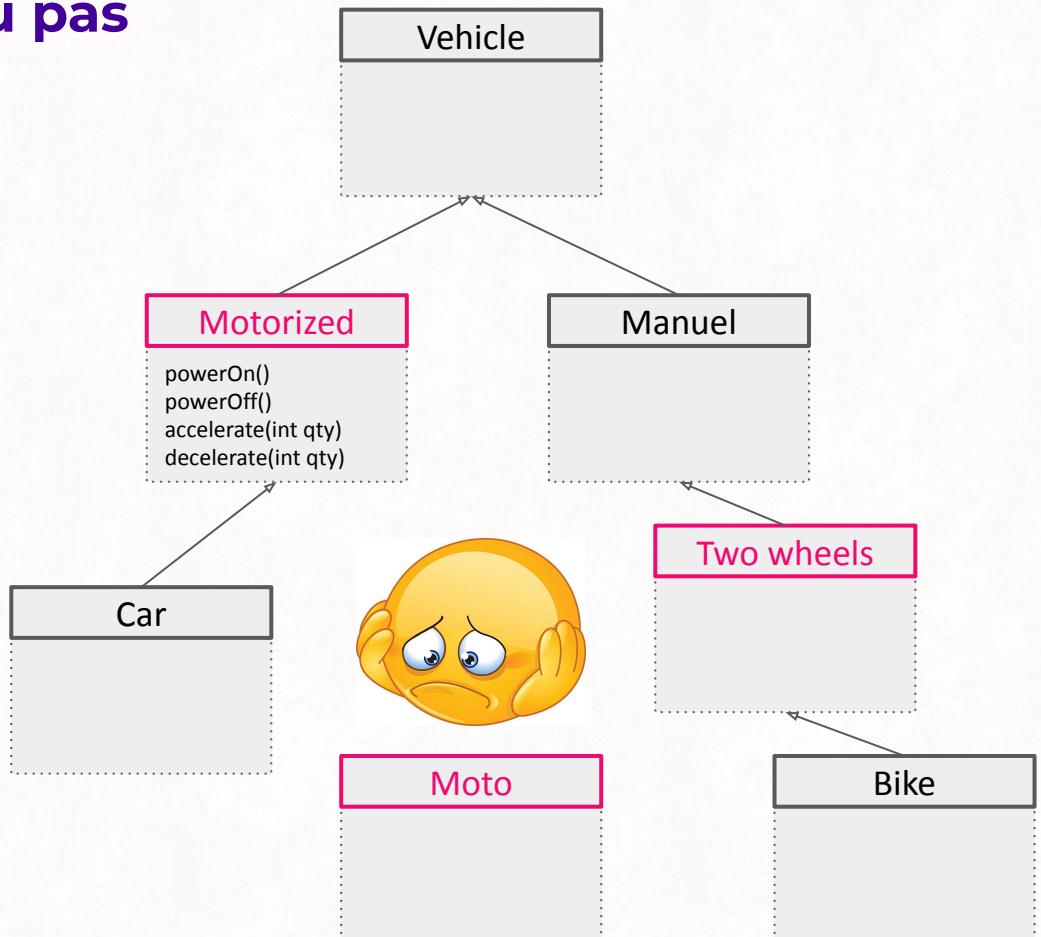
Quand on commence dans le dev

L'héritage ou pas



Quand on commence dans le dev

L’héritage ou pas



“

L’héritage, c’est une contrainte forte et hiérarchisée (<=> arborescence)

”

“

L’héritage, ce n’est pas optimal pour raisonner en comportements

”

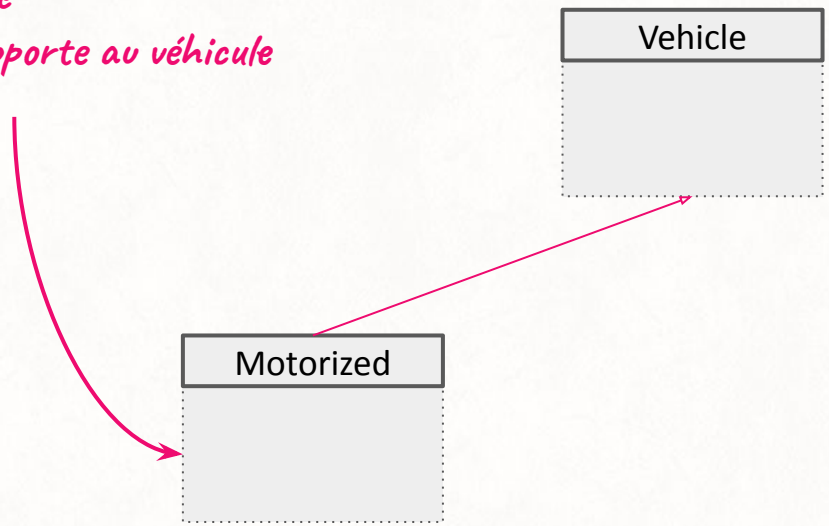
Quand on commence dans le dev

L’héritage ou pas

Quand on commence dans le dev

L'héritage ou pas

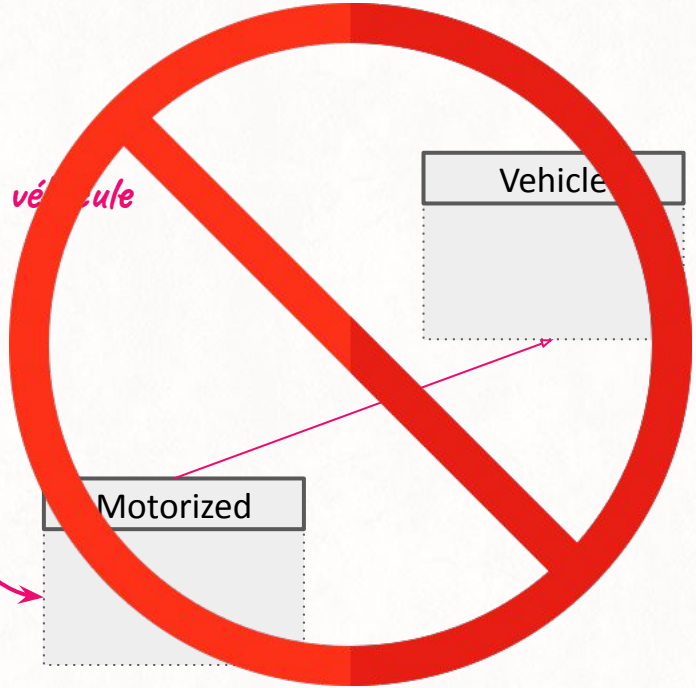
Tester ce que
le moteur apporte au véhicule



Quand on commence dans le dev

L'héritage ou pas

Tester ce que
le moteur apporte au véhicule



Quand on commence dans le dev

L'héritage

Problème de testabilité

Quand on commence dans le dev

L'héritage

Problème de testabilité

Problème pour débugger

Quand on commence dans le dev

L'héritage

Problème de testabilité

Problème de responsabilité unique

Problème pour débugger

Quand on commence dans le dev

L'héritage

Problème de testabilité

Problème de responsabilité unique

Problème pour débugger

Problème de couplage

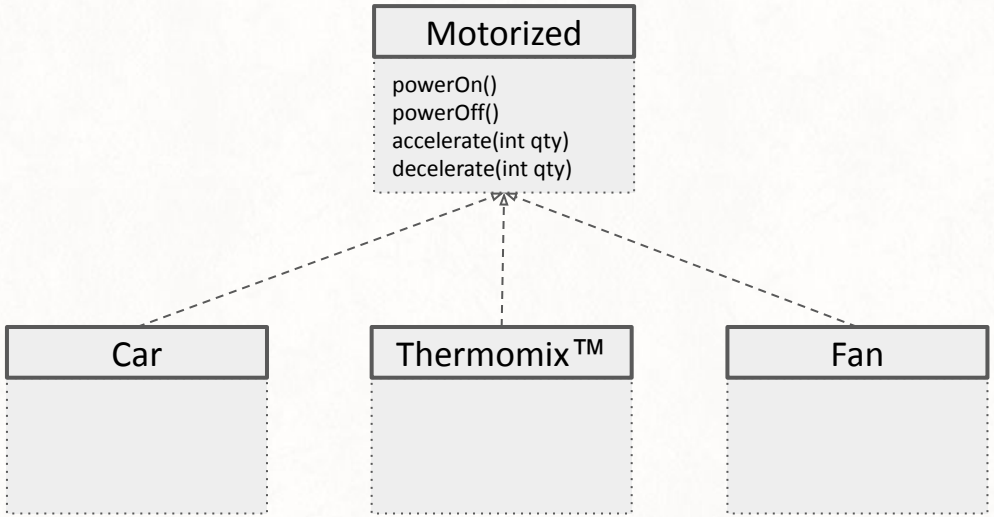
“

Héritage ne modélise pas les comportements, les interfaces et la composition si !

”

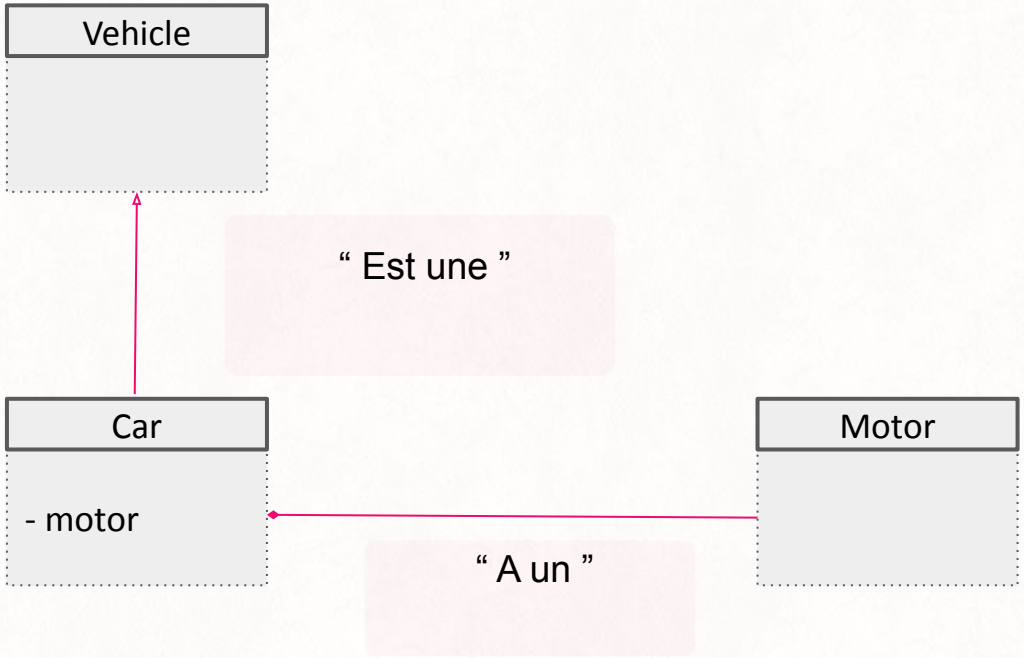
Quand on commence dans le dev

Faire des interfaces



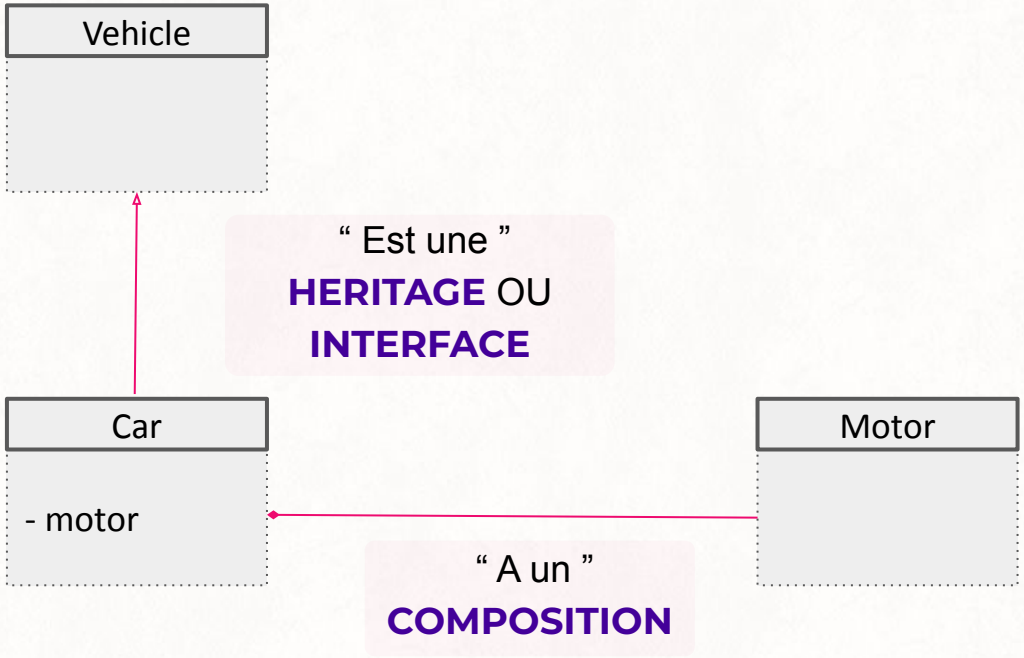
Quand on commence dans le dev

L’héritage vs Composition



Quand on commence dans le dev

L’héritage vs Composition



Quand on commence dans le dev, voilà nos caractéristiques

Ce qu'on a remarqué

1. La complexité avant tout ! Avec une pointe **d'over engineering**
2. **L'héritage** à toutes les sauces
3. Câblé **relationnel**
4. Mauvaise compréhension des **tests** et de leurs utilités
5. **Faible** connaissance en architecture
6. **Ajouter** du code plutôt que de refacto

La progression en conception

Les étapes

Déesse du Design

Et après

Design Patterns

Principes de Conception

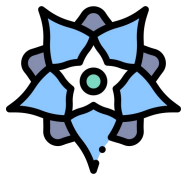
Dev

Ce que vous devez savoir

Les design patterns

“

Design object oriented software is hard, and designing reusable object-oriented software is even harder.



Gang of 4 Design Pattern

”

Les différents types de pattern

- Patterns de création
- Patterns structurels
- Patterns comportementaux

Les différents types de pattern

Patterns de création

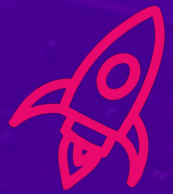
- Factory
- Builder
- Singleton

Patterns structurels

- Facade
- Proxy

Patterns comportementaux

- Strategy
- Visitor
- Template Methods
- Observer
- State



Ceux qu'on voit partout !

Les Patterns courants

Le factory pattern

Factory

→ Gérer la création de différents objets à travers une interface

→ Simplifie la création d'objet

- **org.slf4j.LoggerFactory**

Le singleton pattern

Patterns de création

Singleton

→ Permet de garantir qu'un objet ne vient qu'en une seule instance

- **La plupart des beans spring sont des singletons**

Le proxy pattern

Proxy

- ➔ Permet d'obtenir un proxy de notre objet et de le modifier à travers ce dernier
- ➔ Ajouter du comportement transverse

- **Lazy Loading dans Hibernate**
- **Annotations dans Spring (@Transactionnal)**

Le builder pattern

Patterns de création

Builder

- ➔ Création d'objet complexe
- ➔ Permettre de le construire plus facilement sans avoir à préciser tous les attributs

- **javax.persistence.criteria.CriteriaBuilder**

Les Design patterns

Le builder pattern

```
public class Burger {
   // all attributes
   private Burger(String bread, String condiments, String meat, String cheese, String dressing)
{...}
   public static Builder builder() {return new Builder();}
   private static class Builder {
       // all attributes
       private Builder() {}
       public Builder withBread(String bread) {
           this.bread = bread;
           return this;
       }
       // other methodes
       public Burger build() {
           return new Burger(bread, condiments, meat, cheese, dressing);}
   }
```

Les Design patterns

Le builder pattern

```java
public class Burger {
   // all attributes
   private Burger(String bread, String condiments, String meat, String cheese, String dressing) {...}
   public static Builder builder() {return new Builder();}
   private static class Builder {
       // all attributes
       private Builder() {}
       public Builder withBread(String bread) {
           this.bread = bread;
           return this;
       }
       // other methodes
       public Burger build() {
           return new Burger(bread, condiments, meat, cheese, dressing);}
   }
```

Un constructor privé

Les Design patterns

Le builder pattern

```
public class Burger {
   // all attributes
   private Burger(String bread, String condiments, String meat, String cheese, String dressing)
{...}
   public static Builder builder() {return new Builder();}
   private static class Builder {
       // all attributes
       private Builder() {}
       public Builder withBread(String bread) {
           this.bread = bread;
           return this;
       }
       // other methodes
       public Burger build() {
           return new Burger(bread, condiments, meat, cheese, dressing);}
   }
```

Une classe builder

Les Design patterns

Le builder pattern

```
public class Burger {
   // all attributes
   private Burger(String bread, String condiments, String meat, String cheese, String dressing)
{...}
   public static Builder builder() {return new Builder();}
   private static class Builder {
       // all attributes
       private Builder() {}
       public Builder withBread(String bread) {
           this.bread = bread;
           return this;
       }
       // other methodes
       public Burger build() {
           return new Burger(bread, condiments, meat, cheese, dressing);}
   }
```

Le builder pattern

```
Burger noCheeseBurger = Burger.builder()
                .withBread("bagel")
                .withMeat("boeuf")
                .withCondiments("cornichons et salade")
                .withDressing("ketchup")
                .build();
```

On précise seulement les propriétés dont on a besoin (pas de fromage)

```
Burger veggieBurger = Burger.builder()
                .withBread("pain brioché")
                .withCheese("cheddar")
                .withCondiments("cornichons, salade, tomate")
                .build();
```

Le builder est moins utile en Kotlin

```
data class Burger(
 val bread: String?,
 val condiments: String?,
 val meat: String?,
 val cheese: String?,
 val dressing: String?)
```

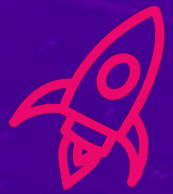
```
val noCheeseBurger = Burger(
     bread = "bagel",
     condiments = "cornichons et salade",
     meat ="boeuf",
     dressing = "ketchup")

val vegyBurger = Burger(
        bread = "pain brioché",
        condiments = "cornichons, salade, tomate",
        cheese = "cheddar")
```

Named Parameters

“

Les design patterns sont des solutions à des problèmes de conception courants résultant de l'utilisation d'un langage de programmation.

