Découverte des patrons de conception

Table des matières

I Contexte

Durée : 120 minutes

Prérequis : aucun

Contexte

Les patrons de conception, ou *design patterns*, sont des solutions éprouvées et réutilisables pour résoudre des problèmes courants dans la conception de logiciels. Ils offrent un cadre de travail standardisé permettant aux développeurs de créer des applications efficaces et maintenables. En général, les patrons de conception permettent de bénéficier d'une structure modulaire, d'une cohésion élevée et d'un faible couplage, ce qui facilite l'évolutivité et la réutilisabilité du code. Dans le contexte de JavaScript, un langage de programmation dynamique et flexible, l'utilisation de patrons est particulièrement intéressante. JavaScript étant souvent utilisé pour le développement web, la mise en œuvre de ces patrons permet de créer des applications web robustes et évolutives. Par exemple, le patron Singleton garantit qu'une classe n'a qu'une seule instance, ce qui est utile pour gérer des ressources partagées. Ou alors, Observable facilite la communication entre les objets en les reliant *via* un mécanisme d'événements. Ainsi, l'utilisation de patrons de conception en JavaScript contribue à la qualité, la performance et la maintenabilité du code.

II Introduction aux patrons de conception en JavaScript

A. Introduction aux patrons de conception en JavaScript

Az Définition

Un patron de conception (*design pattern*) est une solution générale et réutilisable à un problème de conception récurrent dans le développement de logiciels. Un patron de conception décrit une approche éprouvée pour résoudre un problème spécifique de manière efficace et élégante. Ce ne sont pas des morceaux de code spécifiques que l'on peut simplement copier et coller dans un projet, mais plutôt des **modèles abstraits** qui décrivent la structure et le comportement des éléments d'un système logiciel. Ils fournissent des directives pour résoudre des problèmes de conception courants, tels que la gestion des dépendances, la gestion des exceptions, la création d'objets et la communication entre les composants.

Tout d'abord, les patrons de conception permettent une **réutilisation** de code. En effet, ils représentent des solutions éprouvées à des problèmes courants de conception logicielle, que les développeurs peuvent adapter à leurs propres besoins. De cette manière, ils évitent de réinventer la roue et de créer du code redondant, ce qui est une perte de temps et d'énergie.

Ensuite, les patrons de conception améliorent la **maintenabilité** du code. En fournissant une structure claire et cohérente aux projets, ils permettent de rendre le code plus facile à comprendre, à modifier et à maintenir. Cela est particulièrement important lorsque plusieurs développeurs travaillent sur un même projet, car cela permet à chacun de savoir où trouver les éléments nécessaires et comment les utiliser. Par conséquent, l'utilisation de patrons de conception peut aider à réduire les erreurs et les bogues dans le code.

Enfin, ils améliorent la **lisibilité** du code. Ils fournissent des noms clairs pour les éléments du code, ainsi que des structures de base, qui rendent le code plus facile à lire et à comprendre. Cela peut aider les développeurs à comprendre rapidement comment un élément du code est utilisé et comment il peut être modifié pour répondre à leurs besoins. En conséquence, les développeurs peuvent gagner du temps et de l'énergie en évitant de devoir déchiffrer du code complexe et difficile à lire.

Définition et utilisation du Singleton

Le Singleton est un patron de conception qui garantit la création d'**une seule instance d'une classe** dans une application. Il est utilisé pour contrôler l'accès à une ressource partagée, comme une base de données ou un fichier de configuration. L'idée derrière le Singleton est d'assurer qu'il n'y ait qu'une seule instance d'une classe, même si elle est instanciée plusieurs fois dans le code. En effet, le Singleton stocke l'instance de la classe créée la première fois, puis la réutilise pour toutes les instances suivantes. De cette façon, il évite de créer des instances supplémentaires qui pourraient entraîner des problèmes de performances ou de cohérence des données.

Le Singleton peut être implémenté de différentes manières, mais la méthode la plus courante consiste à utiliser une propriété statique privée qui stocke l'instance de la classe. La classe elle-même a une méthode publique statique qui permet d'obtenir cette instance. Si l'instance n'existe pas, elle est créée et stockée dans la propriété statique. Un exemple d'utilisation courante du Singleton est pour la gestion de la connexion à une base de données. En effet, il est important d'avoir une seule connexion ouverte à une base de données pour éviter les conflits de données et

les problèmes de performance. En utilisant un Singleton pour la gestion de la connexion à la base de données, les développeurs peuvent s'assurer qu'il n'y aura qu'une seule connexion ouverte à la fois.

Cependant, l'utilisation du Singleton peut avoir des inconvénients. En effet, il peut rendre le code plus difficile à tester et à maintenir, car l'instance unique est partagée entre tous les utilisateurs de la classe. De plus, il peut être difficile d'étendre une classe qui utilise un Singleton, car il n'est pas possible de créer des sous-classes qui ont des instances différentes de la classe mère.

Exemple

Voici un exemple d'implémentation du Singleton en JavaScript :

```
const Singleton = (() => {
  let instance;

  // Fonction qui crée l'instance unique de la classe
  function createInstance() {
    const object = new Object("Je suis l'instance unique !");
    return object;
  }

  // Méthode publique pour obtenir l'instance unique de la classe
  return {
    getInstance : function() {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    }
  };
})();

// Utilisation de la méthode publique pour obtenir l'instance unique
const instance1 = Singleton.getInstance();
const instance2 = Singleton.getInstance();

// Vérification que les deux instances sont identiques
console.log(instance1 === instance2); // true
```

Dans cet exemple, le Singleton est implémenté comme une fonction anonyme auto-exécutée, qui retourne un objet avec une méthode publique appelée `getInstance`. La méthode `getInstance` utilise une variable instance pour stocker l'instance unique de la classe. Si cette variable est nulle, la méthode `createInstance` est appelée pour créer une nouvelle instance de la classe. La méthode `createInstance` crée simplement un objet avec une chaîne de caractères qui indique qu'il s'agit de l'instance unique de la classe. Une fois que l'instance unique a été créée, elle est stockée dans la variable instance et retournée par la méthode `getInstance`. Enfin, dans l'exemple, 2 instances de la classe sont créées en appelant la méthode `getInstance` 2 fois, puis la comparaison entre ces 2 instances est effectuée à l'aide de l'opérateur `'==='`. Comme prévu, la comparaison renvoie « *true* », car les 2 instances sont identiques.

Cette implémentation utilise une propriété privée `let instance` qui ne peuvent pas être accédées depuis l'extérieur de la fonction anonyme auto-exécutée. De cette manière, l'instance unique de la classe est cachée à l'utilisateur de la classe, qui ne peut pas créer de nouvelles instances ou modifier l'instance existante.

Définition et utilisation de la Factory

La Factory est un patron de conception qui permet de créer des objets **sans spécifier leur classe exacte**. Elle fournit une interface commune pour créer différents types d'objets en utilisant une méthode de fabrication. Cette méthode décide quelle classe d'objets créer en fonction des paramètres fournis. L'idée derrière la Factory est de séparer la création d'objets de leur utilisation. En effet, au lieu de créer des objets directement dans le code, la Factory fournit une méthode pour créer des objets à partir d'une classe abstraite ou d'une interface commune. De cette façon, les développeurs peuvent créer des objets sans connaître les détails de leur implémentation.

La Factory peut être implémentée de différentes manières. La méthode la plus courante consiste à utiliser une fonction de fabrication, qui prend en compte les paramètres fournis pour décider quelle classe d'objets créer. Cette fonction retourne ensuite l'objet créé, qui peut être utilisé dans le code. Un exemple courant d'utilisation de la Factory est pour créer des objets qui partagent une même interface, mais qui ont des implémentations différentes. Par exemple, dans une application de traitement de paiements, la Factory peut être utilisée pour créer différents types de paiements, tels que les paiements par carte de crédit, les paiements par virement bancaire ou les paiements par PayPal. Chaque type de paiement peut avoir une implémentation différente, mais tous doivent partager une même interface, comme la méthode `processPayment`.

En utilisant une Factory pour créer les objets de paiement, les développeurs peuvent facilement ajouter de nouveaux types de paiement à l'application, sans avoir à modifier le code existant. Ils peuvent également ajouter de nouvelles implémentations pour chaque type de paiement, en s'assurant que tous les objets de paiement créés utilisent la même interface.

Cependant, l'utilisation de la Factory peut également avoir des inconvénients. En effet, elle peut ajouter une complexité supplémentaire au code, en introduisant une couche de création d'objets qui peut être difficile à comprendre. De plus, elle peut rendre le code plus difficile à tester et à maintenir, car elle peut créer des dépendances supplémentaires entre les objets de l'application.

