EBook Gratuito

APPRENDIMENTO
Design patterns

Free unaffiliated eBook created from
Stack Overflow contributors.

#design-patterns

Sommario

Di ... 1

Capitolo 1: Iniziare con i modelli di progettazione ... 2

Osservazioni ... 2

Examples ... 2

introduzione ... 2

Capitolo 2: Adattatore ... 4

Examples ... 4

Adapter Pattern (PHP) ... 4

Adattatore (Java) ... 4

Esempio di Java ... 5

Adattatore (UML e situazione di esempio) ... 6

Capitolo 3: Apri Close Principle ... 11

introduzione ... 11

Osservazioni ... 11

Examples ... 11

Apri la violazione del principio di chiusura ... 11

Apri il supporto del principio di chiusura ... 12

Capitolo 4: carico pigro ... 13

introduzione ... 13

Examples ... 13

Caricamento pigro JAVA ... 13

Capitolo 5: Catena di responsabilità ... 15

Examples ... 15

Esempio di Chain of Responsibility (Php) ... 15

Capitolo 6: Fabbrica ... 17

Osservazioni ... 17

Examples ... 17

Fabbrica semplice (Java) ... 17

Fabbrica astratta (C ++) ... 18

Semplice esempio di Factory che utilizza un IoC (C #) ... 20

Una fabbrica astratta ... 22
Esempio di fabbrica implementando il metodo Factory (Java) ... 23
Fabbrica di pesi volanti (C #) ... 27
Metodo di fabbrica ... 28
Capitolo 7: Facciata ... 29
Examples ... 29
Facciata del mondo reale (C #) ... 29
Esempio di facciata in java ... 29
Capitolo 8: Iniezione di dipendenza ... 33
introduzione ... 33
Osservazioni ... 33
Examples ... 34
Iniezione setter (C #) ... 34
Iniezione costruttore (C #) ... 34
Capitolo 9: lavagna ... 36
Examples ... 36
Campione C # ... 36
Capitolo 10: Metodo del modello ... 40
Examples ... 40
Implementazione del metodo template in java ... 40
Capitolo 11: Metodo di fabbrica statico ... 44
Examples ... 44
Metodo di fabbrica statico ... 44
Nascondere l'accesso diretto al costruttore ... 44
Metodo statico di fabbrica C # ... 45
Capitolo 12: Modello composito ... 47
Examples ... 47
Logger composito ... 47
Capitolo 13: Modello composito ... 49
introduzione ... 49
Osservazioni ... 49

Examples ... 49
pseudocodice per un gestore di file stupido ... 49
Capitolo 14: Modello costruttore ... 51
Osservazioni ... 51
Examples ... 51
Builder Pattern / C # / Fluent Interrface ... 51
Builder Pattern / Java Implementation ... 52
Modello di generatore in Java con composizione ... 54
Java / Lombok ... 57
Modello di generatore avanzato con espressione Lambda di Java 8 ... 58
Capitolo 15: Modello del mediatore ... 61
Examples ... 61
Esempio di modello di mediatore in java ... 61
Capitolo 16: Modello di comando ... 64
Examples ... 64
Esempio di modello di comando in Java ... 64
Capitolo 17: Modello di ponte ... 67
Examples ... 67
Implementazione del pattern bridge in java ... 67
Capitolo 18: Modello di prototipo ... 70
introduzione ... 70
Osservazioni ... 70
Examples ... 70
Prototype Pattern (C ++) ... 70
Modello di prototipo (C #) ... 71
Prototype Pattern (JavaScript) ... 71
Capitolo 19: Modello di strategia ... 73
Examples ... 73
Nascondere i dettagli di implementazione della strategia ... 73
Esempio di modello di strategia in java con classe Context ... 74
Modello di strategia senza classe di contesto / Java ... 76
Utilizzo di interfacce funzionali Java 8 per implementare il modello di strategia ... 77

La versione Java classica ... 77
Utilizzo di interfacce funzionali Java 8 ... 78
Strategia (PHP) ... 79
Capitolo 20: Modello di visitatore ... 81
Examples ... 81
Esempio di pattern Visitor in C ++ ... 81
Esempio di modello di visitatore in java ... 83
Esempio di visitatore in C ++ ... 86
Attraversando oggetti di grandi dimensioni ... 87
Capitolo 21: Modello oggetto nullo ... 89
Osservazioni ... 89
Examples ... 89
Pattern di oggetto nullo (C ++) ... 89
Oggetto Null Java che utilizza enum ... 90
Capitolo 22: Monostate ... 92
Osservazioni ... 92
Examples ... 92
Il modello Monostate ... 92
Gerarchie basate su monostazione ... 93
Capitolo 23: Motivo decorativo ... 95
introduzione ... 95
Parametri ... 95
Examples ... 95
VendingMachineDecorator ... 95
Caching Decorator ... 99
Capitolo 24: multiton ... 101
Osservazioni ... 101
Examples ... 101
Pool of Singletons (esempio PHP) ... 101
Registro di Singletons (esempio PHP) ... 102
Capitolo 25: MVC, MVVM, MVP ... 104

Osservazioni ... 104
Examples ... 104
Model View Controller (MVC) ... 104
Model View ViewModel (MVVM) ... 105
Capitolo 26: Osservatore ... 109
Osservazioni ... 109
Examples ... 109
Observer / Java ... 109
Osservatore che utilizza IObservable e IObserver (C #) ... 111
Capitolo 27: Pattern del repository ... 113
Osservazioni ... 113
Examples ... 113
Repository di sola lettura (C #) ... 113
Le interfacce ... 113
Un esempio di implementazione che utilizza ElasticSearch come tecnologia (con NEST) ... 113
Pattern del repository utilizzando Entity Framework (C #) ... 114
Capitolo 28: Pattern di progettazione Data Access Object (DAO) ... 117
Examples ... 117
Data Access Object Modello di progettazione J2EE con Java ... 117
Capitolo 29: Pattern Iterator ... 120
Examples ... 120
Il pattern Iterator ... 120
Capitolo 30: Publish-Subscribe ... 122
Examples ... 122
Pubblica-Iscriviti in Java ... 122
Semplice esempio pub-sub in JavaScript ... 123
Capitolo 31: Singleton ... 124
Osservazioni ... 124
Examples ... 124
Singleton (C #) ... 124
Pattern Singleton sicuro per thread ... 124

Singleton (Java) ... 125
Singleton (C ++) ... 127
Esempio pratico Lazy Singleton in java ... 127
C # Esempio: Singleton multithread ... 129
Singleton (PHP) ... 130
Modello Singleton Design (in generale) ... 131
Capitolo 32: SOLIDO ... 132
introduzione ... 132
Examples ... 132
SRP - Principio della singola responsabilità ... 132
Titoli di coda ... 137

Di

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

Capitolo 1: Iniziare con i modelli di progettazione

Osservazioni

Questa sezione fornisce una panoramica di quali sono i modelli di progettazione e perché uno sviluppatore potrebbe volerlo utilizzare. Gli esempi possono fornire una rappresentazione grafica del modello, uno scenario costituito da un problema dato un contesto in cui è possibile utilizzare un modello e menzionare possibili compromessi.

Dovrebbe anche menzionare eventuali soggetti di grandi dimensioni all'interno di schemi di progettazione e collegarsi agli argomenti correlati. Poiché la Documentazione per i modelli di progettazione è nuova, potrebbe essere necessario creare versioni iniziali di tali argomenti correlati.

Examples

introduzione

Secondo Wikipedia :

> [A] *modello di progettazione software* è una soluzione generale riutilizzabile per un problema che si verifica comunemente in un determinato contesto nella progettazione del software. Non è un progetto finito che può essere trasformato direttamente in codice sorgente o macchina. È una descrizione o un modello per come risolvere un problema che può essere utilizzato in molte situazioni diverse. I modelli di progettazione sono le migliori pratiche formalizzate che il programmatore può utilizzare per risolvere problemi comuni durante la progettazione di un'applicazione o di un sistema.
>
> (Estratto: 13-10-2016)

Esistono molti modelli di progettazione software riconosciuti e ne vengono proposti di nuovi su base regolare. Altri argomenti coprono molti dei modelli più comuni e l'articolo di Wikipedia fornisce un elenco più esteso.

Allo stesso modo, ci sono diversi modi per classificare i modelli di progettazione, ma la classificazione originale è:

- **Modelli di creazione** : Factory , Builder , Singleton , ecc.
- **Modelli strutturali** : adattatore , composito , proxy, ecc.
- **Modelli comportamentali** : Iterator , Strategia , Visitatore , ecc.
- **Pattern di concorrenza** : ActiveObject, Monitor, ecc.

L'idea dei modelli di progettazione è stata estesa a *modelli di progettazione specifici del dominio*

per domini quali la progettazione dell'interfaccia utente, la visualizzazione dei dati, la progettazione sicura, il web design e la progettazione di modelli di business.

Infine, esiste un concetto correlato chiamato *pattern dell'architettura software* che è descritto come l'analogo per i pattern di progettazione applicati alle architetture software.

Leggi Iniziare con i modelli di progettazione online: https://riptutorial.com/it/design-patterns/topic/1012/iniziare-con-i-modelli-di-progettazione

Capitolo 2: Adattatore

Examples

Adapter Pattern (PHP)

Un esempio del mondo reale che utilizza un esperimento scientifico in cui determinate routine vengono eseguite su diversi tipi di tessuto. La classe contiene di default due funzioni per ottenere separatamente il tessuto o la routine. In una versione successiva l'abbiamo quindi adattato utilizzando una nuova classe per aggiungere una funzione che ottiene entrambi. Ciò significa che non abbiamo modificato il codice originale e quindi non corriamo alcun rischio di rompere la nostra classe esistente (e nessuna ripetizione).

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

Adattatore (Java)

Supponiamo che nella tua base di codice attuale esista `MyLogger` interfaccia `MyLogger` modo:

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

Diciamo che hai creato alcune implementazioni concrete di questi, come `MyFileLogger` e `MyConsoleLogger` .

Hai deciso di voler utilizzare un framework per il controllo della connettività Bluetooth della tua

applicazione. Questo framework contiene un `BluetoothManager` con il seguente costruttore:

```
class BluetoothManager {
    private FrameworkLogger logger;

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

Il `BluetoothManager` accetta anche un logger, che è fantastico! Tuttavia, si aspetta un logger di cui l'interfaccia è stata definita dal framework e che hanno usato l'overloading del metodo invece di nominare le loro funzioni in modo diverso:

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

Hai già un sacco di implementazioni `MyLogger` che vorresti riutilizzare, ma non si adattano all'interfaccia di `FrameworkLogger` . È qui che entra in gioco il modello di design dell'adattatore:

```
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);
    }
}
```

Definendo una classe adattatore che implementa l'interfaccia `FrameworkLogger` e accetta un'implementazione `MyLogger` la funzionalità può essere mappata tra le diverse interfacce. Ora è possibile utilizzare `BluetoothManager` con tutte le implementazioni di `MyLogger` modo:

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

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

Esempio di Java

Un grande esempio esistente del pattern Adapter possono essere trovate nelle SWT

MouseListener e MouseAdapter classi.

L'interfaccia MouseListener ha il seguente aspetto:

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

Ora immagina uno scenario in cui stai costruendo un'interfaccia utente e aggiungendo questi listener, ma la maggior parte delle volte non ti interessa nulla se non quando qualcosa viene cliccato (mouseUp). Non vorrai creare costantemente implementazioni vuote:

```
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
    }

});
```

Invece, possiamo usare MouseAdapter:

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

Fornendo implementazioni predefinite vuote, siamo liberi di ignorare solo i metodi che ci interessano dall'adattatore. Seguendo l'esempio precedente:

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

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

});
```

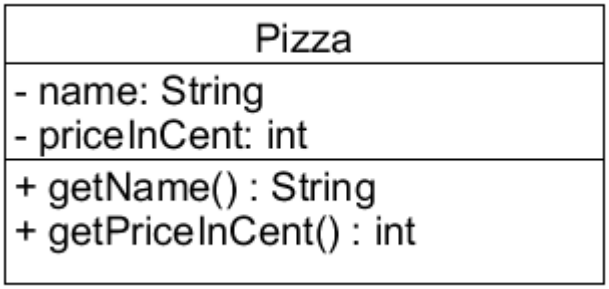
Adattatore (UML e situazione di esempio)

Per rendere l'uso del modello di adattatore e il tipo di situazione in cui può essere applicato più

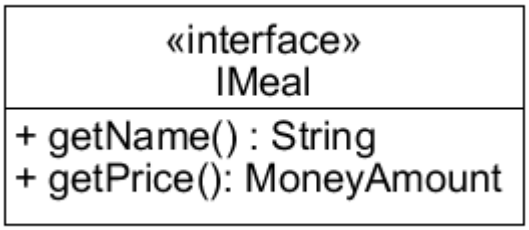
immaginabile, qui viene fornito un esempio piccolo, semplice e molto concreto. Non ci sarà codice qui, solo UML e una descrizione della situazione di esempio e il suo problema. A dire il vero, il contenuto UML è scritto come Java. (Bene, il testo suggerito diceva "I buoni esempi sono per lo più codice", penso che i modelli di design siano abbastanza astratti da poter essere introdotti in un modo diverso).

In generale, il modello dell'adattatore è una soluzione adeguata per una situazione in cui si hanno interfacce incompatibili e nessuna di esse può essere riscritta direttamente.

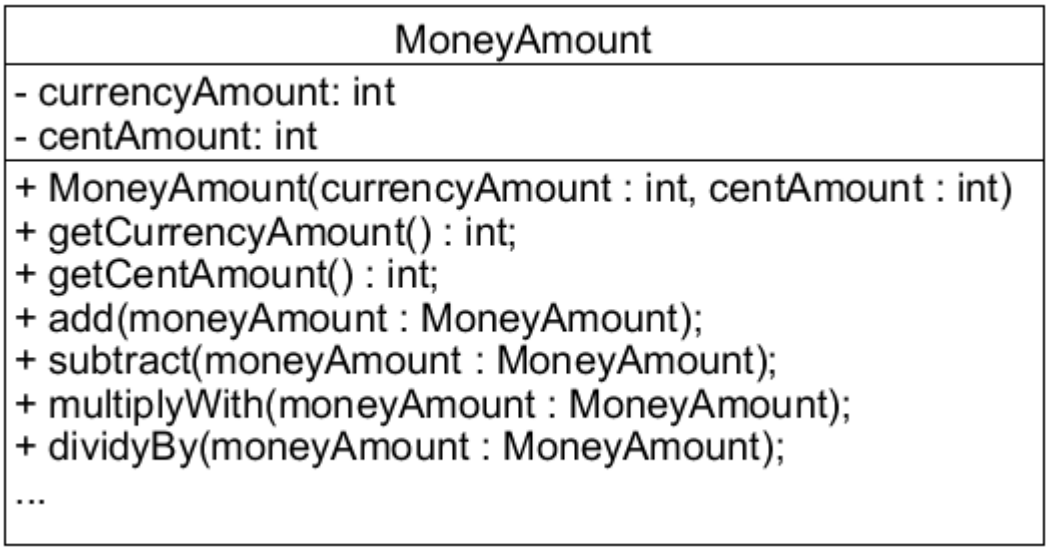
Immagina di gestire un piccolo servizio di consegna della pizza. I clienti possono ordinare online sul tuo sito web e hai un piccolo sistema che utilizza una `Pizza` classe per rappresentare le tue pizze e calcolare le bollette, i rapporti fiscali e altro. Il prezzo delle tue pizze viene indicato come un singolo intero che rappresenta il prezzo in centesimi (della valuta di tua scelta).