”

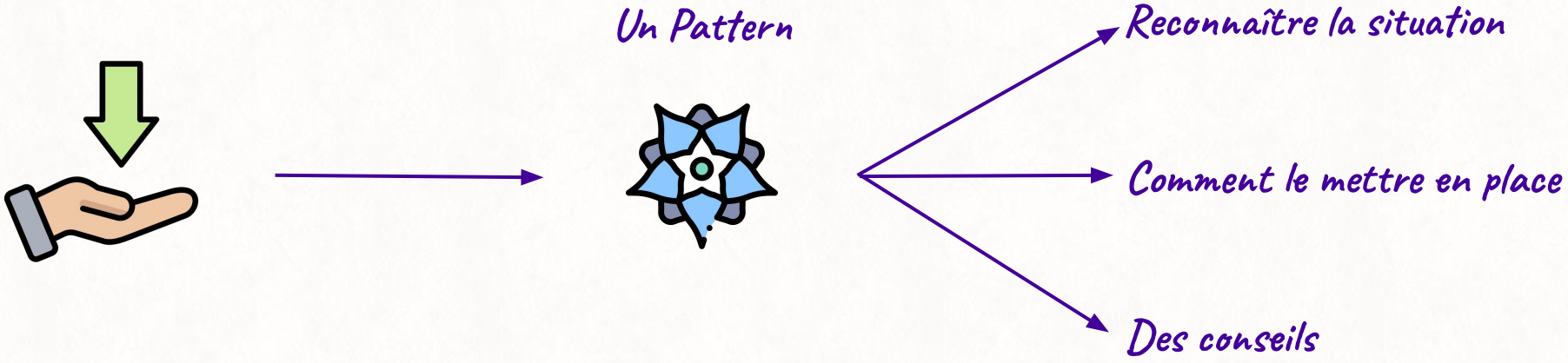


Tous les patterns ne se valent pas

Notre Best-of

Notre Best-of

Comment on va procéder ?



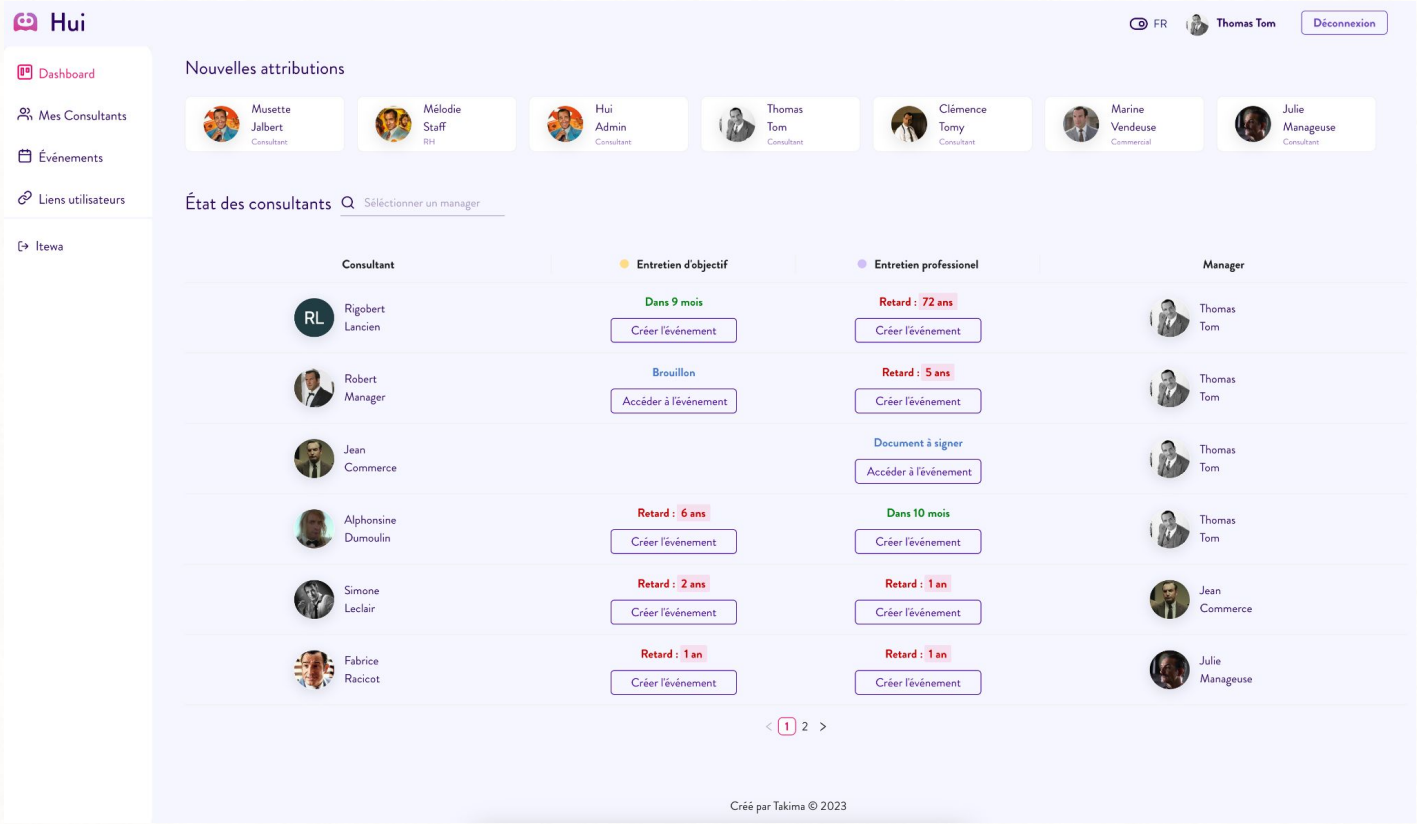
“

C’est quand même mieux avec un exemple

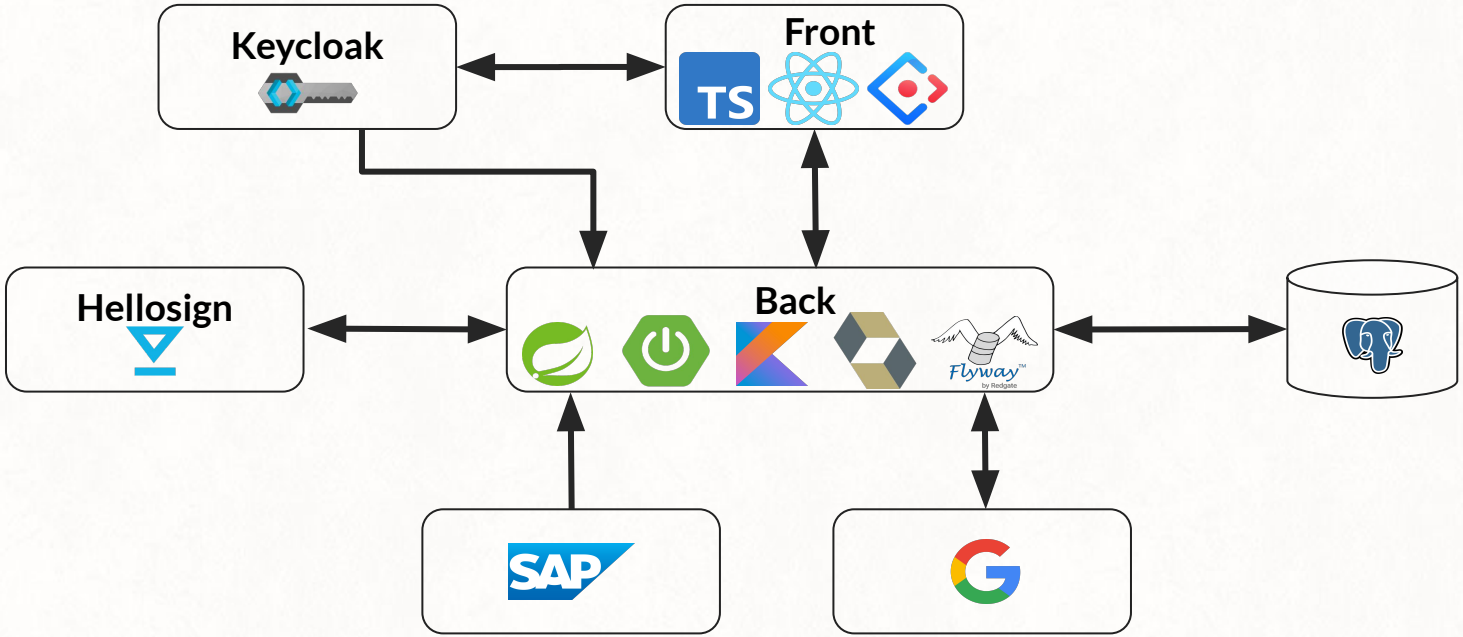
”

Projet HUI

Une application de suivi managérial



La stack



L'équipe

Lead Dev

Po

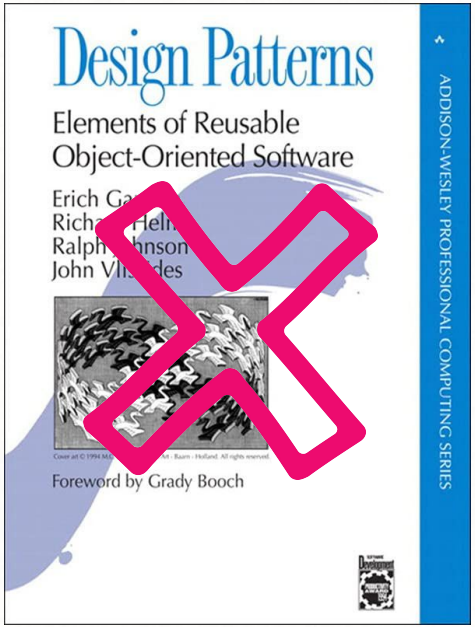
Développeurs en formation

L'équipe

Lead Dev

Po

Développeurs en formation



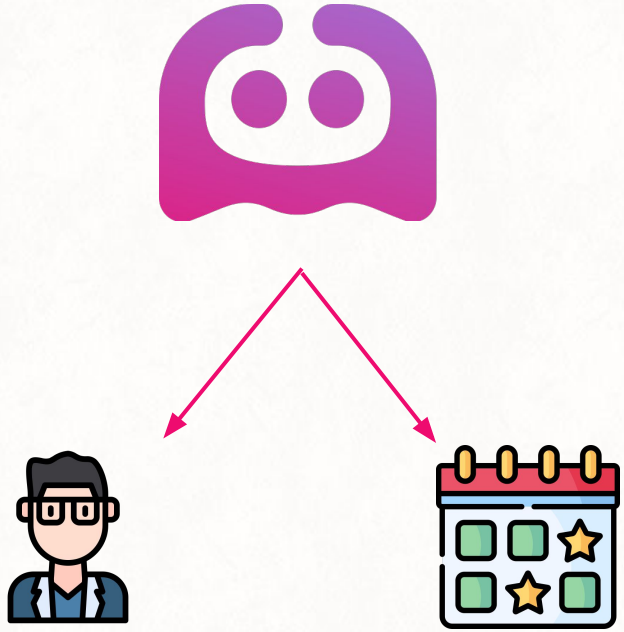
L'équipe

Un terrain de jeu parfait

Premier besoin

Différents types d'événement

Contexte du projet HUI



Différents types d’évènements

Les événements

Entretien d’objectifs

Entretien de suivi

Formation

Différents types d'évènements

Des caractéristiques par événement

DOCUMENT À REMPLIR

Entretien d'objectifs

Entretien de suivi

Formation

Différents types d'évènements

Des caractéristiques par événement

Entretien d'objectifs

SIGNATURE D'UN DOCUMENT

Entretien de suivi

Formation

Différents types d'évènements

Des caractéristiques par événement

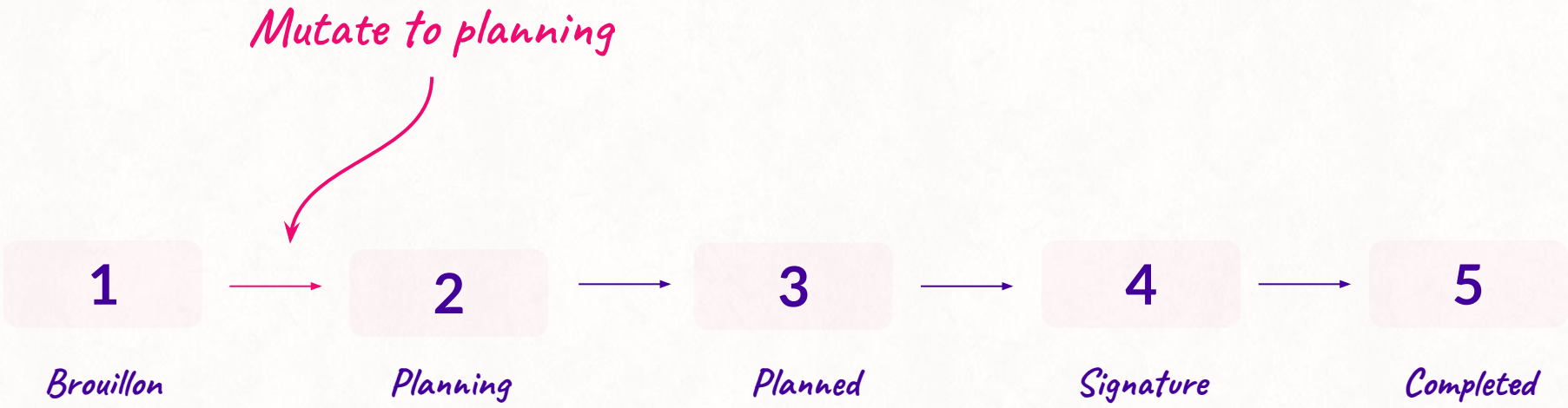
Entretien d'objectifs

Entretien de suivi

INFORMATIF

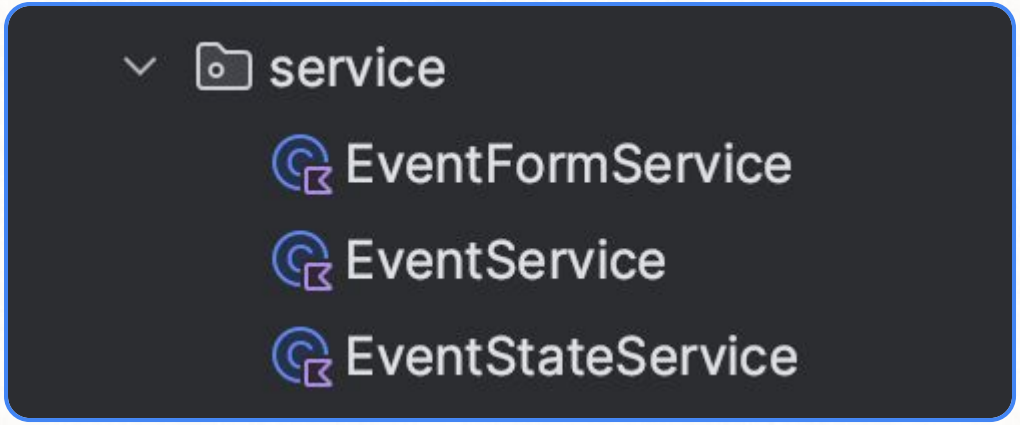
Formation

Progression d'un événement



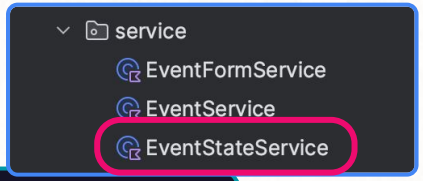
Différents types d’évènements

Les classes présentes



Différents types d’évènements

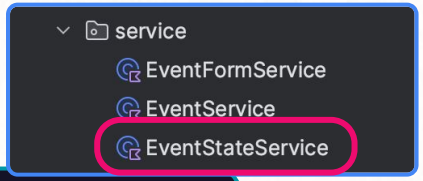
L’état du code




```
private fun mutateToPlanning(event: Event) {
    if (event.eventType.hasGoogleDoc) {
        eventFormService.instantiateForm(event)
        eventFormService.fillForm(event)
    }
    when (event.eventType) {
        GOALS_INTERVIEW,
        PROFESSIONAL_INTERVIEW,
        FOLLOW_UP_INTERVIEW -> {
            doSomething(...)
        }
        OTHER_HR_EVENT -> {
            doSomething(...)
        }
```