Exemple

Voici un exemple d'implémentation de la Factory en JavaScript :

```
// Définition de l'interface commune pour les objets de paiement
class Payment {
  processPayment() {}
}

// Implémentation de la classe de paiement par carte de crédit
class CreditCardPayment extends Payment {
  processPayment() {
    console.log("Traitement du paiement par carte de crédit...");
  }
}

// Implémentation de la classe de paiement par virement bancaire
class BankTransferPayment extends Payment {
  processPayment() {
    console.log("Traitement du paiement par virement bancaire...");
  }
}

// Implémentation de la Factory pour créer les objets de paiement
class PaymentFactory {
  createPayment(paymentType) {
    if (paymentType === "CreditCard") {
      return new CreditCardPayment();
    } else if (paymentType === "BankTransfer") {
      return new BankTransferPayment();
    } else {
```

```
      throw new Error("Type de paiement invalide !");
    }
  }
}

// Utilisation de la Factory pour créer différents types de paiements
const paymentFactory = new PaymentFactory();
const creditCardPayment = paymentFactory.createPayment("CreditCard");
const bankTransferPayment = paymentFactory.createPayment("BankTransfer");

// Utilisation des objets de paiement créés
creditCardPayment.processPayment(); // Traitement du paiement par carte de crédit...
bankTransferPayment.processPayment(); // Traitement du paiement par virement bancaire...
```

Dans cet exemple, la Factory est implémentée sous forme d'une classe `PaymentFactory`. Cette classe a une méthode `createPayment`, qui prend en compte le type de paiement fourni en paramètre pour décider quelle classe d'objets de paiement créer. Dans cet exemple, 2 types de paiement sont disponibles : « *CreditCard* » et « *BankTransfer* ».

La Factory utilise ensuite une série de conditions pour décider quelle classe d'objets créer en fonction du type de paiement fourni en paramètre. Dans cet exemple, la Factory crée des instances de « *CreditCardPayment* » ou de « *BankTransferPayment* ».

Une fois les objets de paiement créés, ils peuvent être utilisés pour traiter les paiements en appelant la méthode `processPayment`. Cette méthode est implémentée de manière différente pour chaque classe d'objets de paiement, mais toutes doivent être conformes à l'interface commune définie par la classe Payment.

Définition et utilisation du Decorator

Le Decorator est un patron de conception qui permet de modifier le comportement d'un objet en lui ajoutant de nouvelles fonctionnalités à la volée, sans modifier son code source. Il permet également de séparer les préoccupations en ajoutant des fonctionnalités à des objets existants sans avoir à modifier leur code. L'idée derrière le Decorator est de créer une classe qui enveloppe une autre classe, en ajoutant des fonctionnalités supplémentaires à l'objet initial. Cette classe enveloppeur, appelée « *le décorateur* », implémente la même interface que la classe initiale, de sorte que les objets de la classe initiale peuvent être utilisés avec ou sans décorateurs. Le Decorator peut être implémenté de différentes manières. La méthode la plus courante consiste à créer une classe abstraite ou une interface commune pour les objets de la classe initiale et les décorateurs. Les classes décorateurs doivent implémenter cette interface commune et contenir une instance de la classe initiale. Un exemple courant d'utilisation du Decorator est pour ajouter des fonctionnalités supplémentaires à des objets graphiques, tels que des bordures, des ombres ou des motifs.

Dans cet exemple, la classe initiale serait la classe de base pour les objets graphiques, tels que les cercles, les carrés ou les triangles. Les décorateurs pourraient ensuite ajouter des fonctionnalités supplémentaires à ces objets, comme une bordure rouge ou une ombre noire. En utilisant le Decorator pour ajouter des fonctionnalités à des objets graphiques, il est plus facile de créer de nouveaux objets avec des fonctionnalités supplémentaires, sans avoir à créer de nouvelles classes d'objets graphiques. De plus, les décorateurs peuvent être combinés pour créer des objets avec des fonctionnalités multiples.

Exemple

Voici un exemple d'implémentation du Decorator en JavaScript :

```
// Définition de la classe initiale pour les objets graphiques
class Shape {
  draw() {
    console.log("Je suis une forme.");
  }
}

// Définition de la classe de décorateur pour ajouter une bordure
class BorderDecorator {
  constructor(shape) {
    this.shape = shape;
  }

  draw() {
    this.shape.draw();
    console.log("Je suis une bordure rouge.");
  }
}

// Définition de la classe de décorateur pour ajouter une ombre
class ShadowDecorator {
  constructor(shape) {
    this.shape = shape;
  }

  draw() {
    this.shape.draw();
    console.log("Je suis une ombre noire.");
  }
}

// Création d'un objet de la classe initiale
const shape = new Shape();

// Création d'un objet de la classe de décorateur pour ajouter une bordure
const shapeWithBorder = new BorderDecorator(shape);

// Création d'un objet de la classe de décorateur pour ajouter une ombre
const shapeWithShadow = new ShadowDecorator(shape);

// Création d'un objet de la classe de décorateur pour ajouter une bordure et une ombre
const shapeWithBorderAndShadow = new ShadowDecorator(new BorderDecorator(shape));

// Utilisation des différents objets créés
shape.draw(); // Je suis une forme.
shapeWithBorder.draw(); // Je suis une forme. Je suis une bordure rouge.
shapeWithShadow.draw(); // Je suis une forme. Je suis une ombre noire.
shapeWithBorderAndShadow.draw(); // Je suis une forme. Je suis une bordure rouge. Je suis une ombre noire.
```

Dans cet exemple, la classe initiale est la classe Shape, qui a une méthode `draw` qui affiche un message indiquant qu'il s'agit d'une forme. 2 classes décorateurs sont ensuite créées : « *BorderDecorator* » et « *ShadowDecorator* ». Chacune de ces classes prend une instance de la classe initiale dans son constructeur, stockée dans une propriété Shape. Chaque classe décorateur implémente également la méthode `draw`, qui appelle la méthode `draw` de la classe initiale stockée dans la propriété Shape, puis affiche un message supplémentaire pour

indiquer la fonctionnalité ajoutée. Ensuite, différents objets sont créés en utilisant des combinaisons de la classe initiale et des classes décorateurs. Chaque objet créé peut être utilisé pour afficher un message différent en appelant sa méthode `draw`.

Définition et utilisation du Composite

Le Composite est un patron de conception qui permet de traiter des collections d'objets de manière uniforme, comme s'il s'agissait d'un seul objet. Il permet également de simplifier la hiérarchie des objets en créant une structure arborescente.

L'idée derrière le Composite est de créer une classe commune pour les objets simples et les objets composés. Les objets simples ne peuvent pas avoir de sous-objets, tandis que les objets composés contiennent une collection d'objets simples et/ou d'autres objets composés. Les objets simples et composés implémentent la même interface, de sorte que les clients peuvent traiter les objets composés et les objets simples de la même manière. Le Composite peut être utilisé pour traiter des structures hiérarchiques, telles que des arbres ou des graphes, en simplifiant leur traitement. Il peut également être utilisé pour traiter des collections d'objets de manière uniforme, sans avoir à vérifier si chaque objet est simple ou composé. Un exemple courant d'utilisation du Composite est pour représenter des documents. Dans cet exemple, un document peut être un simple document texte, ou un document composé de plusieurs sous-documents, tels que des tableaux, des images ou des paragraphes. Le document Composite contiendra une collection de sous-documents, qui peuvent être eux-mêmes des documents simples ou composés.

Exemple

Voici un exemple d'implémentation du Composite en JavaScript :

```
// Définition de l'interface commune pour les objets simples et les objets composés
class DocumentComponent {
  constructor(name) {
    this.name = name;
  }

  add(component) {}
  remove(component) {}
  getChild(index) {}
  display() {}
}

// Définition de la classe pour les objets simples
class Paragraph extends DocumentComponent {
  display() {
    console.log("Je suis un paragraphe.");
  }
}

// Définition de la classe pour les objets composés
class Document extends DocumentComponent {
  constructor(name) {
    super(name);
    this.children = [];
  }

  add(component) {
    this.children.push(component);
  }

  remove(component) {
```

```
    const index = this.children.indexOf(component);
    if (index !== -1) {
      this.children.splice(index, 1);
    }
  }

  getChild(index) {
    return this.children[index];
  }

  display() {
    console.log(`Je suis le document ${this.name}.`);
    for (const child of this.children) {
      child.display();
    }
  }
}

// Création de différents objets simples et composés
const paragraph1 = new Paragraph("premier paragraphe");
const paragraph2 = new Paragraph("deuxième paragraphe");
const myDocument = new Document("document principal");
const subDocument = new Document("sous-document");

// Ajout des objets simples au sous-document
subDocument.add(paragraph1);
subDocument.add(paragraph2);

// Ajout du sous-document et d'un paragraphe au document principal
myDocument.add(subDocument);
myDocument.add(new Paragraph("dernier paragraphe"));

// Affichage du document principal, qui affiche tous les objets ajoutés à la hiérarchie
myDocument.display();
```

Dans cet exemple, une interface commune est définie pour les objets simples et les objets composés, appelée « *DocumentComponent* ». Cette interface contient 4 méthodes : `add`, `remove`, `getChild` et `display`. Les objets simples, tels que les paragraphes, étendent cette interface et implémentent la méthode `display`. Les objets composés, tels que les documents, étendent également cette interface et contiennent une collection d'objets simples et/ou d'autres objets composés. La classe Document représente un objet composé, qui contient une collection d'objets simples et/ou d'autres objets composés. Cette classe implémente les méthodes `add`, `remove`, `getChild` et `display`. Les méthodes `add` et `remove` permettent d'ajouter et de supprimer des objets de la collection, tandis que la méthode `getChild` permet d'obtenir un objet à partir de sa position dans la collection. La méthode `display` affiche le nom du document, puis appelle la méthode `display` de chaque objet de la collection. Ensuite, différents objets simples et composés sont créés et ajoutés à la hiérarchie en utilisant les méthodes `add` et `remove` de la classe Document. Les objets sont ensuite affichés en appelant la méthode `display` de l'objet principal, qui affiche tous les objets ajoutés à la hiérarchie.

