



Introduction aux design patterns



Programme détaillé ou sommaire

Quelques rappels Objet

Introduction aux GRASP

Introduction aux Design Patterns

Patterns architecturaux

Patterns créationnels

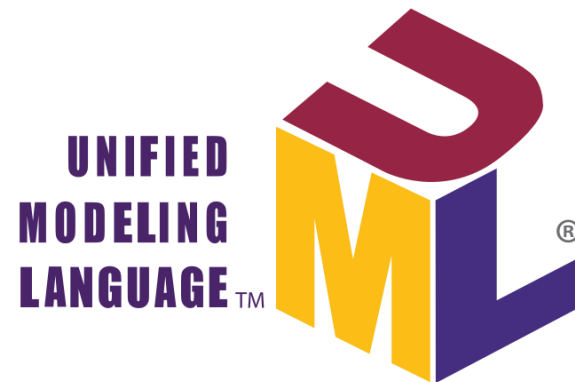
Patterns structuraux

Patterns comportementaux

Découpage en couches et bonnes pratiques

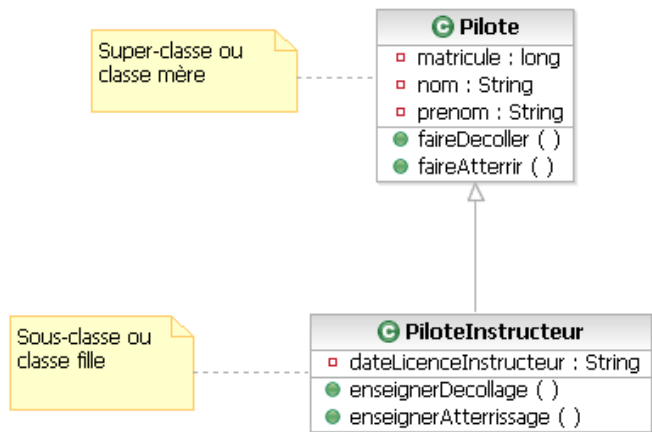
Chapitre 1

Rappels objets /



Héritage

- ❑ Mécanisme par lequel un objet hérite des attributs et méthodes déclarés dans sa classe mère.
- ❑ Par héritage la classe **PiloteInstructeur** a 4 attributs et 4 méthodes.



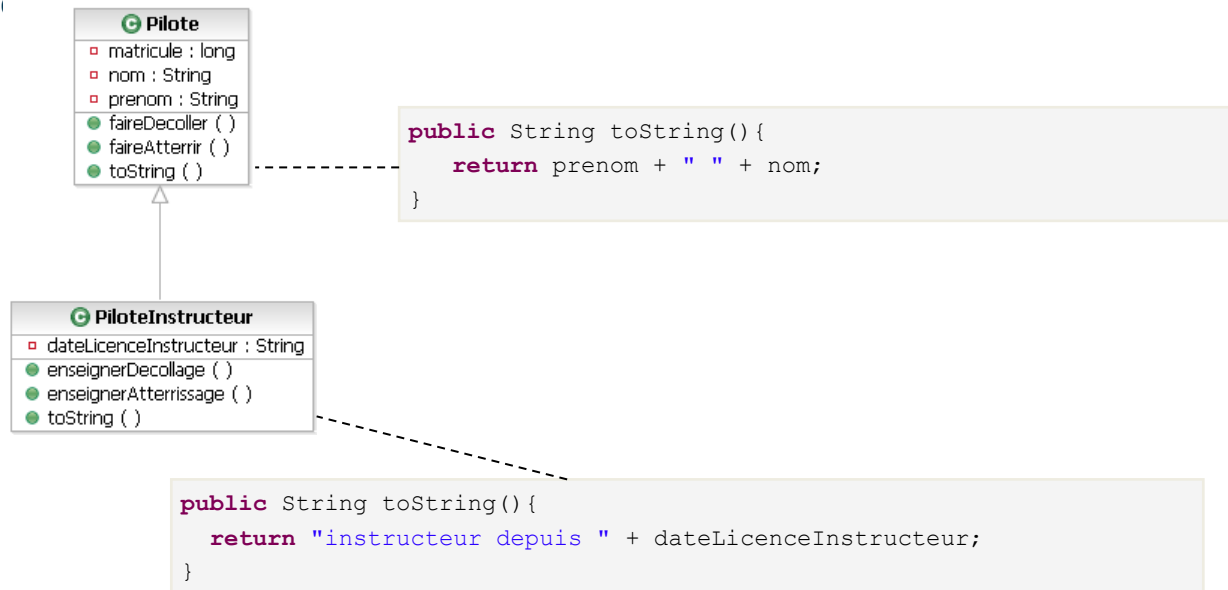
```
public class PiloteInstructeur extends Pilote {
    private String dateLicenceInstructeur;

    public void enseignerDecollage() {
        // code pour enseigner le décollage
    }

    public void enseignerAtterrissage() {
        // code pour enseigner le décollage
    }
}
```

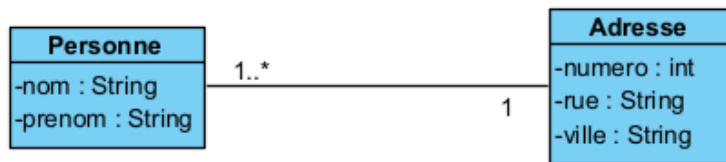
Redéfinition de méthodes

- ❑ Mécanisme qui consiste à « écraser » dans la classe fille la méthode déclarée dans la classe mère
- ❑ Le but est évidemment que la classe fille utilise sa propre méthode et non celle de la classe mère



Associations

- ❑ Une association est une dépendance forte entre 2 classes.
- ❑ Idée de "possession" (agrégation ou composition) ou de relation étroite.
 - ✓ Une **personne** qui possède une **adresse**
 - ✓ Un **client** qui possède plusieurs **comptes**
- ❑ Il y a association dès lors qu'une propriété n'est pas atomique/primitive



```
public class Personne {  
  
    private String nom;  
    private String prenom;  
    private Adresse adresse;  
}
```

```
public class Adresse {  
  
    private int numero;  
    private String rue;  
    private String ville;  
}
```

Classe et méthodes abstraites

- ❑ C'est une classe qui ne peut pas être instanciée
- ❑ Une classe abstraite peut contenir des méthodes abstraites (sans corps)
- ❑ Les classes filles ont l'obligation de redéfinir les méthodes abstraites de la classe mère

```
public abstract class Collaborateur {  
    protected String nom;  
    protected String prenom;  
  
    public Collaborateur(String nom, String prenom) {  
        this.nom = nom;  
        this.prenom = prenom;  
    }  
  
    public abstract double calculerSalaire();  
}
```

```
public class Pilote extends Collaborateur {  
    private double salaire;  
  
    public Pilote(String nom, String prenom, double salaire) {  
        super(nom, prenom);  
        this.salaire = salaire;  
    }  
  
    @Override  
    public double calculerSalaire() {  
        return salaire;  
    }  
}
```


Polymorphisme de méthode

- ❑ Une méthode peut prendre plusieurs formes.
- ❑ La classe ci-dessous peut prendre en paramètres n'importe quelle instance de classe qui hérite de Collaborateur : Pilote, Hotesse, etc..



```
new Pilote()           new Hotesse()

public class GestionnairePaie {

    public void editerFeuilleSalaire(Collaborateur collaborateur) {

        double salaire = collaborateur.calculerSalaire();
        //TODO editer PDF
    }
}
```

Polymorphisme

❑ Exemple

```
public static void main(String[] args) {  
    GestionnairePaie paie = new GestionnairePaie();  
  
    Hotesse uneHotesse = new Hotesse();  
    paie.editerFeuilleSalaire(uneHotesse);  
  
    Pilote unPilote = new Pilote();  
    paie.editerAttestationSalaire(unPilote);  
}
```

```
public class GestionnairePaie {  
  
    public void editerFeuilleSalaire(Collaborateur collaborateur) {  
        double salaire = collaborateur.calculerSalaire();  
        //TODO editer PDF  
    }  
}
```

Interfaces

- ❑ Super-abstraction dans laquelle la plupart des méthodes sont abstraites hormis quelques méthodes (depuis Java 8)
- ❑ Une classe qui implémente une interface doit **redéfinir toutes les méthodes de l'interface**

```
public interface Pilotable {  
    public static final int ALTITUDE_SECURITE = 100;  
  
    public long getPuissance();  
    public void setPuissance(long puissanceCible);  
    public long getAltitude();  
    public void rentrerTrainAtterrissage();  
    public void sortirTrainAtterrissage();  
    public void inclinerVolets(long angle);  
    public void setPositionManche(long position);  
    public void virer(long angle);  
}
```

Interfaces

❑ Exemple

```
public class Avion implements Pilotable {  
    private Moteur moteur;  
    private CapteurAltitude capteurAltitude;  
  
    public long getPuissance() { return moteur.getPuissance(); }  
    public void setPuissance(long puissance) { moteur.setPuissance(puissance); }  
    public long getAltitude() { return capteurAltitude.getAltitude(); }  
    public void sortirTrainAtterrissage() { /* code */ }  
    public void rentrerTrainAtterrissage() { /* code */ }  
    public void inclinerVolets(long angle) { /* code */ }  
    public void setPositionManche(long position) { /* code */ }  
    public void virer(long angle) { /* code */ }  
}
```

Chapitre 2

Introduction aux GRASP

GRASP

General Responsibility Assignment Software

GRASP
Design Principles
in OOAD

General
Responsibility
Assignment
Software
Patterns



Classe et responsabilités

L'attribution des responsabilités aux différentes classes est une des clés de la programmation orientée objet.

Classe et responsabilités

Il est fondamental de se poser la question des responsabilités d'une classe :

- ❑ Une classe ne doit pas être un sac à méthodes qui fait tout et n'importe quoi
- ❑ Une classe représente un concept
- ❑ Il faut se poser les questions suivantes
 - « Qui fait quoi ? »
 - Comment les classes interagissent entre elles ? Ex: découpage en couches.

A éviter:

- ❑ Les classes fourre-tout
- ❑ Développer une méthode, ou du code, sans se poser la question de la classe d'accueil

Que sont les GRASPS

Un GRASP:

- ❑ General Responsibility Assignment Software Pattern.
- ❑ Pattern de responsabilités

Les GRASPS sont une source d'inspiration, un guide, pour vous aider à assigner au mieux les responsabilités à vos classes.

Exemple pour comprendre les GRASP

Que pensez-vous de cette classe ?

```
public class Application {  
  
    public static void main(String[] args) {  
        LocalDate dateDebut = toLocalDate("12/06/2022");  
    }  
  
    Public static LocalDate toLocalDate(String dateStr) {  
        SimpleDateFormat format = new SimpleDateFormat("dd/MM/yyyy");  
        return format.parse(dateStr);  
    }  
}
```

A propos de l'exemple précédent

Quelles sont les responsabilités d'une classe de type "Contrôleur" ?

- ☐ Être le point d'entrée, d'échange d'informations, entre Front et Back.
- ☐ Exécuter un C.U. (Cas d'Utilisation) et retourner un résultat
- ☐ Traiter les éventuelles exceptions et redirections

L'exemple précédent ajoute une responsabilité :

- ☐ Elle devient de facto une spécialiste des transformations String <-> LocalDate
- ☐ Dans ce cas chaque contrôleur devrait-il posséder cette méthode ?
- ☐ Comment font les autres classes qui ont besoin de convertir des String en LocalDate ?