Il tuo servizio di consegna sta funzionando alla grande, ma a un certo punto non puoi più gestire il crescente numero di clienti da solo ma desideri comunque espanderlo. Decidi di aggiungere le tue pizze al menu di un grande servizio di consegna meta online. Offrono molti pasti diversi, non solo le pizze, quindi il loro sistema fa un uso più `IMeal` dell'astrazione e ha un `IMeal` interfaccia `IMeal` rappresenta i pasti accompagnati da una classe `MoneyAmount` rappresenta denaro.



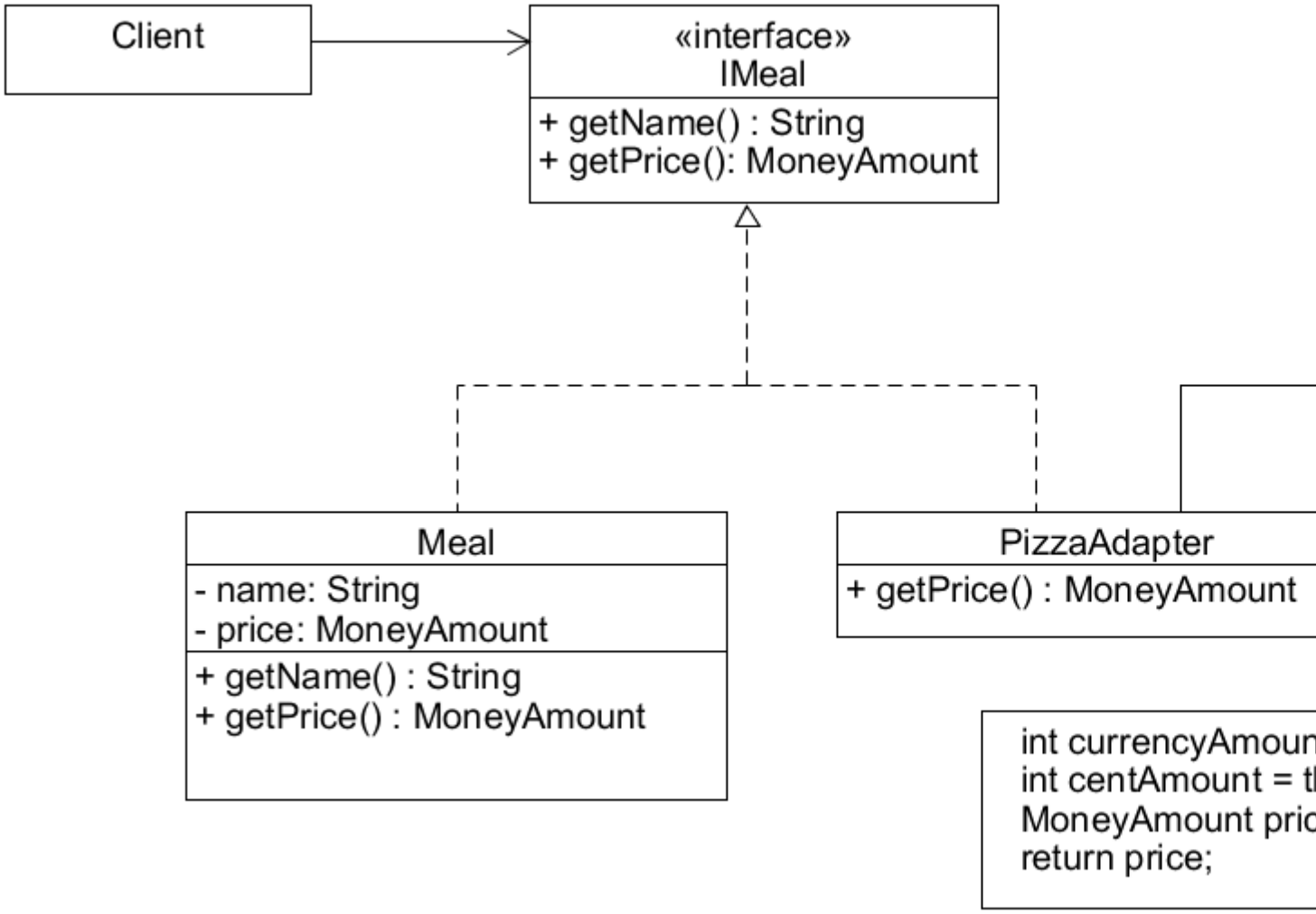
`MoneyAmount` consiste di due interi come input, uno per l'importo (o una valuta casuale) prima della virgola e uno per l'importo centesimo da 0 a 99 dopo la virgola;



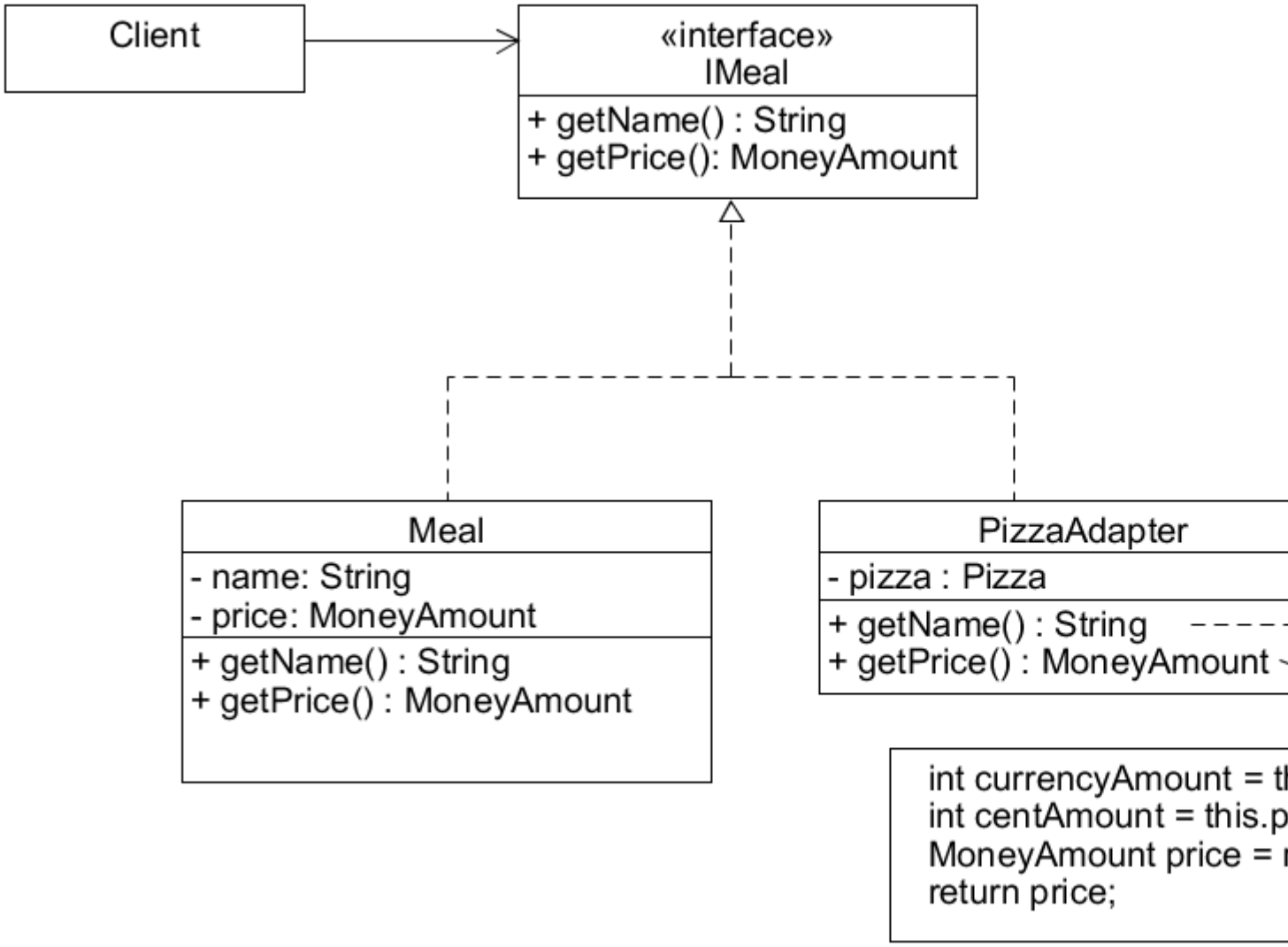
A causa del fatto che il prezzo della tua `Pizza` è un numero intero singolo che rappresenta il prezzo totale come una quantità di centesimo (> 99), non è compatibile con `IMeal` . Questo è il punto in cui entra in gioco il modello di adattatore: nel caso in cui ci vorrebbe troppo tempo per modificare il proprio sistema o crearne uno nuovo e si deve implementare un'interfaccia incompatibile, si consiglia di applicare il patttern dell'adattatore.

Esistono due modi per applicare il modello: adattatore di classe e adattatore di oggetti.

Entrambi hanno in comune che un adattatore (`PizzaAdapter`) funziona come una sorta di traduttore tra la nuova interfaccia e l'adaptee (`Pizza` in questo esempio). L'adattatore implementa la nuova interfaccia (`IMeal`) e quindi eredita da `Pizza` e converte il proprio prezzo da un numero intero a due (adattatore classe)



o ha un oggetto di tipo `Pizza` come attributo e converte i valori di quello (oggetto adattatore).



Applicando il modello dell'adattatore, sarà possibile "tradurre" tra interfacce incompatibili.

Leggi Adattatore online: https://riptutorial.com/it/design-patterns/topic/4580/adattatore

Capitolo 3: Apri Close Principle

introduzione

L'Open Close Principle afferma che la progettazione e la scrittura del codice dovrebbero essere fatte in modo tale da aggiungere nuove funzionalità con modifiche minime nel codice esistente. Il design dovrebbe essere fatto in modo da consentire l'aggiunta di nuove funzionalità come nuove classi, mantenendo il più possibile il codice esistente invariato. Le entità software come classi, moduli e funzioni dovrebbero essere aperte per estensione ma chiuse per modifiche.

Osservazioni

Come ogni principio Open Close Principle è solo un principio. Realizzare un design flessibile richiede tempo e sforzi aggiuntivi per esso e introduce un nuovo livello di astrazione che aumenta la complessità del codice. Quindi questo principio dovrebbe essere applicato in quelle aree che sono più suscettibili di essere modificate. Esistono molti modelli di progettazione che ci aiutano ad estendere il codice senza cambiarlo, ad esempio il decoratore.

Examples

Apri la violazione del principio di chiusura

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

Apri il supporto del principio di chiusura

```
/*
* 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
     }
 }
```

Leggi Apri Close Principle online: https://riptutorial.com/it/design-patterns/topic/9199/apri-close-principle

Capitolo 4: carico pigro

introduzione

il caricamento ansioso è costoso o l'oggetto da caricare potrebbe non essere affatto necessario

Examples

Caricamento pigro JAVA

Chiama da main ()

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

Heavy.class

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

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

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

HolderNaive.class

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

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

```
  private Heavy heavy;

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

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

Leggi carico pigro online: https://riptutorial.com/it/design-patterns/topic/9951/carico-pigro

Capitolo 5: Catena di responsabilità

Examples

Esempio di Chain of Responsibility (Php)

Un metodo chiamato in un oggetto si muoverà su per la catena di oggetti finché non si troverà uno che può gestire correttamente la chiamata. Questo particolare esempio utilizza esperimenti scientifici con funzioni che possono ottenere solo il titolo dell'esperimento, l'identificazione degli esperimenti o il tessuto utilizzato nell'esperimento.

```
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 }
}
```

Leggi Catena di responsabilità online: https://riptutorial.com/it/design-patterns/topic/6083/catena-di-responsabilita

Capitolo 6: Fabbrica

Osservazioni

> Fornire un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete.
>
> *- GOF 1994*

Examples

Fabbrica semplice (Java)

Una factory riduce l'accoppiamento tra codice che deve creare oggetti dal codice di creazione dell'oggetto. La creazione dell'oggetto non viene effettuata in modo esplicito chiamando un costruttore di classi ma chiamando una funzione che crea l'oggetto per conto del chiamante. Un semplice esempio Java è il seguente:

```
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);
  }
}
```

In questo esempio, l'utente fornisce solo qualche indicazione su ciò di cui ha bisogno e la fabbrica è libera di costruire qualcosa di appropriato. È **un'inversione di dipendenza** : l'implementatore di `Car` concept è libero di restituire `Car` concreta appropriata richiesta dall'utente che a sua volta non conosce i dettagli dell'oggetto concreto costruito.

Questo è un semplice esempio di come funziona la fabbrica, ovviamente in questo esempio è sempre possibile istanziare classi concrete; ma si può prevenirlo nascondendo le classi concrete in un pacchetto, in modo tale che l'utente sia costretto a usare la fabbrica.

.Net Fiddle per esempio precedente.

Fabbrica astratta (C ++)

Il modello di **fabbrica astratto** fornisce un modo per ottenere una collezione coerente di oggetti attraverso una serie di funzioni di fabbrica. Come per ogni modello, l'accoppiamento si riduce astraendo il modo in cui viene creato un insieme di oggetti, in modo che il codice utente non sia a conoscenza dei molti dettagli degli oggetti di cui ha bisogno.

Il seguente esempio C ++ illustra come ottenere diversi tipi di oggetti della stessa (ipotetica) GUI:

```
#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);
}
```

Se l'eseguibile generato è denominato `abstractfactory` output può dare:

```
$ ./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
```

Semplice esempio di Factory che utilizza un IoC (C #)

Le fabbriche possono essere utilizzate insieme alle librerie Inversion of Control (IoC).

- Il tipico caso d'uso per una tale fabbrica è quando vogliamo creare un oggetto basato su parametri che non sono noti fino al momento dell'esecuzione (come l'utente corrente).
- In questi casi a volte può essere difficile (se non impossibile) configurare la libreria IoC da sola per gestire questo tipo di informazioni contestuali sul runtime, in modo che possiamo eseguire il wrapping in una fabbrica.

Esempio

- Supponiamo di avere una classe `User` , le cui caratteristiche (ID, livello di sicurezza, ecc.) Sono sconosciute fino al runtime (poiché l'utente corrente potrebbe essere chiunque utilizzi l'applicazione).
- Dobbiamo prendere l'utente corrente e ottenere un `ISecurityToken` per loro, che può quindi essere utilizzato per verificare se l'utente è autorizzato a eseguire determinate azioni o meno.
- L'implementazione di ISecurityToken varierà a seconda del livello dell'utente - in altre parole, ISecurityToken utilizza il *polimorfismo* .

In questo caso, abbiamo due implementazioni, che utilizzano anche *Marker Interfaces* per semplificare l'identificazione con la libreria IoC; la libreria IoC in questo caso è appena costituita e identificata dall'astrazione `IContainer` .

> Si noti inoltre che molte fabbriche IoC moderne hanno funzionalità native o plug-in che consentono l'auto-creazione di fabbriche oltre a evitare la necessità di interfacce marker come mostrato di seguito; tuttavia, poiché non tutti lo fanno, questo esempio si rivolge a un concetto di funzionalità comune semplice e più basso.

