Introduction aux design patterns


DESIGN PATTERNS

Programme détaillé ou sommaire

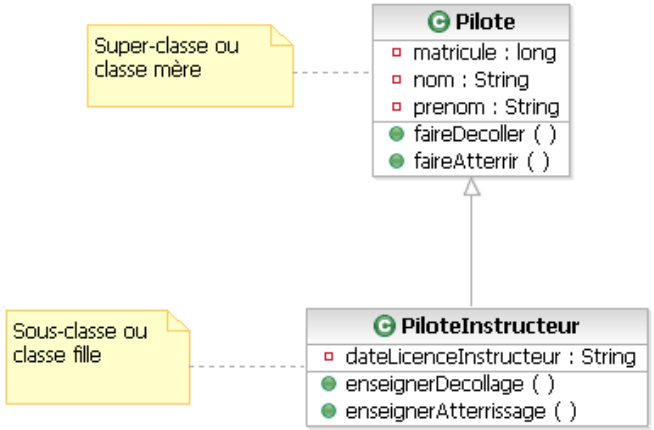
Quelques rappels Objet
Introduction aux GRASP
Introduction aux Design Patterns
Patterns architecturaux
Patterns créationnels
Patterns structuraux
Patterns comportementaux
Découpage en couches et bonnes pratiques

Chapitre 1
Rappels objets / UML

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Héritage

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

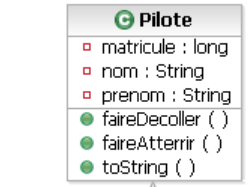



```
class PiloteInstructeur extends Pilote {
   constructor(nom, prenom, matricule, dateLicence) {
       super(nom, prenom, matricule);
       this.dateLicence = dateLicence;
   }
   enseignerDecollage() {
   }
   enseignerAtterrissage() {
   }
}
```

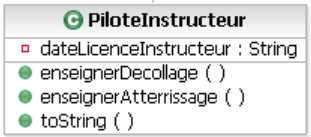

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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




```
toString() {
    return `Pilote: ${this.prenom} ${this.nom} (Matricule: ${this.matricule})`;
}
```

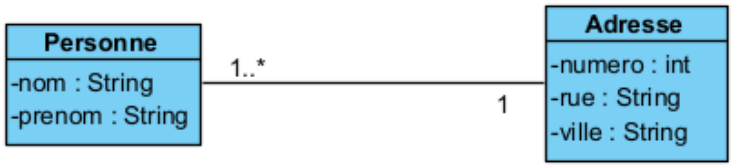



```
// Redéfinition de la méthode toString()
toString() {
    return `${super.afficherInfos()} - Instructeur (Licence: ${this.dateLicence})`;
}
```


diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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




```
class Personne {
    constructor(nom, prenom, adresse) {
        this.nom = nom;
        this.prenom = prenom;
        this.adresse = adresse;
    }
}
```

```
class Adresse {
    constructor(numero, rue, ville) {
        this.numero = numero;
        this.rue = rue;
        this.ville = ville;
    }
}
```


diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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

```
class Collaborateur {
    constructor(nom, prenom) {
        if (this.constructor === Collaborateur) {
            throw new Error("Collaborateur est une classe
abstraite et ne peut pas être instanciée directement");
        }
        this.nom = nom;
        this.prenom = prenom;
    }
    calculerSalaire() {
        throw new Error("La méthode calculerSalaire() doit
être implémentée dans la classe fille");
    }
}
```

```
class Pilote extends Collaborateur {
    constructor(nom, prenom, matricule, heuresVol) {
        super(nom, prenom);
        this.matricule = matricule;
        this.heuresVol = heuresVol;
    }
    // Implémentation obligatoire de la méthode abstraite
    calculerSalaire() {
        const salaireBase = 3000;
        const primeHeures = this.heuresVol * 50;
        return salaireBase + primeHeures;
    }
}
```


diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Enumération

- Plusieurs manières de créer une énumération en javascript

```
class TypeTransportClasse {
    static CAMION = 'camion';
    static BATEAU = 'bateau';
    static AVION = 'avion';
}
```


diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Chapitre 2
Introduction aux GRASP

GRASP

General **R**esponsibility **A**ssignment **S**oftware

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Classe et responsabilités

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

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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.