A propos de l'exemple précédent

Si tous les développeurs ont ce genre de réflexes, le risque est:

- ❑ De voir fleurir des méthodes qui font ce genre d'opérations dans diverses classes
- ❑ D'avoir de la duplication de code
- ❑ Au final, avoir des classes hétérogènes

Avantage d'une classe spécialisée

En créant une classe spécialisée dans les transformations String <-> LocalDate:

- ❑ Tous les développeurs utilisent une classe unique spécialisée
- ❑ Les méthodes sont réutilisables
- ❑ La classe spécialisée peut être complétée avec de nouvelles méthodes
- ❑ On évite de la duplication de code et le risque de bug

```
public class DateUtils {  
  
    public static LocalDate toLocalDate(String dateStr) {  
        return LocalDate.parse(dateStr, DateTimeFormatter.ofPattern("dd/MM/yyyy"));  
    }  
}
```

Exemple 2 pour comprendre les GRASP

Que pensez-vous de ce code ?

```
public class GererAvionController {  
  
    public Avion creerAvion(@RequestBody Params params) {  
        String matricule = params.getParameter("matricule");  
        String compagnie = params.getParameter("compagnie");  
        String typeMoteur = params.getParameter("typeMoteur");  
  
        Avion avion = new Avion(matricule);  
        avion.setCompagnie(compagnie);  
        avion.setMoteur(new Moteur(typeMoteur));  
  
        // Suite traitement  
    }  
}
```

A propos de l'exemple précédent

Mauvaise pratique:

- ❑ Le contrôleur n'a pas pour responsabilité de savoir comment "construire" un objet en invoquant un constructeur puis en invoquant les différents setters.
- ❑ Est-ce que chaque contrôleur qui devra « créer » une instance d'Avion doit posséder ce même savoir ?

Bonnes pratiques:

- ❑ Il faut une classe spécialisée pour créer une instance d'Avion: Factory ou Builder.
- ❑ De plus, étant donné que le Moteur est associé à la classe Avion (l'avion possède un moteur), c'est à la classe Avion de créer le Moteur.

Exemple 2 pour comprendre les GRASP

Après modification

```
public class GererAvionController {  
  
    public Avion creerAvion(@RequestBody Params params) {  
        String matricule = params.getParameter("matricule");  
        String compagnie = params.getParameter("compagnie");  
        String typeMoteur = params.getParameter("typeMoteur");  
  
        Avion avion = AvionFactory.getInstance(matricule, compagnie, typeMoteur);  
    }  
}  
  
public class AvionFactory {  
  
    public static Avion getInstance(String matricule, String compagnie, String typeMoteur) {  
        Avion avion = new Avion(matricule, typeMoteur);  
        avion.setCompagnie(compagnie);  
        return avion;  
    }  
}
```


Les patterns de responsabilité: GRASPS

9 patterns de responsabilité:

- ☐ **Contrôleur**
- ☐ **Créateur**
- ☐ **Expert de l'information**
- ☐ **Forte cohésion**
- ☐ **Couplage faible**
- ☐ **Pure fabrication**
- ☐ **Protection des variations**
- ☐ **Indirection**
- ☐ **Polymorphisme**

Forte cohésion

Forte cohésion:

- ❑ Pour une classe donnée, les responsabilités doivent être cohérentes.
- ❑ La classe doit avoir un type de responsabilité donnée, ex: transformer des String en LocalDate et vice versa.
- ❑ Eviter la classe "Poireau" qui possède des dizaines de méthodes

Pure fabrication

- ❑ Afin d'avoir des classes cohérentes, on va forcément devoir créer des classes qu'on n'avait pas forcément imaginées lors des spécifications avec le client: Banque, Compte, Client, Operation, etc..
- ❑ On va devoir créer des classes pour un certain type de tâches ou de traitements.
- ❑ Exemple: transformer des String en LocalDate et vice versa.
- ❑ C'est ce type de classes qu'on appelle **Pure Fabrication** car elles ne font pas partie du domaine métier initial du client.
- ❑ Les classes pure fabrication permettent de laisser du « code propre » dans les classes métier.

Créateur

- ❑ Le pattern créateur se pose la question de savoir qui est responsable de l'instanciation d'une classe.
- ❑ Que pensez-vous de ce code ?

```
public class Service {  
    public void traitement() {  
  
        Individu individu = new Individu();  
  
        Jambe jambeDroite = new Jambe();  
        Jambe jambeGauche = new Jambe();  
  
        individu.setJambeGauche(jambeGauche);  
        individu.setJambeDroite(jambeDroite);  
    }  
}
```

Créateur

❑ L'exemple précédent ne respecte pas le pattern créateur qui dit ceci:

- La classe peut instancier une autre classe si:
 - Elle possède les informations pour le faire
 - Elle possède des instances de cette classe

```
public class Service {  
    public void traitement() {  
  
        Individu individu = new Individu();  
    }  
}
```

```
public class Individu {  
  
    private Jambe jambeGauche;  
    private Jambe jambeDroite;  
  
    public Individu() {  
        this.jambeDroite = new Jambe();  
        this.jambeGauche = new Jambe();  
    }  
}
```

Créateur

- ❑ Eventuellement on peut avoir recours à une classe spécialisée pour la construction d'un individu et qu'on appelle une Factory

```
public class Service {  
  
    private IndividuFactory factory = new IndividuFactory();  
  
    public void traitement() {  
        Individu individu = factory.getInstance();  
    }  
}
```

- ❑ **Problème:**

- On est obligé de créer une instance de **IndividuFactory**...
- C'est la raison d'être de frameworks comme Spring avec les Autowired.

Créateur

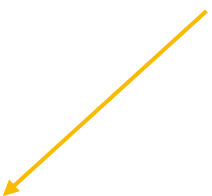
- ❑ On va supprimer les instantiations définitivement en utilisant un mécanisme d'injection de dépendances.
- ❑ Demander à un framework technique de créer des instances de classes purement techniques :
OK

```
public class Service {  
  
    @Autowired  
    private IndividuFactory factory;  
  
    public void traitement() {  
        Individu individu = factory.getInstance();  
    }  
}
```

Faible couplage

- ❑ L'idée est de supprimer au maximum le couplage entre classes concrètes.
- ❑ Par exemple dans l'exemple ci-dessous on a un couplage entre la classe Service et la classe IndividuFactory.
- ❑ Pour réduire ce couplage il est recommandé d'utiliser des interfaces.
- ❑ La classe ci-dessous met en œuvre une interface mais il y a toujours un **couplage** avec la classe IndividuFactory.

```
public class Service {  
  
    private InterfaceIndividuFactory factory = new IndividuFactory();  
  
    public void traitement() {  
        Individu individu = factory.getInstance();  
    }  
}
```



Faible couplage

- ❑ Spring nous aide également à diminuer le couplage
- ❑ Par rapport au pattern Créateur, j'ai remplacé la classe IndividuFactory par une interface afin de réduire le couplage.

```
public class Service {  
  
    @Autowired  
    private InterfaceIndividuFactory factory;  
  
    public void traitement() {  
        Individu individu = factory.getInstance();  
    }  
}
```

Couplage faible vs couplage fort

- ❑ Le couplage d'une classe avec une autre classe est appelé couplage fort
- ❑ Le couplage d'une classe avec une interface est appelé couplage faible
- ❑ Exemple de couplage faible

```
public class Service {  
  
    @Autowired  
    private InterfaceIndividuFactory factory;  
  
    public void traitement() {  
        Individu individu = factory.getInstance();  
    }  
}
```

Protection des variations

- ❑ L'idée de ce pattern est de réduire les impacts en cas de modification d'une classe.
- ❑ Exemple d'écriture répandue en Java mais qui augmente les vulnérabilités en cas de variation:

```
Double valeur = client.getCompteBancaire().getOperation(date).getMontant();
```

- ❑ Imaginez maintenant que je veuille modifier mon diagramme de classes en ajoutant un objet intermédiaire entre CompteBancaire et les opérations ! L'impact peut être conséquent !
- ❑ Le mieux est de développer une méthode dans la classe Client qui sera la seule à être sensible aux variations.

```
Double valeur = client.getMontantOperation(date);
```

Indirection

- ❑ L'idée de ce pattern est de réduire le couplage entre des ensembles de classes.
- ❑ Imaginons que certaines classes de mon application doivent utiliser une librairie permettant d'envoyer des SMS.
 - Cette librairie est assez complexe: beaucoup de classes
 - Si j'utilise directement les classes de cette librairie partout où j'en ai besoin, je vais créer un couplage fort entre mes classes et les classes de cette librairie.
 - Idée: créer une classe intermédiaire qui va être la spécialiste de l'envoi des SMS.
- ❑ Avantage de la classe d'Indirection:
 - Si je change de librairie de SMS je ne change qu'une seule classe
 - Le couplage entre mon application et les services SMS est réduit à une seule et unique classe.

Atelier (TP)

OBJECTIFS : Mettre en place les GRASP dans le cadre d'un refactoring

DESCRIPTION :

- Dans le TP **GRASPS** vous allez devoir mettre en œuvre ce que vous savez des GRASPS.

Chapitre 2

Introduction aux design patterns

Les langages objet

- ❑ Les concepts de la programmation orientée objet sont définis à la fin des années 60 et au début des années 70.

- ❑ Les premiers langages objets à succès apparaissent au début des années 1980
 - **C++ en 1983**
 - Objective-C en 1984
 - Eiffel en 1986
 - **Java en 1995**

- ❑ Les années 90 sont les années de l'explosion de la programmation orientée objet

Les langages objet

- ❑ Au milieu des années 90, certaines personnes prennent du recul et réalisent que des motifs de conception reviennent sans arrêt et pourraient être définis comme des bonnes pratiques.
- ❑ C'est en 1995 qu'apparaît le 1^{er} livre de référence sur les techniques de conception.
- ❑ Design Patterns – Elements of Reusable Object-Oriented Software - 1995
 - Gang of Four: Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides

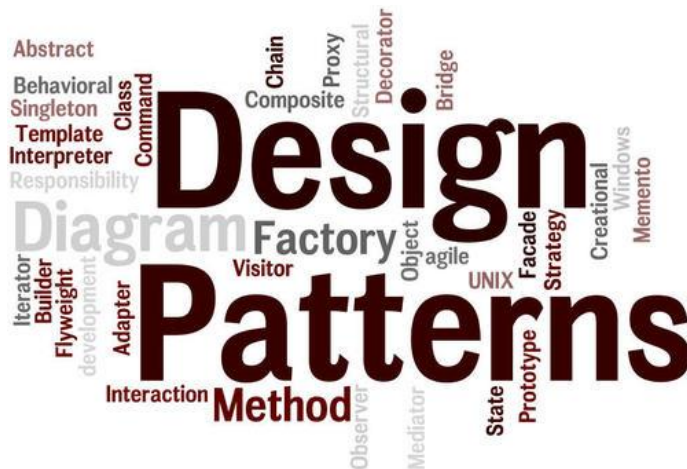
Qu'est-ce qu'un design pattern

❑ Un design pattern est une solution de conception

- impliquant une ou plusieurs classes,
- exploitant le plus souvent le polymorphisme et les interfaces
- Considéré comme une bonne pratique à mettre en œuvre

