eBook Gratuit

APPRENEZ

Design patterns

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#design-patterns

Table des matières

À propos 1

Chapitre 1: Démarrer avec les modèles de conception 2

Remarques 2

Examples 2

introduction 2

Chapitre 2: Adaptateur 4

Examples 4

Modèle d'adaptateur (PHP) 4

Adaptateur (Java) 4

Exemple Java 6

Adaptateur (UML et exemple de situation) 7

Chapitre 3: Chaîne de responsabilité 11

Examples 11

Exemple de chaîne de responsabilité (Php) 11

Chapitre 4: chargement paresseux 13

Introduction 13

Examples 13

Chargement paresseux JAVA 13

Chapitre 5: Façade 15

Examples 15

Façade du monde réel (C #) 15

Exemple de façade en java 15

Chapitre 6: Injection de dépendance 19

Introduction 19

Remarques 19

Examples 20

Injection Setter (C #) 20

Constructeur Injection (C #) 20

Chapitre 7: Méthode d'usine statique 22

Examples 22

Méthode d'usine statique 22
Cacher l'accès direct au constructeur 22
Méthode d'usine statique C # 23
Chapitre 8: Méthode du modèle 25
Examples 25
Implémentation de la méthode de modèle en Java 25
Chapitre 9: Modèle d'itérateur 29
Examples 29
Le modèle d'itérateur 29
Chapitre 10: Modèle d'objet nul 31
Remarques 31
Examples 31
Modèle d'objet nul (C ++) 31
Objet Null Java utilisant enum 32
Chapitre 11: Modèle de commande 34
Examples 34
Exemple de modèle de commande en Java 34
Chapitre 12: Modèle de conception DAO (Data Access Object) 37
Examples 37
Modèle de conception de l'objet d'accès aux données J2EE avec Java 37
Chapitre 13: Modèle de médiateur 40
Examples 40
Exemple de modèle de médiateur en Java 40
Chapitre 14: Modèle de pont 43
Examples 43
Implémentation d'un modèle de pont en Java 43
Chapitre 15: Modèle de stratégie 46
Examples 46
Cacher les détails de la mise en œuvre de la stratégie 46
Exemple de modèle de stratégie en Java avec classe de contexte 47
Modèle de stratégie sans classe de contexte / Java 49

Utilisation des interfaces fonctionnelles Java 8 pour implémenter le modèle de stratégie 50
La version Java classique 50
Utilisation des interfaces fonctionnelles Java 8 51
Stratégie (PHP) 52
Chapitre 16: Monostate 54
Remarques 54
Examples 54
Le modèle de monostate 54
Hiérarchies basées sur Monostate 55
Chapitre 17: Motif composite 57
Examples 57
Enregistreur composite 57
Chapitre 18: Motif composite 59
Introduction 59
Remarques 59
Examples 59
pseudocode pour un gestionnaire de fichiers stupide 59
Chapitre 19: Motif de constructeur 61
Remarques 61
Examples 61
Modèle de générateur / C # / Interface fluide 61
Modèle de générateur / Implémentation Java 62
Modèle de constructeur en Java avec composition 64
Java / Lombok 67
Modèle de générateur avancé avec une expression Java 8 Lambda 68
Chapitre 20: Motif de décorateur 71
Introduction 71
Paramètres 71
Examples 71
VendingMachineDecorator 71
Caching Decorator 75

Chapitre 21: Motif de visiteur 77
Examples 77
Exemple de motif visiteur en C ++ 77
Exemple de modèle de visiteur en java 79
Exemple de visiteur en C ++ 82
Traverser de grands objets 83
Chapitre 22: Motif Prototype 85
Introduction 85
Remarques 85
Examples 85
Motif Prototype (C ++) 85
Motif Prototype (C #) 86
Motif Prototype (JavaScript) 86
Chapitre 23: Multiton 88
Remarques 88
Examples 88
Pool of Singletons (exemple PHP) 88
Registre des singletons (exemple PHP) 89
Chapitre 24: MVC, MVVM et MVP 91
Remarques 91
Examples 91
Model View Controller (MVC) 91
Modèle View ViewModel (MVVM) 92
Chapitre 25: Observateur 96
Remarques 96
Examples 96
Observateur / Java 96
Observateur utilisant IObservable et IObserver (C #) 98
Chapitre 26: Ouvrir le principe de fermeture 100
Introduction 100
Remarques 100

Examples ... 100
Ouvrir Fermer Principe de violation ... 100
Open Close Principle support ... 101
Chapitre 27: Publier-S'abonner ... 102
Examples ... 102
Publier-S'abonner en Java ... 102
Exemple de pub-sub simple en JavaScript ... 103
Chapitre 28: Repository ... 104
Remarques ... 104
Examples ... 104
Référentiels en lecture seule (C #) ... 104
Les interfaces ... 104
Un exemple d'implémentation utilisant ElasticSearch comme technologie (avec NEST) ... 104
Modèle de référentiel utilisant Entity Framework (C #) ... 105
Chapitre 29: Singleton ... 108
Remarques ... 108
Examples ... 108
Singleton (C #) ... 108
Motif Singleton Thread-Safe ... 108
Singleton (Java) ... 109
Singleton (C ++) ... 111
Lazy Singleton exemple pratique en java ... 111
C # Exemple: Singleton multithread ... 113
Singleton (PHP) ... 114
Singleton Design pattern (en général) ... 115
Chapitre 30: SOLIDE ... 116
Introduction ... 116
Examples ... 116
SRP - Principe de responsabilité unique ... 116
Chapitre 31: tableau noir ... 121
Examples ... 121

Echantillon C # 121
Chapitre 32: Usine 125
Remarques 125
Examples 125
Usine simple (Java) 125
Fabrique abstraite (C ++) 126
Exemple simple de Factory qui utilise un IoC (C #) 128
Une fabrique abstraite 130
Exemple d'usine en implémentant la méthode Factory (Java) 131
Usine de poids mouche (C #) 135
Méthode d'usine 136
Crédits 137

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: design-patterns

It is an unofficial and free Design patterns ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Design patterns.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec les modèles de conception

Remarques

Cette section fournit une vue d'ensemble de ce qu'est un modèle de conception et pourquoi un développeur peut vouloir l'utiliser. Les exemples peuvent fournir une représentation graphique du motif, un scénario consistant en un problème étant donné le contexte dans lequel un motif peut être utilisé et mentionner les compromis possibles.

Il devrait également mentionner tous les grands sujets dans les modèles de conception, et établir un lien avec les sujets connexes. La documentation pour les modèles de conception étant nouvelle, vous devrez peut-être créer des versions initiales de ces rubriques connexes.

Examples

introduction

Selon Wikipedia :

> [A] *le modèle de conception de logiciel* est une solution réutilisable générale à un problème commun dans un contexte donné dans la conception de logiciel. Ce n'est pas un design fini qui peut être transformé directement en code source ou en code machine. C'est une description ou un modèle sur la façon de résoudre un problème qui peut être utilisé dans de nombreuses situations différentes. Les modèles de conception sont des meilleures pratiques formalisées que le programmeur peut utiliser pour résoudre des problèmes courants lors de la conception d'une application ou d'un système.
>
> (Récupéré: 2016-10-13)

Il existe de nombreux modèles de conception de logiciels reconnus, et de nouveaux modèles sont proposés régulièrement. D'autres sujets couvrent un grand nombre des modèles les plus courants et l'article de Wikipedia fournit une liste plus complète.

De même, il existe différentes manières de classer les modèles de conception, mais la classification originale est la suivante:

- **Motifs de création** : Factory , Builder , Singleton , etc.
- **Modèles structurels** : adaptateur , composite , proxy, etc.
- **Modèles comportementaux** : itérateur , stratégie , visiteur , etc.
- **Modèles de concurrence** : ActiveObject, Monitor, etc.

L'idée des modèles de conception a été étendue aux *modèles de conception spécifiques* à un *domaine* pour des domaines tels que la conception d'interface utilisateur, la visualisation de

données, la conception sécurisée, la conception Web et la conception de modèles commerciaux.

Enfin, il existe un concept apparenté appelé *modèle d'architecture logicielle,* décrit comme étant l'analogue des modèles de conception appliqués aux architectures logicielles.

Lire Démarrer avec les modèles de conception en ligne: https://riptutorial.com/fr/design-patterns/topic/1012/demarrer-avec-les-modeles-de-conception

Chapitre 2: Adaptateur

Examples

Modèle d'adaptateur (PHP)

Un exemple réel utilisant une expérience scientifique où certaines routines sont effectuées sur différents types de tissus. La classe contient deux fonctions par défaut pour obtenir le tissu ou la routine séparément. Dans une version ultérieure, nous l'avons ensuite adaptée en utilisant une nouvelle classe pour ajouter une fonction qui permet les deux. Cela signifie que nous n'avons pas édité le code d'origine et que nous ne courons donc aucun risque de casser notre classe existante (et de ne pas procéder à de nouveaux tests).

```
class Experiment {
    private $routine;
    private $tissue;
    function __construct($routine_in, $tissue_in) {
        $this->routine = $routine_in;
        $this->tissue  = $tissue_in;
    }
    function getRoutine() {
        return $this->routine;
    }
    function getTissue() {
        return $this->tissue;
    }
}

class ExperimentAdapter {
    private $experiment;
    function __construct(Experiment $experiment_in) {
        $this->experiment = $experiment_in;
    }
    function getRoutineAndTissue() {
        return $this->experiment->getTissue().' (. $this->experiment->getRoutine().')';
    }
}
```

Adaptateur (Java)

Supposons que dans votre base de code actuelle, il existe `MyLogger` interface `MyLogger` comme celle-ci:

```
interface MyLogger {
    void logMessage(String message);
    void logException(Throwable exception);
}
```

Disons que vous avez créé quelques implémentations concrètes, telles que `MyFileLogger` et `MyConsoleLogger` .

Vous avez décidé d'utiliser un cadre pour contrôler la connectivité Bluetooth de votre application. Ce framework contient un `BluetoothManager` avec le constructeur suivant:

```
class BluetoothManager {
    private FrameworkLogger logger;

    public BluetoothManager(FrameworkLogger logger) {
        this.logger = logger;
    }
}
```

Le `BluetoothManager` accepte également un enregistreur, ce qui est génial! Cependant, il attend un enregistreur dont l'interface a été définie par le framework et qui a utilisé une surcharge de méthode au lieu de nommer ses fonctions différemment:

```
interface FrameworkLogger {
    void log(String message);
    void log(Throwable exception);
}
```

Vous souhaitez déjà réutiliser un `MyLogger` implémentations `MyLogger` , mais elles ne correspondent pas à l'interface de `FrameworkLogger` . C'est là que le modèle de conception de l'adaptateur entre en jeu:

```
class FrameworkLoggerAdapter implements FrameworkLogger {
    private MyLogger logger;

    public FrameworkLoggerAdapter(MyLogger logger) {
        this.logger = logger;
    }

    @Override
    public void log(String message) {
        this.logger.logMessage(message);
    }

    @Override
    public void log(Throwable exception) {
        this.logger.logException(exception);
    }
}
```

En définissant une classe d'adaptateur qui implémente l'interface `FrameworkLogger` et accepte une implémentation `MyLogger` la fonctionnalité peut être mappée entre les différentes interfaces. Maintenant, il est possible d'utiliser `BluetoothManager` avec toutes les implémentations `MyLogger` comme ceci:

```
FrameworkLogger fileLogger = new FrameworkLoggerAdapter(new MyFileLogger());
BluetoothManager manager = new BluetoothManager(fileLogger);

FrameworkLogger consoleLogger = new FrameworkLoggerAdapter(new MyConsoleLogger());
BluetoothManager manager2 = new BluetoothManager(consoleLogger);
```

Exemple Java

Un excellent exemple existant du modèle d'adaptateur peut être trouvé dans les classes SWT MouseListener et MouseAdapter .

L'interface MouseListener se présente comme suit:

```
public interface MouseListener extends SWTEventListener {
    public void mouseDoubleClick(MouseEvent e);
    public void mouseDown(MouseEvent e);
    public void mouseUp(MouseEvent e);
}
```

Imaginez maintenant un scénario où vous construisez une interface utilisateur et ajoutez ces écouteurs, mais la plupart du temps, vous ne vous souciez de rien d'autre que lorsque vous cliquez sur un seul clic (mouseUp). Vous ne voudriez pas créer constamment des implémentations vides:

```
obj.addMouseListener(new MouseListener() {

    @Override
    public void mouseDoubleClick(MouseEvent e) {
    }

    @Override
    public void mouseDown(MouseEvent e) {
    }

    @Override
    public void mouseUp(MouseEvent e) {
        // Do the things
    }

});
```

Au lieu de cela, nous pouvons utiliser MouseAdapter:

```
public abstract class MouseAdapter implements MouseListener {
    public void mouseDoubleClick(MouseEvent e) { }
    public void mouseDown(MouseEvent e) { }
    public void mouseUp(MouseEvent e) { }
}
```

En fournissant des implémentations vides par défaut, nous sommes libres de ne remplacer que les méthodes qui nous intéressent de l’adaptateur. En suivant l'exemple ci-dessus:

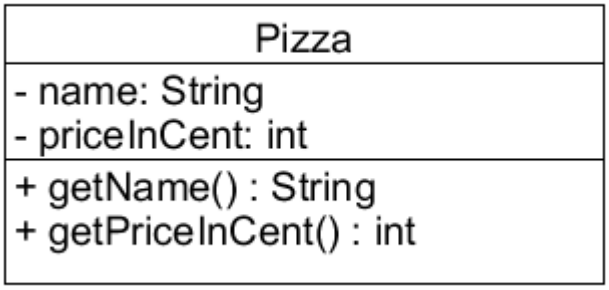
```
obj.addMouseListener(new MouseAdapter() {

    @Override
    public void mouseUp(MouseEvent e) {
        // Do the things
    }

});
```

Adaptateur (UML et exemple de situation)

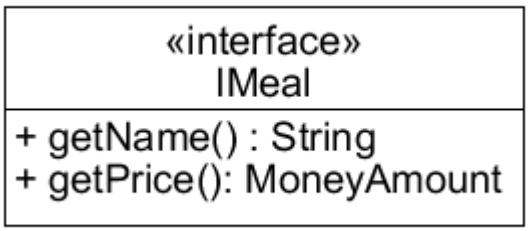
Pour utiliser le modèle d'adaptateur et le type de situation où il peut être appliqué, un exemple simple, très simple et très concret est donné ici. Il n'y aura pas de code ici, juste UML et une description de l'exemple de situation et de son problème. Certes, le contenu UML est écrit comme Java. (Eh bien, le texte de conseil a dit "Les bons exemples sont principalement du code", je pense que les modèles de conception sont assez abstraits pour être introduits d'une manière différente.)

En général, le modèle d'adaptateur est une solution adéquate pour une situation où vous avez des interfaces incompatibles et qu'aucun d'entre eux ne peut être réécrit directement.

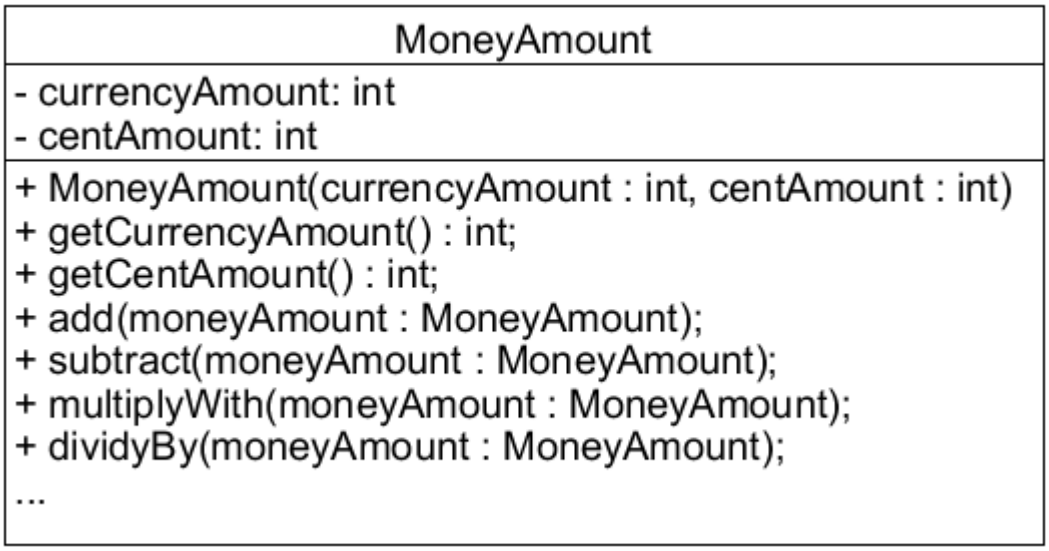
Imaginez que vous dirigez un bon petit service de livraison de pizza. Les clients peuvent commander en ligne sur votre site Web et vous avez un petit système utilisant une classe `Pizza` pour représenter vos pizzas et calculer des factures, des rapports fiscaux et plus encore. Le prix de vos pizzas est donné sous la forme d'un entier représentant le prix en cent (de la devise de votre choix).



Votre service de distribution fonctionne très bien, mais à un moment donné, vous ne pouvez plus gérer vous-même le nombre croissant de clients, mais vous souhaitez tout de même vous développer. Vous décidez d'ajouter vos pizzas au menu d'un grand service de livraison de méta en ligne. Ils offrent beaucoup de repas différents - pas seulement des pizzas - de sorte que leur système utilise plus l'abstraction et dispose d'une interface `IMeal` représentant les repas accompagnant une classe `MoneyAmount` représentant de l'argent.



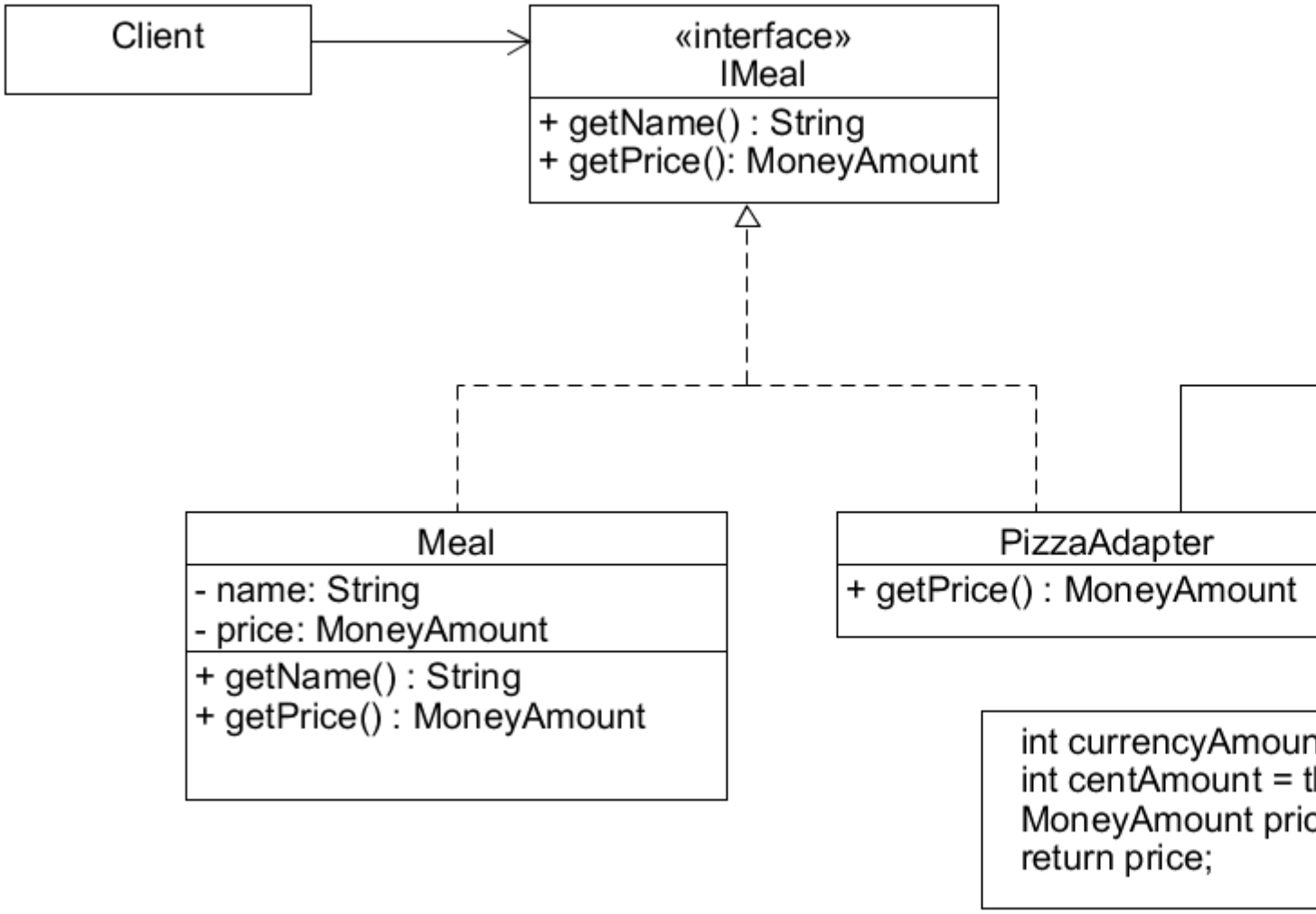
`MoneyAmount` compose de deux entiers en entrée, un pour le montant (ou une devise aléatoire) avant la virgule, et un pour le montant en cents de 0 à 99 après la virgule;



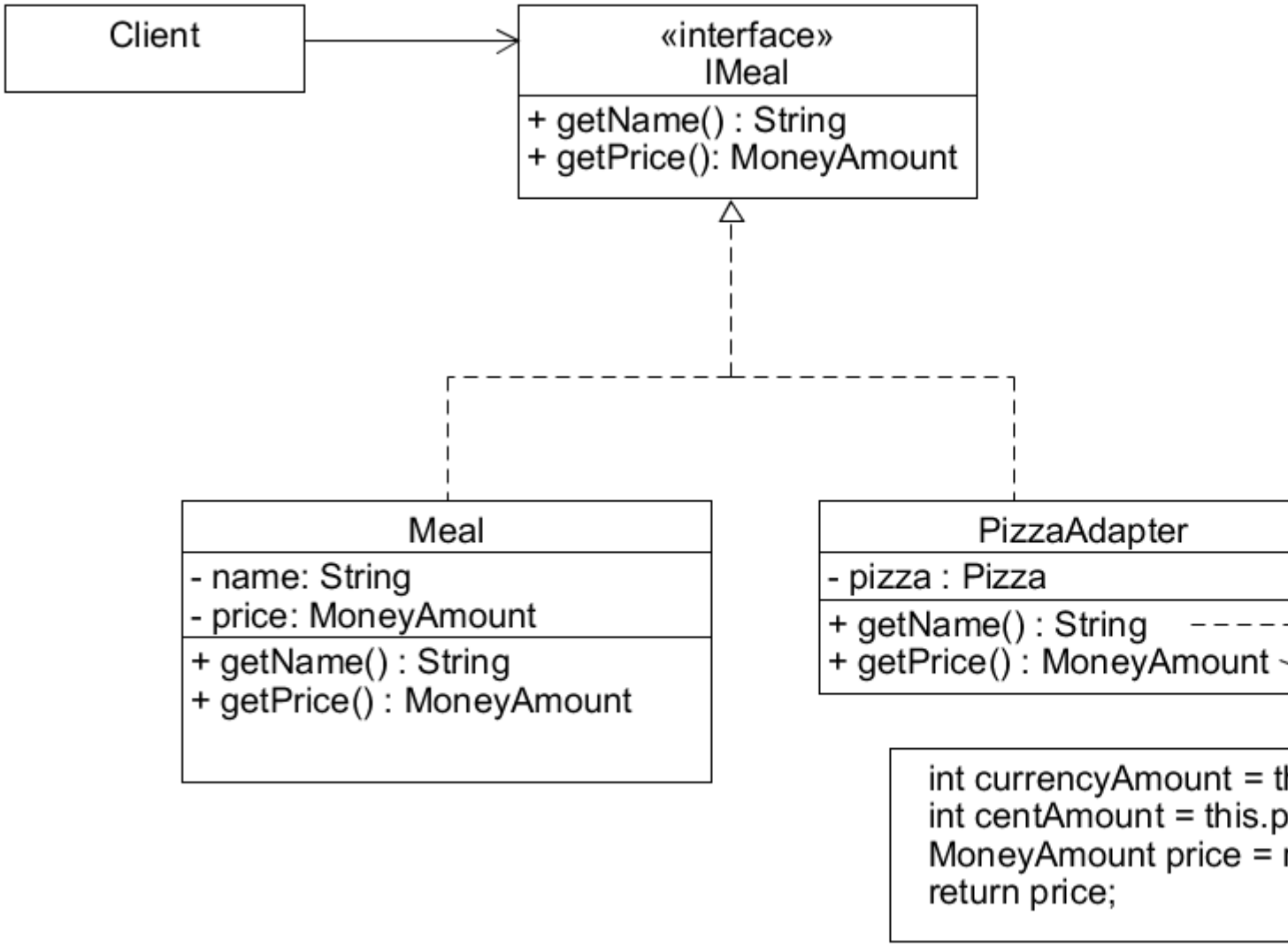
Étant donné que le prix de votre `Pizza` est un nombre entier représentant le prix total en un centime (> 99), il n'est pas compatible avec `IMeal` . C'est à ce moment-là qu'intervient le modèle d'adaptateur: au cas où il vous faudrait trop d'effort pour changer votre propre système ou en créer un nouveau et que vous devez implémenter une interface incompatible, vous pouvez appliquer le modèle d'adaptateur.

Il existe deux manières d'appliquer le modèle: l'adaptateur de classe et l'adaptateur d'objet.

Les deux ont en commun qu'un adaptateur (`PizzaAdapter`) fonctionne comme une sorte de traducteur entre la nouvelle interface et l'adaptee (`Pizza` dans cet exemple). L'adaptateur implémente la nouvelle interface (`IMeal`), puis hérite de `Pizza` et convertit son propre prix d'un entier à deux (adaptateur de classe)



ou possède un objet de type `Pizza` comme attribut et convertit les valeurs de cet objet (adaptateur d'objet).



En appliquant le modèle d'adaptateur, vous allez "traduire" entre des interfaces incompatibles.