Différents types d'évènements

L'état du code

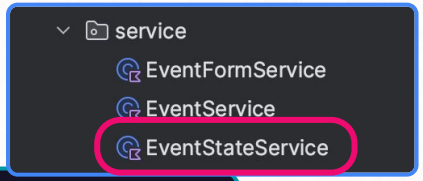



```
private fun mutateToPlanning(event: Event) {
    if (event.eventType.hasGoogleDoc) {
        eventFormService.instantiateForm(event)
        eventFormService.fillForm(event)
    }
    when (event.eventType) {
        GOALS_INTERVIEW,
        PROFESSIONAL_INTERVIEW,
        FOLLOW_UP_INTERVIEW -> {
            doSomething(...)
        }
        OTHER_HR_EVENT -> {
            doSomething(...)
        }
```

Création du formulaire via google doc

Différents types d'évènements

L'état du code

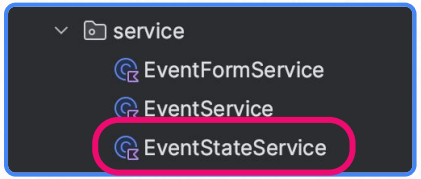



```
private fun mutateToPlanning(event: Event) {
    if (event.eventType.hasGoogleDoc) {
        eventFormService.instantiateForm(event)
        eventFormService.fillForm(event)
    }
    when (event.eventType) {
        GOALS_INTERVIEW,
        PROFESSIONAL_INTERVIEW,
        FOLLOW_UP_INTERVIEW -> {
            doSomething(...)
        }
        OTHER_HR_EVENT -> {
            doSomething(...)
        }
```

Spécifique aux entretiens

Différents types d'évènements

L'état du code




```
            PROFESSIONAL_INTERVIEW_TRANSVERSE -> {
                doSomething(...)
            }
        }

        event.eventState = PLANNING
    }
```

Mise à jour de l'état de l'événement

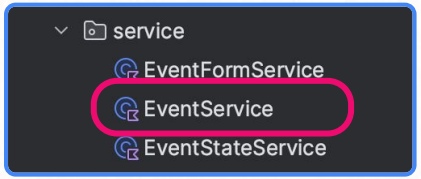
Différents types d'évènements

L'état du code

Formation

```
DRAFT -> {
    if (event.eventType == TRAINING_COURSE) {
        mutateToComplete(event)
    } else {
        mutateToPlanning(event)
    }
}
```

Planification d'un training course à part



Les problèmes

Problème de lisibilité

Différents types d'évènements

Les problèmes

Problème de lisibilité

Changements à plusieurs endroits

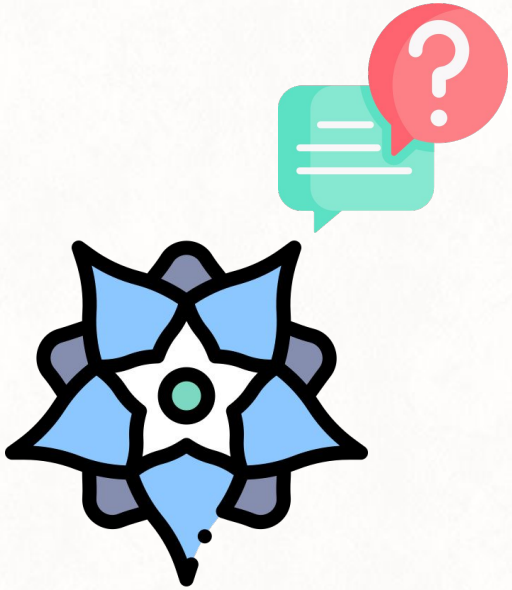
Les problèmes

Problème de lisibilité

Changements à plusieurs endroits

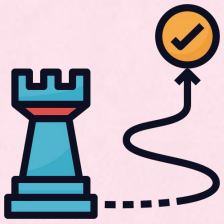
Une étape différente par type d'événements

Quel pattern ?



Le Strategy pattern

Carte d'identité



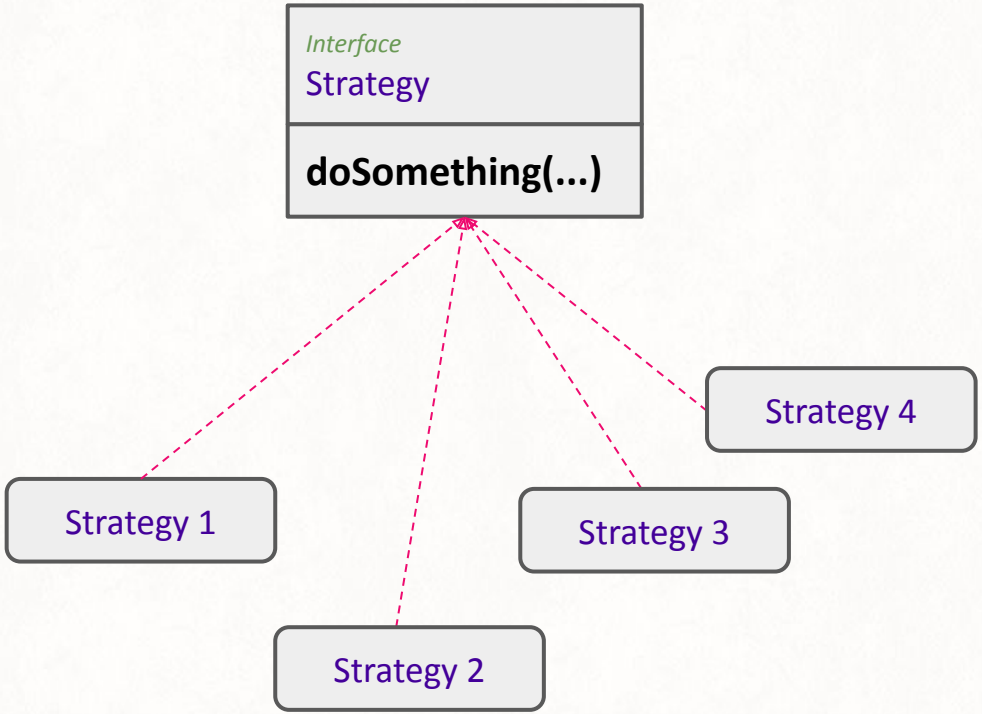
Strategy

Famille : pattern **comportemental**

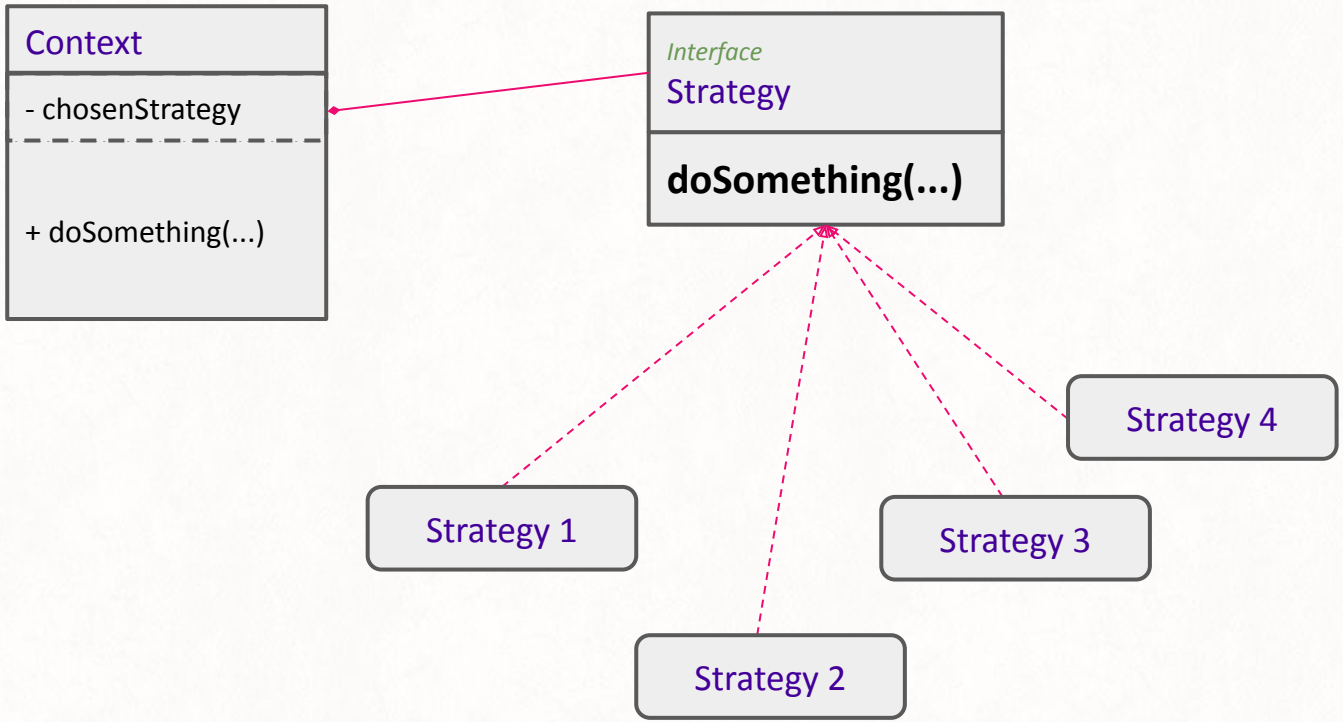
Utilité : Rend une partie d'un algorithme **interchangeable** en fonction d'un choix donné

Carte d'identité

Carte d'identité



Carte d'identité



Créer les stratégies pour deux types d'événements

Entretien d'objectifs

Formation

Deux Stratégies

Extraire les stratégies

Choisir la bonne stratégie en fonction du type

```
interface EventTypeStrategy {
    fun supports(eventType: EventType): Boolean
    fun plan(event: Event)
}
```

Extraire les stratégies


```kotlin
interface EventTypeStrategy {
    fun supports(eventType: EventType): Boolean
    fun plan(event: Event)
}
```

Planification d'un événement

Extraire les stratégies

Entretien d'objectifs

```
@Component
class GoalsStrategy(private val eventFormService: EventFormService,
                    private val mailService: MailService) : EventTypeStrategy {
    override fun supports(eventType: EventType): Boolean {
        return eventType == EventType.GOALS_INTERVIEW;
    }

    override fun plan(event: Event) {
        eventFormService.createForm(event)
        mailService.sendPlanningMailForInterviewEvent(event)
        event.eventState = EventStateType.PLANNING
    }
}
```

Extraire les stratégies

Entretien d'objectifs

```
@Component
class GoalsStrategy(private val eventFormService: EventFormService,
                    private val mailService: MailService) : EventTypeStrategy {
    override fun supports(eventType: EventType): Boolean {
        return eventType == EventType.GOALS_INTERVIEW;
    }

    override fun plan(event: Event) {
        eventFormService.createForm(event)
        mailService.sendPlanningMailForInterviewEvent(event)
        event.eventState = EventStateType.PLANNING
    }
}
```

Extraire les stratégies

Entretien d'objectifs

```
@Component
class GoalsStrategy(private val eventFormService: EventFormService,
                    private val mailService: MailService) : EventTypeStrategy {
    override fun supports(eventType: EventType): Boolean {
        return eventType == EventType.GOALS_INTERVIEW;
    }

    override fun plan(event: Event) {
        eventFormService.createForm(event)
        mailService.sendPlanningMailForInterviewEvent(event)
        event.eventState = EventStateType.PLANNING
    }
}
```

Création d'un formulaire + envoi d'email + passage à planning

Extraire les stratégies


```
@Component
class TrainingCourseStrategy : EventTypeStrategy {
    override fun supports(eventType: EventType): Boolean {
        return eventType == EventType.TRAINING_COURSE
    }

    override fun plan(event: Event) {
        event.eventState = EventStateType.COMPLETE
    }
}
```

Extraire les stratégies

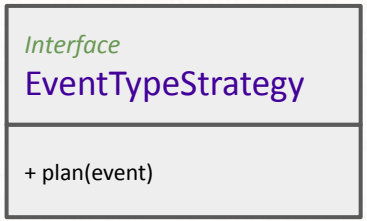

```
@Component
class TrainingCourseStrategy : EventTypeStrategy {
    override fun supports(eventType: EventType): Boolean {
        return eventType == EventType.TRAINING_COURSE
    }

    override fun plan(event: Event) {
        event.eventState = EventStateType.COMPLETE
    }
}
```

Passage à Complete

Comment appeler nos stratégies

Entretien d'objectifs

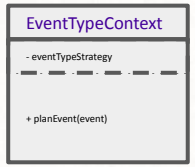
Formation



Une interface

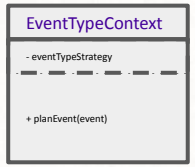
Deux implémentations

Le contexte




```
class EventTypeContext(private val eventTypeStrategy: EventTypeStrategy) {

    fun planEvent(event: Event) {
        eventTypeStrategy.plan(event)
    }

}
```

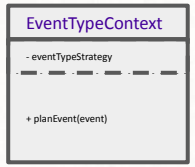
Le contexte




```
class EventTypeContext(private val eventTypeStrategy: EventTypeStrategy) {

    fun planEvent(event: Event) {
        eventTypeStrategy.plan(event)
    }

}
```

On donne la stratégie à notre contexte.

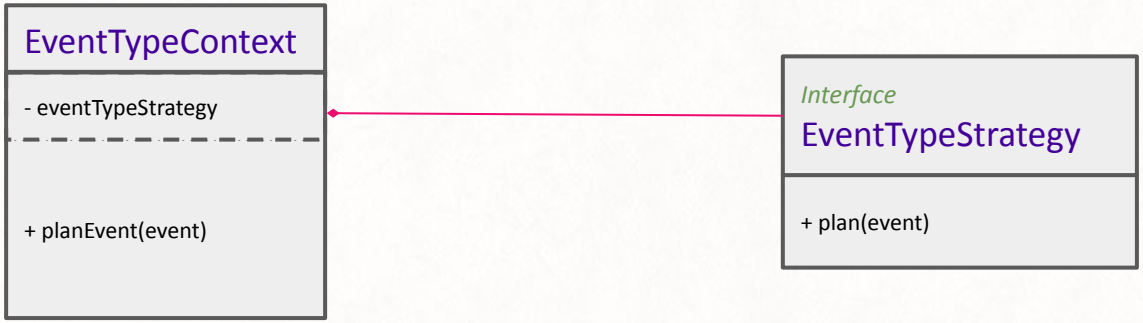
Le contexte




```
class EventTypeContext(private val eventTypeStrategy: EventTypeStrategy) {

    fun planEvent(event: Event) {
        eventTypeStrategy.plan(event)
    }

}
```

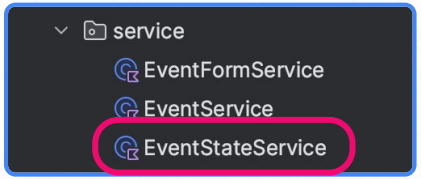
La méthode de notre algo pour planifier un événement qui appelle la bonne stratégie.