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Exemple pour comprendre les GRASP

Que pensez-vous de cette classe ?

```
class Adresse {
    constructor(numeroRue, rue, ville, dateCreation) {
        this.numeroRue = numeroRue;
        this.rue = rue;
        this.ville = ville;
        this.dateCreation = dateCreation;
    }
    // Méthode pour convertir une chaîne de caractères en date avec moment
    convertirEnDate(chaineDate) { // Supposons que moment.js est chargé const
        dateConvertie = moment(chaineDate);
        if (!dateConvertie.isValid()) {
            throw new Error(`Date invalide: ${chaineDate}`);
        }
        return dateConvertie;
    }
}
```


diginamic FORMATION
« Votre passeport pour l'emploi numérique »

A propos de l'exemple précédent

Quelles sont les responsabilités de la classe Adresse ?

- ❑ Représenter un concept métier

L'exemple précédent lui ajoute une responsabilité :

- ❑ Elle devient de facto une spécialiste des transformations chaine <-> date
- ❑ Dans ce cas est-ce que chaque classe métier devrait posséder cette méthode ?
- ❑ Comment font les autres classes qui ont besoin de convertir des chaines en dates ?

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Avantage d'une classe spécialisée

En créant une classe spécialisée dans les transformations chaines <-> dates:

- ❑ 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

```
class DateUtils {

    convertirEnDate(chaineDate) { // Supposons que moment.js est chargé const
                dateConvertie = moment(chaineDate);
                if (!dateConvertie.isValid()) {
          throw new Error(`Date invalide: ${chaineDate}`);
                }
      return dateConvertie;
    }
}
```


diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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**

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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.

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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 ?

```
class Service {
    traitement() {

        const individu = new Individu();

        const jambeDroite = new Jambe();
        const jambeGauche = new Jambe();

        individu.jambeGauche = jambeGauche;
        individu.jambeDroite = jambeDroite;
    }
}
```


diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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 a une association avec cette classe

```
class Service {
    void traitement() {

        const individu = new Individu();
    }
}
```

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


diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Créateur

- Il est préférable d'avoir recours à une classe spécialisée pour la construction d'un individu et qu'on appelle une Factory

```
class Service {

    traitement() {
        const individu = IndividuFactory.getInstance();
    }
}
```


diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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:

```
const valeur = client.compteBancaire.getOperation(date).montant;
```

- 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.

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


diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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.

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Chapitre 2
Introduction aux design patterns

diginamic
FORMATION
« Votre passeport pour l'emploi numérique »

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

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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'apparait le 1er livre de référence sur les techniques de conception.

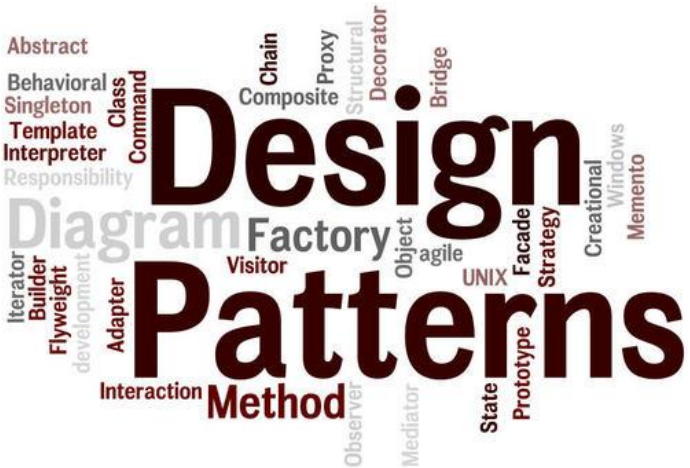
- Design Patterns – Elements of Reusable Object-Oriented Software - 1995
  - Gang of Four: Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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



diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Les types de design patterns

- 4 grands types de design patterns:
  - Architecturaux
  - Créationnels
  - Structurels
  - Comportementaux

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Chapitre 3
Patterns architecturaux

diginamic
FORMATION
« Votre passeport pour l'emploi numérique »