Définition et utilisation de l'Adapter

L'Adapter est un patron de conception qui permet d'adapter une interface existante pour la rendre compatible avec une autre interface requise par le client. Il permet de faire communiquer des objets qui ne pourraient pas autrement en raison d'incompatibilités d'interfaces. Le but de l'Adapter est de convertir l'interface d'un objet en une autre interface, sans modifier le code source de l'objet. Il est donc utile lorsque le code source de l'objet ne peut pas être modifié, ou lorsque les clients nécessitent une interface différente de celle fournie par l'objet. Un exemple courant d'utilisation de

l'Adapter est pour adapter des bibliothèques tierces. Dans ce cas, l'Adapter peut être utilisé pour adapter l'interface de la bibliothèque tierce à l'interface requise par l'application. De cette manière, l'application peut utiliser la bibliothèque tierce sans avoir à modifier son code source. Un autre exemple d'utilisation de l'Adapter est pour adapter des objets d'un format de données à un autre. Dans ce cas, l'Adapter peut être utilisé pour adapter l'interface de l'objet source au format de données requis par le client.

Exemple

Voici un exemple d'implémentation de l'Adapter en JavaScript :

```
// Interface requise par le client
class Target {
  request() {}
}

// Objet existant avec une interface incompatible avec celle requise par le client
class Adaptee {
  specificRequest() {
    return "Requête spécifique de l'Adaptee";
  }
}

// Adapter qui adapte l'interface de l'Adaptee à celle requise par le client
class Adapter extends Target {
  constructor(adaptee) {
    super();
    this.adaptee = adaptee;
  }

  request() {
    const specificRequestResult = this.adaptee.specificRequest();
    return `Adapter: (TRANSLATED) ${specificRequestResult}`;
  }
}

// Utilisation du client avec l'Adapter
function clientCode(target) {
  console.log(target.request());
}

// Utilisation de l'Adaptee avec une interface incompatible avec celle requise par le client
const adaptee = new Adaptee();
console.log("Adaptee: ", adaptee.specificRequest());

// Utilisation de l'Adapter pour adapter l'interface de l'Adaptee à celle requise par le client
const adapter = new Adapter(adaptee);
console.log("Adapter : ", adapter.request());

// Utilisation du client avec l'Adapter
console.log("Client : ");
clientCode(adapter);
```

Dans cet exemple, une interface Target est définie pour représenter l'interface requise par le client. L'objet existant Adaptee a une interface incompatible avec celle requise par le client, et ne peut donc pas être utilisé directement. L'Adapter est créé pour adapter l'interface de l'Adaptee à celle requise par le client.

La classe Adapter hérite de la classe Target et contient une référence à l'objet Adaptee. Elle implémente la méthode `request` de l'interface Target, qui est appelée par le client. Lorsque cette méthode est appelée, l'Adapter appelle la méthode `specificRequest` de l'objet Adaptee et adapte sa réponse pour répondre à l'interface requise par le client.

Ensuite, le client utilise l'Adaptee avec son interface incompatible pour appeler la méthode `specificRequest`. Cette méthode retourne un résultat qui ne peut pas être utilisé directement par le client. Ensuite, l'Adapter est utilisé pour adapter l'interface de l'Adaptee à celle requise par le client. Le client peut alors utiliser l'Adapter pour appeler la méthode `request` de l'interface requise.

Les patrons de comportement

Les patrons de comportement sont un ensemble de patrons de conception qui permettent de gérer les interactions entre les objets et de définir le comportement des objets en réponse à différents événements ou situations. Ces patrons se concentrent sur le comportement des objets plutôt que sur leur structure ou leur création. Les patrons de comportement sont utilisés pour simplifier la gestion des interactions entre les objets et pour réduire la complexité du code. Ils permettent de définir des comportements réutilisables qui peuvent être appliqués à différents objets et dans différentes situations.

Voici une vidéo expliquant 2 de ces patrons, Strategy et Template :

Complément

Voici les codes pour Strategy et Template :

```
class Strategy {
  execute() {
    throw new Error('La méthode execute() doit être implémentée');
  }
}
class ConcreteStrategyA extends Strategy {
  execute() {
    return 'Résultat de la stratégie A';
  }
}

class ConcreteStrategyB extends Strategy {
  execute() {
    return 'Résultat de la stratégie B';
  }
}
class Context {
  constructor(strategy) {
    this.strategy = strategy;
  }

  setStrategy(strategy) {
    this.strategy = strategy;
  }

  executeStrategy() {
    return this.strategy.execute();
  }
}
const context = new Context(new ConcreteStrategyA());
console.log(context.executeStrategy()); // Résultat de la stratégie A

context.setStrategy(new ConcreteStrategyB());
```

```
console.log(context.executeStrategy()); // Résultat de la stratégie B
```

```
class AbstractClass {
  templateMethod() {
    this.operation1();
    this.operation2();
    // ...
  }

  operation1() {
    throw new Error('La méthode operation1() doit être implémentée');
  }

  operation2() {
    throw new Error('La méthode operation2() doit être implémentée');
  }
}
class ConcreteClassA extends AbstractClass {
  operation1() {
    console.log('Opération 1 de la classe A');
  }

  operation2() {
    console.log('Opération 2 de la classe A');
  }
}

class ConcreteClassB extends AbstractClass {
  operation1() {
    console.log('Opération 1 de la classe B');
  }

  operation2() {
    console.log('Opération 2 de la classe B');
  }
}
const concreteA = new ConcreteClassA();
concreteA.templateMethod();
// Output:
// Opération 1 de la classe A
// Opération 2 de la classe A

const concreteB = new ConcreteClassB();
concreteB.templateMethod();
// Output:
// Opération 1 de la classe B
// Opération 2 de la classe B
```

B. Exercice : Quiz

[solution n°1 p. 33]

Supposons que vous développiez une application de gestion de commandes pour un restaurant et que vous souhaitiez utiliser JavaScript et les patrons de conception pour structurer votre code. Pour cela, vous devez utiliser les notions de Singleton, Factory et Decorator que vous avez apprises.

Question 1

Le code ci-dessous est une classe Singleton appelée « *Menu* » qui contient un attribut « *plats* » sous forme d'objet avec les éléments suivants : « *pizza* » - 10, « *hamburger* » - 8 et « *salade* » - 6. Que va afficher la ligne `console.log(menu1 === menu2 );` ?

```
const Menu = (() => {
  let instance;


  function createInstance() {
    return  {
        pizza: 10,
        hamburger: 8,
        salade: 6,
      };
  }


  return {
    getInstance : function() {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    }
  };
})();


const menu1  = Menu.getInstance();
const menu2  = Menu.getInstance();
```

- True
- False
- Error

Question 2

Que fait le code ci-dessous ?

```
class Commande {
constructor() {
this.plats = {};
}
}

class CommandeFactory {
static creerCommande() {
return new Commande();
}
}

const commande = CommandeFactory.creerCommande();
```

- La variable commande ne stocke qu'une référence à la méthode `creerCommande` de la CommandeFactory
- Le code définit une classe CommandeFactory qui crée une nouvelle instance de la classe Commande à chaque fois que la méthode statique `creerCommande` est appelée

En appelant `CommandeFactory.creerCommande()`, cela crée une instance de la CommandeFactory

Question 3

Quel patron de conception est implémenté par la classe JavaScript CommandeWrapper dans le code suivant ?

```
class CommandeWrapper {
  constructor(commande) {
    this.commande = commande;
  }

  ajouterPlat(plat, quantite) {
    this.commande.plats[plat] = (this.commande.plats[plat] || 0) + quantite;
  }
}

const wrapper= new CommandeWrapper(commande);
wrapper.ajouterPlat("pizza", 2);
console.log(commande.plats);
```

Decorator

Strategy

Template

Question 4

Quel patron de conception est implémenté par le code JavaScript suivant ?

```
class Commande {
  constructor(choice) {
    this.choice = choice;
  }

  calculerTotal(plats) {
    return this.choice.calculerTotal(plats);
  }
}

class CommandNormal extends Commande {
  calculerTotal(plats) {
    let total = 0;
    for (const plat in plats) {
      total += plats[plat].prix * plats[plat].quantite;
    }
    return total;
  }
}

class CommandPromotion extends Commande {
  calculerTotal(plats) {
    let total = 0;
    for (const plat in plats) {
      total += (plats[plat].prix * plats[plat].quantite) * 0.9; // 10% de réduction
    }
```

```
    return total;
  }
}

const commandeNormal = new Commande(new CommandNormal ());
console.log(commandeNormal.calculerTotal({
  pizza: {prix: 10, quantite: 2},
  hamburger: {prix: 8, quantite: 1},
}));

const commandePromotion = new Commande(new CommandPromotion ());
console.log(commandePromotion.calculerTotal({
  pizza: {prix: 10, quantite: 2},
  hamburger: {prix: 8, quantite: 1},
}));
```

