EBook Gratis

APRENDIZAJE Design patterns

Free unaffiliated eBook created from
Stack Overflow contributors.

#design-patterns

Tabla de contenido

Acerca de

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

Capítulo 1: Comenzando con los patrones de diseño

Observaciones

Esta sección proporciona una descripción general de qué son los patrones de diseño y por qué un desarrollador puede querer usarlo. Los ejemplos pueden proporcionar una representación gráfica del patrón, un escenario que consiste en un problema dado un contexto en el que se puede usar un patrón y mencionar posibles compensaciones.

También debe mencionar cualquier tema grande dentro de los patrones de diseño y vincular a los temas relacionados. Dado que la Documentación para patrones de diseño es nueva, es posible que deba crear versiones iniciales de esos temas relacionados.

Examples

Introducción

Según Wikipedia :

> [A] *el patrón de diseño de software* es una solución general reutilizable para un problema común dentro de un contexto dado en el diseño de software. No es un diseño terminado que se pueda transformar directamente en código fuente o máquina. Es una descripción o plantilla sobre cómo resolver un problema que se puede utilizar en muchas situaciones diferentes. Los patrones de diseño se formalizan según las mejores prácticas que el programador puede usar para resolver problemas comunes al diseñar una aplicación o sistema.
>
> (Recuperado: 2016-10-13)

Hay muchos patrones de diseño de software reconocidos, y se proponen otros nuevos de forma regular. Otros temas cubren muchos de los patrones más comunes, y el artículo de Wikipedia proporciona una lista más extensa.

De manera similar, hay diferentes formas de clasificar los patrones de diseño, pero la clasificación original es:

- **Patrones de creación** : Fábrica , Constructor , Singleton , etc.
- **Patrones estructurales** : Adaptador , Compuesto , Proxy, etc.
- **Patrones de comportamiento** : iterador , estrategia , visitante , etc.
- **Patrones de concurrencia** : ActiveObject, Monitor, etc.

La idea de patrones de diseño se ha extendido a *patrones de diseño específicos de dominio* para dominios tales como diseño de interfaz de usuario, visualización de datos, diseño seguro, diseño web y diseño de modelo de negocio.

Finalmente, hay un concepto relacionado llamado *patrón de arquitectura de software* que se describe como el análogo de los patrones de diseño aplicados a las arquitecturas de software.

Lea Comenzando con los patrones de diseño en línea: https://riptutorial.com/es/design-patterns/topic/1012/comenzando-con-los-patrones-de-diseno

Capítulo 2: Adaptador

Examples

Patrón Adaptador (PHP)

Un ejemplo del mundo real que utiliza un experimento científico en el que se realizan ciertas rutinas en diferentes tipos de tejido. La clase contiene dos funciones por defecto para obtener el tejido o la rutina por separado. En una versión posterior, lo adaptamos utilizando una nueva clase para agregar una función que obtiene ambas. Esto significa que no hemos editado el código original y, por lo tanto, no corremos ningún riesgo de romper nuestra clase existente (y no volver a realizar la prueba).

```
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().')';
    }
}
```

Adaptador (Java)

Supongamos que en su base de código actual, existe `MyLogger` interfaz de `MyLogger` así:

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

Digamos que ha creado algunas implementaciones concretas de estos, como `MyFileLogger` y `MyConsoleLogger` .

Ha decidido que desea usar un marco para controlar la conectividad Bluetooth de su aplicación.

Este marco contiene un `BluetoothManager` con el siguiente constructor:

```
class BluetoothManager {
    private FrameworkLogger logger;

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

El `BluetoothManager` también acepta un registrador, ¡lo cual es genial! Sin embargo, espera un registrador cuya interfaz fue definida por el marco y han usado la sobrecarga de métodos en lugar de nombrar sus funciones de manera diferente:

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

Ya tiene un montón de implementaciones de `MyLogger` que le gustaría reutilizar, pero no se ajustan a la interfaz de `FrameworkLogger` . Aquí es donde entra el patrón de diseño del adaptador:

```
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);
    }
}
```

Al definir una clase de adaptador que implementa la interfaz de `FrameworkLogger` y acepta una implementación de `MyLogger` , la funcionalidad se puede asignar entre las diferentes interfaces. Ahora es posible usar el `MyLogger` `BluetoothManager` con todas las implementaciones de `MyLogger` como:

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

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

Ejemplo de Java

Un gran ejemplo existente del patrón del adaptador se puede encontrar en las clases SWT MouseListener y MouseAdapter .

La interfaz de MouseListener tiene el siguiente aspecto:

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

Ahora imagine un escenario en el que está creando una UI y agregando estos oyentes, pero la mayoría de las veces no le importa nada más que cuando se hace clic en algo (mouseUp). No querrías estar constantemente creando implementaciones vacías:

```
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
    }

});
```

En su lugar, podemos usar MouseAdapter:

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

Al proporcionar implementaciones vacías y predeterminadas, somos libres de anular solo aquellos métodos que nos interesan en el adaptador. Siguiendo el ejemplo anterior:

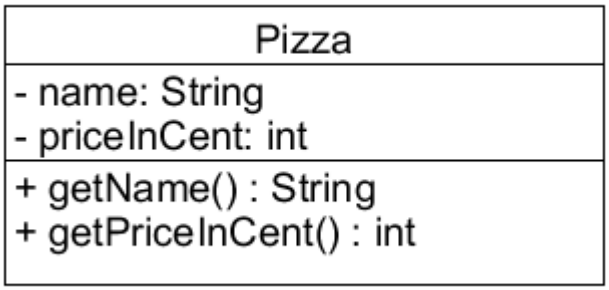
```
obj.addMouseListener(new MouseAdapter() {

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

});
```

Adaptador (UML y ejemplo de situación)

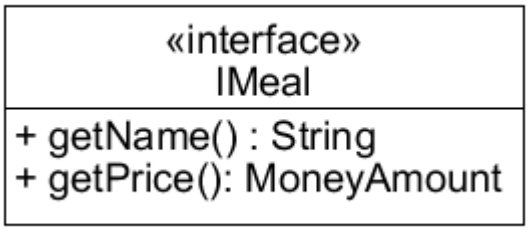
Para hacer un uso del patrón de adaptador y el tipo de situación en la que se puede aplicar más imaginable, aquí se ofrece un ejemplo pequeño, simple y muy concreto. No habrá ningún código aquí, solo UML y una descripción de la situación de ejemplo y su problema. Es cierto que el contenido UML está escrito como Java. (Bueno, el texto de la sugerencia decía "Los buenos ejemplos son principalmente código", creo que los patrones de diseño son lo suficientemente abstractos como para ser presentados de una manera diferente, también).

En general, el patrón de adaptador es una solución adecuada para una situación en la que tiene interfaces incompatibles y ninguna de ellas puede reescribirse directamente.

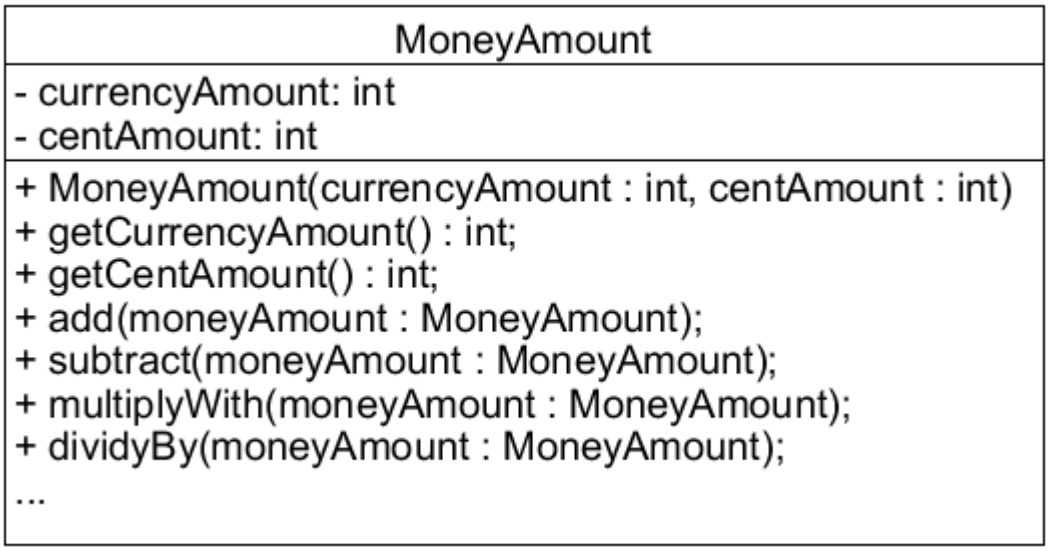
Imagina que tienes un buen servicio de reparto de pizzas. Los clientes pueden realizar pedidos en línea en su sitio web y usted tiene un sistema pequeño que utiliza una `Pizza` clase para representar sus pizzas y calcular facturas, informes de impuestos y más. El precio de sus pizzas se da como un solo entero que representa el precio en centavos (de la moneda de su elección).



Su servicio de entrega está funcionando muy bien, pero en algún momento ya no puede manejar el creciente número de clientes por su cuenta, pero aún desea expandirse. Decide agregar sus pizzas al menú de un gran servicio de entrega meta en línea. Ofrecen muchas comidas diferentes, no solo pizzas, por lo que su sistema hace un mayor uso de la abstracción y tiene una Interfaz `IMeal` representa las comidas junto con una clase de `MoneyAmount` representa el dinero.



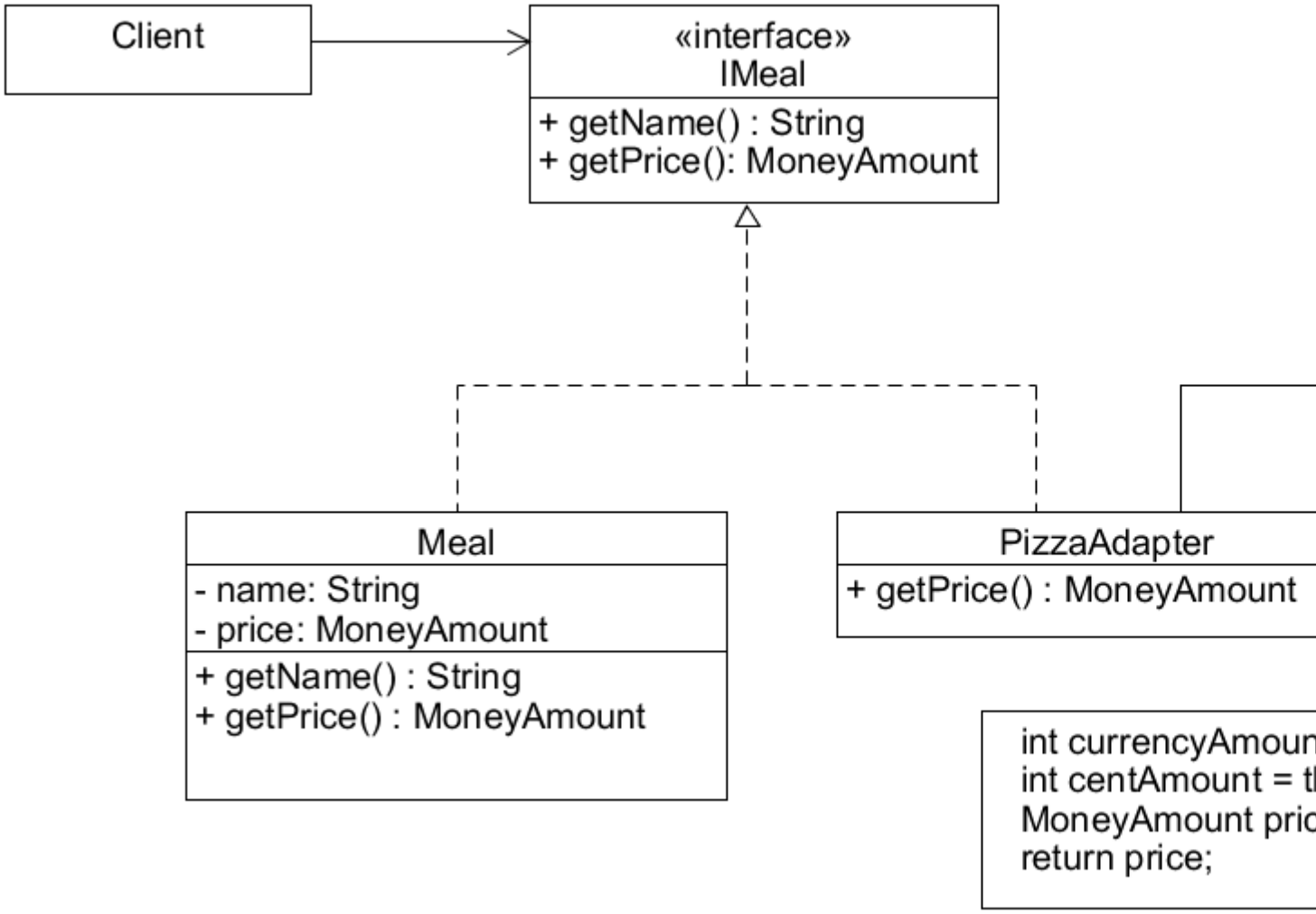
`MoneyAmount` consta de dos enteros como entrada, uno para la cantidad (o alguna moneda aleatoria) antes de la coma, y otro para la cantidad de centavos de 0 a 99 después de la coma;



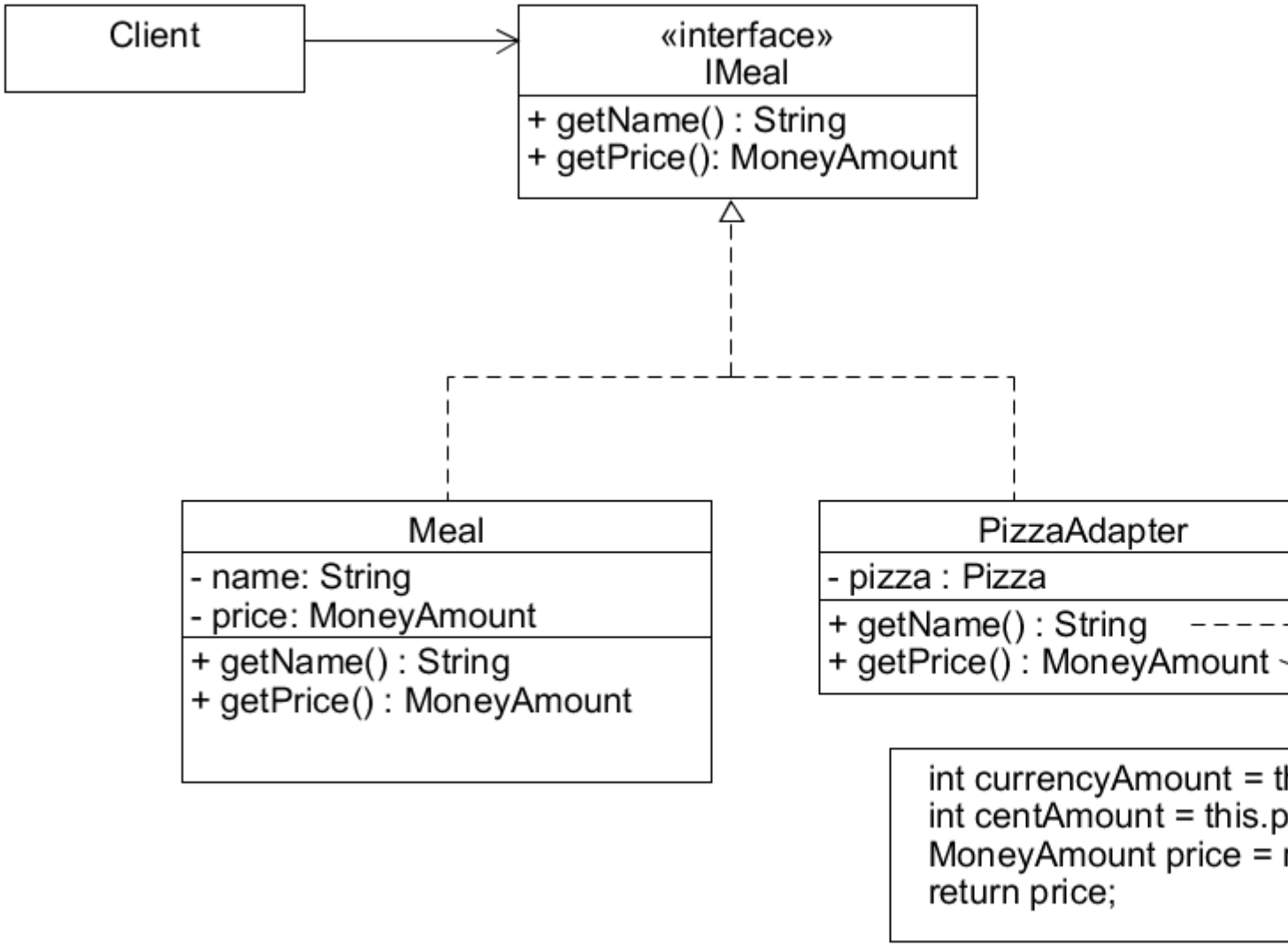
Debido al hecho de que el precio de su `Pizza` es un solo entero que representa el precio total como una cantidad de centavo (> 99), no es compatible con `IMeal` . Este es el punto en el que el patrón del adaptador entra en juego: en caso de que requiera demasiado esfuerzo cambiar su propio sistema o crear uno nuevo y tenga que implementar una interfaz incompatible, es posible que desee aplicar el patrón del adaptador.

Hay dos formas de aplicar el patrón: adaptador de clase y adaptador de objeto.

Ambos tienen en común que un adaptador (`PizzaAdapter`) funciona como algún tipo de traductor entre la nueva interfaz y el adaptee (`Pizza` en este ejemplo). El adaptador implementa la nueva interfaz (`IMeal`) y luego hereda de `Pizza` y convierte su propio precio de un entero a dos (adaptador de clase)



o tiene un objeto de tipo `Pizza` como atributo y convierte los valores de ese (adaptador de objeto).



Al aplicar el patrón de adaptador, usted podrá "traducir" entre interfaces incompatibles.

Lea Adaptador en línea: https://riptutorial.com/es/design-patterns/topic/4580/adaptador

Capítulo 3: Cadena de responsabilidad

Examples

Cadena de responsabilidad ejemplo (php)

Un método llamado en un objeto subirá la cadena de objetos hasta que se encuentre uno que pueda manejar la llamada correctamente. Este ejemplo particular utiliza experimentos científicos con funciones que solo pueden obtener el título del experimento, la identificación de los experimentos o el tejido utilizado en el experimento.