```
//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;
     }
}
```

Successivamente creeremo uno stabilimento `SecurityToken` , che avrà come dipendenza il nostro `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 volta registrati con `IContainer` :

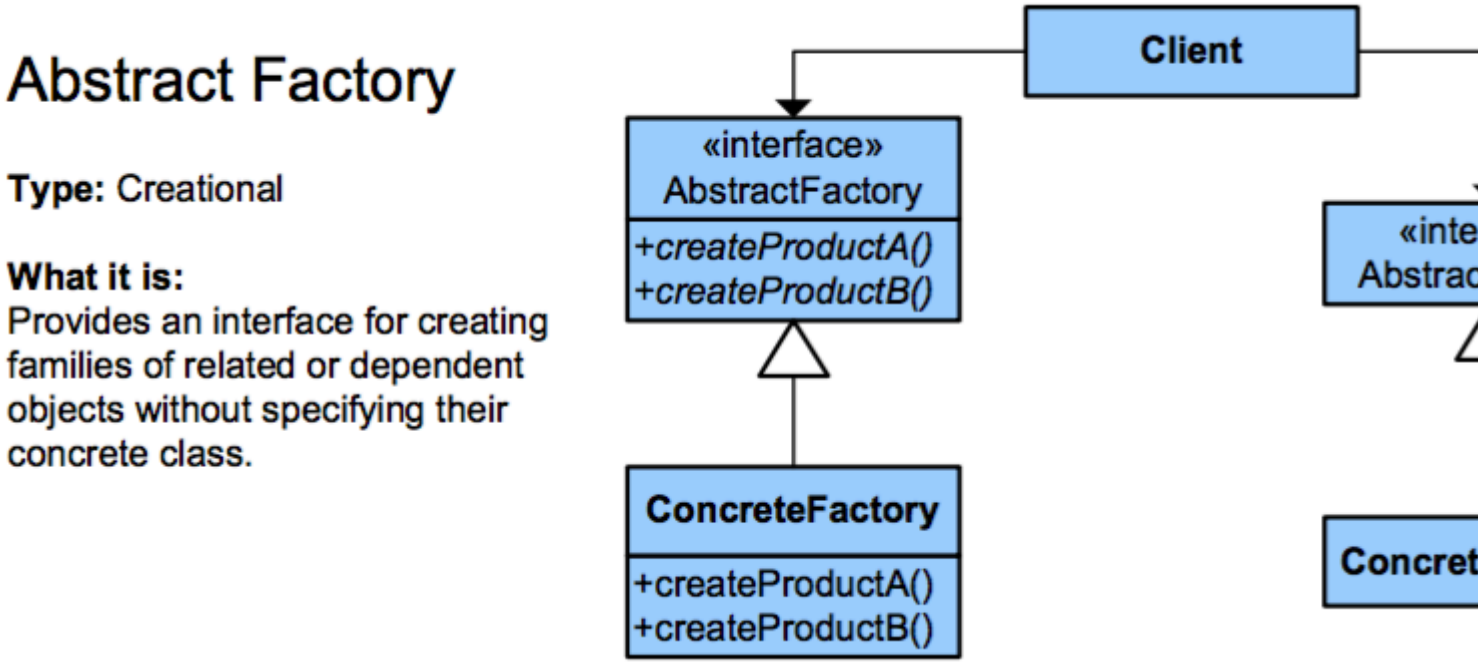
```
IContainer.For<SecurityTokenFactory>().Use<SecurityTokenFactory>().Singleton(); //we only need
a single instance per app
IContainer.For<IBasicToken>().Use<BasicToken>().PerRequest(); //we need an instance per-
request
IContainer.For<ISuperToken>().Use<SuperToken>().PerRequest();//we need an instance per-request
```

il codice che consuma può usarlo per ottenere il token corretto in fase di runtime:

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

> In questo modo beneficiamo dell'incapsulamento fornito dallo stabilimento e anche dalla gestione del ciclo di vita fornita dalla libreria IoC.

Una fabbrica astratta



Il seguente modello di progettazione è classificato come un modello creativo.

Una fabbrica astratta viene utilizzata per fornire un'interfaccia per la creazione di famiglie di oggetti correlati, senza specificare classi concrete e può essere utilizzata per nascondere classi specifiche della piattaforma.

```
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();
    }
}
```

Quindi sarebbe utilizzato un fornitore / produttore di qualche tipo che passerebbe informazioni che gli consentirebbero di restituire il tipo corretto di implementazione di fabbrica:

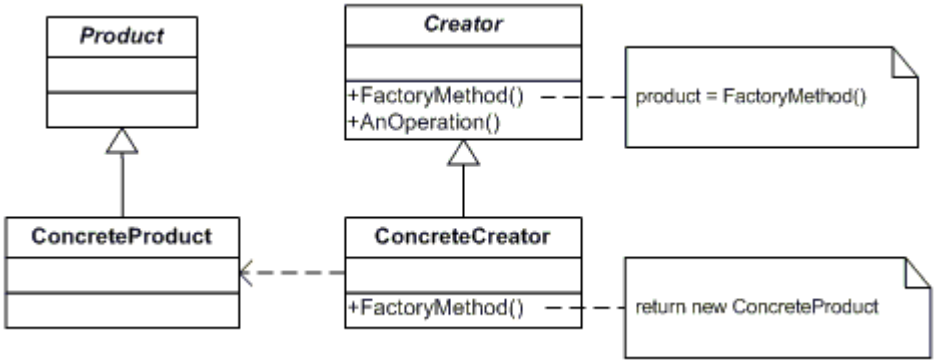
```
public final class FactorySupplier {

    // 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;
    }
}
```

Esempio di fabbrica implementando il metodo Factory (Java)

Intento:

Definisci un'interfaccia per creare un oggetto, ma lascia che le classi secondarie decidano quale classe istanziare. Metodo di fabbrica consente a una classe di rinviare l'istanzazione alle sottoclassi.

Diagramma UML:



Prodotto: definisce un'interfaccia degli oggetti creati dal metodo Factory.

ConcreteProduct: implementa l'interfaccia del prodotto

Creatore: dichiara il metodo di fabbrica

ConcreateCreator: implementa il metodo Factory per restituire un'istanza di ConcreteProduct

Dichiarazione di problema: creare una fabbrica di giochi utilizzando i metodi di fabbrica, che definisce l'interfaccia di gioco.

Snippet di codice:

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

produzione:

```
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
```

Questo esempio mostra una classe `Factory` implementando un `FactoryMethod` .

1. `Game` è l'interfaccia per tutti i tipi di giochi. Definisce il metodo complesso: `createGame()`
2. `Chess, Ludo, Checkers` sono diverse varianti di giochi, che forniscono implementazione per `createGame()`
3. `public Game getGame(String gameName)` è `FactoryMethod` nella classe `IGameFactory`
4. `GameFactory` pre-crea diversi tipi di giochi nel costruttore. Implementa il metodo di fabbrica `IGameFactory` .
5. Il nome del gioco viene passato come argomento della riga di comando a `NotStaticFactoryDemo`
6. `getGame` in `GameFactory` accetta un nome di gioco e restituisce l'oggetto di `Game` corrispondente.

Quando usare:

1. ***Fabbrica*** : quando non si desidera esporre la logica di istanziazione degli oggetti al client / chiamante
2. ***Fabbrica astratta*** : quando si desidera fornire un'interfaccia alle famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete
3. ***Metodo di fabbrica:*** per definire un'interfaccia per la creazione di un oggetto, ma lasciare che le sottoclassi decidano quale classe istanziare

Confronto con altri modelli creativi:

1. Il design inizia con il **metodo Factory** (proliferano le sottoclassi meno complicate, più

personalizzabili) e si evolve verso **Abstract Factory, Prototype o Builder** (più flessibile, più complesso) quando il designer scopre dove è necessaria più flessibilità

2. **Le** classi **Abstract Factory** sono spesso implementate con **Metodi Factory** , ma possono anche essere implementate usando **Prototype**

Riferimenti per ulteriori letture: modelli di progettazione di Sourcemaking

Fabbrica di pesi volanti (C #)

In parole semplici:

Una fabbrica Flyweight che per una data chiave già conosciuta darà sempre lo stesso oggetto di risposta. Per le nuove chiavi creerà l'istanza e la restituirà.

Usando la fabbrica:

```
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);
```

Implementazione:

```
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>();
}
```

Note extra

Raccomanderei di aggiungere a questa soluzione l'uso di un `IoC Container` (come spiegato in un altro esempio qui) anziché creare le proprie nuove istanze. Si può fare aggiungendo una nuova registrazione per `TResult` al contenitore e poi risolvendo da essa (invece del `dictionary` nell'esempio).

Metodo di fabbrica

Il pattern del metodo Factory è un pattern creazionale che astrae la logica di istanziazione di un oggetto al fine di separare il codice client da esso.

Quando un metodo factory appartiene a una classe che è un'implementazione di un altro pattern factory come Abstract factory , di solito è più appropriato fare riferimento al pattern implementato da quella classe piuttosto che al pattern del metodo Factory.

Lo schema del metodo Factory viene più comunemente utilizzato quando si descrive un metodo factory che appartiene a una classe che non è principalmente una factory.

Ad esempio, può essere vantaggioso posizionare un metodo factory su un oggetto che rappresenta un concetto di dominio se quell'oggetto incapsula uno stato che semplificherebbe il processo di creazione di un altro oggetto. Un metodo factory può anche portare a un design più allineato con l'Ubiquitous Language di uno specifico contesto.

Ecco un esempio di codice:

```
//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");
```

Leggi Fabbrica online: https://riptutorial.com/it/design-patterns/topic/1375/fabbrica

Capitolo 7: Facciata

Examples

Facciata del mondo reale (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");
    }
}
```

Esempio di facciata in java

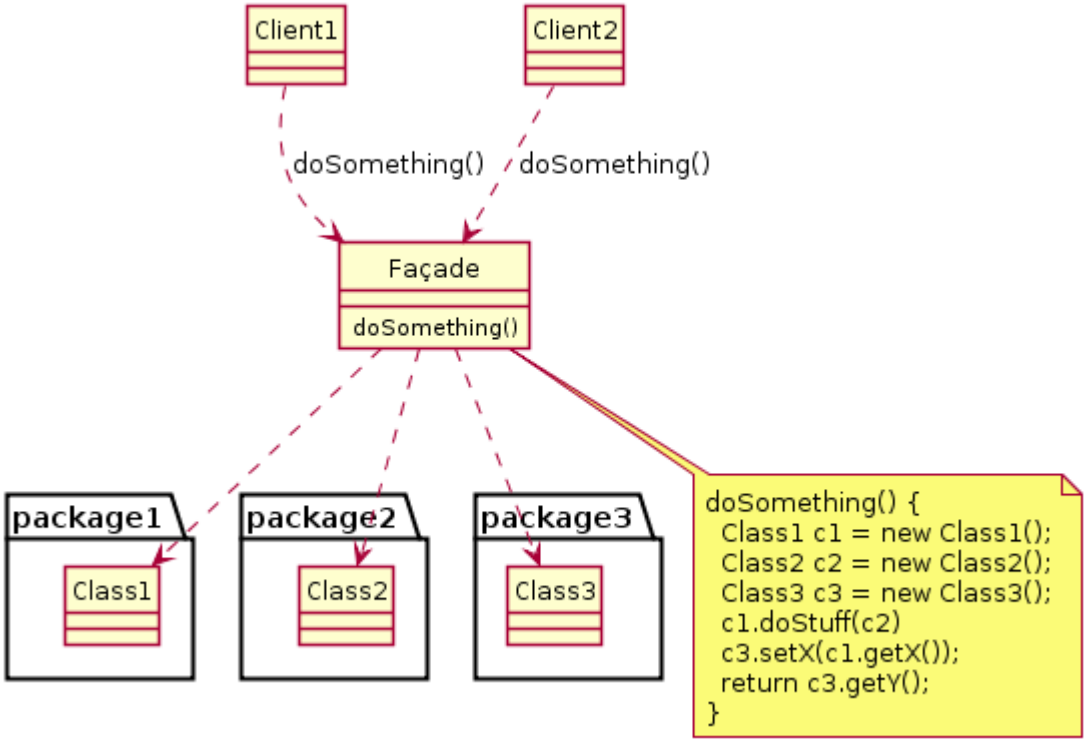
La facciata è un modello di progettazione strutturale. Nasconde le complessità del grande sistema e fornisce una semplice interfaccia al client.

Il client utilizza solo **Facade** e non è preoccupato per le dipendenze tra sottosistemi.

Definizione dal libro Gang of Four:

> Fornire un'interfaccia unificata a un insieme di interfacce in un sottosistema. Façade definisce un'interfaccia di livello superiore che semplifica l'utilizzo del sottosistema

Struttura:



Esempio di mondo reale:

Pensa ad alcuni siti di prenotazione di viaggi come makemytrip, cleartrip che offre servizi per prenotare treni, voli e hotel.