Lire Adaptateur en ligne: https://riptutorial.com/fr/design-patterns/topic/4580/adaptateur

Chapitre 3: Chaîne de responsabilité

Examples

Exemple de chaîne de responsabilité (Php)

Une méthode appelée dans un objet montera dans la chaîne d'objets jusqu'à ce que l'on trouve un objet capable de gérer correctement l'appel. Cet exemple particulier utilise des expériences scientifiques avec des fonctions qui peuvent simplement obtenir le titre de l'expérience, l'ID des expériences ou le tissu utilisé dans l'expérience.

```
abstract class AbstractExperiment {
    abstract function getExperiment();
    abstract function getTitle();
}

class Experiment extends AbstractExperiment {
    private $experiment;
    private $tissue;
    function __construct($experiment_in) {
        $this->experiment = $experiment_in;
        $this->tissue = NULL;
    }
    function getExperiment() {
        return $this->experiment;
    }
    //this is the end of the chain - returns title or says there is none
    function getTissue() {
      if (NULL != $this->tissue) {
        return $this->tissue;
      } else {
        return 'there is no tissue applied';
      }
    }
}

class SubExperiment extends AbstractExperiment {
    private $experiment;
    private $parentExperiment;
    private $tissue;
    function __construct($experiment_in, Experiment $parentExperiment_in) {
      $this->experiment = $experiment_in;
      $this->parentExperiment = $parentExperiment_in;
      $this->tissue = NULL;
    }
    function getExperiment() {
      return $this->experiment;
    }
    function getParentExperiment() {
      return $this->parentExperiment;
    }
    function getTissue() {
      if (NULL != $this->tissue) {
        return $this->tissue;
      } else {
```

```
        return $this->parentExperiment->getTissue();
      }
    }
}

//This class and all further sub classes work in the same way as SubExperiment above
class SubSubExperiment extends AbstractExperiment {
    private $experiment;
    private $parentExperiment;
    private $tissue;
    function __construct($experiment_in, Experiment $parentExperiment_in) { //as above }
    function getExperiment() { //same as above }
    function getParentExperiment() { //same as above }
    function getTissue() { //same as above }
}
```

Lire Chaîne de responsabilité en ligne: https://riptutorial.com/fr/design-patterns/topic/6083/chaine-de-responsabilite

Chapitre 4: chargement paresseux

Introduction

le chargement rapide est coûteux ou l'objet à charger peut ne pas être nécessaire du tout

Examples

Chargement paresseux JAVA

Appeler de main ()

```
// Simple lazy loader - not thread safe
HolderNaive holderNaive = new HolderNaive();
Heavy heavy = holderNaive.getHeavy();
```

Heavy.class

```
/**
 *
 * Heavy objects are expensive to create.
 *
 */
public class Heavy {

  private static final Logger LOGGER = LoggerFactory.getLogger(Heavy.class);

  /**
   * Constructor
   */
  public Heavy() {
    LOGGER.info("Creating Heavy ...");
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      LOGGER.error("Exception caught.", e);
    }
    LOGGER.info("... Heavy created");
  }
}
```

HolderNaive.class

```
/**
 *
 * Simple implementation of the lazy loading idiom. However, this is not thread safe.
 *
 */
public class HolderNaive {

  private static final Logger LOGGER = LoggerFactory.getLogger(HolderNaive.class);
```

```
  private Heavy heavy;

  /**
   * Constructor
   */
  public HolderNaive() {
    LOGGER.info("HolderNaive created");
  }

  /**
   * Get heavy object
   */
  public Heavy getHeavy() {
    if (heavy == null) {
      heavy = new Heavy();
    }
    return heavy;
  }
}
```

Lire chargement paresseux en ligne: https://riptutorial.com/fr/design-patterns/topic/9951/chargement-paresseux

Chapitre 5: Façade

Examples

Façade du monde réel (C #)

```
public class MyDataExporterToExcell
{
    public static void Main()
    {
        GetAndExportExcelFacade facade = new GetAndExportExcelFacade();

        facade.Execute();
    }
}

public class GetAndExportExcelFacade
{

    // All services below do something by themselves, determine location for data,
    // get the data, format the data, and export the data
    private readonly DetermineExportDatabaseService _determineExportData = new
DetermineExportDatabaseService();
    private readonly GetRawDataToExportFromDbService _getRawData = new
GetRawDataToExportFromDbService();
    private readonly TransformRawDataForExcelService _transformData = new
TransformRawDataForExcelService();
    private readonly CreateExcelExportService _createExcel = new CreateExcelExportService();

    // the facade puts all the individual pieces together, as its single responsibility.
    public void Execute()
    {
        var dataLocationForExport = _determineExportData.GetDataLocation();
        var rawData = _getRawData.GetDataFromDb(dataLocationForExport);
        var transformedData = _transformData.TransformRawToExportableObject(rawData);
        _createExcel.GenerateExcel("myFilename.xlsx");
    }
}
```

Exemple de façade en java

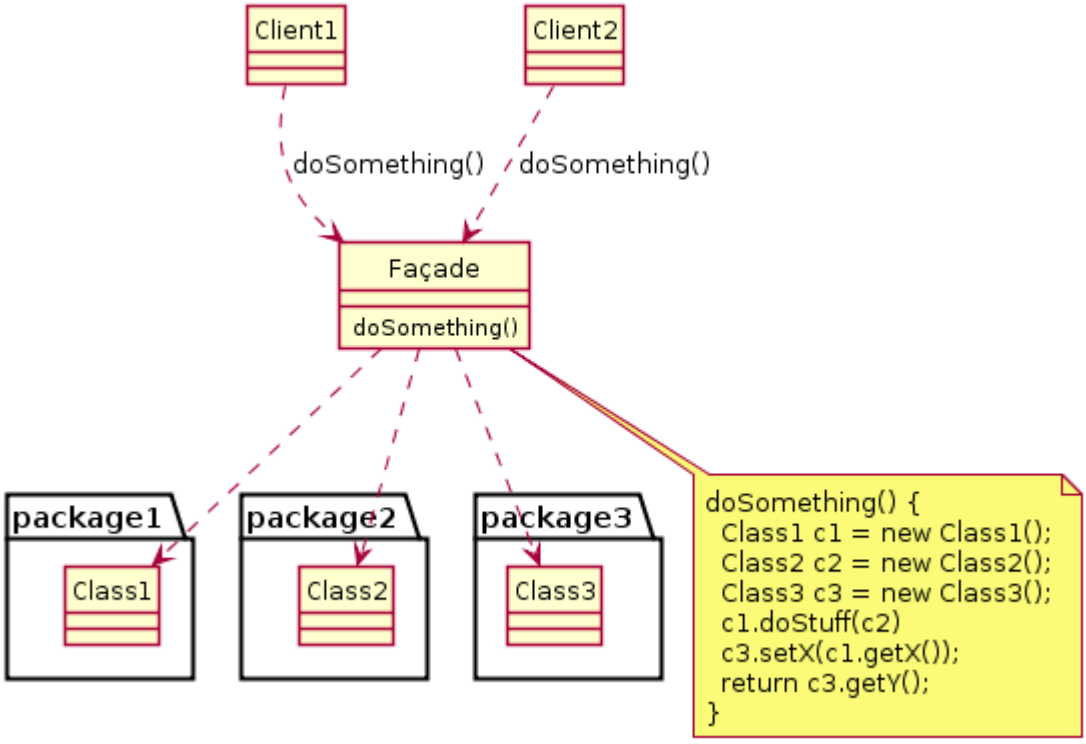
La façade est un modèle de conception structurelle. Il cache les complexités d'un système de grande taille et fournit une interface simple au client.

Le client n'utilise que **Facade** et il n'est pas préoccupé par les interdépendances des sous-systèmes.

Définition de Gang of Four book:

> Fournir une interface unifiée à un ensemble d'interfaces dans un sous-système. La façade définit une interface de niveau supérieur qui facilite l'utilisation du sous-système

Structure:



Exemple du monde réel:

Pensez à certains sites de réservation de voyages comme makemytrip, cleartrip, qui propose des services pour réserver des trains, des vols et des hôtels.

Extrait de code:

```
import java.util.*;

public class TravelFacade{
    FlightBooking flightBooking;
    TrainBooking trainBooking;
    HotelBooking hotelBooking;

    enum BookingType {
        Flight,Train,Hotel,Flight_And_Hotel,Train_And_Hotel;
    };

    public TravelFacade(){
        flightBooking = new FlightBooking();
        trainBooking = new TrainBooking();
        hotelBooking = new HotelBooking();
    }
    public void book(BookingType type, BookingInfo info){
        switch(type){
            case Flight:
                // book flight;
                flightBooking.bookFlight(info);
                return;
            case Hotel:
                // book hotel;
                hotelBooking.bookHotel(info);
                return;
            case Train:
                // book Train;
```

```
                trainBooking.bookTrain(info);
                return;
            case Flight_And_Hotel:
                // book Flight and Hotel
                flightBooking.bookFlight(info);
                hotelBooking.bookHotel(info);
                return;
             case Train_And_Hotel:
                // book Train and Hotel
                trainBooking.bookTrain(info);
                hotelBooking.bookHotel(info);
                return;
        }
    }
}
class BookingInfo{
    String source;
    String destination;
    Date    fromDate;
    Date     toDate;
    List<PersonInfo> list;
}
class PersonInfo{
    String name;
    int       age;
    Address address;
}
class Address{

}
class FlightBooking{
    public FlightBooking(){

    }
    public void bookFlight(BookingInfo info){

    }
}
class HotelBooking{
    public HotelBooking(){

    }
    public void bookHotel(BookingInfo info){

    }
}
class TrainBooking{
    public TrainBooking(){

    }
    public void bookTrain(BookingInfo info){

    }
}
```

Explication:

1. `FlightBooking, TrainBooking and HotelBooking` sont différents sous-systèmes de grands systèmes: `TravelFacade`

2. `TravelFacade` offre une interface simple pour réserver l'une des options ci-dessous

```
Flight Booking
Train Booking
Hotel Booking
Flight + Hotel booking
Train + Hotel booking
```

3. livre API de TravelFacade appels internes en dessous des API des sous-systèmes

```
flightBooking.bookFlight
trainBooking.bookTrain(info);
hotelBooking.bookHotel(info);
```

4. De cette façon, `TravelFacade` fournit des API plus simples et plus faciles sans exposer les API du sous-système.

Applicabilité et cas d'utilisation (de Wikipedia):

1. Une interface simple est nécessaire pour accéder à un système complexe.
2. Les abstractions et les implémentations d'un sous-système sont étroitement liées.
3. Besoin d'un point d'entrée pour chaque niveau de logiciel en couches.
4. Le système est très complexe ou difficile à comprendre.

Lire Façade en ligne: https://riptutorial.com/fr/design-patterns/topic/3516/facade

Chapitre 6: Injection de dépendance

Introduction

L'idée générale derrière l'injection de dépendance est que vous concevez votre application autour de composants faiblement couplés tout en adhérant au principe d'inversion de dépendance. En ne dépendant pas d'implémentations concrètes, permet de concevoir des systèmes hautement flexibles.

Remarques

L'idée de base de l'injection de dépendance est de créer un code plus faiblement couplé. Lorsqu'une classe, plutôt que de mettre à jour ses propres dépendances, prend plutôt ses dépendances, la classe devient plus simple à tester en tant qu'unité (test unitaire).

Pour développer davantage le couplage lâche - l'idée est que les classes deviennent dépendantes des abstractions, plutôt que des concrétions. Si la classe `A` dépend d'une autre classe concrète `B` , alors il n'y a pas de test réel de `A` sans `B` Bien que ce type de test puisse être correct, il ne se prête pas au code testable de l'unité. Une conception faiblement couplée définirait une abstraction `IB` (par exemple) dont dépendrait la classe `A` On peut alors simuler `IB` pour fournir un comportement testable, plutôt que de compter sur l'implémentation réelle de `B` pour pouvoir fournir des scénarios testables à `A`

Exemple à couplage étroit (C #):

```
public class A
{
    public void DoStuff()
    {
        B b = new B();
        b.Foo();
    }
}
```

Dans ce qui précède, la classe `A` dépend de `B` Il n'y a pas de test `A` sans le béton `B` Bien que ce soit bien dans un scénario de test d'intégration, il est difficile de test unitaire `A` .

Une implémentation plus souple de ce qui précède pourrait ressembler à ceci:

```
public interface IB
{
    void Foo();
}

public class A
{
    private readonly IB _iB;

    public A(IB iB)
```

```
    {
        _iB = iB;
    }

    public void DoStuff()
    {
        _b.Foo();
    }
}
```

Les deux implémentations semblent assez similaires, il y a cependant une différence importante. La classe `A` ne dépend plus directement de la classe `B` , elle dépend maintenant de `IB` . La classe `A` n'a plus la responsabilité de renouveler ses propres dépendances - elles doivent maintenant être fournies ***à*** `A`

Examples

Injection Setter (C #)

```
public class Foo
{
    private IBar _iBar;
    public IBar iBar { set { _iBar = value; } }

    public void DoStuff()
    {
        _iBar.DoSomething();
    }
}

public interface IBar
{
    void DoSomething();
}
```

Constructeur Injection (C #)

```
public class Foo
{
    private readonly IBar _iBar;

    public Foo(IBar iBar)
    {
        _iBar = iBar;
    }

    public void DoStuff()
    {
        _bar.DoSomething();
    }
}

public interface IBar
{
    void DoSomething();
```

```
}
```

Lire Injection de dépendance en ligne: https://riptutorial.com/fr/design-patterns/topic/1723/injection-de-dependance

Chapitre 7: Méthode d'usine statique

Examples

Méthode d'usine statique

Nous pouvons fournir un nom significatif à nos constructeurs.

Nous pouvons fournir plusieurs constructeurs avec le même nombre et le même type de paramètres, ce que nous avons vu précédemment, ce que nous ne pouvons pas faire avec les constructeurs de classes.

```
public class RandomIntGenerator {
    private final int min;
    private final int max;

    private RandomIntGenerator(int min, int max) {
        this.min = min;
        this.max = max;
    }

    public static RandomIntGenerator between(int max, int min) {
        return new RandomIntGenerator(min, max);
    }

    public static RandomIntGenerator biggerThan(int min) {
        return new RandomIntGenerator(min, Integer.MAX_VALUE);
    }

    public static RandomIntGenerator smallerThan(int max) {
        return new RandomIntGenerator(Integer.MIN_VALUE, max);
    }

    public int next() {...}
}
```

Cacher l'accès direct au constructeur

Nous pouvons éviter de fournir un accès direct aux constructeurs exigeants en ressources, comme pour les bases de données. classe publique DbConnection {private static final int MAX_CONNS = 100; private static int totalConnections = 0;

```
   private static Set<DbConnection> availableConnections = new HashSet<DbConnection>();

   private DbConnection()
   {
     // ...
     totalConnections++;
   }

   public static DbConnection getDbConnection()
   {
```

```
    if(totalConnections < MAX_CONNS)
    {
      return new DbConnection();
    }

    else if(availableConnections.size() > 0)
    {
        DbConnection dbc = availableConnections.iterator().next();
        availableConnections.remove(dbc);
        return dbc;
    }

    else {
      throw new NoDbConnections();
    }
  }

  public static void returnDbConnection(DbConnection dbc)
  {
    availableConnections.add(dbc);
    //...
  }
}
```

Méthode d'usine statique C

La *méthode d'usine statique* est une variante du modèle de *méthode d'usine* . Il est utilisé pour créer des objets sans avoir à appeler le constructeur vous-même.

Quand utiliser la méthode d'usine statique

- si vous voulez donner un nom significatif à la méthode qui génère votre objet.
- si vous voulez éviter la création d'objets trop complexes, voir Tuple Msdn .
- si vous souhaitez limiter le nombre d'objets créés (mise en cache)
- si vous voulez retourner un objet de n'importe quel sous-type de leur type de retour.

Il y a quelques inconvénients comme

- Les classes sans constructeur public ou protégé ne peuvent pas être initialisées dans la méthode de fabrique statique.
- Les méthodes d'usine statiques sont comme les méthodes statiques normales, elles ne peuvent donc pas être distinguées des autres méthodes statiques (cela peut varier d'un IDE à l'autre)

Exemple

Pizza.cs

```
public class Pizza
{
    public int SizeDiameterCM
    {
        get;
        private set;
```

```
    }

    private Pizza()
    {
        SizeDiameterCM = 25;
    }

    public static Pizza GetPizza()
    {
        return new Pizza();
    }

    public static Pizza GetLargePizza()
    {
        return new Pizza()
        {
            SizeDiameterCM = 35
        };
    }

    public static Pizza GetSmallPizza()
    {
        return new Pizza()
        {
            SizeDiameterCM = 28
        };
    }

    public override string ToString()
    {
        return String.Format("A Pizza with a diameter of {0} cm",SizeDiameterCM);
    }
}
```

Program.cs

```
class Program
{
    static void Main(string[] args)
    {
        var pizzaNormal = Pizza.GetPizza();
        var pizzaLarge = Pizza.GetLargePizza();
        var pizzaSmall = Pizza.GetSmallPizza();


        String pizzaString = String.Format("{0} and {1} and {2}",pizzaSmall.ToString(),
pizzaNormal.ToString(), pizzaLarge.ToString());
        Console.WriteLine(pizzaString);
    }
}
```

Sortie

> Une pizza d'un diamètre de 28 cm et une pizza d'un diamètre de 25 cm et une pizza d'un diamètre de 35 cm

Lire Méthode d'usine statique en ligne: https://riptutorial.com/fr/design-patterns/topic/6024/methode-d-usine-statique

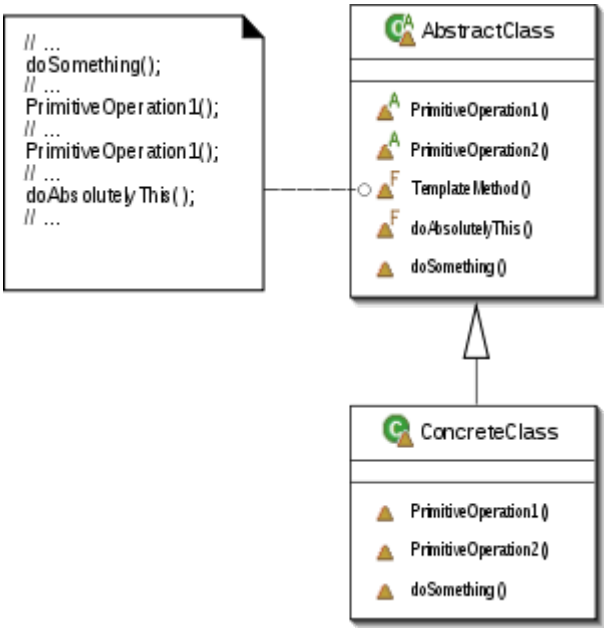
Chapitre 8: Méthode du modèle

Examples

Implémentation de la méthode de modèle en Java

Le modèle de méthode de modèle est un modèle de conception comportementale qui définit le squelette de programme d'un algorithme dans une opération, en reportant certaines étapes à des sous-classes.

Structure:



Notes clés:

1. La méthode du modèle utilise l'héritage
2. La méthode Template implémentée par la classe de base ne doit pas être remplacée. De cette façon, la structure de l'algorithme est contrôlée par la super-classe et les détails sont implémentés dans les sous-classes.

Exemple de code:

```
import java.util.List;

class GameRule{

}
class GameInfo{
    String gameName;
    List<String> players;
    List<GameRule> rules;
}

abstract class Game{
```

```
    protected GameInfo info;
    public Game(GameInfo info){
        this.info = info;
    }
    public abstract void createGame();
    public abstract void makeMoves();
    public abstract void applyRules();

    /* playGame is template method. This algorithm skeleton can't be changed by sub-classes.
sub-class can change
       the behaviour only of steps like createGame() etc. */

    public void playGame(){
        createGame();
        makeMoves();
        applyRules();
        closeGame();
    }
    protected void closeGame(){
        System.out.println("Close game:"+this.getClass().getName());
        System.out.println("--------------------");
    }
}
class Chess extends Game{
    public Chess(GameInfo info){
        super(info);
    }
    public void createGame(){
        // Use GameInfo and create Game
        System.out.println("Creating Chess game");
    }
    public void makeMoves(){
        System.out.println("Make Chess moves");
    }
    public void applyRules(){
        System.out.println("Apply Chess rules");
    }
}
class Checkers extends Game{
    public Checkers(GameInfo info){
        super(info);
    }
    public void createGame(){
        // Use GameInfo and create Game
        System.out.println("Creating Checkers game");
    }
    public void makeMoves(){
        System.out.println("Make Checkers moves");
    }
    public void applyRules(){
        System.out.println("Apply Checkers rules");
    }

}
class Ludo extends Game{
    public Ludo(GameInfo info){
        super(info);
    }
    public void createGame(){
        // Use GameInfo and create Game
        System.out.println("Creating Ludo game");
```

```
    }
    public void makeMoves(){
        System.out.println("Make Ludo moves");
    }
    public void applyRules(){
        System.out.println("Apply Ludo rules");
    }
}

public class TemplateMethodPattern{
    public static void main(String args[]){
        System.out.println("--------------------");

        Game game = new Chess(new GameInfo());
        game.playGame();

        game = new Ludo(new GameInfo());
        game.playGame();

        game = new Checkers(new GameInfo());
        game.playGame();
    }
}
```

Explication:

1. `Game` est une super classe `abstract` , qui définit une méthode de template: `playGame()`
2. Le squelette de `playGame()` est défini dans la classe de base: `Game`
3. Les sous-classes telles que `Chess, Ludo` et `Checkers` ne peuvent pas changer le squelette de `playGame()` . Mais ils peuvent modifier le comportement de certaines étapes comme

```
createGame();
makeMoves();
applyRules();
```

sortie:

```
--------------------
Creating Chess game
Make Chess moves
Apply Chess rules
Close game:Chess
--------------------
Creating Ludo game
Make Ludo moves
Apply Ludo rules
Close game:Ludo
--------------------
Creating Checkers game
Make Checkers moves
Apply Checkers rules
Close game:Checkers
--------------------
```

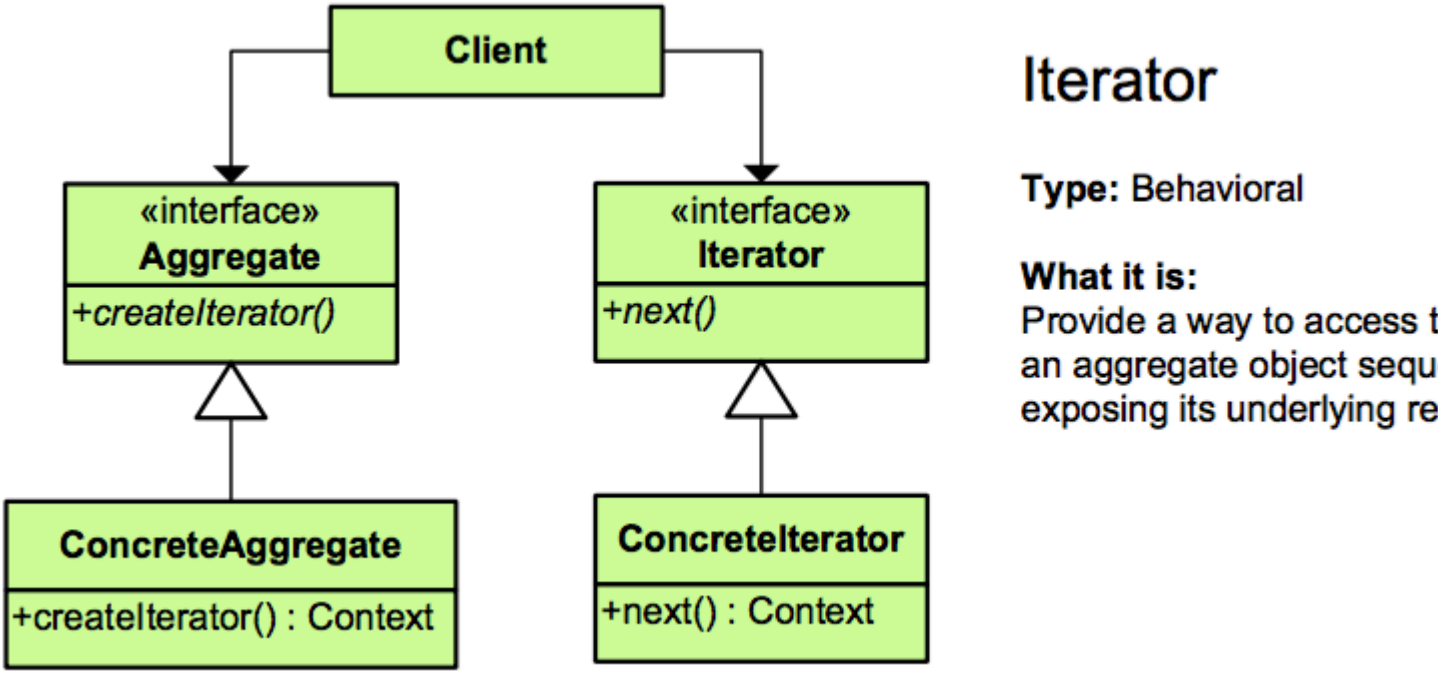
Lire Méthode du modèle en ligne: https://riptutorial.com/fr/design-patterns/topic/4867/methode-du-

modele

Chapitre 9: Modèle d'itérateur

Examples

Le modèle d'itérateur



Les collections sont l'une des structures de données les plus couramment utilisées en génie logiciel. Une collection n'est qu'un groupe d'objets. Une collection peut être une liste, un tableau, une carte, un arbre ou tout autre objet. Ainsi, une collection devrait fournir un moyen d'accéder à ses éléments sans exposer sa structure interne. Nous devrions pouvoir le parcourir dans le même, quel que soit le type de collection.

L'idée du modèle d'itérateur est de prendre la responsabilité d'accéder à l'objet d'une collection et de le placer dans un objet itérateur. En retour, l'objet itérateur maintiendra l'ordre d'itération, conservera une trace de l'élément en cours et aura un moyen de récupérer l'élément suivant.

Habituellement, la classe de collection comporte deux composants: la classe elle-même et son `Iterator`.

```
public interface Iterator {
   public boolean hasNext();
   public Object next();
}

public class FruitsList {
    public String fruits[] = {"Banana", "Apple", "Pear", "Peach", "Blueberry"};

    public Iterator getIterator() {
       return new FruitIterator();
```

```
    }

    private class FruitIterator implements Iterator {
       int index;

       @Override
       public boolean hasNext() {
           return index < fruits.length;
       }

       @Override
       public Object next() {

          if(this.hasNext()) {
            return names[index++];
          }
          return null;
       }
    }
}
```

Lire Modèle d'itérateur en ligne: https://riptutorial.com/fr/design-patterns/topic/7061/modele-d-iterateur

Chapitre 10: Modèle d'objet nul

Remarques

Null Object est un objet sans valeur référencée ou avec un comportement neutre défini. Son but est de supprimer le besoin de vérification de pointeur / référence null.