```
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 }
}
```

Lea Cadena de responsabilidad en línea: https://riptutorial.com/es/design-patterns/topic/6083/cadena-de-responsabilidad

Capítulo 4: carga lenta

Introducción

la carga impaciente es costosa o el objeto a cargar podría no ser necesario en absoluto

Examples

JAVA carga perezosa

Llamar desde main ()

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

Clase pesada

```
/**
 *
 * 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;
  }
}
```

Lea carga lenta en línea: https://riptutorial.com/es/design-patterns/topic/9951/carga-lenta

Capítulo 5: Fábrica

Observaciones

> Proporcionar una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.
>
> *- GOF 1994*

Examples

Fábrica simple (Java)

Una fábrica reduce el acoplamiento entre el código que necesita crear objetos a partir del código de creación de objetos. La creación de objetos no se hace explícitamente llamando a un constructor de clase, sino llamando a alguna función que crea el objeto en nombre del llamante. Un ejemplo simple de Java es el siguiente:

```
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);
  }
}
```

En este ejemplo, el usuario solo da una pista sobre lo que necesita y la fábrica es libre de construir algo apropiado. Es una **inversión de dependencia** : el implementador del concepto de `Car` es libre de devolver un `Car` concreto apropiado solicitado por el usuario que a su vez no conoce los detalles del objeto concreto construido.

Este es un ejemplo simple de cómo funciona la fábrica; por supuesto, en este ejemplo, siempre es posible instanciar clases concretas; pero uno puede evitarlo ocultando clases concretas en un paquete, de manera que el usuario se vea obligado a usar la fábrica.

.Net Fiddle para el ejemplo anterior.

Fábrica abstracta (C ++)

El patrón **abstracto de fábrica** proporciona una manera de obtener una colección coherente de objetos a través de una colección de funciones de fábricas. En cuanto a cada patrón, el acoplamiento se reduce al abstraer la forma en que se crea un conjunto de objetos, de modo que el código de usuario no es consciente de los muchos detalles de los objetos que necesita.

El siguiente ejemplo de C ++ ilustra cómo obtener diferentes tipos de objetos de la misma familia de GUI (hipotética):

```
#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 el ejecutable generado se llama `abstractfactory` , la salida puede dar:

```
$ ./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
```

Ejemplo simple de Factory que usa un IoC (C #)

Las fábricas también se pueden utilizar junto con las bibliotecas de Inversión de control (IoC).

- El caso de uso típico de una fábrica de este tipo es cuando queremos crear un objeto basado en parámetros que no se conocen hasta el tiempo de ejecución (como el usuario actual).
- En estos casos, a veces puede ser difícil (si no imposible) configurar la biblioteca IoC sola para manejar este tipo de información contextual en tiempo de ejecución, por lo que podemos envolverla en una fábrica.

Ejemplo

- Supongamos que tenemos una clase de `User` , cuyas características (ID, nivel de autorización de seguridad, etc.) se desconocen hasta el tiempo de ejecución (ya que el usuario actual podría ser cualquiera que use la aplicación).
- Necesitamos tomar el Usuario actual y obtener un `ISecurityToken` para ellos, que luego se puede usar para verificar si el usuario tiene permiso para realizar ciertas acciones o no.
- La implementación de ISecurityToken variará dependiendo del nivel del Usuario; en otras palabras, ISecurityToken usa *polimorfismo* .

En este caso, tenemos dos implementaciones, que también utilizan *interfaces de marcador* para facilitar su identificación en la biblioteca IoC; La biblioteca IoC en este caso solo está formada e identificada por la abstracción `IContainer` .

> Tenga en cuenta también que muchas fábricas modernas de IoC tienen capacidades o complementos nativos que permiten la creación automática de fábricas, además de evitar la necesidad de interfaces de marcador como se muestra a continuación; sin embargo, como no todos lo hacen, este ejemplo se adapta a un concepto de funcionalidad común más simple y más bajo.

```
//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;
    }
}
```

A continuación, crearemos una fábrica de `SecurityToken` , que tomará como dependencia a nuestro `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>();
      }
   }
}
```

Una vez que hayamos registrado estos con el `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
```

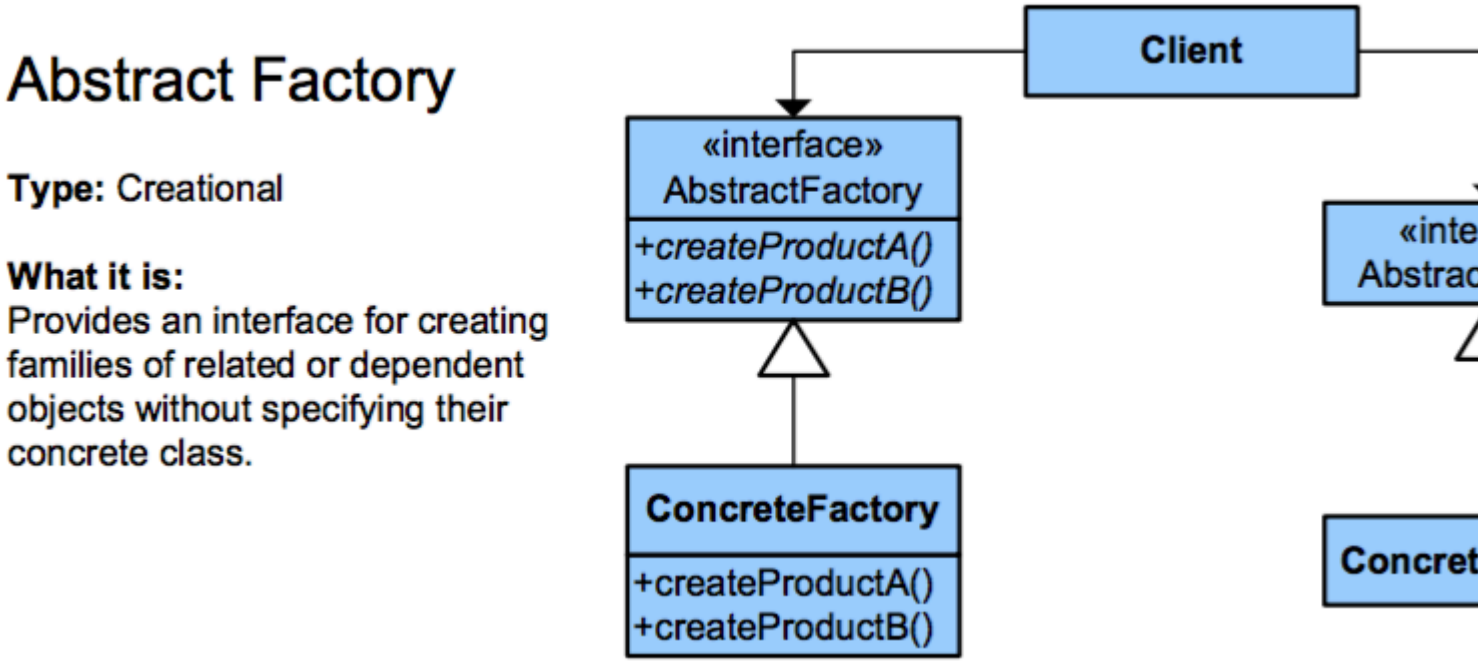
el código consumidor puede usarlo para obtener el token correcto en tiempo de ejecución:

```
readonly SecurityTokenFactory _tokenFactory;
...
...
public void LogIn(User user)
{
    var token = _tokenFactory.GetToken(user);
    user.SetSecurityToken(token);
```

```
}
```

De esta manera, nos beneficiamos de la encapsulación provista por la fábrica y también de la administración del ciclo de vida provista por la biblioteca IoC.

Una fabrica abstracta



El siguiente patrón de diseño se clasifica como un patrón creacional.

Se utiliza una fábrica abstracta para proporcionar una interfaz para crear familias de objetos relacionados, sin especificar clases concretas y se puede usar para ocultar clases específicas de la plataforma.

```
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();
    }
}
```

Luego, se usaría un proveedor / productor de algún tipo al que se le pasaría información que le permitiría devolver el tipo correcto de implementación de fábrica:

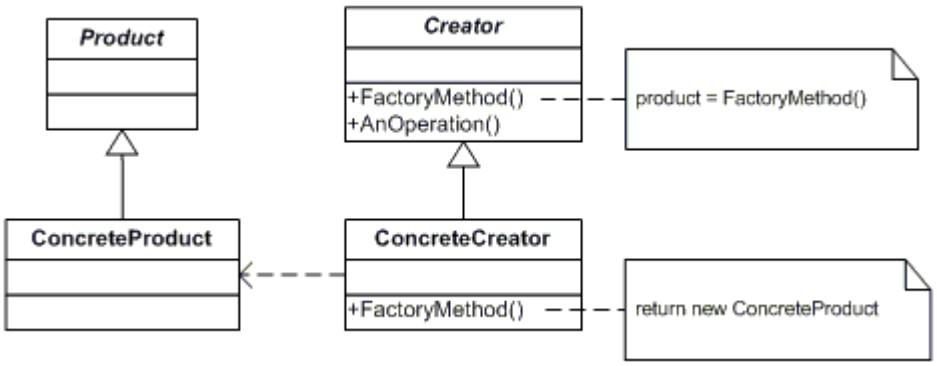
```
public final class FactorySupplier {

    // 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;
    }
}
```

Ejemplo de fábrica implementando el método de fábrica (Java)

Intención:

Defina una interfaz para crear un objeto, pero deje que las subclases decidan qué clase crear una instancia. Método de fábrica permite que una clase difiera la creación de instancias a subclases.

Diagrama UML:



Producto: Define una interfaz de los objetos que crea el método Factory.

ConcreteProduct: Implementa la interfaz del producto.

Creador: Declara el método de fábrica.

ConcreateCreator: implementa el método Factory para devolver una instancia de ConcreteProduct

Declaración de problema: cree una Factory of Games usando Factory Methods, que define la interfaz del juego.

Fragmento de código:

```
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());
    }
}
```

salida:

```
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
```

Este ejemplo muestra una clase `Factory` al implementar un `FactoryMethod` .

1. `Game` es la interfaz para todo tipo de juegos. Se define el método complejo: `createGame()`
2. `Chess, Ludo, Checkers` son diferentes variantes de juegos, que proporcionan implementación a `createGame()`
3. `public Game getGame(String gameName)` es `FactoryMethod` en la clase `IGameFactory`
4. `GameFactory` crea previamente diferentes tipos de juegos en el constructor. Implementa el método de fábrica `IGameFactory` .
5. Nombre del juego se pasa como argumento de línea de comando a `NotStaticFactoryDemo`
6. `getGame` en `GameFactory` acepta un nombre de juego y devuelve el objeto de `Game` correspondiente.