- Decorator
- Strategy
- Template

Question 5

Quel patron de conception est implémenté par le code JavaScript suivant ?

```
class Commande {
  constructor(plats) {
    this.plats = plats;
  }


  finaliserCommande() {
    this.choisirPlats();
    this.preparerPlats();
    this.servirPlats();
  }


  choisirPlats() { }
  preparerPlats() { }
  servirPlats() { }
}

class CommandeSurPlace extends Commande {
  choisirPlats() {
    console.log("Le client a choisi les plats suivants : ", this.plats);
  }
  preparerPlats() {
    console.log("La cuisine prépare les plats.");
  }
  servirPlats() {
    console.log("Les plats sont servis à la table du client.");
  }
}

class CommandeALivrer extends Commande {
  choisirPlats() {
    console.log("Le client a choisi les plats suivants pour la livraison : ",
this.plats);
```

```
  }
  preparerPlats() {
    console.log("La cuisine prépare les plats pour la livraison.");
  }
  servirPlats() {
    console.log("Les plats sont prêts pour la livraison.");
  }
}

const commandeSurPlace = new CommandeSurPlace(['pizza', 'hamburger']);
commandeSurPlace.finaliserCommande();

const commandeALivrer = new CommandeALivrer(['salade']);
commandeALivrer.finaliserCommande();
```

- Decorator
- Strategy
- Template

III Patrons de conception avancés en JavaScript

A. Patrons de conception avancés en JavaScript

MVC

Az Définition

Le MVC (Modèle-Vue-Contrôleur) est un patron de conception qui permet de séparer la logique de présentation, la logique métier et la logique de gestion des entrées utilisateur d'une application. Il permet de simplifier la conception et la maintenance d'applications en divisant le code en 3 parties distinctes : le Modèle, la Vue et le Contrôleur.

Le **Modèle** représente la logique métier de l'application et les données associées. Il contient les règles de validation et les opérations qui permettent de manipuler les données. Le Modèle peut être utilisé pour communiquer avec la base de données ou pour effectuer des opérations de calculs. La **Vue** représente la partie de l'interface utilisateur qui permet de visualiser les données. Elle est responsable de la présentation des données au client. La Vue est passive et ne contient aucun logique métier. Le **Contrôleur** représente la partie de l'application qui gère les entrées utilisateur et coordonne l'interaction entre la Vue et le Modèle. Il est responsable de l'envoi des données du Modèle à la Vue et de la réception des entrées utilisateur pour les traiter.

L'utilisation du MVC permet de simplifier la maintenance de l'application en séparant les différentes parties du code. Cela permet également de faciliter la collaboration entre les différents membres de l'équipe de développement, car chaque partie du code peut être modifiée sans affecter les autres parties. Lorsque l'utilisateur interagit avec l'application, le Contrôleur est chargé de recevoir et de traiter les entrées utilisateur. Il récupère les données du Modèle et les envoie à la Vue pour les afficher. Lorsque l'utilisateur modifie les données, le Contrôleur récupère ces modifications et les envoie au Modèle pour les mettre à jour.

Le MVC est largement utilisé dans le développement web, où il est souvent associé à des frameworks tels que Angular, React ou Vue.js. Ces frameworks fournissent des outils pour simplifier la mise en œuvre du MVC et pour automatiser certaines tâches, telles que la mise à jour de la Vue en réponse aux modifications du Modèle.

Exemple

Voici un exemple d'implémentation du MVC en JavaScript :

```
// Modèle
class Model {
  constructor() {
    this.data = [];
  }

  addData(item) {
    this.data.push(item);
  }

  getData() {
    return this.data;
  }
}

```

```
// Vue
class View {
  constructor() {}

  render(data) {
    console.log(`Données actuelles : ${JSON.stringify(data)}`);
  }
}

// Contrôleur
class Controller {
  constructor(model, view) {
    this.model = model;
    this.view = view;
  }

  addData(item) {
    this.model.addData(item);
    this.view.render(this.model.getData());
  }
}

// Utilisation du MVC
const model = new Model();
const view = new View();
const controller = new Controller(model, view);

// Ajout de données
controller.addData("Donnée 1");
controller.addData("Donnée 2");
```

Dans cet exemple, le Modèle est représenté par la classe Model, qui contient une propriété data pour stocker les données et 2 méthodes : `addData` pour ajouter des données et `getData` pour récupérer les données.

La Vue est représentée par la classe View, qui contient une méthode `render` pour afficher les données à l'utilisateur.

Le Contrôleur est représenté par la classe Controller, qui contient une référence au Modèle et à la Vue. Il contient également une méthode `addData` pour ajouter des données au Modèle et mettre à jour la Vue avec les nouvelles données.

Ensuite, le MVC est utilisé en créant une instance du Modèle, de la Vue et du Contrôleur, et en utilisant le Contrôleur pour ajouter des données au Modèle.

Lorsque `addData` est appelée sur le Contrôleur, elle ajoute les données au Modèle en appelant `addData` sur le Modèle. Ensuite, elle appelle `render` sur la Vue en passant les données mises à jour du Modèle. La Vue affiche ensuite les données à l'utilisateur.

Méthode

Voici une vidéo montrant comment utiliser le MVC :

Le patron Commande

Le patron de conception Command est un patron comportemental qui permet de séparer la logique de l'invocation d'une commande de la logique de sa mise en œuvre. Il encapsule une demande en tant qu'objet, ce qui permet de paramétrer des méthodes avec différentes demandes, d'encapsuler des demandes dans des objets pour les traiter à distance et de permettre l'annulation

des opérations. Le patron Command est composé de 4 éléments principaux : la commande elle-même, l'objet récepteur, l'invocateur et le client. La **commande** est l'objet qui contient la demande à exécuter et les paramètres nécessaires pour l'exécuter. L'objet **récepteur** est l'objet qui contient la logique métier qui sera exécutée par la commande. L'**invocateur** est l'objet qui invoque la commande pour exécuter la demande. Le **client** est l'objet qui crée la commande et spécifie l'objet récepteur et la méthode à invoquer. L'utilisation du patron Command permet de réduire le couplage entre les objets, de faciliter la gestion des opérations, de fournir des fonctionnalités d'annulation et de rétablissement, et de fournir une base pour les opérations transactionnelles.

Exemple

Voici un exemple d'implémentation de Command en JavaScript :

```
// Récepteur
class Receiver {


  action() {
    console.log("Action effectuée.");
  }
}

// Commande
class Command {
  constructor(receiver) {
    this.receiver = receiver;
  }

  execute() {
    console.log("Commande exécutée.");
    this.receiver.action();
  }

  undo() {
    console.log("Commande annulée.");
  }
}

// Invocateur
class Invoker {
  constructor() {
    this.commands = [];
  }

  setCommand(command) {
    this.commands.push(command);
  }

  executeCommands() {
    this.commands.forEach((command) => {
      command.execute();
    });
  }

  undoCommands() {
    this.commands.reverse().forEach((command) => {
      command.undo();
    });
  }
}

```

```
// Utilisation de Command
const receiver = new Receiver();
const command1 = new Command(receiver);
const command2 = new Command(receiver);
const invoker = new Invoker();

// Configuration des commandes pour l'invocateur
invoker.setCommand(command1);
invoker.setCommand(command2);

// Exécution des commandes
invoker.executeCommands();

// Annulation des commandes
invoker.undoCommands();
```

La classe Receiver représente l'objet qui exécute la commande. Elle contient une méthode action qui affiche simplement « *Action effectuée* ».

La classe Command encapsule une commande à exécuter. Elle prend une référence au Récepteur dans son constructeur et contient 2 méthodes : `execute` qui invoque la méthode action du Récepteur, et `undo` qui est appelée pour annuler la commande.

La classe Invoker contient une liste de commandes (this.commands) et 2 méthodes : `setCommand` pour ajouter des commandes à la liste et `executeCommands` pour exécuter toutes les commandes de la liste. Elle contient également une méthode `undoCommands` pour annuler toutes les commandes dans l'ordre inverse.

Ensuite, 3 objets sont créés : un objet receiver de la classe Receiver pour représenter l'objet qui exécute la commande, et 2 objets command1 et command2 de la classe Command pour encapsuler les commandes à exécuter.

Ensuite, un objet invoker de la classe Invoker est créé pour invoquer les commandes. Les 2 commandes sont ajoutées à l'invocateur à l'aide de la méthode `setCommand`, puis toutes les commandes sont exécutées à l'aide de la méthode `executeCommands`.

Enfin, toutes les commandes sont annulées dans l'ordre inverse à l'aide de la méthode `undoCommands`.