❑ Un design pattern est

- indépendant du langage de programmation
- Utilise le langage UML pour la formalisation



Les types de design patterns

❑ 4 grands types de design patterns:

- Architecturaux
- Créationnels
- Structurels
- Comportementaux

Chapitre 3

Patterns architecturaux

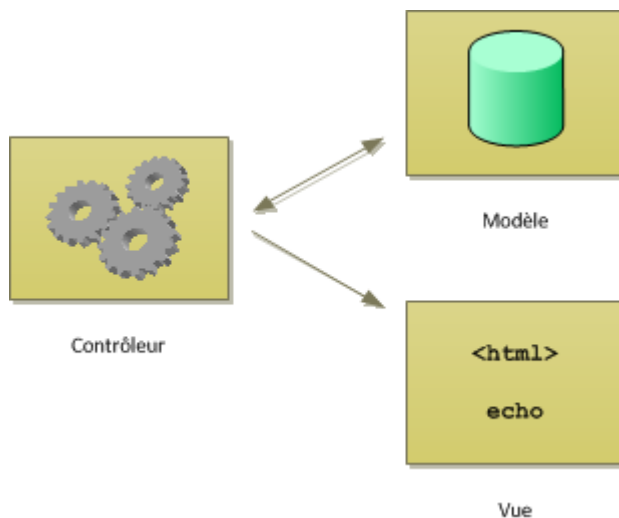
Les patterns architecturaux

- ❑ Un pattern architectural a un impact fort sur l'organisation générale du code au sein de votre application

- ❑ Exemple de patterns architecturaux
 - ❑ Client / serveur
 - ❑ **MVC**
 - ❑ **Layered (en couche)**
 - ❑ *Couche de présentation*
 - ❑ *Couche de service*
 - ❑ *Couche domaine (ou métier): les classes Java qui représentent le métier (Article, Panier, Achat, CompteClient, etc.)*
 - ❑ *Couche de persistance (exemple: DAO)*

Le pattern MVC

- ❑ MVC: Modèle, Vue, Contrôleur
- ❑ Dans ce pattern, la "Vue" représente l'IHM, mais peut aussi représenter un objet exploité par la vue:



Pattern MVC et bonnes pratiques (1/2)

- ❑ Mauvaise pratique : ajoutez des **attributs** dans la classe métier car on en a besoin dans la vue

```
public class Client {  
    private String nom;  
    private String prenom;  
    private LocalDate dateNaissance;  
}
```



```
public class Client {  
    private String nom;  
    private String prenom;  
    private String identite;  
    private LocalDate dateNaissance;  
    private String dateFormatee;  
    private int age;  
}
```

**Pas
bien**

- ❑ Besoin en HTML :

Identité	Date de naissance	Age
Jean Martin	12 avril 1990	29
Ian Liu	11 août 1982	36

Pattern MVC et bonnes pratiques (2/2)

- ❑ La classe métier représente le métier tel qu'exprimé par le client.
- ❑ Si la vue à un besoin, on crée une classe qui **est l'image de la vue**
- ❑ Une classe métier peut avoir **n** classes type "Vue" en fonction des besoins.

```
public class Client {  
    private String nom;  
    private String prenom;  
    private LocalDate dateNaissance;  
}
```



```
public class ClientVue {  
    private String identite;  
    private String dateNaissance;  
    private int age;  
}
```



- ❑ Appellations:
 - Classe de type "Vue"
 - Classe de type "Dto" pour Data Transfer Object

Pattern Layered et bonnes pratiques

- ❑ On évite de mettre tout le code dans la classe contrôleur
- ❑ <https://github.com/java-plus/gestion-des-absences/blob/master/src/main/java/fr/gda/controller/AdminFerieRttEmpController.java>

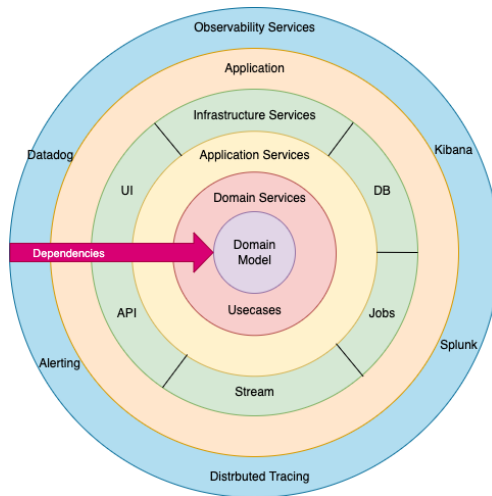


Pas
bien

- ❑ Il faut **séparer les responsabilités**.
- ❑ Les bonnes pratiques: le CU n'est pas exécuté par le contrôleur mais par une classe de service de type "Application services" chargée d'exécuter un cas d'utilisation complet.

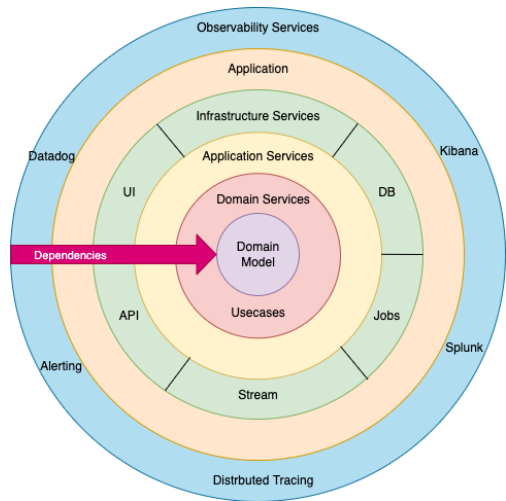
Introduction

- ❑ Le découpage en couches permet de bien structurer son application
- ❑ Les avantages sont multiples : évolutivité, réutilisabilité, robustesse, testabilité
- ❑ L'architecture en couches modèle, ou en oignons :



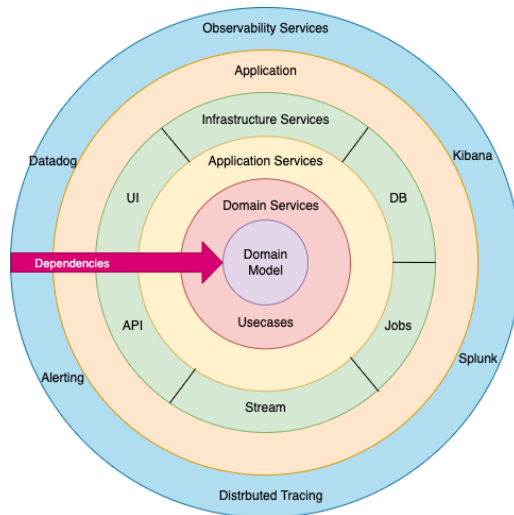
Observability services

- ❑ C'est la couche responsable du **monitoring** de l'application.
- ❑ C'est une couche extérieure à l'application et constituée d'outils pour surveiller et alerter si besoin.
- ❑ Elle peut également contenir des outils pour analyser et comprendre des problèmes de vie courante (bugs, piratage, etc..)



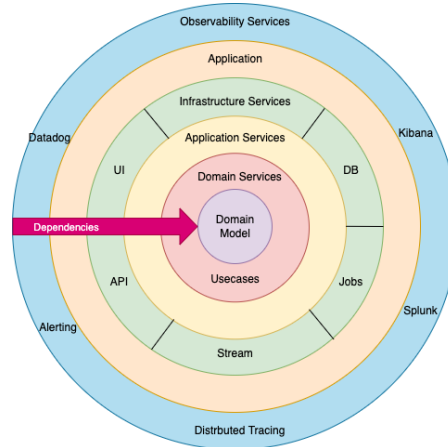
Infrastructure services ou contrôleurs

- ❑ C'est la couche à laquelle appartient les **contrôleurs** ou encore les **jobs**.
- ❑ Un contrôleur est un point d'entrée dans l'application : (endpoint pour une API, classe dotée d'une méthode exécutable main, etc.)
- ❑ C'est une couche technique, i.e. non-métier mais indispensable.



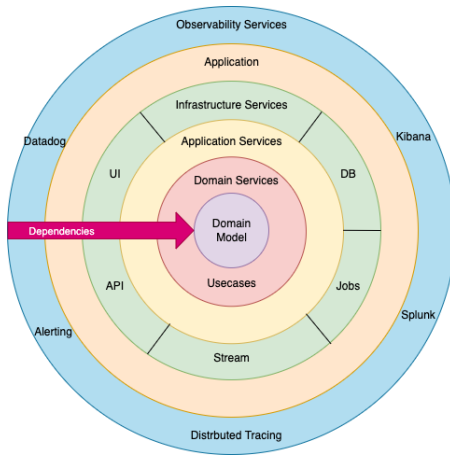
Application services

- ❑ C'est la couche responsable de la réalisation d'un cas d'utilisation, par exemple "créer un nouvel élève".
- ❑ Il y a une classe de type "Application services" par cas d'utilisation. Parfois on les appelle managers et peuvent s'appeler par exemple CreerEleveMgr.
- ❑ C'est ce type de classe qui orchestre les différents domain services (services métier) afin de réaliser le cas d'utilisation.



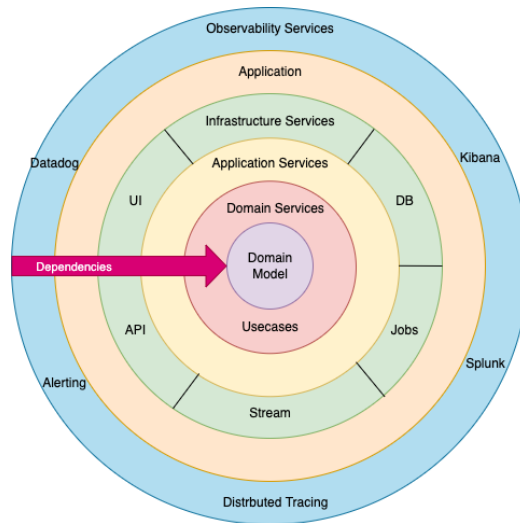
Domain services

- ❑ C'est la couche métier de l'application.
- ❑ Elle contient les règles (attributs obligatoires par exemple) et algorithmes.
- ❑ En général il y a un service par entité métier : EleveService, ClasseService, BulletinService, etc.
- ❑ La couche **Application Services** utilise une ou plusieurs classes de type **Domain services** pour les contrôles métier.



Domain model

- ❑ Ce sont les entités métier : Eleve, Classe, Bulletin, etc.
- ❑ Les entités métier n'ont aucune dépendance technique à l'exception des annotations qui sont tolérées. Les annotations permettent notamment de configurer la couche ORM.