Examples

Modèle d'objet nul (C ++)

En supposant une classe abstraite:

```
class ILogger {
    virtual ~ILogger() = default;
    virtual Log(const std::string&) = 0;
};
```

Au lieu de

```
void doJob(ILogger* logger) {
    if (logger) {
         logger->Log("[doJob]:Step 1");
    }
    // ...
    if (logger) {
         logger->Log("[doJob]:Step 2");
    }
    // ...
    if (logger) {
         logger->Log("[doJob]:End");
    }
}

void doJobWithoutLogging()
{
    doJob(nullptr);
}
```

Vous pouvez créer un enregistreur d'objets Null:

```
class NullLogger : public ILogger
{
    void Log(const std::string&) override { /* Empty */ }
};
```

puis changez `doJob` dans ce qui suit:

```
void doJob(ILogger& logger) {
    logger.Log("[doJob]:Step1");
```

```
    // ...
    logger.Log("[doJob]:Step 2");
    // ...
    logger.Log("[doJob]:End");
}

void doJobWithoutLogging()
{
    NullLogger logger;
    doJob(logger);
}
```

Objet Null Java utilisant enum

Étant donné une interface:

```
public interface Logger {
   void log(String message);
}
```

Plutôt que d'utiliser:

```
public void doJob(Logger logger) {
    if (logger != null) {
       logger.log("[doJob]:Step 1");
    }
    // ...
    if (logger != null) {
       logger.log("[doJob]:Step 2");
    }
    // ...
    if (logger != null) {
       logger.log("[doJob]:Step 3");
    }
}

public void doJob() {
    doJob(null); // Without Logging
}
```

Étant donné que les objets nuls n'ont pas d'état, il est logique d'utiliser un singulier enum, ce qui signifie qu'un objet null doit être implémenté comme suit:

```
public enum NullLogger implements Logger {
    INSTANCE;

    @Override
    public void log(String message) {
        // Do nothing
    }
}
```

Vous pouvez alors éviter les vérifications nulles.

```
public void doJob(Logger logger) {
```

```
    logger.log("[doJob]:Step 1");
    // ...
    logger.log("[doJob]:Step 2");
    // ...
    logger.log("[doJob]:Step 3");
}

public void doJob() {
    doJob(NullLogger.INSTANCE);
}
```

Lire Modèle d'objet nul en ligne: https://riptutorial.com/fr/design-patterns/topic/6177/modele-d-objet-nul

Chapitre 11: Modèle de commande

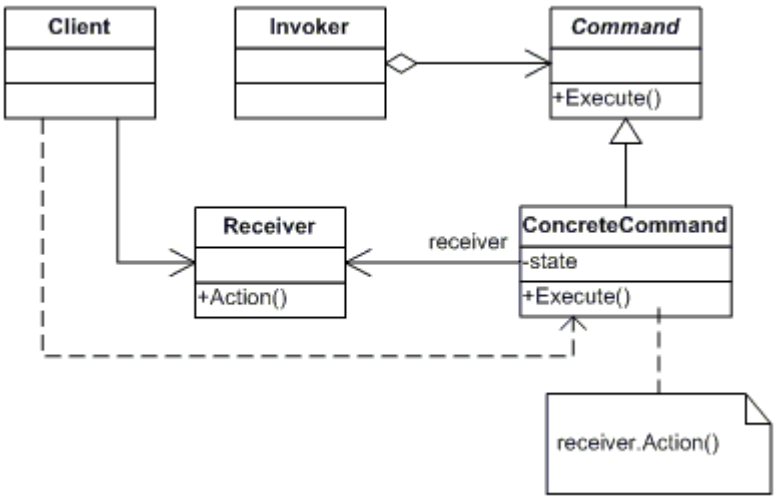
Examples

Exemple de modèle de commande en Java

définition de wikipedia :

> Le modèle de commande est un modèle de conception comportementale dans lequel un objet est utilisé pour encapsuler toutes les informations nécessaires pour effectuer une action ou déclencher un événement ultérieurement.

Diagramme UML de dofactory :



Composants de base et workflow:

1. `Command` déclare une interface pour les commandes abstraites telles que `execute()`
2. `Receiver` sait comment exécuter une commande particulière
3. `Invoker` contient `ConcreteCommand` , qui doit être exécuté
4. `Client` crée `ConcreteCommand` et attribue le `Receiver`
5. `ConcreteCommand` définit la liaison entre la `Command` et le `Receiver`

De cette manière, le modèle de commande dissocie **Sender** (Client) de **Receiver** via **Invoker** . **Invoker** a une connaissance complète de la **commande** à exécuter et la **commande** sait quel **récepteur** invoquer pour exécuter une opération particulière.

Extrait de code:

```
interface Command {
    void execute();
}
class Receiver {
    public void switchOn(){
        System.out.println("Switch on from:"+this.getClass().getSimpleName());
    }
```

```
}
class OnCommand implements Command{
    private Receiver receiver;

    public OnCommand(Receiver receiver){
        this.receiver = receiver;
    }
    public void execute(){
        receiver.switchOn();
    }
}
class Invoker {
    private Command command;

    public Invoker(Command command){
        this.command = command;
    }
    public void execute(){
        this.command.execute();
    }
}

class TV extends Receiver{

    public String toString(){
        return this.getClass().getSimpleName();
    }
}
class DVDPlayer extends Receiver{

    public String toString(){
        return this.getClass().getSimpleName();
    }
}

public class CommandDemoEx{
    public static void main(String args[]){
        // On command for TV with same invoker
        Receiver receiver = new TV();
        Command onCommand = new OnCommand(receiver);
        Invoker invoker = new Invoker(onCommand);
        invoker.execute();

        // On command for DVDPlayer with same invoker
        receiver = new DVDPlayer();
        onCommand = new OnCommand(receiver);
        invoker = new Invoker(onCommand);
        invoker.execute();
    }
}
```

sortie:

```
Switch on from:TV
Switch on from:DVDPlayer
```

Explication:

Dans cet exemple,

1. **L'** interface de **commande** définit la méthode `execute()` .
2. **OnCommand** est **ConcreteCommand** , qui implémente la méthode `execute()` .
3. **Receiver** est la classe de base.
4. **TV** et **DVDPlayer** sont deux types de **récepteurs** , qui sont transmis à ConcreteCommand comme OnCommand.
5. **Invoker** contient la **commande** . C'est la clé pour séparer l'expéditeur du **récepteur** .
6. **Invoker** reçoit **OnCommand** -> qui appelle **Receiver** (TV) pour exécuter cette commande.

En utilisant Invoker, vous pouvez activer TV et DVDPlayer. Si vous prolongez ce programme, vous désactivez également TV et DVDPlayer.

Principaux cas d'utilisation:

1. Pour implémenter le mécanisme de rappel
2. Pour implémenter les fonctionnalités d'annulation et de restauration
3. Maintenir un historique des commandes

Lire Modèle de commande en ligne: https://riptutorial.com/fr/design-patterns/topic/2677/modele-de-commande

Chapitre 12: Modèle de conception DAO (Data Access Object)

Examples

Modèle de conception de l'objet d'accès aux données J2EE avec Java

Le modèle de conception *DAO (Data Access Object)* est un modèle de conception J2EE standard.

Dans cette conception, les données de modèle sont accessibles via des classes contenant des méthodes permettant d'accéder aux données provenant de bases de données ou d'autres sources, appelées *objets d'accès aux données* . La pratique standard suppose qu'il existe des classes POJO. DAO peut être mélangé avec d'autres modèles de conception pour accéder à des données, comme avec MVC (contrôleur de vue de modèle), modèles de commandes, etc.

Voici un exemple de modèle de conception DAO. Il a une classe **Employee** , un DAO pour Employee appelé **EmployeeDAO** et une classe **ApplicationView** pour illustrer les exemples.

Employee.java

```
public class Employee {
    private Integer employeeId;
    private String firstName;
    private String lastName;
    private Integer salary;

    public Employee(){

    }

    public Employee(Integer employeeId, String firstName, String lastName, Integer salary) {
        super();
        this.employeeId = employeeId;
        this.firstName = firstName;
        this.lastName = lastName;
        this.setSalary(salary);
    }

    //standard setters and getters

}
```

EmployeeDAO

```
public class EmployeeDAO {

    private List<Employee> employeeList;

    public EmployeeDAO(List<Employee> employeeList){
        this.employeeList = employeeList;
    }
```

```
    public List<Employee> getAllEmployees(){
        return employeeList;
    }

    //add other retrieval methods as you wish
    public Employee getEmployeeWithMaxSalary(){
        Employee employee = employeeList.get(0);
        for (int i = 0; i < employeeList.size(); i++){
            Employee e = employeeList.get(i);
            if (e.getSalary() > employee.getSalary()){
                employee = e;
            }
        }

        return employee;
    }

}
```

ApplicationView.java

```
public class ApplicationView {

    public static void main(String[] args) {
        // See all the employees with data access object

        List<Employee> employeeList = setEmployeeList();
        EmployeeDAO eDAO = new EmployeeDAO(employeeList);

        List<Employee> allEmployees = eDAO.getAllEmployees();

        for (int i = 0; i < allEmployees.size(); i++) {
            Employee e = employeeList.get(i);
            System.out.println("UserId: " + e.getEmployeeId());
        }

        Employee employeeWithMaxSalary = eDAO.getEmployeeWithMaxSalary();

        System.out.println("Maximum Salaried Employee" + " FirstName:" +
employeeWithMaxSalary.getFirstName()
        + " LastName:" + employeeWithMaxSalary.getLastName() + " Salary: " +
employeeWithMaxSalary.getSalary());

    }

    public static List<Employee> setEmployeeList() {
        Employee employee1 = new Employee(1, "Pete", "Samprus", 3000);
        Employee employee2 = new Employee(2, "Peter", "Russell", 4000);
        Employee employee3 = new Employee(3, "Shane", "Watson", 2000);

        List<Employee> employeeList = new ArrayList<>();
        employeeList.add(employee1);
        employeeList.add(employee2);
        employeeList.add(employee3);
        return employeeList;
    }

}
```

Par conséquent, nous avons un exemple où nous voyons comment utiliser le modèle de conception d'objet d'accès aux données.

Lire Modèle de conception DAO (Data Access Object) en ligne: https://riptutorial.com/fr/design-patterns/topic/6351/modele-de-conception-dao--data-access-object-

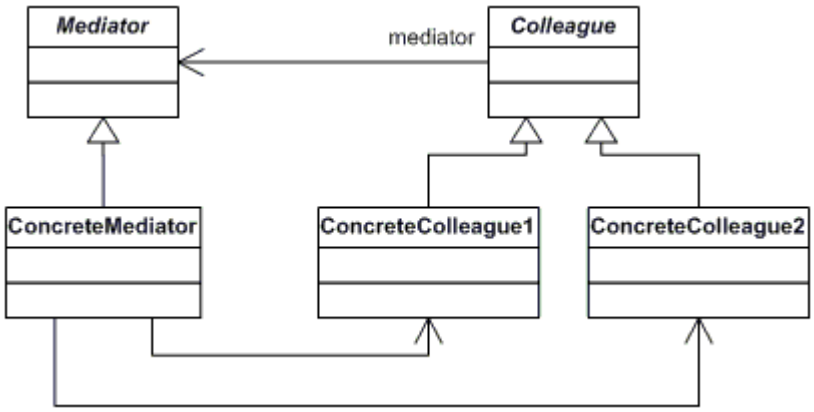
Chapitre 13: Modèle de médiateur

Examples

Exemple de modèle de médiateur en Java

Le modèle de *médiateur* définit un objet (Mediator) qui encapsule la manière dont un ensemble d'objets interagit. Il permet une communication plusieurs à plusieurs.

Diagramme UML:



Composants clés:

`Mediator`: Définit une interface pour la communication entre collègues.

`Colleague` : est un cours abstrait, qui définit les événements à communiquer entre collègues.

`ConcreteMediator` : `ConcreteMediator` œuvre un comportement coopératif en coordonnant les objets `Colleague` et maintient ses collègues

`ConcreteColleague` : Implémente les opérations de notification reçues via `Mediator` , qui ont été générées par d'autres `Colleague`

Un exemple concret:

Vous gérez un réseau d'ordinateurs dans la topologie `Mesh` .

Un réseau maillé est une topologie de réseau dans laquelle chaque nœud relaie des données pour le réseau. Tous les nœuds de maillage coopèrent dans la distribution des données dans le réseau.

Si un nouvel ordinateur est ajouté ou si l'ordinateur existant est supprimé, tous les autres ordinateurs de ce réseau doivent connaître ces deux événements.

Voyons comment le motif Mediator s'y intègre.

Extrait de code:

```
import java.util.List;
import java.util.ArrayList;

/* Define the contract for communication between Colleagues.
   Implementation is left to ConcreteMediator */
interface Mediator{
    void register(Colleague colleague);
    void unregister(Colleague colleague);
}
/* Define the contract for notification events from Mediator.
   Implementation is left to ConcreteColleague
*/
abstract class Colleague{
    private Mediator mediator;
    private String name;

    public Colleague(Mediator mediator,String name){
        this.mediator = mediator;
        this.name = name;
    }
    public String toString(){
        return name;
    }
    public abstract void receiveRegisterNotification(Colleague colleague);
    public abstract void receiveUnRegisterNotification(Colleague colleague);
}
/*  Process notification event raised by other Colleague through Mediator.
*/
class ComputerColleague extends Colleague {
    private Mediator mediator;

    public ComputerColleague(Mediator mediator,String name){
        super(mediator,name);
    }
    public  void receiveRegisterNotification(Colleague colleague){
        System.out.println("New Computer register event with name:"+colleague+
        ": received @"+this);
        // Send further messages to this new Colleague from now onwards
    }
    public  void receiveUnRegisterNotification(Colleague colleague){
        System.out.println("Computer left unregister event with name:"+colleague+
        ":received @"+this);
        // Do not send further messages to this Colleague from now onwards
    }
}
/* Act as a central hub for communication between different Colleagues.
   Notifies all Concrete Colleagues on occurrence of an event
*/
class NetworkMediator implements Mediator{
    List<Colleague> colleagues = new ArrayList<Colleague>();

    public NetworkMediator(){

    }

    public void register(Colleague colleague){
        colleagues.add(colleague);
        for (Colleague other : colleagues){
            if ( other != colleague){
                other.receiveRegisterNotification(colleague);
            }
```

```
        }
    }
    public void unregister(Colleague colleague){
        colleagues.remove(colleague);
        for (Colleague other : colleagues){
            other.receiveUnRegisterNotification(colleague);
        }
    }
}

public class MediatorPatternDemo{
    public static void main(String args[]){
        Mediator mediator = new NetworkMediator();
        ComputerColleague colleague1 = new ComputerColleague(mediator,"Eagle");
        ComputerColleague colleague2 = new ComputerColleague(mediator,"Ostrich");
        ComputerColleague colleague3 = new ComputerColleague(mediator,"Penguin");
        mediator.register(colleague1);
        mediator.register(colleague2);
        mediator.register(colleague3);
        mediator.unregister(colleague1);
    }
}
```

sortie:

```
New Computer register event with name:Ostrich: received @Eagle
New Computer register event with name:Penguin: received @Eagle
New Computer register event with name:Penguin: received @Ostrich
Computer left unregister event with name:Eagle:received @Ostrich
Computer left unregister event with name:Eagle:received @Penguin
```

Explication:

1. `Eagle` est ajouté au réseau lors du premier événement de registre. Aucune notification à d'autres collègues puisque Eagle est le premier.
2. Lorsque l' `Ostrich` est ajoutée au réseau, `Eagle` est averti: la ligne 1 de la sortie est rendue maintenant.
3. Lorsque `Penguin` est ajouté au réseau, `Eagle` et `Ostrich` ont été notifiés: les lignes 2 et 3 de la sortie sont maintenant affichées.
4. Lorsque `Eagle` quitté le réseau par le biais d'un événement de désinscription, l' `Ostrich` et le `Penguin` ont été notifiés. Les lignes 4 et 5 de la sortie sont maintenant affichées.

Lire Modèle de médiateur en ligne: https://riptutorial.com/fr/design-patterns/topic/6184/modele-de-mediateur

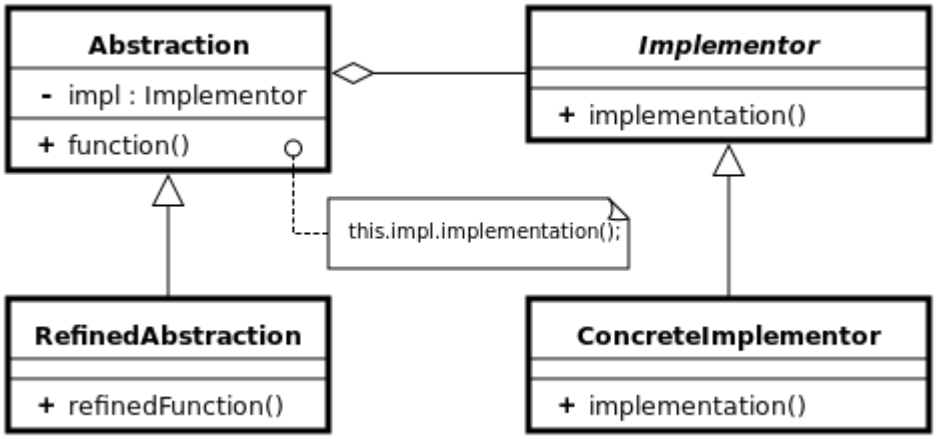
Chapitre 14: Modèle de pont

Examples

Implémentation d'un modèle de pont en Java

Le modèle de pont sépare l'abstraction de la mise en œuvre pour que les deux puissent varier indépendamment. Cela a été réalisé avec la composition plutôt que l'héritage.

Diagramme UML Bridge de Wikipédia:



Vous avez quatre composants dans ce modèle.

`Abstraction` : définit une interface

`RefinedAbstraction` : implémente l'abstraction:

`Implementor` : définit une interface pour la mise en œuvre

`ConcreteImplementor` : Il implémente l'interface Implementor.

`The crux of Bridge pattern :` Deux hiérarchies de classes orthogonales utilisant la composition (et pas d'héritage). La hiérarchie d'abstraction et la hiérarchie d'implémentation peuvent varier de manière indépendante. La mise en œuvre ne fait jamais référence à l'abstraction. Abstraction contient une interface d'implémentation en tant que membre (via composition). Cette composition réduit encore un niveau de hiérarchie d'héritage.

Mot réel Cas d'utilisation:

Permettre à différents véhicules d'avoir les deux versions du système de vitesse manuel et automatique.

Exemple de code:

```
/* Implementor interface*/
interface Gear{
    void handleGear();
}

/* Concrete Implementor - 1 */
class ManualGear implements Gear{
    public void handleGear(){
        System.out.println("Manual gear");
    }
}
/* Concrete Implementor - 2 */
class AutoGear implements Gear{
    public void handleGear(){
        System.out.println("Auto gear");
    }
}
/* Abstraction (abstract class) */
abstract class Vehicle {
    Gear gear;
    public Vehicle(Gear gear){
        this.gear = gear;
    }
    abstract void addGear();
}
/* RefinedAbstraction - 1*/
class Car extends Vehicle{
    public Car(Gear gear){
        super(gear);
        // initialize various other Car components to make the car
    }
    public void addGear(){
        System.out.print("Car handles ");
        gear.handleGear();
    }
}
/* RefinedAbstraction - 2 */
class Truck extends Vehicle{
    public Truck(Gear gear){
        super(gear);
        // initialize various other Truck components to make the car
    }
    public void addGear(){
        System.out.print("Truck handles " );
        gear.handleGear();
    }
}
/* Client program */
public class BridgeDemo {
    public static void main(String args[]){
        Gear gear = new ManualGear();
        Vehicle vehicle = new Car(gear);
        vehicle.addGear();

        gear = new AutoGear();
        vehicle = new Car(gear);
        vehicle.addGear();

        gear = new ManualGear();
        vehicle = new Truck(gear);
        vehicle.addGear();
```

```
        gear = new AutoGear();
        vehicle = new Truck(gear);
        vehicle.addGear();
    }
}
```

sortie:

```
Car handles Manual gear
Car handles Auto gear
Truck handles Manual gear
Truck handles Auto gear
```

Explication:

1. `Vehicle` est une abstraction.
2. `Car` and `Truck` sont deux applications concrètes du `Vehicle` .
3. `Vehicle` définit une méthode abstraite: `addGear()` .
4. `Gear` est une interface de mise en œuvre
5. `ManualGear` et `AutoGear` sont deux implémentations de `Gear`
6. `Vehicle` contient une interface de mise en `implementor` plutôt que d'implémenter l'interface. `Compositon` de l'interface de l'implémenteur est le noeud de ce modèle: *il permet à l'abstraction et à l'implémentation de varier indépendamment.*
7. `Car` and `Truck` definit l'implémentation (abstraction redéfinie) pour abstraction: `addGear()` : Contient `Gear` - `Manual` ou `Auto`

Cas d'utilisation du pattern Bridge :

1. **L'abstraction** et l' **implémentation** peuvent changer d'indépendance les unes des autres et elles ne sont pas liées au moment de la compilation
2. Mappez les hiérarchies orthogonales: une pour l' *abstraction* et une pour l' *implémentation* .

Lire Modèle de pont en ligne: https://riptutorial.com/fr/design-patterns/topic/4011/modele-de-pont

Chapitre 15: Modèle de stratégie

Examples

Cacher les détails de la mise en œuvre de la stratégie

Une ligne directrice très courante dans la conception orientée objet est "aussi peu que possible mais autant que nécessaire". Cela s'applique également au modèle de stratégie: il est généralement conseillé de masquer les détails d'implémentation, par exemple les classes qui implémentent réellement des stratégies.

Pour les stratégies simples qui ne dépendent pas de paramètres externes, l'approche la plus courante consiste à rendre la classe d'implémentation elle-même privée (classes imbriquées) ou package-private et à exposer une instance via un champ statique d'une classe publique:

```
public class Animal {

  private static class AgeComparator implements Comparator<Animal> {
    public int compare(Animal a, Animal b) {
      return a.age() - b.age();
    }
  }

  // Note that this field has the generic type Comparator<Animal>, *not*
  // Animal.AgeComparator!
  public static final Comparator<Animal> AGE_COMPARATOR = new AgeComparator();

  private final int age;

  Animal(int age) {
    this.age = age;
  }

  public int age() {
    return age;
  }

}

List<Animal> myList = new LinkedList<>();
myList.add(new Animal(10));
myList.add(new Animal(5));
myList.add(new Animal(7));
myList.add(new Animal(9));

Collections.sort(
  myList,
  Animal.AGE_COMPARATOR
);
```

Le champ public `Animal.AGE_COMPARATOR` définit une stratégie qui peut ensuite être utilisée dans des méthodes telles que `Collections.sort` , mais cela ne nécessite pas que l'utilisateur sache rien de son implémentation, pas même de la classe d'implémentation.

Si vous préférez, vous pouvez utiliser un cours anonyme:

```
public class Animal {

  public static final Comparator<Animal> AGE_COMPARATOR = new Comparator<Animal> {
    public int compare(Animal a, Animal b) {
      return a.age() - b.age();
    }
  };

  // other members...
}
```

Si la stratégie est un peu plus complexe et nécessite des paramètres, il est très courant d'utiliser des méthodes d'usine statiques telles que `Collections.reverseOrder(Comparator<T>)` . Le type de retour de la méthode ne doit pas exposer de détails d'implémentation, par exemple `reverseOrder()` est implémenté comme

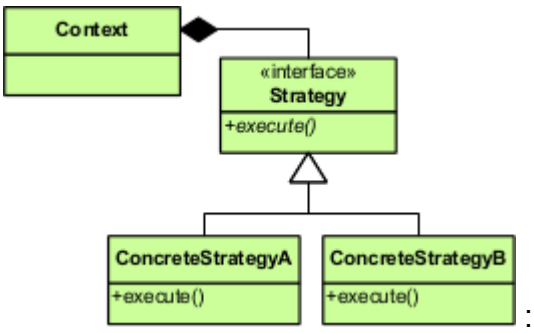
```
public static <T> Comparator<T> reverseOrder(Comparator<T> cmp) {
  // (Irrelevant lines left out.)
  return new ReverseComparator2<>(cmp);
}
```

Exemple de modèle de stratégie en Java avec classe de contexte

Stratégie:

`Strategy` est un modèle de comportement qui permet de modifier l'algorithme de façon dynamique à partir d'une famille d'algorithmes connexes.

UML du modèle de stratégie de Wikipedia



:

```
import java.util.*;

/* Interface for Strategy */
interface OfferStrategy {
    public String getName();
    public double getDiscountPercentage();
}
/* Concrete implementation of base Strategy */
class NoDiscountStrategy implements OfferStrategy{
    public String getName(){
        return this.getClass().getName();
    }
```

```
    public double getDiscountPercentage(){
        return 0;
    }
}
/* Concrete implementation of base Strategy */
class QuarterDiscountStrategy implements OfferStrategy{
    public String getName(){
        return this.getClass().getName();
    }
    public double getDiscountPercentage(){
        return 0.25;
    }
}
/* Context is optional. But if it is present, it acts as single point of contact
   for client.

   Multiple uses of Context
   1. It can populate data to execute an operation of strategy
   2. It can take independent decision on Strategy creation.
   3. In absence of Context, client should be aware of concrete strategies. Context acts a
wrapper and hides internals
   4. Code re-factoring will become easy
*/
class StrategyContext {
    double price; // price for some item or air ticket etc.
    Map<String,OfferStrategy> strategyContext = new HashMap<String,OfferStrategy>();
    StrategyContext(double price){
        this.price= price;
        strategyContext.put(NoDiscountStrategy.class.getName(),new NoDiscountStrategy());
        strategyContext.put(QuarterDiscountStrategy.class.getName(),new
QuarterDiscountStrategy());
    }
    public void applyStrategy(OfferStrategy strategy){
        /*
        Currently applyStrategy has simple implementation. You can Context for populating some
more information,
        which is required to call a particular operation
        */
        System.out.println("Price before offer :"+price);
        double finalPrice = price - (price*strategy.getDiscountPercentage());
        System.out.println("Price after offer:"+finalPrice);
    }
    public OfferStrategy getStrategy(int monthNo){
        /*
            In absence of this Context method, client has to import relevant concrete
Strategies everywhere.
            Context acts as single point of contact for the Client to get relevant Strategy
        */
        if ( monthNo < 6 )  {
            return strategyContext.get(NoDiscountStrategy.class.getName());
        }else{
            return strategyContext.get(QuarterDiscountStrategy.class.getName());
        }

    }
}
public class StrategyDemo{
    public static void main(String args[]){
        StrategyContext context = new StrategyContext(100);
        System.out.println("Enter month number between 1 and 12");
        int month = Integer.parseInt(args[0]);
```

```
        System.out.println("Month ="+month);
        OfferStrategy strategy = context.getStrategy(month);
        context.applyStrategy(strategy);
    }

}
```

sortie:

```
Enter month number between 1 and 12
Month =1
Price before offer :100.0
Price after offer:100.0

Enter month number between 1 and 12
Month =7
Price before offer :100.0
Price after offer:75.0
```

Énoncé du problème: Offre 25% de réduction sur le prix de l'article pour les mois de juillet-décembre. Ne fournissez aucune réduction pour les mois de janvier à juin.

L'exemple ci-dessus montre l'utilisation du modèle de `Strategy` avec le `Context` . `Context` peut être utilisé comme point de contact unique pour le `Client` .

Deux stratégies - `NoOfferStrategy` et `QuarterDiscountStrategy` ont été déclarées conformément à l'énoncé du problème.

Comme indiqué dans la colonne de sortie, vous obtiendrez une remise en fonction du mois que vous avez entré

Cas d'utilisation du modèle de stratégie :

1. Utilisez ce modèle lorsque vous disposez d'une famille d'algorithmes interchangeables et que vous devez modifier l'algorithme au moment de l'exécution.