Iterator

Le patron de conception Iterator est un patron comportemental qui fournit une manière de parcourir une collection d'objets sans exposer la structure interne de la collection. Il définit une interface pour accéder aux éléments d'une collection de manière séquentielle, sans connaître la structure interne de la collection. Le patron Iterator est composé de 2 éléments principaux : l'agrégat et l'itérateur. **L'agrégat** est l'objet qui contient les éléments à parcourir, et **l'itérateur** est l'objet qui permet de parcourir les éléments de l'agrégat. L'itérateur contient des méthodes pour accéder à l'élément suivant et pour vérifier s'il reste des éléments à parcourir.

L'utilisation du patron Iterator permet de simplifier la logique de parcours des collections et de fournir une interface uniforme pour parcourir différents types de collections.

Exemple

Pour implémenter le patron de conception Iterator en JavaScript, nous avons besoin de 2 éléments principaux : l'agrégat et l'itérateur.

L'agrégat est l'objet qui contient les éléments à parcourir. Dans cet exemple, nous allons utiliser une classe PersonCollection pour représenter l'agrégat. Cette classe doit avoir une méthode `getIterator` qui retourne un objet itérateur de type PersonIterator.

```javascript
class PersonCollection {
  constructor() {
    this.persons = [];
  }

  addPerson(person) {
    this.persons.push(person);
  }

  getIterator() {
    return new PersonIterator(this.persons);
  }
}


class PersonIterator {
  constructor(persons) {
    this.index = 0;
    this.persons = persons;
  }

  hasNext() {
    return this.index < this.persons.length;
  }

  next() {
    const person = this.persons[this.index];
    this.index++;
    return person;
  }
}


class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}

const personCollection = new PersonCollection();
personCollection.addPerson(new Person("Alice", 25));
personCollection.addPerson(new Person("Bob", 30));
personCollection.addPerson(new Person("Charlie", 35));

const iterator = personCollection.getIterator();

while (iterator.hasNext()) {
  const person = iterator.next();
  console.log(person.name + ", " + person.age + " ans");
}
```

L'itérateur est l'objet qui permet de parcourir les éléments de l'agrégat. Dans cet exemple, nous utilisons une classe PersonIterator pour représenter l'itérateur. Cette classe doit avoir une référence à la collection PersonCollection, ainsi qu'une méthode `hasNext` pour vérifier s'il reste des éléments à parcourir, et une méthode `next` pour accéder à l'élément suivant de la collection.

Ensuite, nous créons une instance de PersonCollection, ajoutons 3 objets Person à la collection, et obtenons un itérateur pour parcourir la collection. Enfin, nous utilisons une boucle `while` pour parcourir la collection et afficher le nom et l'âge de chaque personne.

Memento

Le patron de conception Memento est un patron comportemental qui permet de capturer et de restaurer l'état interne d'un objet sans violer l'encapsulation. Le Memento lui-même est un objet qui stocke l'état de l'objet d'origine à un moment donné, et le Caretaker est l'objet qui gère la sauvegarde et la restauration de l'état à partir du Memento. La définition du Memento implique la création d'un objet qui stocke l'état interne d'un objet. Cet objet ne doit être accessible que par l'objet lui-même (d'où la nécessité de ne pas violer l'encapsulation). Le Memento doit fournir une méthode pour récupérer l'état précédent de l'objet. Le Caretaker est l'objet qui gère la sauvegarde et la restauration de l'état de l'objet à partir du Memento. Il stocke les Mementos dans une pile et fournit des méthodes pour sauvegarder l'état courant de l'objet, pour restaurer l'état de l'objet à partir d'un Memento précédemment sauvegardé, et pour annuler les sauvegardes précédentes.

Exemple

Pour implémenter le patron de conception Memento en JavaScript, nous avons besoin de 2 éléments principaux : l'objet Memento et l'objet Caretaker.

L'objet Memento est l'objet qui stocke l'état interne de l'objet d'origine à un moment donné. Dans cet exemple, nous allons utiliser une classe Memento pour représenter l'objet Memento. Cette classe doit avoir une méthode `getState` pour retourner l'état de l'objet, et une méthode `setState` pour restaurer l'état de l'objet. L'objet Caretaker est l'objet qui gère la sauvegarde et la restauration de l'état à partir du Memento. Dans cet exemple, nous allons utiliser une classe Caretaker pour représenter l'objet Caretaker. Cette classe doit avoir une référence à l'objet d'origine dont nous souhaitons sauvegarder l'état, ainsi qu'une pile de Mementos pour stocker les états précédents de l'objet. Ensuite, nous pouvons créer une instance de l'objet d'origine et une instance de l'objet Caretaker. Nous pouvons modifier l'état de l'objet d'origine et sauvegarder l'état courant en appelant la méthode `save` de l'objet Caretaker. Nous pouvons également annuler les modifications précédentes en appelant la méthode `undo` de l'objet Caretaker.

Dans cet exemple, nous créons une instance de l'objet d'origine (Originator) avec un état initial, et une instance de l'objet Caretaker (Caretaker) avec une référence à l'objet d'origine. Ensuite, nous modifions l'état de l'objet d'origine, sauvegardons l'état courant en appelant la méthode `save` de l'objet Caretaker, et affichons l'état actuel de l'objet d'origine.

```
class Memento {
  constructor(state) {
    this.state = state;
  }

  getState() {
    return this.state;
  }

  setState(state) {
    this.state = state;
  }
}

class Caretaker {
  constructor(originator) {
    this.originator = originator;
    this.mementos = [];
  }

  save() {
    const state = this.originator.getState();
    const memento = new Memento(state);
    this.mementos.push(memento);
```

```
  }

  undo() {
    const memento = this.mementos.pop();
    this.originator.setState(memento.getState());
  }
}

class Originator {
  constructor(state) {
    this.state = state;
  }

  getState() {
    return this.state;
  }

  setState(state) {
    this.state = state;
  }
}

const originator = new Originator("état initial");
const caretaker = new Caretaker(originator);

console.log(originator.getState());  // affiche "état initial"

caretaker.save();
originator.setState("état modifié");
caretaker.save();

console.log(originator.getState()); // affiche "état modifié"

caretaker.undo();
console.log(originator.getState()); // affiche "état initial"
```

Observable

RxJS est une bibliothèque JavaScript qui implémente le patron de conception Observable. Observable est un patron de conception qui permet de gérer les événements asynchrones en utilisant une approche de programmation réactive.

L'objet Observable émet des valeurs asynchrones dans le temps, et permet à des observateurs de souscrire à ces valeurs. Lorsque l'Observable émet une valeur, tous les observateurs qui ont souscrit reçoivent cette valeur.

La bibliothèque RxJS fournit une implémentation complète du patron de conception Observable, avec de nombreuses méthodes pour créer, combiner et manipuler des Observables.

Exemple

Voici un exemple d'utilisation de RxJS pour créer un Observable qui émet des nombres aléatoires toutes les secondes :

```
const { Observable } = rxjs;
const observable = new Observable((observer) => {
  const intervalId = setInterval(() => {
    const random = Math.random();
    observer.next(random);
  }, 1000);

```

```
    return () => {
      clearInterval(intervalId);
    };
  });

  observable.subscribe((value) => {
    console.log(value);
  });
```

Dans cet exemple, nous créons un Observable qui émet des nombres aléatoires toutes les secondes. Nous utilisons la méthode `Observable` de RxJS pour créer l'Observable, et nous passons une fonction qui définit la logique de l'Observable. Dans cette fonction, nous utilisons la méthode `setInterval` pour émettre des nombres aléatoires toutes les secondes, et la méthode `observer.next` pour émettre chaque nombre aléatoire à tous les observateurs qui ont souscrit.

Nous utilisons également la valeur de retour de la fonction pour nettoyer les ressources lorsque l'Observable est terminé, en appelant la méthode `clearInterval` pour arrêter l'émission de nombres aléatoires. Ensuite, nous utilisons la méthode `subscribe` de l'Observable pour souscrire à l'Observable et recevoir les valeurs émises. Nous passons une fonction qui définit ce qui doit être fait avec chaque valeur émise. Dans cet exemple, nous affichons simplement chaque valeur dans la console.

Complément

Voici une vidéo expliquant comment installer et utiliser RxJs :

Voici le code de la vidéo :

```
import { Observable } from 'rxjs';

const observable = new Observable((observer) => {
  observer.next('Hello');
  observer.next('World');
  observer.complete();
});

observable.subscribe((value) => console.log(value));
```

```
import { Observable, interval } from 'rxjs';
import { map, filter, take } from 'rxjs/operators';

// Crée un Observable qui émet des nombres entiers à intervalles réguliers
const source = interval(1000);

// Applique les opérateurs map, filter et take à l'Observable
const transformed = source.pipe(
  map((value) => value * 2),
  filter((value) => value % 3 === 0),
  take(5)
);

// Affiche les valeurs émises par l'Observable transformé
transformed.subscribe((value) => console.log(value));
```

BehaviourSubject

Un BehaviorSubject est un type spécial d'Observable provenant de la bibliothèque RxJS. Il est particulièrement utile lorsque nous avons besoin de partager une valeur qui peut changer au fil du temps entre plusieurs observateurs.

