Exercice : Implémentation du Pattern Builder

Objectif : Créer une classe Product et une classe ProductBuilder pour construire des objets Product de manière flexible et étape par étape.

Étapes :

1. **Définir la classe Product :**
    - Créez une classe nommée Product avec plusieurs propriétés (par exemple, Name, Price, Category).
    - Ajoutez un constructeur privé pour empêcher la création directe d'instances.
2. **Créer la classe ProductBuilder :**
    - Créez une classe nommée ProductBuilder.
    - Ajoutez des méthodes pour définir les propriétés de Product.
    - Ajoutez une méthode Build qui retourne une instance de Product.
3. **Tester le Builder :**
    - Utilisez la classe ProductBuilder pour créer des instances de Product.

Exemple de code :

```
public class Product
{
  public string Name { get; private set; }
  public double Price { get; private set; }
  public string Category { get; private set; }
  private Product() { }
  public class ProductBuilder
  {
    private Product product = new Product();
    public ProductBuilder SetName(string name)
    {
      product.Name = name;
      return this;
```

```
        }
        public ProductBuilder SetPrice(double price)
        {
            product.Price = price;
            return this;
        }
        public ProductBuilder SetCategory(string category)
        {
            product.Category = category;
            return this;
        }
        public Product Build()
        {
            return product;
        }
    }
}
public class Program
{
    public static void Main(string[] args)
    {
        Product product = new Product.ProductBuilder()
            .SetName("Laptop")
            .SetPrice(999.99)
            .SetCategory("Electronics")
            .Build();
        Console.WriteLine($"Name: {product.Name}, Price: {product.Price}, Category: {
product.Category}");
```

```
    }
}
```

Points à vérifier :

- Assurez-vous que le constructeur de la classe Product est privé.
- Vérifiez que les méthodes de ProductBuilder retournent l'instance de ProductBuilder pour permettre le chaînage.
- Testez le code pour confirmer que les propriétés de Product sont correctement définies.

Exercice sur les Proxy en C#

Création et utilisation d’un Proxy pour une classe de service

Objectif

L'objectif de cet exercice est de comprendre comment créer et utiliser un Proxy pour une classe de service en C#, en appliquant le pattern Proxy pour contrôler l'accès à la classe de service.

Énoncé de l’exercice

Vous allez créer une classe de service `WeatherService` qui fournira des informations météorologiques. Ensuite, vous allez implémenter un Proxy `WeatherServiceProxy` pour contrôler l'accès à la classe de service.

Étapes à suivre

- Créez une interface `IWeatherService` avec une méthode `GetWeather(string city)` qui retourne un string contenant les informations météorologiques de la ville spécifiée.
- Implémentez la classe `WeatherService` qui réalise l'interface `IWeatherService`. La méthode `GetWeather` doit retourner des informations météorologiques fictives.

- Implémentez la classe `WeatherServiceProxy` qui réalise également l'interface `IWeatherService`. Cette classe doit contenir une instance de `WeatherService` et contrôler l'accès à la méthode `GetWeather`.
- Dans le Proxy, ajoutez des conditions pour, par exemple, limiter les requêtes ou vérifier l'authentification avant de déléguer l'appel à `WeatherService`.
- Testez le proxy et la classe de service pour vérifier que le proxy contrôle correctement l'accès.

Code exemple

Interface IWeatherService

```csharp
public interface IWeatherService
{
string GetWeather(string city);
}
```

Classe WeatherService

```csharp
public class WeatherService : IWeatherService
{
public string GetWeather(string city)
{
// Retourner des informations météorologiques fictives
return $"Weather for {city}: Sunny, 25°C";
}
}
```

Classe WeatherServiceProxy

```csharp
public class WeatherServiceProxy : IWeatherService
```

```
{
private WeatherService weatherService = new WeatherService();
private int requestCount = 0;
private const int maxRequests = 5;
public string GetWeather(string city)
{
if (requestCount >= maxRequests)
{
return "Max requests reached. Access denied.";
}
// Incrémenter le compteur de requêtes
requestCount++;
// Déléguer l'appel à WeatherService
return weatherService.GetWeather(city);
}
}
```