Cuándo usar:

1. ***Fábrica*** : cuando no desea exponer la lógica de creación de instancias de objetos al cliente / llamante
2. ***Abstract Factory*** : cuando desea proporcionar interfaz a familias de objetos relacionados o dependientes sin especificar sus clases concretas

3. ***Método de fábrica:*** para definir una interfaz para crear un objeto, pero deje que las subclases decidan qué clase crear una instancia.

Comparación con otros patrones creacionales:

1. El diseño comienza con **Factory Method** (menos complicado, más personalizable, las subclases proliferan) y evoluciona hacia **Abstract Factory, Prototype o Builder** (más flexible, más complejo) a medida que el diseñador descubre dónde se necesita más flexibilidad.

2. **Las** clases **abstractas de fábrica a** menudo se implementan con **métodos de fábrica** , pero también se pueden implementar utilizando **prototipos**

Referencias para lectura adicional: Sourcemaking design-patterns

Fábrica de peso mosca (C #)

En palabras simples:

Una fábrica de peso mosca que para una clave dada, ya conocida, siempre dará el mismo objeto como respuesta. Para las nuevas claves creará la instancia y la devolverá.

Usando la fábrica:

```
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);
```

Implementación:

```
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>();
}
```

Notas adicionales

Recomendaría agregar a esta solución el uso de un `IoC Container` (como se explica en un ejemplo diferente aquí) en lugar de crear sus propias nuevas instancias. Uno puede hacerlo agregando un nuevo registro para el `TResult` al contenedor y luego resolviéndolo (en lugar del `dictionary` en el ejemplo).

Método de fábrica

El patrón del método Factory es un patrón creacional que abstrae la lógica de creación de instancias de un objeto para desacoplar el código del cliente.

Cuando un método de fábrica pertenece a una clase que es una implementación de otro patrón de fábrica como Abstract factory , generalmente es más apropiado hacer referencia al patrón implementado por esa clase en lugar del patrón de método Factory.

El patrón del método de fábrica es más comúnmente referenciado cuando se describe un método de fábrica que pertenece a una clase que no es principalmente una fábrica.

Por ejemplo, puede ser ventajoso colocar un método de fábrica en un objeto que represente un concepto de dominio si ese objeto encapsula algún estado que simplifique el proceso de creación de otro objeto. Un método de fábrica también puede conducir a un diseño más alineado con el lenguaje ubicuo de un contexto específico.

Aquí hay un ejemplo de código:

```
//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");
```

Lea Fábrica en línea: https://riptutorial.com/es/design-patterns/topic/1375/fabrica

Capítulo 6: Fachada

Examples

Fachada del mundo real (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");
    }
}
```

Ejemplo de fachada en java.

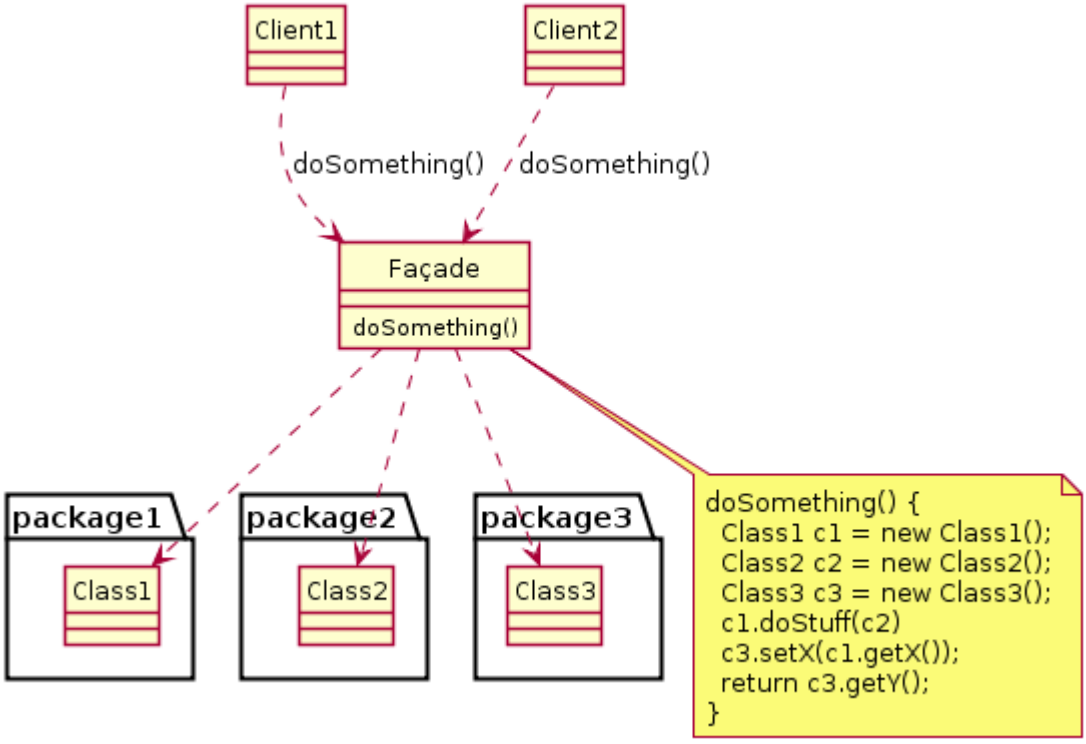
La fachada es patrón de diseño estructural. Oculta las complejidades de un sistema grande y proporciona una interfaz simple para el cliente.

El cliente solo utiliza **Fachada** y no le preocupan las interdependencias de los subsistemas.

Definición del libro Gang of Four:

> Proporcionar una interfaz unificada a un conjunto de interfaces en un subsistema. Façade define una interfaz de nivel superior que hace que el subsistema sea más fácil de usar

Estructura:



Ejemplo del mundo real:

Piense en algunos sitios de reserva de viajes como makemytrip, cleartrip, que ofrece servicios para reservar trenes, vuelos y hoteles.

Fragmento de código:

```
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){

    }
}
```

Explicación:

1. `FlightBooking, TrainBooking and HotelBooking` son diferentes subsistemas de grandes sistemas: `TravelFacade`

2. `TravelFacade` ofrece una interfaz simple para reservar una de las siguientes opciones

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

3. API de libro de TravelFacade realiza llamadas internas por debajo de las API de los subsistemas

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

4. De esta manera, `TravelFacade` proporciona una API más simple y sencilla sin exponer las API del subsistema.

Aplicabilidad y casos de uso (de Wikipedia):

1. Se requiere una interfaz simple para acceder a un sistema complejo.
2. Las abstracciones y las implementaciones de un subsistema están estrechamente acopladas.
3. Necesita un punto de entrada a cada nivel de software en capas.
4. El sistema es muy complejo o difícil de entender.

Lea Fachada en línea: https://riptutorial.com/es/design-patterns/topic/3516/fachada

Capítulo 7: Inyección de dependencia

Introducción

La idea general detrás de la inyección de dependencia es que usted diseña su aplicación en torno a componentes débilmente acoplados mientras se adhiere al principio de inversión de dependencia. Al no depender de implementaciones concretas, permite diseñar sistemas altamente flexibles.

Observaciones

La idea básica detrás de la inyección de dependencia es crear un código acoplado de forma más flexible. Cuando una clase, en lugar de crear sus propias dependencias, toma sus dependencias, la clase se vuelve más sencilla de probar como una unidad (prueba de unidad).

Para profundizar más en el acoplamiento flexible, la idea es que las clases se vuelvan dependientes de las abstracciones, en lugar de concreciones. Si la clase `A` depende de otra clase concreta `B` , entonces no hay pruebas reales de `A` sin `B` Si bien este tipo de prueba puede estar bien, no se presta al código de unidad comprobable. Un diseño débilmente acoplado definiría una abstracción `IB` (como ejemplo) de la cual dependería la clase `A` `IB` puede ser burlado para proporcionar un comportamiento comprobable, en lugar de confiar en la implementación real de `B` para poder proporcionar escenarios comprobables a `A`

Ejemplo estrechamente acoplado (C #):

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

En lo anterior, la clase `A` depende de `B` No hay pruebas `A` sin el hormigón `B` Si bien esto está bien en un escenario de pruebas de integración, es difícil realizar `A` prueba unitaria `A`

Una implementación más holgada de lo anterior podría ser:

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

public class A
{
    private readonly IB _iB;

    public A(IB iB)
```

```
    {
        _iB = iB;
    }

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

Las dos implementaciones parecen bastante similares, sin embargo hay una diferencia importante. La clase `A` ya no depende directamente de la clase `B` , ahora depende de `IB` . La Clase `A` ya no tiene la responsabilidad de renovar sus propias dependencias; ahora debe proporcionarlas ***a*** `A`

Examples

Inyección de Setter (C #)

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

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

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

Inyección Constructor (C #)

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

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

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

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

```
}
```

Lea Inyección de dependencia en línea: https://riptutorial.com/es/design-patterns/topic/1723/inyeccion-de-dependencia

Capítulo 8: Método de fábrica estático

Examples

Método de fábrica estática

Podemos proporcionar un nombre significativo para nuestros constructores.

Podemos proporcionar a varios constructores el mismo número y tipo de parámetros, algo que, como vimos anteriormente, no podemos hacer con los constructores de clases.

```
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() {...}
}
```

Ocultar acceso directo al constructor.

Podemos evitar proporcionar acceso directo a constructores que hacen un uso intensivo de recursos, como para bases de datos. clase pública 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);
    //...
  }
}
```

Método de fábrica estático C

El *método estático de fábrica* es una variación del patrón de *método de fábrica* . Se utiliza para crear objetos sin tener que llamar al constructor.

Cuándo usar el método de fábrica estática

- si desea dar un nombre significativo al método que genera su objeto.
- Si desea evitar la creación de objetos demasiado complejos, consulte Tuple Msdn .
- Si desea limitar el número de objetos creados (almacenamiento en caché)
- si desea devolver un objeto de cualquier subtipo de su tipo de retorno.

Hay algunas desventajas como

- Las clases sin un constructor público o protegido no pueden inicializarse en el método de fábrica estática.
- Los métodos estáticos de fábrica son como los métodos estáticos normales, por lo que no se distinguen de otros métodos estáticos (esto puede variar de IDE a IDE)

Ejemplo

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);
    }
}
```

Programa.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);
    }
}
```

Salida

> Una pizza con un diámetro de 28 cm y una pizza con un diámetro de 25 cm y una pizza con un diámetro de 35 cm

Lea Método de fábrica estático en línea: https://riptutorial.com/es/design-patterns/topic/6024/metodo-de-fabrica-estatico

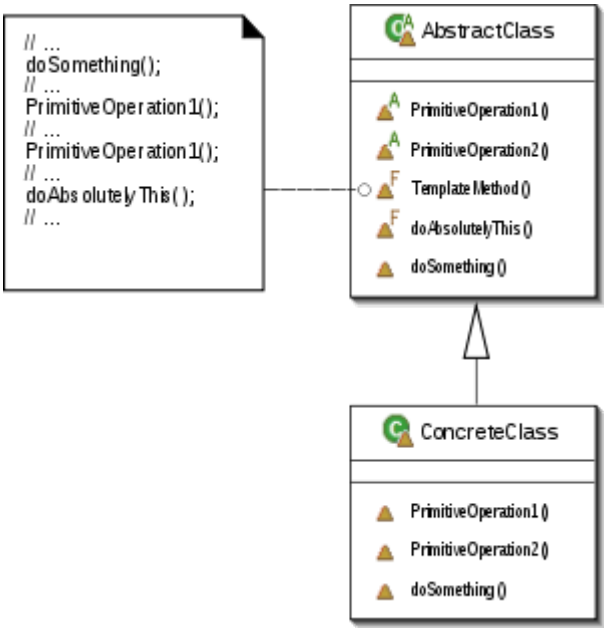
Capítulo 9: Método de plantilla

Examples

Implementación del método de plantilla en java.

El patrón de método de plantilla es un patrón de diseño de comportamiento que define el esqueleto del programa de un algoritmo en una operación, pasando algunos pasos a las subclases.

Estructura:



Notas clave:

1. Método de plantilla utiliza herencia
2. El método de plantilla implementado por la clase base no debe ser anulado. De esta manera, la estructura del algoritmo está controlada por la superclase y los detalles se implementan en las subclases.

Ejemplo de código:

```
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();
    }
}
```

Explicación:

1. `Game` es una súper clase `abstract` , que define un método de plantilla: `playGame()`
2. El esqueleto de `playGame()` se define en la clase base: `Game`
3. Las `playGame()` como `Chess, Ludo` y `Checkers` no pueden cambiar el esqueleto de `playGame()` . Pero pueden modificar el comportamiento de algunos pasos como

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

salida:

```
--------------------
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
--------------------
```

Lea Método de plantilla en línea: https://riptutorial.com/es/design-patterns/topic/4867/metodo-de-

plantilla

Capítulo 10: Monostato

Observaciones

Como nota al margen, algunas ventajas del patrón de `Monostate` sobre el `Singleton` :

- No hay un método de 'instancia' para poder acceder a una instancia de la clase.
- Un `Singleton` no se ajusta a la notación de los beans de Java, pero un `Monostate` sí lo hace.
- La vida de las instancias puede ser controlada.
- Los usuarios de `Monostate` no saben que están usando un `Monostate` .
- El polimorfismo es posible.

Examples

El Patrón de Monostato

El patrón de `Monostate` se suele denominar *azúcar sintáctico* sobre el patrón de `Singleton` o como un *`Singleton` conceptual* .

Evita todas las complicaciones de tener una sola instancia de una clase, pero todas las instancias usan los mismos datos.
Esto se logra principalmente mediante el uso de miembros de datos `static` .
Una de las características más importantes es que es absolutamente transparente para los usuarios, que desconocen por completo que están trabajando con un `Monostate` . Los usuarios pueden crear tantas instancias de `Monostate` como deseen y cualquier instancia es tan buena como otra para acceder a los datos.

La clase `Monostate` generalmente viene con una clase complementaria que se usa para actualizar la configuración si es necesario.