Les patterns architecturaux

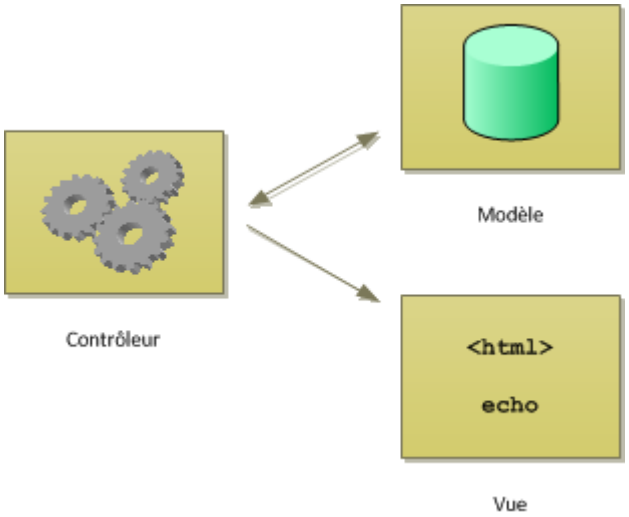
- Un pattern architecturel a un impact fort sur l'organisation générale du code au sein de votre application

- 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 persistence (exemple: DAO)*

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Le pattern MVC

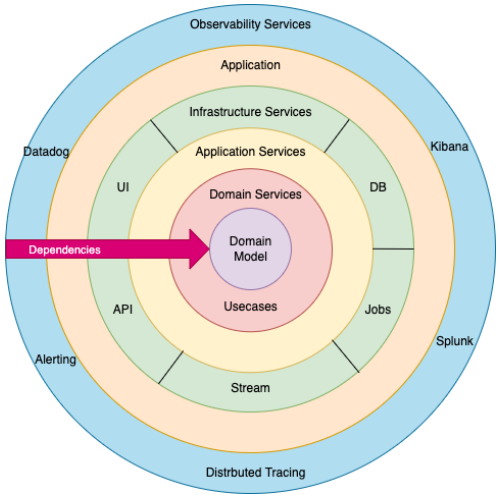
- MVC: Modèle, Vue, Contrôleur
- Dans ce pattern, la "Vue" représente l'IHM:
- Le modèle désigne les classes métier, exemple : Film, Acteur, Réalisateur, Role, etc.



diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Introduction pattern layered

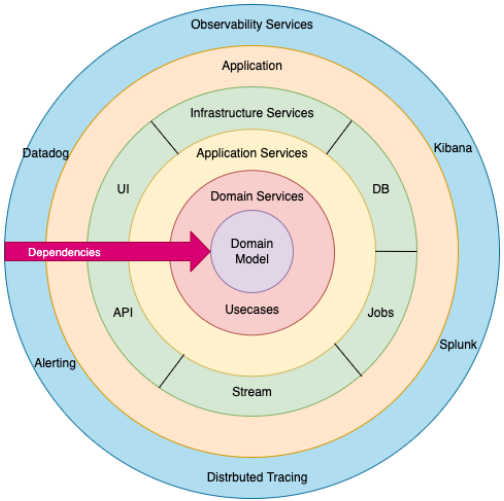
- Le découpage en couches permet de bien structurer son application
- Les avantages sont multiples : évolutivité, réutilisabilité, robustesse, testabilité
- L'architecture en couches modèle, ou en oignions :



diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Observability services

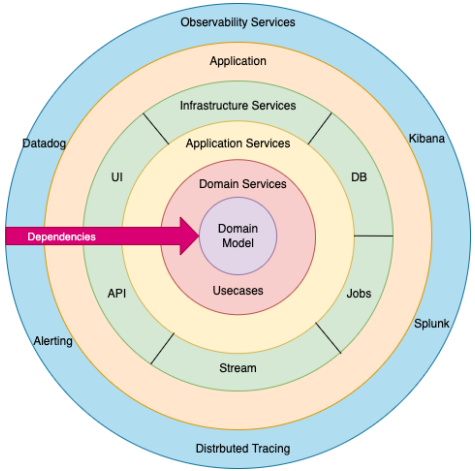
- C'est la couche responsable du **monitoring** de l'application.
- C'est une couche extérieure à l'application et constituée d'outils pour surveiller et alerter si besoin.
- Elle peut également contenir des outils pour analyser et comprendre des problèmes de vie courante (bugs, piratage, etc..)



diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Infrastructure services ou contrôleurs

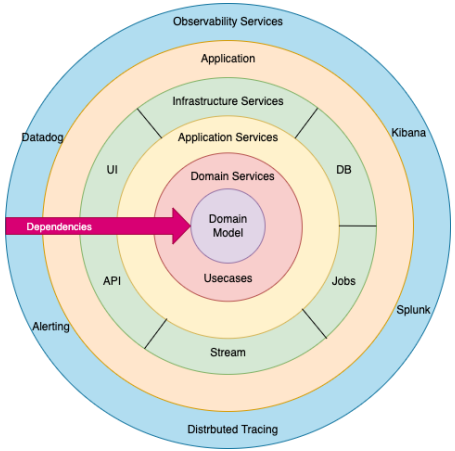
- C'est la couche à laquelle appartient les **contrôleurs** ou encore les **jobs**.
- On peut mettre dans cette couche également toutes les classes utilitaires.
- Un contrôleur est un point d'entrée dans l'application : (endpoint pour une API, classe dotée d'une méthode exécutable main, etc.)



diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Application services

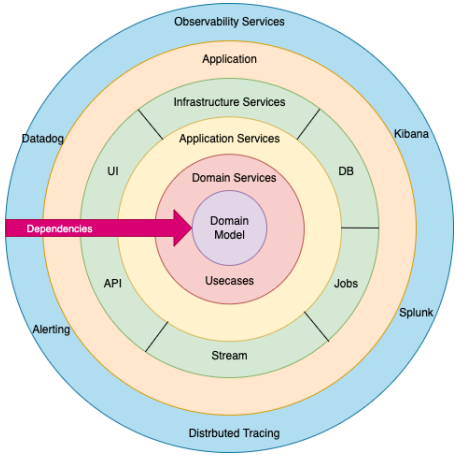
- C'est la couche responsable de la réalisation d'un cas d'utilisation, par exemple "créer un nouvel élève".
- Il y a une classe de type "Application services" par cas d'utilisation. Parfois on les appelle managers et peuvent s'appeler par exemple CreerEleveMgr.
- C'est ce type de classe qui orchestre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



diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Domain services

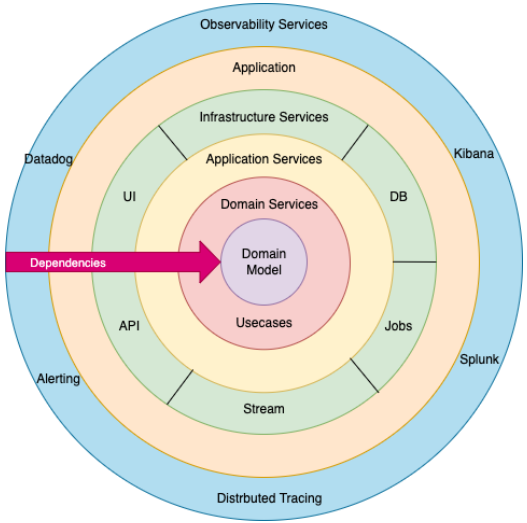
- C'est la couche métier de l'application.
- Elle contient les règles (attributs obligatoires par exemple) et algorithmes.
- En général il y a un service par entité métier : EleveService, ClasseService, BulletinService, etc.
- La couche **Application Services** utilise une ou plusieurs classes de type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



diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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.



diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Les échanges de données avec l'extérieur

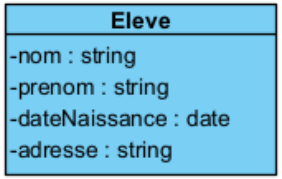
- Aujourd'hui les **échanges avec le front**, ou des **applications externes**, utilisent principalement **JSON** ou **XML**.
- Mais il peut arriver aussi qu'on utilise des fichiers plus basiques, type CSV ou même positionnels sur les vieilles applications.
- Dans une API classique les contrôleurs échangent des données au format JSON avec l'extérieur.
- 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 :



- 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 :

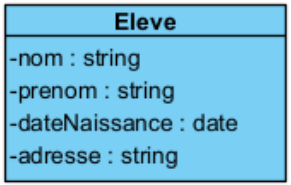
Nom	Prénom	Date de naissance	Age	Adresse
DOTENET	Camille	08/02/2011	13	4 rue des Lilas ...
PITON	Karim	15/10/2010	13	7 rue du Bois fleuri ...
GEAVAS	Bertrand	08/07/2010	13	1 rue du Marché ...