Tester le proxy

```csharp
class Program
{
static void Main(string[] args)
{
IWeatherService weatherService = new WeatherServiceProxy();
Console.WriteLine(weatherService.GetWeather("Paris"));
Console.WriteLine(weatherService.GetWeather("London"));
Console.WriteLine(weatherService.GetWeather("New York"));
```

Console.WriteLine(weatherService.GetWeather("Tokyo"));

Console.WriteLine(weatherService.GetWeather("Sydney"));

Console.WriteLine(weatherService.GetWeather("Berlin")); // Cette requête devrait être refusée

}

}

```

## Points à vérifier

- Assurez-vous que la classe `WeatherService` implémente correctement l'interface `IWeatherService`.
- Vérifiez que le proxy `WeatherServiceProxy` contrôle l'accès à la méthode `GetWeather` selon les conditions définies.
- Testez le code pour confirmer que le proxy limite correctement le nombre de requêtes.

# Exercice sur le patron de conception Décorateur

Création d'un service de météo avec un contrôle d'accès

## Objectif

L'objectif de cet exercice est de mettre en pratique le patron de conception Décorateur en créant un service de météo qui contrôle l'accès à ses méthodes.

## Instructions

### Étape 1 : Créez l'interface IWeatherService

Créez une interface nommée `IWeatherService` avec la méthode suivante :

```csharp

public interface IWeatherService

{

string GetWeather(string location);

}

```

Étape 2 : Implémentez la classe WeatherService

Créez une classe `WeatherService` qui implémente l'interface `IWeatherService`. Cette classe doit retourner une chaîne de caractères simulant la météo d'une localisation donnée.

```csharp

public class WeatherService : IWeatherService

{

public string GetWeather(string location)

{

// Simule la météo pour la localisation donnée

return $"La météo de {location} est ensoleillée.";

}

}

```

Étape 3 : Créez la classe WeatherServiceProxy

Créez une classe `WeatherServiceProxy` qui implémente également `IWeatherService`. Cette classe doit utiliser une instance de `WeatherService` et contrôler l'accès à sa méthode `GetWeather` pour limiter le nombre de requêtes.

```csharp

public class WeatherServiceProxy : IWeatherService

{

private readonly WeatherService _weatherService;

private readonly HashSet _allowedLocations;

private int _requestCount;

private readonly int _maxRequests;

public WeatherServiceProxy(WeatherService weatherService, int maxRequests)

```
{
_weatherService = weatherService;
_allowedLocations = new HashSet { "Tokyo", "Sydney", "New York" };
_requestCount = 0;
_maxRequests = maxRequests;
}
public string GetWeather(string location)
{
if (_requestCount >= _maxRequests)
{
return "Limite de requêtes atteinte.";
}
if (!_allowedLocations.Contains(location))
{
return "Accès refusé pour cette localisation.";
}
_requestCount++;
return _weatherService.GetWeather(location);
}
}
```

Étape 4 : Testez votre code

Créez un programme principal pour tester le fonctionnement de votre `WeatherServiceProxy`.

```csharp
class Program
{
static void Main(string[] args)
```

{

WeatherService weatherService = new WeatherService();

WeatherServiceProxy weatherServiceProxy = new
WeatherServiceProxy(weatherService, 3);

Console.WriteLine(weatherServiceProxy.GetWeather("Tokyo"));

Console.WriteLine(weatherServiceProxy.GetWeather("Sydney"));

Console.WriteLine(weatherServiceProxy.GetWeather("Berlin")); // Cette requête devrait être refusée

Console.WriteLine(weatherServiceProxy.GetWeather("New York")); // Cette requête devrait être acceptée

Console.WriteLine(weatherServiceProxy.GetWeather("Tokyo")); // Cette requête devrait être refusée car limite atteinte