Le contexte pour appeler la bonne stratégie



Le strategy pattern

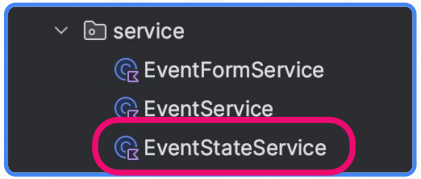
Le contexte




```
@Service
class EventStateService(
  private val eventFormService: EventFormService,
  private val eventStrategies: Set<EventTypeStrategy>,
) {
   //...
   fun mutateToPlanning(event: Event) {
      val eventStrategy = eventStrategies
          .find{strategy -> strategy.supports(event.eventType)}
          .orElseThrow()
      val context = EventTypeContext(eventStrategy)
      context.planEvent(event)
   }
}
```

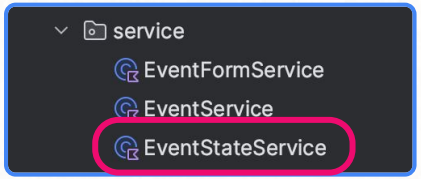
Le contexte

```
@Service
class EventStateService(
  private val eventFormService: EventFormService,
  private val eventStrategies: Set<EventTypeStrategy>,
) {
   //...
   fun mutateToPlanning(event: Event) {
      val eventStrategy = eventStrategies
          .find{strategy -> strategy.supports(event.eventType)}
          .orElseThrow()
      val context = EventTypeContext(eventStrategy)
      context.planEvent(event)
   }
}
```



Set de stratégies disponibles

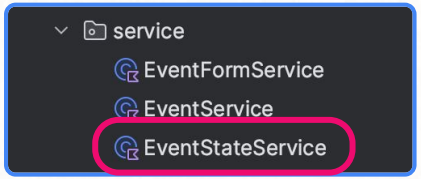
Le contexte




```
@Service
class EventStateService(
  private val eventFormService: EventFormService,
  private val eventStrategies: Set<EventTypeStrategy>,
) {
   //...
   fun mutateToPlanning(event: Event) {
      val eventStrategy = eventStrategies
          .find{strategy -> strategy.supports(event.eventType)}
          .orElseThrow()
      val context = EventTypeContext(eventStrategy)
      context.planEvent(event)
   }
}
```

Sélection de la stratégie

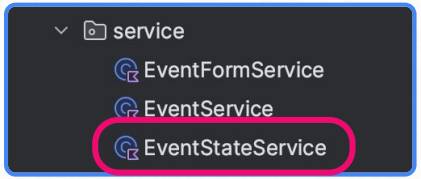
Le contexte




```
@Service
class EventStateService(
  private val eventFormService: EventFormService,
  private val eventStrategies: Set<EventTypeStrategy>,
) {
   //...
   fun mutateToPlanning(event: Event) {
      val eventStrategy = eventStrategies
          .find{strategy -> strategy.supports(event.eventType)}
          .orElseThrow()
      val context = EventTypeContext(eventStrategy)
      context.planEvent(event)
   }
}
```

Création du contexte

Le contexte

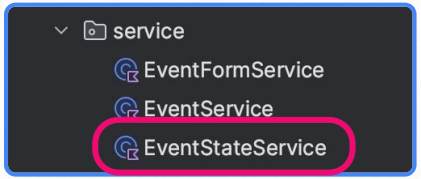



```
@Service
class EventStateService(
  private val eventFormService: EventFormService,
  private val eventStrategies: Set<EventTypeStrategy>,
) {
   //...
   fun mutateToPlanning(event: Event) {
      val eventStrategy = eventStrategies
          .find{strategy -> strategy.supports(event.eventType)}
          .orElseThrow()
      val context = EventTypeContext(eventStrategy)
      context.planEvent(event)
   }
}
```

Planification de l'événement

Le strategy pattern

Avant vs Après

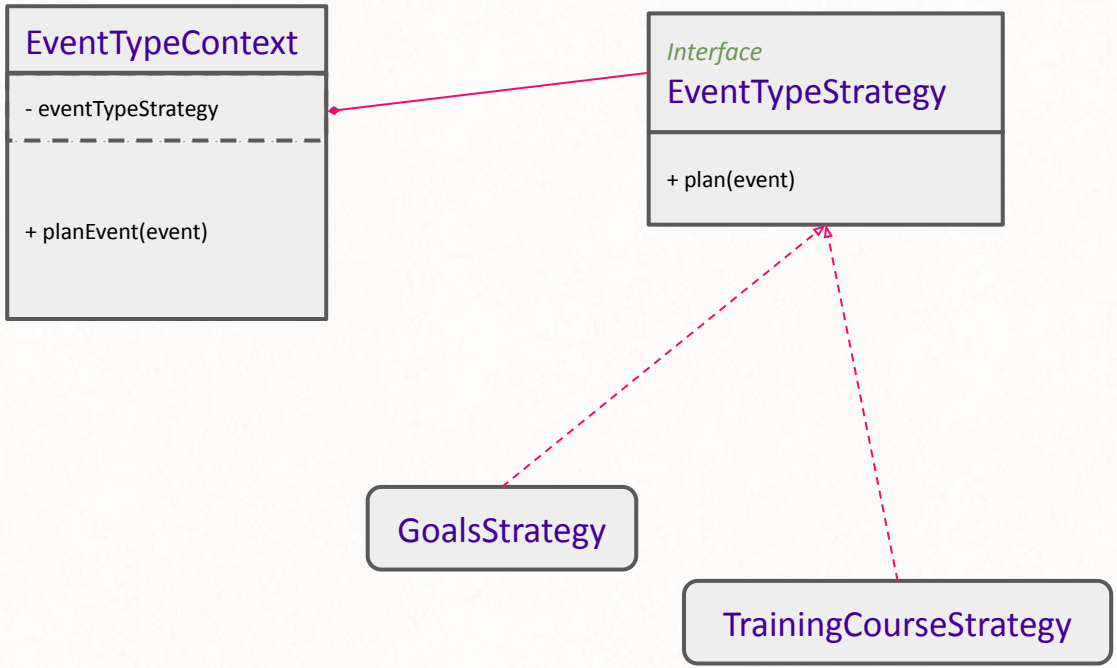



```
private fun mutateToPlanning(event: Event) {
    if (event.eventType.hasGoogleDoc) {
        eventFormService.instantiateForm(event)
        eventFormService.fillForm(event)
    }

    when (event.eventType) {
        GOALS_INTERVIEW,
        PROFESSIONAL_INTERVIEW,
        FOLLOW_UP_INTERVIEW -> {
            doSomething(...)
        }
        MANAGER_TEAM_BUILDING -> {
            doSomething(...)
        }
```

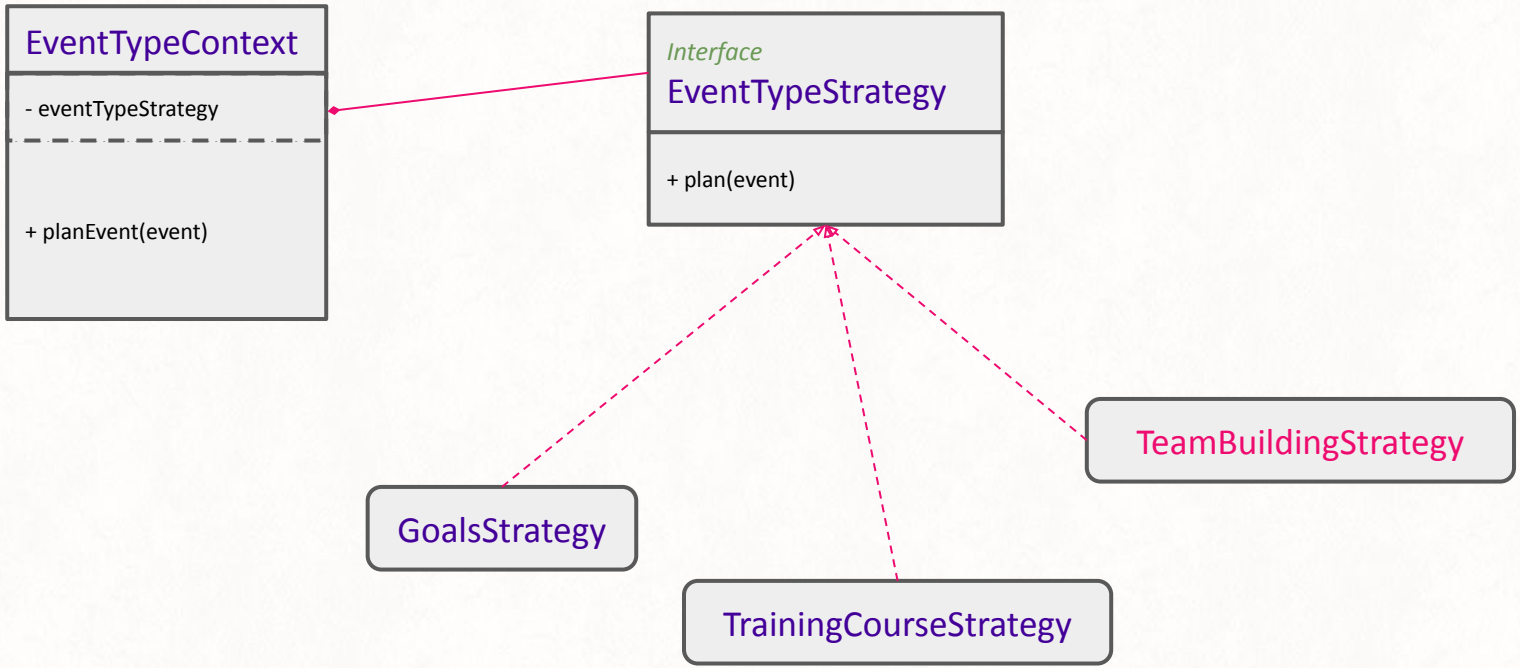
```
private fun mutateToPlanning(event: Event) {
    val eventStrategy = eventStrategies
        .find {strategy ->
strategy.supports(event.eventType)}
        .orElseThrow()
    val context = EventTypeContext(eventStrategy)
    context.planEvent(event)
}
```

SIMPLE

Schématisation



Nouveau type



Nouvelle stratégie

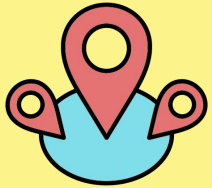
Je rajoute juste une nouvelle classe

```
@Component
class TeamBuildingStrategy(private val eventFormService: EventFormService) : EventTypeStrategy {
    override fun supports(eventType: EventType): Boolean {
        return eventType == EventType.TEAM_BUILDING
    }

    override fun plan(event: Event) {
        val participants = event.participants
            .filter { it.participantId.self != event.creator }
            .map { it.participantId.self }
        mailService.sendPlanningMailForTeamBuildingEvent(
            participants,
            event.id,
        )
        event.eventState = EventStateType.PLANNED
    }
}
```

Il permet de rendre une partie de notre algorithme interchangeable en fonction du choix client.

Ce qu'il faut retenir

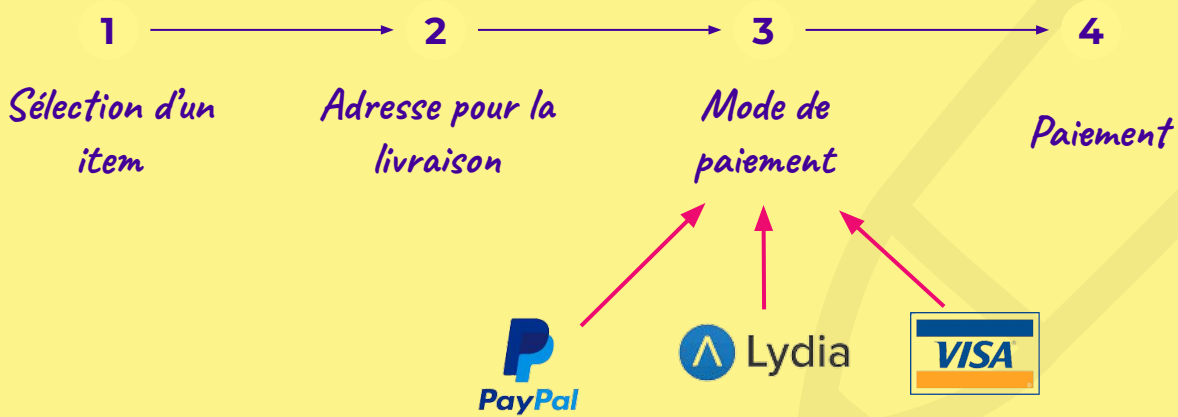
Lecture ++



Un seul endroit

Évolution simplifiée

- Processus d'achat sur internet
- Le paiement qui **varie** selon le choix du client



➔ Mode de transport pour venir au MixIt

➔ En fonction du **coût** et du **temps**, vous choisissez un **type de transport**





“

Ça, c’était le Strategy pattern !

”

Différents types d'événements

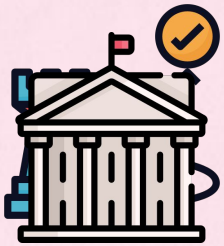
Deux secondes … des états ?

1 → 2 → 3 → 4 → 5

Brouillon — Planning — Planned — Signature — Completed

Le State pattern

Carte d'identité



State

Famille : pattern **comportemental**

Utilité : Il permet à un objet de modifier son comportement lorsque son état interne change.

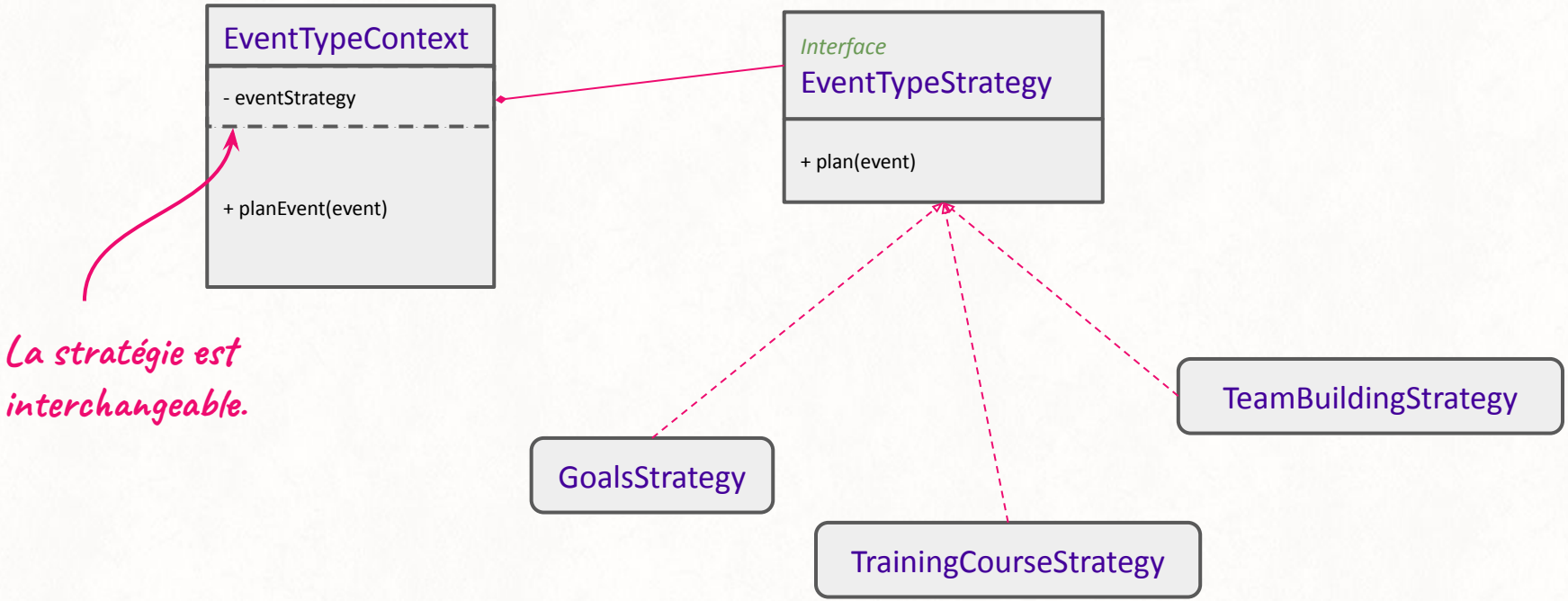
Les évolutions du pattern

Le state pattern est une **très proche** du strategy pattern.

Ils se basent sur la **composition**.

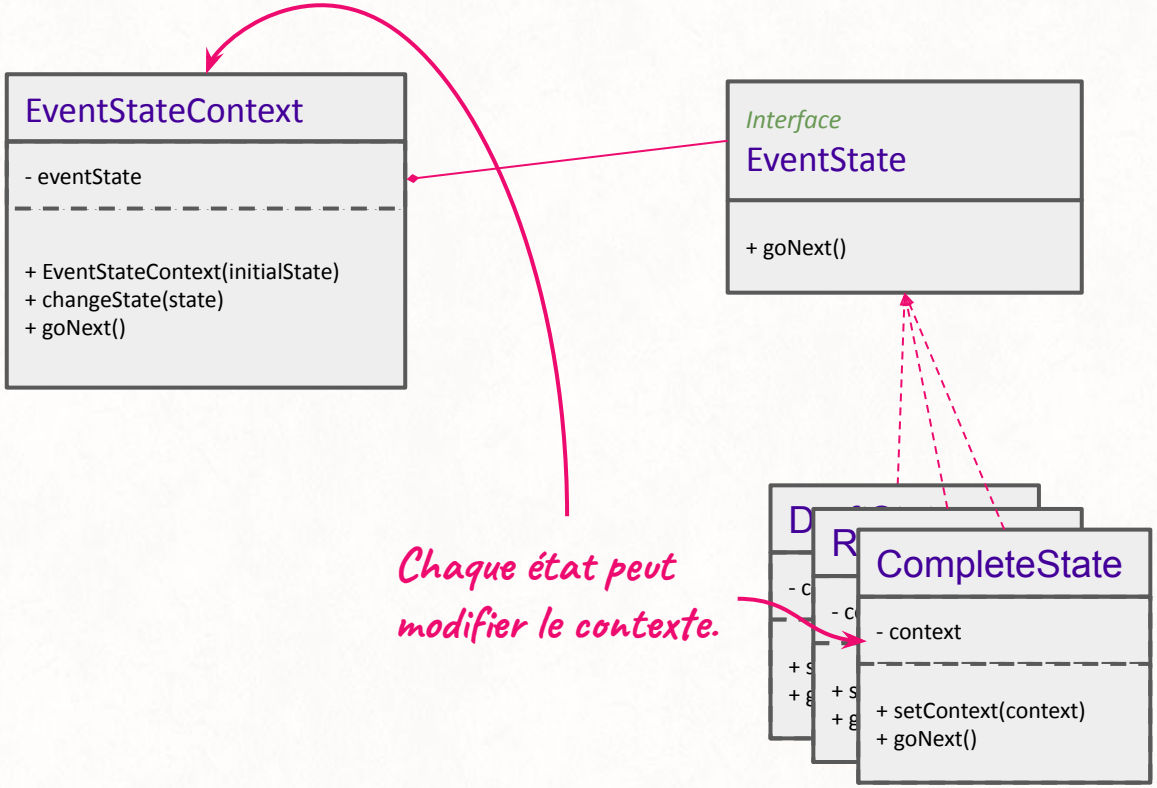
Le state pattern **autorise** les états à **changer directement le contexte**.

