TP9 : Principe d'Inversion de Dépendance (DIP)

Objectif

Comprendre et appliquer le principe d'Inversion de Dépendance (Dependency Inversion Principle), le cinquième des principes SOLID.

Rappel du Principe

"Les modules de haut niveau ne devraient pas dépendre des modules de bas niveau. Les deux devraient dépendre d'abstractions."

"Les abstractions ne devraient pas dépendre des détails. Les détails devraient dépendre des abstractions."

Ce principe stipule que nous devrions dépendre d'abstractions (interfaces ou classes abstraites) plutôt que de dépendre directement d'implémentations concrètes. Cela favorise le découplage et la flexibilité du code.

Problème

Le code suivant représente un système de notifications pour un jeu. Les modules de haut niveau (comme le `GameController`) dépendent directement des modules de bas niveau (comme `EmailService` et `SMSService`), ce qui viole le principe DIP.

```
// EmailService.cs - Module de bas niveau
public class EmailService
{
    public void SendEmail(string to, string subject, string body)
    {
        // Logique d'envoi d'email
        Console.WriteLine($"Email envoyé à {to}, Sujet: {subject}, Corps:
{body}");
    }
}

// SMSService.cs - Module de bas niveau
public class SMSService
{
    public void SendSMS(string phoneNumber, string message)
    {
        // Logique d'envoi de SMS
        Console.WriteLine($"SMS envoyé à {phoneNumber}, Message: {message}");
    }
}

// Player.cs
public class Player
{
    public string Name { get; private set; }
```

```
    public string Email { get; private set; }
    public string PhoneNumber { get; private set; }

    public Player(string name, string email, string phoneNumber)
    {
        Name = name;
        Email = email;
        PhoneNumber = phoneNumber;
    }
}

// NotificationService.cs - Service utilisant les modules de bas niveau
public class NotificationService
{
    private readonly EmailService emailService;
    private readonly SMSService smsService;

    public NotificationService()
    {
        emailService = new EmailService();
        smsService = new SMSService();
    }

    public void NotifyPlayerAchievementUnlocked(Player player, string
achievementName)
    {
        // Envoyer un email
        emailService.SendEmail(
            player.Email,
            "Félicitations pour votre réussite!",
            $"Cher {player.Name}, vous avez débloqué l'achievement:
{achievementName}!"
        );

        // Envoyer un SMS
        smsService.SendSMS(
            player.PhoneNumber,
            $"Félicitations! Vous avez débloqué l'achievement: {achievementName}!"
        );
    }

    public void NotifyPlayerGameInvite(Player inviter, Player invitee, string
gameName)
    {
        // Envoyer un email à l'invité
        emailService.SendEmail(
            invitee.Email,
            "Invitation à rejoindre une partie",
            $"Cher {invitee.Name}, {inviter.Name} vous invite à rejoindre une
partie de {gameName}!"
        );

        // Envoyer un SMS à l'invité
        smsService.SendSMS(
```

```
            invitee.PhoneNumber,
            $"{inviter.Name} vous invite à rejoindre une partie de {gameName}!"
        );
    }
}

// GameController.cs - Module de haut niveau
public class GameController
{
    private readonly NotificationService notificationService;

    public GameController()
    {
        notificationService = new NotificationService();
    }

    public void UnlockAchievement(Player player, string achievementName)
    {
        // Logique de déverrouillage d'achievement
        Console.WriteLine($"Achievement '{achievementName}' débloqué pour
{player.Name}");

        // Notification
        notificationService.NotifyPlayerAchievementUnlocked(player,
achievementName);
    }

    public void InvitePlayerToGame(Player inviter, Player invitee, string
gameName)
    {
        // Logique d'invitation
        Console.WriteLine($"{inviter.Name} a invité {invitee.Name} à jouer à
{gameName}");

        // Notification
        notificationService.NotifyPlayerGameInvite(inviter, invitee, gameName);
    }
}

// Exemple d'utilisation
public class Program
{
    public static void Main()
    {
        Player player1 = new Player("Alice", "alice@example.com", "123-456-7890");
        Player player2 = new Player("Bob", "bob@example.com", "098-765-4321");

        GameController gameController = new GameController();

        // Débloquer un achievement
        gameController.UnlockAchievement(player1, "Master Strategist");

        // Inviter un joueur
        gameController.InvitePlayerToGame(player1, player2, "Chess Champions");
```

```
    }
}
```

Problèmes avec ce code

1. Les modules de haut niveau (`GameController`, `NotificationService`) dépendent directement des modules de bas niveau (`EmailService`, `SMSService`)
2. Les dépendances sont créées à l'intérieur des classes qui les utilisent, ce qui rend difficile le remplacement de ces dépendances
3. Le code est difficile à tester car on ne peut pas facilement simuler (mock) les dépendances
4. L'ajout d'un nouveau mode de notification (comme les notifications push) nécessiterait de modifier plusieurs classes

Exercice

Refactorisez le code en appliquant le principe d'Inversion de Dépendance :

1. Créez des abstractions (interfaces) pour les services de notification
2. Faites en sorte que les modules de haut niveau dépendent de ces abstractions plutôt que des implémentations concrètes
3. Utilisez l'injection de dépendances pour fournir les implémentations concrètes aux classes qui en ont besoin
4. Assurez-vous que l'ajout d'un nouveau type de notification (comme les notifications push) soit facile

Avantages attendus

- Code plus flexible et plus facile à étendre
- Meilleure testabilité grâce à la possibilité de simuler les dépendances
- Découplage des modules de haut niveau et de bas niveau
- Possibilité de changer les implémentations sans modifier le code client

Conseils

- Identifiez les dépendances directes entre les modules et créez des abstractions appropriées
- Utilisez l'injection de dépendances via les constructeurs ou les propriétés
- Pensez aux principes de séparation des interfaces (ISP) en conjonction avec DIP