- 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 !

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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 ...

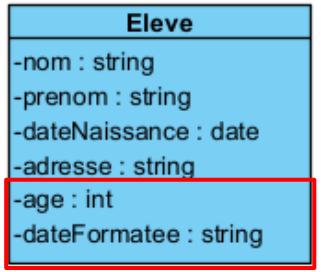


- Solution 1:
  - on envoie au front une liste d'élèves et le front se débrouille
- Problèmes:
  - Le front devient de facto un expert en calcul des âges, ce qui ne doit pas être une responsabilité du front
  - Le front devient également de facto un expert en formatage des dates. Pourquoi pas mais il n'a pas la connaissance qu'a le back sur les préférences de l'utilisateur par exemple.

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Exemple Eleve vs EleveModel

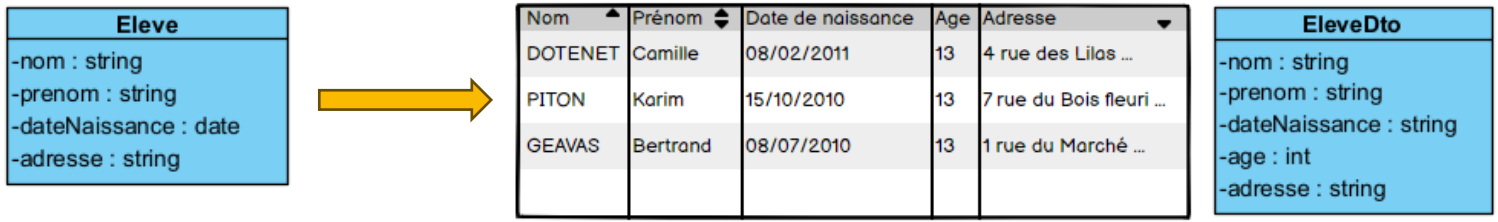
Nom	Prénom	Date de naissance	Age	Adresse
DOTENET	Camille	08/02/2011	13	4 rue des Lilas ...
PITON	Karim	15/10/2010	13	7 rue du Bois fleuri ...
GEAVAS	Bertrand	08/07/2010	13	1 rue du Marché ...



- 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.

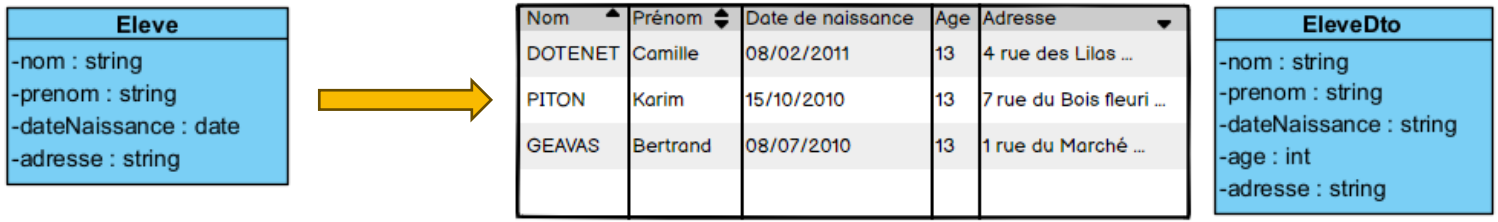
diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Exemple Eleve vs EleveDto



- Solution 3:
  - On créé une classe EleveModel, ou EleveDto qui va avoir exactement les données attendues par la vue.
- Problèmes:
  - Inflation du nombre de classes mais facile à faire évoluer selon les besoins des vues.
  - Il faut maintenir toute une zoologie de Dtos.