Snippet di codice:

```
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){

    }
}
```

Spiegazione:

1. `FlightBooking, TrainBooking and HotelBooking` sono diversi sottosistemi di grande sistema: `TravelFacade`

2. `TravelFacade` offre un'interfaccia semplice per prenotare una delle seguenti opzioni

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

3. book API di TravelFacade chiama internamente sotto API di sottosistemi

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

4. In questo modo, `TravelFacade` fornisce `TravelFacade` semplice e semplice che non espone le API dei sottosistemi.

Casi di applicabilità e utilizzo (da Wikipedia):

1. Per accedere a un sistema complesso è necessaria una semplice interfaccia.
2. Le astrazioni e le implementazioni di un sottosistema sono strettamente accoppiate.
3. È necessario un punto di ingresso per ogni livello di software stratificato.
4. Il sistema è molto complesso o difficile da capire.

Leggi Facciata online: https://riptutorial.com/it/design-patterns/topic/3516/facciata

Capitolo 8: Iniezione di dipendenza

introduzione

L'idea generale alla base di Dipendenza Iniezione è che si progetta la propria applicazione intorno a componenti liberamente accoppiati mentre si aderisce al Principio di Inversione di dipendenza. Non dipendendo da implementazioni concrete, consente di progettare sistemi altamente flessibili.

Osservazioni

L'idea alla base dell'iniezione delle dipendenze è quella di creare un codice più liberamente accoppiato. Quando una classe, invece di riannodare le proprie dipendenze, prende invece le sue dipendenze, la classe diventa più semplice da testare come unità (test di unità).

Per approfondire ulteriormente l'accoppiamento lento - l'idea è che le classi diventano dipendenti dalle astrazioni, piuttosto che dalle concrezioni. Se la classe `A` dipende da un'altra classe concreta `B` , allora non esiste un vero test di `A` senza `B` Mentre questo tipo di test può essere OK, non si presta al codice testabile dell'unità. Un disegno liberamente accoppiato definirebbe un'astrazione `IB` (ad esempio) dalla quale la classe `A` dipenderebbe. `IB` può quindi essere deriso per fornire un comportamento testabile, piuttosto che basarsi sulla reale implementazione di `B` per essere in grado di fornire scenari testabili ad `A`

Esempio strettamente accoppiato (C #):

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

In quanto sopra, la classe `A` dipende da `B` Non c'è un test `A` senza il cemento `B` Sebbene ciò sia soddisfacente in uno scenario di test di integrazione, è difficile eseguire il test unitario `A`

Un'implementazione più facilmente accoppiata di quanto sopra potrebbe sembrare:

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

public class A
{
    private readonly IB _iB;

    public A(IB iB)
    {
```

```
        _iB = iB;
    }

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

Le due implementazioni sembrano abbastanza simili, tuttavia c'è una differenza importante. La classe `A` non dipende più direttamente dalla classe `B` , ora dipende da `IB` . La Classe `A` non ha più la responsabilità di rinnovare le proprie dipendenze: ora devono essere fornite ***ad*** `A`

Examples

Iniezione setter (C #)

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

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

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

Iniezione costruttore (C #)

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

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

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

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

Leggi Iniezione di dipendenza online: https://riptutorial.com/it/design-patterns/topic/1723/iniezione-di-dipendenza

Capitolo 9: lavagna

Examples

Campione C

Blackboard.cs

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


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


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

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

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

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

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

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

    }
}
```

Control.cs

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


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

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

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

        }

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

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

ControlData.cs

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


namespace Blackboard
{
    public class ControlData:ICloneable
    {
        public string problem;
        public object[] input;
        public object[] output;
        public string updateby;
        public DateTime updated;


        public ControlData()
        {
            this.problem = null;
            this.input = this.output = null;
        }

        public ControlData(string problem, object[] input)
        {
            this.problem = problem;
            this.input = input;
            this.updated = DateTime.Now;
        }

        public object getResult()
        {
            return this.output;
        }

        public object Clone()
        {
```

```
            ControlData clone;
            clone = new ControlData(this.problem, this.input);
            clone.updated = this.updated;
            clone.updateby = this.updateby;
            clone.output = this.output;
            return clone;
        }
    }
}
```

KnowledgeWorker.cs

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

namespace Blackboard {
    /// <summary>
    /// each knowledgeworker is resposible for knowing the conditions under which it can
contribute to a solution.
    /// </summary>
    abstract public  class KnowledgeWorker
    {
        protected Boolean canContribute;
        protected string Name;
        public BlackBoard blackboard = null;
        protected List<KeyValuePair<string, ControlData>> keys;
        public KnowledgeWorker(BlackBoard blackboard, String Name)
        {
            this.blackboard = blackboard;
            this.Name = Name;
        }

        public KnowledgeWorker(String Name)
        {
            this.Name = Name;
        }

        public string getName()
        {
            return this.Name;
        }

        abstract public void executeAction();

        abstract public void executeCondition();

        abstract public void updateBlackboard();

    } }
```

Leggi lavagna online: https://riptutorial.com/it/design-patterns/topic/6519/lavagna

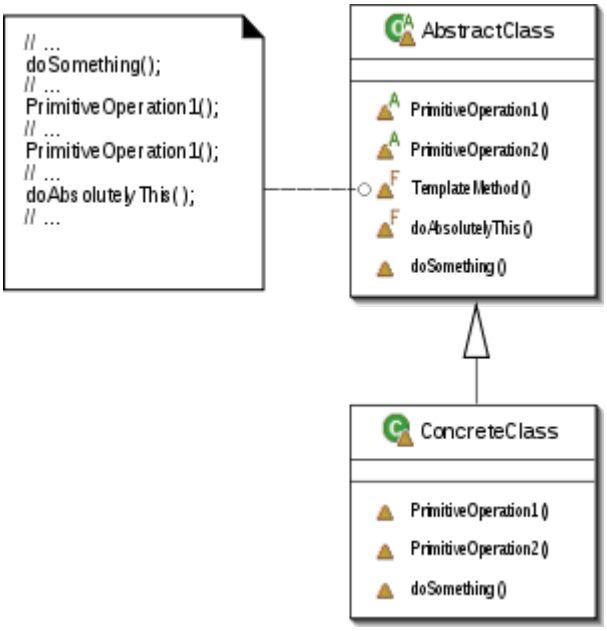
Capitolo 10: Metodo del modello

Examples

Implementazione del metodo template in java

Il modello di modello di modello è un modello di progettazione comportamentale che definisce lo scheletro del programma di un algoritmo in un'operazione, rinviando alcuni passaggi a sottoclassi.

Struttura:



Note chiave:

1. Il metodo Template utilizza l'ereditarietà
2. Il metodo Template implementato dalla classe base non deve essere sovrascritto. In questo modo, la struttura dell'algoritmo è controllata dalla super classe e i dettagli sono implementati nelle sottoclassi

Esempio di codice:

```
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();
    }
}
```

Spiegazione:

1. `Game` è una super classe `abstract` , che definisce un metodo di modello: `playGame()`
2. Scheletro di `playGame()` è definito nella classe base: `Game`
3. Sottoclassi come `Chess, Ludo` e `Checkers` non possono cambiare lo scheletro di `playGame()` . Ma possono modificare il comportamento di alcuni passaggi come

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

produzione:

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

Leggi Metodo del modello online: https://riptutorial.com/it/design-patterns/topic/4867/metodo-del-

modello

Capitolo 11: Metodo di fabbrica statico

Examples

Metodo di fabbrica statico

Possiamo fornire un nome significativo per i nostri costruttori.

Possiamo fornire diversi costruttori con lo stesso numero e tipo di parametri, qualcosa che come abbiamo visto in precedenza non possiamo fare con i costruttori di classi.

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

Nascondere l'accesso diretto al costruttore

Possiamo evitare di fornire un accesso diretto a costruttori con risorse intensive, come per i database. classe pubblica 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);
    //...
  }
}
```

Metodo statico di fabbrica C

Il *metodo statico di fabbrica* è una variazione del modello di *metodo di fabbrica* . È usato per creare oggetti senza dover chiamare personalmente il costruttore.

Quando utilizzare il metodo di produzione statica

- se vuoi dare un nome significativo al metodo che genera il tuo oggetto.
- se vuoi evitare la creazione di oggetti troppo complessi, vedi Tuple Msdn .
- se si desidera limitare il numero di oggetti creati (memorizzazione nella cache)
- se vuoi restituire un oggetto di qualsiasi sottotipo del loro tipo di ritorno.

Ci sono alcuni svantaggi come

- Le classi senza un costruttore pubblico o protetto non possono essere inizializzate nel metodo statico di fabbrica.
- I metodi di produzione statici sono come normali metodi statici, quindi non sono distinguibili da altri metodi statici (questo può variare da IDE a IDE)

Esempio

Pizza.cs

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

    private Pizza()
    {
```

```
        SizeDiameterCM = 25;
    }

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

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

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

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

Program.cs

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


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

Produzione

> Una pizza con un diametro di 28 cm e una pizza con un diametro di 25 cm e una pizza con un diametro di 35 cm

Leggi Metodo di fabbrica statico online: https://riptutorial.com/it/design-patterns/topic/6024/metodo-di-fabbrica-statico

Capitolo 12: Modello composito

Examples

Logger composito

Il pattern Composite è un pattern di progettazione che consente di trattare un gruppo di oggetti come una singola istanza di un oggetto. È uno dei modelli di progettazione strutturale di Gang of Four.

L'esempio seguente mostra come può essere utilizzato Composite per accedere a più posti utilizzando la chiamata al singolo registro. Questo approccio aderisce ai principi SOLID perché consente di aggiungere nuovi meccanismi di registrazione senza violare il principio di Responsabilità singola (ogni registratore ha una sola responsabilità) o principio Aperto / chiuso (È possibile aggiungere un nuovo logger che effettuerà il log in nuovo posto aggiungendo nuova implementazione e non modificando quelli esistenti).

```
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 ricordare che i logger compositi possono essere nidificati (uno dei parametri per il costruttore di logger compositi può essere un altro logger composito) che crea una struttura ad albero.

Leggi Modello composito online: https://riptutorial.com/it/design-patterns/topic/4515/modello-composito

Capitolo 13: Modello composito

introduzione

Il composito consente ai clienti di trattare in modo uniforme singoli oggetti e composizioni di oggetti. Ad esempio, considera un programma che manipola un file system. I file sono oggetti semplici e le cartelle sono composte da file e cartelle. Tuttavia, ad esempio, entrambi hanno funzioni di dimensione, nome, ecc. Sarebbe più facile e più conveniente trattare uniformemente gli oggetti di file e cartelle definendo un'interfaccia di risorse file system

Osservazioni

Il modello composito si applica quando c'è una gerarchia di oggetti parte-intera e un client ha bisogno di trattare gli oggetti in modo uniforme indipendentemente dal fatto che un oggetto possa essere una foglia (oggetto semplice) o un ramo (oggetto composito).

Examples

pseudocodice per un gestore di file stupido

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

Leggi Modello composito online: https://riptutorial.com/it/design-patterns/topic/9197/modello-composito

Capitolo 14: Modello costruttore

Osservazioni

Separa la costruzione di un oggetto complesso dalla sua rappresentazione in modo che lo stesso processo di costruzione possa creare rappresentazioni differenti.

- Separa la logica dalla rappresentazione.
- Riutilizzare la logica per lavorare con diversi set di dati.

Examples

Builder Pattern / C # / Fluent Interrface

```
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;
    }
}
```

Esempio di utilizzo:

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

Builder Pattern / Java Implementation

Il pattern Builder ti consente di creare un'istanza di una classe con molte variabili opzionali in un modo facile da leggere.

Considera il seguente codice:

```
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...

}
```

Questo va tutto bene se tutti i parametri sono necessari. Cosa succede se ci sono molte più variabili e / o alcune di esse sono opzionali? Non si desidera creare un numero elevato di costruttori con ciascuna combinazione possibile di parametri obbligatori e facoltativi perché diventa difficile da mantenere e da comprendere per gli sviluppatori. Potresti anche non voler avere una lunga lista di parametri in cui molti potrebbero dover essere inseriti come nulli dall'utente.

Il modello Builder crea una classe interna chiamata Builder che viene utilizzata per creare solo le variabili opzionali desiderate. Questo viene fatto attraverso i metodi per ogni variabile opzionale che prende il tipo di variabile come parametro e restituisce un oggetto Builder in modo che i metodi possano essere concatenati l'uno con l'altro. Tutte le variabili richieste vengono inserite nel costruttore Builder in modo che non possano essere escluse.

Il Builder include anche un metodo chiamato `build()` che restituisce l'oggetto in cui si trova e deve essere chiamato alla fine della catena di chiamate al metodo durante la creazione dell'oggetto.

Seguendo l'esempio precedente, questo codice utilizza il modello Builder per la classe Computer.

```
public class Computer {

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

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

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

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

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

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

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

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

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

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

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

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

Un esempio di come questa classe verrebbe utilizzata:

```
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();
    }

}
```

Questo esempio mostra come il modello di builder può consentire molta flessibilità nel modo in cui una classe viene creata con un minimo sforzo. L'oggetto Computer può essere implementato in base alla configurazione desiderata del chiamante in modo facile da leggere con poco sforzo.

Modello di generatore in Java con composizione

Intento:

Separa la costruzione di un oggetto complesso dalla sua rappresentazione in modo che lo stesso processo di costruzione possa creare rappresentazioni differenti

Il modello di builder è utile quando hai pochi attributi obbligatori e molti attributi facoltativi per costruire un oggetto. Per creare un oggetto con diversi attributi obbligatori e facoltativi, devi fornire un costruttore complesso per creare l'oggetto. Il modello di costruzione fornisce un semplice processo passo-passo per costruire un oggetto complesso.

Caso di utilizzo della vita reale:

Diversi utenti in FaceBook hanno attributi diversi, che consistono in attributi obbligatori come nome utente e attributi facoltativi come UserBasicInfo e ContactInfo. Alcuni utenti forniscono semplicemente informazioni di base. Alcuni utenti forniscono informazioni dettagliate, tra cui informazioni di contatto. In assenza di pattern Builder, è necessario fornire un costruttore con tutti i parametri obbligatori e facoltativi. Ma il modello Builder semplifica il processo di costruzione fornendo un semplice processo passo-passo per costruire l'oggetto complesso.

Suggerimenti:

1. Fornire una classe di build nidificata statica.
2. Fornire il costruttore per gli attributi obbligatori dell'oggetto.
3. Fornire metodi setter e getter per attributi opzionali dell'oggetto.
4. Restituisce lo stesso oggetto Builder dopo aver impostato gli attributi opzionali.
5. Fornire il metodo build (), che restituisce l'oggetto complesso

Snippet di codice:

```
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(":").appenc

        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);
    }
}
```

produzione:

```
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
```

Spiegazione:

1. `FaceBookUser` è un oggetto complesso con gli attributi seguenti che utilizzano la composizione:

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

2. `FaceBookUserBuilder` è una classe di build statica, che contiene e costruisce `FaceBookUser` .