Schématisation

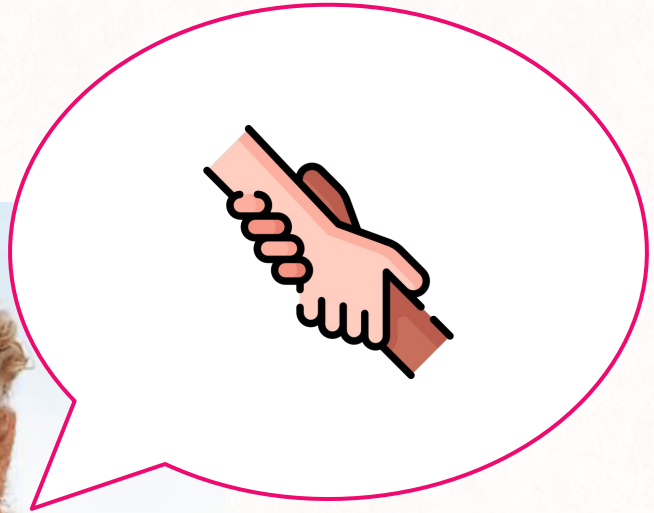


La stratégie est interchangeable.

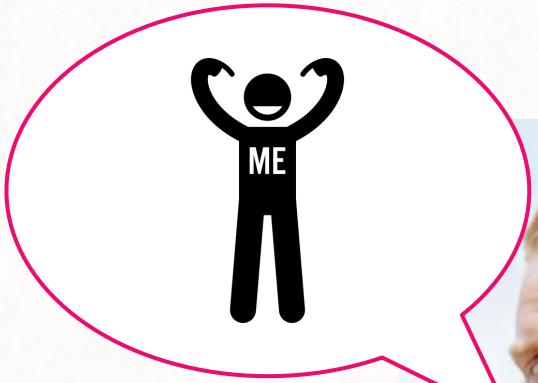
Schématisation



Les différencier



Les différencier



Stratégie

Etat

Les différencier

Je permets de créer des algorithmes interchangeables.

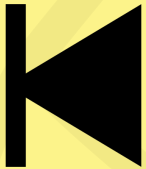
Stratégie

Etat

J'aide le contexte à contrôler un comportement.

Il permet à un objet de modifier son comportement lorsque son état interne change.

- ➔ Un **player média**
- ➔ Différents comportements selon l'état du média player



“

Ça, c'était le State pattern !

”

Quel pattern prendre entre les deux ?

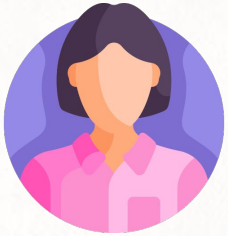
Deuxième besoin

Entretiens disponibles en fonction de l'organisateur

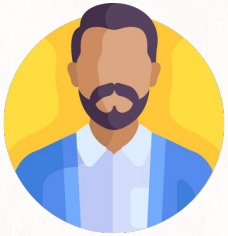
Entretiens disponibles en fonction de l’organisateur

Personas

Collaborateurs



Managers

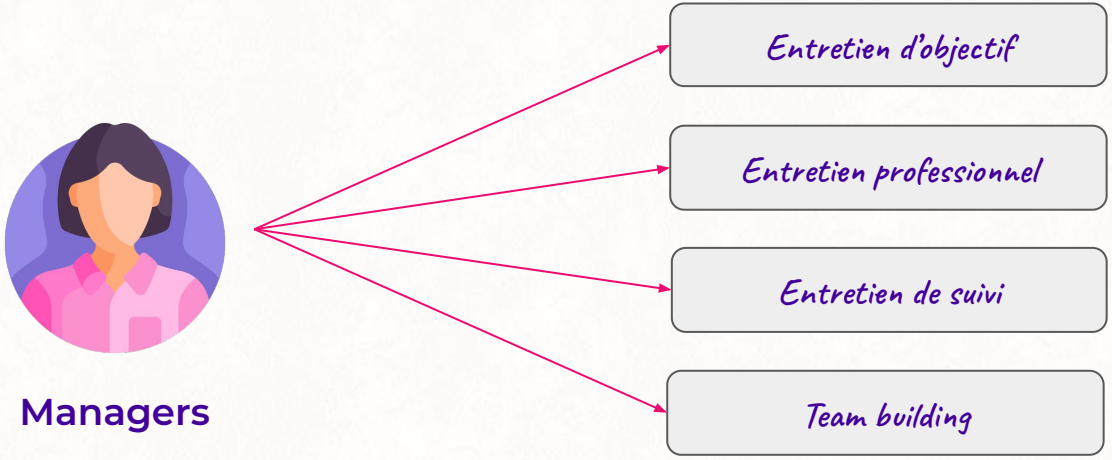


RH

Entretiens disponibles en fonction de l'organisateur

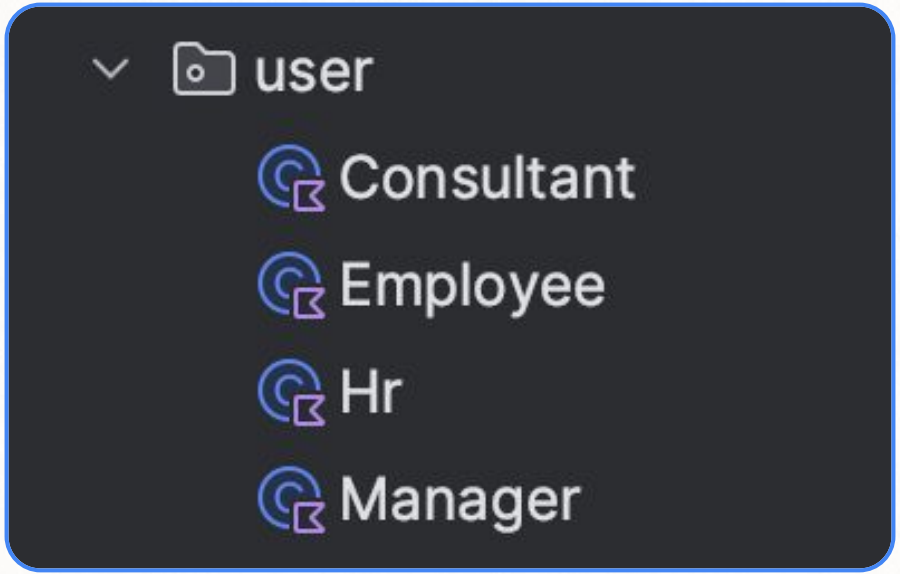
Pour un manager

➔ Retourner en fonction d'une classe une liste spécifique d'événements



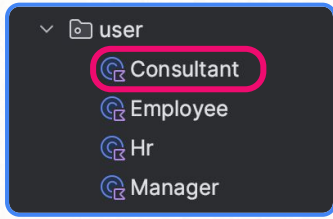
Entretiens disponibles en fonction de l’organisateur

Les classes présentes



Entretiens disponibles en fonction de l'organisateur

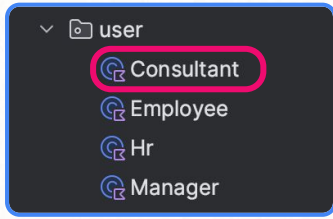
Classe consultant




```
class Consultant : Employee {

    fun signDocument(event: Event, signatureDocumentId: String) {
        // ...
    }

    fun addFormation(formation: Event) {
        //...
    }
}
```

Entretiens disponibles en fonction de l'organisateur

Classe consultant

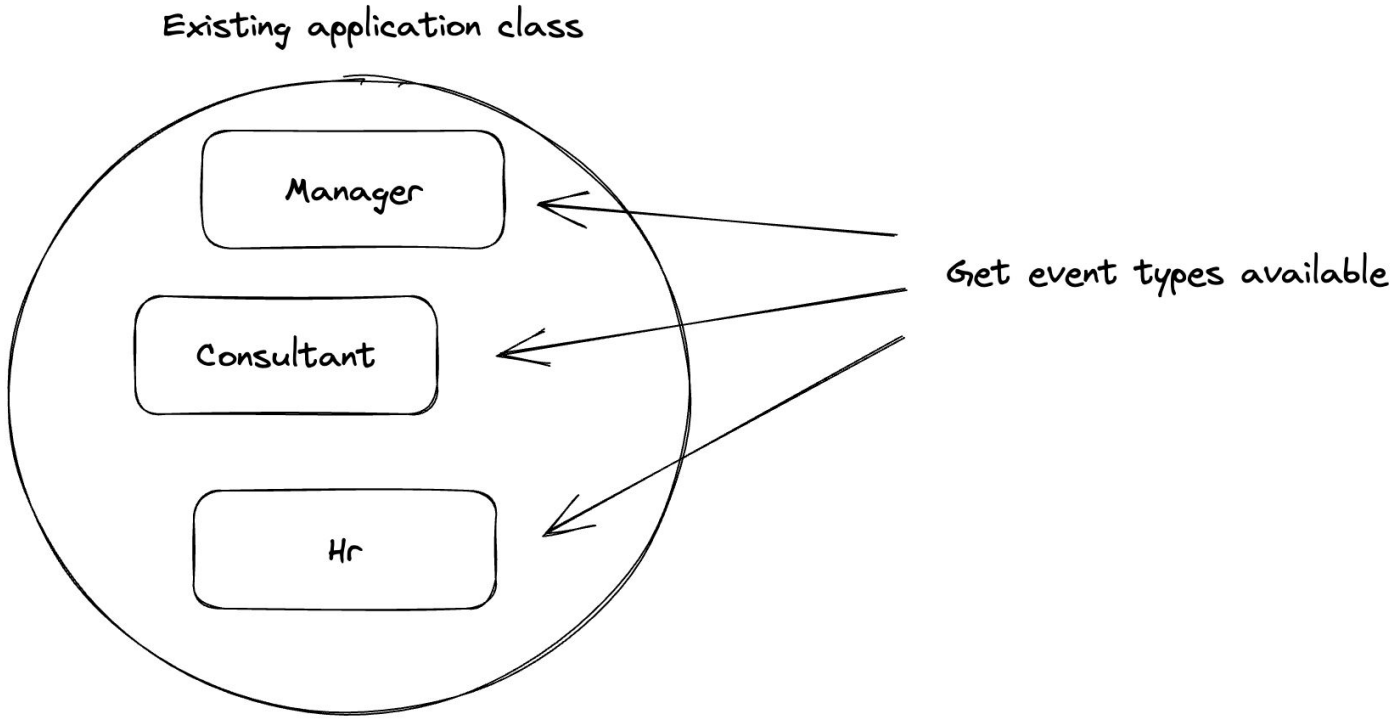



```
class Consultant : Employee {

    fun signDocument(event: Event, signatureDocumentId: String) {
        // ...
    }

    fun addFormation(formation: Event) {
        //...
    }
}
```

Validé, Testé, Qualifié

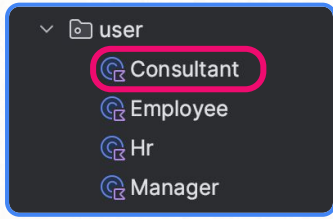
Entretiens disponibles en fonction de l'organisateur

Implémentation dans chacune des classes



Entretiens disponibles en fonction de l'organisateur

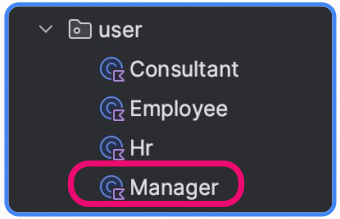
Classe consultant




```
class Consultant : Employee {
    fun getEventTypes(): Set<EventType> {
        return EnumSet.of(EventType.TRAINING_COURSE)
    }

    fun signDocument(event: Event, signatureDocumentId: String) {
        // ...
    }
    fun addFormation(formation: Event) {
        //...
    }
}
```

Entretiens disponibles en fonction de l'organisateur

Classe manager




```
class Manager(private val eventService: EventService) : Employee {
    fun getEventTypes(userId: String): Set<EventType> {
        // Fetch managed user
        // Extract user type
        // Get event type
        return eventTypes
    }

    fun signDocument(event: Event, signatureDocumentId: String) {
        // ...
    }
    fun addConsultantToTeamBuilding(teamBuilding: Event, participants:
List<Participant>){
        //...
    }
}
```

On ajoute un service

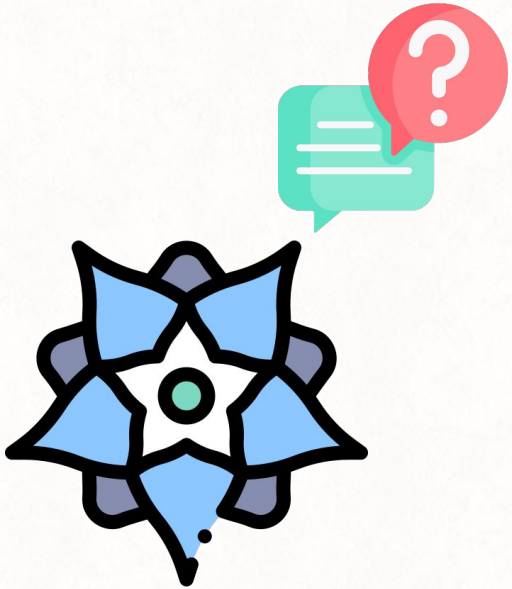
Les problèmes

Fragilise l'existant

Copier/coller dans tout notre arbre de classes

Entretiens disponibles en fonction de l'organisateur

Quel pattern ?



Le pattern Visitor

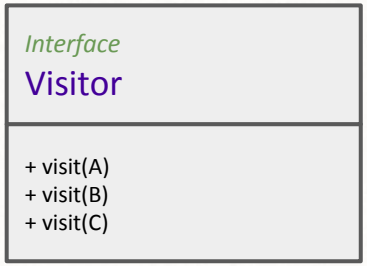
Carte d'identité

Visitor

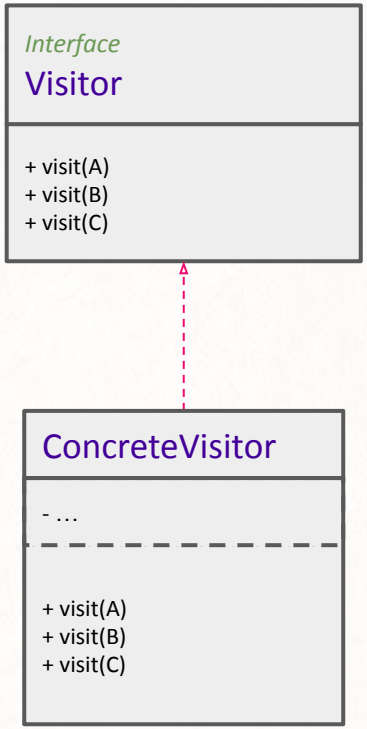
Famille : pattern **comportemental**

Utilité : Il permet de réaliser des opérations sur une structure d'objets complexes et de nettoyer la logique métier du code auxiliaire.

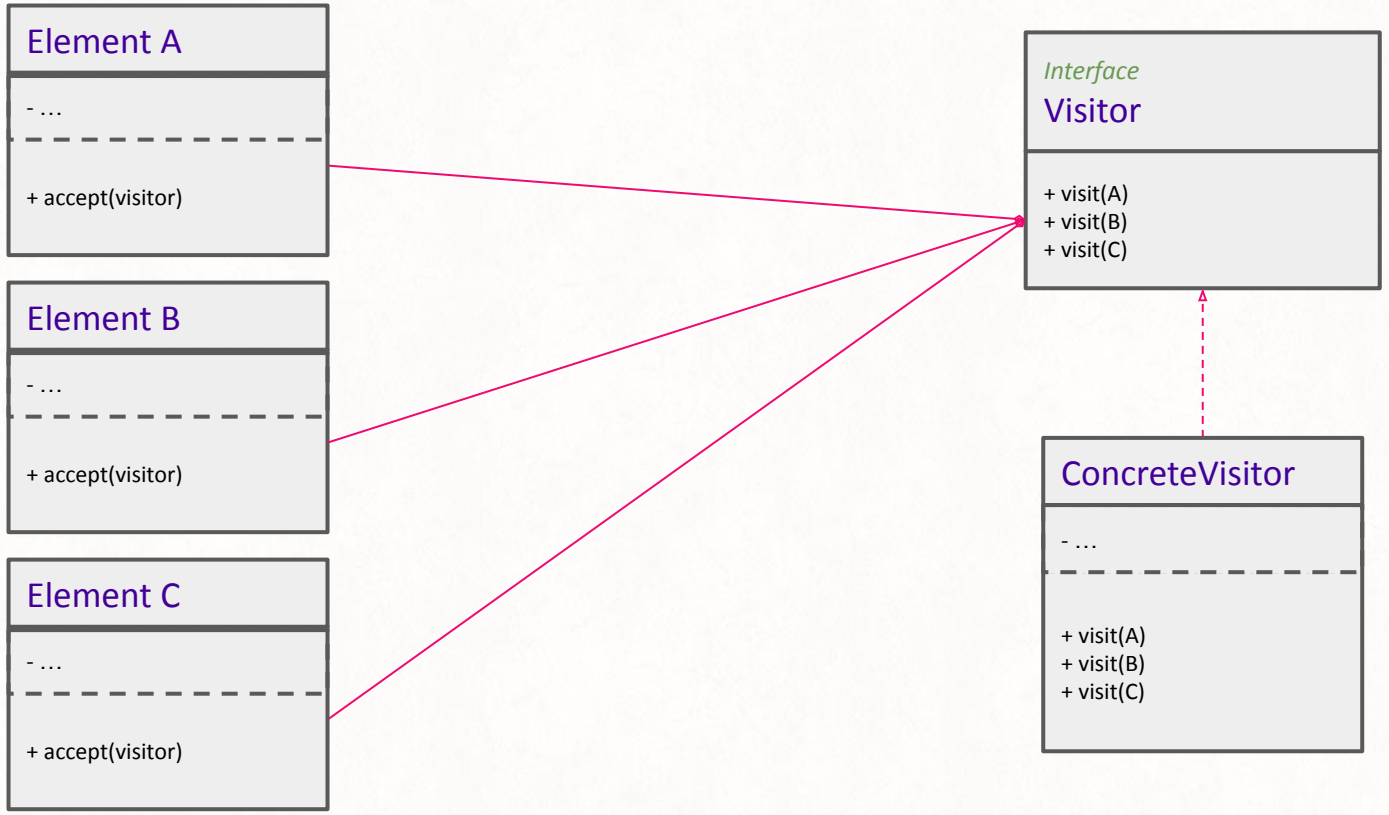
Schématisation



Schématisation



Schématisation



Créer la classe visiteur


```
interface Visitor {
    fun visit(manager: Manager): Set<EventType>
    fun visit(hr: Hr): Set<EventType>
    fun visit(consultant: Consultant): Set<EventType>
}
```