2. Gardez le code plus propre en supprimant les instructions conditionnelles

Modèle de stratégie sans classe de contexte / Java

Voici un exemple simple d'utilisation du modèle de stratégie sans classe de contexte. Deux stratégies d'implémentation implémentent l'interface et résolvent le même problème de différentes manières. Les utilisateurs de la classe EnglishTranslation peuvent appeler la méthode de traduction et choisir la stratégie à utiliser pour la traduction, en spécifiant la stratégie souhaitée.

```
// The strategy interface
public interface TranslationStrategy {
    String translate(String phrase);
}

// American strategy implementation
public class AmericanTranslationStrategy implements TranslationStrategy {
```

```
    @Override
    public String translate(String phrase) {
        return phrase + ", bro";
    }
}

// Australian strategy implementation
public class AustralianTranslationStrategy implements TranslationStrategy {

    @Override
    public String translate(String phrase) {
        return phrase + ", mate";
    }
}

// The main class which exposes a translate method
public class EnglishTranslation {

    //  translate a phrase using a given strategy
    public static String translate(String phrase, TranslationStrategy strategy) {
        return strategy.translate(phrase);
    }

    // example usage
    public static void main(String[] args) {

        // translate a phrase using the AustralianTranslationStrategy class
        String aussieHello = translate("Hello", new AustralianTranslationStrategy());
        // Hello, mate

        // translate a phrase using the AmericanTranslationStrategy class
        String usaHello = translate("Hello", new AmericanTranslationStrategy());
        // Hello, bro
    }
}
```

Utilisation des interfaces fonctionnelles Java 8 pour implémenter le modèle de stratégie

Le but de cet exemple est de montrer comment nous pouvons réaliser le modèle de stratégie en utilisant les interfaces fonctionnelles Java 8. Nous allons commencer par un simple cas d'utilisation des codes en Java classique, puis le recoder sous la forme de Java 8.

L'exemple que nous utilisons est une famille d'algorithmes (stratégies) qui *décrivent* différentes manières de communiquer à distance.

La version Java classique

Le contrat pour notre famille d'algorithmes est défini par l'interface suivante:

```
public interface CommunicateInterface {
    public String communicate(String destination);
}
```

Ensuite, nous pouvons implémenter un certain nombre d'algorithmes, comme suit:

```
public class CommunicateViaPhone implements CommunicateInterface {
    @Override
    public String communicate(String destination) {
        return "communicating " + destination +" via Phone..";
    }
}

public class CommunicateViaEmail implements CommunicateInterface {
    @Override
    public String communicate(String destination) {
        return "communicating " + destination + " via Email..";
    }
}

public class CommunicateViaVideo implements CommunicateInterface {
    @Override
    public String communicate(String destination) {
        return "communicating " + destination + " via Video..";
    }
}
```

Ceux-ci peuvent être instanciés comme suit:

```
CommunicateViaPhone communicateViaPhone = new CommunicateViaPhone();
CommunicateViaEmail communicateViaEmail = new CommunicateViaEmail();
CommunicateViaVideo communicateViaVideo = new CommunicateViaVideo();
```

Ensuite, nous implémentons un service utilisant la stratégie:

```
public class CommunicationService {
    private CommunicateInterface communcationMeans;

    public void setCommuncationMeans(CommunicateInterface communcationMeans) {
        this.communcationMeans = communcationMeans;
    }

    public void communicate(String destination) {
        this.communcationMeans.communicate(destination);
    }
}
```

Finalement, nous pouvons utiliser les différentes stratégies comme suit:

```
CommunicationService communicationService = new CommunicationService();

// via phone
communicationService.setCommuncationMeans(communicateViaPhone);
communicationService.communicate("1234567");

// via email
communicationService.setCommuncationMeans(communicateViaEmail);
communicationService.communicate("hi@me.com");
```

Utilisation des interfaces fonctionnelles Java 8

Le contrat des différentes implémentations d'algorithmes ne nécessite pas d'interface dédiée. Au lieu de cela, nous pouvons le décrire en utilisant l'interface existante `java.util.function.Function<T, R>` .

Les différents algorithmes composant `the family of algorithms` peuvent être exprimés sous forme d'expressions lambda. Cela remplace les classes de stratégie *et* leurs instanciations.

```
Function<String, String> communicateViaEmail =
        destination -> "communicating " + destination + " via Email..";
Function<String, String> communicateViaPhone =
        destination -> "communicating " + destination + " via Phone..";
Function<String, String> communicateViaVideo =
        destination -> "communicating " + destination + " via Video..";
```

Ensuite, nous pouvons coder le "service" comme suit:

```
public class CommunicationService {
    private Function<String, String> communcationMeans;

    public void setCommuncationMeans(Function<String, String> communcationMeans) {
        this.communcationMeans = communcationMeans;
    }

    public void communicate(String destination) {
        this.communcationMeans.communicate(destination);
    }
}
```

Enfin, nous utilisons les stratégies comme suit

```
CommunicationService communicationService = new CommunicationService();

// via phone
communicationService.setCommuncationMeans(communicateViaPhone);
communicationService.communicate("1234567");

// via email
communicationService.setCommuncationMeans(communicateViaEmail);
communicationService.communicate("hi@me.com");
```

Ou même:

```
communicationService.setCommuncationMeans(
    destination -> "communicating " + destination + " via Smoke signals.." );
CommunicationService.communicate("anyone");
```

Stratégie (PHP)

Exemple de www.phptherightway.com

```
<?php

interface OutputInterface
```

```
{
    public function load();
}

class SerializedArrayOutput implements OutputInterface
{
    public function load()
    {
        return serialize($arrayOfData);
    }
}

class JsonStringOutput implements OutputInterface
{
    public function load()
    {
        return json_encode($arrayOfData);
    }
}

class ArrayOutput implements OutputInterface
{
    public function load()
    {
        return $arrayOfData;
    }
}
```

Lire Modèle de stratégie en ligne: https://riptutorial.com/fr/design-patterns/topic/1331/modele-de-strategie

Chapitre 16: Monostate

Remarques

En parallèle, quelques avantages du pattern `Monostate` sur le `Singleton` :

- Il n'y a pas de méthode 'instance` pour pouvoir accéder à une instance de la classe.
- Un `Singleton` n'est pas conforme à la notation Java Beans, mais un `Monostate` fait.
- La durée de vie des instances peut être contrôlée.
- Les utilisateurs du `Monostate` ne savent pas qu'ils utilisent un `Monostate` .
- Le polymorphisme est possible.

Examples

Le modèle de monostate

Le modèle `Monostate` est généralement appelé *sucre syntaxique* sur le modèle `Singleton` ou *`Singleton` conceptuel* .

Il évite toutes les complications liées à une seule instance d'une classe, mais toutes les instances utilisent les mêmes données.
Ceci est accompli principalement en utilisant `static` membres de données `static` .
L'une des caractéristiques les plus importantes est qu'il est absolument transparent pour les utilisateurs qui ignorent complètement qu'ils travaillent avec un `Monostate` . Les utilisateurs peuvent créer autant d'instances d'un `Monostate` qu'ils le souhaitent et chaque instance peut être considérée comme une autre pour accéder aux données.

La classe `Monostate` est généralement fournie avec une classe compagnon utilisée pour mettre à jour les paramètres si nécessaire.

Il suit un exemple minimal de `Monostate` en C ++:

```
struct Settings {
    Settings() {
        if(!initialized) {
            initialized = true;
            // load from file or db or whatever
            // otherwise, use the SettingsEditor to initialize settings
            Settings::width_ = 42;
            Settings::height_ = 128;
        }
    }

    std::size_t width() const noexcept { return width_; }
    std::size_t height() const noexcept { return height_; }

private:
    friend class SettingsEditor;
```

```
    static bool initialized;
    static std::size_t width_;
    static std::size_t height_;
};

bool Settings::initialized = false;
std::size_t Settings::width_;
std::size_t Settings::height_;

struct SettingsEditor {
    void width(std::size_t value) noexcept { Settings::width_ = value; }
    void height(std::size_t value) noexcept { Settings::height_ = value; }
};
```

Voici un exemple d'implémentation simple d'un `Monostate` en Java:

```
public class Monostate {
    private static int width;
    private static int height;

    public int getWidth() {
        return Monostate.width;
    }

    public int getHeight() {
        return Monostate.height;
    }

    public void setWidth(int value) {
        Monostate.width = value;
    }

    public void setHeight(int value) {
        Monostate.height = value;
    }

    static {
        width = 42;
        height = 128;
    }
}
```

Hiérarchies basées sur Monostate

Contrairement au `Singleton` , le `Monostate` peut être hérité pour étendre ses fonctionnalités, tant que les méthodes membres ne sont pas `static` .
Il suit un exemple minimal en C ++:

```
struct Settings {
    virtual std::size_t width() const noexcept { return width_; }
    virtual std::size_t height() const noexcept { return height_; }

private:
    static std::size_t width_;
    static std::size_t height_;
};
```

```
std::size_t Settings::width_{0};
std::size_t Settings::height_{0};

struct EnlargedSettings: Settings {
    std::size_t width() const noexcept override { return Settings::height() + 1; }
    std::size_t height() const noexcept override { return Settings::width()  + 1; }
};
```

Lire Monostate en ligne: https://riptutorial.com/fr/design-patterns/topic/6186/monostate

Chapitre 17: Motif composite

Examples

Enregistreur composite

Le motif composite est un motif de conception permettant de traiter un groupe d'objets en tant qu'instance unique d'un objet. C'est l'un des modèles de conception structurelle du Gang of Four.

L'exemple ci-dessous montre comment Composite peut être utilisé pour se connecter à plusieurs endroits en utilisant un appel de journal unique. Cette approche respecte les principes SOLID car elle vous permet d'ajouter un nouveau mécanisme de journalisation sans violer le principe de responsabilité unique (chaque enregistreur n'a qu'une seule responsabilité) ou principe Open / closed (vous pouvez ajouter un nouveau logger au nouvel emplacement en ajoutant une nouvelle implémentation). pas en modifiant ceux existants).

```
public interface ILogger
{
    void Log(string message);
}

public class CompositeLogger : ILogger
{
    private readonly ILogger[] _loggers;

    public CompositeLogger(params ILogger[] loggers)
    {
        _loggers = loggers;
    }

    public void Log(string message)
    {
        foreach (var logger in _loggers)
        {
            logger.Log(message);
        }
    }
}

public class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        //log to console
    }
}

public class FileLogger : ILogger
{
    public void Log(string message)
    {
        //log to file
    }
}
```

```
var compositeLogger = new CompositeLogger(new ConsoleLogger(), new FileLogger());
compositeLogger.Log("some message"); //this will be invoked both on ConsoleLogger and
FileLogger
```

Il convient de mentionner que les enregistreurs composites peuvent être imbriqués (l'un des paramètres du constructeur des enregistreurs composites peut être un autre enregistreur composite) créant une structure arborescente.

Lire Motif composite en ligne: https://riptutorial.com/fr/design-patterns/topic/4515/motif-composite

Chapitre 18: Motif composite

Introduction

Composite permet aux clients de traiter des objets individuels et des compositions d'objets de manière uniforme. Par exemple, considérez un programme qui manipule un système de fichiers. Les fichiers sont des objets simples et les dossiers sont la composition de fichiers et de dossiers. Cependant, par exemple, ils ont tous deux des fonctions de taille, de nom, etc. Il serait plus facile et plus pratique de traiter les objets de fichier et de dossier de manière uniforme en définissant une interface de ressource de système de fichiers.

Remarques

Le modèle composite s'applique lorsqu'il existe une hiérarchie partielle d'objets et qu'un client doit traiter les objets de manière uniforme, même si un objet peut être une feuille (objet simple) ou une branche (objet composite).

Examples

pseudocode pour un gestionnaire de fichiers stupide

```
/*
* Component is an interface
* which all elements (files,
* folders, links ...) will implement
*/
class Component
{
public:
    virtual int getSize() const = 0;
};

/*
* File class represents a file
* in file system.
*/
class File : public Component
{
public:
    virtual int getSize() const {
        // return file size
    }
};

/*
* Folder is a component and
* also may contain files and
* another folders. Folder is a
* composition of components
*/
class Folder : public Component
```

```
{
public:
    void addComponent(Component* aComponent) {
        // mList append aComponent;
    }
    void removeComponent(Component* aComponent) {
        // remove aComponent from mList
    }
    virtual int getSize() const {
        int size = 0;
        foreach(component : mList) {
            size += component->getSize();
        }
        return size;
    }

private:
    list<Component*> mList;
};
```

Lire Motif composite en ligne: https://riptutorial.com/fr/design-patterns/topic/9197/motif-composite

Chapitre 19: Motif de constructeur

Remarques

Séparer la construction d'un objet complexe de sa représentation afin qu'un même processus de construction puisse créer différentes représentations.

- Séparer la logique de la représentation.
- Réutiliser la logique pour travailler avec différents ensembles de données.

Examples

Modèle de générateur / C # / Interface fluide

```
public class Email
{
    public string To { get; set; }
    public string From { get; set; }
    public string Subject { get; set; }
    public string Body { get; set; }
}

public class EmailBuilder
{
    private readonly Email _email;

    public EmailBuilder()
    {
        _email = new Email();
    }

    public EmailBuilder To(string address)
    {
        _email.To = address;
        return this;
    }

    public EmailBuilder From(string address)
    {
        _email.From = address;
        return this;
    }

    public EmailBuilder Subject(string title)
    {
        _email.Subject = title;
        return this;
    }

    public EmailBuilder Body(string content)
    {
        _email.Body = content;
        return this;
    }
```

```
    public Email Build()
    {
        return _email;
    }
}
```

Exemple d'utilisation:

```
var emailBuilder = new EmailBuilder();
var email = emailBuilder
    .To("email1@email.com")
    .From("email2@email.com")
    .Subject("Email subject")
    .Body("Email content")
    .Build();
```

Modèle de générateur / Implémentation Java

Le modèle Builder vous permet de créer une instance d'une classe avec de nombreuses variables facultatives d'une manière facile à lire.

Considérez le code suivant:

```
public class Computer {

    public GraphicsCard graphicsCard;
    public Monitor[] monitors;
    public Processor processor;
    public Memory[] ram;
    //more class variables here...

    Computer(GraphicsCard g, Monitor[] m, Processer p, Memory ram) {
        //code omitted for brevity...
    }

    //class methods omitted...

}
```

Tout va bien si tous les paramètres sont nécessaires. Et s'il y a beaucoup plus de variables et / ou certaines d'entre elles sont facultatives? Vous ne voulez pas créer un grand nombre de constructeurs avec chaque combinaison possible de paramètres obligatoires et facultatifs, car cela devient difficile à gérer et à comprendre pour les développeurs. Il se peut également que vous ne souhaitiez pas avoir une longue liste de paramètres dans lesquels l'utilisateur peut avoir à saisir beaucoup de paramètres nuls.

Le modèle de générateur crée une classe interne appelée Builder qui permet d'instancier uniquement les variables facultatives souhaitées. Cela se fait par des méthodes pour chaque variable facultative qui prend le type de variable comme paramètre et renvoie un objet Builder afin que les méthodes puissent être chaînées les unes avec les autres. Toutes les variables requises sont placées dans le constructeur Builder afin qu'elles ne puissent pas être omises.

Le générateur inclut également une méthode appelée `build()` qui renvoie l'objet dans lequel elle se trouve et doit être appelée à la fin de la chaîne d'appels de méthode lors de la construction de l'objet.

À la suite de l'exemple précédent, ce code utilise le modèle Builder pour la classe Computer.

```
public class Computer {

    private GraphicsCard graphicsCard;
    private Monitor[] monitors;
    private Processor processor;
    private Memory[] ram;
    //more class variables here...

    private Computer(Builder builder) {
        this.graphicsCard = builder.graphicsCard;
        this.monitors = builder.monitors;
        this.processor = builder.processor;
        this.ram = builder.ram;
    }

    public GraphicsCard getGraphicsCard() {
        return this.graphicsCard;
    }

    public Monitor[] getMonitors() {
        return this.monitors;
    }

    public Processor getProcessor() {
        return this.processor;
    }

    public Memory[] getRam() {
        return this.ram;
    }

    public static class Builder {
        private GraphicsCard graphicsCard;
        private Monitor[] monitors;
        private Processor processor;
        private Memory[] ram;

        public Builder(Processor p){
            this.processor = p;
        }

        public Builder graphicsCard(GraphicsCard g) {
            this.graphicsCard = g;
            return this;
        }

        public Builder monitors(Monitor[] mg) {
            this.monitors = mg;
            return this;
        }

        public Builder ram(Memory[] ram) {
            this.ram = ram;
            return this;
```

```
        }

        public Computer build() {
            return new Computer(this);
        }
    }
}
```

Un exemple d'utilisation de cette classe:

```
public class ComputerExample {

    public static void main(String[] args) {
        Computer headlessComputer = new Computer.Builder(new Processor("Intel-i3"))
                .graphicsCard(new GraphicsCard("GTX-960"))
                .build();

        Computer gamingPC = new Computer.Builder(new Processor("Intel-i7-quadcode"))
                .graphicsCard(new GraphicsCard("DX11"))
                .monitors(new Monitor[] = {new Monitor("acer-s7"), new Monitor("acer-s7")})
                .ram(new Memory[] = {new Memory("2GB"), new Memory("2GB"), new Memory("2GB"),
new Memory("2GB")})
                .build();
    }

}
```

Cet exemple montre comment le modèle de générateur peut permettre une grande flexibilité dans la création d'une classe avec un effort assez réduit. L'objet Ordinateur peut être implémenté en fonction de la configuration souhaitée par les appelants dans un format facile à lire et sans effort.

Modèle de constructeur en Java avec composition

Intention:

Séparer la construction d'un objet complexe de sa représentation afin qu'un même processus de construction puisse créer différentes représentations

Le modèle de générateur est utile lorsque vous avez peu d'attributs obligatoires et de nombreux attributs facultatifs pour construire un objet. Pour créer un objet avec différents attributs obligatoires et facultatifs, vous devez fournir un constructeur complexe pour créer l'objet. Le modèle de générateur fournit un processus pas à pas simple pour construire un objet complexe.

Cas d'utilisation réel:

Différents utilisateurs de FaceBook ont des attributs différents, qui consistent en des attributs obligatoires tels que le nom d'utilisateur et des attributs facultatifs tels que UserBasicInfo et ContactInfo. Certains utilisateurs fournissent simplement des informations de base. Certains utilisateurs fournissent des informations détaillées, y compris les informations de contact. En l'absence de modèle Builder, vous devez fournir un constructeur avec tous les paramètres obligatoires et facultatifs. Mais le modèle Builder simplifie le processus de construction en fournissant un processus simple, étape par étape, pour construire l'objet complexe.

Conseils:

1. Fournit une classe de générateur imbriquée statique.
2. Fournit un constructeur pour les attributs obligatoires de l'objet.
3. Fournit des méthodes de réglage et de lecture pour les attributs optionnels de l'objet.
4. Renvoie le même objet Builder après la définition des attributs facultatifs.
5. Fournit la méthode build (), qui renvoie un objet complexe

Extrait de code:

```
import java.util.*;

class UserBasicInfo{
    String nickName;
    String birthDate;
    String gender;

    public UserBasicInfo(String name,String date,String gender){
        this.nickName = name;
        this.birthDate = date;
        this.gender = gender;
    }

    public String toString(){
        StringBuilder sb = new StringBuilder();

sb.append("Name:DOB:Gender:").append(nickName).append(":").append(birthDate).append(":").
        append(gender);
        return sb.toString();
    }
}

class ContactInfo{
    String eMail;
    String mobileHome;
    String mobileWork;

    public ContactInfo(String mail, String homeNo, String mobileOff){
        this.eMail = mail;
        this.mobileHome = homeNo;
        this.mobileWork = mobileOff;
    }
    public String toString(){
        StringBuilder sb = new StringBuilder();

sb.append("email:mobile(H):mobile(W):").append(eMail).append(":").append(mobileHome).append(":").appen

        return sb.toString();
    }
}
class FaceBookUser {
    String userName;
    UserBasicInfo userInfo;
    ContactInfo contactInfo;

    public FaceBookUser(String uName){
        this.userName = uName;
    }
    public void setUserBasicInfo(UserBasicInfo info){
```

```
        this.userInfo = info;
    }
    public void setContactInfo(ContactInfo info){
        this.contactInfo = info;
    }
    public String getUserName(){
        return userName;
    }
    public UserBasicInfo getUserBasicInfo(){
        return userInfo;
    }
    public ContactInfo getContactInfo(){
        return contactInfo;
    }

    public String toString(){
        StringBuilder sb = new StringBuilder();

sb.append("|User|").append(userName).append("|UserInfo|").append(userInfo).append("|ContactInfo|").appe

        return sb.toString();
    }

    static class FaceBookUserBuilder{
        FaceBookUser user;
        public FaceBookUserBuilder(String userName){
            this.user = new FaceBookUser(userName);
        }
        public FaceBookUserBuilder setUserBasicInfo(UserBasicInfo info){
            user.setUserBasicInfo(info);
            return this;
        }
        public FaceBookUserBuilder setContactInfo(ContactInfo info){
            user.setContactInfo(info);
            return this;
        }
        public FaceBookUser build(){
            return user;
        }
    }
}
public class BuilderPattern{
    public static void main(String args[]){
        FaceBookUser fbUser1 = new FaceBookUser.FaceBookUserBuilder("Ravindra").build(); //
Mandatory parameters
        UserBasicInfo info = new UserBasicInfo("sunrise","25-May-1975","M");

        // Build User name + Optional Basic Info
        FaceBookUser fbUser2 = new FaceBookUser.FaceBookUserBuilder("Ravindra").
                                            setUserBasicInfo(info).build();

        // Build User name + Optional Basic Info + Optional Contact Info
        ContactInfo cInfo = new ContactInfo("xxx@xyz.com","1111111111","2222222222");
        FaceBookUser fbUser3 = new FaceBookUser.FaceBookUserBuilder("Ravindra").
                                            setUserBasicInfo(info).
                                            setContactInfo(cInfo).build();

        System.out.println("Facebook user 1:"+fbUser1);
        System.out.println("Facebook user 2:"+fbUser2);
        System.out.println("Facebook user 3:"+fbUser3);
    }
```

```
}
```

sortie:

```
Facebook user 1:|User|Ravindra|UserInfo|null|ContactInfo|null
Facebook user 2:|User|Ravindra|UserInfo|Name:DOB:Gender:sunrise:25-May-1975:M|ContactInfo|null
Facebook user 3:|User|Ravindra|UserInfo|Name:DOB:Gender:sunrise:25-May-
1975:M|ContactInfo|email:mobile(H):mobile(W):xxx@xyz.com:1111111111:2222222222
```

Explication:

1. `FaceBookUser` est un objet complexe dont les attributs ci-dessous utilisent la composition:

   ```
   String userName;
   UserBasicInfo userInfo;
   ContactInfo contactInfo;
   ```

2. `FaceBookUserBuilder` est une classe de construction statique qui contient et construit `FaceBookUser` .

3. `userName` est uniquement un paramètre obligatoire pour créer `FaceBookUser`

4. `FaceBookUserBuilder` construit `FaceBookUser` en définissant des paramètres facultatifs: `UserBasicInfo` et `ContactInfo`

5. Cet exemple illustre trois FaceBookUsers différents avec des attributs différents, construits à partir de Builder.

   1. fbUser1 a été construit comme FaceBookUser avec l'attribut userName uniquement
   2. fbUser2 a été construit comme FaceBookUser avec userName et UserBasicInfo
   3. fbUser3 a été créé avec FaceBookUser avec userName, UserBasicInfo et ContactInfo

Dans l'exemple ci-dessus, la composition a été utilisée au lieu de dupliquer tous les attributs de FaceBookUser dans la classe Builder.

Dans les modèles de création, nous commencerons par un modèle simple, comme `FactoryMethod` et `FactoryMethod` vers des modèles plus flexibles et complexes, tels que `AbstractFactory` et `Builder` .

Java / Lombok

```
import lombok.Builder;

@Builder
public class Email {

    private String to;
    private String from;
    private String subject;
    private String body;

}
```

Exemple d'utilisation:

```
Email.builder().to("email1@email.com")
        .from("email2@email.com")
        .subject("Email subject")
        .body("Email content")
        .build();
```

Modèle de générateur avancé avec une expression Java 8 Lambda

```
public class Person {
private final String salutation;
private final String firstName;
private final String middleName;
private final String lastName;
private final String suffix;
private final Address address;
private final boolean isFemale;
private final boolean isEmployed;
private final boolean isHomewOwner;

public Person(String salutation, String firstName, String middleName, String lastName, String
suffix, Address address, boolean isFemale, boolean isEmployed, boolean isHomewOwner) {
    this.salutation = salutation;
    this.firstName = firstName;
    this.middleName = middleName;
    this.lastName = lastName;
    this.suffix = suffix;
    this.address = address;
    this.isFemale = isFemale;
    this.isEmployed = isEmployed;
    this.isHomewOwner = isHomewOwner;
 }
}
```

Ancienne voie

```
public class PersonBuilder {
private String salutation;
private String firstName;
private String middleName;
private String lastName;
private String suffix;
private Address address;
private boolean isFemale;
private boolean isEmployed;
private boolean isHomewOwner;

public PersonBuilder withSalutation(String salutation) {
    this.salutation = salutation;
    return this;
}

public PersonBuilder withFirstName(String firstName) {
    this.firstName = firstName;
    return this;
}
```

```
public PersonBuilder withMiddleName(String middleName) {
    this.middleName = middleName;
    return this;
}

public PersonBuilder withLastName(String lastName) {
    this.lastName = lastName;
    return this;
}

public PersonBuilder withSuffix(String suffix) {
    this.suffix = suffix;
    return this;
}

public PersonBuilder withAddress(Address address) {
    this.address = address;
    return this;
}

public PersonBuilder withIsFemale(boolean isFemale) {
    this.isFemale = isFemale;
    return this;
}

public PersonBuilder withIsEmployed(boolean isEmployed) {
    this.isEmployed = isEmployed;
    return this;
}

public PersonBuilder withIsHomewOwner(boolean isHomewOwner) {
    this.isHomewOwner = isHomewOwner;
    return this;
}

public Person createPerson() {
    return new Person(salutation, firstName, middleName, lastName, suffix, address, isFemale,
isEmployed, isHomewOwner);
}
```

Manière avancée:

```
public class PersonBuilder {
public String salutation;
public String firstName;
public String middleName;
public String lastName;
public String suffix;
public Address address;
public boolean isFemale;
public boolean isEmployed;
public boolean isHomewOwner;

public PersonBuilder with(
        Consumer<PersonBuilder> builderFunction) {
    builderFunction.accept(this);
    return this;
}
```

```
public Person createPerson() {
    return new Person(salutation, firstName, middleName,
            lastName, suffix, address, isFemale,
            isEmployed, isHomewOwner);
}
```

}

Usage:

```
Person person = new PersonBuilder()
    .with($ -> {
        $.salutation = "Mr.";
        $.firstName = "John";
        $.lastName = "Doe";
        $.isFemale = false;
        $.isHomewOwner = true;
        $.address =
            new PersonBuilder.AddressBuilder()
                .with($_address -> {
                    $_address.city = "Pune";
                    $_address.state = "MH";
                    $_address.pin = "411001";
                }).createAddress();
    })
    .createPerson();
```

Se référer: https://medium.com/beingprofessional/think-functional-advanced-builder-pattern-using-lambda-284714b85ed5#.d9sryx3g9

Lire Motif de constructeur en ligne: https://riptutorial.com/fr/design-patterns/topic/1811/motif-de-constructeur

Chapitre 20: Motif de décorateur

Introduction

Le motif de décorateur permet à un utilisateur d'ajouter de nouvelles fonctionnalités à un objet existant sans modifier sa structure. Ce type de motif de conception est inclus dans le motif structurel, car ce motif agit comme une enveloppe pour la classe existante.