Les échanges de données avec l'extérieur

- ❑ Aujourd'hui les **échanges avec le front**, ou des **applications externes**, utilisent principalement **JSON** ou **XML**.
- ❑ Mais il peut arriver aussi qu'on utilise des fichiers plus basiques, type CSV ou même positionnels sur les vieilles applications.
- ❑ Dans une API classique les contrôleurs échangent des données au format JSON avec l'extérieur (par défaut avec Spring Boot).
- ❑ Une mauvaise pratique est d'utiliser une entité métier, par exemple Eleve, pour mapper le message JSON.
- ❑ Une bonne pratique est d'utiliser une classe EleveModel ou EleveDto pour mapper le message JSON. Dans la couche **application services**, l'instance d'EleveModel est transformée en instance d'Eleve.

Le concept de DTO

- ❑ DTO est l'acronyme de Data Transfert Object.
- ❑ Lorsqu'un contrôleur renvoie des données à l'extérieur, il renvoie en réalité des objets qui sont convertis automatiquement en JSON par une couche technique.
- ❑ Avec Spring c'est une librairie appelée Jackson qui le fait.
- ❑ De même lorsqu'un contrôleur reçoit des données du front, le JSON a été transformé en instances d'objet par cette même couche technique.
- ❑ Une mauvaise pratique est d'utiliser une entité métier, par exemple Eleve, pour mapper le message JSON.
- ❑ Une bonne pratique est d'utiliser une classe EleveModel ou EleveDto pour mapper le message JSON. Dans la couche **application services**, l'instance d'EleveModel est transformée en instance d'Eleve.

Exemple Eleve vs EleveDto

- ❑ Supposons qu'on ait l'entité métier Eleve ci-dessous :

Eleve
-nom : string
-prenom : string
-dateNaissance : date
-adresse : string

- ❑ Et supposons également qu'on ait besoin d'afficher une liste d'élèves côté front avec l'âge et un format d'affichage de la date de naissance particulier :

Nom ▲	Prénom ▼	Date de naissance	Age	Adresse ▼
DOTENET	Camille	08/02/2011	13	4 rue des Lilas ...
PITON	Karim	15/10/2010	13	7 rue du Bois fleuri ...
GEAVAS	Bertrand	08/07/2010	13	1 rue du Marché ...

- ❑ Problèmes :
 - Qui calcule l'âge ? Ça ne va pas se faire tout seul !
 - Qui doit formater la date au format JJ/MM/AAAA ? Ça ne va pas se faire tout seul !

Exemple Eleve vs EleveDto

Nom ▲	Prénom ▼	Date de naissance	Age	Adresse ▼
DOTENET	Camille	08/02/2011	13	4 rue des Lilas ...
PITON	Karim	15/10/2010	13	7 rue du Bois fleuri ...
GEAVAS	Bertrand	08/07/2010	13	1 rue du Marché ...

Eleve
-nom : string
-prenom : string
-dateNaissance : date
-adresse : string

❑ Solution 1:

- on envoie au front une liste d'élèves et le front se débrouille

❑ Problèmes:

- Le front devient de facto un expert en calcul des âges, ce qui ne doit pas être une responsabilité du front
- Le front devient également de facto un expert en formatage des dates. Pourquoi pas mais il n'a pas la connaissance qu'a le back sur les préférences de l'utilisateur par exemple.

Exemple Eleve vs EleveModel

Nom ▲	Prénom ▼	Date de naissance	Age	Adresse ▼
DOTENET	Camille	08/02/2011	13	4 rue des Lilas ...
PITON	Karim	15/10/2010	13	7 rue du Bois fleuri ...
GEAVAS	Bertrand	08/07/2010	13	1 rue du Marché ...

Eleve
-nom : string
-prenom : string
-dateNaissance : date
-adresse : string
-age : int
-dateFormatee : string

- ❑ Solution 2:
 - on ajoute à l'entité métier **Eleve** 2 attributs: un attribut age et un attribut "date formatée"
 - On réalise côté back le calcul de l'âge et le formatage de la date en tenant compte des préférences de l'utilisateur.
- ❑ Problèmes:
 - On commet un "crime contre le métier"
 - Le métier doit être indépendant des contraintes de présentation.

Exemple Eleve vs EleveDto



- ❑ **Solution 3:**
 - On crée une classe EleveModel, ou EleveDto qui va avoir exactement les données attendues par la vue.
- ❑ **Problèmes:**
 - Ça donne un peu plus de travail au départ mais c'est très facile à faire évoluer en fonction des besoins des vues.
 - Il faut maintenir toute une zoologie de Dtos.

Conclusion



- ❑ De toutes les solutions c'est la solution 3 qui est **actuellement** considérée comme la meilleure pratique même si elle n'est pas exempte de défauts.

Chapitre 4

Patterns créationnels

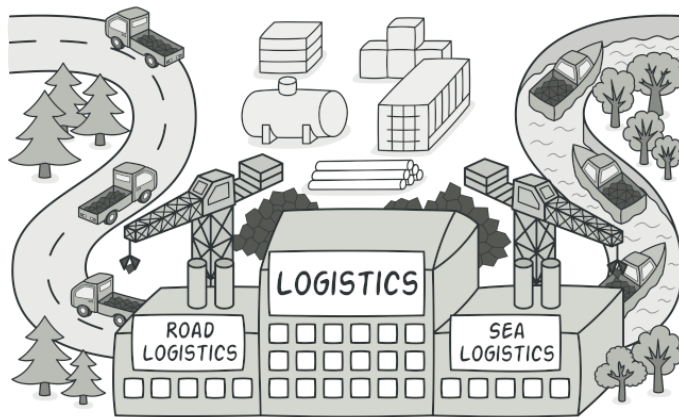
Les patterns créationnels

❑ Prennent en charge la création d'objets et l'instantiation:

- **Factory Method**
- **Abstract Factory**
- **Builder**
- Object Pool
- Prototype
- **Singleton**

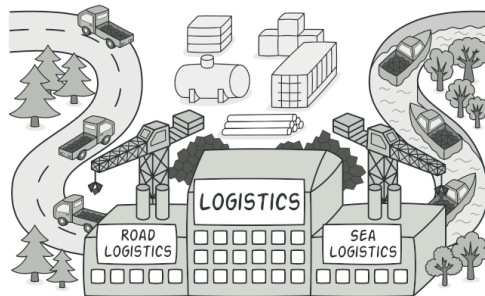
Factory method

- ❑ Prend en charge l'instantiation d'objets d'une même hiérarchie (ex: Transport)
- ❑ Utilisation du polymorphisme: l'appelant ne connaît pas le type concret de la classe retournée



Factory method

- ❑ Exemple: une méthode qui a type de retour Transport mais qui peut retourner une instance d'une classe qui hérite de Transport.
- ❑ Ce type de méthode a en général un paramètre qui permet à l'appelant de demander un type spécifique (exemple: 1 pour un camion, 2 pour un bateau, etc.)



Exemple

- ❑ Est souvent une classe indépendante avec une méthode static
- ❑ Cette méthode peut comporter d'autres paramètres que le type (marque, modèle, etc.)

```
public class TransportFactory implements ITransportFactory {
```

```
    public static Transport getTransport(int type) {  
        if (type == 1) {  
            return new Camion();  
        }  
        else if (type == 2) {  
            return new Bateau();  
        }  
        return null;  
    }  
}
```

```
// tr1 est une référence vers une instance de Camion
```

```
Transport tr1 = TransportFactory.getTransport(1);
```

```
// tr2 est une référence vers une instance de Bateau
```

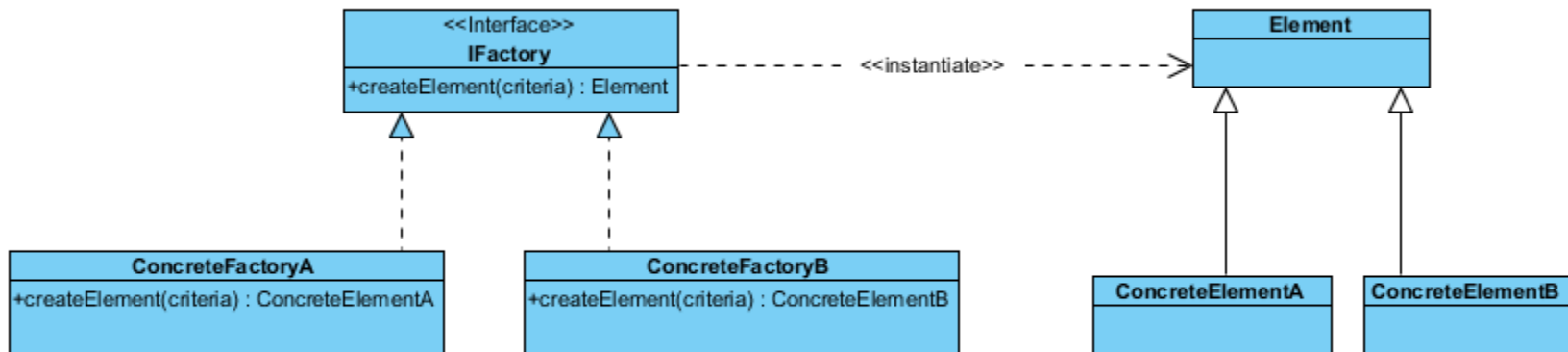
```
Transport tr2 = TransportFactory.getTransport(2);
```

```
// tr3 est null
```

```
Transport tr3 = TransportFactory.getTransport(3);
```

Factory method UML

❑ UML associé



Atelier (TP)

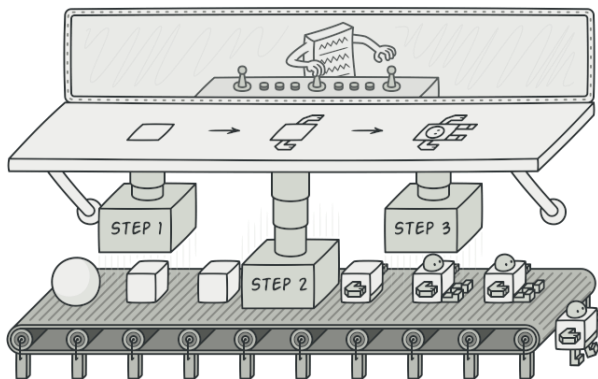
OBJECTIFS : Mettre en place le pattern Factory method

DESCRIPTION :

- Dans le TP **factory method** vous allez devoir mettre en place une Factory.

Builder - principes

- ❑ Un **builder** est une classe qui propose des méthodes pour créer de manière incrémentale un objet complexe.
- ❑ A la différence de la **Factory** qui concerne une hiérarchie d'objets, le **builder** se focalise sur la création d'une classe particulièrement complexe.
- ❑ Exemple: **StringBuilder** avec ses méthodes **append** pour créer une **String**.



Builder – « fluent » is cool !