Sigue un ejemplo mínimo de un `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; }
};
```

Aquí hay un ejemplo de una implementación simple de 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;
    }
}
```

Jerarquías basadas en el monstruo

En contraste con el `Singleton` , el `Monostate` es adecuado para ser heredado para extender sus funcionalidades, siempre que los métodos de los miembros no sean `static` .
Sigue un ejemplo mínimo 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; }
};
```

Lea Monostato en línea: https://riptutorial.com/es/design-patterns/topic/6186/monostato

Capítulo 11: Multiton

Observaciones

Multitonitis

Igual que Singleton , Multiton puede considerarse una mala práctica. Sin embargo, hay ocasiones en las que puede usarlo sabiamente (por ejemplo, si está creando un sistema como ORM / ODM para conservar múltiples objetos).

Examples

Pool of Singletons (ejemplo PHP)

Multiton se puede utilizar como un contenedor para singletons. Esta es la implementación de Multiton, es una combinación de patrones Singleton y Pool.

Este es un ejemplo de cómo se puede crear la clase de grupo abstracta Multiton abstracta:

```
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 esta manera podemos crear una instancia de varios grupos de Singleton.

Registro de Singletons (ejemplo PHP)

Este patrón se puede usar para contener grupos de Singletons registrados, cada uno distinguido por un ID único:

```
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() {}
}
```

Esta es una forma simplificada de patrón que se puede usar para que ORM almacene varias entidades de un tipo dado.

Lea Multiton en línea: https://riptutorial.com/es/design-patterns/topic/6857/multiton

Capítulo 12: MVC, MVVM, MVP

Observaciones

Se puede argumentar que MVC y los patrones relacionados son en realidad patrones de arquitectura de software en lugar de patrones de diseño de software.

Examples

Controlador de vista de modelo (MVC)

1. ¿Qué es MVC?

El patrón del controlador de vista de modelo (MVC) es un patrón de diseño más comúnmente utilizado para crear interfaces de usuario. La principal ventaja de MVC es que separa:

- la representación interna del estado de la aplicación (el Modelo),
- cómo se presenta la información al usuario (la Vista), y
- la lógica que controla cómo el usuario interactúa con el estado de la aplicación (el Controlador).

2. Casos de uso de MVC.

El caso de uso principal de MVC está en la programación de la interfaz gráfica de usuario (GUI). El componente Ver escucha el componente del Modelo en busca de cambios. El modelo actúa como una emisora; cuando hay un modo de cambio en el Modelo, transmite sus cambios a la Vista y al Controlador. El controlador es utilizado por la vista para modificar el componente del modelo.

3. Implementación

Considere la siguiente implementación de MVC, donde tenemos una clase de modelo llamada `Animals` , una clase de vista llamada `DisplayAnimals` y una clase de controlador llamada `AnimalController` . El siguiente ejemplo es una versión modificada del tutorial en 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. Fuentes utilizadas:

Patrones de diseño - Patrón MVC

Diseño de aplicaciones Java SE con MVC

Modelo – vista – controlador

Modelo View ViewModel (MVVM)

1. ¿Qué es MVVM?

El patrón Model View ViewModel (MVVM) es un patrón de diseño más comúnmente utilizado para crear interfaces de usuario. Se deriva del popular patrón "Model View Controller" (MVC). La principal ventaja de MVVM es que separa:

- La representación interna del estado de la aplicación (el Modelo).
- Cómo se presenta la información al usuario (la Vista).
- La "lógica del convertidor de valores" responsable de exponer y convertir los datos del modelo para que los datos se puedan administrar y presentar fácilmente en la vista (el ViewModel).

2. Casos de uso de MVVM.

El caso de uso principal de MVVM es la programación de la interfaz gráfica de usuario (GUI). Se utiliza simplemente para programar eventos de interfaz de usuario mediante la separación de la capa de vista de la lógica de back-end que administra los datos.

En Windows Presentation Foundation (WPF), por ejemplo, la vista se diseña utilizando el lenguaje de marcado de marco XAML. Los archivos XAML están vinculados a ViewModels mediante el enlace de datos. De esta manera, la vista solo es responsable de la presentación y el modelo de vista solo es responsable de administrar el estado de la aplicación trabajando en los datos del modelo.

También se utiliza en la biblioteca de JavaScript KnockoutJS.

3. Implementación

Considere la siguiente implementación de MVVM usando C # .Net y WPF. Tenemos una clase de modelo llamada Animales, una clase de vista implementada en Xaml y un modelo de vista llamado AnimalViewModel. El siguiente ejemplo es una versión modificada del tutorial en MVC de Design Patterns - MVC Pattern .

Mira cómo el modelo no sabe nada, el ViewModel solo sabe sobre el modelo y la vista solo conoce el ViewModel.

El evento OnNotifyPropertyChanged permite actualizar tanto el modelo como la vista para que cuando ingrese algo en el cuadro de texto en la vista, se actualice el modelo. Y si algo actualiza el modelo, la vista se actualiza.

```
/*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. Fuentes utilizadas:

Model – view – viewmodel

Un ejemplo simple de MVVM

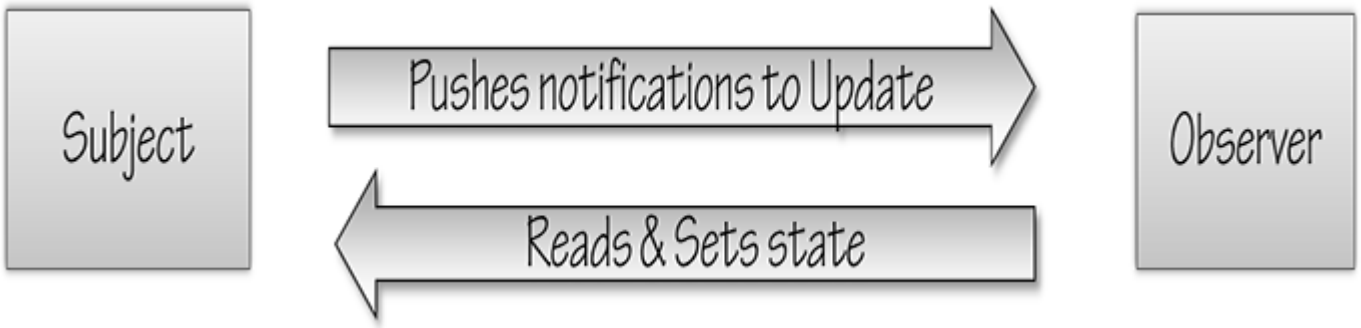
El ejemplo MVVM de C # WPF más simple del mundo

El patrón MVVM

Lea MVC, MVVM, MVP en línea: https://riptutorial.com/es/design-patterns/topic/7405/mvc--mvvm--mvp

Capítulo 13: Observador

Observaciones



¿Cuál es la intención?

- Adoptar el principio de separación de preocupaciones.
- Crea una separación entre el sujeto y el observador.
- Permita que los observadores múltiples reaccionen para cambiar un solo tema.

¿Cuál es la estructura?

- El sujeto proporciona una forma de registrarse, anular el registro, notificar.
- El observador proporciona una manera de actualizar.

Examples

Observador / Java

El patrón de observador permite a los usuarios de una clase suscribirse a eventos que suceden cuando esta clase procesa datos, etc. y ser notificado cuando ocurren estos eventos. En el siguiente ejemplo, creamos una clase de procesamiento y una clase de observador a las que se notificará mientras se procesa una frase, si encuentra palabras que tengan más de 5 letras.

La interfaz de `LongWordsObserver` define el observador. Implementar esta interfaz para registrar un observador en eventos.

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

La clase `WordEvent` es el evento que se enviará a las clases de observadores una vez que ocurran ciertos eventos (en este caso, se encontraron palabras largas)

```
// 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 clase `PhraseProcessor` es la clase que procesa la frase dada. Permite que los observadores se registren utilizando el método `addObserver` . Una vez que se encuentran palabras largas, se llamará a estos observadores utilizando una instancia de la clase `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 clase `LongWordsExample` muestra cómo registrar observadores, llamar al método de `process` y recibir alertas cuando se encuentran palabras largas.

```
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
    }
}
```

Observador que utiliza IObservable e IObserver (C #)

`IObserver<T>` e `IObservable<T>` se pueden usar para implementar un patrón de observador en .NET

- `IObservable<T>` interfaz `IObservable<T>` representa la clase que envía notificaciones
- `IObserver<T>` interfaz `IObserver<T>` representa la clase que los recibe

```
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();
  }
}
```

Uso

```
...
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: Patrones y prácticas de diseño en .NET: el patrón Observer

Lea Observador en línea: https://riptutorial.com/es/design-patterns/topic/3185/observador

Capítulo 14: Patrón compuesto

Examples

Maderero compuesto

El patrón compuesto es un patrón de diseño que permite tratar un grupo de objetos como una instancia única de un objeto. Es uno de los patrones de diseño estructural de Gang of Four.

El siguiente ejemplo muestra cómo se puede utilizar Composite para iniciar sesión en múltiples lugares mediante una invocación única de registro. Este enfoque se adhiere a los principios de SOLID porque le permite agregar un nuevo mecanismo de registro sin violar el principio de responsabilidad única (cada registrador tiene solo una responsabilidad) o el principio de apertura / cierre (puede agregar un nuevo registrador que se registrará en un nuevo lugar agregando una nueva implementación y no modificando los existentes).

```
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
```

Vale la pena mencionar que los registradores compuestos se pueden anidar (uno de los parámetros del constructor de registradores compuestos puede ser otro registrador compuesto) creando una estructura similar a un árbol.

Lea Patrón compuesto en línea: https://riptutorial.com/es/design-patterns/topic/4515/patron-compuesto

Capítulo 15: Patrón compuesto

Introducción

Compuesto permite a los clientes tratar objetos individuales y composiciones de objetos de manera uniforme. Por ejemplo, considere un programa que manipula un sistema de archivos. Los archivos son objetos simples y las carpetas son composiciones de archivos y carpetas. Sin embargo, por ejemplo, ambos tienen funciones de tamaño, nombre, etc. Sería más fácil y más conveniente tratar los objetos de archivos y carpetas de manera uniforme definiendo una Interfaz de recursos del sistema de archivos

Observaciones

El patrón compuesto se aplica cuando hay una jerarquía parcial de objetos y un cliente necesita tratar los objetos de manera uniforme, independientemente del hecho de que un objeto pueda ser una hoja (objeto simple) o una rama (objeto compuesto).

Examples

pseudocódigo para un administrador de archivos tontos

```
/*
* 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;
};
```

Lea Patrón compuesto en línea: https://riptutorial.com/es/design-patterns/topic/9197/patron-compuesto

Capítulo 16: Patrón de comando

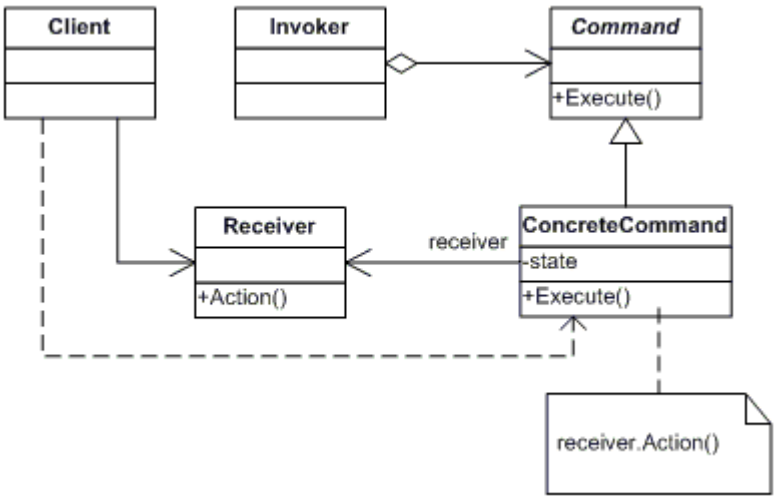
Examples

Ejemplo de patrón de comando en Java

definición de wikipedia :

> El patrón de comando es un patrón de diseño de comportamiento en el que un objeto se utiliza para encapsular toda la información necesaria para realizar una acción o desencadenar un evento en un momento posterior

Diagrama UML de dofactory :



Componentes básicos y flujo de trabajo:

1. `Command` declara una interfaz para los comandos abstractos como `execute()`
2. `Receiver` sabe cómo ejecutar un comando en particular
3. `Invoker` posee `ConcreteCommand` , que debe ser ejecutado.
4. `Client` crea `ConcreteCommand` y asigna un `Receiver`
5. `ConcreteCommand` define el enlace entre `Command` y `Receiver`

De esta manera, el patrón de comando desacopla al **remitente** (cliente) del **receptor a** través del **invocador** . **El Invoker** tiene un conocimiento completo de qué **Comando** se ejecutará y el **Comando** sabe qué **Receptor se** debe invocar para ejecutar una operación en particular.

Fragmento de código:

```
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();
    }
}
```

salida:

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

Explicación:

En este ejemplo,

1. **La** interfaz de **comandos** define el método `execute()` .
2. **OnCommand** es **ConcreteCommand** , que implementa el método `execute()` .
3. **El receptor** es la clase base.
4. **TV** y **DVDPlayer** son dos tipos de **receptores** , que se pasan a ConcreteCommand como OnCommand.
5. **Invoker** contiene el **comando** . Es la clave para separar el remitente del **receptor** .
6. **El invocador** recibe **OnCommand** -> que llama a **Receiver** (TV) para ejecutar este comando.

Al usar Invoker, puedes encender TV y DVDPlayer. Si extiendes este programa, también apagarás TV y DVDPlayer.

Casos de uso clave:

1. Para implementar el mecanismo de devolución de llamada.
2. Implementar funcionalidad deshacer y rehacer.
3. Mantener un historial de comandos.

Lea Patrón de comando en línea: https://riptutorial.com/es/design-patterns/topic/2677/patron-de-comando

Capítulo 17: Patrón de constructor

Observaciones

Separa la construcción de un objeto complejo de su representación para que el mismo proceso de construcción pueda crear diferentes representaciones.

- Separa la lógica de la representación.
- Reutilizar la lógica para trabajar con diferentes conjuntos de datos.

Examples

Patrón del constructor / C # / Interfaz fluida

```
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;
    }
}
```

Ejemplo de uso:

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

Patrón Builder / Implementación Java

El patrón Builder le permite crear una instancia de una clase con muchas variables opcionales de una manera fácil de leer.

Considere el siguiente código:

```
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...