3. `userName` è solo un parametro obbligatorio per creare `FaceBookUser`

4. `FaceBookUserBuilder` crea `FaceBookUser` impostando parametri opzionali: `UserBasicInfo` e `ContactInfo`

5. Questo esempio illustra tre diversi FaceBookUser con attributi diversi, creati da Builder.

   1. fbUser1 è stato creato come FaceBookUser con solo attributo userName
   2. fbUser2 è stato creato come FaceBookUser con userName e UserBasicInfo
   3. fbUser3 è stato creato come FaceBookUser con userName, UserBasicInfo e ContactInfo

Nell'esempio precedente, la composizione è stata utilizzata invece di duplicare tutti gli attributi di FaceBookUser nella classe Builder.

Nei pattern creativi, inizieremo innanzitutto con pattern semplici come `FactoryMethod` e ci muoviamo verso pattern più flessibili e complessi come `AbstractFactory` e `Builder` .

Java / Lombok

```
import lombok.Builder;

@Builder
public class Email {

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

}
```

Esempio di utilizzo:

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

Modello di generatore avanzato con espressione Lambda di Java 8

```
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;
 }
}
```

Vecchio modo

```
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);
}
```

Modo avanzato:

```
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();
```

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

Leggi Modello costruttore online: https://riptutorial.com/it/design-patterns/topic/1811/modello-costruttore

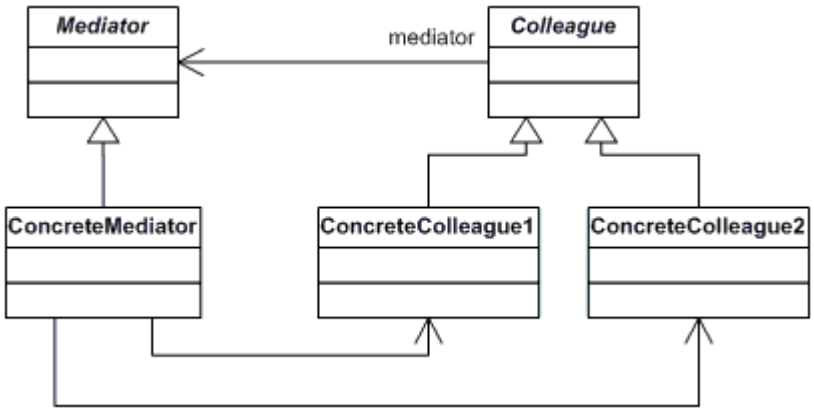
Capitolo 15: Modello del mediatore

Examples

Esempio di modello di mediatore in java

Il modello del *mediatore* definisce un oggetto (mediatore) che incapsula il modo in cui un insieme di oggetti interagisce. Permette la comunicazione many-to-many.

Diagramma UML:



Componenti chiave:

`Mediator`: definisce un'interfaccia per la comunicazione tra colleghi.

`Colleague` : è una classe astratta, che definisce gli eventi da comunicare tra colleghi

`ConcreteMediator` : implementa il comportamento cooperativo coordinando gli oggetti del `Colleague` e mantenendo i suoi colleghi

`ConcreteColleague` : implementa le operazioni di notifica ricevute tramite `Mediator` , che è stata generata da un altro `Colleague`

Un esempio del mondo reale:

Si sta mantenendo una rete di computer nella topologia `Mesh` .

Una rete mesh è una topologia di rete in cui ogni nodo trasmette i dati per la rete. Tutti i nodi mesh collaborano alla distribuzione dei dati nella rete.

Se viene aggiunto un nuovo computer o viene rimosso un computer esistente, tutti gli altri computer in tale rete dovrebbero essere a conoscenza di questi due eventi.

Vediamo come si inserisce il modello del mediatore.

Snippet di codice:

```
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);
    }
}
```

produzione:

```
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
```

Spiegazione:

1. `Eagle` viene aggiunto alla rete in un primo momento attraverso l'evento di registrazione. Nessuna notifica a nessun altro collega dal momento che Eagle è il primo.
2. Quando `Ostrich` viene aggiunto alla rete, viene inviata una notifica a `Eagle` : la riga 1 dell'output viene visualizzata ora.
3. Quando `Penguin` viene aggiunto alla rete, sia `Eagle` sia `Ostrich` sono stati avvisati: la riga 2 e la riga 3 dell'output sono ora renderizzate.
4. Quando `Eagle` lasciato la rete attraverso un evento di annullamento della registrazione, sia gli `Ostrich` che i `Penguin` sono stati avvisati. La riga 4 e la riga 5 dell'output sono ora renderizzate.

Leggi Modello del mediatore online: https://riptutorial.com/it/design-patterns/topic/6184/modello-del-mediatore

Capitolo 16: Modello di comando

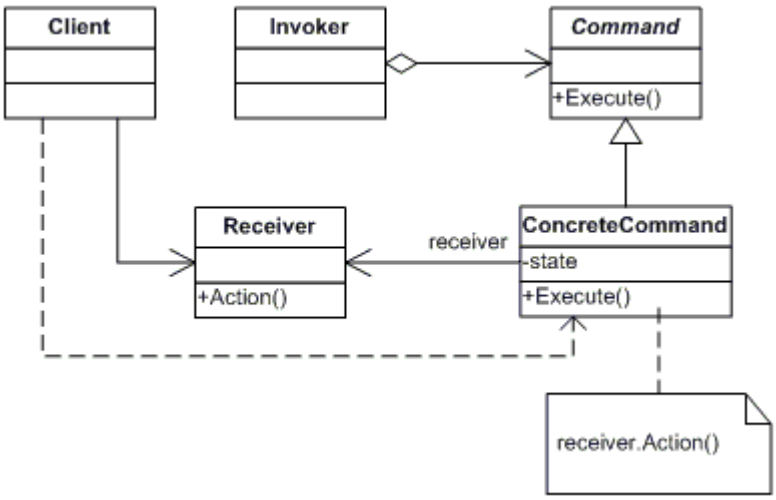
Examples

Esempio di modello di comando in Java

definizione di wikipedia :

> Il modello di comando è un modello di progettazione comportamentale in cui un oggetto viene utilizzato per incapsulare tutte le informazioni necessarie per eseguire un'azione o attivare un evento in un secondo momento

Diagramma UML da dofactory :



Componenti di base e flusso di lavoro:

1. `Command` dichiara un'interfaccia per comandi astratti come `execute()`
2. `Receiver` sa come eseguire un particolare comando
3. `Invoker` detiene `ConcreteCommand` , che deve essere eseguito
4. `Client` crea `ConcreteCommand` e assegnare `Receiver`
5. `ConcreteCommand` definisce il binding tra `Command` e `Receiver`

In questo modo, Command pattern disaccoppia **Sender** (Client) dal **Receiver** tramite **Invoker** . **Invoker** ha una conoscenza completa di cui **comando** da eseguire e **di comando** sa quale **ricevitore** da richiamare per eseguire una particolare operazione.

Snippet di codice:

```
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();
    }
}
```

produzione:

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

Spiegazione:

In questo esempio,

1. **L'** interfaccia di **comando** definisce il metodo `execute()` .
2. **OnCommand** è **ConcreteCommand** , che implementa il metodo `execute()` .
3. **Il ricevitore** è la classe base.
4. **TV** e **DVDPlayer** sono due tipi di **ricevitori** , che vengono passati a ConcreteCommand come OnCommand.
5. **Invoker** contiene **Command** . È la chiave per decompilare il Sender dal **ricevitore** .
6. **Invoker** riceve **OnCommand** -> che chiama **Receiver** (TV) per eseguire questo comando.

Usando Invoker, puoi attivare TV e DVDPlayer. Se estendi questo programma, spegni anche TV e DVDPlayer.

Casi d'uso principali:

1. Per implementare il meccanismo di callback
2. Per implementare la funzionalità di annullamento e ripetizione
3. Mantenere una cronologia dei comandi

Leggi Modello di comando online: https://riptutorial.com/it/design-patterns/topic/2677/modello-di-comando

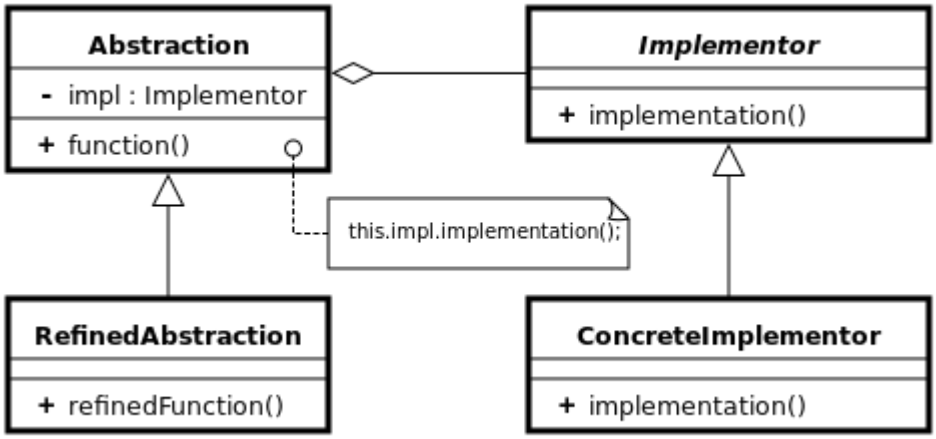
Capitolo 17: Modello di ponte

Examples

Implementazione del pattern bridge in java

Il pattern del bridge disaccoppia l'astrazione dall'implementazione in modo che entrambi possano variare in modo indipendente. È stato realizzato con la composizione piuttosto che con l'eredità.

Diagramma UML di Bridge da wikipedia:



Hai quattro componenti in questo modello.

`Abstraction` : definisce un'interfaccia

`RefinedAbstraction` astrazione: implementa l'astrazione:

`Implementor` : definisce un'interfaccia per l'implementazione

`ConcreteImplementor` : implementa l'interfaccia Implementor.

`The crux of Bridge pattern :` due gerarchie di classi ortogonali che utilizzano la composizione (e nessuna ereditarietà). La gerarchia di astrazione e la gerarchia di implementazione possono variare in modo indipendente. L'implementazione non fa mai riferimento all'astrazione. L'astrazione contiene l'interfaccia di implementazione come membro (attraverso la composizione). Questa composizione riduce un ulteriore livello di gerarchia dell'ereditarietà.

Caso di utilizzo di parole reali:

Abilitare diversi veicoli per avere entrambe le versioni del sistema di cambio manuale e automatico.

Codice di esempio:

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

produzione:

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

Spiegazione:

1. `Vehicle` è un'astrazione.
2. `Car` e `Truck` sono due implementazioni concrete del `Vehicle` .
3. `Vehicle` definisce un metodo astratto: `addGear()` .
4. `Gear` è un'interfaccia implementatore
5. `ManualGear` e `AutoGear` sono due implementazioni di `Gear`
6. `Vehicle` contiene un'interfaccia `implementor` piuttosto che l'implementazione dell'interfaccia. `Compositon` of implementor interface è un punto cruciale di questo modello: *consente all'astrazione e all'implementazione di variare in modo indipendente.*
7. `Car` and `Truck` definisce l'implementazione (astrazione ridefinita) per l'astrazione: `addGear()` : contiene `Gear` - `Manual` o `Auto`

Utilizzare il / i case / i per il modello Bridge :

1. **L'astrazione** e l' **implementazione** possono cambiare indipendentemente l'una dall'altra e non sono vincolate al momento della compilazione
2. Mappa gerarchie ortogonali - Una per l' *astrazione* e una per l' *implementazione* .

Leggi Modello di ponte online: https://riptutorial.com/it/design-patterns/topic/4011/modello-di-ponte

Capitolo 18: Modello di prototipo

introduzione

Il pattern Prototype è un pattern creazionale che crea nuovi oggetti clonando l'oggetto prototipo esistente. Il modello del prototipo accelera l'istanziazione delle classi quando la copia degli oggetti è più veloce.

Osservazioni

Il modello prototipo è un modello di design creativo. Viene utilizzato quando il tipo di oggetti da creare è determinato da un'istanza prototipica, che viene *"clonata"* per produrre nuovi oggetti.

Questo modello viene utilizzato quando una classe ha bisogno di un *"costruttore di materiali polimorfici (copia)"* .

Examples

Prototype Pattern (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; }
};
```

Ciò consente di costruire la classe derivata da un puntatore della classe 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();
```

Modello di prototipo (C #)

Il modello prototipo può essere implementato utilizzando l'interfaccia ICloneable in .NET.

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

Prototype Pattern (JavaScript)

Nei linguaggi classici come Java, C # o C ++ iniziamo creando una classe e quindi possiamo creare nuovi oggetti dalla classe o estendere la classe.

In JavaScript per prima cosa creiamo un oggetto, quindi possiamo aumentare l'oggetto o creare nuovi oggetti da esso. Quindi penso che JavaScript mostri un prototipo reale rispetto al linguaggio

classico.

Esempio :

```
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";
    }
};
```

Qui, creiamo un oggetto denominato `Customer` , e quindi senza creare *nuovo oggetto* abbiamo esteso l' `Customer object` esistente utilizzando la parola chiave *prototype* . Questa tecnica è conosciuta come **Prototype Pattern** .

Leggi Modello di prototipo online: https://riptutorial.com/it/design-patterns/topic/5867/modello-di-prototipo

Capitolo 19: Modello di strategia

Examples

Nascondere i dettagli di implementazione della strategia

Una linea guida molto comune nel design orientato agli oggetti è "il meno possibile ma quanto necessario". Ciò vale anche per il modello di strategia: in genere è consigliabile nascondere i dettagli di implementazione, ad esempio quali classi implementano effettivamente le strategie.

Per strategie semplici che non dipendono da parametri esterni, l'approccio più comune consiste nel rendere la classe di implementazione stessa privata (classi nidificate) o package-private e nell'esporre un'istanza attraverso un campo statico di una classe pubblica:

```
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
);
```

Il campo pubblico `Animal.AGE_COMPARATOR` definisce una strategia che può quindi essere utilizzata in metodi come `Collections.sort` , ma non richiede all'utente di sapere nulla sulla sua implementazione, nemmeno sulla classe di implementazione.