La solution à privilégier



- 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.

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Chapitre 4
Patterns créationnels

diginamic
FORMATION
« Votre passeport pour l'emploi numérique »

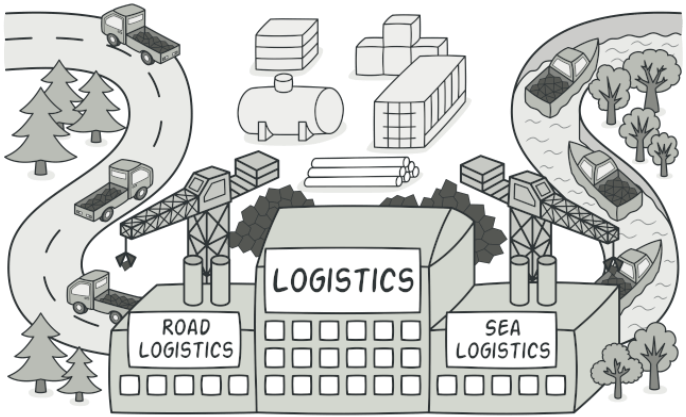
Les patterns créationnels

- Prennent en charge la création d'objets et l'instantiation:
  - **Factory Method**
  - Abstract Factory
  - **Builder**
  - Object Pool
  - Prototype
  - **Singleton**

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Factory method

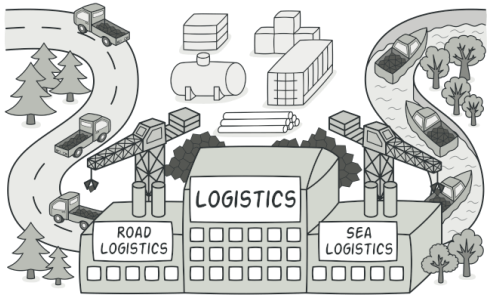
- Prend en charge l'instantiation d'objets d'une même hiérarchie (ex: Transport)
- Utilisation du polymorphisme: l'appelant ne connait pas le type concret de la classe retournée



diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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.)



diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Exemple

❑ Les classes

```
class Transport {
    constructor(nom) { this.nom = nom; }
    deplacer() {
         return `${this.nom} se déplace`;
    }
}
class Camion extends Transport {
    deplacer() {
        return `${this.nom} roule sur la route`;
    }
}
class Bateau extends Transport {
    deplacer() {
        return `${this.nom} navigue sur l'eau`;
    }
}
```


diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Exemple

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

```
class TransportFactory {
    static creer(type, nom) {
         switch(type) {
             case 'camion':
                return new Camion(nom);
             case 'bateau':
                return new Bateau(nom);
             default:
                return null;
        }
    }
}
```

```
// Utilisation
const transport1 = TransportFactory.creer('camion', 'Truck-01');
const transport2 = TransportFactory.creer('bateau', 'Ship-01');

// transport3 est null
const transport3 = TransportFactory.creer('avion', 'Plane-01');
```


diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Atelier (TP)

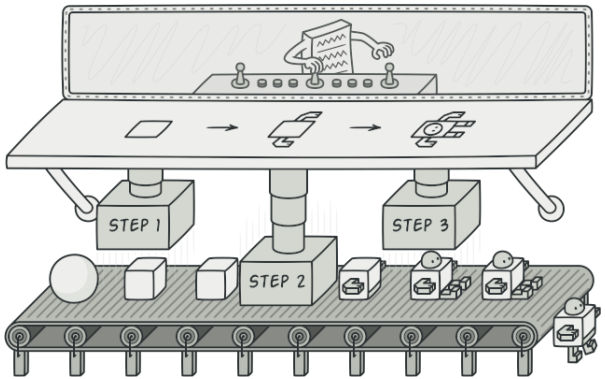
OBJECTIFS : Mettre en place le pattern Factory method

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.



diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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**.