}

}

```

## Questions de révision

- Expliquez le rôle du patron de conception Décorateur dans cet exercice.
- Comment la classe `WeatherServiceProxy` contrôle-t-elle l'accès à la méthode `GetWeather` ?
- Quels autres types de contrôle d'accès pourriez-vous implémenter avec ce patron de conception ?

# Exercice sur le Patron de Conception Bridge : Commande, Paiement et Facture

## Objectif

Comprendre et implémenter le patron de conception Bridge pour séparer l'abstraction de son implémentation, en appliquant ce modèle aux concepts de commande, paiement et facture.
```

Contexte

Vous êtes responsable de la conception d'un système de gestion de commande pour un magasin en ligne. Le système doit gérer différentes méthodes de paiement et de facturation, tout en permettant l'ajout facile de nouvelles méthodes sans modifier les classes existantes.

Instructions

- Créez une interface IOrder représentant une commande avec les méthodes suivantes :
- ProcessOrder() : Méthode abstraite pour traiter la commande

Créez une classe abstraite Order qui implémente l'interface IOrder et qui contient un membre de type IPayment et un membre de type IInvoice :

Le constructeur de Order prend comme paramètres les implémentations de IPayment et IInvoice.

ProcessOrder() appelle les méthodes de traitement de paiement et de facturation respectives.

Créez des classes concrètes OnlineOrder et InStoreOrder qui héritent de Order et implémentent la méthode ProcessOrder().

Créez une interface IPayment avec la méthode ProcessPayment().

Implémentez les classes concrètes de paiement suivantes :

CreditCardPayment : Traitement des paiements par carte de crédit

PayPalPayment : Traitement des paiements via PayPal

Créez une interface IInvoice avec la méthode GenerateInvoice().

Implémentez les classes concrètes de facturation suivantes :

EmailInvoice : Envoi de la facture par email

PaperInvoice : Envoi de la facture papier par courrier

Exemple de Code

Voici un exemple simplifié de l'implémentation du patron de conception Bridge pour ce système :

interface IOrder {

```
void ProcessOrder();
}
abstract class Order : IOrder {
protected IPayment payment;
protected IInvoice invoice;
public Order(IPayment payment, IInvoice invoice) {
this.payment = payment;
this.invoice = invoice;
}
public abstract void ProcessOrder();
}
class OnlineOrder : Order {
public OnlineOrder(IPayment payment, IInvoice invoice) : base(payment, invoice) { }
public override void ProcessOrder() {
Console.WriteLine("Processing online order...");
payment.ProcessPayment();
invoice.GenerateInvoice();
}
}
class InStoreOrder : Order {
public InStoreOrder(IPayment payment, IInvoice invoice) : base(payment, invoice) { }
public override void ProcessOrder() {
Console.WriteLine("Processing in-store order...");
payment.ProcessPayment();
invoice.GenerateInvoice();
}
}
interface IPayment {
```

```
void ProcessPayment();
}
class CreditCardPayment : IPayment {
public void ProcessPayment() {
Console.WriteLine("Processing credit card payment...");
}
}
class PayPalPayment : IPayment {
public void ProcessPayment() {
Console.WriteLine("Processing PayPal payment...");
}
}
interface IInvoice {
void GenerateInvoice();
}
class EmailInvoice : IInvoice {
public void GenerateInvoice() {
Console.WriteLine("Generating email invoice...");
}
}
class PaperInvoice : IInvoice {
public void GenerateInvoice() {
Console.WriteLine("Generating paper invoice...");
}
}
// Utilisation de l'exercice
class Program {
static void Main(string[] args) {
```

```
IPayment creditCardPayment = new CreditCardPayment();

IInvoice emailInvoice = new EmailInvoice();

IOrder onlineOrder = new OnlineOrder(creditCardPayment, emailInvoice);

onlineOrder.ProcessOrder();

IPayment paypalPayment = new PayPalPayment();

IInvoice paperInvoice = new PaperInvoice();

IOrder inStoreOrder = new InStoreOrder(paypalPayment, paperInvoice);

inStoreOrder.ProcessOrder();

}

}
```

Conclusion

En suivant cet exercice, vous comprendrez comment le patron de conception Bridge permet de séparer les abstractions des implémentations, facilitant ainsi l'ajout de nouvelles fonctionnalités sans modifier le code existant. Cette flexibilité est essentielle pour construire des systèmes évolutifs et maintenables.