Une méthode par type d'employé

Le visitor pattern

Ajouter la méthode qui accepte le visitor

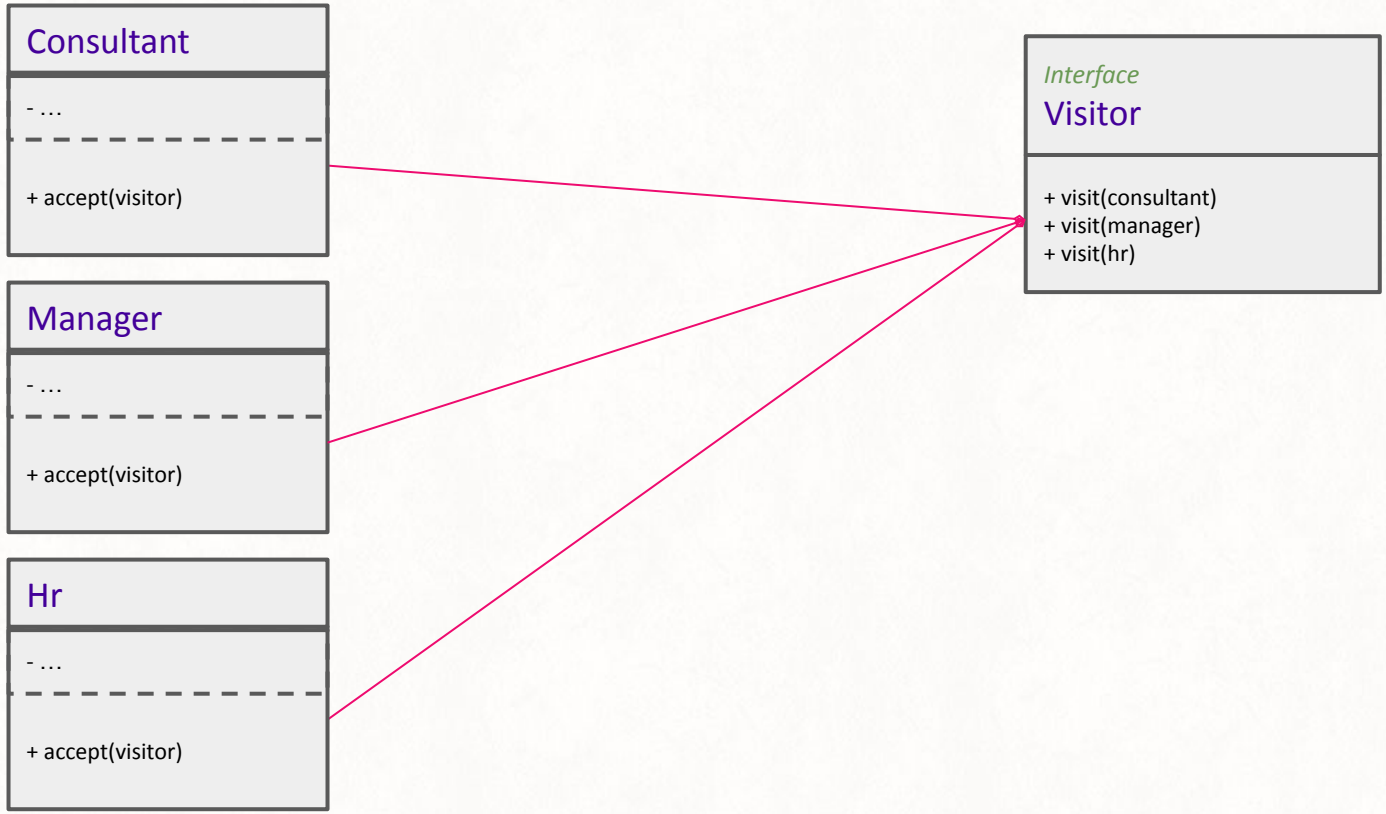
```
interface Employee {
    fun accept(v: Visitor): Set<EventType>
}
```

```
class Consultant : Employee {
    override fun accept(v: Visitor): Set<EventType> {
        return v.visit(this)
    }
    // ...
}
```

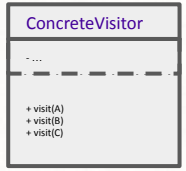
```
class Manager : Employee {
    override fun accept(v: Visitor): Set<EventType> {
        return v.visit(this)
    }
    // ...
}
```

```
class Hr : Employee {
    override fun accept(v: Visitor): Set<EventType> {
        return v.visit(this)
    }
    // ...
}
```

Schématisation

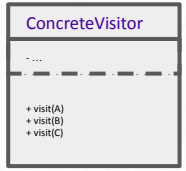


Implémenter les méthodes




```
@Component
class EventTypeByEmployeeType (private val eventService: EventService) : Visitor
{
    fun computeEventTypesFor (employee: Employee): Set<EventType> {
        return employee.accept(this)
    }
    override fun visit(manager: Manager): Set<EventType> {
        // Fetch managed user
        // Extract user type
        // Get event type
        return eventTypes
    }
    override fun visit(hr: Hr): Set<EventType> {
        return EnumSet.of(EventType.OTHER_HR_EVENT, EventType.TRAINING_COURSE)
    }
     // other visit methods
```

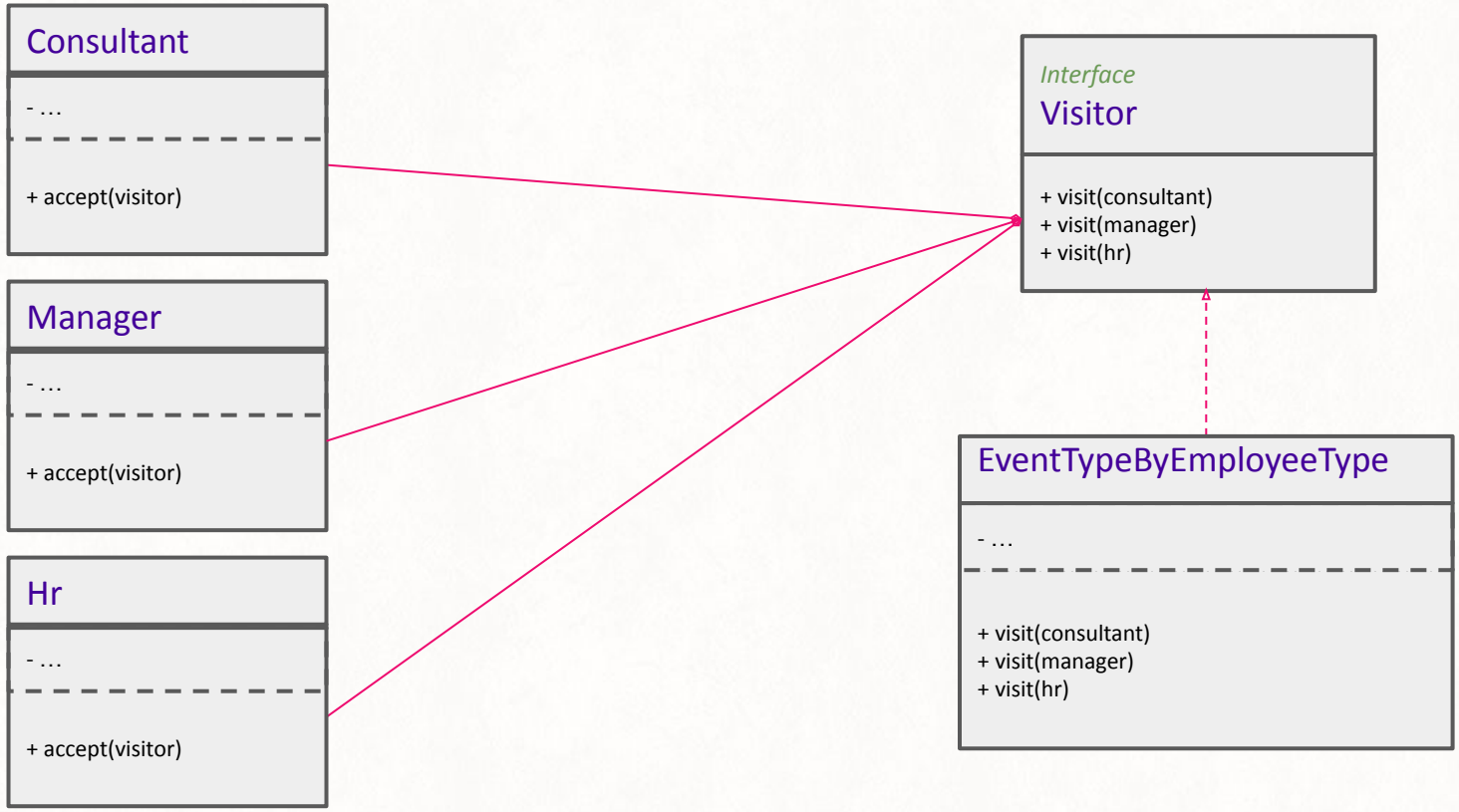
Implémenter les méthodes



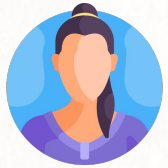

```
@Component
class EventTypeByEmployeeType (private val eventService: EventService) : Visitor
{
    fun computeEventTypesFor (employee: Employee): Set<EventType> {
        return employee.accept(this)
    }
    override fun visit(manager: Manager): Set<EventType> {
        // Fetch managed user
        // Extract user type
        // Get event type
        return eventTypes
    }
    override fun visit(hr: Hr): Set<EventType> {
        return EnumSet.of(EventType.OTHER_HR_EVENT, EventType.TRAINING_COURSE)
    }
     // other visit methods
```

Une implémentation par type

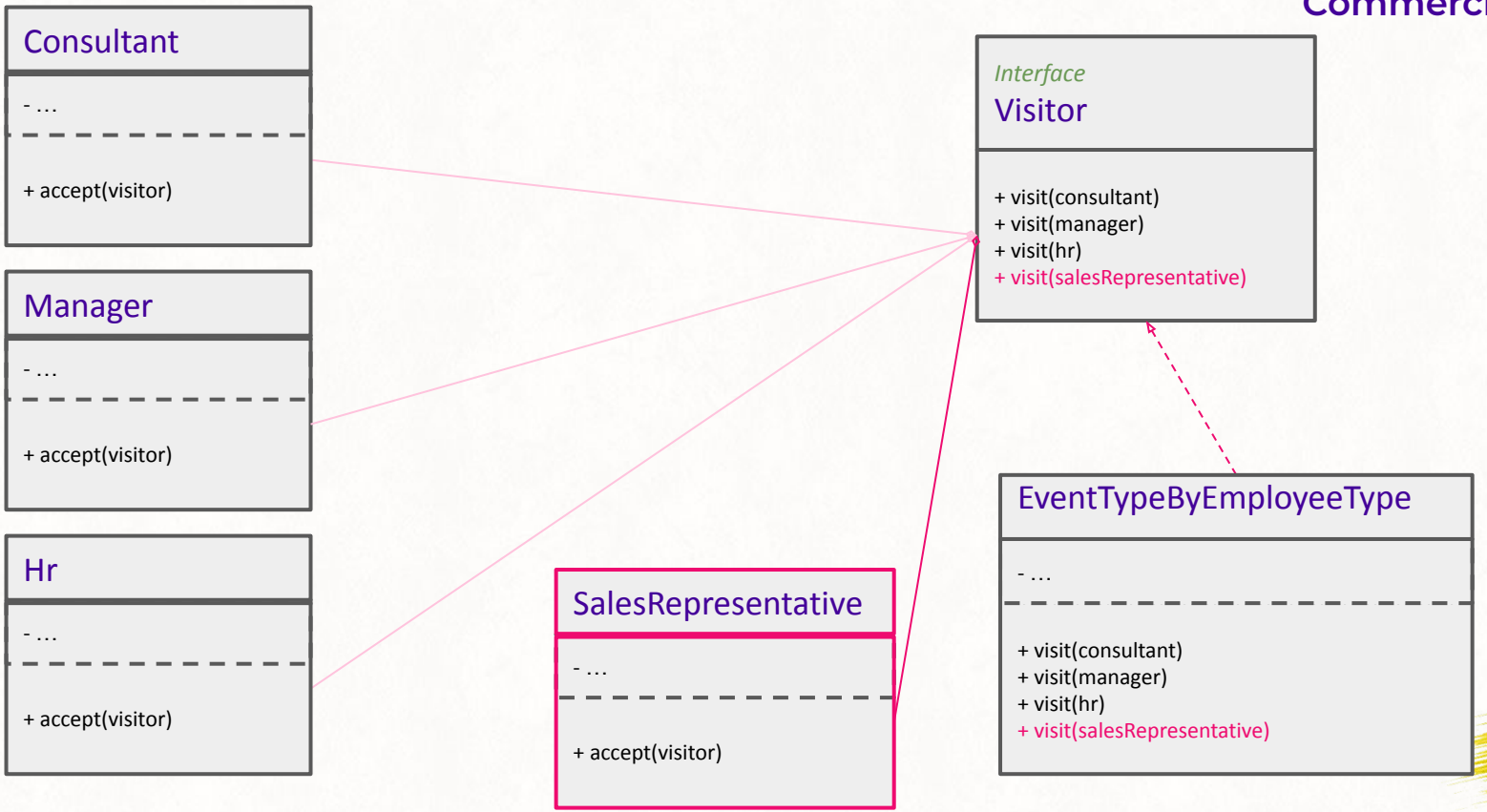
Schématisation



Schématisation



Commerciaux



On a des commerciaux aussi

Je rajoute une nouvelle classe

```
class SalesRepresentative : Employee {
    override fun accept(v: Visitor): Set<EventType> {
        return v.visit(this)
    }
    // ...
}
```

Une méthode pour implémenter le retour pour un commercial

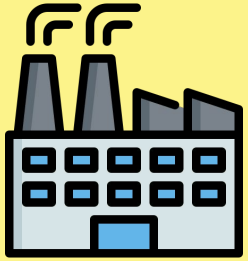
```
class EventTypeByEmployeeType : Visitor {
    // ...
    override fun visit(sr:SalesRepresentative): Set<EventType> {
        return EnumSet.of(
            EventType.LUNCH_WITH_CLIENT,
            EventType.INTERVIEW_WITH_CLIENT,
        )
    }
}
```

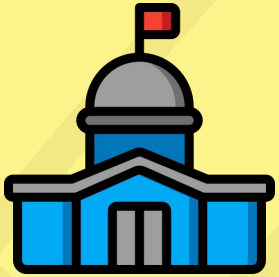
Il permet de réaliser des opérations sur une structure d'objets complexes et de nettoyer la logique métier du code auxiliaire.

Traitement en fonction de la classe

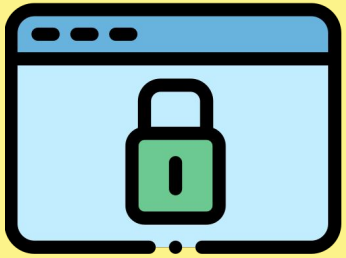
Open/Closed Principe

➔ Un assureur qui vend des assurances en faisant du porte à porte.





➔ Des exemples de cas concrets :



Ensemble de classes fermées

Parser JSON

“

Ça, c’était le Visitor pattern !