Un BehaviorSubject a 2 principales caractéristiques :

- Il stocke la dernière valeur émise à ses observateurs et la transmet immédiatement à tout nouvel observateur qui s'abonne.
- Il permet de mettre à jour cette valeur et de l'émettre à tous les observateurs actifs.

Cela le rend particulièrement utile pour les scénarios où nous avons besoin de partager une valeur qui peut changer au fil du temps, comme un état partagé entre plusieurs composants dans une application.

```
import { BehaviorSubject } from 'rxjs';

// Crée un BehaviorSubject avec une valeur initiale
const subject = new BehaviorSubject('Hello');

// Les observateurs peuvent s'abonner au BehaviorSubject
const observer1 = {
  next: value => console.log('Observer 1:', value),
};

const observer2 = {
  next: value => console.log('Observer 2:', value),
};

// Lorsqu'un observateur s'abonne, il reçoit immédiatement la dernière valeur émise
subject.subscribe(observer1); // Affiche "Observer 1: Hello"

// La valeur du BehaviorSubject peut être mise à jour
subject.next('World');

// Les observateurs actifs reçoivent la nouvelle valeur
// Affiche "Observer 1: World"

// Les nouveaux observateurs reçoivent également la dernière valeur émise lorsqu'ils s'abonnent
subject.subscribe(observer2); // Affiche "Observer 2: World"
```

Dans cet exemple, nous créons un BehaviorSubject avec une valeur initiale 'Hello'. Lorsque le premier observateur s'abonne, il reçoit immédiatement cette valeur initiale. Ensuite, nous mettons à jour la valeur du BehaviorSubject avec 'World' et le premier observateur reçoit cette nouvelle valeur. Lorsque le deuxième observateur s'abonne, il reçoit également la dernière valeur émise, qui est 'World'.

From

La fonction `from` est un opérateur de création qui permet de convertir divers types de données en Observable. La fonction `from` peut accepter divers types de données en entrée, notamment les tableaux, les chaînes de caractères, les Promesses, les objets itérables (comme les Set ou les Map), et les objets Observable-like. Lorsque `from` reçoit un de ces types de données, il crée un Observable qui émet les éléments de la collection un par un.

```
import { from } from 'rxjs';

// Crée un tableau
const items = [1, 2, 3, 4, 5];

// Crée un Observable à partir du tableau
const observable = from(items);

// Un observateur qui affiche les valeurs émises
const observer = {
  next: value => console.log(value),
  error : err => console.error(err),
  complete: () => console.log('Terminé'),
```

```
};

// S'abonne à l'Observable et affiche les valeurs émises
observable.subscribe(observer);
// Affiche :
// 1
// 2
// 3
// 4
// 5
// Terminé
```

B. Exercice : Quiz

[solution n°2 p. 37]

Question 1

Quel est le rôle de la classe 'Model' dans l'architecture MVC (Modèle-Vue-Contrôleur) du code JavaScript suivant ?

```
class Model {
constructor() {
this.messages = [];
}

ajouterMessage(message) {
this.messages.push(message);
}
}
```

La classe 'Model' sert de médiateur entre la 'Vue' et le 'Contrôleur', facilitant ainsi la communication entre eux

La classe 'Model' représente les données de l'application

La classe 'Model' est responsable de la mise à jour de la 'Vue' chaque fois que les données du modèle changent

Question 2

Quel est le rôle de la classe 'View' dans l'architecture MVC (Modèle-Vue-Contrôleur) du code JavaScript suivant ?

```
class View {
afficherMessage(message) {
console.log(message);
}
}
```

La classe 'View' est responsable du stockage et de la manipulation des données de l'application

La classe 'View' représente la sortie de l'application, c'est-à-dire comment les données sont présentées à l'utilisateur

La classe 'View' sert de lien entre l'utilisateur et le système, traitant les entrées de l'utilisateur et transmettant les commandes au 'Contrôleur'

Question 3

Quel est le rôle de la classe 'Controller' dans l'architecture MVC (Modèle-Vue-Contrôleur) du code JavaScript suivant ?

```
class Controller {
constructor(model, view) {
this.model = model;
this.view = view;
}

nouveauMessage(message) {
this.model.ajouterMessage(message);
this.view.afficherMessage(message);
}
}
```

- La classe 'Controller' agit comme une interface pour les utilisateurs interagir avec l'application, et est responsable de la coordination des actions entre le 'Model' et la 'View'
- La classe 'Controller' est principalement utilisée pour stocker et gérer les données de l'application
- La classe 'Controller' est utilisée pour présenter les données de l'application à l'utilisateur

Question 4

Que fait le code JavaScript suivant en utilisant la bibliothèque RxJS ?

```
import { from } from 'rxjs';

const messages = ['Bonjour', 'Comment ça va ?', 'Au revoir'];
const observable = from(messages);
```

- Le code crée un 'observable' à partir de la bibliothèque RxJS qui émettra des messages une fois que quelque chose s'y sera abonné
- Le code crée une nouvelle instance de la classe 'rxjs' et stocke un tableau de messages dans celle-ci
- Le code transforme chaque message du tableau 'messages' en un nouvel 'observable' séparé de la bibliothèque RxJS

Question 5

Que fait le code suivant (il réutilise le code des questions précédentes) ?

```
const model = new Model();
const view = new View();
const controller = new Controller(model, view);

const observer = {
next: message => controller.nouveauMessage(message),
error: err => console.error(err),
complete: () => console.log('Terminé')
};

```

```
observable.subscribe(observer);
```

Le code crée des instances des classes Model, View et Controller, et souscrit ensuite à un observable qui, à chaque émission, envoie un nouveau message au Controller pour être traité et affiché

Le code crée une nouvelle instance du modèle et s'abonne à un observable qui, lorsqu'il émet une valeur, envoie cette valeur à la vue pour être affichée

Le code crée un observable et s'abonne à celui-ci. Chaque fois que l'observable émet une valeur, cette valeur est stockée dans le modèle

IV Essentiel

Les patrons de conception sont des modèles de conception utilisés pour résoudre des problèmes courants dans la programmation. Les patrons que nous avons étudiés sont classés en 3 catégories : les patrons de création, de structure et de comportement. Les patrons de création sont utilisés pour instancier des objets d'une manière contrôlée, en utilisant des classes abstraites et des méthodes de création. Nous avons étudié ces patrons : Singleton et Factory. Les patrons de structure sont utilisés pour organiser les objets d'une manière particulière, en utilisant des classes et des interfaces. Nous avons étudié 3 patrons de structure : Decorator, Composite et Adapter. Les patrons de comportement sont utilisés pour gérer les interactions entre les objets et les algorithmes, en utilisant des classes et des interfaces. Nous avons étudié 2 patrons de comportement : Strategy et Template. MVC est utilisé pour structurer une application en séparant la logique métier, la présentation et le contrôle, tandis que Observable est utilisé pour créer des objets qui notifient automatiquement les observateurs lorsqu'ils sont modifiés. RxJS est une bibliothèque JavaScript qui implémente le patron de conception Observable pour gérer les événements asynchrones de manière réactive. Observable est utilisé pour émettre des valeurs asynchrones dans le temps et permettre à des observateurs de souscrire à ces valeurs.

V Auto-évaluation

A. Exercice

Vous travaillez en tant que développeur pour une entreprise de gestion de projets. On vous demande de créer un système qui permet de gérer les différentes tâches d'un projet, en utilisant les patrons de conception appris dans les 2 parties précédentes. Les tâches sont représentées par des objets qui ont un titre, une description, un statut et une liste d'observateurs.

Question 1

[solution n°3 p. 40]

Utilisez le patron Singleton pour créer une classe « *Projet* » qui contient une liste de tâches. La classe doit avoir des méthodes pour ajouter des tâches, supprimer des tâches et récupérer une tâche par son titre.

Question 2

[solution n°4 p. 41]

Créez une classe « *Tache* » qui a un titre, une description et un statut (par exemple, « *en cours* », « *terminée* » ou « *en attente* »).

Question 3

[solution n°5 p. 41]

Utilisez RxJS pour créer un Observable qui émet des notifications lorsque le statut d'une tâche change. Les notifications doivent afficher le titre et le nouveau statut de la tâche.

Question 4

[solution n°6 p. 42]

Utilisez les classes « *Projet* », « *Tache* » et l'Observable pour créer un projet contenant plusieurs tâches. Changez le statut de certaines tâches et vérifiez que les notifications sont correctement affichées.

B. Test

Exercice 1 : Quiz

[solution n°7 p. 42]

Question 1

Quelle est la différence entre le patron de conception Singleton et le patron de conception Factory ?

- Le Singleton assure qu'une classe n'ait qu'une seule instance, tandis que le Factory crée des instances d'objets sans spécifier leurs classes concrètes
- Le Singleton crée des instances d'objets sans spécifier leurs classes concrètes, tandis que le Factory assure qu'une classe n'ait qu'une seule instance
- Le Singleton et le Factory sont identiques et servent à créer des instances d'objets

Question 2

Dans le patron de conception MVC, à quoi sert le contrôleur ?