- ❑ La mode pour les **builders** est de mettre en place une classe type « Fluent ».
- ❑ En français on appelle cela une **désignation chaînée** ou **chainage de méthodes**.
- ❑ Un bon exemple du « **fluent pattern** » est la classe **StringBuilder**

```
StringBuilder builder = new StringBuilder();  
String chaine = builder.append("Nous sommes le ").append(LocalDate.now()).toString();
```

Builder - exemple

```
public class ProduitBuilder {
```

```
    private Produit produit;
```

```
    public ProduitBuilder() {  
        this.produit = new Produit();  
    }
```

Crée l'instance de Produit

```
    public ProduitBuilder appendNom(String nom) {  
        produit.setNom(nom);  
        return this;  
    }
```

Complète le nom du Produit puis on retourne l'instance courante (this) du builder

```
    public Produit get() {  
        return produit;  
    }  
}
```

Retourne l'instance de Produit

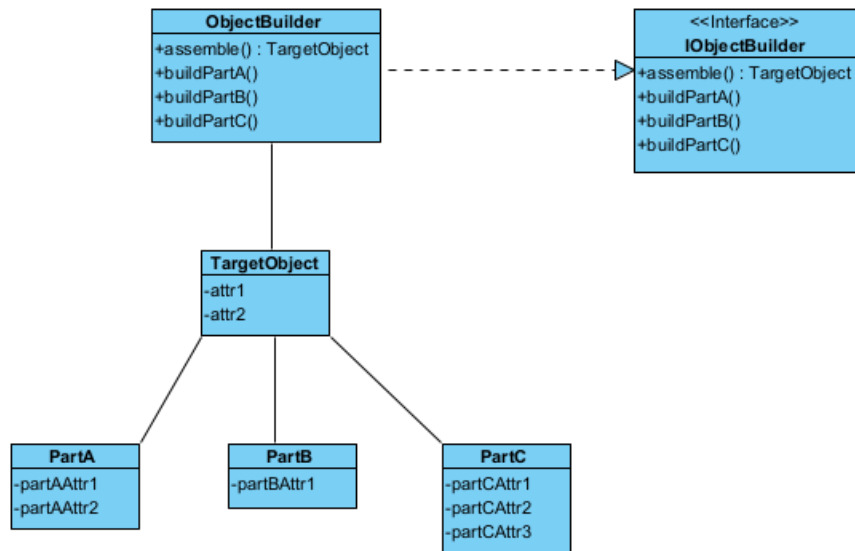
- ❑ Exemple un peu naïf. L'intérêt est de mettre de l'intelligence dans les méthodes append pour réaliser des traitements un peu plus sophistiqués.

Builder – bonnes pratiques

- ❑ **Problématique:** supposons que vous ayez créé un builder pour une classe Produit. Comment forcer les autres développeurs à utiliser votre builder plutôt que le constructeur de la classe Produit et ses setters ?
- ❑ **Solution:**
 - Mettre le builder dans le même package que la classe Produit
 - Mettre le constructeur de la classe Produit en visibilité package de manière à ce que seul le builder puisse créer une instance de Produit.

Builder – UML

❑ UML associé:



❑ **Important:** Les différentes parties de l'objet **TargetObject** peuvent être facultatives. Par exemple dans certains cas on veut construire un objet avec seulement **PartA** et **PartB** ou seulement **PartC** par exemple.

Builder – UML

❑ Exemple de code Java :

```
AvionBuilder builder = new AvionBuilder();  
Avion avion = builder.appendModele("Airbus", "A320-111")  
                    .appendMoteurs(2, Moteur.CFM56)  
                    .appendRangee(Classe.FIRST, 3, 2, 2)  
                    .appendRangee(Classe.ECO, 15, 3, 3).get();
```

- ❑ On ne passe que des types primitifs aux méthodes, ou des types de données, et l'assemblage se fait en interne.
- ❑ La complexité de la construction de l'objet est masquée.

Atelier (TP)

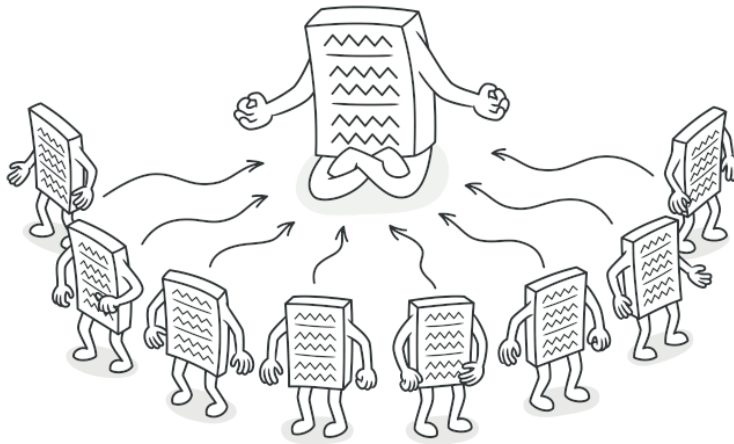
OBJECTIFS : Mettre en place le pattern **Builder**

DESCRIPTION :

- Dans le TP **builder** vous allez devoir mettre en place un **Builder**.

Le singleton

- ❑ Pattern de conception qui garantit qu'une seule instance d'un objet donné peut être créé.
- ❑ Fournit un point d'accès global à cette instance unique



Implémentation basique

❑ Exemple

```
public class Singleton
{
    /** Constructeur privé */
    private Singleton()
    {}

    /** Instance unique pré-initialisée */
    private static Singleton INSTANCE = new Singleton();

    /** Point d'accès pour l'instance unique du singleton */
    public static Singleton getInstance()
    {
        return INSTANCE;
    }
}
```

- ❑ Défaut de cette implémentation: on crée une instance du singleton systématiquement au démarrage de l'application.

Implémentation avec lazy loading

❑ Exemple

```
public class Singleton
{
    /** Constructeur privé */
    private Singleton()
    {}

    /** Instance unique non préinitialisée */
    private static Singleton INSTANCE = null;

    /** Point d'accès pour l'instance unique du singleton */
    public static Singleton getInstance()
    {
        if (INSTANCE == null)
        {
            INSTANCE = new Singleton();
        }
        return INSTANCE;
    }
}
```

❑ Défaut de cette implémentation: non thread-safe

Implémentation thread-safe

❑ Exemple

```
public class Singleton
{
    /** Constructeur privé */
    private Singleton()
    {}

    /** Instance unique non préinitialisée */
    private static Singleton INSTANCE = null;

    /** Point d'accès pour l'instance unique du singleton */
    public static synchronized Singleton getInstance()
    {
        if (INSTANCE == null)
        {
            INSTANCE = new Singleton();
        }
        return INSTANCE;
    }
}
```

- ❑ Défaut de cette implémentation: la méthode synchronized occasionne un surcoût en terme de performance.

Implémentation type holder

❑ Exemple

```
public class Singleton
{
    /** Constructeur privé */
    private Singleton()
    {}

    /** Holder */
    private static class SingletonHolder
    {
        /** Instance unique non préinitialisée */
        private final static Singleton instance = new Singleton();
    }

    /** Point d'accès pour l'instance unique du singleton */
    public static Singleton getInstance()
    {
        return SingletonHolder.instance;
    }
}
```

- ❑ Zéro défaut: la classe interne est chargée au moment où getInstance() est appelée.

Exemple complet

- ❑ Exemple pour charger un fichier une seule fois et proposer une méthode de recherche de données dans ce fichier.

```
public class Configuration {  
  
    private static Configuration instance = new Configuration();  
    private File fichier;  
  
    private Configuration() {  
        fichier = new File("...");  
    }  
  
    public static Configuration getInstance() {  
        return instance;  
    }  
  
    public String getValue(String key) {  
        String value = null;  
  
        //TODO accéder au fichier pour rechercher une donnée  
        return value;  
    }  
}
```

Atelier (TP)

OBJECTIFS : Mettre en place le pattern **Singleton**

DESCRIPTION :

- Dans le TP **singleton** vous allez devoir mettre en place un **Singleton**.

Chapitre 5

Patterns structuraux

Les patterns structuraux

- ❑ Définissent les différentes manières d'assembler les objets entre eux pour modéliser des concepts.
- ❑ Exemple:
 - la représentation d'un arbre (notions de branches et de feuilles)
 - Une branche peut posséder une branche et aussi des feuilles.

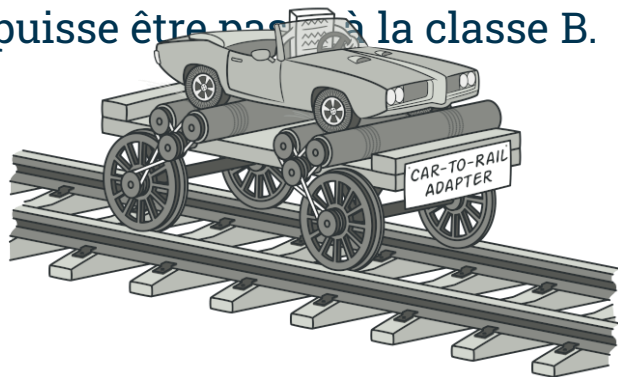
Les patterns structuraux

❑ Les patterns structuraux:

- **Adapter**
- **Bridge**
- **Composite**
- Decorator
- **Facade**
- Flyweight
- Private Class Data
- Proxy

L'adapter

- ❑ L'adapter permet de "faire rentrer des formes rondes dans des trous carrés", ou encore permet de mettre en relation des objets qui manipulent des objets différents.
- ❑ Imaginer qu'un objet fournisse un résultat sous la forme d'une classe A mais qu'un autre objet attende le résultat sous la forme d'une classe B.
- ❑ L'adapter va être capable d'adapté A pour qu'il puisse être passé à la classe B.



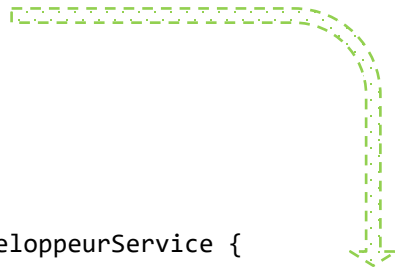
Problème

```
public interface IDeveloppeur {  
    String ecrireCode();  
}
```



```
class DeveloppeurExpert implements IDeveloppeur {  
    public String ecrireCode() {  
        return "code ultra efficace";  
    }  
}
```

```
class Architecte {  
    public String ecrireAlgorithme() {  
        return "algorithme très sophistiqué";  
    }  
}
```



```
public class DeveloppeurService {  
    public void afficheDeveloppeur(IDeveloppeur developpeur) {  
        System.out.println(developpeur.ecrireCode());  
    }  
}
```

**Architecte n'implémente pas
IDeveloppeur**

Solution avec mise en place de l'adapteur

```
public interface IDeveloppeur {  
    String ecrireCode();  
}
```

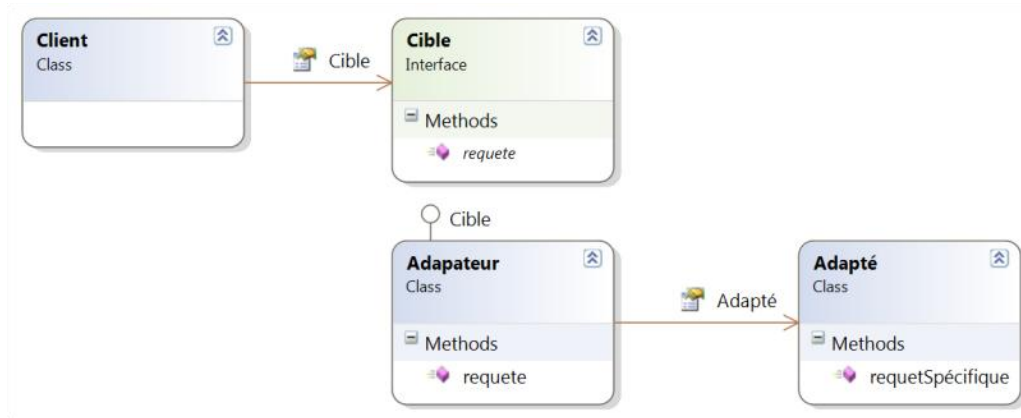


```
class ArchitecteAdapter implements IDeveloppeur {  
  
    private Architecte architecte;  
    public ArchitecteAdapter(Architecte architecte) {  
        this.architecte = architecte;  
    }  
    public String ecrireCode() {  
        return architecte.ecrireAlgorithme();  
    }  
}
```



```
public class DeveloppeurService {  
    public void afficheDeveloppeur(IDeveloppeur developpeur) {  
        System.out.println(developpeur.ecrireCode());  
    }  
}
```


UML Adapter



- ❑ Le client c'est la classe `DeveloppeurService` qui utilise une cible, l'interface `IDeveloppeur`
- ❑ La classe `Adapté` c'est la classe `Architecte`
- ❑ La classe `ArchitecteAdapter` c'est l'`Adaptateur`.