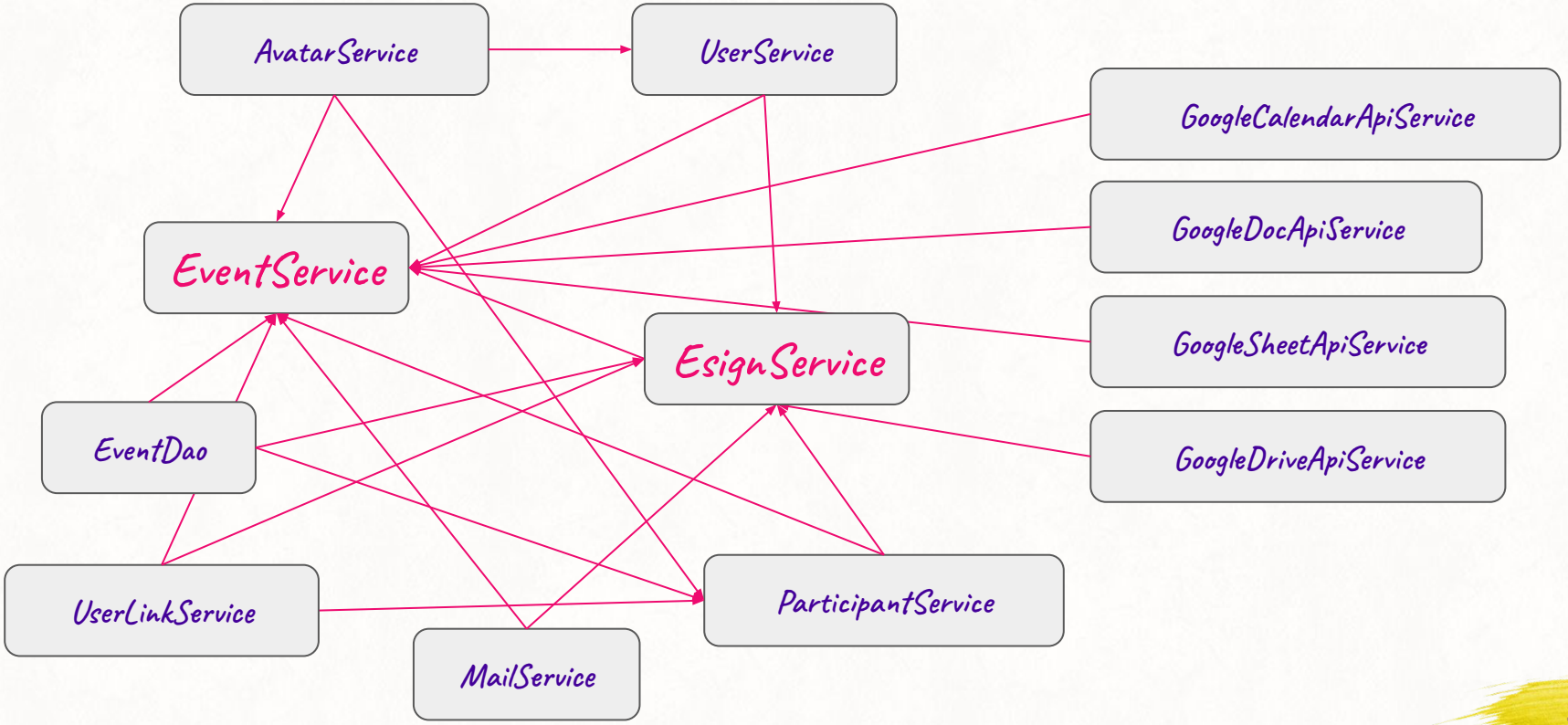
”

Refacto time

Notre problème de couplage démentiel

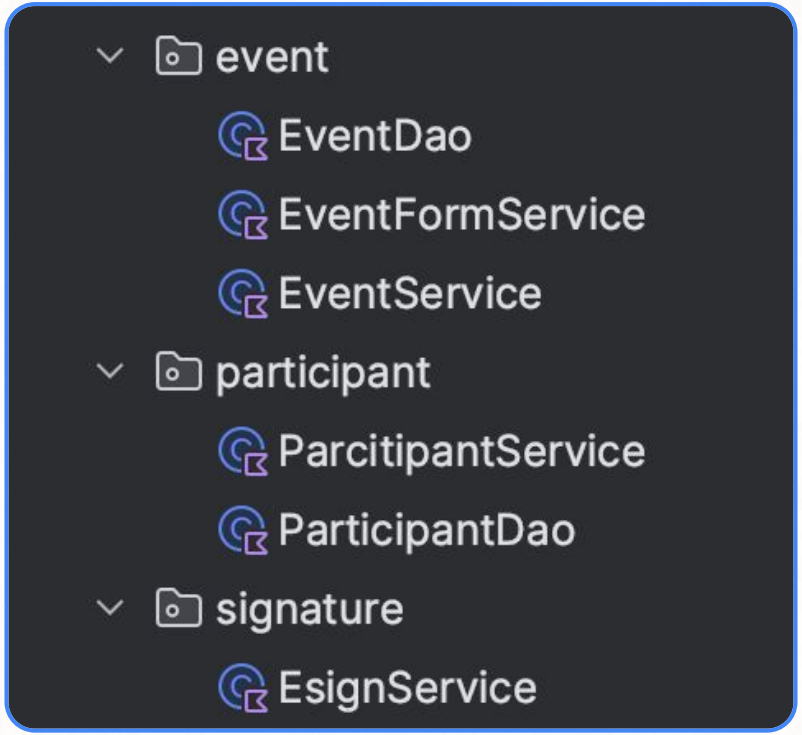
Notre problème de couplage démentiel

Nos classes ...

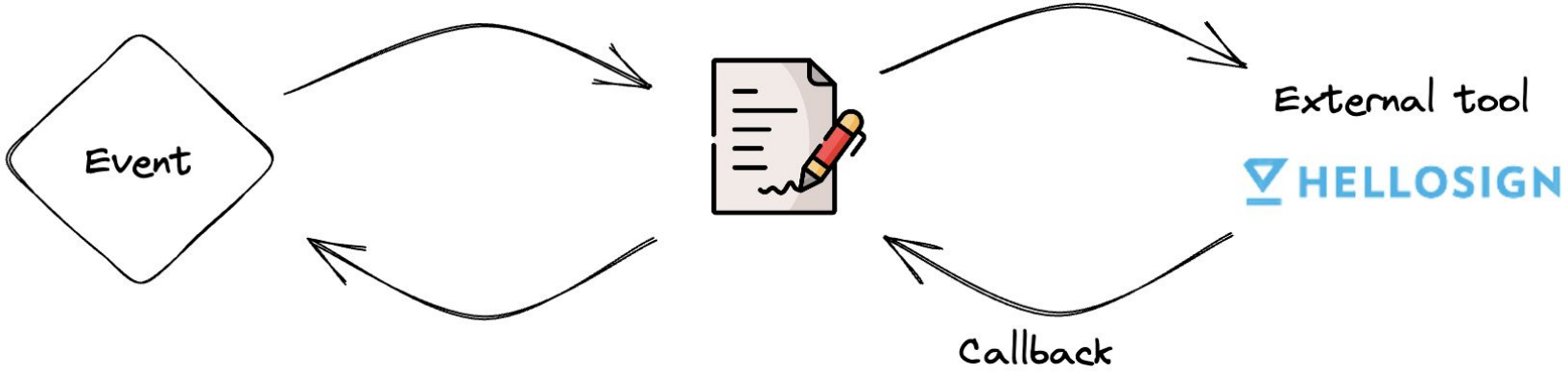


Notre problème de couplage démentiel

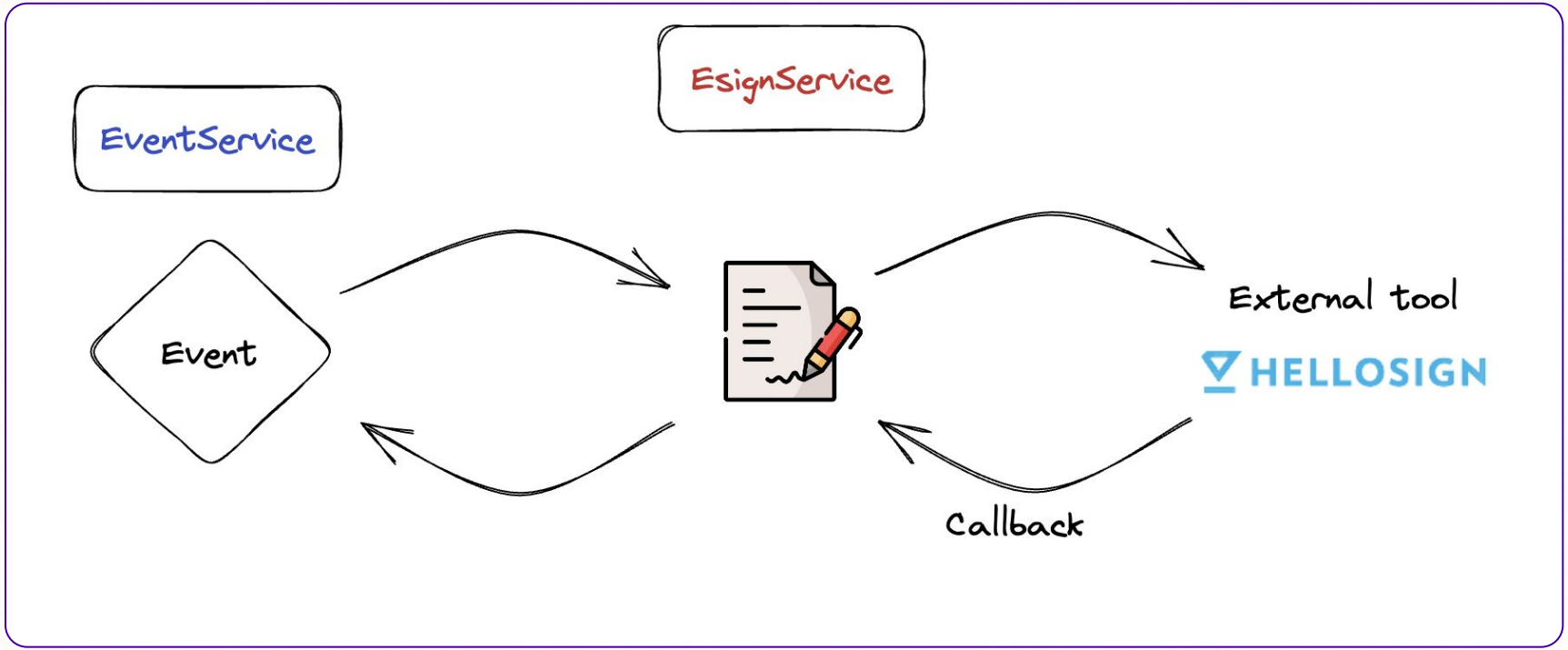
Les classes présentes



Le processus de signature



Les dépendances



On a une dépendance cyclique et nos services sont fortement couplés.

Recommencer une signature

EventService

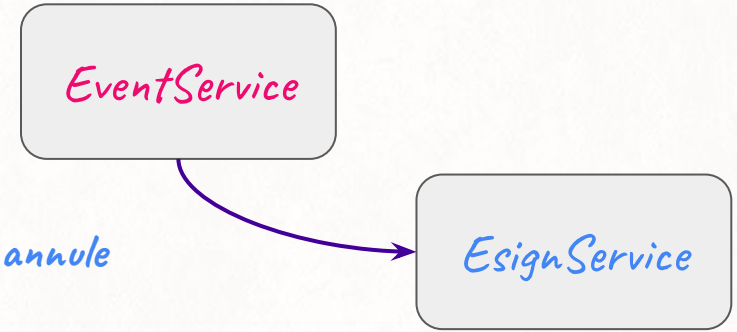
Récupérer l'événement

EsignService

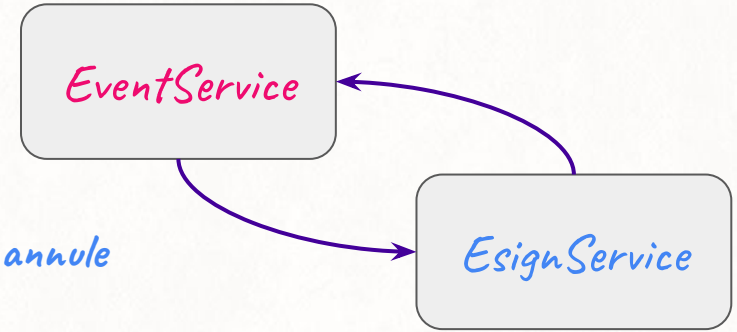
Recommencer une signature

Récupérer l'événement

Si processus de signature en cours -> on annule



Recommencer une signature

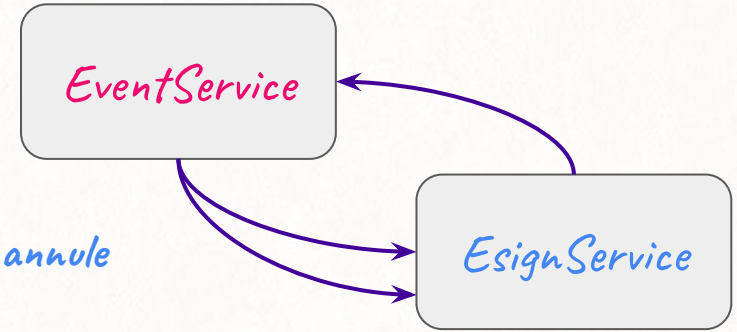


Récupérer l'événement

Si processus de signature en cours -> on annule

Créer le document à signer

Recommencer une signature



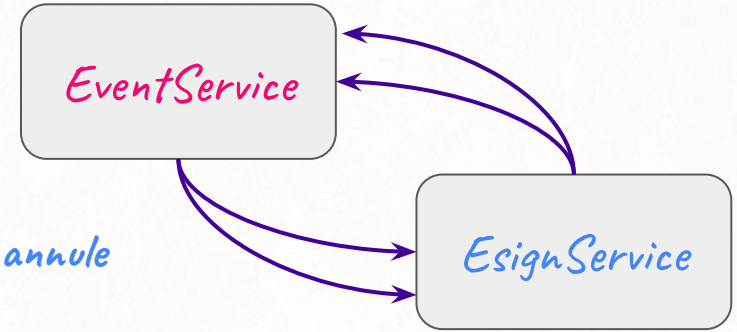
Récupérer l'événement

Si processus de signature en cours -> on annule

Créer le document à signer

Initialiser la nouvelle signature sur helloSign avec le document

Recommencer une signature



Récupérer l'événement

Si processus de signature en cours -> on annule

Créer le document à signer

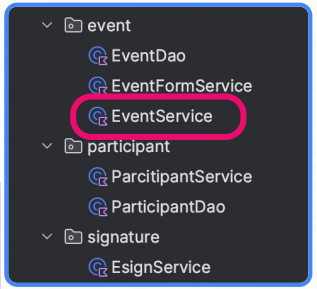
Initialiser la nouvelle signature sur helloSign avec le document

Mettre à jour l'event avec la nouvelle ID de signature

Notre problème de couplage démentiel

Le code actuel

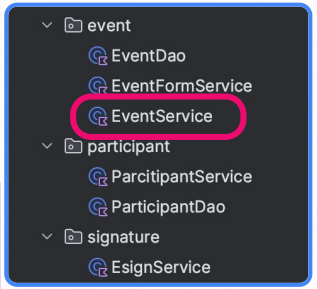
```
    @Transactional
    override fun restartEsignProcess(eventId: UUID): Event {
        val event = getById(eventId)

        esignService.cancelSignatureRequest(event.signatureRequestId)
        val filePath = eventFormService.importFromEventDriveToLocalPdf(event)

        esignService.initEsignProcess(event, filePath)

        event.eventState = SIGNATURE
        event.document = eventFormService.saveDocumentToDrive(event,
event.document)
        return event
    }
```



Notre problème de couplage démentiel

Le code actuel

```
@Transactional
override fun restartEsignProcess(eventId: UUID): Event {
    val event = getById(eventId)

    esignService.cancelSignatureRequest(event.signatureRequestId)
    val filePath = eventFormService.importFromEventDriveToLocalPdf(event)

    esignService.initEsignProcess(event, filePath)

    event.eventState = SIGNATURE
    event.document = eventFormService.saveDocumentToDrive(event,
event.document)
    return event
}
```

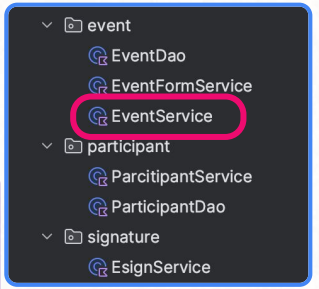


Appels à EsignService

Notre problème de couplage démentiel

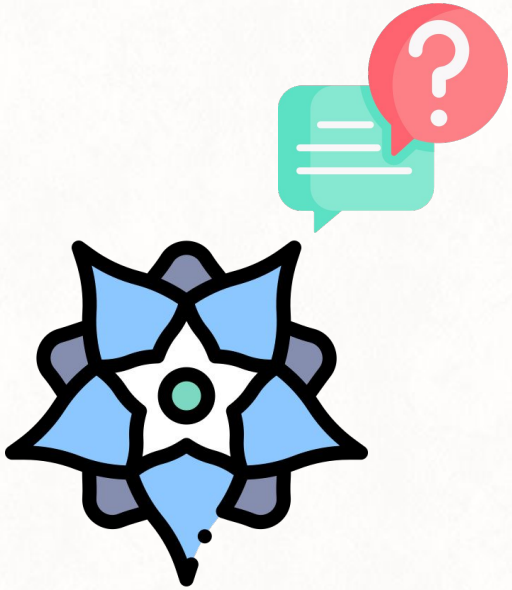
Le code actuel

```
@Transactional
override fun restartEsignProcess(eventId: UUID): Event {
    val event = getById(eventId)

    esignService.cancelSignatureRequest(event.signatureRequestId)
    val filePath = eventFormService.importFromEventDriveToLocalPdf(event)

    esignService.initEsignProcess(event, filePath)

    event.eventState = SIGNATURE
    event.document = eventFormService.saveDocumentToDrive(event,
event.document)
    return event
}
```

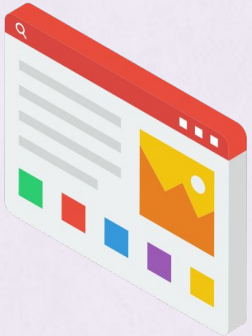


Une transaction avec des appels à un service externe

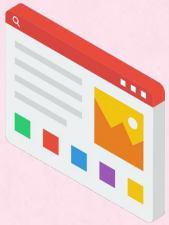
Quel pattern ?