Se preferisci, puoi usare una classe anonima:

```
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...
}
```

Se la strategia è un po 'più complessa e richiede parametri, è molto comune utilizzare metodi di factory static come `Collections.reverseOrder(Comparator<T>)` . Il tipo di ritorno del metodo non dovrebbe esporre dettagli di implementazione, ad esempio `reverseOrder()` è implementato come

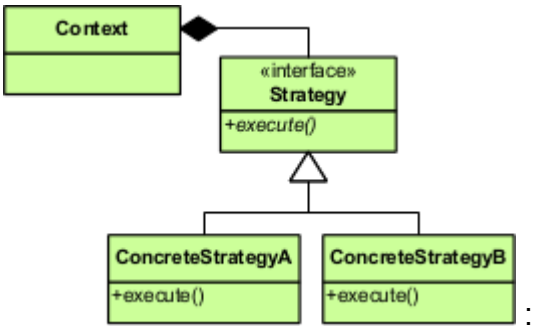
```
public static <T> Comparator<T> reverseOrder(Comparator<T> cmp) {
  // (Irrelevant lines left out.)
  return new ReverseComparator2<>(cmp);
}
```

Esempio di modello di strategia in java con classe Context

Strategia:

`Strategy` è un modello comportamentale, che consente di modificare dinamicamente l'algoritmo da una famiglia di algoritmi correlati.

UML del modello di strategia da 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);
    }

}
```

produzione:

```
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
```

Dichiarazione di problema: offerta sconto del 25% sul prezzo dell'articolo per i mesi di luglio-dicembre. Non fornire sconti per i mesi di gennaio-giugno.

L'esempio sopra mostra l'utilizzo del modello di `Strategy` con il `Context` . `Context` può essere utilizzato come punto di contatto unico per il `Client` .

Due strategie: `NoOfferStrategy` e `QuarterDiscountStrategy` sono state dichiarate come da dichiarazione del problema.

Come mostrato nella colonna di output, otterrai uno sconto in base al mese che hai inserito

Usa caso / i per modello di strategia :

1. Utilizzare questo modello quando si dispone di una famiglia di algoritmi intercambiabili e si deve modificare l'algoritmo in fase di esecuzione.

2. Mantieni il codice più pulito rimuovendo le dichiarazioni condizionali

Modello di strategia senza classe di contesto / Java

Quello che segue è un semplice esempio di utilizzo del modello di strategia senza una classe di contesto. Esistono due strategie di implementazione che implementano l'interfaccia e risolvono lo stesso problema in modi diversi. Gli utenti della classe EnglishTranslation possono chiamare il metodo di traduzione e scegliere la strategia che vorrebbero utilizzare per la traduzione, specificando la strategia desiderata.

```
// 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
    }
}
```

Utilizzo di interfacce funzionali Java 8 per implementare il modello di strategia

Lo scopo di questo esempio è mostrare come possiamo realizzare il modello della strategia usando le interfacce funzionali Java 8. Inizieremo con i codici di un caso semplice in Java classico e quindi lo ricodificheremo in Java 8.

Il problema di esempio che usiamo è una famiglia di algoritmi (strategie) che *descrivono* diversi modi di comunicare a distanza.

La versione Java classica

Il contratto per la nostra famiglia di algoritmi è definito dalla seguente interfaccia:

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

Quindi possiamo implementare un numero di algoritmi, come segue:

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

Questi possono essere istanziati come segue:

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

Successivamente, implementiamo un servizio che utilizza la strategia:

```
public class CommunicationService {
    private CommunicateInterface communcationMeans;

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

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

Infine, possiamo utilizzare le diverse strategie come segue:

```
CommunicationService communicationService = new CommunicationService();

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

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

Utilizzo di interfacce funzionali Java 8

Il contratto delle diverse implementazioni dell'algoritmo non ha bisogno di un'interfaccia dedicata.

Invece, possiamo descriverlo usando l'interfaccia `java.util.function.Function<T, R>` esistente.

I diversi algoritmi che compongono `the family of algorithms` possono essere espressi come espressioni lambda. Questo sostituisce le classi di strategia *e le* loro istanziazioni.

```
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..";
```

Successivamente, possiamo codificare il "servizio" come segue:

```
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 usiamo le strategie come segue

```
CommunicationService communicationService = new CommunicationService();

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

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

O anche:

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

Strategia (PHP)

Esempio da 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;
    }
}
```

Leggi Modello di strategia online: https://riptutorial.com/it/design-patterns/topic/1331/modello-di-strategia

Capitolo 20: Modello di visitatore

Examples

Esempio di pattern Visitor in C ++

Invece di

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

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

I visitatori possono essere utilizzati:

```
// 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;
};
```

Ora le forme concrete:

```
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;
};
```

quindi i visitatori:

```
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.;
};
```

E usalo:

```
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;
}
```

Uscita prevista:

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

dimostrazione

Spiegazione :

- In `void Square::accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }` , il tipo statico di `this` è noto, quindi il sovraccarico scelto (in fase di compilazione) è `void IVisitor::visit(const Square&);` .
- Per `square.accept(visitor);` chiamata, l'invio dinamico tramite `virtual` viene utilizzato per sapere quale `accept` chiamare.

Pro :

- Puoi aggiungere nuove funzionalità (`SerializeAsXml` , ...) alla classe `IShape` semplicemente aggiungendo un nuovo visitatore.

Contro :

- L'aggiunta di una nuova forma concreta (`Triangle` , ...) richiede la modifica di tutti i visitatori.

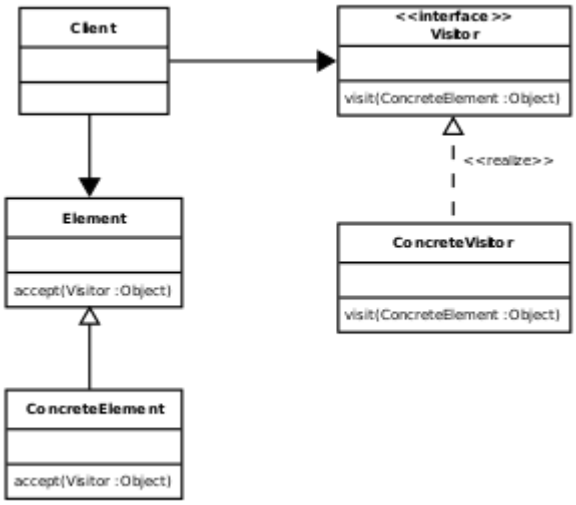
L'alternativa di mettere tutte le funzionalità come metodi `virtual` in `IShape` ha opposti vantaggi e svantaggi: l'aggiunta di nuove funzionalità richiede la modifica di tutte le forme esistenti, ma l'aggiunta di una nuova forma non ha alcun impatto sulle classi esistenti.

Esempio di modello di visitatore in java

`Visitor` pattern `Visitor` consente di aggiungere nuove operazioni o metodi a un set di classi senza modificare la struttura di tali classi.

Questo modello è particolarmente utile quando si desidera centralizzare una particolare operazione su un oggetto senza estendere l'oggetto o senza modificare l'oggetto.

Diagramma UML da wikipedia:



Snippet di codice:

```
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);
        }
    }
}
```

Spiegazione:

1. `Visitable` (`Element`) è un'interfaccia e questo metodo di interfaccia deve essere aggiunto a un insieme di classi.
2. `Visitor` è un'interfaccia, che contiene metodi per eseguire un'operazione su elementi `Visitable` .
3. `GameVisitor` è una classe che implementa l'interfaccia `Visitor` (`ConcreteVisitor`).
4. Ogni elemento `Visitable` accetta `Visitor` e invoca un metodo rilevante di interfaccia `Visitor` .
5. Puoi trattare `Game` as `Element` e giochi di cemento come `Chess,Checkers and Ludo` come `ConcreteElements` .

Nell'esempio sopra, `Chess, Checkers and Ludo` sono tre giochi diversi (e classi `Visitable`). In un bel giorno, ho incontrato uno scenario per registrare le statistiche di ogni gioco. Quindi, senza modificare la classe individuale per implementare la funzionalità statistica, è possibile centralizzare tale responsabilità nella classe `GameVisitor` , che fa il trucco per te senza modificare la struttura di ogni gioco.

produzione:

```
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
```

Casi d'uso / Applicabilità:

1. *Operazioni simili devono essere eseguite* su oggetti di diversi tipi raggruppati in una struttura
2. È necessario eseguire molte operazioni distinte e non correlate. *Separa l'operazione dagli oggetti Struttura*
3. Nuove operazioni devono essere aggiunte *senza modifiche nella struttura dell'oggetto*
4. *Raccogli le operazioni correlate in una singola classe* anziché costringerti a modificare o ricavare classi
5. Aggiungi funzioni alle librerie di classi per le quali *non hai la sorgente o non puoi cambiare la fonte*

Ulteriori riferimenti:

oodesign

sourcemaking

Esempio di visitatore in 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 esempio di utilizzo:

```
// 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);
}
```

Attraversando oggetti di grandi dimensioni

Il modello visitatore può essere utilizzato per attraversare strutture.

```
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);
        }
    }
}
```

Leggi Modello di visitatore online: https://riptutorial.com/it/design-patterns/topic/4579/modello-di-visitatore

Capitolo 21: Modello oggetto nullo

Osservazioni

Oggetto Nullo è un oggetto senza valore di riferimento o con comportamento neutro definito. Il suo scopo è quello di rimuovere la necessità di controllo puntatore / riferimento null.

Examples

Pattern di oggetto nullo (C ++)

Supponendo una classe astratta:

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

Invece di

```
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);
}
```

Puoi creare un Null Object Logger:

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

e quindi modificare `doJob` nel modo seguente:

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

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

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

Oggetto Null Java che utilizza enum

Data un'interfaccia:

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

Piuttosto che usare:

```
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
}
```

Poiché gli oggetti nulli non hanno uno stato, ha senso usare un enum singleton per questo, quindi dato un oggetto nullo implementato in questo modo:

```
public enum NullLogger implements Logger {
    INSTANCE;

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

È quindi possibile evitare i controlli nulli.

```
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);
}
```

Leggi Modello oggetto nullo online: https://riptutorial.com/it/design-patterns/topic/6177/modello-oggetto-nullo

Capitolo 22: Monostate

Osservazioni

Come nota a margine, alcuni vantaggi del modello `Monostate` rispetto a `Singleton` :

- Non esiste un metodo 'instance` per poter accedere a un'istanza della classe.
- Un `Singleton` non è conforme alla notazione Java bean, ma un `Monostate` fa.
- La durata delle istanze può essere controllata.
- Gli utenti della `Monostate` non sanno che stanno usando una `Monostate` .
- Il polimorfismo è possibile.

Examples

Il modello Monostate

Il pattern `Monostate` viene solitamente definito *zucchero sintattico* sul pattern `Singleton` o come `Singleton` *concettuale* .

Evita tutte le complicazioni di avere una singola istanza di una classe, ma tutte le istanze usano gli stessi dati.
Ciò si ottiene principalmente utilizzando membri di dati `static` .
Una delle caratteristiche più importanti è che è assolutamente trasparente per gli utenti, che sono completamente all'oscuro del fatto che stanno lavorando con una `Monostate` . Gli utenti possono creare tutte le istanze di `Monostate` come vogliono e ogni istanza è buona come un'altra per accedere ai dati.

La classe `Monostate` viene solitamente fornita con una classe companion che viene utilizzata per aggiornare le impostazioni, se necessario.

Segue un esempio minimale di `Monostate` in 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; }
};
```

Ecco un esempio di una semplice implementazione di un `Monostate` in 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;
    }
}
```

Gerarchie basate su monostazione

In contrasto con `Singleton` , il `Monostate` è adatto per essere ereditato per estendere le sue funzionalità, purché i metodi dei membri non siano `static` .
Segue un esempio minimale in 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; }
};
```

Leggi Monostate online: https://riptutorial.com/it/design-patterns/topic/6186/monostate

Capitolo 23: Motivo decorativo

introduzione

Il pattern Decorator consente all'utente di aggiungere nuove funzionalità a un oggetto esistente senza alterarne la struttura. Questo tipo di modello di progettazione rientra in un modello strutturale poiché questo modello funge da involucro per la classe esistente.

Questo modello crea una classe decoratore che racchiude la classe originale e fornisce funzionalità aggiuntive mantenendo intatta la firma dei metodi di classe.

Parametri

Parametro	Descrizione
bevanda	può essere tè o caffè

Examples

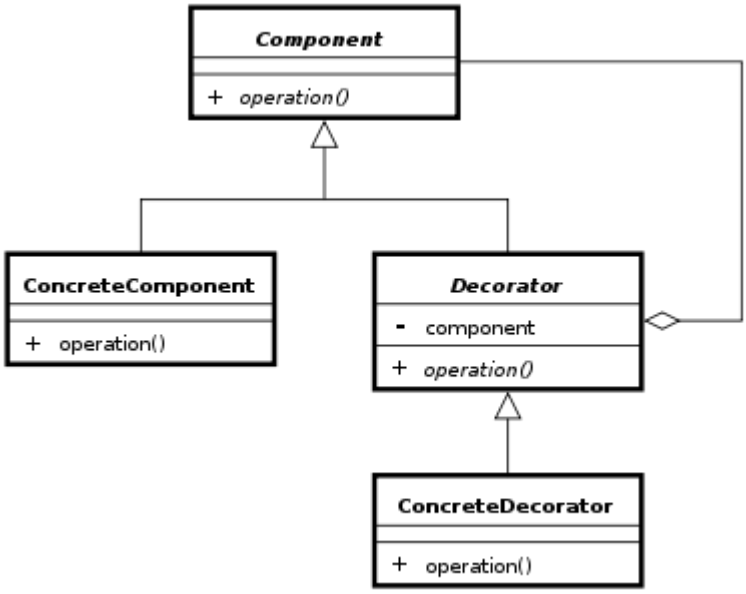
VendingMachineDecorator

Definizione di Decoratore secondo Wikiepdia:

Il pattern Decorator può essere utilizzato per estendere (decorare) la funzionalità di un determinato oggetto in modo statico, o in alcuni casi in fase di esecuzione, indipendentemente da altre istanze della stessa classe, a condizione che alcune attività di base vengano eseguite in fase di progettazione.

Decoratore attribuisce responsabilità aggiuntive a un oggetto in modo dinamico. I decoratori offrono un'alternativa flessibile alla sottoclasse per estendere le funzionalità.

Il motivo decoratore contiene quattro componenti.



1. Interfaccia componente: definisce un'interfaccia per eseguire operazioni particolari
2. ConcreteComponent: implementa le operazioni definite nell'interfaccia Component
3. Decoratore (Abstract): è una classe astratta, che estende l'interfaccia del componente. Contiene l'interfaccia Component. In assenza di questa classe, sono necessarie molte sottoclassi di ConcreteDecorators per combinazioni diverse. La composizione del componente riduce sottoclassi non necessarie.
4. ConcreteDecorator: detiene l'implementazione di Abstract Decorator.