Travail Pratique : Le Patron de Conception Chain of Responsibility

Introduction

Le patron de conception Chain of Responsibility permet de traiter les demandes à travers une chaîne d'objets. Chaque objet de la chaîne vérifie s'il peut traiter la demande, sinon il la passe au prochain objet dans la chaîne. Ce patron est particulièrement utile pour découpler l'expéditeur et le destinataire de la demande, permettant ainsi une flexibilité accrue et une gestion simplifiée des responsabilités.

Objectifs

- Comprendre le fonctionnement du patron de conception Chain of Responsibility.

- Appliquer le patron de conception pour résoudre un problème concret.
- Découpler la logique de traitement de la demande de l'expéditeur et du destinataire.

Exercice Pratique

Étape 1 : Définir l'interface Handler

Créez une interface IHandler avec les méthodes suivantes :

- SetNextHandler(IHandler handler) : Permet de définir le prochain handler dans la chaîne.
- HandleRequest(Request request) : Traite la demande ou la passe au prochain handler.

Étape 2 : Implémenter des Handlers Concrets

Créez des classes concrètes qui implémentent l'interface IHandler. Par exemple :

- AuthorizationHandler : Vérifie si l'utilisateur est autorisé.
- ValidationHandler : Valide les données de la demande.
- ProcessingHandler : Traite la demande.

Étape 3 : Créer la Classe Request

Créez une classe Request qui contient les informations nécessaires pour les handlers. Par exemple :

- Utilisateur
- Données de la demande
- Statut de traitement

Étape 4 : Configurer la Chaîne de Responsabilité

Dans la méthode Main, configurez la chaîne de responsabilité en créant les handlers et en les chaînant. Par exemple :

```csharp

IHandler authorizationHandler = new AuthorizationHandler();

IHandler validationHandler = new ValidationHandler();

IHandler processingHandler = new ProcessingHandler();

authorizationHandler.SetNextHandler(validationHandler);

validationHandler.SetNextHandler(processingHandler);
```

```
Request request = new Request(/* données de la demande */);

authorizationHandler.HandleRequest(request);
```

Conclusion

En suivant cet exercice, vous apprendrez comment le patron de conception Chain of Responsibility permet de traiter les demandes de manière flexible et découplée. Cette approche facilite l'ajout de nouvelles responsabilités dans le système sans modifier le code existant, rendant ainsi le système évolutif et maintenable.

Exercice : Patron de conception Command

Introduction

Le patron de conception Command permet d'encapsuler une demande en tant qu'objet, ce qui permet de paramétrer les clients avec des requêtes, des files d'attente ou des journaux, et de prendre en charge les opérations annulables. Vous allez mettre en œuvre ce patron afin de mieux comprendre comment il fonctionne et comment il peut être utilisé pour améliorer la flexibilité et la maintenabilité de votre code.

Instructions

Conclusion

En réalisant cet exercice, vous comprendrez comment le patron de conception Command permet de découpler les objets qui émettent des demandes des objets qui les exécutent. Cette approche favorise la flexibilité et la maintenabilité du code, en permettant d'ajouter, de supprimer ou de modifier des commandes sans affecter les autres parties du système.

Exercice de patron de conception Mediator en C#

Introduction

Le patron de conception Mediator permet de réduire les dépendances entre les objets en les faisant communiquer via un objet intermédiaire appelé médiateur. Cela facilite la gestion de la communication et améliore la flexibilité et la maintenabilité du code.

Objectif

En réalisant cet exercice, vous allez implémenter le patron de conception Mediator en C# afin de comprendre comment il permet de gérer la communication entre plusieurs objets de manière centralisée.