}
```

Todo esto está muy bien si todos los parámetros son necesarios. ¿Qué pasa si hay muchas más variables y / o algunas de ellas son opcionales? No desea crear un gran número de constructores con cada combinación posible de parámetros obligatorios y opcionales porque resulta difícil mantenerlos y comprenderlos los desarrolladores. Es posible que tampoco desee tener una larga lista de parámetros en los que el usuario deba ingresar muchos de ellos como nulos.

El patrón Builder crea una clase interna llamada Builder que se utiliza para instanciar solo las variables opcionales deseadas. Esto se realiza a través de métodos para cada variable opcional que toman el tipo de variable como un parámetro y devuelven un objeto Builder para que los métodos se puedan encadenar entre sí. Todas las variables necesarias se colocan en el constructor del generador para que no se puedan omitir.

El Generador también incluye un método llamado `build()` que devuelve el objeto en el que se

encuentra y debe llamarse al final de la cadena de llamadas de métodos al generar el objeto.

Siguiendo con el ejemplo anterior, este código usa el patrón de Generador para la clase de Computadora.

```
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 ejemplo de cómo se usaría esta clase:

```
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();
    }

}
```

Este ejemplo muestra cómo el patrón de constructor puede permitir una gran flexibilidad en la forma en que se crea una clase con bastante poco esfuerzo. El objeto Computadora se puede implementar en función de la configuración deseada de los que llaman de una manera fácil de leer con poco esfuerzo.

Patrón de constructor en Java con composición.

Intención:

Separe la construcción de un objeto complejo de su representación para que el mismo proceso de construcción pueda crear diferentes representaciones

El patrón de generador es útil cuando tiene pocos atributos obligatorios y muchos atributos opcionales para construir un objeto. Para crear un objeto con diferentes atributos obligatorios y opcionales, debe proporcionar un constructor complejo para crear el objeto. El patrón del generador proporciona un proceso paso a paso simple para construir un objeto complejo.

Caso de uso de la vida real:

Los diferentes usuarios de FaceBook tienen atributos diferentes, que consisten en atributos obligatorios como el nombre de usuario y atributos opcionales como UserBasicInfo y ContactInfo. Algunos usuarios simplemente proporcionan información básica. Algunos usuarios proporcionan información detallada que incluye información de contacto. En ausencia del patrón Builder, debe proporcionar un constructor con todos los parámetros obligatorios y opcionales. Pero el patrón Builder simplifica el proceso de construcción al proporcionar un proceso paso a paso simple para construir el objeto complejo.

Consejos:

1. Proporcionar una clase constructora anidada estática.
2. Proporcionar constructor para atributos obligatorios de objeto.
3. Proporcionar métodos de establecimiento y obtención para atributos opcionales del objeto.
4. Devuelve el mismo objeto Builder después de configurar los atributos opcionales.
5. Proporcionar el método build (), que devuelve un objeto complejo.

Fragmento de código:

```
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);
    }
```

```
}
```

salida:

```
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
```

Explicación:

1. `FaceBookUser` es un objeto complejo con los siguientes atributos que utilizan composición:

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

2. `FaceBookUserBuilder` es una clase de constructor estático, que contiene y construye `FaceBookUser` .

3. `userName` es solo un parámetro obligatorio para construir `FaceBookUser`

4. `FaceBookUserBuilder` construye `FaceBookUser` configurando parámetros opcionales: `UserBasicInfo` y `ContactInfo`

5. Este ejemplo ilustra tres diferentes FaceBookUsers con diferentes atributos, construidos desde Builder.

   1. fbUser1 se creó como FaceBookUser con solo el atributo userName
   2. fbUser2 fue creado como FaceBookUser con userName y UserBasicInfo
   3. fbUser3 fue creado como FaceBookUser con userName, UserBasicInfo y ContactInfo

En el ejemplo anterior, se ha utilizado la composición en lugar de duplicar todos los atributos de FaceBookUser en la clase Builder.

En los patrones de creación, primero comenzaremos con patrones simples como `FactoryMethod` y `FactoryMethod` hacia patrones más flexibles y complejos como `AbstractFactory` y `Builder` .

Java / Lombok

```
import lombok.Builder;

@Builder
public class Email {

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

}
```

Ejemplo de uso:

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

Patrón de generador avanzado con Java 8 expresión 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;
 }
}
```

Vieja forma

```
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);
}
```

Manera avanzada:

```
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);
}
```

}

Uso:

```
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();
```

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

Lea Patrón de constructor en línea: https://riptutorial.com/es/design-patterns/topic/1811/patron-de-constructor

Capítulo 18: Patrón de diseño del objeto de acceso a datos (DAO)

Examples

Patrón de diseño de objetos de acceso a datos J2EE con Java

El patrón de diseño del *objeto de acceso a datos (DAO)* es un patrón de diseño estándar de J2EE.

En este patrón de diseño, se accede a los datos a través de clases que contienen métodos para acceder a datos desde bases de datos u otras fuentes, que se denominan *objetos de acceso a datos* . La práctica estándar asume que hay clases POJO. DAO se puede combinar con otros patrones de diseño para acceder a los datos, como con MVC (controlador de vista de modelo), patrones de comando, etc.

El siguiente es un ejemplo de patrón de diseño DAO. Tiene una clase de **Empleado** , un DAO para Empleado llamado **EmployeeDAO** y una clase de **ApplicationView** para demostrar los ejemplos.

Empleado.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

}
```

Empleado

```
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;
    }
```

```
}
```

Por lo tanto, tenemos un ejemplo en el que vemos cómo usar el patrón de diseño de Data Access Object.

Lea Patrón de diseño del objeto de acceso a datos (DAO) en línea: https://riptutorial.com/es/design-patterns/topic/6351/patron-de-diseno-del-objeto-de-acceso-a-datos--dao-

Capítulo 19: Patrón de estrategia

Examples

Ocultar los detalles de la implementación de la estrategia.

Una guía muy común en el diseño orientado a objetos es "lo menos posible pero lo necesario". Esto también se aplica al patrón de estrategia: por lo general, es recomendable ocultar los detalles de la implementación, por ejemplo, qué clases implementan las estrategias.

Para estrategias simples que no dependen de parámetros externos, el enfoque más común es hacer que la clase implementadora sea privada (clases anidadas) o paquete-privada y exponga una instancia a través de un campo estático de una clase pública:

```
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
);
```

El campo público `Animal.AGE_COMPARATOR` define una estrategia que luego se puede usar en métodos como `Collections.sort` , pero no requiere que el usuario sepa nada sobre su implementación, ni siquiera la clase implementadora.

Si lo prefieres, puedes usar una clase anónima:

```
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 estrategia es un poco más compleja y requiere parámetros, es muy común usar métodos de fábrica estáticos como `Collections.reverseOrder(Comparator<T>)` . El tipo de retorno del método no debe exponer ningún detalle de implementación, por ejemplo, `reverseOrder()` se implementa como

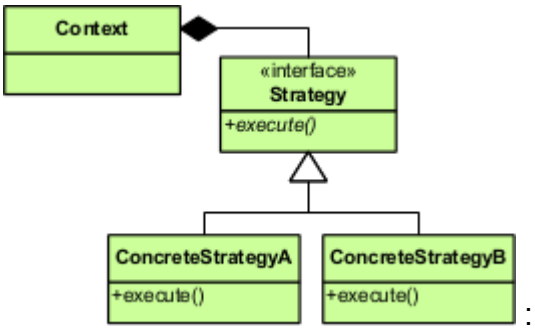
```
public static <T> Comparator<T> reverseOrder(Comparator<T> cmp) {
  // (Irrelevant lines left out.)
  return new ReverseComparator2<>(cmp);
}
```

Ejemplo de patrón de estrategia en java con clase de contexto

Estrategia:

`Strategy` es un patrón de comportamiento, que permite cambiar el algoritmo dinámicamente de una familia de algoritmos relacionados.

UML de patrón de estrategia 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);
    }

}
```

salida:

```
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
```

Declaración de problema: Ofrezca un descuento del 25% en el precio del artículo para los meses de julio a diciembre. No ofrezca ningún descuento por los meses de enero-junio.

El ejemplo anterior muestra el uso del patrón de `Strategy` con `Context` . `Context` se puede utilizar como un único punto de contacto para el `Client` .

Dos estrategias: `NoOfferStrategy` y `QuarterDiscountStrategy` se han declarado según la declaración del problema.

Como se muestra en la columna de salida, obtendrá un descuento dependiendo del mes que haya ingresado

Caso (s) de uso para el patrón de estrategia :

1. Use este patrón cuando tenga una familia de algoritmos intercambiables y tenga que cambiar el algoritmo en tiempo de ejecución.

2. Mantenga el código limpio eliminando las declaraciones condicionales

Patrón de estrategia sin una clase de contexto / Java

El siguiente es un ejemplo simple de usar el patrón de estrategia sin una clase de contexto. Existen dos estrategias de implementación que implementan la interfaz y resuelven el mismo problema de diferentes maneras. Los usuarios de la clase de traducción en inglés pueden llamar al método de traducción y elegir qué estrategia les gustaría usar para la traducción, especificando la estrategia deseada.

```
// 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
    }
}
```

Usando interfaces funcionales de Java 8 para implementar el patrón de Estrategia

El propósito de este ejemplo es mostrar cómo podemos realizar el patrón de estrategia utilizando las interfaces funcionales de Java 8. Comenzaremos con un simple uso de códigos de casos en Java clásico, y luego lo recodificaremos en la forma de Java 8.

El problema de ejemplo que usamos es una familia de algoritmos (estrategias) que *describen* diferentes formas de comunicación a distancia.

La versión clásica de Java

El contrato para nuestra familia de algoritmos se define mediante la siguiente interfaz:

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

Luego podemos implementar una serie de algoritmos, de la siguiente manera:

```
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..";
    }
}
```

Estos pueden ser instanciados de la siguiente manera:

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

A continuación, implementamos un servicio que utiliza la estrategia:

```
public class CommunicationService {
    private CommunicateInterface communcationMeans;

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

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

Finalmente, podemos utilizar las diferentes estrategias de la siguiente manera:

```
CommunicationService communicationService = new CommunicationService();

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

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

Usando interfaces funcionales de Java 8

El contrato de las diferentes implementaciones de algoritmos no necesita una interfaz dedicada. En su lugar, podemos describirlo utilizando la interfaz `java.util.function.Function<T, R>` .

Los diferentes algoritmos que componen `the family of algorithms` se pueden expresar como expresiones lambda. Esto reemplaza las clases de estrategia *y* sus instancias.

```
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..";
```

A continuación, podemos codificar el "servicio" de la siguiente manera:

```
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);
    }
}
```

Finalmente utilizamos las siguientes estrategias.

```
CommunicationService communicationService = new CommunicationService();

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

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

O incluso:

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

Estrategia (PHP)

Ejemplo de www.phptherightway.com

```
<?php
```

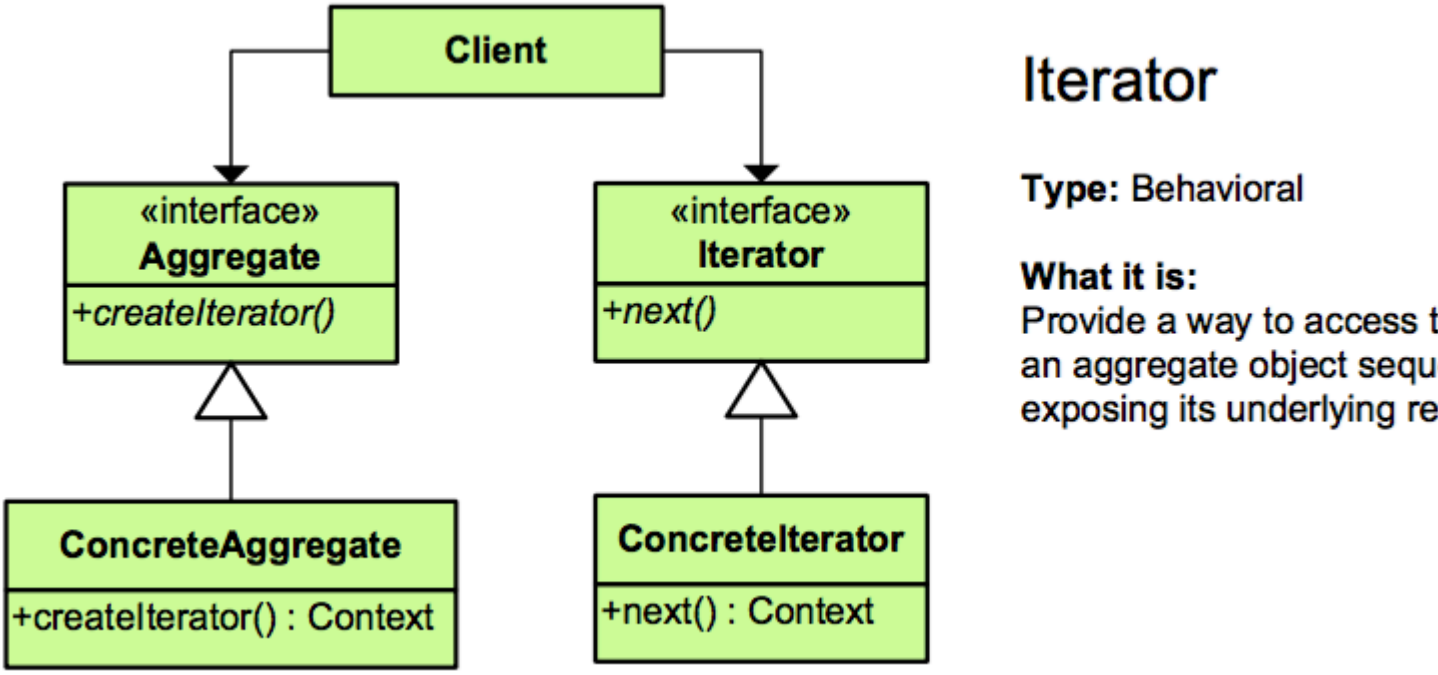
```
interface OutputInterface
{
    public function load();
}

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;
    }
}
```

Lea Patrón de estrategia en línea: https://riptutorial.com/es/design-patterns/topic/1331/patron-de-estrategia

Capítulo 20: Patrón de iterador

Examples

El patrón iterador



Las colecciones son una de las estructuras de datos más utilizadas en ingeniería de software. Una Colección es solo un grupo de objetos. Una colección puede ser una Lista, una matriz, un mapa, un árbol o cualquier cosa. Por lo tanto, una colección debe proporcionar alguna forma de acceder a sus elementos sin exponer su estructura interna. Deberíamos poder recorrerlo de la misma manera, independientemente del tipo de colección que sea.

La idea del patrón de iterador es asumir la responsabilidad de acceder al objeto de una colección y colocarlo en un objeto iterador. El objeto iterador a cambio mantendrá el orden de iteración, mantendrá un seguimiento del elemento actual y deberá tener una forma de obtener el siguiente elemento.

Generalmente, la clase de colección lleva dos componentes: la clase en sí, y es `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;
       }
    }
}
```

Lea Patrón de iterador en línea: https://riptutorial.com/es/design-patterns/topic/7061/patron-de-iterador

Capítulo 21: Patrón de objeto nulo

Observaciones

El objeto nulo es un objeto sin valor referenciado o con un comportamiento neutral definido. Su propósito es eliminar la necesidad de puntero nulo / verificación de referencia.

