Patrons Stratégie et Observateur

Pour cette étude, notre objectif est de concevoir une solution capable de compresser des données de différents types (image, vidéo et son). Nous allons partir d'une solution antipattern mal conçue et implémentée, qui viole les principes SOLID, pour arriver à une solution basée sur des patrons de conception appropriés.

1 Patron Stratégie

Dans la solution antipattern, la classe `DataCompression` contient toutes les fonctions nécessaires pour compresser les différents types de données. Cette classe implémente une méthode distincte pour chaque type de donnée. Cependant, cette approche est peu évolutive, car avec l'ajout de nouveaux types et formats de données, la classe deviendra de plus en plus volumineuse et difficile à maintenir.

Question 1 • A partir de la réalisation antipattern fournie, donnez le diagramme de classes correspondant. Quel est le problème avec cette solution en termes de principes SOLID ?

Nous allons maintenant utiliser le patron de conception *Stratégie* pour résoudre le problème de la classe antipattern précédente. Nous créons une interface `DataCompressionStrategy` qui définit la méthode `processData()` et nous créons une classe concrète pour chaque type de traitement. De cette façon, nous pouvons facilement ajouter de nouveaux types de compression en créant une nouvelle classe implémentant l'interface `DataCompressionStrategy`. La classe `DataCompression` n'a plus besoin de connaître tous les types de traitements disponibles, elle utilise simplement l'objet `DataCompressionStrategy` qui lui est passé en paramètre. Ce patron permet de respecter le principe SOLID de la séparation des préoccupations, en permettant de séparer les algorithmes de compression de la classe de traitement des données.

Question 2 • Donnez le diagramme de classes qui utilise le patron de conception Stratégie.

Question 3 • Comment la solution avec le patron de conception Stratégie résout-elle ce problème et respecte-t-elle les principes SOLID ?

Question 4 • Quels sont les avantages de la solution avec le patron de conception Stratégie par rapport à la solution antipattern ?

Question 5 • Implémentez votre solution avec la classe `DataCompression`, l'interface `DataCompressionStrategy` et les trois classes concrètes : `ImageCompressionStrategy`, `VideoCompressionStrategy`, et `AudioCompressionStr`

Question 6 • Ajoutez une méthode `setDataCompressionStrategy(DataCompressionStrategy strategy)` à la classe `DataCompression`. Cette méthode prendra en charge l'instanciation de la stratégie appropriée en fonction du type de données.

Question 7 • Testez votre implémentation en utilisant l'application principale fournie avec la solution antipattern.

Question 8 • Proposez une extension possible pour votre implémentation du pattern Stratégie.

2 Patron Observateur

Nous souhaitons maintenant aborder la deuxième partie de l'implémentation antipattern, qui est dédiée à la collecte d'informations à chaque fois qu'une donnée est compressée. L'objectif est de maintenir un journal de bord qui contient, pour chaque type de donnée, des informations relatives à la compression effectuée.

La classe `Observer` est une interface qui définit la méthode `update` qui sera appelée par la classe `DataCompression` lorsque de nouvelles données sont disponibles. Les trois classes `ImageObserver`, `VideoObserver` et `SoundObserver` implémentent cette interface et représentent les observateurs spécifiques à chaque type de données.

La solution antipattern qui est fournie utilise la classe `DataCompression` pour traiter tous les types de données en utilisant des blocs de code conditionnels pour déterminer le type de donnée à traiter.

Question 9 • Donnez le diagramme de classes de la solution antipattern. Quel est le problème avec cette solution en termes de principes SOLID ?

Question 10 • Donnez le diagramme de classes qui utilise le patron de conception Observateur.

Question 11 • Quels sont les avantages de la solution avec le patron de conception Observateur par rapport à la solution antipattern ?

Question 12 • Créez l'interface `DataObserver` avec une méthode `update()`. Cette interface sera implémentée par trois classes concrètes : `ImageObserver`, `VideoObserver`, et `AudioObserver`. Chacune de ces classes sera responsable de recevoir une notification de `DataCompression` lorsque le traitement d'un type de données est terminé.

Question 13 • Ajoutez une méthode `addDataObserver(DataObserver observer)` à la classe `DataCompression`. Cette méthode permettra à `DataCompression` d'ajouter des observateurs pour chaque type de données.

Question 14 • Testez votre implémentation en utilisant l'application principale fournie avec la solution antipattern.

Question 15 • Proposez une extension possible pour votre implémentation du pattern Observateur.