- Il gère les interactions entre la vue et le modèle
- Il affiche les données à l'utilisateur

Il stocke et gère les données de l'application

Question 3

Quel est l'objectif principal du patron de conception Memento dans la programmation orientée objet ?

Faciliter la communication et les interactions entre les objets

Simplifier la création d'objets en déléguant cette responsabilité aux sous-classes

Permettre la sauvegarde et la restauration de l'état d'un objet sans violer son encapsulation

Question 4

Qu'est-ce qu'un Iterator dans le patron de conception Iterator ?

Un objet qui permet d'accéder aux éléments d'un conteneur sans exposer sa représentation interne

Un objet qui reçoit des notifications lorsqu'un autre objet change d'état

Un objet qui crée des instances d'objets sans spécifier leurs classes concrètes

Question 5

Quel est l'objectif du patron de conception Observable avec RxJS ?

Assurer qu'une classe n'ait qu'une seule instance

Permettre à un objet de notifier automatiquement les autres objets de ses changements d'état en utilisant des streams de données

Fournir une interface standard pour créer des objets dans une superclasse

Solutions des exercices

Solution n°1

[exercice p. 13]

Question 1

Le code ci-dessous est une classe Singleton appelée « *Menu* » qui contient un attribut « *plats* » sous forme d'objet avec les éléments suivants : « *pizza* » - 10, « *hamburger* » - 8 et « *salade* » - 6. Que va afficher la ligne `console.log(menu1 === menu2 );` ?

```
const Menu = (() => {
  let instance;


  function createInstance() {
    return  {
        pizza: 10,
        hamburger: 8,
        salade: 6,
      };
  }


  return {
    getInstance : function() {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    }
  };
})();


const menu1  = Menu.getInstance();
const menu2  = Menu.getInstance();
```

- ✓ True
- False
- Error

Le patron Singleton assure que l'instance d'une classe est unique. Donc menu1 est bien égal à menu2.

Question 2

Que fait le code ci-dessous ?

```
class Commande {
constructor() {
this.plats = {};
}
}

class CommandeFactory {
static creerCommande() {
return new Commande();
}
}

const commande = CommandeFactory.creerCommande();
```

La variable commande ne stocke qu'une référence à la méthode `creerCommande` de la CommandeFactory

✓ Le code définit une classe CommandeFactory qui crée une nouvelle instance de la classe Commande à chaque fois que la méthode statique `creerCommande` est appelée

En appelant `CommandeFactory.creerCommande()`, cela crée une instance de la CommandeFactory

Ce code est un exemple du patron de conception Factory, qui fournit une interface pour créer des objets dans une classe mère, mais permet aux classes filles de modifier les types d'objets qui seront créés.

Question 3

Quel patron de conception est implémenté par la classe JavaScript CommandeWrapper dans le code suivant ?

```
class CommandeWrapper {
  constructor(commande) {
    this.commande = commande;
  }

  ajouterPlat(plat, quantite) {
    this.commande.plats[plat] = (this.commande.plats[plat] || 0) + quantite;
  }
}

const wrapper= new CommandeWrapper(commande);
wrapper.ajouterPlat("pizza", 2);
console.log(commande.plats);
```

✓ Decorator

Strategy

Template

Le patron Decorator, qui permet d'ajouter ou de modifier le comportement d'un objet à la volée sans affecter le comportement d'autres objets de la même classe. Dans cet exemple, la classe CommandeWrapper est un décorateur qui ajoute une méthode `ajouterPlat` à une instance de la classe Commande, lui permettant ainsi d'ajouter des plats à la commande.

Question 4

Quel patron de conception est implémenté par le code JavaScript suivant ?

```
class Commande {
  constructor(choice) {
    this.choice = choice;
  }

  calculerTotal(plats) {
    return this.choice.calculerTotal(plats);
  }
}

class CommandNormal extends Commande {
  calculerTotal(plats) {
    let total = 0;
    for (const plat in plats) {
      total += plats[plat].prix * plats[plat].quantite;
    }
    return total;
  }
}

class CommandPromotion extends Commande {
  calculerTotal(plats) {
    let total = 0;
    for (const plat in plats) {
      total += (plats[plat].prix * plats[plat].quantite) * 0.9; // 10% de réduction
    }
    return total;
  }
}

const commandeNormal = new Commande(new CommandNormal ());
console.log(commandeNormal.calculerTotal({
  pizza: {prix: 10, quantite: 2},
  hamburger: {prix: 8, quantite: 1},
}));

const commandePromotion = new Commande(new CommandPromotion ());
console.log(commandePromotion.calculerTotal({
  pizza: {prix: 10, quantite: 2},
  hamburger: {prix: 8, quantite: 1},
}));
```

Decorator

✓ Strategy

Template

Le patron Strategy, qui définit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables. Strategy permet aux algorithmes de varier indépendamment des clients qui les utilisent. Dans ce code, la classe Commande utilise une Strategy pour calculer le total de la commande. 2 stratégies différentes sont définies : Normal et Promotion, qui peuvent être utilisées indépendamment pour calculer le total de la commande.

Question 5

Quel patron de conception est implémenté par le code JavaScript suivant ?

```
class Commande {
  constructor(plats) {
    this.plats = plats;
  }


  finaliserCommande() {
    this.choisirPlats();
    this.preparerPlats();
    this.servirPlats();
  }


  choisirPlats() { }
  preparerPlats() { }
  servirPlats() { }
}

class CommandeSurPlace extends Commande {
  choisirPlats() {
    console.log("Le client a choisi les plats suivants : ", this.plats);
  }
  preparerPlats() {
    console.log("La cuisine prépare les plats.");
  }
  servirPlats() {
    console.log("Les plats sont servis à la table du client.");
  }
}

class CommandeALivrer extends Commande {
  choisirPlats() {
    console.log("Le client a choisi les plats suivants pour la livraison : ",
this.plats);
  }
  preparerPlats() {
    console.log("La cuisine prépare les plats pour la livraison.");
  }
  servirPlats() {
    console.log("Les plats sont prêts pour la livraison.");
  }
}

const commandeSurPlace = new CommandeSurPlace(['pizza', 'hamburger']);
commandeSurPlace.finaliserCommande();

const commandeALivrer = new CommandeALivrer(['salade']);
commandeALivrer.finaliserCommande();
```

Decorator

Strategy

✓ Template

Le patron Template, qui définit le squelette d'un algorithme dans une méthode de superclasse, délègue certaines étapes aux sous-classes. Les sous-classes peuvent redéfinir certaines étapes de l'algorithme sans changer sa structure générale. Dans le code, Commande est une superclasse qui a un algorithme de base pour finaliser une commande. Les classes CommandeSurPlace et CommandeALivrer sont des sous-classes qui adaptent ce algorithme à des contextes spécifiques.

Solution n°2

[exercice p. 27]

Question 1

Quel est le rôle de la classe 'Model' dans l'architecture MVC (Modèle-Vue-Contrôleur) du code JavaScript suivant ?

```
class Model {
constructor() {
this.messages = [];
}

ajouterMessage(message) {
this.messages.push(message);
}
}
```

La classe 'Model' sert de médiateur entre la 'Vue' et le 'Contrôleur', facilitant ainsi la communication entre eux

✓ La classe 'Model' représente les données de l'application

La classe 'Model' est responsable de la mise à jour de la 'Vue' chaque fois que les données du modèle changent

La classe 'Model' représente les données de l'application, ici une liste de messages, et définit la logique pour accéder et manipuler ces données. Dans l'architecture MVC, le 'Model' contient la logique métier et les données de l'application. Il est responsable de l'accès aux données et de leur stockage, ainsi que de la logique permettant de manipuler ces données. Dans le code donné, la classe 'Model' a une propriété 'messages' pour stocker les messages et une méthode 'ajouterMessage' pour ajouter un nouveau message à la liste.

Question 2

Quel est le rôle de la classe 'View' dans l'architecture MVC (Modèle-Vue-Contrôleur) du code JavaScript suivant ?

```
class View {
afficherMessage(message) {
console.log(message);
}
}
```

La classe 'View' est responsable du stockage et de la manipulation des données de l'application

✓ La classe 'View' représente la sortie de l'application, c'est-à-dire comment les données sont présentées à l'utilisateur

La classe 'View' sert de lien entre l'utilisateur et le système, traitant les entrées de l'utilisateur et transmettant les commandes au 'Contrôleur'

Dans l'architecture MVC, 'View' est responsable de l'affichage des données à l'utilisateur. Il s'agit essentiellement de la représentation visuelle des données du 'Model'. Dans le code donné, la classe 'View' a une méthode 'afficherMessage' qui affiche le message à l'utilisateur.