Examples

Patrón de objeto nulo (C ++)

Suponiendo una clase abstracta:

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

En lugar 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);
}
```

Puede crear un registrador de objetos nulos:

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

y luego cambia `doJob` en lo siguiente:

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

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

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

Objeto nulo Java usando enumeración

Dada una interfaz:

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

En lugar de uso:

```
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
}
```

Debido a que los objetos nulos no tienen un estado, tiene sentido utilizar un singleton de enumeración para ellos, por lo que, dado un objeto nulo implementado así:

```
public enum NullLogger implements Logger {
    INSTANCE;

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

A continuación, puede evitar los cheques nulos.

```
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);
}
```

Lea Patrón de objeto nulo en línea: https://riptutorial.com/es/design-patterns/topic/6177/patron-de-objeto-nulo

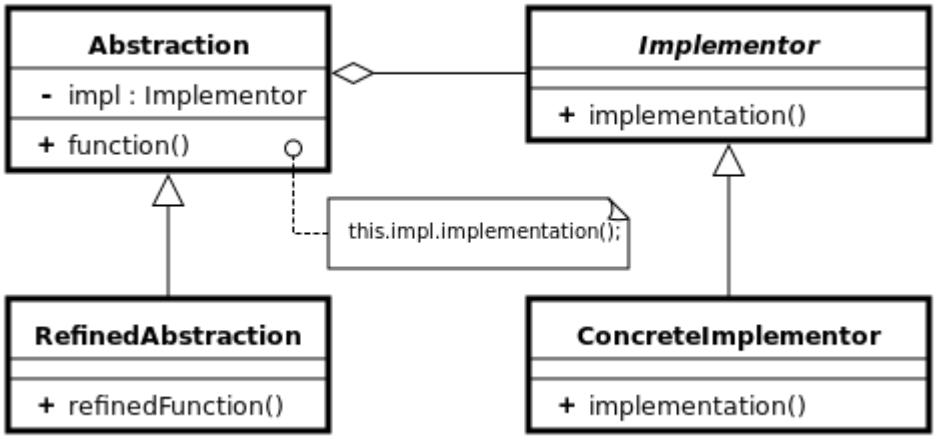
Capítulo 22: Patrón de puente

Examples

Implementación de patrón puente en java

El patrón de puente desacopla la abstracción de la implementación para que ambos puedan variar independientemente. Se ha logrado con la composición en lugar de la herencia.

Puente UML diagrama de wikipedia:



Tienes cuatro componentes en este patrón.

`Abstraction` : define una interfaz.

`RefinedAbstraction` : Implementa la abstracción:

`Implementor` : define una interfaz para la implementación.

`ConcreteImplementor` : Implementa la interfaz del implementador.

`The crux of Bridge pattern :` dos jerarquías de clases ortogonales que usan composición (y no herencia). La jerarquía de abstracción y la jerarquía de implementación pueden variar independientemente. Implementación nunca se refiere a la abstracción. La abstracción contiene la interfaz de implementación como miembro (a través de la composición). Esta composición reduce un nivel más de jerarquía de herencia.

Caso de uso de la palabra real:

Permitir que diferentes vehículos tengan ambas versiones de sistema manual y automático.

Código de ejemplo:

```
/* 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();
    }
}
```

salida:

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

Explicación:

1. `Vehicle` es una abstracción.
2. `Car` y `Truck` son dos implementaciones concretas de `Vehicle` .
3. `Vehicle` define un método abstracto: `addGear()` .
4. `Gear` es la interfaz del implementador.
5. `ManualGear` y `AutoGear` son dos implementaciones de `Gear`
6. `Vehicle` contiene la interfaz del `implementor` en lugar de implementar la interfaz. `Compositon` de la interfaz del implementador es el eje de este patrón: *permite que la abstracción y la implementación varíen independientemente.*
7. `Car` y `Truck` definen implementación (abstracción redefinida) para la abstracción: `addGear()` : Contiene `Gear` - `Manual` o `Auto`

Caso (s) de uso para el patrón de puente :

1. **La abstracción** y la **implementación** pueden ser independientes entre sí y no están vinculadas en el momento de la compilación
2. Mapa de jerarquías ortogonales: una para *abstracción* y otra para *implementación* .

Lea Patrón de puente en línea: https://riptutorial.com/es/design-patterns/topic/4011/patron-de-puente

Capítulo 23: Patrón de repositorio

Observaciones

Acerca de la implementación de `IEnumerable<TEntity> Get(Expression<Func<TEntity, bool>> filter)` : La idea de esto es usar Expresiones como `i => x.id == 17` para escribir solicitudes genéricas. Es una forma de consultar datos sin utilizar el lenguaje de consulta específico de su tecnología. La implementación es bastante extensa, por lo tanto, es posible que desee considerar otras alternativas, como métodos específicos en sus repositorios implementados: un `CompanyRepository` imaginario podría proporcionar el método `GetByName(string name)` .

Examples

Repositorios de solo lectura (C #)

Se puede usar un patrón de repositorio para encapsular el código específico de almacenamiento de datos en los componentes designados. La parte de su aplicación que necesita los datos solo funcionará con los repositorios. Querrá crear un repositorio para cada combinación de elementos que almacene y su tecnología de base de datos.

Los repositorios de solo lectura se pueden usar para crear repositorios a los que no se les permite manipular datos.

Las 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);
}
```

Una implementación de ejemplo utilizando ElasticSearch como tecnología (con 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;
    }
}
```

Con esta implementación, ahora puede crear Repositorios específicos para los elementos que desea almacenar o acceder. Cuando se utiliza la búsqueda elástica, es común que algunos componentes solo lean los datos, por lo que se deben usar repositorios de solo lectura.

Patrón de repositorio utilizando Entity Framework (C #)

Interfaz de repositorio;

```
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);
}
```

Repositorio genérico;

```
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);
    }
}
```

Ejemplo de uso utilizando una clase de hotel de demostración;

```
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();
```

Lea Patrón de repositorio en línea: https://riptutorial.com/es/design-patterns/topic/6254/patron-de-repositorio

Capítulo 24: Patrón de visitante

Examples

Ejemplo de patrón de visitante en C ++

En lugar de

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

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

Los visitantes pueden ser utilizados:

```
// 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;
};
```

Ahora las formas concretas:

```
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;
};
```

luego los visitantes:

```
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.;
};
```

Y úsalo:

```
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;
}
```

Rendimiento esperado:

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

Manifestación

Explicación :

- In `void Square::accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }` , se conoce el tipo estático de `this` y, por lo tanto, la sobrecarga elegida (en el momento de la compilación) es `void IVisitor::visit(const Square&);` .
- Para `square.accept(visitor);` llamada, el envío dinámico a través de `virtual` se utiliza para saber qué `accept` para llamar.

Pros :

- Puede agregar una nueva funcionalidad (`SerializeAsXml` , ...) a la clase `IShape` simplemente agregando un nuevo visitante.

Contras :

- Agregar una nueva forma concreta (`Triangle` , ...) requiere modificar todos los visitantes.

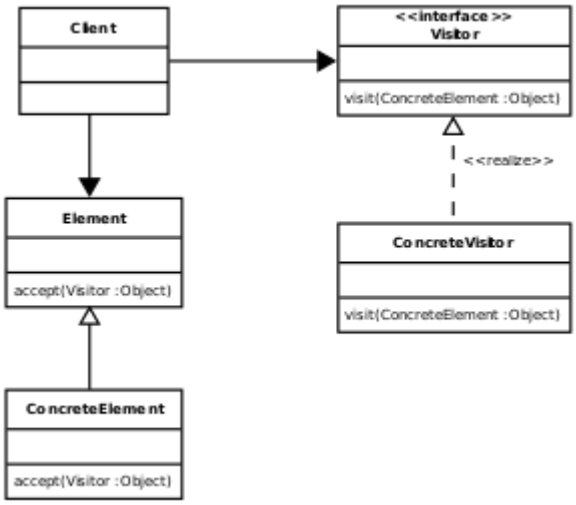
La alternativa de poner todas las funcionalidades como métodos `virtual` en `IShape` tiene ventajas y desventajas opuestas: agregar una nueva funcionalidad requiere modificar todas las formas existentes, pero agregar una nueva forma no afecta las clases existentes.

Ejemplo de patrón de visitante en java

`Visitor` patrón de `Visitor` permite agregar nuevas operaciones o métodos a un conjunto de clases sin modificar la estructura de esas clases.

Este patrón es especialmente útil cuando desea centralizar una operación particular en un objeto sin extender el objeto o sin modificar el objeto.

Diagrama UML de wikipedia:



Fragmento de código:

```
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);
        }
    }
}
```

Explicación:

1. `Visitable` (`Element`) es una interfaz y este método de interfaz debe agregarse a un conjunto de clases.
2. `Visitor` es una interfaz, que contiene métodos para realizar una operación en elementos `Visitable` .
3. `GameVisitor` es una clase que implementa una interfaz de `Visitor` (`ConcreteVisitor`).
4. Cada elemento `Visitable` acepta `Visitor` e invoca un método relevante de interfaz de `Visitor` .
5. Puedes tratar el `Game` como `Element` y los juegos concretos como `Chess,Checkers and Ludo` como `Element ConcreteElements` .

En el ejemplo anterior, `Chess, Checkers and Ludo` son tres juegos diferentes (y clases `Visitable`). En un buen día, me encontré con un escenario para registrar estadísticas de cada juego. Por lo tanto, sin modificar la clase individual para implementar la funcionalidad de estadísticas, puede centralizar esa responsabilidad en la clase `GameVisitor` , que hace el truco por usted sin modificar la estructura de cada juego.

salida:

```
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
```

Casos de uso / Aplicabilidad:

1. *Se deben realizar operaciones similares* en objetos de diferentes tipos agrupados en una estructura
2. Necesita ejecutar muchas operaciones distintas y no relacionadas. *Separa la operación de los objetos. Estructura.*
3. Nuevas operaciones deben ser agregadas *sin cambios en la estructura del objeto.*
4. *Reúna las operaciones relacionadas en una sola clase en* lugar de obligarlo a cambiar o derivar clases
5. Agregue funciones a las bibliotecas de clases para las cuales *no tiene la fuente o no puede cambiar la fuente*

Referencias adicionales:

oodesign

sourcemaking

Ejemplo de visitante 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 ejemplo de uso:

```
// 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);
}
```

Atravesando objetos grandes

El patrón visitante se puede utilizar para atravesar estructuras.

```
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);
        }
    }
}
```

Lea Patrón de visitante en línea: https://riptutorial.com/es/design-patterns/topic/4579/patron-de-visitante

Capítulo 25: Patrón decorador

Introducción

El patrón Decorator permite que un usuario agregue una nueva funcionalidad a un objeto existente sin alterar su estructura. Este tipo de patrón de diseño se encuentra bajo un patrón estructural ya que este patrón actúa como un envoltorio para la clase existente.

Este patrón crea una clase decoradora que envuelve la clase original y proporciona funcionalidad adicional manteniendo intacta la firma de los métodos de clase.

Parámetros

Parámetro	Descripción
Bebida	puede ser té o café

Examples

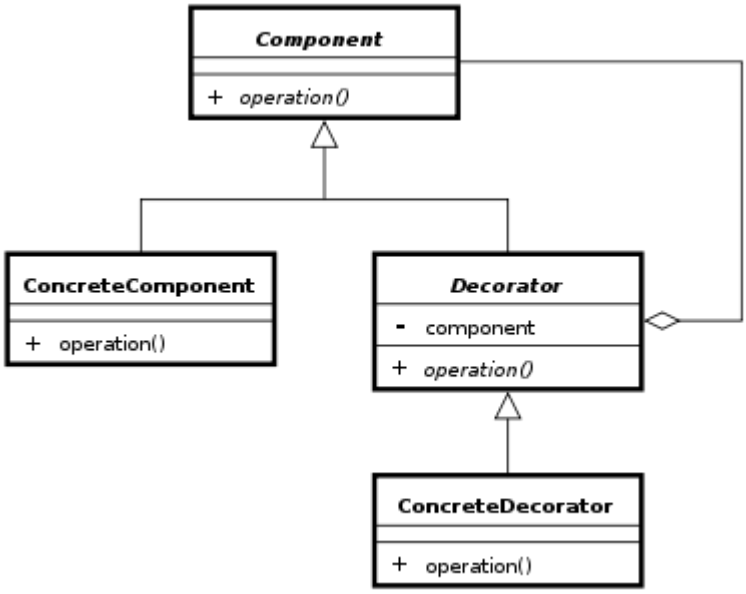
VendingMachineDecorator

Definición de decorador según Wikiepdia:

El patrón Decorator se puede usar para extender (decorar) la funcionalidad de un determinado objeto de forma estática, o en algunos casos en tiempo de ejecución, independientemente de otras instancias de la misma clase, siempre que se realicen algunos trabajos básicos en el momento del diseño.

El decorador asigna responsabilidades adicionales a un objeto dinámicamente. Los decoradores ofrecen una alternativa flexible a las subclases para extender la funcionalidad.

Patrón decorador contiene cuatro componentes.



1. Interfaz de componentes: define una interfaz para ejecutar operaciones particulares
2. ConcreteComponent: implementa las operaciones definidas en la interfaz de componentes
3. Decorador (abstracto): es una clase abstracta, que amplía la interfaz del componente. Contiene interfaz de componentes. En ausencia de esta clase, necesita muchas subclases de ConcreteDecorators para diferentes combinaciones. La composición del componente reduce las subclases innecesarias.
4. ConcreteDecorator: Posee la implementación de Abstract Decorator.

Volviendo al código de ejemplo,

1. *La bebida* es componente. Define un método abstracto: decorateBeverage
2. *El té* y el *café* son implementaciones concretas de *bebidas* .
3. *BeverageDecorator* es una clase abstracta, que contiene *Beverage*
4. SugarDecorator y LemonDecorator son decoradores de concreto para BeverageDecorator.

EDITAR: Cambié el ejemplo para reflejar el escenario del mundo real de calcular el precio de la bebida agregando uno o más sabores como el azúcar, el limón, etc. (los sabores son decoradores)

```
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();
    }
}
```

salida:

```
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
```

Este ejemplo calcula el costo de la bebida en la máquina expendedora después de agregar muchos sabores a la bebida.