Ce modèle crée une classe de décorateur qui enveloppe la classe d'origine et fournit des fonctionnalités supplémentaires en gardant intacte la signature des méthodes de classe.

Paramètres

Paramètre	La description
Boisson	ce peut être du thé ou du café

Examples

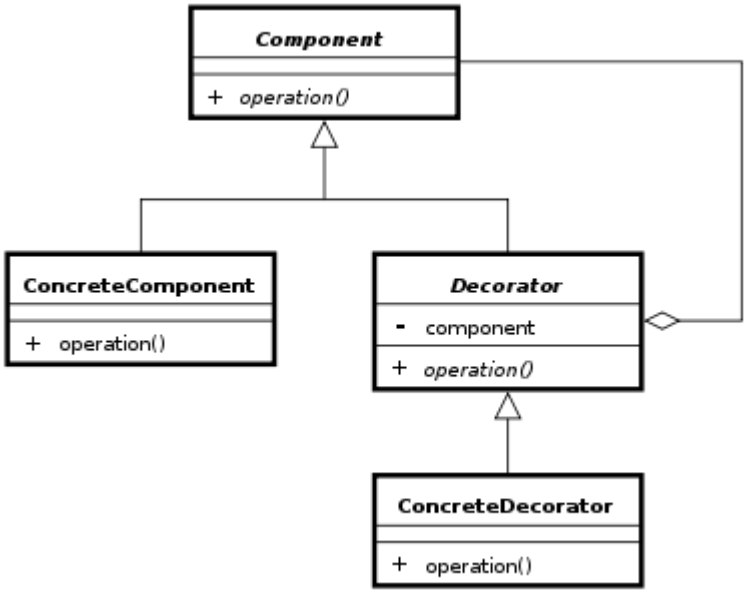
VendingMachineDecorator

Définition du décorateur selon Wikiepdia:

Le motif Decorator peut être utilisé pour étendre (décorer) de manière statique ou, dans certains cas, au moment de l'exécution, les fonctionnalités d'un objet donné, indépendamment des autres instances de la même classe, à condition que le travail de conception soit effectué.

Le décorateur attribue des responsabilités supplémentaires à un objet de manière dynamique. Les décorateurs offrent une alternative flexible au sous-classement pour étendre les fonctionnalités.

Le motif de décorateur contient quatre composants.



1. Interface de composant: définit une interface pour exécuter des opérations particulières
2. ConcreteComponent: implémente les opérations définies dans l'interface de composant
3. Decorator (Abstract): c'est une classe abstraite qui étend l'interface du composant. Il contient une interface de composant. En l'absence de cette classe, vous avez besoin de plusieurs sous-classes de ConcreteDecorator pour différentes combinaisons. La composition du composant réduit les sous-classes non nécessaires.
4. ConcreteDecorator: Il contient l'implémentation de Abstract Decorator.

Revenons à l'exemple de code,

1. *La boisson* est un composant. Il définit une méthode abstraite: decorateBeverage
2. *Le thé* et le *café* sont des implémentations concrètes de *boissons* .
3. *BeverageDecorator* est une classe abstraite contenant des *boissons*
4. SugarDecorator et LemonDecorator sont des décorateurs concrets de BeverageDecorator.

EDIT: Changé l'exemple pour refléter le scénario réel de calcul du prix des boissons en ajoutant un ou plusieurs arômes tels que le sucre, le citron, etc. (les arômes sont des décorateurs)

```
abstract class Beverage {
    protected String name;
    protected int price;
    public Beverage(){

    }
    public  Beverage(String name){
        this.name = name;
    }
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
    protected void setPrice(int price){
        this.price = price;
    }
```

```
    protected int getPrice(){
        return price;
    }
    protected abstract void decorateBeverage();

}
class Tea extends Beverage{
    public Tea(String name){
        super(name);
        setPrice(10);
    }
    public void decorateBeverage(){
        System.out.println("Cost of:"+ name +":"+ price);
        // You can add some more functionality
    }
}
class Coffee extends Beverage{
    public Coffee(String name){
        super(name);
        setPrice(15);
    }
    public void decorateBeverage(){
        System.out.println("Cost of:"+ name +":"+ price);
        // You can add some more functionality
    }
}
abstract class BeverageDecorator extends Beverage {
    protected Beverage beverage;
    public BeverageDecorator(Beverage beverage){
        this.beverage = beverage;
        setName(beverage.getName()+"+"+getDecoratedName());
        setPrice(beverage.getPrice()+getIncrementPrice());
    }
    public void decorateBeverage(){
        beverage.decorateBeverage();
        System.out.println("Cost of:"+getName()+":"+getPrice());
    }
    public abstract int getIncrementPrice();
    public abstract String getDecoratedName();
}
class SugarDecorator extends BeverageDecorator{
    public SugarDecorator(Beverage beverage){
        super(beverage);
    }
    public void decorateBeverage(){
        super.decorateBeverage();
        decorateSugar();
    }
    public void decorateSugar(){
        System.out.println("Added Sugar to:"+beverage.getName());
    }
    public int getIncrementPrice(){
        return 5;
    }
    public String getDecoratedName(){
        return "Sugar";
    }
}
class LemonDecorator extends BeverageDecorator{
    public LemonDecorator(Beverage beverage){
        super(beverage);
```

```
    }
    public void decorateBeverage(){
        super.decorateBeverage();
        decorateLemon();
    }
    public void decorateLemon(){
        System.out.println("Added Lemon to:"+beverage.getName());
    }
    public int getIncrementPrice(){
        return 3;
    }
    public String getDecoratedName(){
        return "Lemon";
    }
}

public class VendingMachineDecorator {
    public static void main(String args[]){
        Beverage beverage = new SugarDecorator(new LemonDecorator(new Tea("Assam Tea")));
        beverage.decorateBeverage();
        beverage = new SugarDecorator(new LemonDecorator(new Coffee("Cappuccino")));
        beverage.decorateBeverage();
    }
}
```

sortie:

```
Cost of:Assam Tea:10
Cost of:Assam Tea+Lemon:13
Added Lemon to:Assam Tea
Cost of:Assam Tea+Lemon+Sugar:18
Added Sugar to:Assam Tea+Lemon
Cost of:Cappuccino:15
Cost of:Cappuccino+Lemon:18
Added Lemon to:Cappuccino
Cost of:Cappuccino+Lemon+Sugar:23
Added Sugar to:Cappuccino+Lemon
```

Cet exemple calcule le coût de la boisson dans Vending Machine après avoir ajouté de nombreuses saveurs à la boisson.

Dans l'exemple ci-dessus:

Coût du thé = 10, citron = 3 et sucre = 5. Si vous faites du sucre + citron + thé, il en coûte 18.

Coût du café = 15, citron = 3 et sucre = 5. Si vous faites du sucre + citron + café, cela coûte 23

En utilisant le même décorateur pour les deux boissons (thé et café), le nombre de sous-classes a été réduit. En l'absence de motif de décorateur, vous devriez avoir différentes sous-classes pour différentes combinaisons.

Les combinaisons seront comme ceci:

```
SugarLemonTea
SugarTea
LemonTea
```

```
SugarLemonCapaccuino
SugarCapaccuino
LemonCapaccuino
```

etc.

En utilisant le même `Decorator` pour les deux boissons, le nombre de sous-classes a été réduit. C'est possible en raison de la `composition` plutôt que du concept d' `inheritance` utilisé dans ce modèle.

Comparaison avec d'autres modèles de conception (à partir d'article de fabrication)

1. *L'adaptateur* fournit une interface différente à son sujet. *Proxy* fournit la même interface. *Le décorateur* fournit une interface améliorée.

2. *L'adaptateur* modifie l'interface d'un objet, *Decorator* améliore les responsabilités d'un objet.

3. *Composite* et *Decorator* ont des diagrammes de structure similaires, reflétant le fait que les deux utilisent une composition récursive pour organiser un nombre illimité d'objets.

4. *Decorator* est conçu pour vous permettre d'ajouter des responsabilités à des objets sans sous-classement. *L'* accent *de Composite* n'est pas sur l'embellissement mais sur la représentation

5. *Le décorateur* et le *mandataire* ont des objectifs différents mais des structures similaires

6. *Le décorateur* vous permet de changer la peau d'un objet. *La stratégie* vous permet de changer les tripes.

Principaux cas d'utilisation:

1. Ajouter des fonctionnalités / responsabilités supplémentaires de manière dynamique
2. Supprimer les fonctionnalités / responsabilités de manière dynamique
3. Évitez de sous-classer trop pour ajouter des responsabilités supplémentaires.

Caching Decorator

Cet exemple montre comment ajouter des fonctionnalités de mise en cache à `DbProductRepository` aide du motif Decorator. Cette approche respecte les principes SOLID car elle vous permet d'ajouter de la mise en cache sans violer le principe de responsabilité unique ou le principe ouvert / fermé .

```
public interface IProductRepository
{
    Product GetProduct(int id);
}

public class DbProductRepository : IProductRepository
{
    public Product GetProduct(int id)
```

```
    {
        //return Product retrieved from DB
    }
}

public class ProductRepositoryCachingDecorator : IProductRepository
{
    private readonly IProductRepository _decoratedRepository;
    private readonly ICache _cache;
    private const int ExpirationInHours = 1;

    public ProductRepositoryCachingDecorator(IProductRepository decoratedRepository, ICache
cache)
    {
        _decoratedRepository = decoratedRepository;
        _cache = cache;
    }

    public Product GetProduct(int id)
    {
        var cacheKey = GetKey(id);
        var product = _cache.Get<Product>(cacheKey);
        if (product == null)
        {
            product = _decoratedRepository.GetProduct(id);
            _cache.Set(cacheKey, product, DateTimeOffset.Now.AddHours(ExpirationInHours));
        }

        return product;
    }

    private string GetKey(int id) => "Product:" + id.ToString();
}

public interface ICache
{
    T Get<T>(string key);
    void Set(string key, object value, DateTimeOffset expirationTime)
}
```

Usage:

```
var productRepository = new ProductRepositoryCachingDecorator(new DbProductRepository(), new
Cache());
var product = productRepository.GetProduct(1);
```

Le résultat de l'appel de `GetProduct` sera le suivant: récupérer le produit à partir du cache (responsabilité du décorateur), si le produit n'était pas dans le cache, procéder à l'appel à `DbProductRepository` et extraire le produit de la base de données. Une fois que ce produit peut être ajouté au cache, les appels suivants n'atteindront pas le DB.

Lire Motif de décorateur en ligne: https://riptutorial.com/fr/design-patterns/topic/1720/motif-de-decorateur

Chapitre 21: Motif de visiteur

Examples

Exemple de motif visiteur en C ++

Au lieu de

```
struct IShape
{
    virtual ~IShape() = default;

    virtual void print() const = 0;
    virtual double area() const = 0;
    virtual double perimeter() const = 0;
    // .. and so on
};
```

Les visiteurs peuvent être utilisés:

```
// The concrete shapes
struct Square;
struct Circle;

// The visitor interface
struct IShapeVisitor
{
    virtual ~IShapeVisitor() = default;
    virtual void visit(const Square&) = 0;
    virtual void visit(const Circle&) = 0;
};

// The shape interface
struct IShape
{
    virtual ~IShape() = default;

    virtual void accept(IShapeVisitor&) const = 0;
};
```

Maintenant, le béton se forme:

```
struct Point {
    double x;
    double y;
};

struct Circle : IShape
{
    Circle(const Point& center, double radius) : center(center), radius(radius) {}

    // Each shape has to implement this method the same way
    void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }
```

```
    Point center;
    double radius;
};

struct Square : IShape
{
    Square(const Point& topLeft, double sideLength) :
         topLeft(topLeft), sideLength(sideLength)
    {}

    // Each shape has to implement this method the same way
    void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }

    Point topLeft;
    double sideLength;
};
```

puis les visiteurs:

```
struct ShapePrinter : IShapeVisitor
{
    void visit(const Square&) override { std::cout << "Square"; }
    void visit(const Circle&) override { std::cout << "Circle"; }
};

struct ShapeAreaComputer : IShapeVisitor
{
    void visit(const Square& square) override
    {
        area = square.sideLength * square.sideLength;
    }

    void visit(const Circle& circle) override
    {
         area = M_PI * circle.radius * circle.radius;
    }

    double area = 0;
};

struct ShapePerimeterComputer : IShapeVisitor
{
    void visit(const Square& square) override { perimeter = 4. * square.sideLength; }
    void visit(const Circle& circle) override { perimeter = 2. * M_PI * circle.radius; }

    double perimeter = 0.;
};
```

Et l'utiliser:

```
const Square square = {{-1., -1.}, 2.};
const Circle circle{{0., 0.}, 1.};
const IShape* shapes[2] = {&square, &circle};

ShapePrinter shapePrinter;
ShapeAreaComputer shapeAreaComputer;
ShapePerimeterComputer shapePerimeterComputer;

for (const auto* shape : shapes) {
```

```
    shape->accept(shapePrinter);
    std::cout << " has an area of ";

    // result will be stored in shapeAreaComputer.area
    shape->accept(shapeAreaComputer);

    // result will be stored in shapePerimeterComputer.perimeter
    shape->accept(shapePerimeterComputer);

    std::cout << shapeAreaComputer.area
              << ", and a perimeter of "
              << shapePerimeterComputer.perimeter
              << std::endl;
}
```

Production attendue:

```
Square has an area of 4, and a perimeter of 8
Circle has an area of 3.14159, and a perimeter of 6.28319
```

Démo

Explication :

- Dans `void Square::accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }` , le type statique de `this` est connu, et donc la surcharge choisie (au moment de la compilation) est `void IVisitor::visit(const Square&);` .
- Pour `square.accept(visitor);` appel, la répartition dynamique par `virtual` est utilisée pour savoir qui `accept` d'appeler.

Avantages :

- Vous pouvez ajouter de nouvelles fonctionnalités (`SerializeAsXml` , ...) à la classe `IShape` en ajoutant simplement un nouveau visiteur.

Contre :

- L'ajout d'une nouvelle forme concrète (`Triangle` , ...) nécessite de modifier tous les visiteurs.

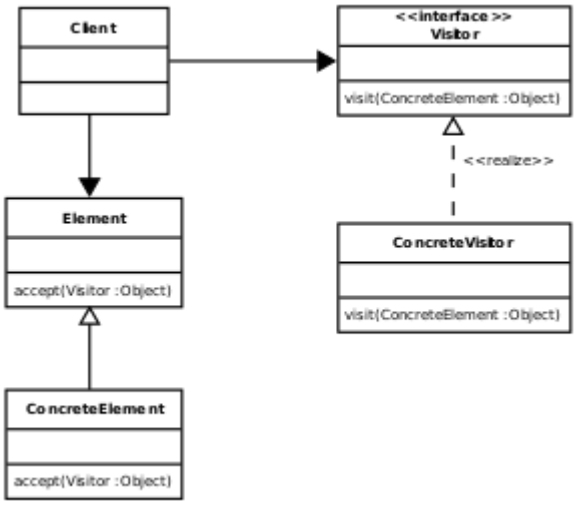
L'alternative `IShape` à placer toutes les fonctionnalités en tant que méthodes `virtual` dans `IShape` présente des avantages et des inconvénients opposés: l'ajout de nouvelles fonctionnalités nécessite de modifier toutes les formes existantes, mais l'ajout d'une nouvelle forme n'affecte pas les classes existantes.

Exemple de modèle de visiteur en java

`Visitor` modèle de `Visitor` vous permet d'ajouter de nouvelles opérations ou méthodes à un ensemble de classes sans modifier la structure de ces classes.

Ce modèle est particulièrement utile lorsque vous souhaitez centraliser une opération particulière sur un objet sans étendre l'objet ou sans modifier l'objet.

Diagramme UML de Wikipedia:



Extrait de code:

```
import java.util.HashMap;

interface Visitable{
    void accept(Visitor visitor);
}

interface Visitor{
    void logGameStatistics(Chess chess);
    void logGameStatistics(Checkers checkers);
    void logGameStatistics(Ludo ludo);
}
class GameVisitor implements Visitor{
    public void logGameStatistics(Chess chess){
        System.out.println("Logging Chess statistics: Game Completion duration, number of
moves etc..");
    }
    public void logGameStatistics(Checkers checkers){
        System.out.println("Logging Checkers statistics: Game Completion duration, remaining
coins of loser");
    }
    public void logGameStatistics(Ludo ludo){
        System.out.println("Logging Ludo statistics: Game Completion duration, remaining coins
of loser");
    }
}

abstract class Game{
    // Add game related attributes and methods here
    public Game(){

    }
    public void getNextMove(){};
    public void makeNextMove(){}
    public abstract String getName();
}
class Chess extends Game implements Visitable{
    public String getName(){
        return Chess.class.getName();
    }
```

```
    public void accept(Visitor visitor){
        visitor.logGameStatistics(this);
    }
}
class Checkers extends Game implements Visitable{
    public String getName(){
        return Checkers.class.getName();
    }
    public void accept(Visitor visitor){
        visitor.logGameStatistics(this);
    }
}
class Ludo extends Game implements Visitable{
    public String getName(){
        return Ludo.class.getName();
    }
    public void accept(Visitor visitor){
        visitor.logGameStatistics(this);
    }
}

public class VisitorPattern{
    public static void main(String args[]){
        Visitor visitor = new GameVisitor();
        Visitable games[] = { new Chess(),new Checkers(), new Ludo()};
        for (Visitable v : games){
            v.accept(visitor);
        }
    }
}
```

Explication:

1. `Visitable` (`Element`) est une interface et cette méthode d'interface doit être ajoutée à un ensemble de classes.
2. `Visitor` est une interface qui contient des méthodes permettant d'effectuer une opération sur les éléments `Visitable` .
3. `GameVisitor` est une classe qui implémente l'interface de `Visitor` (`ConcreteVisitor`).
4. Chaque élément `Visitable` accepte les `Visitor` et appelle une méthode appropriée de l'interface du `Visitor` .
5. Vous pouvez traiter `Game` as `Element` et des jeux concrets tels que `Chess,Checkers and Ludo` comme `ConcreteElements` .

Dans l'exemple ci-dessus, `Chess, Checkers and Ludo` sont trois jeux différents (et des classes `Visitable`). Un beau jour, j'ai rencontré un scénario pour enregistrer les statistiques de chaque jeu. Donc, sans modifier la classe individuelle pour implémenter la fonctionnalité de statistiques, vous pouvez centraliser cette responsabilité dans la classe `GameVisitor` , ce qui fait l'affaire pour vous sans modifier la structure de chaque jeu.

sortie:

```
Logging Chess statistics: Game Completion duration, number of moves etc..
Logging Checkers statistics: Game Completion duration, remaining coins of loser
Logging Ludo statistics: Game Completion duration, remaining coins of loser
```

Cas d'utilisation / Applicabilité:

1. *Des opérations similaires doivent être effectuées* sur des objets de différents types regroupés dans une structure
2. Vous devez exécuter de nombreuses opérations distinctes et non liées. *Il sépare l'opération de la structure des objets*
3. De nouvelles opérations doivent être ajoutées *sans modification de la structure de l'objet*
4. *Rassemblez les opérations liées dans une classe unique* plutôt que de vous forcer à changer ou à dériver des classes
5. Ajoutez des fonctions aux bibliothèques de classes pour lesquelles vous *n'avez pas la source ou ne pouvez pas changer la source*

Références supplémentaires:

oodesign

fabrication de pain

Exemple de visiteur en C ++

```
// A simple class hierarchy that uses the visitor to add functionality.
//
class VehicleVisitor;
class Vehicle
{
    public:
        // To implement the visitor pattern
        // The class simply needs to implement the accept method
        // That takes a reference to a visitor object that provides
        // new functionality.
        virtual void accept(VehicleVisitor& visitor) = 0
};
class Plane: public Vehicle
{
    public:
        // Each concrete representation simply calls the visit()
        // method on the visitor object passing itself as the parameter.
        virtual void accept(VehicleVisitor& visitor) {visitor.visit(*this);}

        void fly(std::string const& destination);
};
class Train: public Vehicle
{
    public:
        virtual void accept(VehicleVisitor& visitor) {visitor.visit(*this);}

        void locomote(std::string const& destination);
};
class Automobile: public Vehicle
{
    public:
        virtual void accept(VehicleVisitor& visitor) {visitor.visit(*this);}

        void drive(std::string const& destination);
};
```

```
class VehicleVisitor
{
    public:
        // The visitor interface implements one method for each class in the
        // hierarchy. When implementing new functionality you just create the
        // functionality required for each type in the appropriate method.

        virtual void visit(Plane& object)      = 0;
        virtual void visit(Train& object)      = 0;
        virtual void visit(Automobile& object) = 0;

    // Note: because each class in the hierarchy needs a virtual method
    // in visitor base class this makes extending the hierarchy ones defined
    // hard.
};
```

Un exemple d'utilisation:

```
// Add the functionality `Move` to an object via a visitor.
class MoveVehicleVisitor
{
    std::string const& destination;
    public:
        MoveVehicleVisitor(std::string const& destination)
            : destination(destination)
    {}
    virtual void visit(Plane& object)      {object.fly(destination);}
    virtual void visit(Train& object)      {object.locomote(destination);}
    virtual void visit(Automobile& object) {object.drive(destination);}
};

int main()
{
    MoveVehicleVisitor  moveToDenver("Denver");
    Vehicle&            object = getObjectToMove();
    object.accept(moveToDenver);
}
```

Traverser de grands objets

Le Pattern visiteur peut être utilisé pour parcourir des structures.

```
class GraphVisitor;
class Graph
{
    public:
        class Node
        {
            using Link = std::set<Node>::iterator;
            std::set<Link>   linkTo;
            public:
                void accept(GraphVisitor& visitor);
        };

        void accept(GraphVisitor& visitor);

    private:
        std::set<Node>  nodes;
```

```
};

class GraphVisitor
{
    std::set<Graph::Node*>  visited;
    public:
        void visit(Graph& graph)
        {
            visited.clear();
            doVisit(graph);
        }
        bool visit(Graph::Node& node)
        {
            if (visited.find(&node) != visited.end()) {
                return false;
            }
            visited.insert(&node);
            doVisit(node);
            return true;
        }
    private:
        virtual void doVisit(Graph& graph)      = 0;
        virtual void doVisit(Graph::Node& node) = 0;
};

void accept(GraphVisitor& visitor)
{
    // Pass the graph to the visitor.
    visitor.visit(*this);

    // Then do a depth first search of the graph.
    // In this situation it is the visitors responsibility
    // to keep track of visited nodes.
    for(auto& node: nodes) {
        node.accept(visitor);
    }
}
void Graph::Node::accept(GraphVisitor& visitor)
{
    // Tell the visitor it is working on a node and see if it was
    // previously visited.
    if (visitor.visit(*this)) {

        // The pass the visitor to all the linked nodes.
        for(auto& link: linkTo) {
            link->accept(visitor);
        }
    }
}
```

Lire Motif de visiteur en ligne: https://riptutorial.com/fr/design-patterns/topic/4579/motif-de-visiteur

Chapitre 22: Motif Prototype

Introduction

Le modèle Prototype est un modèle de création qui crée de nouveaux objets en clonant un objet prototype existant. Le modèle prototype accélère l'instanciation des classes lorsque la copie d'objets est plus rapide.

Remarques

Le motif prototype est un motif de conception créative. Il est utilisé lorsque le type d'objets à créer est déterminé par une instance de prototype, qui est *"clonée"* pour produire de nouveaux objets.

Ce modèle est utilisé lorsqu'une classe a besoin d'un constructeur *"polymorphe (copie)"* .

Examples

Motif Prototype (C ++)

```
class IPrototype  {
public:
    virtual ~IPrototype() = default;

    auto Clone() const { return std::unique_ptr<IPrototype>{DoClone()}; }
    auto Create() const { return std::unique_ptr<IPrototype>{DoCreate()}; }

private:
    virtual IPrototype* DoClone() const = 0;
    virtual IPrototype* DoCreate() const = 0;
};

class A : public IPrototype {
public:
    auto Clone() const { return std::unique_ptr<A>{DoClone()}; }
    auto Create() const { return std::unique_ptr<A>{DoCreate()}; }
private:
    // Use covariant return type :)
    A* DoClone() const override { return new A(*this); }
    A* DoCreate() const override { return new A; }
};

class B : public IPrototype {
public:
    auto Clone() const { return std::unique_ptr<B>{DoClone()}; }
    auto Create() const { return std::unique_ptr<B>{DoCreate()}; }
private:
    // Use covariant return type :)
    B* DoClone() const override { return new B(*this); }
    B* DoCreate() const override { return new B; }
};
```

```
class ChildA : public A {
public:
    auto Clone() const { return std::unique_ptr<ChildA>{DoClone()}; }
    auto Create() const { return std::unique_ptr<ChildA>{DoCreate()}; }

private:
    // Use covariant return type :)
    ChildA* DoClone() const override { return new ChildA(*this); }
    ChildA* DoCreate() const override { return new ChildA; }
};
```

Cela permet de construire la classe dérivée à partir d'un pointeur de classe de base:

```
ChildA childA;
A& a = childA;
IPrototype& prototype = a;

// Each of the following will create a copy of `ChildA`:
std::unique_ptr<ChildA> clone1 = childA.Clone();
std::unique_ptr<A> clone2 = a.Clone();
std::unique_ptr<IPrototype> clone3 = prototype.Clone();

// Each of the following will create a new default instance `ChildA`:
std::unique_ptr<ChildA> instance1 = childA.Create();
std::unique_ptr<A> instance2 = a.Create();
std::unique_ptr<IPrototype> instance3 = prototype.Create();
```

Motif Prototype (C #)

Le modèle prototype peut être implémenté à l'aide de l'interface ICloneable dans .NET.

```
class Spoon {
}
class DessertSpoon : Spoon, ICloneable {
  ...
  public object Clone() {
    return this.MemberwiseClone();
  }
}
class SoupSpoon : Spoon, ICloneable {
  ...
  public object Clone() {
    return this.MemberwiseClone();
  }
}
```

Motif Prototype (JavaScript)

Dans les langages classiques comme Java, C # ou C ++, nous commençons par créer une classe, puis nous pouvons créer de nouveaux objets à partir de la classe ou nous pouvons étendre la classe.

En JavaScript, nous créons d'abord un objet, puis nous pouvons augmenter l'objet ou en créer de nouveaux. Donc, je pense, JavaScript montre un prototype réel que le langage classique.