Étapes de l'exercice

Étape 1 : Créer les classes de composants

- Créez une classe abstraite Participant représentant les objets qui communiquent via le médiateur. Elle devrait avoir une méthode Send pour envoyer des messages et une méthode Receive pour recevoir des messages.
- Implémentez deux classes concrètes ConcreteParticipant1 et ConcreteParticipant2 qui héritent de Participant.

Étape 2 : Créer l'interface Mediator

- Créez une interface IMediator déclarant les méthodes Register (pour enregistrer les participants) et SendMessage (pour envoyer des messages).

Étape 3 : Implémenter la classe Mediator

- Créez une classe ConcreteMediator qui implémente Imediator. Elle devrait maintenir une liste de participants et gérer la communication entre eux.

Étape 4 : Mettre en œuvre la communication

- Implémentez les méthodes SendMessage dans ConcreteMediator pour envoyer des messages aux participants enregistrés.
- Modifiez les classes ConcreteParticipant1 et ConcreteParticipant2 pour qu'elles utilisent le médiateur pour envoyer et recevoir des messages.

Exemple de code

```csharp
using System;
using System.Collections.Generic;
// Étape 1 : Créer les classes de composants
public abstract class Participant
{
protected Imediator mediator;
public Participant(Imediator mediator)
{
this.mediator = mediator;
}
public abstract void Send(string message);
public abstract void Receive(string message);
}
public class ConcreteParticipant1 : Participant
{
public ConcreteParticipant1(Imediator mediator) : base(mediator) { }
public override void Send(string message)
{
Console.WriteLine(« Participant 1 envoie le message : «  + message) ;
mediator.SendMessage(this, message);
}
public override void Receive(string message)
{
Console.WriteLine(« Participant 1 reçoit le message : «  + message) ;
}
}
```

```
public class ConcreteParticipant2 : Participant
{
public ConcreteParticipant2(Imediator mediator) : base(mediator) { }
public override void Send(string message)
{
Console.WriteLine(« Participant 2 envoie le message : «  + message) ;
mediator.SendMessage(this, message) ;
}
public override void Receive(string message)
{
Console.WriteLine(« Participant 2 reçoit le message : «  + message) ;
}
}
// Étape 2 : Créer l'interface Mediator
public interface Imediator
{
void Register(Participant participant) ;
void SendMessage(Participant sender, string message) ;
}
// Étape 3 : Implémenter la classe Mediator
public class ConcreteMediator : Imediator
{
private List participants = new List() ;
public void Register(Participant participant)
{
participants.Add(participant) ;
}
public void SendMessage(Participant sender, string message)
```

```
{
foreach (Participant participant in participants)
{
if (participant != sender)
{
participant.Receive(message) ;
}
}
}
}
// Étape 4 : Mettre en œuvre la communication
public class Program
{
public static void Main()
{
ConcreteMediator mediator = new ConcreteMediator() ;
Participant participant1 = new ConcreteParticipant1(mediator) ;
Participant participant2 = new ConcreteParticipant2(mediator) ;
mediator.Register(participant1) ;
mediator.Register(participant2) ;
participant1.Send(« Bonjour de Participant 1 ») ;
participant2.Send(« Salut de Participant 2 ») ;
}
}
```

Conclusion

En réalisant cet exercice, vous comprendrez comment le patron de conception Mediator permet de centraliser la communication entre plusieurs objets, réduisant ainsi les dépendances et améliorant la flexibilité et la maintenabilité du code.

Exercices sur l'Iterator en C#

Comprendre et implémenter le patron de conception Iterator

Introduction

Le patron de conception Iterator permet de parcourir des éléments d'une collection sans exposer les détails de son implémentation. C'est une manière de fournir un accès séquentiel aux éléments d'une collection.

Exercice 1: Implémentation de l'Iterator

Objectif :

Implémenter une classe iterator qui permet de parcourir une collection d'objets.

Instructions :

- Créez une interface IIterator avec les méthodes HasNext() et Next().
- Créez une classe ConcreteIterator qui implémente IIterator pour parcourir une collection.
- Créez une interface ICollection avec une méthode GetIterator().
- Créez une classe ConcreteCollection qui implémente ICollection et contient une collection d'objets.