Tornando al codice di esempio,

1. *La bevanda* è componente. Definisce un metodo astratto: decora la bevanda
2. *Tè* e *caffè* sono implementazioni concrete della *bevanda* .
3. *BeverageDecorator* è una classe astratta, che contiene *Beverage*
4. SugarDecorator e LemonDecorator sono Decorators in calcestruzzo per BeverageDecorator.

EDIT: cambiato l'esempio per riflettere lo scenario del mondo reale di calcolare il prezzo della bevanda aggiungendo uno o più aromi come zucchero, limone ecc. (I sapori sono decoratori)

```
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();
    }
}
```

produzione:

```
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
```

Questo esempio calcola il costo della bevanda nel distributore automatico dopo aver aggiunto molti aromi alla bevanda.

Nell'esempio sopra:

Costo del tè = 10, Limone = 3 e Zucchero = 5. Se produci Zucchero + Limone + Tè, costa 18.

Costo del caffè = 15, Limone = 3 e Zucchero = 5. Se produci Zucchero + Limone + Caffè, costa 23

Utilizzando lo stesso decoratore per entrambe le bevande (tè e caffè), il numero di sottoclassi è stato ridotto. In assenza del pattern Decorator, è necessario avere classi secondarie diverse per combinazioni diverse.

Le combinazioni saranno così:

```
SugarLemonTea
SugarTea
LemonTea
```

```
SugarLemonCapaccuino
SugarCapaccuino
LemonCapaccuino
```

eccetera.

Usando lo stesso `Decorator` per entrambe le bevande, il numero di sottoclassi è stato ridotto. È possibile a causa della `composition` piuttosto che del concetto di `inheritance` utilizzato in questo modello.

Confronto con altri modelli di design (dall'articolo di sourcemaking)

1. *L'adattatore* fornisce un'interfaccia diversa al suo soggetto. *Proxy* fornisce la stessa interfaccia. *Decorator* fornisce un'interfaccia migliorata.

2. *Adapter* modifica l'interfaccia di un oggetto, *Decorator* migliora le responsabilità di un oggetto.

3. *Composito* e *Decoratore* hanno diagrammi di struttura simili, il che riflette il fatto che entrambi si basano sulla composizione ricorsiva per organizzare un numero di oggetti aperti

4. *Decorator* è progettato per consentire di aggiungere responsabilità agli oggetti senza sottoclassi. *L'* attenzione *del composito* non è sull'abbellimento ma sulla rappresentazione

5. *Decoratore* e *Proxy* hanno scopi diversi ma strutture simili

6. *Decoratore* ti consente di cambiare la pelle di un oggetto. *La strategia* ti consente di cambiare il coraggio.

Casi d'uso principali:

1. Aggiungi funzionalità / responsabilità aggiuntive in modo dinamico
2. Rimuovere dinamicamente funzionalità / responsabilità
3. Evita troppe sottoclasse per aggiungere ulteriori responsabilità.

Caching Decorator

Questo esempio mostra come aggiungere funzionalità di memorizzazione nella cache a `DbProductRepository` utilizzando il pattern Decorator. Questo approccio è conforme ai principi SOLID perché consente di aggiungere il caching senza violare il principio di Responsabilità singola o principio Aperto / chiuso .

```
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);
```

Il risultato del `GetProduct` di `GetProduct` sarà: recuperare il prodotto dalla cache (responsabilità del decoratore), se il prodotto non era nella cache procedere con l'invocazione a `DbProductRepository` e recuperare il prodotto dal DB. Dopo questo prodotto può essere aggiunto alla cache in modo che le chiamate successive non colpiranno il DB.

Leggi Motivo decorativo online: https://riptutorial.com/it/design-patterns/topic/1720/motivo-decorativo

Capitolo 24: multiton

Osservazioni

Multitonitis

Come Singleton , Multiton può essere considerato una cattiva pratica. Tuttavia, ci sono momenti in cui è possibile utilizzarlo con saggezza (ad esempio, se si costruisce un sistema come ORM / ODM per mantenere più oggetti).

Examples

Pool of Singletons (esempio PHP)

Multiton può essere usato come contenitore per singleton. Questa è l'implementazione Multiton è una combinazione di modelli Singleton e Pool.

Questo è un esempio di come può essere creata la classe Pool astratta Multiton comune:

```
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() {}
}
```

In questo modo possiamo istanziare un certo pool Singleton.

Registro di Singletons (esempio PHP)

Questo modello può essere usato per contenere un Pool di Singleton registrati, ciascuno distinto dall'ID univoco:

```
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() {}
}
```

Questa è una forma semplificata di pattern che può essere utilizzata per ORM per memorizzare diverse entità di un determinato tipo.

Leggi multiton online: https://riptutorial.com/it/design-patterns/topic/6857/multiton

Capitolo 25: MVC, MVVM, MVP

Osservazioni

Si può sostenere che MVC e pattern correlati sono in realtà schemi di architettura software piuttosto che modelli di progettazione software.

Examples

Model View Controller (MVC)

1. Che cos'è MVC?

Il modello Model View Controller (MVC) è un modello di progettazione più comunemente utilizzato per la creazione di interfacce utente. Il principale vantaggio di MVC è che si separa:

- la rappresentazione interna dello stato dell'applicazione (il modello),
- come vengono presentate le informazioni all'utente (la vista), e
- la logica che controlla il modo in cui l'utente interagisce con lo stato dell'applicazione (il Controller).

2. Utilizzare i casi di MVC

Il caso d'uso principale per MVC è nella programmazione della GUI (Graphical User Interface). Il componente Visualizza ascolta il componente Modello per le modifiche. Il modello funge da emittente; quando c'è una modalità di modifica al modello, trasmette le sue modifiche alla vista e al controller. Il controller viene utilizzato dalla vista per modificare il componente del modello.

3. Implementazione

Considera la seguente implementazione di MVC, in cui abbiamo una classe Model chiamata `Animals` , una classe View denominata `DisplayAnimals` e una classe controller chiamata `AnimalController` . L'esempio sotto è una versione modificata del tutorial su MVC da 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. Fonti utilizzate:

Design Patterns - Pattern MVC

Java SE Application Design con MVC

Model-View-Controller

Model View ViewModel (MVVM)

1. Che cos'è MVVM?

Il modello Model View ViewModel (MVVM) è un modello di progettazione più comunemente utilizzato per la creazione di interfacce utente. È derivato dal popolare modello "Model View Controller" (MVC). Il principale vantaggio di MVVM è che si separa:

- La rappresentazione interna dello stato dell'applicazione (il modello).
- Come vengono presentate le informazioni all'utente (la vista).
- La "logica del convertitore di valore" è responsabile dell'esposizione e della conversione dei dati dal modello in modo che i dati possano essere facilmente gestiti e presentati nella vista (ViewModel).

2. Utilizzare i casi di MVVM

Il caso d'uso principale di MVVM è la programmazione della GUI (Graphical User Interface). Viene utilizzato per la semplice programmazione basata sugli eventi delle interfacce utente, separando il livello di visualizzazione dalla logica di backend che gestisce i dati.

Ad esempio, in Windows Presentation Foundation (WPF), la vista è progettata utilizzando il linguaggio di markup framework XAML. I file XAML sono associati a ViewModels utilizzando l'associazione dati. In questo modo la vista è responsabile solo della presentazione e il viewmodel è responsabile solo della gestione dello stato dell'applicazione lavorando sui dati nel modello.

Viene anche utilizzato nella libreria JavaScript KnockoutJS.

3. Implementazione

Considerare la seguente implementazione di MVVM utilizzando C # .Net e WPF. Abbiamo una classe Model chiamata Animals, una classe View implementata in Xaml e un ViewModel chiamato AnimalViewModel. L'esempio sotto è una versione modificata del tutorial su MVC da Design Patterns - MVC Pattern .

Guarda come il Modello non sa nulla, il ViewModel conosce solo il Modello e la Vista conosce solo il ViewModel.

L'evento OnNotifyPropertyChanged consente di aggiornare sia il modello che la vista in modo tale che quando si inserisce qualcosa nella casella di testo nella vista il modello viene aggiornato. E se qualcosa aggiorna il modello, la vista viene aggiornata.

```
/*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. Fonti utilizzate:

Model-View-ViewModel

Un semplice esempio MVVM

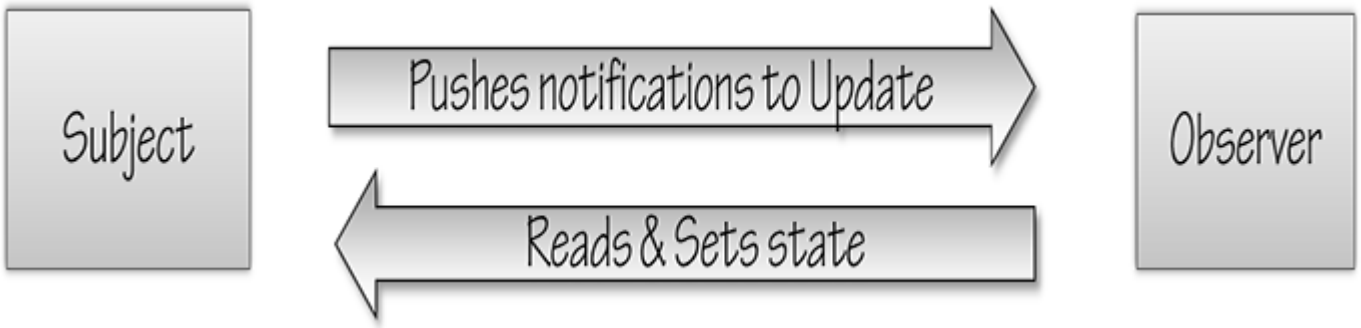
L'esempio MVVM C # WPF più semplice al mondo

Il pattern MVVM

Leggi MVC, MVVM, MVP online: https://riptutorial.com/it/design-patterns/topic/7405/mvc--mvvm--mvp

Capitolo 26: Osservatore

Osservazioni



Qual è l'intento?

- Adottare il principio di separazione delle preoccupazioni.
- Crea una separazione tra il soggetto e l'osservatore.
- Permetti a più osservatori di reagire per cambiare un singolo soggetto.

Qual è la struttura?

- L'oggetto fornisce un modo per registrare, annullare la registrazione, notifica.
- Observer fornisce un modo per aggiornare.

Examples

Observer / Java

Il pattern observer consente agli utenti di una classe di sottoscrivere eventi che accadono quando questa classe elabora i dati, ecc. E di essere avvisati quando si verificano questi eventi. Nell'esempio seguente creiamo una classe di elaborazione e una classe observer che verrà notificata durante l'elaborazione di una frase, se trova le parole che sono più lunghe di 5 lettere.

L'interfaccia `LongWordsObserver` definisce l'osservatore. Implementa questa interfaccia per registrare un osservatore negli eventi.

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

La classe `WordEvent` è l'evento che verrà inviato alle classi observer una volta che si verificano determinati eventi (in questo caso sono state trovate lunghe parole)

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

```
    private String word;

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

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

La classe `PhraseProcessor` è la classe che elabora la frase specificata. Permette agli osservatori di essere registrati usando il metodo `addObserver` . Una volta trovate le parole lunghe, questi osservatori verranno chiamati utilizzando un'istanza della classe `WordEvent` .

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

public class PhraseProcessor {

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

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

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

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

La classe `LongWordsExample` mostra come registrare osservatori, chiamare il metodo di `process` e ricevere avvisi quando sono state trovate parole lunghe.

```
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
    }
}
```

Osservatore che utilizza IObservable e IObserver (C #)

`IObserver<T>` e `IObservable<T>` possono essere utilizzate per implementare pattern di osservatori in .NET

- `IObservable<T>` rappresenta la classe che invia le notifiche
- `IObserver<T>` interfaccia `IObserver<T>` rappresenta la classe che li riceve

```
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: Modelli e pratiche di progettazione in .NET: il modello Observer

Leggi Osservatore online: https://riptutorial.com/it/design-patterns/topic/3185/osservatore

Capitolo 27: Pattern del repository

Osservazioni

Informazioni sull'implementazione di **`IEnumerable<TEntity> Get(Expression<Func<TEntity, bool>> filter)`** : L'idea è di usare espressioni come `i => x.id == 17` per scrivere richieste generiche. È un modo per interrogare i dati senza utilizzare il linguaggio di query specifico della tua tecnologia. L'implementazione è piuttosto estesa, quindi, si potrebbe prendere in considerazione altre alternative, come i metodi specifici sui vostri repository attuate: Un immaginario `CompanyRepository` potrebbe fornire il metodo `GetByName(string name)` .

Examples

Repository di sola lettura (C #)

Un modello di repository può essere utilizzato per incapsulare il codice specifico di archiviazione dei dati in componenti designati. La parte della tua applicazione, che ha bisogno dei dati, funzionerà solo con i repository. Dovrai creare un repository per ogni combinazione di elementi che archivi e tecnologia del tuo database.

I repository di sola lettura possono essere utilizzati per creare repository che non sono autorizzati a manipolare i dati.

Le interfacce

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

    TEntity Get(TKey id);
}

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

    bool Delete(TKey id);

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

Un esempio di implementazione che utilizza ElasticSearch come tecnologia (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;
    }
}
```

Utilizzando questa implementazione, è ora possibile creare repository specifici per gli elementi che si desidera archiviare o accedere. Quando si utilizza la ricerca elastica, è normale che alcuni componenti debbano solo leggere i dati, quindi è necessario utilizzare i repository di sola lettura.

Pattern del repository utilizzando Entity Framework (C #)

Interfaccia del repository;

```
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);
}
```

Repository generico;

```
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);
    }
}
```

Esempio di utilizzo utilizzando una demo hotel class;

```
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();
```

Leggi Pattern del repository online: https://riptutorial.com/it/design-patterns/topic/6254/pattern-del-repository

Capitolo 28: Pattern di progettazione Data Access Object (DAO)

Examples

Data Access Object Modello di progettazione J2EE con Java

Il modello di progettazione *DAO (Data Access Object)* è un modello di progettazione J2EE standard.

In questo modello di progettazione si accede ai dati attraverso classi contenenti metodi per accedere ai dati da database o altre fonti, che sono chiamati *oggetti di accesso ai dati* . La pratica standard presuppone che esistano classi POJO. DAO può essere combinato con altri Pattern design per accedere ai dati, come con MVC (model view controller), Command Patterns, ecc.