Question 3

Quel est le rôle de la classe 'Controller' dans l'architecture MVC (Modèle-Vue-Contrôleur) du code JavaScript suivant ?

```
class Controller {
constructor(model, view) {
this.model = model;
this.view = view;
}

nouveauMessage(message) {
this.model.ajouterMessage(message);
this.view.afficherMessage(message);
}
}
```

✓ La classe 'Controller' agit comme une interface pour les utilisateurs interagir avec l'application, et est responsable de la coordination des actions entre le 'Model' et la 'View'

La classe 'Controller' est principalement utilisée pour stocker et gérer les données de l'application

La classe 'Controller' est utilisée pour présenter les données de l'application à l'utilisateur

Dans l'architecture MVC, le 'Controller' accepte les entrées et convertit celles-ci en commandes pour le 'Model' ou la 'View'. Dans le code donné, la classe 'Controller' a une méthode 'nouveauMessage' qui prend un message en entrée, met à jour le 'Model' en ajoutant le nouveau message à la liste des messages, puis utilise la 'View' pour afficher le nouveau message à l'utilisateur.

Question 4

Que fait le code JavaScript suivant en utilisant la bibliothèque RxJS ?

```
import { from } from 'rxjs';

const messages = ['Bonjour', 'Comment ça va ?', 'Au revoir'];
const observable = from(messages);
```

✓ Le code crée un 'observable' à partir de la bibliothèque RxJS qui émettra des messages une fois que quelque chose s'y sera abonné

Le code crée une nouvelle instance de la classe 'rxjs' et stocke un tableau de messages dans celle-ci

Le code transforme chaque message du tableau 'messages' en un nouvel 'observable' séparé de la bibliothèque RxJS

En utilisant la fonction 'from' de la bibliothèque RxJS, on crée un nouvel 'observable' qui émet les valeurs de l'objet 'messages'. Lorsqu'un observateur s'abonne à cet 'observable', il reçoit séquentiellement chaque valeur de l'objet 'messages'.

Question 5

Que fait le code suivant (il réutilise le code des questions précédentes) ?

```
const model = new Model();
const view = new View();
const controller = new Controller(model, view);

const observer = {
next: message => controller.nouveauMessage(message),
error: err => console.error(err),
complete: () => console.log('Terminé')
};

observable.subscribe(observer);
```

✓ Le code crée des instances des classes Model, View et Controller, et souscrit ensuite à un observable qui, à chaque émission, envoie un nouveau message au Controller pour être traité et affiché

Le code crée une nouvelle instance du modèle et s'abonne à un observable qui, lorsqu'il émet une valeur, envoie cette valeur à la vue pour être affichée

Le code crée un observable et s'abonne à celui-ci. Chaque fois que l'observable émet une valeur, cette valeur est stockée dans le modèle

Dans ce code, on utilise le patron de conception MVC (Modèle-Vue-Contrôleur) avec un observable RxJS. L'observable émet les valeurs séquentiellement et à chaque émission, la fonction 'next' de l'observer est appelée avec la valeur émise en tant que message. Cette fonction 'next' appelle alors la méthode 'nouveauMessage' du contrôleur avec le message, qui ajoute le message au modèle et demande à la vue d'afficher le message.

[exercice p. 31]

Solution n°3

```
class Projet {
// Déclaration de la propriété statique et privée pour stocker l'instance
static #instance;

constructor() {

// Vérifier si une instance existe déjà
if (Projet.#instance) {
// Si une instance existe, retourner cette instance
return Projet.#instance;
}

// Initialisation des propriétés de l'instance
this._taches = [];

// Stocker cette instance dans la propriété statique privée
Projet.#instance = this;
}

// Méthode statique pour obtenir l'instance du singleton
static getInstance() {
if (!Projet.#instance) {
// Si aucune instance n'existe, en créer une nouvelle
Projet.#instance = new Projet();
}
// Retourner l'instance unique
return Projet.#instance;
}

// Exemple de méthode pour manipuler l'état de l'objet
ajouterTache(tache) {
this._taches.push(tache);
}

// Exemple de méthode pour récupérer les tâches
obtenirTaches() {
return this._taches;
}
}

// Utilisation du Singleton
const projet1 = Projet.getInstance();
projet1.ajouterTache('Tâche 1');

const projet2 = Projet.getInstance();
```

```
console.log(projet2.obtenirTaches()); // ["Tâche 1"]

console.log(projet1 === projet2); // true
```

[exercice p. 31]

Solution n°4

```
class Tache {
constructor(titre, description, statut = 'en attente') {
this._titre = titre;
this._description = description;
this._statut = statut;
}

get titre() {
return this._titre;
}

get statut() {
return this._statut;
}

set statut(nouveauStatut) {
this._statut = nouveauStatut;
}
}
```

[exercice p. 31]

Solution n°5

```
import { BehaviorSubject } from 'rxjs';

class Tache {
constructor(titre, description, statut = 'en attente') {
    this._titre = titre;
    this._description = description;
    this._statut = statut;
    this._subject = new BehaviorSubject({ titre: this._titre, statut: this._statut
});
  }

//…
set statut(nouveauStatut) {
this._statut = nouveauStatut;
this._subject.next({ titre: this._titre, statut : this._statut });
}

subscribe(observer) {
this._subject.subscribe(observer);
}
}

const tache = new Tache('Tache 1', 'Description');
const observer = {
next: data => console.log(La tâche "${data.titre}" a un nouveau statut :
${data.statut}),
error: err => console.error(err),
complete: () => console.log('Terminé')
};

tache.subscribe(observer);
tache.statut = 'en cours';
```

[exercice p. 31] Solution n°6

```
const tache1 = new Tache('Tache 1', 'Première tâche');
const tache2 = new Tache('Tache 2', 'Deuxième tâche');
const tache3 = new Tache('Tache 3', 'Troisième tâche');

const observer = {
next: data => console.log(`La tâche "${data.titre}" a un nouveau statut : ${data.statut}`),
error: err => console.error(err),
complete: () => console.log('Terminé')
};

tache1.subscribe(observer);
tache2.subscribe(observer);
tache3.subscribe(observer);

projet.ajouterTache(tache1);
projet.ajouterTache(tache2);
projet.ajouterTache(tache3);

tache1.statut = 'en cours';
tache2.statut = 'terminée';
tache3.statut = 'en attente';

console.log(projet.recupererTache('Tache 1'));
console.log(projet.recupererTache('Tache 2'));
console.log(projet.recupererTache('Tache 3'));

projet.supprimerTache('Tache 2');
console.log(projet.recupererTache('Tache 2'));
```

Solution n°7

[exercice p. 31]

Question 1

Quelle est la différence entre le patron de conception Singleton et le patron de conception Factory ?

- ✓ Le Singleton assure qu'une classe n'ait qu'une seule instance, tandis que le Factory crée des instances d'objets sans spécifier leurs classes concrètes
- Le Singleton crée des instances d'objets sans spécifier leurs classes concrètes, tandis que le Factory assure qu'une classe n'ait qu'une seule instance
- Le Singleton et le Factory sont identiques et servent à créer des instances d'objets

Le patron de conception Singleton garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance. Le patron de conception Factory permet de créer des objets sans spécifier leurs classes concrètes, en déléguant cette responsabilité à des sous-classes.

Question 2

Dans le patron de conception MVC, à quoi sert le contrôleur ?

- ✓ Il gère les interactions entre la vue et le modèle
- Il affiche les données à l'utilisateur
- Il stocke et gère les données de l'application

Le contrôleur dans le patron de conception MVC gère les interactions entre la vue (l'interface utilisateur) et le modèle (la logique métier et les données). Il reçoit les événements provenant de la vue, effectue les actions nécessaires et met à jour la vue et le modèle en conséquence.

Question 3

Quel est l'objectif principal du patron de conception Memento dans la programmation orientée objet ?

- Faciliter la communication et les interactions entre les objets
- Simplifier la création d'objets en déléguant cette responsabilité aux sous-classes
- ✓ Permettre la sauvegarde et la restauration de l'état d'un objet sans violer son encapsulation

Le patron de conception Memento permet de sauvegarder et de restaurer l'état d'un objet sans violer son encapsulation. Il est utilisé pour implémenter des fonctionnalités telles que les annulations et les rétablissements dans les applications.

Question 4

Qu'est-ce qu'un Iterator dans le patron de conception Iterator ?

- ✓ Un objet qui permet d'accéder aux éléments d'un conteneur sans exposer sa représentation interne
- Un objet qui reçoit des notifications lorsqu'un autre objet change d'état
- Un objet qui crée des instances d'objets sans spécifier leurs classes concrètes

L'Iterator est un objet qui permet d'accéder aux éléments d'un conteneur (par exemple, une liste ou un arbre) sans exposer sa représentation interne. Il fournit une interface standard pour parcourir les éléments d'un conteneur.

Question 5

Quel est l'objectif du patron de conception Observable avec RxJS ?

Assurer qu'une classe n'ait qu'une seule instance

✓ Permettre à un objet de notifier automatiquement les autres objets de ses changements d'état en utilisant des streams de données

Fournir une interface standard pour créer des objets dans une superclasse

Le patron de conception Observable avec RxJS permet à un objet de notifier automatiquement les autres objets de ses changements d'état en utilisant des streams de données. Les Observables peuvent émettre plusieurs valeurs au fil du temps, et les observateurs peuvent s'abonner pour recevoir ces valeurs.