Atelier (TP)

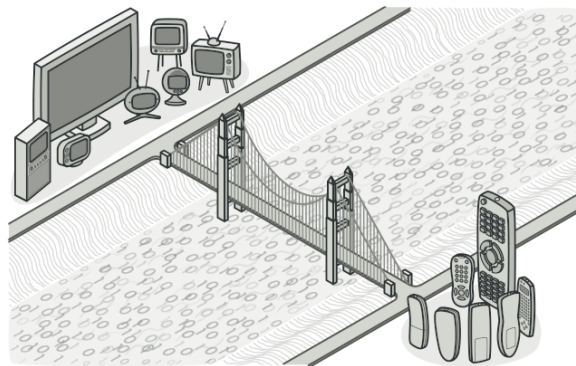
OBJECTIFS : Mettre en place le pattern Adapter

DESCRIPTION :

- Dans le TP adapter vous allez devoir implémenter le pattern Adapter.

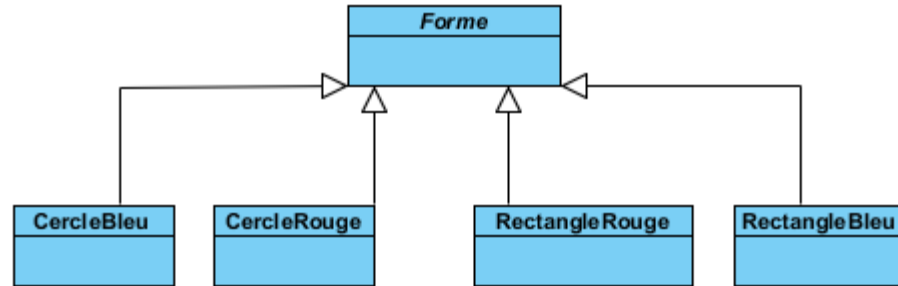
Le Bridge

- ❑ Le bridge remplace un héritage par de l'association.
- ❑ Utile lorsque la relation d'héritage n'est pas forcément pertinente et donne naissance à de très grosses hiérarchies d'objets.



Problème (1/2)

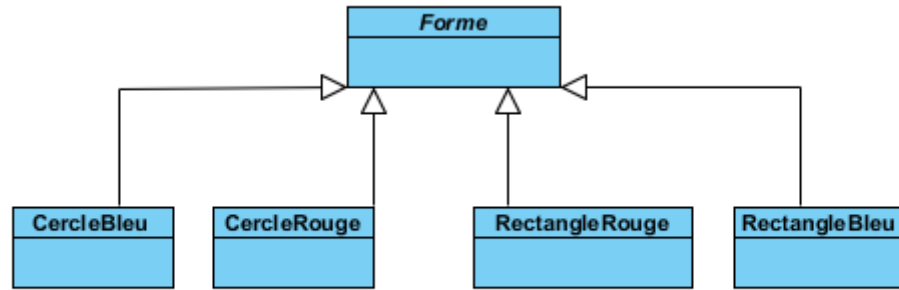
- ❑ On a une classe abstraite **Forme** avec des formes de base: **Cercle** et **Rectangle**
- ❑ On veut également ajouter la dimension couleur: **Rouge** et **Bleu** pour chaque forme



- ❑ Une solution possible est celle indiquée ci-dessus

Problème (2/2)

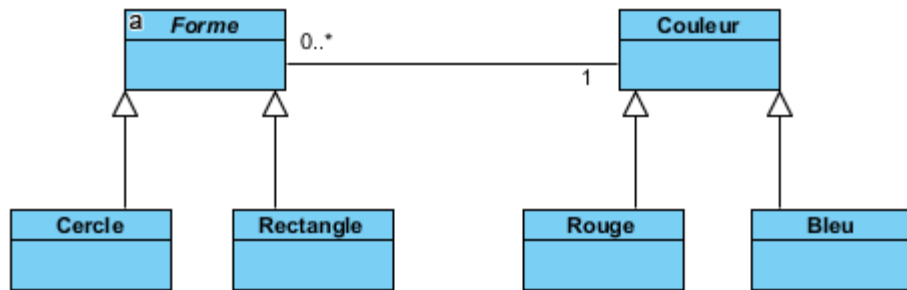
- ❑ Mais, avec cette abstraction, le nombre de classes filles augmente de manière exponentielle: à chaque nouvelle forme et chaque nouvelle couleur.



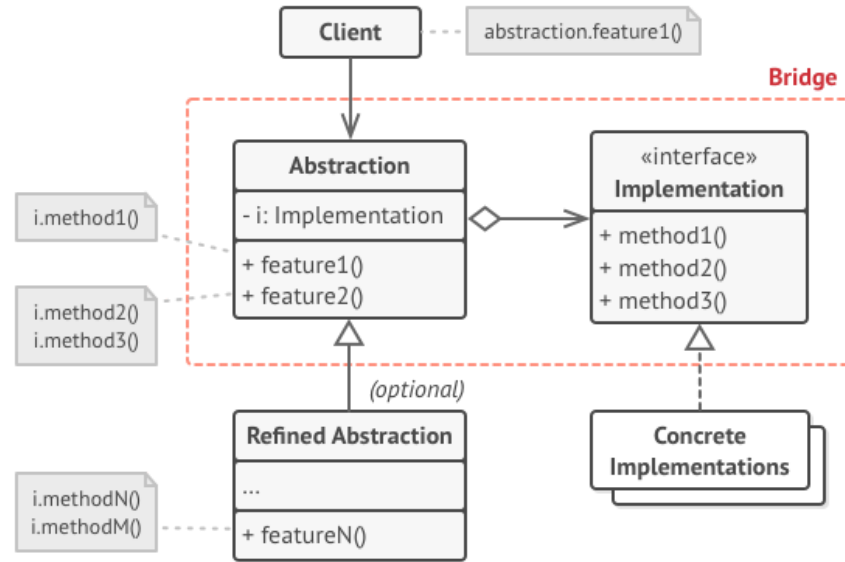
- ❑ Le problème vient d'une confusion entre ce qui relève de l'abstraction (la forme) et ce qui relève d'une propriété (la couleur).

Solution avec mise en place du Bridge

- ❑ La solution consiste à faire de la **couleur** un **objet associé** avec sa propre hiérarchie.
- ❑ Dans la solution, l'objet **Forme**, qui est soit un **Cercle**, soit un **Rectangle**, a un attribut de type **Couleur**, qui est soit **Bleu**, soit **Rouge**.



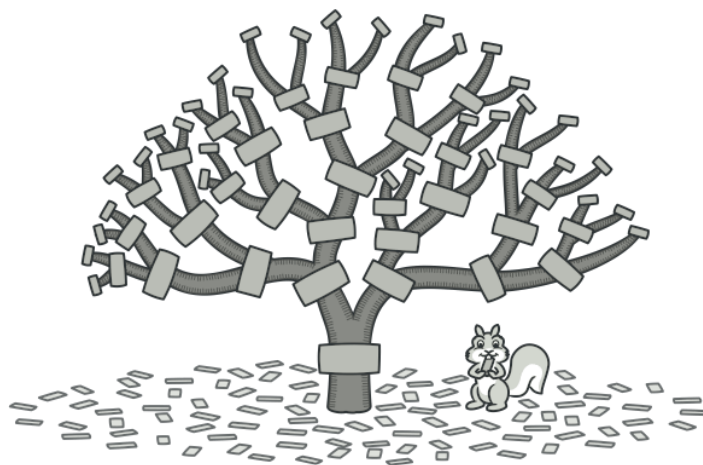
UML Bridge



- ❑ Ici l'abstraction est l'objet Forme qui a une référence sur une Implémentation, la classe Couleur
- ❑ Redefined abstraction: Rectangle, Cercle
- ❑ Concrete implementations: Rouge, Bleu

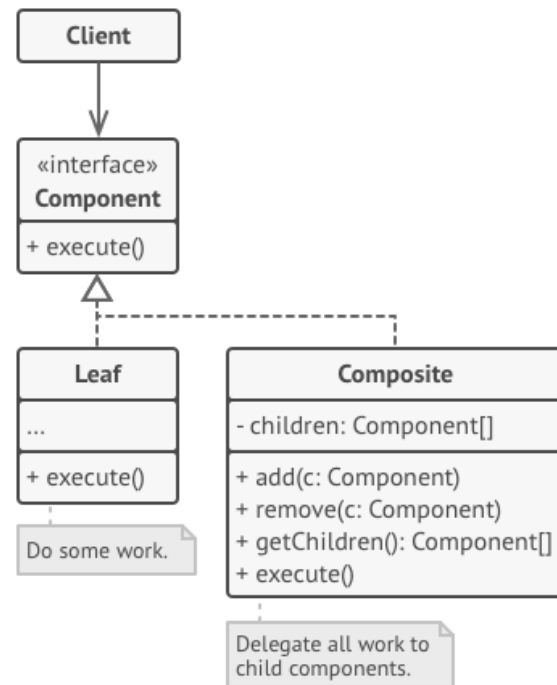
Le Composite

- ❑ Le Composite sert à représenter les objets de type "arbre" avec des branches et des feuilles.
- ❑ Exemple: l'organisation d'une entreprise, le file system sur votre disque dur avec ses répertoires et ses fichiers, etc..



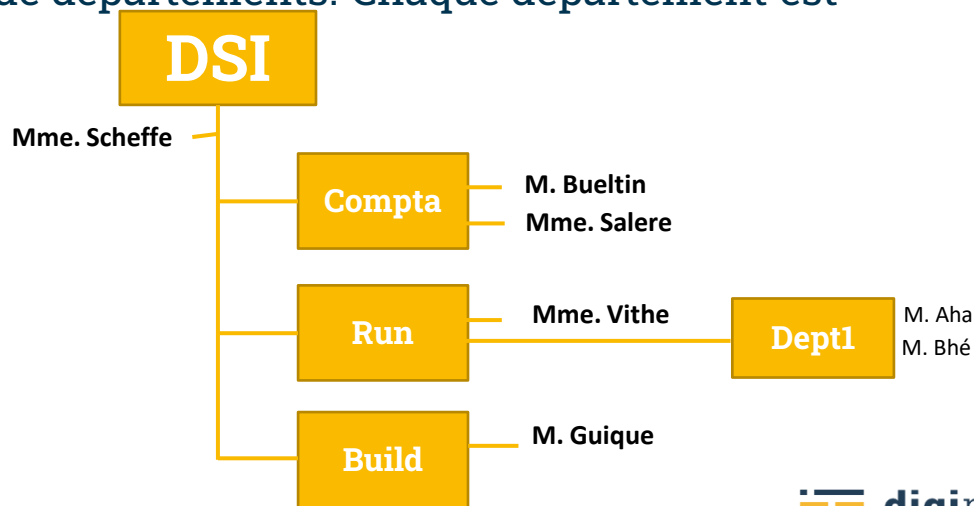
UML Composite

- ❑ Ici le Component représente une abstraction
- ❑ L'objet Leaf hérite de Component et est une feuille
- ❑ L'objet Composite hérite aussi de Component mais peut contenir d'autres Component.