Exemple :

```
var myApp = myApp || {};

myApp.Customer = function (){
    this.create = function () {
        return "customer added";
    }
};

myApp.Customer.prototype = {
    read: function (id) {
        return "this is the customer with id = " + id;
    },
    update: function () {
        return "customer updated";
    },
    remove: function () {
        return "customer removed";
    }
};
```

Ici, nous créons un objet nommé `Customer` , puis sans créer de *nouvel objet,* nous avons étendu l' `Customer object` existant à l'aide d' *un* mot-clé *prototype* . Cette technique est appelée **modèle de prototype** .

Lire Motif Prototype en ligne: https://riptutorial.com/fr/design-patterns/topic/5867/motif-prototype

Chapitre 23: Multiton

Remarques

Multitonite

Comme Singleton , Multiton peut être considéré comme une mauvaise pratique. Cependant, il y a des moments où vous pouvez l'utiliser avec sagesse (par exemple, si vous créez un système comme ORM / ODM pour conserver plusieurs objets).

Examples

Pool of Singletons (exemple PHP)

Multiton peut être utilisé comme conteneur pour les singletons. Ceci est l'implémentation de Multiton est une combinaison de motifs Singleton et Pool.

Voici un exemple de la façon dont la classe de pool abstraite Multiton peut être créée:

```
abstract class MultitonPoolAbstract
{
    /**
     * @var array
     */
    protected static $instances = [];

    final protected function __construct() {}

    /**
     * Get class name of lately binded class
     *
     * @return string
     */
    final protected static function getClassName()
    {
        return get_called_class();
    }

    /**
     * Instantiates a calling class object
     *
     * @return static
     */
    public static function getInstance()
    {
        $className = static::getClassName();

        if( !isset(self::$instances[$className]) ) {
            self::$instances[$className] = new $className;
        }

        return self::$instances[$className];
    }
```

```
    /**
     * Deletes a calling class object
     *
     * @return void
     */
    public static function deleteInstance()
    {
        $className = static::getClassName();

        if( isset(self::$instances[$className]) )
            unset(self::$instances[$className]);
    }

    /*------------------------------------------------------------------------
    | Seal methods that can instantiate the class
    |-------------------------------------------------------------------------*/

    final protected function __clone() {}

    final protected function __sleep() {}

    final protected function __wakeup() {}
}
```

De cette façon, nous pouvons instancier plusieurs pools de singleton.

Registre des singletons (exemple PHP)

Ce modèle peut être utilisé pour contenir des pools de singletons enregistrés, chacun se distinguant par un identifiant unique:

```
abstract class MultitonRegistryAbstract
{
    /**
     * @var array
     */
    protected static $instances = [];

    /**
     * @param string $id
     */
    final protected function __construct($id) {}

    /**
     * Get class name of lately binded class
     *
     * @return string
     */
    final protected static function getClassName()
    {
        return get_called_class();
    }

    /**
     * Instantiates a calling class object
     *
     * @return static
```

```
     */
    public static function getInstance($id)
    {
        $className = static::getClassName();

        if( !isset(self::$instances[$className]) ) {
            self::$instances[$className] = [$id => new $className($id)];
        } else {
            if( !isset(self::$instances[$className][$id]) ) {
                self::$instances[$className][$id] = new $className($id);
            }
        }

        return self::$instances[$className][$id];
    }

    /**
     * Deletes a calling class object
     *
     * @return void
     */
    public static function unsetInstance($id)
    {
        $className = static::getClassName();

        if( isset(self::$instances[$className]) ) {
            if( isset(self::$instances[$className][$id]) ) {
                unset(self::$instances[$className][$id]);
            }

            if( empty(self::$instances[$className]) ) {
                unset(self::$instances[$className]);
            }
        }
    }

    /*-------------------------------------------------------------------------
    | Seal methods that can instantiate the class
    |-------------------------------------------------------------------------*/

    final protected function __clone() {}

    final protected function __sleep() {}

    final protected function __wakeup() {}
}
```

Ceci est une forme simplifiée de modèle qui peut être utilisé pour ORM pour stocker plusieurs entités d'un type donné.

Lire Multiton en ligne: https://riptutorial.com/fr/design-patterns/topic/6857/multiton

Chapitre 24: MVC, MVVM et MVP

Remarques

On peut faire valoir que les modèles MVC et connexes sont en réalité des modèles d'architecture logicielle plutôt que des modèles de conception de logiciels.

Examples

Model View Controller (MVC)

1. Qu'est-ce que MVC?

Le modèle MVC (Model View Controller) est un modèle de conception le plus couramment utilisé pour créer des interfaces utilisateur. L'avantage majeur de MVC est qu'il sépare:

- la représentation interne de l'état de l'application (le modèle),
- comment l'information est présentée à l'utilisateur (la vue), et
- la logique qui contrôle la manière dont l'utilisateur interagit avec l'état de l'application (le contrôleur).

2. Cas d'utilisation de MVC

Le principal cas d'utilisation de MVC réside dans la programmation de l'interface utilisateur graphique (GUI). Le composant View écoute le composant Model pour les modifications. Le modèle agit en tant que diffuseur; lorsqu'il existe un mode de modification du modèle, il diffuse ses modifications dans la vue et le contrôleur. Le contrôleur est utilisé par la vue pour modifier le composant de modèle.

3. Mise en œuvre

Considérons l'implémentation suivante de MVC, où nous avons une classe Model appelée `Animals` , une classe View appelée `DisplayAnimals` , et une classe de contrôleur appelée `AnimalController` . L'exemple ci-dessous est une version modifiée du didacticiel sur MVC de Design Patterns - MVC Pattern .

```
/* Model class */
public class Animals {
    private String name;
    private String gender;

    public String getName() {
        return name;
    }

    public String getGender() {
        return gender;
    }
```

```
    public void setName(String name) {
         this.name = name;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }
}

/* View class */
public class DisplayAnimals {
    public void printAnimals(String tag, String gender) {
        System.out.println("My Tag name for Animal:" + tag);
        System.out.println("My gender: " + gender);
    }
}

/* Controller class */
public class AnimalController {
     private Animal model;
     private DisplayAnimals view;

   public AnimalController(Animal model, DisplayAnimals view) {
      this.model = model;
      this.view = view;
   }

   public void setAnimalName(String name) {
      model.setName(name);
   }

   public String getAnimalName() {
      return model.getName();
   }

   public void setAnimalGender(String animalGender) {
      model.setGender(animalGender);
   }

   public String getGender() {
      return model.getGender();
   }

   public void updateView() {
      view.printAnimals(model.getName(), model.getGender());
   }
}
```

4. Sources utilisées:

Modèles de conception - Modèle MVC

Java SE Application Design avec MVC

Modèle Vue Contrôleur

Modèle View ViewModel (MVVM)

1. Qu'est-ce que MVVM?

Le modèle ViewModel (MVVM) de vue modèle est un modèle de conception le plus couramment utilisé pour créer des interfaces utilisateur. Il est dérivé du modèle populaire "Model View Controller" (MVC). L'avantage majeur de MVVM est qu'il sépare:

- La représentation interne de l'état de l'application (le modèle).
- Comment l'information est présentée à l'utilisateur (la vue).
- La "logique de conversion de valeur" responsable de l'exposition et de la conversion des données du modèle afin que les données puissent être facilement gérées et présentées dans la vue (le ViewModel).

2. Cas d'utilisation de MVVM

Le principal cas d'utilisation de MVVM est la programmation de l'interface utilisateur graphique (GUI). Il est utilisé pour programmer simplement les interfaces utilisateur en fonction des événements en séparant la couche de vue de la logique de gestion gérant les données.

Dans Windows Presentation Foundation (WPF), par exemple, la vue est conçue à l'aide du langage de balisage de structure XAML. Les fichiers XAML sont liés à ViewModels à l'aide de la liaison de données. De cette façon, la vue est uniquement responsable de la présentation et la vue est uniquement responsable de la gestion de l'état de l'application en travaillant sur les données du modèle.

Il est également utilisé dans la bibliothèque JavaScript KnockoutJS.

3. Mise en œuvre

Considérez l'implémentation suivante de MVVM en utilisant C # .Net et WPF. Nous avons une classe Model appelée Animals, une classe View implémentée dans Xaml et un ViewModel appelé AnimalViewModel. L'exemple ci-dessous est une version modifiée du didacticiel sur MVC de Design Patterns - MVC Pattern .

Regardez comment le modèle ne sait rien, le ViewModel ne connaît que le modèle et la vue ne connaît que le ViewModel.

L'événement OnNotifyPropertyChanged permet de mettre à jour le modèle et la vue afin que, lorsque vous entrez quelque chose dans la zone de texte de la vue, le modèle soit mis à jour. Et si quelque chose met à jour le modèle, la vue est mise à jour.

```
/*Model class*/
public class Animal
{
    public string Name { get; set; }

    public string Gender { get; set; }
}

/*ViewModel class*/
public class AnimalViewModel : INotifyPropertyChanged
{
```

```
    private Animal _model;

    public AnimalViewModel()
    {
        _model = new Animal {Name = "Cat", Gender = "Male"};
    }

    public string AnimalName
    {
        get { return _model.Name; }
        set
        {
            _model.Name = value;
            OnPropertyChanged("AnimalName");
        }
    }

    public string AnimalGender
    {
        get { return _model.Gender; }
        set
        {
            _model.Gender = value;
            OnPropertyChanged("AnimalGender");
        }
    }

    //Event binds view to ViewModel.
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged(string propertyName)
    {
        if (this.PropertyChanged != null)
        {
            var e = new PropertyChangedEventArgs(propertyName);
            this.PropertyChanged(this, e);
        }
    }
}


<!-- Xaml View -->
<Window x:Class="WpfApplication6.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525"
        xmlns:viewModel="clr-namespace:WpfApplication6">

    <Window.DataContext>
        <viewModel:AnimalViewModel/>
    </Window.DataContext>

    <StackPanel>
        <TextBox Text="{Binding AnimalName}" Width="120" />
        <TextBox Text="{Binding AnimalGender}" Width="120" />
    </StackPanel>
</Window>
```

4. Sources utilisées:

Model – view – viewmodel

Un exemple simple de MVVM

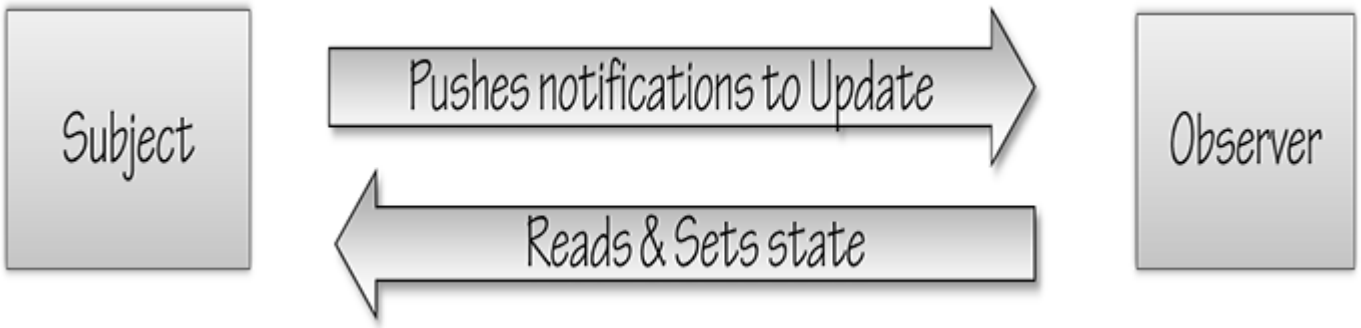
L'exemple MVVM C # WPF le plus simple au monde

Le pattern MVVM

Lire MVC, MVVM et MVP en ligne: https://riptutorial.com/fr/design-patterns/topic/7405/mvc--mvvm-et-mvp

Chapitre 25: Observateur

Remarques



Quelle est l'intention?

- Adoptez le principe de la séparation des préoccupations.
- Créez une séparation entre le sujet et l'observateur.
- Permettre à plusieurs observateurs de réagir pour changer un seul sujet.

Quelle est la structure?

- L'objet fournit un moyen de s'inscrire, d'annuler l'enregistrement, de notifier.
- Observer fournit un moyen de mise à jour.

Examples

Observateur / Java

Le modèle d'observateur permet aux utilisateurs d'une classe de s'abonner aux événements qui se produisent lorsque cette classe traite des données, etc. et d'être avertis lorsque ces événements se produisent. Dans l'exemple suivant, nous créons une classe de traitement et une classe d'observateurs qui seront notifiées lors du traitement d'une phrase - si elle trouve des mots de plus de 5 lettres.

L'interface `LongWordsObserver` définit l'observateur. Implémentez cette interface afin d'inscrire un observateur aux événements.

```
// an observe that can be registered and receive notifications
public interface LongWordsObserver {
    void notify(WordEvent event);
}
```

La classe `WordEvent` est l'événement qui sera envoyé aux classes d'observateur une fois que certains événements se sont produits (dans ce cas, les mots longs ont été trouvés)

```
// An event class which contains the long word that was found
```

```
public class WordEvent {

    private String word;

    public WordEvent(String word) {
        this.word = word;
    }

    public String getWord() {
        return word;
    }
}
```

La classe `PhraseProcessor` est la classe qui traite la phrase donnée. Il permet aux observateurs d'être enregistrés en utilisant la méthode `addObserver` . Une fois les mots longs trouvés, ces observateurs seront appelés en utilisant une instance de la classe `WordEvent` .

```
import java.util.ArrayList;
import java.util.List;

public class PhraseProcessor {

    // the list of observers
    private List<LongWordsObserver> observers = new ArrayList<>();

    // register an observer
    public void addObserver(LongWordsObserver observer) {
        observers.add(observer);
    }

    // inform all the observers that a long word was found
    private void informObservers(String word) {
        observers.forEach(o -> o.notify(new WordEvent(word)));
    }

    // the main method - process a phrase and look for long words. If such are found,
    // notify all the observers
    public void process(String phrase) {
        for (String word : phrase.split(" ")) {
            if (word.length() > 5) {
                informObservers(word);
            }
        }
    }
}
```

La classe `LongWordsExample` montre comment enregistrer des observateurs, appeler la méthode `process` et recevoir des alertes lorsque des mots longs ont été trouvés.

```
import java.util.ArrayList;
import java.util.List;

public class LongWordsExample {

    public static void main(String[] args) {

        // create a list of words to be filled when long words were found
        List<String> longWords = new ArrayList<>();
```

```
        // create the PhraseProcessor class
        PhraseProcessor processor = new PhraseProcessor();

        // register an observer and specify what it should do when it receives events,
        // namely to append long words in the longwords list
        processor.addObserver(event -> longWords.add(event.getWord()));

        // call the process method
        processor.process("Lorem ipsum dolor sit amet, consectetuer adipiscing elit");

        // show the list of long words after the processing is done
        System.out.println(String.join(", ", longWords));
        // consectetuer, adipiscing
    }
}
```

Observateur utilisant IObservable et IObserver (C #)

`IObserver<T>` et `IObservable<T>` peuvent être utilisées pour implémenter un motif d'observateur dans .NET

- `IObservable<T>` représente la classe qui envoie les notifications
- `IObserver<T>` représente la classe qui les reçoit

```
public class Stock {
  private string Symbol { get; set; }
  private decimal Price { get; set; }
}

public class Investor : IObserver<Stock> {
  public IDisposable unsubscriber;
  public virtual void Subscribe(IObservable<Stock> provider) {
    if(provider != null) {
      unsubscriber = provider.Subscribe(this);
    }
  }
  public virtual void OnCompleted() {
    unsubscriber.Dispose();
  }
  public virtual void OnError(Exception e) {
  }
  public virtual void OnNext(Stock stock) {
  }
}

public class StockTrader : IObservable<Stock> {
  public StockTrader() {
    observers = new List<IObserver<Stock>>();
  }
  private IList<IObserver<Stock>> observers;
  public IDisposable Subscribe(IObserver<Stock> observer) {
    if(!observers.Contains(observer)) {
      observers.Add(observer);
    }
    return new Unsubscriber(observers, observer);
  }
  public class Unsubscriber : IDisposable {
```

```
    private IList<IObserver<Stock>> _observers;
    private IObserver<Stock> _observer;

    public Unsubscriber(IList<IObserver<Stock>> observers, IObserver<Stock> observer) {
      _observers = observers;
      _observer = observer;
    }

    public void Dispose() {
      Dispose(true);
    }
    private bool _disposed = false;
    protected virtual void Dispose(bool disposing) {
      if(_disposed) {
        return;
      }
      if(disposing) {
        if(_observer != null && _observers.Contains(_observer)) {
          _observers.Remove(_observer);
        }
      }
      _disposed = true;
    }
  }
  public void Trade(Stock stock) {
    foreach(var observer in observers) {
      if(stock== null) {
        observer.OnError(new ArgumentNullException());
      }
      observer.OnNext(stock);
    }
  }
  public void End() {
    foreach(var observer in observers.ToArray()) {
      observer.OnCompleted();
    }
    observers.Clear();
  }
}
```

Usage

```
...
var provider = new StockTrader();
var i1 = new Investor();
i1.Subscribe(provider);
var i2 = new Investor();
i2.Subscribe(provider);

provider.Trade(new Stock());
provider.Trade(new Stock());
provider.Trade(null);
provider.End();
...
```

REF: Modèles et pratiques de conception dans .NET: le modèle Observer

Lire Observateur en ligne: https://riptutorial.com/fr/design-patterns/topic/3185/observateur

Chapitre 26: Ouvrir le principe de fermeture

Introduction

Le principe d'ouverture et de fermeture stipule que la conception et la rédaction du code doivent être effectuées de manière à ce que les nouvelles fonctionnalités soient ajoutées avec un minimum de modifications dans le code existant. La conception doit être faite de manière à permettre l'ajout de nouvelles fonctionnalités en tant que nouvelles classes, en conservant autant que possible le code existant. Les entités logicielles telles que les classes, les modules et les fonctions doivent être ouvertes pour extension, mais fermées pour modification.

Remarques

Comme tous les principes, Open Close Principe n'est qu'un principe. Faire une conception flexible implique du temps et des efforts supplémentaires pour cela et cela introduit un nouveau niveau d'abstraction augmentant la complexité du code. Ce principe devrait donc être appliqué dans les domaines les plus susceptibles d'être modifiés. De nombreux modèles de conception nous aident à étendre le code sans le modifier, par exemple le décorateur.

Examples

Ouvrir Fermer Principe de violation

```
/*
* This design have some major issues
* For each new shape added the unit testing
* of the GraphicEditor should be redone
* When a new type of shape is added the time
* for adding it will be high since the developer
* who add it should understand the logic
* of the GraphicEditor.
* Adding a new shape might affect the existing
* functionality in an undesired way,
* even if the new shape works perfectly
*/

class GraphicEditor {
    public void drawShape(Shape s) {
        if (s.m_type==1)
            drawRectangle(s);
        else if (s.m_type==2)
            drawCircle(s);
    }
    public void drawCircle(Circle r) {....}
    public void drawRectangle(Rectangle r) {....}
}

 class Shape {
     int m_type;
 }
```

```
class Rectangle extends Shape {
    Rectangle() {
        super.m_type=1;
    }
}

class Circle extends Shape {
    Circle() {
        super.m_type=2;
    }
}
```

Open Close Principle support

```
/*
* For each new shape added the unit testing
* of the GraphicEditor should not be redone
* No need to understand the sourcecode
* from GraphicEditor.
* Since the drawing code is moved to the
* concrete shape classes, it's a reduced risk
* to affect old functionallity when new
* functionallity is added.
*/
class GraphicEditor {
     public void drawShape(Shape s) {
         s.draw();
     }
 }

 class Shape {
     abstract void draw();
 }

 class Rectangle extends Shape  {
     public void draw() {
         // draw the rectangle
     }
 }
```

Lire Ouvrir le principe de fermeture en ligne: https://riptutorial.com/fr/design-patterns/topic/9199/ouvrir-le-principe-de-fermeture

Chapitre 27: Publier-S'abonner

Examples

Publier-S'abonner en Java

L'éditeur-abonné est un concept familier étant donné l'essor de YouTube, de Facebook et d'autres services de médias sociaux. Le concept de base est qu'un `Publisher` génère du contenu et un `Subscriber` qui consomme du contenu. Chaque fois que l' `Publisher` génère du contenu, chaque `Subscriber` est averti. `Subscribers` peuvent théoriquement être abonnés à plusieurs éditeurs.

Il existe généralement un `ContentServer` situé entre l'éditeur et l'abonné pour faciliter la transmission des messages.

```
public class Publisher {
    ...
    public Publisher(Topic t) {
        this.topic = t;
    }

    public void publish(Message m) {
        ContentServer.getInstance().sendMessage(this.topic, m);
    }
}
```

```
public class ContentServer {
    private Hashtable<Topic, List<Subscriber>> subscriberLists;

    private static ContentServer serverInstance;

    public static ContentServer getInstance() {
        if (serverInstance == null) {
            serverInstance = new ContentServer();
        }
        return serverInstance;
    }

    private ContentServer() {
        this.subscriberLists = new Hashtable<>();
    }

    public sendMessage(Topic t, Message m) {
        List<Subscriber> subs = subscriberLists.get(t);
        for (Subscriber s : subs) {
            s.receivedMessage(t, m);
        }
    }

    public void registerSubscriber(Subscriber s, Topic t) {
        subscriberLists.get(t).add(s);
    }
```

```
public class Subscriber {
```

```
    public Subscriber(Topic...topics) {
        for (Topic t : topics) {
            ContentServer.getInstance().registerSubscriber(this, t);
        }
    }

    public void receivedMessage(Topic t, Message m) {
        switch(t) {
            ...
        }
    }
}
```

Habituellement, le modèle de conception de pub-sub est implémenté avec une vue multithreadée. L'une des implémentations les plus courantes considère chaque `Subscriber` comme un thread distinct, le `ContentServer` gérant un pool de threads.

Exemple de pub-sub simple en JavaScript

Les éditeurs et les abonnés n'ont pas besoin de se connaître. Ils communiquent simplement à l'aide de files d'attente de messages.

```
(function () {
        var data;

        setTimeout(function () {
            data = 10;
            $(document).trigger("myCustomEvent");
        }, 2000);

        $(document).on("myCustomEvent", function () {
            console.log(data);
        });
})();
```

Ici, nous avons publié un événement personnalisé nommé **myCustomEvent** et abonné à cet événement. Ils n'ont donc pas besoin de se connaître.

Lire Publier-S'abonner en ligne: https://riptutorial.com/fr/design-patterns/topic/7260/publier-s-abonner

Chapitre 28: Repository

Remarques

A propos de l'implémentation de `IEnumerable<TEntity> Get(Expression<Func<TEntity, bool>> filter)` : L'idée est d'utiliser des expressions comme `i => x.id == 17` pour écrire des requêtes génériques. C'est un moyen d'interroger des données sans utiliser le langage de requête spécifique de votre technologie. L'implémentation est plutôt étendue, vous pouvez donc envisager d'autres alternatives, comme des méthodes spécifiques sur vos référentiels implémentés: Un `CompanyRepository` imaginaire pourrait fournir la méthode `GetByName(string name)` .

Examples

Référentiels en lecture seule (C #)

Un modèle de référentiel peut être utilisé pour encapsuler un code spécifique au stockage de données dans des composants désignés. La partie de votre application qui nécessite les données ne fonctionnera qu'avec les référentiels. Vous souhaiterez créer un référentiel pour chaque combinaison d'éléments que vous stockez et votre technologie de base de données.

Les référentiels en lecture seule peuvent être utilisés pour créer des référentiels non autorisés à manipuler des données.

Les interfaces

```
public interface IReadOnlyRepository<TEntity, TKey> : IRepository
{
    IEnumerable<TEntity> Get(Expression<Func<TEntity, bool>> filter);

    TEntity Get(TKey id);
}

public interface IRepository<TEntity, TKey> : IReadOnlyRepository<TEntity, TKey>
{
    TKey Add(TEntity entity);

    bool Delete(TKey id);

    TEntity Update(TKey id, TEntity entity);
}
```

Un exemple d'implémentation utilisant ElasticSearch comme technologie (avec

NEST)

```
public abstract class ElasticReadRepository<TModel> : IReadOnlyRepository<TModel, string>
    where TModel : class
{

    protected ElasticClient Client;

    public ElasticReadRepository()
    {
        Client = Connect();
    }

    protected abstract ElasticClient Connect();

    public TModel Get(string id)
    {
        return Client.Get<TModel>(id).Source;
    }

    public IEnumerable<TModel> Get(Expression<Func<TModel, bool>> filter)
    {
        /* To much code for this example */
        throw new NotImplementedException();
    }
}

public abstract class ElasticRepository<TModel>
    : ElasticReadRepository<TModel>, IRepository<TModel, string>
    where TModel : class
{
    public string Add(TModel entity)
    {
        return Client.Index(entity).Id;
    }

    public bool Delete(string id)
    {
        return Client.Delete<TModel>(id).Found;
    }

    public TModel Update(string id, TModel entity)
    {
        return Connector.Client.Update<TModel>(
            id,
            update => update.Doc(entity)
        ).Get.Source;
    }
}
```

Grâce à cette implémentation, vous pouvez désormais créer des référentiels spécifiques pour les éléments que vous souhaitez stocker ou accéder. Lors de l'utilisation de la recherche élastique, il est courant que certains composants ne lisent que les données, il convient donc d'utiliser des référentiels en lecture seule.

Modèle de référentiel utilisant Entity Framework (C #)

Interface de référentiel;

```
public interface IRepository<T>
{
    void Insert(T entity);
    void Insert(ICollection<T> entities);
    void Delete(T entity);
    void Delete(ICollection<T> entity);
    IQueryable<T> SearchFor(Expression<Func<T, bool>> predicate);
    IQueryable<T> GetAll();
    T GetById(int id);
}
```

Référentiel générique;

```
public class Repository<T> : IRepository<T> where T : class
{
    protected DbSet<T> DbSet;

    public Repository(DbContext dataContext)
    {
        DbSet = dataContext.Set<T>();
    }

    public void Insert(T entity)
    {
        DbSet.Add(entity);
    }

    public void Insert(ICollection<T> entities)
    {
        DbSet.AddRange(entities);
    }

    public void Delete(T entity)
    {
        DbSet.Remove(entity);
    }

    public void Delete(ICollection<T> entities)
    {
        DbSet.RemoveRange(entities);
    }

    public IQueryable<T> SearchFor(Expression<Func<T, bool>> predicate)
    {
        return DbSet.Where(predicate);
    }

    public IQueryable<T> GetAll()
    {
        return DbSet;
    }

    public T GetById(int id)
    {
        return DbSet.Find(id);
    }
}
```

Exemple d'utilisation d'une classe d'hôtel de démonstration;

```
var db = new DatabaseContext();
var hotelRepo = new Repository<Hotel>(db);

var hotel = new Hotel("Hotel 1", "42 Wallaby Way, Sydney");
hotelRepo.Insert(hotel);
db.SaveChanges();
```

Lire Repository en ligne: https://riptutorial.com/fr/design-patterns/topic/6254/repository

Chapitre 29: Singleton

Remarques

Le motif de conception Singleton est parfois considéré comme " *Anti-pattern* ". Cela est dû au fait qu'il a des problèmes. Vous devez décider par vous-même si vous pensez qu'il convient de l'utiliser. Ce sujet a été discuté à plusieurs reprises sur StackOverflow.