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Builder - exemple

```
class ProduitBuilder {
  #nom;
  #ingredients;
  #marque;
  #categorie;
  #valeurNutritionnelle;

  constructor() {
    this.#ingredients = [];
    this.#valeurNutritionnelle = {};
  }

  setNom(nom) {
    this.#nom = nom;
    return this;
  }

  addIngredient(nom, quantite) {
    this.#ingredients.push({ nom, quantite });
    return this;
  }

  build() {
    return this;
  }
}
```

Créé l'instance de Produit

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

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.

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Builder - exemple

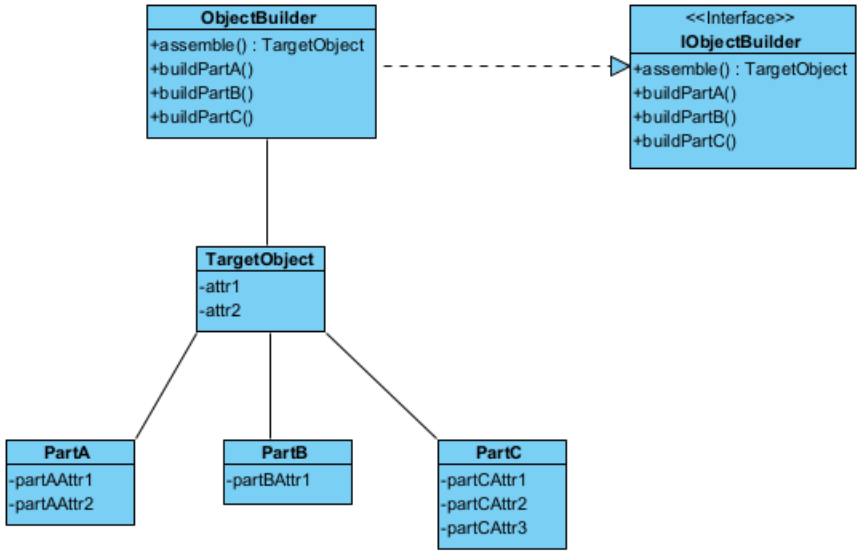
```
const produit = new ProduitBuilder()
  .setNom("Yaourt bio fraise")
  .addIngredient("Lait", "150ml")
  .addIngredient("Fraise", "20g")
  .setMarque("Ferme des Prés")
  .setCategorie("Produits laitiers")
  .setValeurNutritionnelle("Calories", "120 kcal")
  .build();
```

- Pattern fluent ou chainage de méthodes très courant dans les builders.

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

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.

diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Atelier (TP)

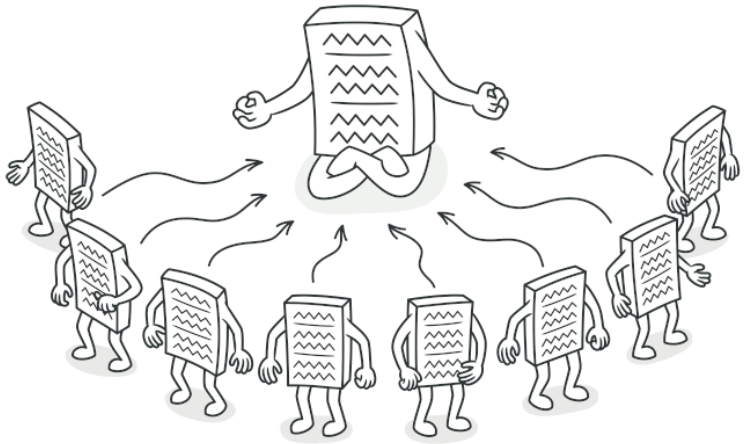
OBJECTIFS : Mettre en place le pattern **Builder**

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



diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Implémentation basique

❑ Exemple

```
class Singleton {
  static #instance = null;

  constructor() {
    if (Singleton.#instance) {
      throw new Error("Utilisez Singleton.getInstance() pour accéder à l'instance.");
    }
  }

  static getInstance() {
    if (!Singleton.#instance) {
      Singleton.#instance = new Singleton();
    }
    return Singleton.#instance;
  }

  direBonjour() {
    console.log("Bonjour depuis le singleton !");
  }
}
```


diginamic FORMATION
« Votre passeport pour l'emploi numérique »

Atelier (TP)

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

DESCRIPTION :

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