Exemple de code :

```csharp

// Interface IIterator

public interface IIterator

{

bool HasNext();
```

```
object Next();

}

// ConcreteIterator

public class ConcreteIterator : IIterator

{

private ConcreteCollection _collection;

private int _index = 0;

public ConcreteIterator(ConcreteCollection collection)

{

_collection = collection;

}

public bool HasNext()

{

return _index   _items.Count;

public void AddItem(object item)

{

_items.Add(item);

}

public object this[int index] => _items[index];

public IIterator GetIterator()

{

return new ConcreteIterator(this);

}

}
```

Exercice 2: Utilisation de l'Iterator

Objectif :

Utiliser la classe iterator pour parcourir une collection d'objets.

Instructions :

- Créez une instance de ConcreteCollection et ajoutez des objets à la collection.
- Obtenez un iterator en appelant GetIterator() sur la collection.
- Utilisez le iterator pour parcourir et afficher les éléments de la collection.

Exemple de code :

```csharp
public class Program
{
public static void Main()
{
ConcreteCollection collection = new ConcreteCollection();
collection.AddItem("Item 1");
collection.AddItem("Item 2");
collection.AddItem("Item 3");
IIterator iterator = collection.GetIterator();
while (iterator.HasNext())
{
Console.WriteLine(iterator.Next());
}
}
}
```

Conclusion

En réalisant ces exercices, vous comprendrez comment le patron de conception Iterator permet de parcourir des éléments d'une collection sans exposer les détails de son implémentation, réduisant ainsi les dépendances et améliorant la flexibilité et la maintenabilité du code.

Exercices pour le patron de conception Observer en C#

Exercice 1 : Implémentation de base de l'Observer

Objectif : Créez une implémentation simple du patron de conception Observer.

- Créez une interface IObserver avec une méthode Update().
- Créez une interface ISubject avec des méthodes pour attacher, détacher et notifier les observateurs.
- Implémentez la classe ConcreteSubject qui contient une liste d'observateurs et un état. La classe doit appeler la méthode Update() de chaque observateur lorsque son état change.
- Implémentez la classe ConcreteObserver qui stocke une référence à l'objet ConcreteSubject et met à jour son propre état lorsque Update() est appelé.

Exercice 2 : Application du patron Observer à une application météo

Objectif : Appliquez le patron Observer à un système de suivi météorologique.

- Créez une classe WeatherData qui implémente l'interface ISubject. Cette classe doit contenir des données météorologiques telles que la température, l'humidité et la pression.
- Créez des classes CurrentConditionsDisplay, StatisticsDisplay et ForecastDisplay qui implémentent l'interface IObserver. Ces classes doivent afficher les données météorologiques sous différentes formes.
- Ajoutez la logique pour que les affichages s'inscrivent auprès de WeatherData et reçoivent des mises à jour lorsque les données changent.

Exercice 3 : Implémentation d'un système de notification d'actualités

Objectif : Implémentez un système de notification pour un site d'actualités.

- Créez une classe NewsAgency qui implémente l'interface ISubject. Cette classe doit contenir des articles d'actualité.
- Créez des classes NewsSubscriber qui implémentent l'interface IObserver et affichent les dernières nouvelles.
- Ajoutez la logique pour que les abonnés puissent s'inscrire auprès de NewsAgency et recevoir des mises à jour lorsque de nouveaux articles sont publiés.

Exercice 4 : Ajout d'une interface graphique

Objectif : Ajoutez une interface graphique à votre implémentation de l'Observer.

- Utilisez Windows Forms pour créer une interface utilisateur qui affiche les données météorologiques ou les actualités.
- Créez des boutons pour permettre à l'utilisateur de s'inscrire ou de se désinscrire des notifications.
- Implémentez la logique pour mettre à jour l'interface utilisateur lorsque les données changent.

En réalisant ces exercices, vous maîtriserez l'implémentation du patron de conception Observer en C#, ainsi que son application pratique dans des scénarios réels.