En el ejemplo anterior:

Costo del té = 10, Limón = 3 y Azúcar = 5. Si haces Azúcar + Limón + Té, cuesta 18.

Costo del café = 15, Limón = 3 y Azúcar = 5. Si haces Azúcar + Limón + Café, cuesta 23

Al utilizar el mismo Decorador para ambas bebidas (té y café), se ha reducido el número de subclases. En ausencia del patrón de Decorator, deberías tener diferentes subclases para diferentes combinaciones.

Las combinaciones serán así:

```
SugarLemonTea
```

```
SugarTea
LemonTea

SugarLemonCapaccuino
SugarCapaccuino
LemonCapaccuino
```

etc.

Al utilizar el mismo `Decorator` para ambas bebidas, el número de subclases se ha reducido. Es posible debido a la `composition` lugar del concepto de `inheritance` utilizado en este patrón.

Comparación con otros patrones de diseño (del artículo de fuente)

1. *Adaptador* proporciona una interfaz diferente a su tema. *Proxy* proporciona la misma interfaz. *Decorador* proporciona una interfaz mejorada.

2. *El adaptador* cambia la interfaz de un objeto, *Decorator* mejora las responsabilidades de un objeto.

3. *Composite* y *Decorator* tienen diagramas de estructura similares, lo que refleja el hecho de que ambos dependen de la composición recursiva para organizar un número abierto de objetos

4. *Decorator* está diseñado para permitirle agregar responsabilidades a los objetos sin crear subclases. *El* enfoque *del compuesto* no es el embellecimiento sino la representación.

5. *Decorador* y *Proxy* tienen diferentes propósitos pero estructuras similares.

6. *Decorador* te permite cambiar la piel de un objeto. *La estrategia* te permite cambiar las agallas.

Casos de uso clave:

1. Añadir funcionalidades / responsabilidades adicionales dinámicamente
2. Eliminar funcionalidades / responsabilidades dinámicamente.
3. Evite demasiada subclasificación para agregar responsabilidades adicionales.

Decorador de caching

Este ejemplo muestra cómo agregar capacidades de almacenamiento en caché a `DbProductRepository` usando el patrón Decorator. Este enfoque se adhiere a los principios de SOLID porque le permite agregar almacenamiento en caché sin violar el principio de responsabilidad única o el principio de apertura / cierre .

```
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)
}
```

Uso:

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

El resultado de invocar `GetProduct` será: recuperar el producto de la memoria caché (responsabilidad del decorador), si el producto no estaba en la memoria caché, continúe con la invocación a `DbProductRepository` y recupere el producto de DB. Después de que este producto se pueda agregar al caché, las llamadas subsiguientes no afectarán a la base de datos.

Lea Patrón decorador en línea: https://riptutorial.com/es/design-patterns/topic/1720/patron-decorador

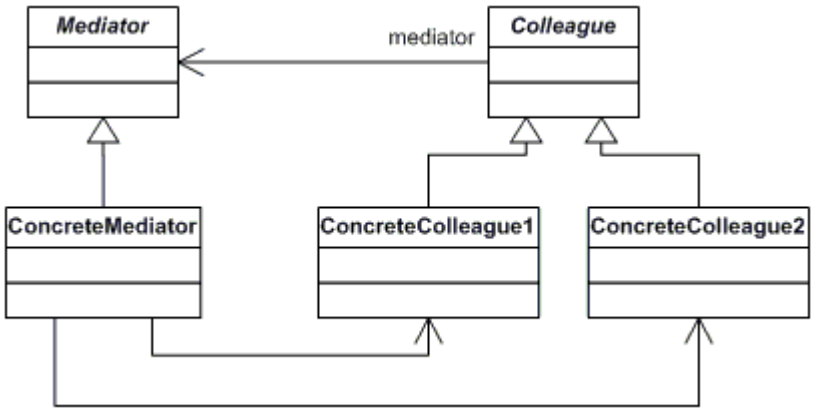
Capítulo 26: Patrón mediador

Examples

Ejemplo de patrón mediador en java

El patrón de *mediador* define un objeto (mediador) que encapsula cómo interactúa un conjunto de objetos. Permite la comunicación de muchos a muchos.

Diagrama UML:



Componentes clave:

`Mediator`: define una interfaz para la comunicación entre colegas.

`Colleague` : es una clase abstracta, que define los eventos que deben comunicarse entre colegas.

`ConcreteMediator` : implementa el comportamiento cooperativo al coordinar los objetos de los `Colleague` y mantiene a sus colegas

`ConcreteColleague` : implementa las operaciones de notificación recibidas a través del `Mediator` , que ha sido generado por otro `Colleague`

Un ejemplo del mundo real:

Usted está manteniendo una red de computadoras en la topología de `Mesh` .

Una red de malla es una topología de red en la que cada nodo transmite datos para la red. Todos los nodos de malla cooperan en la distribución de datos en la red.

Si se agrega una computadora nueva o si se quita la computadora existente, todas las otras computadoras en esa red deben conocer estos dos eventos.

Veamos cómo encaja el patrón de mediador en él.

Fragmento de código:

```
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);
    }
}
```

salida:

```
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
```

Explicación:

1. `Eagle` se agrega a la red en primer lugar a través del evento de registro. No hay notificaciones a ningún otro compañero ya que Eagle es el primero.
2. Cuando se agrega `Ostrich` a la red, se notifica a `Eagle` : la línea 1 de la salida se procesa ahora.
3. Cuando se agrega `Penguin` a la red, tanto `Eagle` como `Ostrich` han sido notificados: la línea 2 y la línea 3 de la salida se procesan ahora.
4. Cuando `Eagle` abandonó la red a través del evento de desregistro, se notificó a `Ostrich` y `Penguin` . La línea 4 y la línea 5 de salida se representan ahora.

Lea Patrón mediador en línea: https://riptutorial.com/es/design-patterns/topic/6184/patron-mediador

Capítulo 27: Patrón prototipo

Introducción

El patrón de prototipo es un patrón de creación que crea nuevos objetos mediante la clonación de un objeto prototipo existente. El patrón prototipo acelera la creación de instancias de clases cuando la copia de objetos es más rápida.

Observaciones

El patrón prototipo es un patrón de diseño creativo. Se utiliza cuando el tipo de objetos a crear está determinado por una instancia prototípica, que se *"clona"* para producir nuevos objetos.

Este patrón se usa cuando una clase necesita un *"constructor polimórfico (copia)"* .

Examples

Patrón de prototipo (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; }
};
```

Eso permite construir la clase derivada desde un puntero de clase 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();
```

Patrón de prototipo (C #)

El patrón prototipo se puede implementar utilizando la interfaz ICloneable en .NET.

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

Patrón de prototipo (JavaScript)

En los lenguajes clásicos como Java, C # o C ++ comenzamos creando una clase y luego podemos crear nuevos objetos de la clase o podemos extender la clase.

En JavaScript primero creamos un objeto, luego podemos aumentar el objeto o crear nuevos objetos a partir de él. Así que creo, JavaScript demuestra prototipo real que el lenguaje clásico.

Ejemplo:

```
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";
    }
};
```

Aquí, creamos un objeto llamado `Customer` , y luego, sin crear un *nuevo objeto* , extendimos el `Customer object` existente mediante la palabra clave *prototipo* . Esta técnica es conocida como **patrón prototipo** .

Lea Patrón prototipo en línea: https://riptutorial.com/es/design-patterns/topic/5867/patron-prototipo

Capítulo 28: pizarra

Examples

Muestra C

Pizarra.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();

    } }
```

Lea pizarra en línea: https://riptutorial.com/es/design-patterns/topic/6519/pizarra

Capítulo 29: Principio de cierre abierto

Introducción

El principio de cierre abierto establece que el diseño y la escritura del código deben realizarse de manera que se agreguen nuevas funciones con cambios mínimos en el código existente. El diseño debe hacerse de manera que permita agregar nuevas funcionalidades como nuevas clases, manteniendo el código existente sin cambios lo más posible. Las entidades de software como clases, módulos y funciones deben estar abiertas para extensión pero cerradas para modificaciones.

Observaciones

Como todo principio, el principio de cierre es solo un principio. Hacer un diseño flexible implica un tiempo y un esfuerzo adicionales, e introduce un nuevo nivel de abstracción que aumenta la complejidad del código. Por lo tanto, este principio debe aplicarse en aquellas áreas que tienen más probabilidades de ser cambiadas. Hay muchos patrones de diseño que nos ayudan a extender el código sin cambiarlo, por ejemplo, decorator.

Examples

Abrir Cerrar Principio de violación

```
/*
* 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;
    }
}
```

Abrir el soporte Principio de cierre

```
/*
* 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
     }
 }
```

Lea Principio de cierre abierto en línea: https://riptutorial.com/es/design-patterns/topic/9199/principio-de-cierre-abierto

Capítulo 30: Publicar-Suscribir

Examples

Publicar-Suscribir en Java

El editor-suscriptor es un concepto familiar dado el auge de YouTube, Facebook y otros servicios de medios sociales. El concepto básico es que hay un `Publisher` que genera contenido y un `Subscriber` que consume contenido. Cada vez que el `Publisher` genera contenido, se notifica a cada `Subscriber` . `Subscribers` pueden, en teoría, suscribirse a más de un editor.

Por lo general, hay un `ContentServer` que se ubica entre el editor y el suscriptor para ayudar a mediar en la mensajería.

```
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) {
            ...
        }
    }
}
```

Por lo general, el patrón de diseño pub-sub se implementa teniendo en cuenta una vista de multiproceso. Una de las implementaciones más comunes ve a cada `Subscriber` como un hilo separado, con el `ContentServer` gestionando un grupo de hilos

Ejemplo simple de pub-sub en JavaScript

Los editores y suscriptores no necesitan conocerse entre sí. Simplemente se comunican con la ayuda de colas de mensajes.

```
(function () {
        var data;

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

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

Aquí publicamos un evento personalizado llamado **myCustomEvent** y suscribimos en ese evento. Así que no necesitan conocerse.

Lea Publicar-Suscribir en línea: https://riptutorial.com/es/design-patterns/topic/7260/publicar-suscribir

Capítulo 31: Semifallo

Observaciones

El patrón de diseño de Singleton a veces se considera como " *patrón Anti* ". Esto se debe al hecho de que tiene algunos problemas. Tienes que decidir por ti mismo si crees que es apropiado usarlo. Este tema ha sido discutido varias veces en StackOverflow.

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

Examples

Singleton (C #)

Los Singletons se utilizan para garantizar que solo se está creando una instancia de un objeto. El singleton permite que solo se cree una instancia de sí mismo, lo que significa que controla su creación. El singleton es uno de los patrones de diseño de Gang of Four y es un **patrón de creación** .

Patrón Singleton seguro para hilos

```
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 proporciona la siguiente implementación para un singleton perezoso y seguro para subprocesos:

```
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)

Los Singletons en Java son muy similares a C #, ya que ambos lenguajes están orientados a objetos. A continuación se muestra un ejemplo de una clase singleton, donde solo una versión del objeto puede estar activa durante la vida útil del programa (Suponiendo que el programa funcione en un solo hilo)

```
public class SingletonExample {

    private SingletonExample() { }

    private static SingletonExample _instance;

    public static SingletonExample getInstance() {

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

Aquí está la versión segura de hilo de ese programa:

```
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 también tiene un objeto llamado `ThreadLocal` , que crea una única instancia de un objeto en una base hilo por hilo. Esto podría ser útil en aplicaciones donde cada hilo necesita su propia versión del objeto.

```
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();
    }
}
```

Aquí también hay una implementación de **Singleton** con `enum` (que contiene solo un elemento):

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

Cualquier implementación de clase **Enum** asegura que *solo* existirá *una* instancia de cada elemento.

Bill Pugh Singleton patrón

Bill Pugh Singleton Pattern es el enfoque más utilizado para la clase Singleton, ya que no requiere sincronización

```
public class SingletonExample {

    private SingletonExample(){}

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

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

con el uso de una clase estática interna privada, el titular no se carga en la memoria hasta que alguien llame al método getInstance. La solución de Bill Pugh es segura para subprocesos y no requiere sincronización.

Hay más ejemplos de Java singleton en el tema Singletons bajo la etiqueta de documentación de Java.

Singleton (C ++)

Según Wiki : en ingeniería de software, el patrón de singleton es un patrón de diseño que restringe la creación de instancias de una clase a un objeto.

Esto es necesario para crear exactamente un objeto para coordinar acciones en todo el sistema.

```
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 ejemplo práctico en java.

Casos de uso de la vida real para el patrón Singleton;

Si está desarrollando una aplicación cliente-servidor, necesita una única entrada de `ConnectionManager` , que administra el ciclo de vida de las conexiones de clientes.

Las APIs básicas en ConnectionManager:

`registerConnection` : Agregar nueva conexión a la lista de conexiones existente

`closeConnection` : `closeConnection` la conexión ya sea del evento desencadenado por el Cliente o el Servidor

`broadcastMessage` : algunas veces, debe enviar un mensaje a todas las conexiones de clientes

suscritas.

No estoy proporcionando una implementación completa del código fuente ya que el ejemplo será muy largo. A alto nivel, el código será así.

```
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);
        }
    }
}
```

Clase de servidor de muestra:

```
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();
        }
    }

}
```

Otros casos de uso práctico para Singletons:

1. Administración de recursos globales como `ThreadPool, ObjectPool, DatabaseConnectionPool` etc.
2. Servicios centralizados como el `Logging` datos de aplicaciones con diferentes niveles de registro como `DEBUG,INFO,WARN,ERROR` etc.
3. Global `RegistryService` donde se registran diferentes servicios con un componente central en el inicio. Ese servicio global puede actuar como una `Facade` para la aplicación

Ejemplo de C #: Singleton multiproceso

La inicialización estática es adecuada para la mayoría de las situaciones. Cuando su aplicación debe retrasar la creación de instancias, use un constructor no predeterminado o realice otras tareas antes de la creación de instancias, y trabaje en un entorno de multiproceso, necesita una solución diferente. Sin embargo, existen casos en los que no puede confiar en el tiempo de ejecución del lenguaje común para garantizar la seguridad de los subprocesos, como en el ejemplo de Inicialización Estática. En tales casos, debe usar capacidades de lenguaje específicas para garantizar que solo se cree una instancia del objeto en presencia de varios subprocesos. Una de las soluciones más comunes es utilizar el modismo Double-Check Locking [Lea99] para evitar que los hilos separados creen nuevas instancias del singleton al mismo tiempo.