Le Facade pattern



Carte d'identité

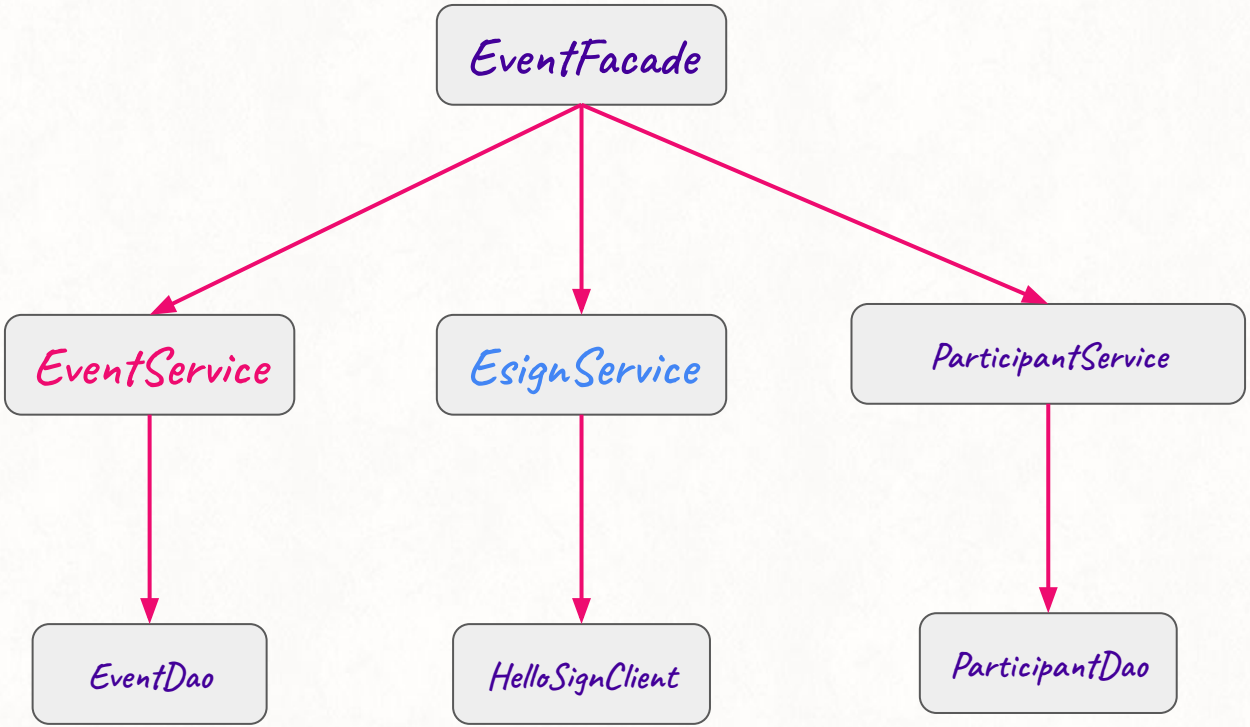


FAÇADE

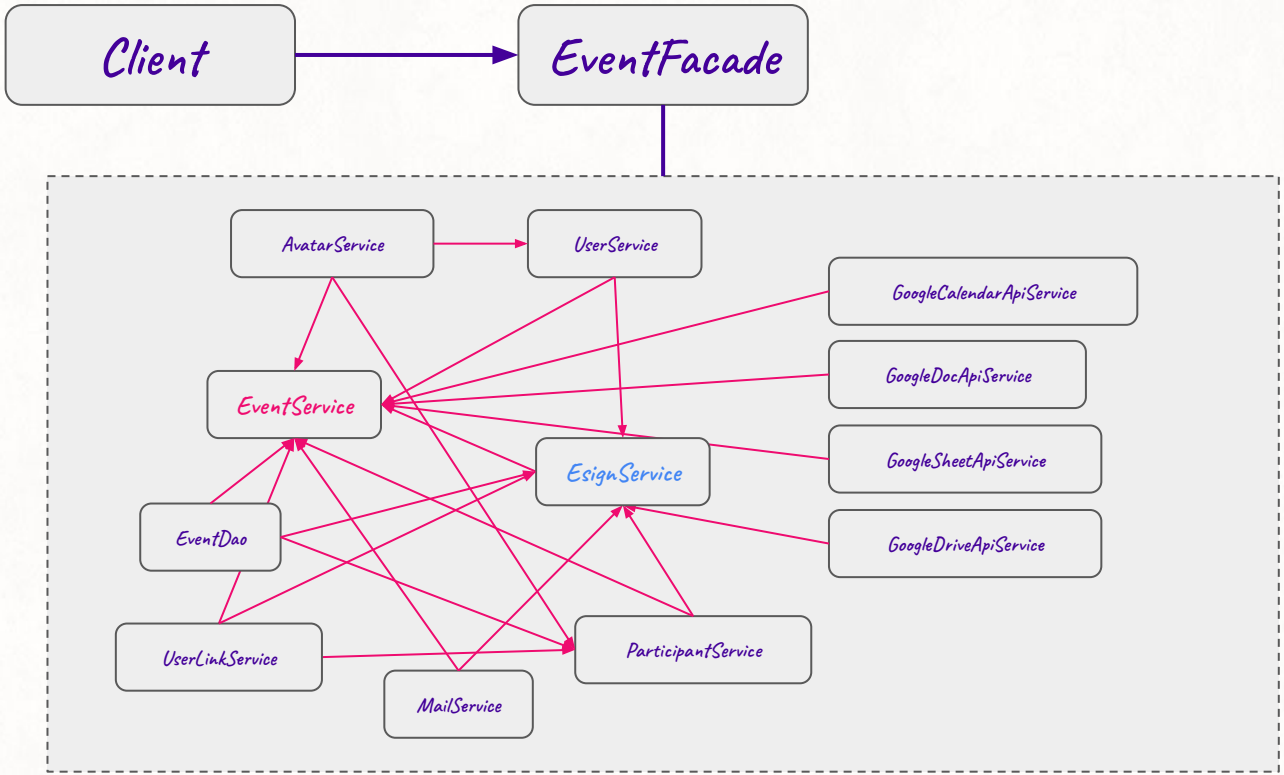
Famille : pattern **structurel**

Utilité : Il permet de fournir une interface simplifiée d'un ensemble plus complexe.

Nouvelle organisation simplifiée avec la façade

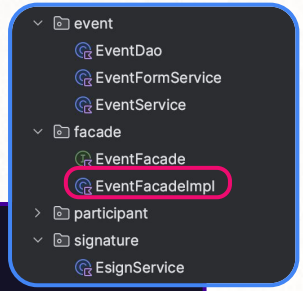


Façade de notre système pour les événements



Le facade pattern

EventFacade

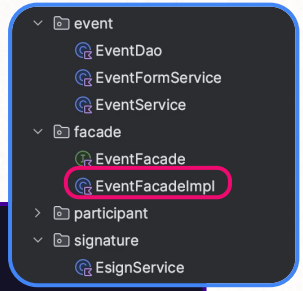



```
    override fun restartSignature(eventId: UUID): EventDto {
        val event = eventService.getById(eventId)
        esignService.cancelSignatureRequest(event.signatureRequestId)

        val document = eventFormService.generatePdfForSignature(event)
        val esignRequestId = esignService.initEsignProcess(event.participants,
document)

        eventService.moveToSignature(esignRequestId, document)
        return event
    }
```

J'appelle mon EventService

Le façade pattern

EventFacade

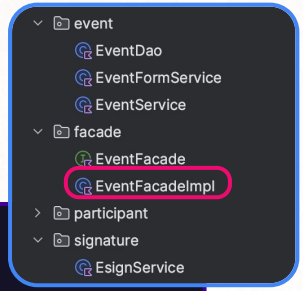



```
override fun restartSignature(eventId: UUID): EventDto {
    val event = eventService.getById(eventId)
    esignService.cancelSignatureRequest(event.signatureRequestId)

    val document = eventFormService.generatePdfForSignature(event)
    val esignRequestId = esignService.initEsignProcess(event.participants, document)

    eventService.moveToSignature(esignRequestId, document)
    return event
}
```

J'appelle mon EsignService

Le facade pattern

EventFacade

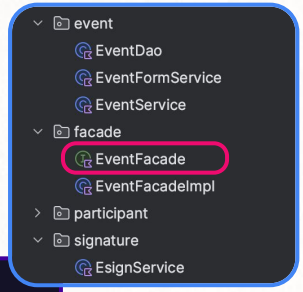



```
    override fun restartSignature(eventId: UUID): EventDto {
        val event = eventService.getById(eventId)
        esignService.cancelSignatureRequest(event.signatureRequestId)

        val document = eventFormService.generatePdfForSignature(event)
        val esignRequestId = esignService.initEsignProcess(event.participants,
document)

        eventService.moveToSignature(esignRequestId, document)
        return event
    }
```

Transaction déplacée sans service externe

Le facade pattern

Finalement




```kotlin
interface EventFacade {
    fun findEvent(id: UUID): EventResponseDto
    fun sendReminderToSubject(id: UUID)
    fun moveEventToNextState(id: UUID): EventResponseDto
    fun markEventAsDeleted(ids: List<UUID>)
    fun createEvent(eventRequest: EventRequestDto): EventResponseDto
    fun deleteEvents(ids: List<UUID>)
    fun restartSignature(eventId: UUID): EventResponseDto
    fun getEventTypes(): List<EventTypeDto>
}
```

Il permet de fournir une interface simplifiée d'un ensemble plus complexe

Ce qu'il faut retenir

Une classe

=

Une responsabilité

Moins de couplage

Moins de spaghetti

➔ Vous appelez le service d'Orange pour **résilier votre abonnement box**

- Service technique
- Service financier
- Service logistique

“

Ça, c’était le **Facade** pattern !

”

La progression en conception

Les étapes

Déesse du Design

Et après

Design Patterns

Principes de Conception

Dev

Et après

Évoluer en conception

Débutant

Je découvre les design patterns c'est compliqué mais ça a l'air cool.

Débutant

Je découvre les design patterns c'est compliqué mais ça a l'air cool.

Je vais en mettre de partout !

Débutant

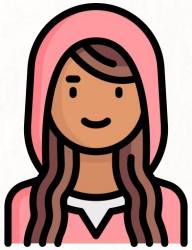
Je vais en mettre de partout !

Je découvre les design patterns c'est compliqué mais ça a l'air cool.

Plus j'utilise les patterns, plus je suis fort.

Intermédiaire

Je commence à voir où j'ai besoin d'un pattern.

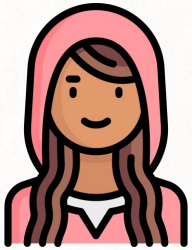


Évoluer en conception

Intermédiaire

J'essaie toujours de temps à autres de caser un rond dans un carré.

Je commence à voir où j'ai besoin d'un pattern.

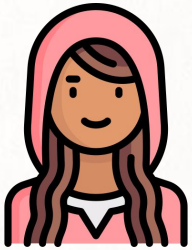


Évoluer en conception

Intermédiaire

J'essaie toujours de temps à autres de caser un rond dans un carré.

Je commence à voir où j'ai besoin d'un pattern.



Je peux adapter le pattern à mon problème.

Maître Pattern sur son arbre perché

Maître Pattern sur son arbre perché

Je réfléchis en termes de principes objets et de compromis.

Je cherche la solution simple à mon problème.

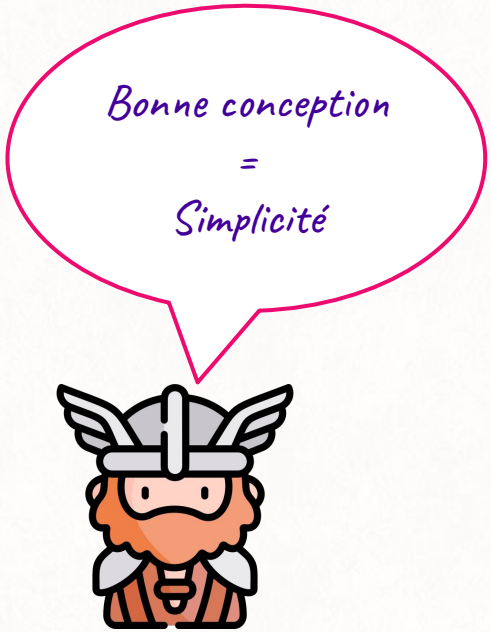
Maître Pattern sur son arbre perché

Je réfléchis en termes de principes objets et de compromis.

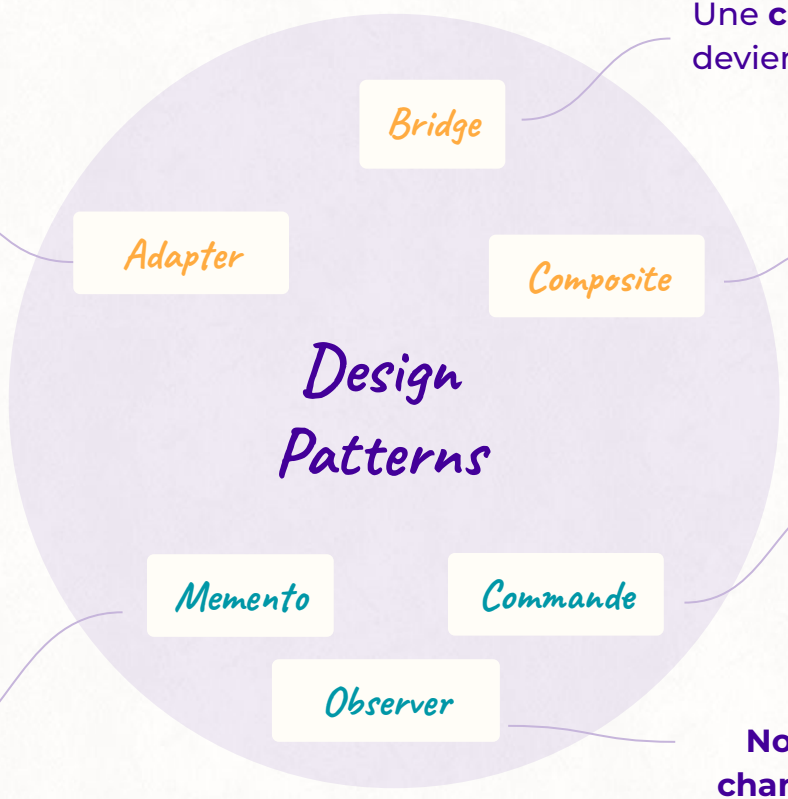
Je cherche la solution simple à mon problème.

Je sais quel pattern résout quel problème.

Les différents niveaux



Les autres patterns



Une **classe monolithique** qui devient **trop grande**

Architecture hexagonale

Une structure d'objets qui ressemble à **une arborescence**

Une **action** transformé en un **objet + détails** de l'action

Rétablir un état d'un objet et **contourner l'encapsulation**

Notifier des changements / Event-driven

“

Les anti-patterns vous montrent comment aller d’un problème à une mauvaise solution.

”

Bref ...

En résumé

En résumé

Les points à retenir

Conception et designs patterns

1. C’est un **langage universel**

En résumé

Les points à retenir

Conception et designs patterns

1. C'est un **langage universel**

2. **Refacto** = temps privilégié pour les design patterns

En résumé

Les points à retenir

Conception et designs patterns

1. C'est un **langage universel**
2. **Refacto** = temps privilégié pour les design patterns
3. Comprenez et **associez des designs patterns** à des problématiques.

En résumé

Les points à retenir

Conception et designs patterns

1. C'est un **langage universel**
2. **Refacto** = temps privilégié pour les design patterns
3. Comprenez et **associez des designs patterns** à des problématiques.
4. Restez le plus simple possible (**KISS**)

En résumé

Les points à retenir

Conception et designs patterns

1. C'est un **langage universel**
2. **Refacto** = temps privilégié pour les design patterns
3. Comprenez et **associez des designs patterns** à des problématiques.
4. Restez le plus simple possible (**KISS**)
5. **Ne cherchez pas à caser** les design patterns partout. Ce n'est pas un **silver bullet**.

En résumé

Les points à retenir

Conception et designs patterns

1. C'est un **langage universel**
2. **Refacto** = temps privilégié pour les design patterns
3. Comprenez et **associez des designs patterns** à des problématiques.
4. Restez le plus simple possible (**KISS**)
5. **Ne cherchez pas à caser** les design patterns partout. Ce n'est pas un **silver bullet**.
6. **Enlevez des patterns** si ça complexifie plus que ça n'aide à résoudre le problème.

En résumé

Les incontournables

Le site
Refactoring Guru

Tous les design patterns expliqués avec des exemples.

Head First :
Design patterns

Allez le lire si vous voulez en apprendre plus sur les design patterns.

Suivre les principes **SOLID**
et du **clean code**

KISS / YAGNI / DRY ...

En résumé

Les étapes

Déesse du Design

Et après

Design Patterns

Principes de Conception

Dev

Merci Beaucoup

Questions ?

takima
mastering technology

Mathilde Lorrain
@mathildelorrain

Benjamin Yvernault
@BenYvernault