Exemple Composite

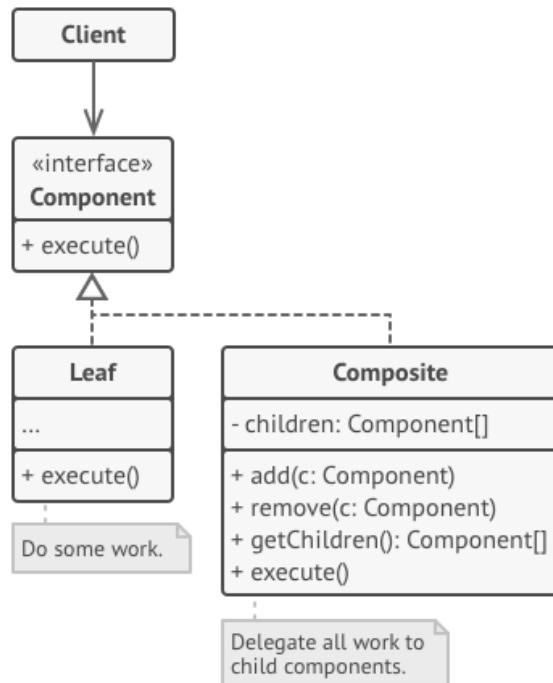
- ❑ Quelles classes utiliseriez vous pour représenter l'organisation dans une entreprise ?
- ❑ Une entreprise est constituée de départements. Chaque département est constitué:
 - D'autres départements
 - De personnes



Exemple Composite

```
public class Personne implements Entite {  
  
    private String nom;  
    private String prenom;  
  
    // + Getters et setters  
}  
  
public class Departement implements Entite {  
  
    private String nom;  
    private List<Entite> entites = new ArrayList<>();  
    // + Getters et setters  
}
```

- ❑ Ici L'Entite est le **Component**
- ❑ La Personne est la **Leaf**
- ❑ Le Département est le **Composite**



Atelier (TP)

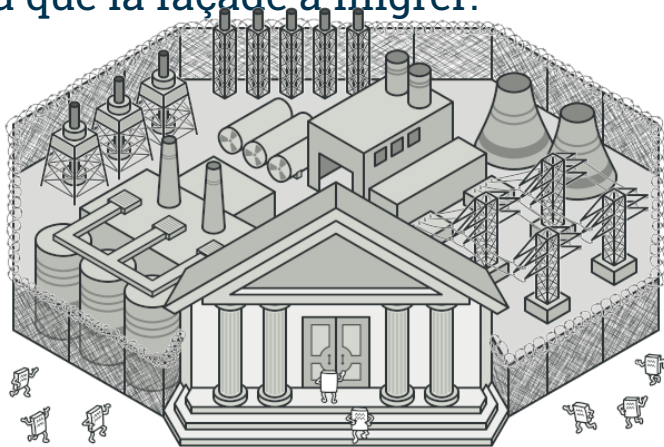
OBJECTIFS : Mettre en place le pattern composite

DESCRIPTION :

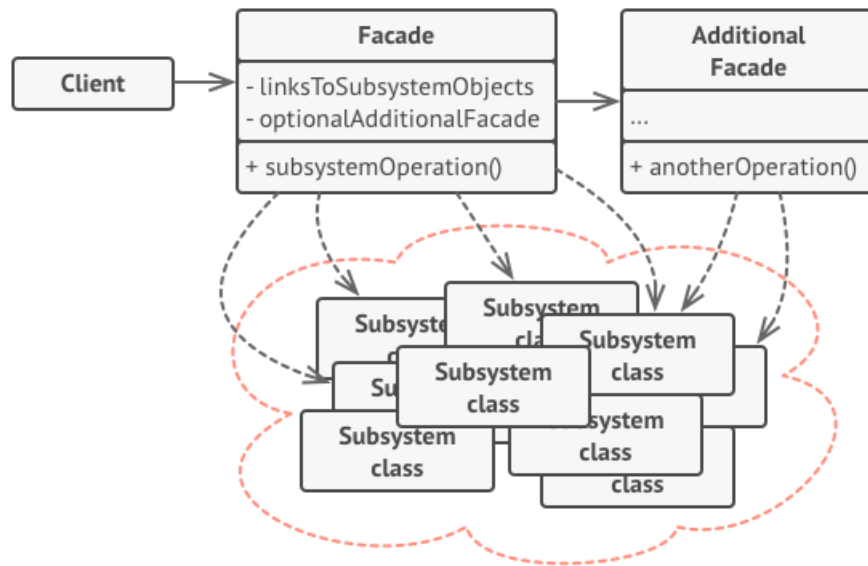
- Dans le TP Composite vous allez devoir créer des classes qui représente l'organisation d'une société avec des services et des employés.

La Façade

- ❑ La Façade est un pattern assez simple qui impose de systématiquement passer par une classe, ou un jeu de classes, lorsqu'on utilise une librairie externe à l'application.
- ❑ Plutôt que d'utiliser la librairie partout dans l'application, on utilise la façade. En cas de changement de librairie il n'y a que la façade à migrer.



UML Façade



- ❑ Le client ne passe que par la façade et ses façades additionnelles pour accéder aux "Subsystem classes" qui sont des classes fournies par une librairie externe.
- ❑ L'idée est de réduire le risque en cas de migration.

Chapitre 6

Patterns comportementaux

Les patterns comportementaux

- ❑ Définissent les différentes manières de faire communiquer les objets entre eux, notamment au sein d'algorithmes.
- ❑ Les patterns comportementaux:
 - Chain of responsibility,
 - Command,
 - Iterator,
 - Mediator,
 - Memento,
 - Observer,
 - **Strategy**,
 - **State**
 - **Template method**
 - Visitor

Pattern Strategy

- ❑ Le pattern Strategy Permet de définir une famille d'algorithmes, à raison d'un algorithme par classe fille et ensuite de choisir un algorithme sur étagère en fonction de son besoin.



Exemple Strategy

- ❑ Lorsque ce pattern est implémenté correctement les algorithmes sont interchangeables

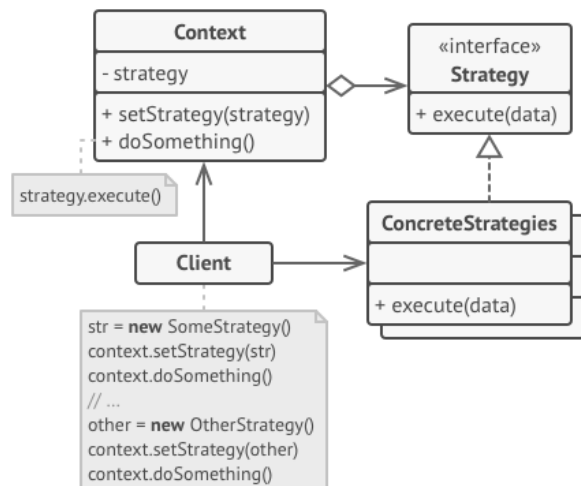
```
public class CalculScoreNutritif {  
  
    private StrategieCalcul strategieCalcul = StrategieCalculFactory.getStrategie(TypeStrategie.DEFAULT);  
  
    public String calcul(Produit produit) {  
  
        return strategieCalcul.evaluer(produit);  
    }  
  
    public void modifierStrategie(StrategieCalcul strategie) {  
        strategieCalcul = strategie;  
    }  
}
```

Exemple Strategy

❑ Appel depuis une classe exécutable

```
public class TestNutriscore {  
  
    public static void main(String[] args) {  
  
        ProduitDao produitDao = new ProduitDao();  
  
        CalculScoreNutritif scoreNutritif = new CalculScoreNutritif();  
        String score1 = scoreNutritif.calcul(produitDao.findById(1));  
  
        scoreNutritif.modifierStrategie(StrategieCalcul.getStrategie(TypeStrategie.IA));  
        String score2 = scoreNutritif.calcul(produitDao.findById(1));  
  
    }  
  
}
```

UML Strategy



- ❑ La classe Client dans l'exemple est la classe TestNutriscore.
- ❑ La classe Context est la classe CalculNutriscore.

Atelier (TP)

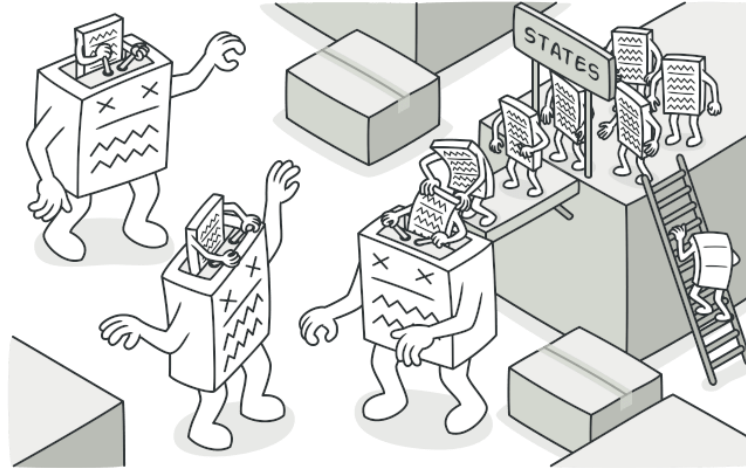
OBJECTIFS : Mettre en place le pattern **strategy**

DESCRIPTION :

- Dans le TP **Strategy** vous allez devoir modifier un code existant qui ne respecte aucune bonne pratique et implémenter le pattern Strategy.

Pattern State

- ❑ Le pattern State permet s'associer un état à un objet.
- ❑ Ce faisant, les méthodes de cet objet effectuent des traitements différents selon l'état.