La siguiente implementación permite que solo un subproceso ingrese al área crítica, que el bloque de bloqueo identifica, cuando aún no se ha creado una instancia de Singleton:

```
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;
        }
    }
}
```

Este enfoque garantiza que solo se crea una instancia y solo cuando la instancia es necesaria. Además, la variable se declara volátil para garantizar que la asignación a la variable de instancia se complete antes de poder acceder a la variable de instancia. Por último, este enfoque utiliza una instancia de syncRoot para bloquear, en lugar de bloquear el propio tipo, para evitar puntos muertos.

Este enfoque de bloqueo de doble comprobación resuelve los problemas de concurrencia de hilos al tiempo que evita un bloqueo exclusivo en cada llamada al método de propiedad de instancia. También le permite retrasar la creación de instancias hasta que se acceda por primera vez al objeto. En la práctica, una aplicación rara vez requiere este tipo de implementación. En la mayoría de los casos, el enfoque de inicialización estática es suficiente.

Referencia: MSDN

Expresiones de gratitud

[Gamma95] Gamma, Helm, Johnson y Vlissides. Patrones de diseño: elementos de software orientado a objetos reutilizables. Addison-Wesley, 1995.

[Lea99] Lea, Doug. Programación concurrente en Java, Segunda Edición. Addison-Wesley, 1999.

[Vende03] Vende, Chris. "Sellado chupa". sellsbrothers.com Noticias. Disponible en: http://www.sellsbrothers.com/news/showTopic.aspx?ixTopic=411 .

Singleton (PHP)

Ejemplo 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)
```

Patrón Singleton Design (en general)

Nota: El singleton es un patrón de diseño.
Pero también se considera un anti-patrón.

El uso de un singleton debe considerarse cuidadosamente antes de su uso. Usualmente hay mejores alternativas.

El problema principal con un singleton es el mismo que el problema con las variables globales. Introducen el estado mutable global externo. Esto significa que las funciones que usan un singleton no dependen únicamente de los parámetros de entrada, sino también del estado del singleton. Esto significa que las pruebas pueden verse gravemente comprometidas (difíciles).

Los problemas con singletons pueden mitigarse usándolos junto con los patrones de creación;

para que la creación inicial del singleton pueda ser controlada.

Lea Semifallo en línea: https://riptutorial.com/es/design-patterns/topic/2179/semifallo

Capítulo 32: SÓLIDO

Introducción

¿Qué es SOLID?

SOLID es un acrónimo mnemotécnico (ayuda de memoria). Los principios de Solid deberían ayudar a los desarrolladores de software a evitar los "olores de código" y deberían conducir a un buen código fuente. Un buen código fuente significa en este contexto que el código fuente es fácil de extender y mantener. El enfoque principal de los principios sólidos son las clases.

Que esperar:

¿Por qué deberías aplicar SOLID?

Cómo aplicar los cinco principios SÓLIDOS (ejemplos)

Examples

SRP - Principio de responsabilidad única

La S en SOLID significa Principio de responsabilidad única (SRP).

Responsabilidad significa que en este contexto las razones para cambiar, por lo que el principio establece que una clase solo debe tener una razón para cambiar.

Robert C. Martin lo declaró (durante su conferencia en la escuela de administración de Yale en 10 de septiembre de 2014) de la siguiente manera:

> También podría decir, no ponga funciones que cambien por diferentes motivos en la misma clase.

o

> No mezcle preocupaciones en sus clases

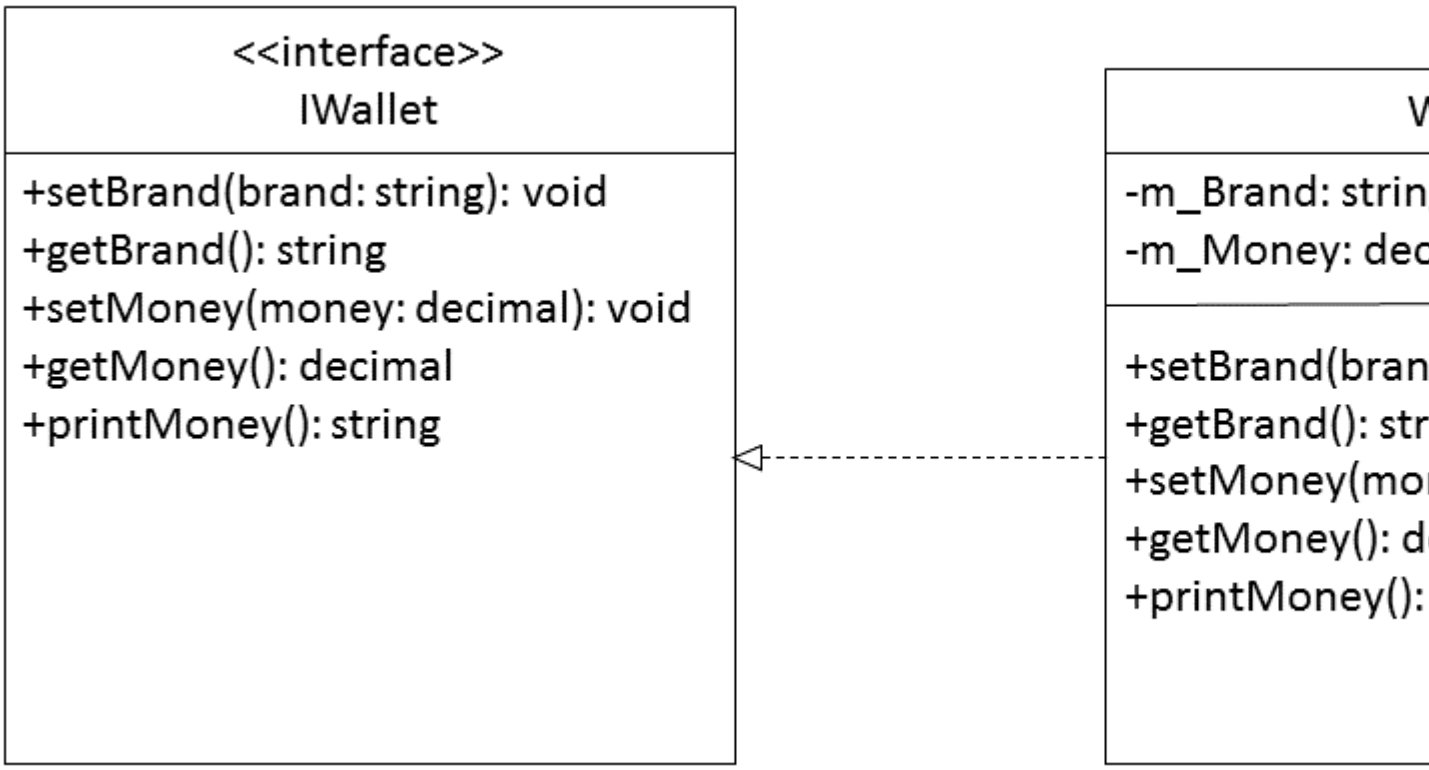
Razón para aplicar el SRP:

Cuando cambia una clase puede afectar la funcionalidad relacionada con otras responsabilidades de la clase. Mantener las responsabilidades en un nivel bajo minimiza el riesgo de efectos secundarios.

Mal ejemplo

Tenemos una interfaz IWallet y una clase Wallet que implementa IWallet. The Wallet tiene nuestro dinero y la marca, además, debe imprimir nuestro dinero como una representación de cadena. La clase es utilizada por

1. un servicio web
2. un escritor de texto que imprime el dinero en euros en un archivo de texto.



El SRP se viola aquí porque tenemos dos preocupaciones:

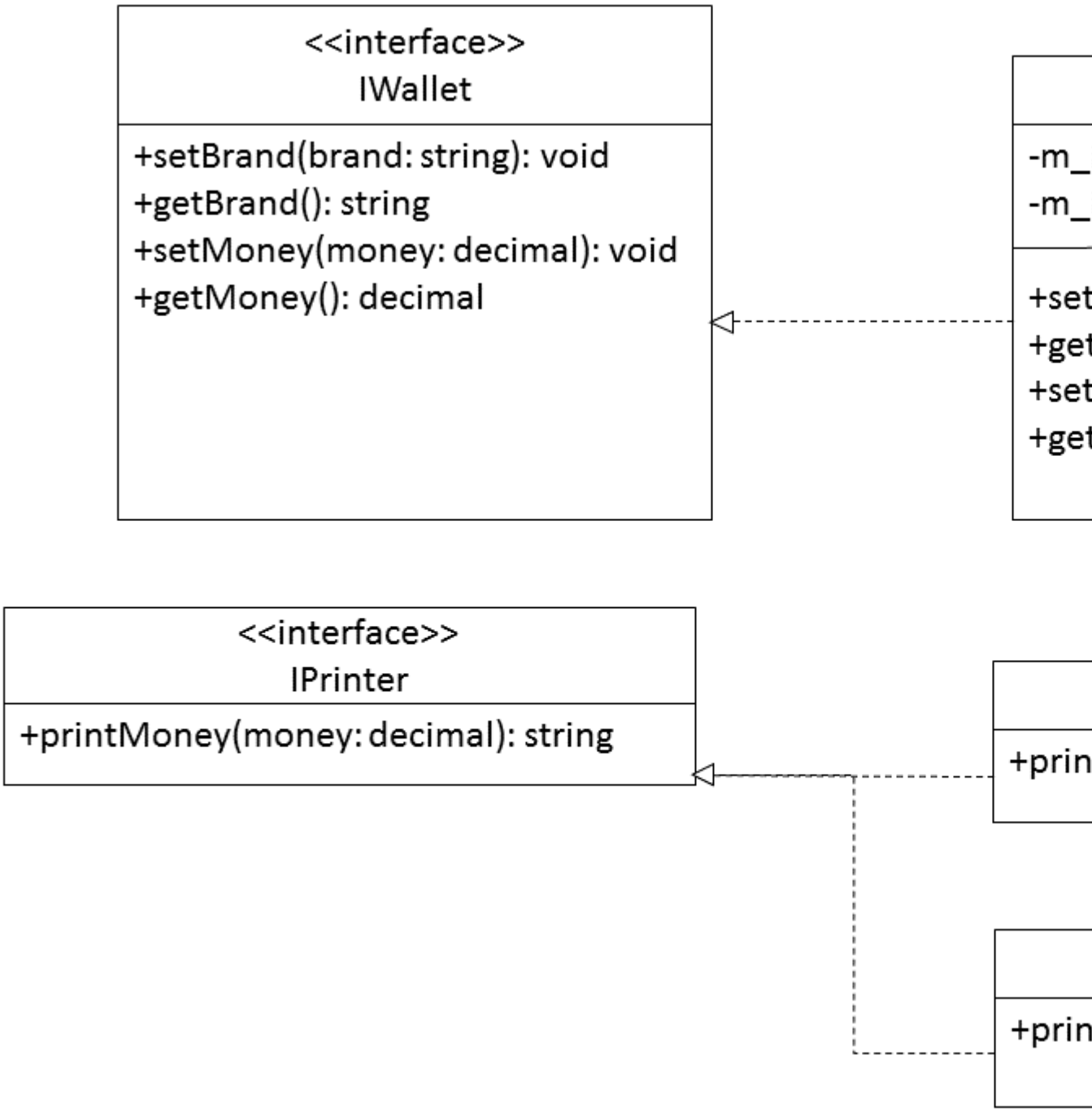
1. El almacenamiento del dinero y la marca.
2. La representación del dinero.

Código de ejemplo C #

```
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();
    }
}
```

Buen ejemplo



Para evitar la violación del SRP, eliminamos el método `printMoney` de la clase Wallet y lo colocamos en una clase de impresora. La clase de Impresora ahora es responsable de la impresión y la Cartera ahora es responsable del almacenamiento de los valores.

Código de ejemplo C #

```
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;
    }
}
```

Lea SÓLIDO en línea: https://riptutorial.com/es/design-patterns/topic/8651/solido

Creditos

S. No	Capítulos	Contributors
1	Comenzando con los patrones de diseño	Community, Ekin, Mateen Ulhaq, meJustAndrew, Sahan Serasinghe, Saurabh Sarode, Stephen C, Tim, ɬolɐəz əɥɬ qoq
2	Adaptador	avojak, Ben Rhys-Lewis, Daniel Käfer, deHaar, Thijs Riezebeek
3	Cadena de responsabilidad	Ben Rhys-Lewis
4	carga lenta	Adhikari Bishwash
5	Fábrica	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
6	Fachada	Kritner, Makoto, Ravindra babu
7	Inyección de dependencia	Kritner, matiaslauriti, user2321864
8	Método de fábrica estático	abbath, Bongo
9	Método de plantilla	meJustAndrew, Ravindra babu
10	Monostato	skypjack
11	Multiton	Kid Binary
12	MVC, MVVM, MVP	Daniel LIn, Jompa234, Stephen C, user1223339
13	Observador	Arif, user2321864, uzilan
14	Patrón compuesto	Krzysztof Branicki
15	Patrón de comando	matiaslauriti, Ravindra babu, Vasiliy Vlasov
16	Patrón de constructor	Alexey Groshev, Arif, Daniel Käfer, fgb, Kyle Morgan, Ravindra babu, Sujit Kamthe, uzilan, Vasiliy Vlasov, yitzih
17	Patrón de diseño del objeto de acceso a datos (DAO)	Pritam Banerjee

18	Patrón de estrategia	Aseem Bansal, dimitrisli, fabian, M.S. Dousti, Marek Skiba, matiaslauriti, Ravindra babu, Shog9, SjB, Stephen C, still_learning, uzilan
19	Patrón de iterador	bw_üezi, Dave Ranjan, Jeeter, Stephen C
20	Patrón de objeto nulo	Jarod42, weston
21	Patrón de puente	Mark, Ravindra babu, Vasiliy Vlasov
22	Patrón de repositorio	bolt19, Leifb
23	Patrón de visitante	Daniel Käfer, Jarod42, Loki Astari, Ravindra babu, Stephen Leppik, Vasiliy Vlasov
24	Patrón decorador	Arif, Krzysztof Branicki, matiaslauriti, Ravindra babu
25	Patrón mediador	Ravindra babu, Vasiliy Vlasov
26	Patrón prototipo	Arif, Jarod42, user2321864
27	pizarra	Leonidas Menendez
28	Principio de cierre abierto	Mher Didaryan
29	Publicar-Suscribir	Arif, Jeeter, Stephen C
30	Semifallo	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
31	SÓLIDO	Bongo