Voir: http://stackoverflow.com/questions/137975/what-is-so-bad-about-singletons

Examples

Singleton (C #)

Les singletons sont utilisés pour garantir qu'une seule instance d'un objet est en cours de création. Le singleton ne permet de créer qu'une seule instance, ce qui signifie qu'il contrôle sa création. Le singleton est l'un des motifs de conception de Gang of Four et est un **modèle de création** .

Motif Singleton Thread-Safe

```
public sealed class Singleton
{
    private static Singleton _instance;
    private static object _lock = new object();

    private Singleton()
    {
    }

    public static Singleton GetSingleton()
    {
        if (_instance == null)
        {
             CreateSingleton();
        }

        return _instance;
    }

    private static void CreateSingleton()
    {
        lock (_lock )
        {
            if (_instance == null)
            {
                 _instance = new Singleton();
            }
        }
    }
}
```

Jon Skeet fournit l'implémentation suivante pour un singleton paresseux et sécurisé:

```
public sealed class Singleton
{
    private static readonly Lazy<Singleton> lazy =
        new Lazy<Singleton>(() => new Singleton());

    public static Singleton Instance { get { return lazy.Value; } }

    private Singleton()
    {
    }
}
```

Singleton (Java)

Les singletons en Java sont très similaires à C #, car les deux langages sont orientés objet. Vous trouverez ci-dessous un exemple de classe singleton, où une seule version de l'objet peut être active pendant la durée de vie du programme (en supposant que le programme fonctionne sur un seul thread)

```
public class SingletonExample {

    private SingletonExample() { }

    private static SingletonExample _instance;

    public static SingletonExample getInstance() {

        if (_instance == null) {
            _instance = new SingletonExample();
        }
        return _instance;
    }
}
```

Voici la version sécurisée du thread de ce programme:

```
public class SingletonThreadSafeExample {

    private SingletonThreadSafeExample () { }

    private static volatile SingletonThreadSafeExample _instance;

    public static SingletonThreadSafeExample getInstance() {
        if (_instance == null) {
                createInstance();
        }
        return _instance;
    }

    private static void createInstance() {
        synchronized(SingletonThreadSafeExample.class) {
            if (_instance == null) {
                _instance = new SingletonThreadSafeExample();
            }
```

```
        }
    }
}
```

Java a également un objet appelé `ThreadLocal` , qui crée une instance unique d'un objet sur une base de thread par thread. Cela pourrait être utile dans les applications où chaque thread a besoin de sa propre version de l'objet

```
public class SingletonThreadLocalExample {

    private SingletonThreadLocalExample () { }

    private static ThreadLocal<SingletonThreadLocalExample> _instance = new
ThreadLocal<SingletonThreadLocalExample>();

    public static SingletonThreadLocalExample getInstance() {
        if (_instance.get() == null) {
            _instance.set(new SingletonThreadLocalExample());
        }
        return _instance.get();
    }
}
```

Voici également une implémentation **Singleton** avec `enum` (contenant un seul élément):

```
public enum SingletonEnum {
    INSTANCE;
    // fields, methods
}
```

Toute implémentation de classe **Enum** garantit qu'il n'existe *qu'une seule* instance de chaque élément.

Bill Pugh Singleton Pattern

Bill Pugh Singleton Pattern est l'approche la plus utilisée pour la classe Singleton car elle ne nécessite pas de synchronisation

```
public class SingletonExample {

    private SingletonExample(){}

    private static class SingletonHolder{
        private static final SingletonExample INSTANCE = new SingletonExample();
    }

    public static SingletonExample getInstance(){
        return SingletonHolder.INSTANCE;
    }
}
```

avec l'utilisation de la classe statique interne privée, le titulaire n'est pas chargé en mémoire tant que quelqu'un n'appelle pas la méthode getInstance. La solution de Bill Pugh est thread-safe et ne nécessite pas de synchronisation.

Il y a d'autres exemples de singleton Java dans la rubrique Singletons sous la balise de documentation Java.

Singleton (C ++)

Selon Wiki : En génie logiciel, le modèle singleton est un modèle de conception qui limite l'instanciation d'une classe à un objet.

Cela est nécessaire pour créer exactement un objet pour coordonner les actions sur le système.

```
class Singleton
{
    // Private constructor so it can not be arbitrarily created.
    Singleton()
    {}
    // Disable the copy and move
    Singleton(Singleton const&)            = delete;
    Singleton& operator=(Singleton const&) = delete;
  public:

    // Get the only instance
    static Singleton& instance()
    {
        // Use static member.
        // Lazily created on first call to instance in thread safe way (after C++ 11)
        // Guaranteed to be correctly destroyed on normal application exit.
        static Singleton _instance;

        // Return a reference to the static member.
        return _instance;
    }
};
```

Lazy Singleton exemple pratique en java

Cas d'utilisation réelle pour le modèle Singleton;

Si vous développez une application client-serveur, vous avez besoin d'une seule instruction de `ConnectionManager` , qui gère le cycle de vie des connexions client.

Les API de base dans ConnectionManager:

`registerConnection` : Ajoute une nouvelle connexion à la liste de connexions existante

`closeConnection` : ferme la connexion à partir de l'événement déclenché par le client ou le serveur

`broadcastMessage` : Parfois, vous devez envoyer un message à toutes les connexions client abonnées.

Je ne fournis pas une implémentation complète du code source, car l'exemple deviendra très long. Au plus haut niveau, le code sera comme ça.

```
import java.util.*;
```

```
import java.net.*;

/* Lazy Singleton - Thread Safe Singleton without synchronization and volatile constructs */
final class  LazyConnectionManager {
    private Map<String,Connection> connections = new HashMap<String,Connection>();
    private LazyConnectionManager() {}
    public static LazyConnectionManager getInstance() {
        return LazyHolder.INSTANCE;
    }
    private static class LazyHolder {
        private static final LazyConnectionManager INSTANCE = new LazyConnectionManager();
    }

    /* Make sure that De-Serailzation does not create a new instance */
    private Object readResolve()  {
        return LazyHolder.INSTANCE;
    }
    public void registerConnection(Connection connection){
        /* Add new connection to list of existing connection */
        connections.put(connection.getConnectionId(),connection);
    }
    public void closeConnection(String connectionId){
        /* Close connection and remove from map */
        Connection connection = connections.get(connectionId);
        if ( connection != null) {
            connection.close();
            connections.remove(connectionId);
        }
    }
    public void broadcastMessage(String message){
        for (Map.Entry<String, Connection> entry : connections.entrySet()){
            entry.getValue().sendMessage(message);
        }
    }
}
```

Classe de serveur exemple:

```
class Server implements Runnable{
    ServerSocket socket;
    int id;
    public Server(){
        new Thread(this).start();
    }
    public void run(){
        try{
            ServerSocket socket = new ServerSocket(4567);
            while(true){
                Socket clientSocket = socket.accept();
                ++id;
                Connection connection = new Connection(""+ id,clientSocket);
                LazyConnectionManager.getInstance().registerConnection(connection);
                LazyConnectionManager.getInstance().broadcastMessage("Message pushed by
server:");
            }
        }catch(Exception err){
            err.printStackTrace();
        }
    }
```

```
}
```

Autres cas pratiques d'utilisation de Singletons:

1. Gestion de ressources globales comme `ThreadPool, ObjectPool, DatabaseConnectionPool` etc.
2. Des services centralisés tels que la `Logging` des données d'application avec différents niveaux de journalisation tels que `DEBUG,INFO,WARN,ERROR` etc.
3. Global `RegistryService` où différents services sont enregistrés avec un composant central au démarrage. Ce service global peut servir de `Facade` pour l'application

C # Exemple: Singleton multithread

L'initialisation statique convient à la plupart des situations. Lorsque votre application doit retarder l'instanciation, utiliser un constructeur autre que celui par défaut ou effectuer d'autres tâches avant l'instanciation et travailler dans un environnement multithread, vous avez besoin d'une solution différente. Il existe toutefois des cas dans lesquels vous ne pouvez pas compter sur le Common Language Runtime pour garantir la sécurité des threads, comme dans l'exemple d'initialisation statique. Dans de tels cas, vous devez utiliser des fonctionnalités linguistiques spécifiques pour vous assurer qu'une seule instance de l'objet est créée en présence de plusieurs threads. L'une des solutions les plus courantes consiste à utiliser l'idiome Double-Check Locking [Lea99] pour empêcher les threads distincts de créer de nouvelles instances du singleton en même temps.

L'implémentation suivante permet à un seul thread d'entrer dans la zone critique, identifiée par le bloc de verrouillage, alors qu'aucune instance de Singleton n'a encore été créée:

```
using System;

public sealed class Singleton {
   private static volatile Singleton instance;
   private static object syncRoot = new Object();

   private Singleton() {}

   public static Singleton Instance    {
      get
      {
         if (instance == null)
         {
            lock (syncRoot)
            {
               if (instance == null)
                  instance = new Singleton();
            }
         }

         return instance;
      }
  }
}
```

Cette approche garantit qu'une seule instance est créée et uniquement lorsque l'instance est nécessaire. De plus, la variable est déclarée volatile pour garantir que l'affectation à la variable

d'instance se termine avant que la variable d'instance soit accessible. Enfin, cette approche utilise une instance syncRoot pour verrouiller, plutôt que de verrouiller le type lui-même, pour éviter les blocages.

Cette approche de verrouillage par double vérification résout les problèmes de concurrence de threads tout en évitant un verrou exclusif dans chaque appel à la méthode de propriété Instance. Il vous permet également de retarder l'instanciation jusqu'à ce que l'objet soit d'abord accédé. En pratique, une application nécessite rarement ce type d'implémentation. Dans la plupart des cas, l'approche d'initialisation statique est suffisante.

Référence: MSDN

Remerciements

[Gamma95] Gamma, Helm, Johnson et Vlissides. Modèles de conception: éléments du logiciel orienté objet réutilisable. Addison-Wesley, 1995.

[Lea99] Lea, Doug. Programmation concurrente en Java, deuxième édition. Addison-Wesley, 1999.

[Sells03] Vend, Chris. "Sucks Suck." sellsbrothers.com Nouvelles. Disponible à l' adresse : http://www.sellsbrothers.com/news/showTopic.aspx?ixTopic=411 .

Singleton (PHP)

Exemple de phptherightway.com

```
<?php
class Singleton
{
    /**
     * @var Singleton The reference to *Singleton* instance of this class
     */
    private static $instance;

    /**
     * Returns the *Singleton* instance of this class.
     *
     * @return Singleton The *Singleton* instance.
     */
    public static function getInstance()
    {
        if (null === static::$instance) {
            static::$instance = new static();
        }

        return static::$instance;
    }

    /**
     * Protected constructor to prevent creating a new instance of the
     * *Singleton* via the `new` operator from outside of this class.
     */
    protected function __construct()
```

```
    {
    }

    /**
     * Private clone method to prevent cloning of the instance of the
     * *Singleton* instance.
     *
     * @return void
     */
    private function __clone()
    {
    }

    /**
     * Private unserialize method to prevent unserializing of the *Singleton*
     * instance.
     *
     * @return void
     */
    private function __wakeup()
    {
    }
}

class SingletonChild extends Singleton
{
}

$obj = Singleton::getInstance();
var_dump($obj === Singleton::getInstance());             // bool(true)

$anotherObj = SingletonChild::getInstance();
var_dump($anotherObj === Singleton::getInstance());      // bool(false)

var_dump($anotherObj === SingletonChild::getInstance()); // bool(true)
```

Singleton Design pattern (en général)

Remarque: le singleton est un motif de conception.
Mais il a également considéré un anti-modèle.

L'utilisation d'un singleton doit être considérée avec soin avant utilisation. Il existe généralement de meilleures alternatives.

Le principal problème avec un singleton est le même que le problème avec les variables globales. Ils introduisent un état mutable global externe. Cela signifie que les fonctions qui utilisent un singleton ne dépendent pas uniquement des paramètres d'entrée, mais également de l'état du singleton. Cela signifie que les tests peuvent être gravement compromis (difficile).

Les problèmes avec les singletons peuvent être atténués en les utilisant conjointement avec les modèles de création; afin que la création initiale du singleton puisse être contrôlée.

Lire Singleton en ligne: https://riptutorial.com/fr/design-patterns/topic/2179/singleton

Chapitre 30: SOLIDE

Introduction

Qu'est ce que SOLID?

SOLID est un acronyme mnémonique (aide mémoire). Les principes de Solid devraient aider les développeurs de logiciels à éviter les «odeurs de code» et devraient déboucher sur un bon code source. Un bon code source signifie dans ce contexte que le code source est facile à étendre et à maintenir. Les principes de Solid sont centrés sur les classes

Quoi attendre:

Pourquoi devriez-vous appliquer SOLID

Comment appliquer les cinq principes SOLID (exemples)

Examples

SRP - Principe de responsabilité unique

Le S dans SOLID signifie principe de responsabilité unique (SRP).

La responsabilité signifie dans ce contexte des raisons de changer, de sorte que le principe stipule qu'une classe ne devrait avoir qu'une seule raison de changer.

Robert C. Martin l'a déclaré (lors de sa conférence à Yale shool of management en 10 septembre 2014) comme suit

> Vous pourriez également dire, ne mettez pas les fonctions qui changent pour différentes raisons dans la même classe.

ou

> Ne mélangez pas les préoccupations dans vos cours

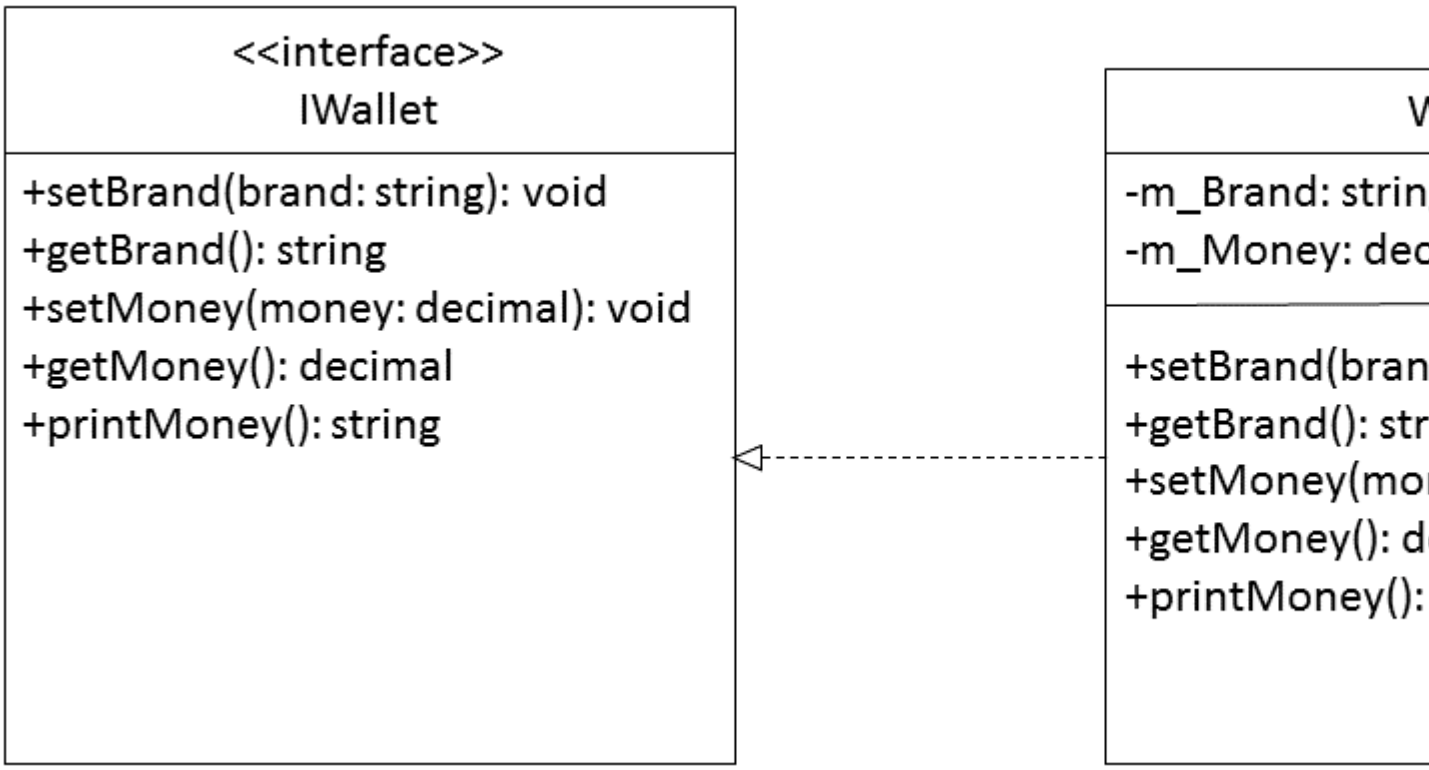
Raison d'appliquer le SRP:

Lorsque vous modifiez une classe, vous pouvez affecter les fonctionnalités liées aux autres responsabilités de la classe. Garder les responsabilités à un faible niveau minimise le risque d'effets secondaires.

Mauvais exemple

Nous avons une interface IWallet et une classe Wallet qui implémente l'IWallet. Le porte-monnaie détient notre argent et la marque, en outre devrait-il imprimer notre argent en tant que représentation de chaîne. La classe est utilisée par

1. un webservice
2. un rédacteur qui imprime l'argent en euros dans un fichier texte.



Le SRP est violé ici parce que nous avons deux préoccupations:

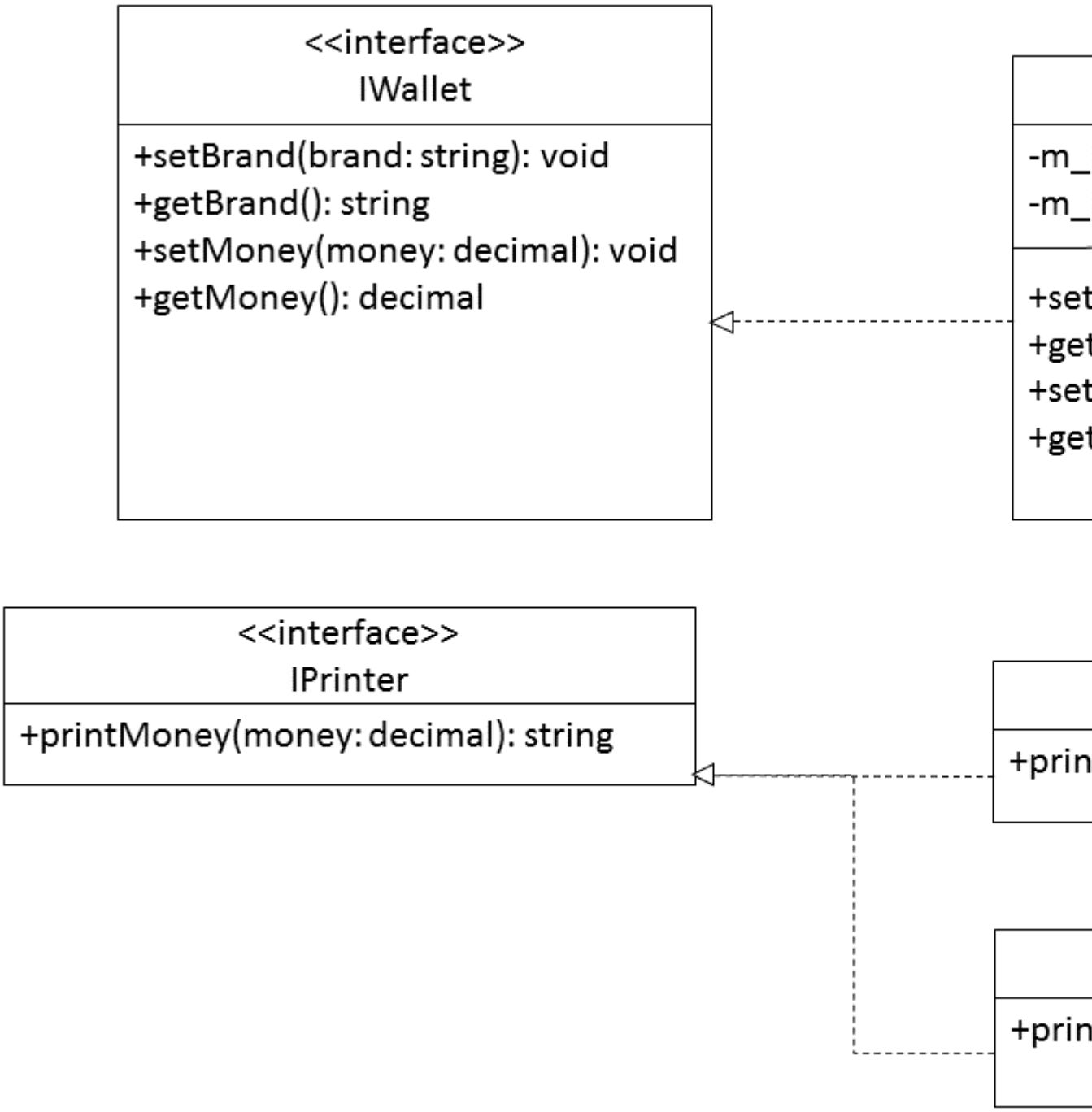
1. Le stockage de l'argent et de la marque
2. La représentation de l'argent.

C # exemple de code

```
public interface IWallet
{
    void setBrand(string brand);
    string getBrand();
    void setMoney(decimal money);
    decimal getMoney();
    string printMoney();
}

public class Wallet : IWallet
{
    private decimal m_Money;
    private string m_Brand;

    public string getBrand()
    {
        return m_Brand;
    }

    public decimal getMoney()
    {
```

```
        return m_Money;
    }

    public void setBrand(string brand)
    {
        m_Brand = brand;
    }

    public void setMoney(decimal money)
    {
        m_Money = money;
    }

    public string printMoney()
    {
        return m_Money.ToString();
    }
}
```

Bon exemple



Pour éviter la violation du SRP, nous avons supprimé la méthode `printMoney` de la classe Wallet et l' `printMoney` placée dans une classe Printer. La classe Printer est maintenant responsable de l'impression et le portefeuille est maintenant responsable du stockage des valeurs.

C # exemple de code

```
public interface IPrinter
{
    void printMoney(decimal money);
}

public class EuroPrinter : IPrinter
{
    public void printMoney(decimal money)
```

```
    {
        //print euro
    }
}

public class DollarPrinter : IPrinter
{
    public void printMoney(decimal money)
    {
        //print Dollar
    }
}

public interface IWallet
{
    void setBrand(string brand);
    string getBrand();
    void setMoney(decimal money);
    decimal getMoney();
}

public class Wallet : IWallet
{
    private decimal m_Money;
    private string m_Brand;

    public string getBrand()
    {
        return m_Brand;
    }

    public decimal getMoney()
    {
        return m_Money;
    }

    public void setBrand(string brand)
    {
        m_Brand = brand;
    }

    public void setMoney(decimal money)
    {
        m_Money = money;
    }
}
```

Lire SOLIDE en ligne: https://riptutorial.com/fr/design-patterns/topic/8651/solide

Chapitre 31: tableau noir

Examples

Echantillon C

Blackboard.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;


namespace Blackboard
{
    public class BlackBoard
    {
        public List<KnowledgeWorker> knowledgeWorkers;
        protected Dictionary<string, ControlData> data;
        public Control control;


        public BlackBoard()
        {
            this.knowledgeWorkers = new List<KnowledgeWorker>();
            this.control = new Control(this);
            this.data = new Dictionary<string, ControlData>();
        }

        public void addKnowledgeWorker(KnowledgeWorker newKnowledgeWorker)
        {
            newKnowledgeWorker.blackboard = this;
            this.knowledgeWorkers.Add(newKnowledgeWorker);
        }

        public Dictionary<string, ControlData> inspect()
        {
            return (Dictionary<string, ControlData>) this.data.ToDictionary(k => k.Key, k =>
(ControlData) k.Value.Clone());
        }
        public void update(KeyValuePair<string, ControlData> blackboardEntry)
        {
            if (this.data.ContainsKey(blackboardEntry.Key))
            {
                this.data[blackboardEntry.Key] = blackboardEntry.Value;
            }
            else
                throw new InvalidOperationException(blackboardEntry.Key + " Not Found!");
        }

        public void update(string key, ControlData data)
        {
            if (this.data.ContainsKey(key))
            {
```

```
                this.data[key] = data;
            }
            else
            {
                this.data.Add(key, data);
            }
        }

        public void print()
        {
            System.Console.WriteLine("Blackboard state");
            foreach (KeyValuePair<string, ControlData> cdata in this.data)
            {
                Console.WriteLine(string.Format("data:{0}", cdata.Key));
                Console.WriteLine(string.Format("\tProblem:{0}", cdata.Value.problem));
                if(cdata.Value.input!=null)
                    Console.WriteLine(string.Format("\tInput:{0}",
string.Join(",",cdata.Value.input)));
                if(cdata.Value.output!=null)
                    Console.WriteLine(string.Format("\tOutput:{0}",
string.Join(",",cdata.Value.output)));
            }
        }

    }
}
```

Control.cs

```
using System;
using System.Collections.Generic;


namespace Blackboard
{
    public class Control
    {
        BlackBoard blackBoard = null;

        public Control(BlackBoard blackBoard)
        {
            this.blackBoard = blackBoard;
        }

        public void loop()
        {
            System.Console.WriteLine("Starting loop");
            if (blackBoard == null)
                throw new InvalidOperationException("blackboard is null");
            this.nextSource();
            System.Console.WriteLine("Loop ended");

        }

        /// <summary>
        /// Selects the next source of knowledge (knowledgeworker by inspecting the
blackgoard)
        /// </summary>
        void nextSource()
```

```
        {
            // observers the blackboard
            foreach (KeyValuePair<string, ControlData> value in this.blackBoard.inspect())
            {
                if (value.Value.problem == "PrimeNumbers")
                {
                    foreach (KnowledgeWorker worker in this.blackBoard.knowledgeWorkers)
                    {
                        if (worker.getName() == "PrimeFinder")
                        {
                            Console.WriteLine("Knowledge Worker Found");
                            worker.executeCondition();
                            worker.executeAction();
                            worker.updateBlackboard();
                        }
                    }
                }
            }
        }
    }
}
```

ControlData.cs

```
using System;
using System.Collections.Generic;


namespace Blackboard
{
    public class ControlData:ICloneable
    {
        public string problem;
        public object[] input;
        public object[] output;
        public string updateby;
        public DateTime updated;


        public ControlData()
        {
            this.problem = null;
            this.input = this.output = null;
        }

        public ControlData(string problem, object[] input)
        {
            this.problem = problem;
            this.input = input;
            this.updated = DateTime.Now;
        }

        public object getResult()
        {
            return this.output;
        }

        public object Clone()
        {
```

```
            ControlData clone;
            clone = new ControlData(this.problem, this.input);
            clone.updated = this.updated;
            clone.updateby = this.updateby;
            clone.output = this.output;
            return clone;
        }
    }
}
```

KnowledgeWorker.cs