Exemple naïf

- ❑ Dans l'exemple ci-dessous, on met en œuvre ce système d'état mais sans utiliser le pattern

```
public class Video {  
  
    private String etat = "";  
  
    public void setEtat(String etat) {  
        this.etat = etat;  
    }  
  
    public void action() {  
        if (etat.equalsIgnoreCase("PLAY")) {  
            System.out.println("La vidéo est en lecture");  
        } else if (etat.equalsIgnoreCase("PAUSE")) {  
            System.out.println("La vidéo est en pause");  
        }  
    }  
}
```

```
public class Main {  
    public static void main(String args[]) {  
        Video video = new Video();  
  
        video.setEtat("PLAY");  
        video.action();  
  
        video.setEtat("PAUSE");  
        video.action();  
    }  
}
```

Exemple naïf

- ❑ L'idée va être de mettre en place un système ressemblant fortement au Pattern Strategy

```
public class Video {  
  
    private String etat = "";  
  
    public void setEtat(String etat) {  
        this.etat = etat;  
    }  
  
    public void action() {  
        if (etat.equalsIgnoreCase("PLAY")) {  
            System.out.println("La vidéo est en lecture");  
        } else if (etat.equalsIgnoreCase("PAUSE")) {  
            System.out.println("La vidéo est en pause");  
        }  
    }  
}
```

```
public class Main {  
    public static void main(String args[]) {  
        Video video = new Video();  
  
        video.setEtat("PLAY");  
        video.action();  
  
        video.setEtat("PAUSE");  
        video.action();  
    }  
}
```


Mise en place pattern State - #1

❑ Etape 1:

- remplacer l'attribut d'instance String `etat` par une interface **VideoEtat** avec une méthode **action()**.
- La « ifelseitude » dans la méthode **action** est remplacée par un appel à la méthode **action()** de **VideoEtat**

```
public class Video {  
  
    private VideoEtat etat;  
  
    public void setEtat(String etat) {  
        this.etat = etat;  
    }  
  
    public void action() {  
        etat.action();  
    }  
}  
  
public interface VideoEtat {  
  
    void action();  
}
```

Mise en place pattern State - #2

❑ Etape 2:

- Créer des classes qui implémentent l'interface **VideoState** et dont la méthode `action()` va implémenter un état particulier (en pause ou en lecture).

```
public interface VideoEtat {  
  
    void action();  
}
```

```
public class VideoPause implements VideoEtat {  
    @Override  
    public void action() {  
        System.out.println("La vidéo est en pause");  
    }  
}
```

```
public class VideoLecture implements VideoEtat {  
    @Override  
    public void action() {  
        System.out.println("La vidéo est en lecture");  
    }  
}
```

Mise en place pattern State - #3

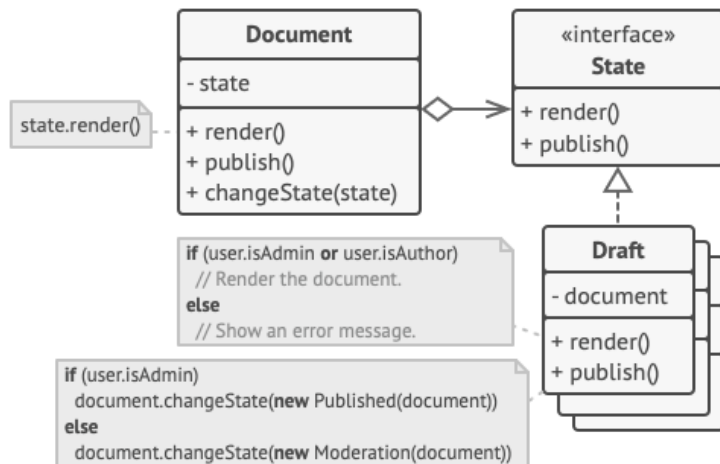
❑ Etape 3:

- Dans la classe appelante on utilise ces implémentations.

```
public class Video {  
  
    private VideoEtat etat;  
  
    public void setEtat(String etat) {  
        this.etat = etat;  
    }  
  
    public void action() {  
        etat.action();  
    }  
}
```

```
public class Main {  
    public static void main(String args[]) {  
        Video video = new Video();  
  
        video.setEtat(new VideoLecture());  
        video.action();  
  
        video.setEtat(new VideoPause());  
        video.action();  
    }  
}
```

UML State



- ❑ La classe Document possède un State
- ❑ Ce state est en réalité une interface qui possède plusieurs implémentations possibles.

Différence entre State et Strategy

- ❑ Conceptuellement les patterns se ressemblent mais
- ❑ Le pattern **Strategy** retourne un résultat identique mais chaque implémentation le calcule de manière différente.
- ❑ Le pattern **State** effectue des traitements différents reflétant ainsi le fait qu'un objet ne se comporte pas toujours de la même manière selon les cas.
- ❑ Le pattern **Strategy** ne contient qu'une seule méthode
- ❑ Le Pattern **State** peut contenir plusieurs méthodes car l'objet à état peut également avoir plusieurs méthodes impactées par un changement d'état.

Atelier (TP)

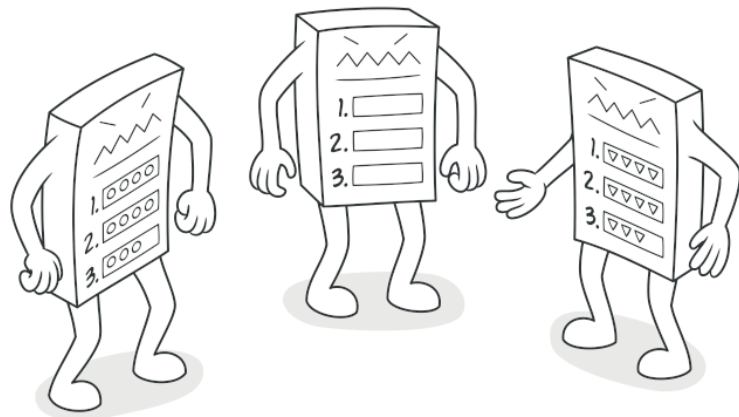
OBJECTIFS : Mettre en place le pattern **State**

DESCRIPTION :

- Dans le TP **State** vous allez devoir modifier un code existant qui ne respecte aucune bonne pratique et implémenter le pattern **State**.

Pattern Template method

- ❑ Le pattern Template method définit le squelette d'un algorithme dans une super-classe et laisse les classes filles redéfinir des étapes spécifiques de l'algorithme.

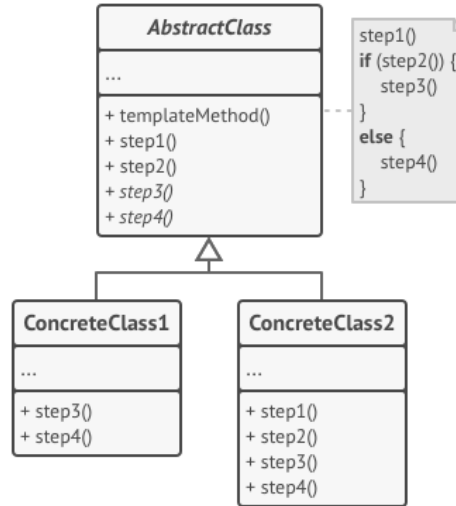


Exemple Template method

- ❑ Lorsque ce pattern est implémenté correctement les algorithmes sont interchangeables
- ❑ Ici la **template method** est la méthode **calcul** qui contient l'algorithme général.

```
public abstract class CalculComptable {  
  
    public final double calcul(Produit produit) {  
  
        double prixRevient = calculCoutMatierePremiere(produit);  
        prixRevient += calculConception(produit);  
        return prixRevient * appliquerTva();  
    }  
  
    public abstract double calculCoutMatierePremiere(Produit produit);  
    public abstract double calculCoutConception(Produit produit);  
    public abstract double appliquerTva();  
}
```


UML Template method



Atelier (TP)

OBJECTIFS : Mettre en place le pattern **template method**

DESCRIPTION :

- Dans le TP **Template method** vous allez devoir refactorer une méthode en appliquant le design pattern correspondant.

Dependency Inversion Principle



Introduction

- ❑ Le principe **Dependency Inversion Principle** est un principe très important.
- ❑ C'est celui qui explique l'organisation des couches dans Spring Boot par exemple
- ❑ Ce principe dit qu'il faut éviter au maximum les dépendances entre classes concrètes.
- ❑ Exemple qui viole le DIP:

```
class Ampoule {  
  
    public void allumer() {  
    }  
  
    public void eteindre() {  
    }  
}
```

```
class Interrupteur {  
  
    private Ampoule ampoule;  
  
    public Interrupteur(Ampoule ampoule) {  
        this.ampoule = ampoule;  
    }  
  
    public void appuyer() {  
        if (ampoule.isOn()) {  
            ampoule.eteindre();  
        } else {  
            ampoule.allumer();  
        }  
    }  
}
```

Exemple

- ❑ **Exemple qui respecte le DIP** : pour respecter le DIP il est nécessaire d'ajouter une abstraction entre Interrupteur et Ampoule sous la forme d'une interface

```
interface Allumable {  
  
    void allumer();  
  
    void eteindre();  
}
```

```
class Ampoule implements Allumable {  
  
    @Override  
    public void allumer() {  
    }  
  
    @Override  
    public void eteindre() {  
    }  
}
```

```
class Interrupteur {  
    private Allumable allumable;  
  
    public Interrupteur(Allumable allumable) {  
        this.allumable = allumable;  
    }  
  
    public void appuyer() {  
        if (allumable.isOn()) {  
            allumable.eteindre();  
        } else {  
            allumable.allumer();  
        }  
    }  
}
```

Intérêt du DIP pour les tests

- ❑ L'intérêt du DIP ne se situe pas réellement au niveau des entités métier mais plutôt au niveau des classes de service.
- ❑ C'est la raison pour laquelle ce principe est à la base de Spring.
- ❑ L'intérêt est de faciliter le remplacement d'une implémentation par une autre et de grandement faciliter la testabilité via l'injection de **mocks**.
- ❑ Imaginons qu'une **classe PersonneService** ait besoin d'une **classe PersonneJpa** pour accéder en base de données.
 - Je ne peux pas créer de tests unitaires de PersonneService sans accéder à la base de données puisqu'il y a une dépendance forte entre les 2 classes.
- ❑ Imaginons maintenant que la classe PersonneService ait une référence sur une **interface PersonneRepository** pour accéder en base de données.
 - Je peux facilement injecter une implémentation PersonneRepositoryTest qui n'accède pas à la base de données dans cette classe.

Exemple pour le test (1/2)

```
@Service
public class PersonneServiceImpl implements PersonneService {

    private PersonneRepository personneRepository;

    public PersonneServiceImpl(PersonneRepository personneRepository) {
        this.personneRepository = personneRepository;
    }

    @Override
    public PersonneAvecRoleDto findById(Long id) {
        Optional<Personne> opt = personneRepository.findById(id);
        if (opt.isPresent()) {
            return new PersonneAvecRoleDto(opt.get());
        }
        return null;
    }
}
```

Spring injecte
L'implémentation par défaut qui est
PersonneRepositoryImpl

```
public interface PersonneRepository {

    Optional<Personne> findById(String idI);
}
```

```
@Service
public class PersonneRepositoryImpl implements
PersonneRepository {

    public Optional<Personne> findById(String idI) {
    }
}
```

Exemple pour le test (2/2)

Pour les tests unitaires, injection d'un mock qui est une classe qui implémente l'interface mais n'accède pas à la bdd.

```
public class PersonneServiceImplTest {  
  
    @Test  
    public void testFindById() {  
        PersonneServiceImpl service = new PersonneServiceImpl(new PersonneRepositoryTest());  
    }  
}
```

```
public class PersonneRepositoryTest implements PersonneRepository {  
  
    @Override  
    public Optional<Personne> findById(Long id) {  
        return Optional.empty();  
    }  
}
```