```
using System; using System.Collections.Generic;

namespace Blackboard {
    /// <summary>
    /// each knowledgeworker is resposible for knowing the conditions under which it can
contribute to a solution.
    /// </summary>
    abstract public  class KnowledgeWorker
    {
        protected Boolean canContribute;
        protected string Name;
        public BlackBoard blackboard = null;
        protected List<KeyValuePair<string, ControlData>> keys;
        public KnowledgeWorker(BlackBoard blackboard, String Name)
        {
            this.blackboard = blackboard;
            this.Name = Name;
        }

        public KnowledgeWorker(String Name)
        {
            this.Name = Name;
        }

        public string getName()
        {
            return this.Name;
        }

        abstract public void executeAction();

        abstract public void executeCondition();

        abstract public void updateBlackboard();

    } }
```

Lire tableau noir en ligne: https://riptutorial.com/fr/design-patterns/topic/6519/tableau-noir

Chapitre 32: Usine

Remarques

> Fournir une interface pour créer des familles d'objets associés ou dépendants sans spécifier leurs classes concrètes.
>
> *- GOF 1994*

Examples

Usine simple (Java)

Une fabrique diminue le couplage entre le code qui doit créer des objets à partir du code de création d'objet. La création d'objet n'est pas faite explicitement en appelant un constructeur de classe, mais en appelant une fonction qui crée l'objet au nom de l'appelant. Un exemple Java simple est le suivant:

```
interface Car {
}

public class CarFactory{
  static public Car create(String s) {
    switch (s) {
    default:
    case "us":
    case "american": return new Chrysler();
    case "de":
    case "german": return new Mercedes();
    case "jp":
    case "japanese": return new Mazda();
    }
  }
}

class Chrysler implements Car {
  public String toString() { return "Chrysler"; }
}

class Mazda implements Car {
  public String toString() { return "Mazda"; }
}

class Mercedes implements Car {
  public String toString() { return "Mercedes"; }
}

public class CarEx {
  public static void main(String args[]) {
    Car car = CarFactory.create("us");
    System.out.println(car);
  }
}
```

Dans cet exemple, l'utilisateur donne un aperçu de ce dont il a besoin et l'usine est libre de construire quelque chose de approprié. C'est une **inversion de dépendance** : l'implémenteur de `Car` concept est libre de renvoyer une `Car` concrète appropriée demandée par l'utilisateur qui à son tour ne connaît pas les détails de l'objet concret construit.

Ceci est un exemple simple de fonctionnement de l'usine, bien sûr, dans cet exemple, il est toujours possible d'instancier des classes concrètes; mais on peut l'empêcher en cachant des classes de béton dans un paquet, de telle sorte que l'utilisateur est obligé d'utiliser l'usine.

.Net Fiddle pour l'exemple ci-dessus.

Fabrique abstraite (C ++)

Le motif d' **usine abstrait** fournit un moyen d'obtenir une collection cohérente d'objets à travers un ensemble de fonctions de fabriques. Comme pour tout modèle, le couplage est réduit en faisant abstraction de la manière dont un ensemble d'objets est créé, de sorte que le code utilisateur ignore les nombreux détails des objets dont il a besoin.

L'exemple C ++ suivant illustre comment obtenir différents types d'objets de la même famille d'interface graphique (hypothétique):

```
#include <iostream>

/* Abstract definitions */
class GUIComponent {
public:
  virtual ~GUIComponent() = default;
  virtual void draw() const = 0;
};
class Frame  : public GUIComponent {};
class Button : public GUIComponent {};
class Label  : public GUIComponent {};

class GUIFactory {
public:
  virtual ~GUIFactory() = default;
  virtual std::unique_ptr<Frame> createFrame() = 0;
  virtual std::unique_ptr<Button> createButton() = 0;
  virtual std::unique_ptr<Label> createLabel() = 0;
  static std::unique_ptr<GUIFactory> create(const std::string& type);
};

/* Windows support */
class WindowsFactory : public GUIFactory {
private:
    class WindowsFrame : public Frame {
    public:
      void draw() const override { std::cout << "I'm a Windows-like frame" << std::endl; }
    };
    class WindowsButton : public Button {
    public:
      void draw() const override { std::cout << "I'm a Windows-like button" << std::endl; }
    };
    class WindowsLabel : public Label {
    public:
```

```
      void draw() const override { std::cout << "I'm a Windows-like label" << std::endl; }
    };
public:
  std::unique_ptr<Frame> createFrame() override { return std::make_unique<WindowsFrame>(); }
  std::unique_ptr<Button> createButton() override { return std::make_unique<WindowsButton>();
}
  std::unique_ptr<Label> createLabel() override { return std::make_unique<WindowsLabel>(); }
};

/* Linux support */
class LinuxFactory : public GUIFactory {
private:
    class LinuxFrame : public Frame {
    public:
      void draw() const override { std::cout << "I'm a Linux-like frame" << std::endl; }
    };
    class LinuxButton : public Button {
    public:
      void draw() const override { std::cout << "I'm a Linux-like button" << std::endl; }
    };
    class LinuxLabel : public Label {
    public:
      void draw() const override { std::cout << "I'm a Linux-like label" << std::endl; }
    };
public:
  std::unique_ptr<Frame> createFrame() override { return std::make_unique<LinuxFrame>(); }
  std::unique_ptr<Button> createButton() override { return std::make_unique<LinuxButton>(); }
  std::unique_ptr<Label> createLabel() override { return std::make_unique<LinuxLabel>(); }
};

std::unique_ptr<GUIFactory> GUIFactory::create(const string& type) {
  if (type == "windows") return std::make_unique<WindowsFactory>();
  return std::make_unique<LinuxFactory>();
}

/* User code */
void buildInterface(GUIFactory& factory) {
  auto frame = factory.createFrame();
  auto button = factory.createButton();
  auto label = factory.createLabel();

  frame->draw();
  button->draw();
  label->draw();
}

int main(int argc, char *argv[]) {
  if (argc < 2) return 1;
  auto guiFactory = GUIFactory::create(argv[1]);
  buildInterface(*guiFactory);
}
```

Si l'exécutable généré s'appelle `abstractfactory` alors la sortie peut donner:

```
$ ./abstractfactory windows
I'm a Windows-like frame
I'm a Windows-like button
I'm a Windows-like label
$ ./abstractfactory linux
I'm a Linux-like frame
```

```
I'm a Linux-like button
I'm a Linux-like label
```

Exemple simple de Factory qui utilise un IoC (C #)

Les usines peuvent également être utilisées avec les bibliothèques Inversion of Control (IoC).

- Le cas d'utilisation typique d'une telle fabrique est le moment où nous voulons créer un objet en fonction de paramètres inconnus avant l'exécution (comme l'utilisateur actuel).
- Dans ces cas, il peut être parfois difficile (voire impossible) de configurer la bibliothèque IoC seule pour gérer ce type d'informations contextuelles à l'exécution, afin que nous puissions l'envelopper dans une usine.

Exemple

- Supposons que nous ayons une classe `User` , dont les caractéristiques (ID, niveau d'autorisation de sécurité, etc.) sont inconnues avant l'exécution (l'utilisateur actuel pouvant être toute personne utilisant l'application).
- Nous devons prendre l'utilisateur actuel et obtenir un `ISecurityToken` pour eux, qui peut ensuite être utilisé pour vérifier si l'utilisateur est autorisé à effectuer certaines actions ou non.
- L'implémentation d'ISecurityToken variera en fonction du niveau de l'utilisateur - en d'autres termes, ISecurityToken utilise un *polymorphisme* .

Dans ce cas, nous avons deux implémentations, qui utilisent également des *interfaces de marqueur* pour faciliter leur identification dans la bibliothèque IoC; la bibliothèque IoC dans ce cas est juste composée et identifiée par l'abstraction `IContainer` .

> Notez également que de nombreuses usines IoC modernes ont des capacités ou des plug-ins natifs qui permettent la création automatique de fabriques, tout en évitant les interfaces de marqueur, comme indiqué ci-dessous; Cependant, comme tous ne le font pas, cet exemple répond à un concept de fonctionnalité commun simple et minimal.

```
//describes the ability to allow or deny an action based on PerformAction.SecurityLevel
public interface ISecurityToken
{
    public bool IsAllowedTo(PerformAction action);
}

//Marker interface for Basic permissions
public interface IBasicToken:ISecurityToken{};
//Marker interface for super permissions
public interface ISuperToken:ISecurityToken{};

//since IBasictoken inherits ISecurityToken, BasicToken can be treated as an ISecurityToken
public class BasicToken:IBasicToken
{
     public bool IsAllowedTo(PerformAction action)
     {
         //Basic users can only perform basic actions
         if(action.SecurityLevel!=SecurityLevel.Basic) return false;
         return true;
```

```
     }
}

public class SuperToken:ISuperToken
{
     public bool IsAllowedTo(PerformAction action)
     {
         //Super users can perform all actions
         return true;
     }
}
```

Ensuite, nous allons créer une usine `SecurityToken` , qui prendra comme dépendance notre `IContainer`

```
public class SecurityTokenFactory
{
   readonly IContainer _container;
   public SecurityTokenFactory(IContainer container)
   {
      if(container==null) throw new ArgumentNullException("container");
   }

   public ISecurityToken GetToken(User user)
   {
      if (user==null) throw new ArgumentNullException("user);
      //depending on the user security level, we return a different type; however all types
implement ISecurityToken so the factory can produce them.
      switch user.SecurityLevel
      {
          case Basic:
           return _container.GetInstance<BasicSecurityToken>();
          case SuperUser:
           return _container.GetInstance<SuperUserToken>();
      }
   }
}
```

Une fois que nous les avons enregistrés avec `IContainer` :

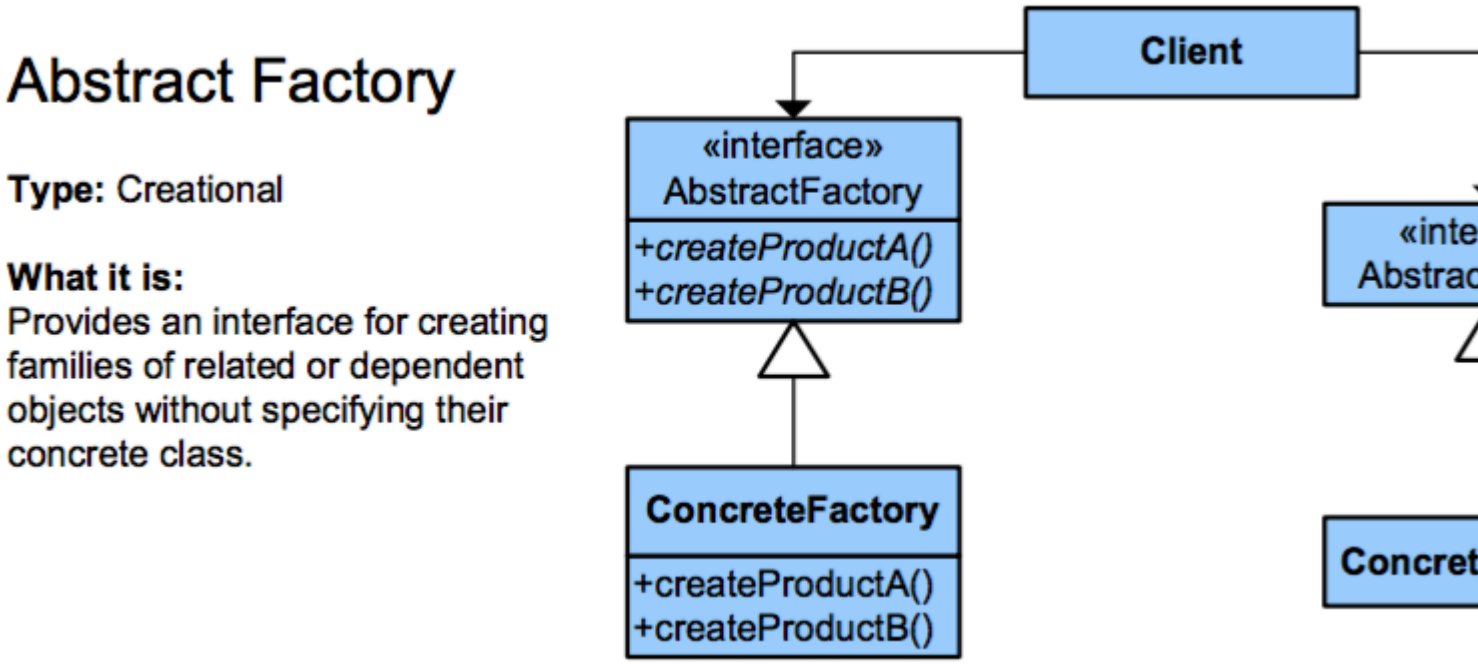
```
IContainer.For<SecurityTokenFactory>().Use<SecurityTokenFactory>().Singleton(); //we only need
a single instance per app
IContainer.For<IBasicToken>().Use<BasicToken>().PerRequest(); //we need an instance per-
request
IContainer.For<ISuperToken>().Use<SuperToken>().PerRequest();//we need an instance per-request
```

le code consommateur peut l'utiliser pour obtenir le bon jeton à l'exécution:

```
readonly SecurityTokenFactory _tokenFactory;
...
...
public void LogIn(User user)
{
    var token = _tokenFactory.GetToken(user);
    user.SetSecurityToken(token);
}
```

De cette manière, nous bénéficions de l'encapsulation fournie par l'usine et de la gestion du cycle de vie fournie par la bibliothèque IoC.

Une fabrique abstraite



Le motif de conception suivant est classé comme motif de création.

Une fabrique abstraite est utilisée pour fournir une interface pour créer des familles d'objets associés, sans spécifier de classes concrètes et peut être utilisée pour masquer des classes spécifiques à une plate-forme.

```
interface Tool {
    void use();
}

interface ToolFactory {
    Tool create();
}

class GardenTool implements Tool {

    @Override
    public void use() {
         // Do something...
    }
}

class GardenToolFactory implements ToolFactory {

    @Override
    public Tool create() {
        // Maybe additional logic to setup...
        return new GardenTool();
    }
}
```

```
class FarmTool implements Tool {

    @Override
    public void use() {
        // Do something...
    }
}

class FarmToolFactory implements ToolFactory {

    @Override
    public Tool create() {
        // Maybe additional logic to setup...
        return new FarmTool();
    }
}
```

Ensuite, on utilisera un fournisseur / producteur qui transmettra des informations lui permettant de restituer le type correct d'implémentation d'usine:

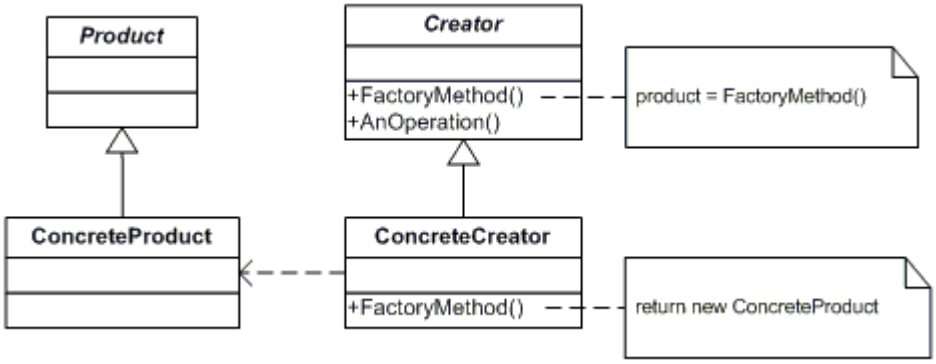
```
public final class FactorySupplier {

    // The supported types it can give you...
    public enum Type {
        FARM, GARDEN
    };

    private FactorySupplier() throws IllegalAccessException {
        throw new IllegalAccessException("Cannot be instantiated");
    }

    public static ToolFactory getFactory(Type type) {

        ToolFactory factory = null;

        switch (type) {
        case FARM:
            factory = new FarmToolFactory();
            break;
        case GARDEN:
            factory = new GardenToolFactory();
            break;
        } // Could potentially add a default case to handle someone passing in null

        return factory;
    }
}
```

Exemple d'usine en implémentant la méthode Factory (Java)

Intention:

Définissez une interface pour créer un objet, mais laissez les sous-classes décider quelle classe instancier. La méthode d'usine permet à une classe de différer l'instanciation aux sous-classes.

Diagramme UML:



Produit: définit une interface des objets créés par la méthode Factory.

ConcreteProduct: Implémente l'interface produit

Créateur: déclare la méthode Factory

ConcreateCreator: Implémente la méthode Factory pour retourner une instance d'un ConcreteProduct

Énoncé du problème: Créez une fabrique de jeux en utilisant les méthodes d'usine, qui définissent l'interface de jeu.

Extrait de code:

```
import java.util.HashMap;


/* Product interface as per UML diagram */
interface Game{
    /* createGame is a complex method, which executes a sequence of game steps */
    public void createGame();
}

/* ConcreteProduct implementation as per UML diagram */
class Chess implements Game{
    public Chess(){
        createGame();
    }
    public void createGame(){
        System.out.println("--------------------------------------");
        System.out.println("Create Chess game");
        System.out.println("Opponents:2");
        System.out.println("Define 64 blocks");
        System.out.println("Place 16 pieces for White opponent");
        System.out.println("Place 16 pieces for Black opponent");
        System.out.println("Start Chess game");
        System.out.println("--------------------------------------");
    }
}
class Checkers implements Game{
    public Checkers(){
        createGame();
    }
    public void createGame(){
        System.out.println("--------------------------------------");
        System.out.println("Create Checkers game");
```

```
        System.out.println("Opponents:2 or 3 or 4 or 6");
        System.out.println("For each opponent, place 10 coins");
        System.out.println("Start Checkers game");
        System.out.println("----------------------------------------");
    }
}
class Ludo implements Game{
    public Ludo(){
        createGame();
    }
    public void createGame(){
        System.out.println("----------------------------------------");
        System.out.println("Create Ludo game");
        System.out.println("Opponents:2 or 3 or 4");
        System.out.println("For each opponent, place 4 coins");
        System.out.println("Create two dices with numbers from 1-6");
        System.out.println("Start Ludo game");
        System.out.println("----------------------------------------");
    }
}

/* Creator interface as per UML diagram */
interface IGameFactory {
    public Game getGame(String gameName);
}

/* ConcreteCreator implementation as per UML diagram */
class GameFactory implements IGameFactory {

    HashMap<String,Game> games = new HashMap<String,Game>();
    /*
        Since Game Creation is complex process, we don't want to create game using new
operator every time.
        Instead we create Game only once and store it in Factory. When client request a
specific game,
        Game object is returned from Factory instead of creating new Game on the fly, which is
time consuming
    */

    public GameFactory(){

        games.put(Chess.class.getName(),new Chess());
        games.put(Checkers.class.getName(),new Checkers());
        games.put(Ludo.class.getName(),new Ludo());
    }
    public Game getGame(String gameName){
        return games.get(gameName);
    }
}

public class NonStaticFactoryDemo{
    public static void main(String args[]){
        if ( args.length < 1){
            System.out.println("Usage: java FactoryDemo gameName");
            return;
        }

        GameFactory factory = new GameFactory();
        Game game = factory.getGame(args[0]);
        System.out.println("Game="+game.getClass().getName());
    }
```

```
}
```

sortie:

```
java NonStaticFactoryDemo Chess
---------------------------------------
Create Chess game
Opponents:2
Define 64 blocks
Place 16 pieces for White opponent
Place 16 pieces for Black opponent
Start Chess game
---------------------------------------
---------------------------------------
Create Checkers game
Opponents:2 or 3 or 4 or 6
For each opponent, place 10 coins
Start Checkers game
---------------------------------------
---------------------------------------
Create Ludo game
Opponents:2 or 3 or 4
For each opponent, place 4 coins
Create two dices with numbers from 1-6
Start Ludo game
---------------------------------------
Game=Chess
```

Cet exemple montre une classe `Factory` en implémentant une `FactoryMethod` .

1. `Game` est l'interface pour tous les types de jeux. Il définit la méthode complexe: `createGame()`
2. `Chess, Ludo, Checkers` sont différentes variantes de jeux, qui implémentent `createGame()`
3. `public Game getGame(String gameName)` est `FactoryMethod` dans la classe `IGameFactory`
4. `GameFactory` pré-crée différents types de jeux dans le constructeur. Il implémente la méthode de fabrication `IGameFactory` .
5. game Name est passé en argument de ligne de commande à `NotStaticFactoryDemo`
6. `getGame` dans `GameFactory` accepte un nom de jeu et renvoie l'objet de `Game` correspondant.

Quand utiliser:

1. ***Factory*** : lorsque vous ne souhaitez pas exposer la logique d'instanciation d'objet au client / à l'appelant
2. ***Fabrique abstraite*** : lorsque vous souhaitez fournir une interface aux familles d'objets associés ou dépendants sans spécifier leurs classes concrètes
3. ***Méthode d'usine:*** Définir une interface pour créer un objet, mais laisser les sous-classes décider quelle classe instancier

Comparaison avec d'autres modèles créatifs:

1. Conception démarrée à l'aide de la **méthode d'usine** (moins compliquée, plus personnalisable, prolifération de sous-classes) et évolution vers la **fabrique abstraite, le prototype ou le générateur** (plus flexible, plus complexe) que le concepteur découvre

2. **Les** classes **Factory abstraites** sont souvent implémentées avec les **méthodes Factory** , mais elles peuvent aussi être implémentées avec **Prototype**

Références pour d'autres lectures: Modèles de conception de fabrication

Usine de poids mouche (C #)

En termes simples:

Une usine Flyweight qui, pour une clé donnée, déjà connue, donnera toujours le même objet que la réponse. Pour les nouvelles clés créera l'instance et la renverra.

En utilisant l'usine:

```
ISomeFactory<string, object> factory = new FlyweightFactory<string, object>();

var result1 = factory.GetSomeItem("string 1");
var result2 = factory.GetSomeItem("string 2");
var result3 = factory.GetSomeItem("string 1");

//Objects from different keys
bool shouldBeFalse = result1.Equals(result2);

//Objects from same key
bool shouldBeTrue = result1.Equals(result3);
```

La mise en oeuvre:

```
public interface ISomeFactory<TKey,TResult> where TResult : new()
{
    TResult GetSomeItem(TKey key);
}

public class FlyweightFactory<TKey, TResult> : ISomeFactory<TKey, TResult> where TResult :
new()
{
    public TResult GetSomeItem(TKey key)
    {
        TResult result;
        if(!Mapping.TryGetValue(key, out result))
        {
            result = new TResult();
            Mapping.Add(key, result);
        }
        return result;
    }

    public Dictionary<TKey, TResult> Mapping { get; set; } = new Dictionary<TKey, TResult>();
}
```

Notes supplémentaires

Je recommande d'ajouter à cette solution l'utilisation d'un `IoC Container` (comme expliqué dans un exemple différent ici) au lieu de créer vos propres nouvelles instances. On peut le faire en ajoutant un nouvel enregistrement pour le `TResult` au conteneur, puis en le résolvant (au lieu du `dictionary` dans l'exemple).

Méthode d'usine

Le modèle de méthode Factory est un modèle de création qui élimine la logique d'instanciation d'un objet afin de découpler le code client.

Lorsqu'une méthode de fabrique appartient à une classe qui est une implémentation d'un autre modèle de fabrique tel qu'une fabrique abstraite, il est généralement plus approprié de référencer le modèle implémenté par cette classe plutôt que le modèle de méthode Factory.

Le modèle de méthode Factory est plus communément utilisé pour décrire une méthode de fabrique appartenant à une classe qui n'est pas principalement une fabrique.

Par exemple, il peut être avantageux de placer une méthode de fabrique sur un objet qui représente un concept de domaine si cet objet encapsule un état qui simplifierait le processus de création d'un autre objet. Une méthode d'usine peut également conduire à une conception plus alignée sur la langue ubiquitaire d'un contexte spécifique.

Voici un exemple de code:

```
//Without a factory method
Comment comment = new Comment(authorId, postId, "This is a comment");

//With a factory method
Comment comment = post.comment(authorId, "This is a comment");
```

Lire Usine en ligne: https://riptutorial.com/fr/design-patterns/topic/1375/usine

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec les modèles de conception	Community, Ekin, Mateen Ulhaq, meJustAndrew, Sahan Serasinghe, Saurabh Sarode, Stephen C, Tim, ɬolɐəz əɥɬ qoq
2	Adaptateur	avojak, Ben Rhys-Lewis, Daniel Käfer, deHaar, Thijs Riezebeek
3	Chaîne de responsabilité	Ben Rhys-Lewis
4	chargement paresseux	Adhikari Bishwash
5	Façade	Kritner, Makoto, Ravindra babu
6	Injection de dépendance	Kritner, matiaslauriti, user2321864
7	Méthode d'usine statique	abbath, Bongo
8	Méthode du modèle	meJustAndrew, Ravindra babu
9	Modèle d'itérateur	bw_üezi, Dave Ranjan, Jeeter, Stephen C
10	Modèle d'objet nul	Jarod42, weston
11	Modèle de commande	matiaslauriti, Ravindra babu, Vasiliy Vlasov
12	Modèle de conception DAO (Data Access Object)	Pritam Banerjee
13	Modèle de médiateur	Ravindra babu, Vasiliy Vlasov
14	Modèle de pont	Mark, Ravindra babu, Vasiliy Vlasov
15	Modèle de stratégie	Aseem Bansal, dimitrisli, fabian, M.S. Dousti, Marek Skiba, matiaslauriti, Ravindra babu, Shog9, SjB, Stephen C, still_learning, uzilan
16	Monostate	skypjack

17	Motif composite	Krzysztof Branicki
18	Motif de constructeur	Alexey Groshev, Arif, Daniel Käfer, fgb, Kyle Morgan, Ravindra babu, Sujit Kamthe, uzilan, Vasiliy Vlasov, yitzih
19	Motif de décorateur	Arif, Krzysztof Branicki, matiaslauriti, Ravindra babu
20	Motif de visiteur	Daniel Käfer, Jarod42, Loki Astari, Ravindra babu, Stephen Leppik, Vasiliy Vlasov
21	Motif Prototype	Arif, Jarod42, user2321864
22	Multiton	Kid Binary
23	MVC, MVVM et MVP	Daniel LIn, Jompa234, Stephen C, user1223339
24	Observateur	Arif, user2321864, uzilan
25	Ouvrir le principe de fermeture	Mher Didaryan
26	Publier-S'abonner	Arif, Jeeter, Stephen C
27	Repository	bolt19, Leifb
28	Singleton	Bongo, didiz, DimaSan, Draken, fgb, Gul Md Ershad, hellyale, jefry jacky, Loki Astari, Marek Skiba, Mateen Ulhaq, matiaslauriti, Max, Panther, Prateek, RamenChef, Ravindra babu, S.L. Barth, Stephen C, Tazbir Bhuiyan, Tejus Prasad, Vasiliy Vlasov, volvis
29	SOLIDE	Bongo
30	tableau noir	Leonidas Menendez
31	Usine	Adil Abbasi, Daniel Käfer, Denis Elkhov, FireAlkazar, Geeky Ninja, Gilad Green, Jarod42, Jean-Baptiste Yunès, kstandell, Leifb, matiaslauriti, Michael Brown, Nijin22, plalx, Ravindra babu, Stephen Byrne, Tejas Pawar