Di seguito è riportato un esempio di modello di progettazione DAO. Ha una classe **Employee** , una DAO per Employee denominata **EmployeeDAO** e una classe **ApplicationView** per dimostrare gli esempi.

Employee.java

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

    public Employee(){

    }

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

    //standard setters and getters

}
```

EmployeeDAO

```
public class EmployeeDAO {

    private List<Employee> employeeList;

    public EmployeeDAO(List<Employee> employeeList){
```

```
        this.employeeList = employeeList;
    }

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

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

        return employee;
    }

}
```

ApplicationView.java

```
public class ApplicationView {

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

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

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

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

        Employee employeeWithMaxSalary = eDAO.getEmployeeWithMaxSalary();

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

    }

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

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

```
}
```

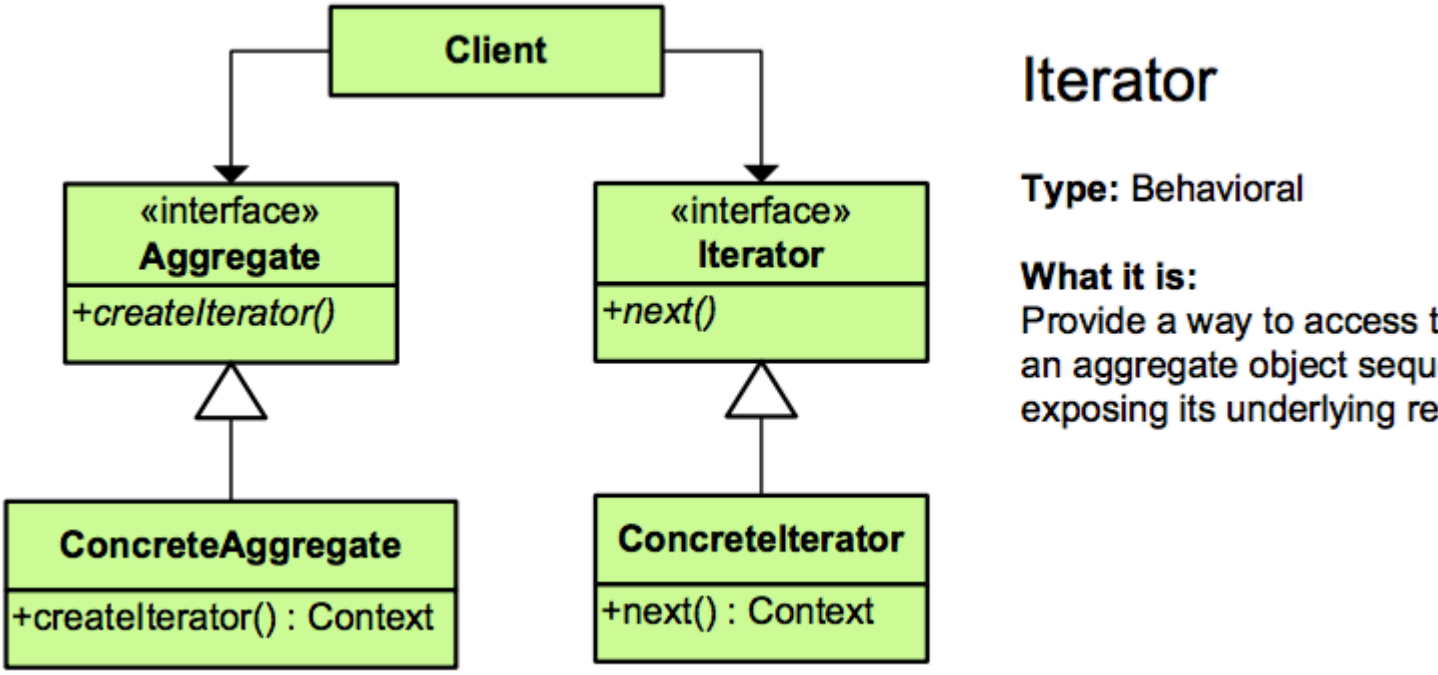
Quindi abbiamo un esempio in cui vediamo come utilizzare il modello di progettazione di Data Access Object.

Leggi Pattern di progettazione Data Access Object (DAO) online: https://riptutorial.com/it/design-patterns/topic/6351/pattern-di-progettazione-data-access-object--dao-

Capitolo 29: Pattern Iterator

Examples

Il pattern Iterator



Le raccolte sono una delle strutture dati più comunemente utilizzate nell'ingegneria del software. Una collezione è solo un gruppo di oggetti. Una raccolta può essere una lista, una matrice, una mappa, un albero o altro. Quindi, una raccolta dovrebbe fornire un modo per accedere ai suoi elementi senza esporre la sua struttura interna. Dovremmo essere in grado di attraversarlo nello stesso modo indipendentemente dal tipo di collezione che è.

L'idea del modello iteratore è di assumersi la responsabilità di accedere all'oggetto di una raccolta e inserirla in un oggetto iteratore. L'oggetto iteratore in cambio manterrà l'ordine di iterazione, manterrà una traccia dell'elemento corrente e dovrà avere un modo per recuperare l'elemento successivo.

Di solito, la classe di raccolta contiene due componenti: la classe stessa, ed è `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;
       }
    }
}
```

Leggi Pattern Iterator online: https://riptutorial.com/it/design-patterns/topic/7061/pattern-iterator

Capitolo 30: Publish-Subscribe

Examples

Pubblica-Iscriviti in Java

L'abbonato editore è un concetto familiare dato l'ascesa di YouTube, Facebook e altri servizi di social media. Il concetto di base è che esiste un `Publisher` che genera contenuti e un `Subscriber` che consuma contenuti. Ogni volta che il `Publisher` genera contenuti, ogni `Subscriber` viene notificato. `Subscribers` possono teoricamente essere abbonati a più di un editore.

Di solito c'è un `ContentServer` che si trova tra editore e sottoscrittore per aiutare a mediare la messaggistica

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

Di solito, lo schema di progettazione pub-sub è implementato con una vista multithreaded in mente. Una delle implementazioni più comuni vede ciascun `Subscriber` come un thread separato, con il `ContentServer` gestisce un pool di thread

Semplice esempio pub-sub in JavaScript

Editori e abbonati non hanno bisogno di conoscersi. Semplicemente comunicano con l'aiuto delle code dei messaggi.

```
(function () {
        var data;

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

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

Qui abbiamo pubblicato un evento personalizzato denominato **myCustomEvent** e sottoscritto su quell'evento. Quindi non hanno bisogno di conoscersi.

Leggi Publish-Subscribe online: https://riptutorial.com/it/design-patterns/topic/7260/publish-subscribe

Capitolo 31: Singleton

Osservazioni

Il motivo di design Singleton è talvolta considerato come " *Anti pattern* ". Ciò è dovuto al fatto che ha alcuni problemi. Devi decidere da solo se pensi che sia appropriato usarlo. Questo argomento è stato discusso più volte su StackOverflow.

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

Examples

Singleton (C #)

I singleton vengono utilizzati per garantire che venga creata una sola istanza di un oggetto. Il singleton consente di creare solo una singola istanza di se stesso, il che significa che ne controlla la creazione. Il singleton è uno dei modelli di design Gang of Four ed è un **modello creativo** .

Pattern Singleton sicuro per thread

```
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 fornisce la seguente implementazione per un singleton pigro e sicuro per i thread:

```
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)

I singleton in Java sono molto simili a C #, poiché entrambi i linguaggi sono orientati agli oggetti. Di seguito è riportato un esempio di una classe singleton, in cui solo una versione dell'oggetto può essere attiva durante la vita del programma (supponendo che il programma funzioni su un thread)

```
public class SingletonExample {

    private SingletonExample() { }

    private static SingletonExample _instance;

    public static SingletonExample getInstance() {

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

Ecco la versione thread-safe di quel programma:

```
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 ha anche un oggetto chiamato `ThreadLocal` , che crea una singola istanza di un oggetto su una base thread per thread. Questo potrebbe essere utile nelle applicazioni in cui ogni thread ha bisogno della propria versione dell'oggetto

```
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();
    }
}
```

Ecco anche un'implementazione di **Singleton** con `enum` (contenente un solo elemento):

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

Qualsiasi implementazione della classe **Enum** garantisce che esista *solo* un'istanza di ciascun elemento.

Bill Pugh Singleton Pattern

Bill Pugh Singleton Pattern è l'approccio più utilizzato per la classe Singleton in quanto non richiede la sincronizzazione

```
public class SingletonExample {

    private SingletonExample(){}

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

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

con l'utilizzo della classe statica interna privata il titolare non viene caricato in memoria finché qualcuno non chiama il metodo getInstance. La soluzione di Bill Pugh è thread-safe e non richiede sincronizzazione.

Esistono altri esempi di singleton Java nell'argomento Singleton sotto il tag della documentazione Java.

Singleton (C ++)

Come per Wiki : nell'ingegneria del software, il modello singleton è un modello di progettazione che limita l'istanziazione di una classe a un oggetto.

Questo è necessario per creare esattamente un oggetto per coordinare le azioni attraverso il 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;
    }
};
```

Esempio pratico Lazy Singleton in java

Casi di utilizzo della vita reale per il modello Singleton;

Se si sta sviluppando un'applicazione client-server, è necessario disporre di un'unica istruzione di `ConnectionManager` , che gestisce il ciclo di vita delle connessioni client.

Le API di base in ConnectionManager:

`registerConnection` : aggiunge una nuova connessione all'elenco esistente di connessioni

`closeConnection` : `closeConnection` la connessione dall'evento attivato dal client o dal server

`broadcastMessage` : alcune volte devi inviare un messaggio a tutte le connessioni client sottoscritte.

Non sto fornendo l'implementazione completa del codice sorgente poiché l'esempio diventerà molto lungo. Ad alto livello, il codice sarà come questo.

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

```
import java.net.*;

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

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

Classe Server di esempio:

```
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();
        }
    }
```

```
}
```

Altri casi di utilizzo pratico per Singletons:

1. Gestione di risorse globali come `ThreadPool, ObjectPool, DatabaseConnectionPool` ecc.
2. Servizi centralizzati come la `Logging` dati dell'applicazione con diversi livelli di registro come `DEBUG,INFO,WARN,ERROR` ecc
3. Global `RegistryService` dove diversi servizi sono registrati con un componente centrale all'avvio. Questo servizio globale può fungere da `Facade` per l'applicazione

C # Esempio: Singleton multithread

L'inizializzazione statica è adatta alla maggior parte delle situazioni. Quando l'applicazione deve ritardare l'istanziazione, utilizzare un costruttore non predefinito o eseguire altre attività prima dell'istanziazione e lavorare in un ambiente con multithreading, è necessaria una soluzione diversa. Esistono tuttavia casi in cui non è possibile fare affidamento sul Common Language Runtime per garantire la sicurezza dei thread, come nell'esempio di Inizializzazione statica. In questi casi, è necessario utilizzare funzionalità linguistiche specifiche per garantire che venga creata una sola istanza dell'oggetto in presenza di più thread. Una delle soluzioni più comuni consiste nell'usare l'idioma Double-Check Locking [Lea99] per impedire a thread separati di creare nuove istanze del singleton contemporaneamente.

La seguente implementazione consente solo ad un singolo thread di entrare nell'area critica, che identifica il blocco di blocco, quando nessuna istanza di Singleton è stata ancora creata:

```
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;
      }
  }
}
```

Questo approccio garantisce che venga creata una sola istanza e solo quando è necessaria l'istanza. Inoltre, la variabile è dichiarata volatile per garantire che l'assegnazione alla variabile di

istanza venga completata prima che sia possibile accedere alla variabile di istanza. Infine, questo approccio utilizza un'istanza syncRoot da bloccare, piuttosto che bloccare il tipo stesso, per evitare deadlock.

Questo approccio di blocco a doppio controllo risolve i problemi di concomitanza di thread evitando un blocco esclusivo in ogni chiamata al metodo di proprietà Instance. Inoltre, consente di ritardare l'istanziazione fino a quando l'oggetto non viene accesso per la prima volta. In pratica, un'applicazione raramente richiede questo tipo di implementazione. Nella maggior parte dei casi, l'approccio di inizializzazione statica è sufficiente.

Riferimento: MSDN

Ringraziamenti

[Gamma95] Gamma, Helm, Johnson e Vlissides. Modelli di progettazione: elementi del software orientato agli oggetti riutilizzabile. Addison-Wesley, 1995.

[Lea99] Lea, Doug. Programmazione simultanea in Java, Seconda edizione. Addison-Wesley, 1999.

[Sells03] Sells, Chris. "Suck Sealed". sellsbrothers.com Notizie. Disponibile all'indirizzo: http://www.sellsbrothers.com/news/showTopic.aspx?ixTopic=411 .

Singleton (PHP)

Esempio da 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)
```

Modello Singleton Design (in generale)

Nota: il singleton è un motivo di progettazione.
Ma ha anche considerato un anti-modello.

L'uso di un singleton deve essere considerato attentamente prima dell'uso. Di solito ci sono alternative migliori.

Il problema principale con un singleton è lo stesso del problema con le variabili globali. Introducono lo stato mutabile globale esterno. Ciò significa che le funzioni che utilizzano un singleton non dipendono esclusivamente dai parametri di input ma anche dallo stato del singleton. Ciò significa che il test può essere gravemente compromesso (difficile).

I problemi con i singleton possono essere mitigati usandoli insieme ai pattern di creazione; in modo che la creazione iniziale del singleton possa essere controllata.

Leggi Singleton online: https://riptutorial.com/it/design-patterns/topic/2179/singleton

Capitolo 32: SOLIDO

introduzione

Cos'è SOLIDO?

SOLID è un acronimo mnemonico (memory aid). I principi di Solid dovrebbero aiutare gli sviluppatori di software ad evitare "odori di codice" e dovrebbero portare a buoni codici sorgente. Un buon codice sorgente significa in questo contesto che il codice sorgente è facile da estendere e mantenere. L'obiettivo principale dei principi Solidi sono le classi

Cosa aspettarsi:

Perché dovresti applicare SOLID

Come applicare i cinque principi SOLID (esempi)

Examples

SRP - Principio della singola responsabilità

La S in SOLID sta per Principio di Responsabilità Singola (SRP).

Responsabilità significa in questo contesto ragioni per cambiare, quindi il principio afferma che una classe dovrebbe avere solo una ragione per cambiare.

Robert C. Martin lo ha affermato (durante la sua conferenza allo Yale shool of management in 10 set 2014) come segue

> Potresti anche dire, non inserire funzioni che cambiano per motivi diversi nella stessa classe.

o

> Non mescolare le preoccupazioni nei tuoi corsi

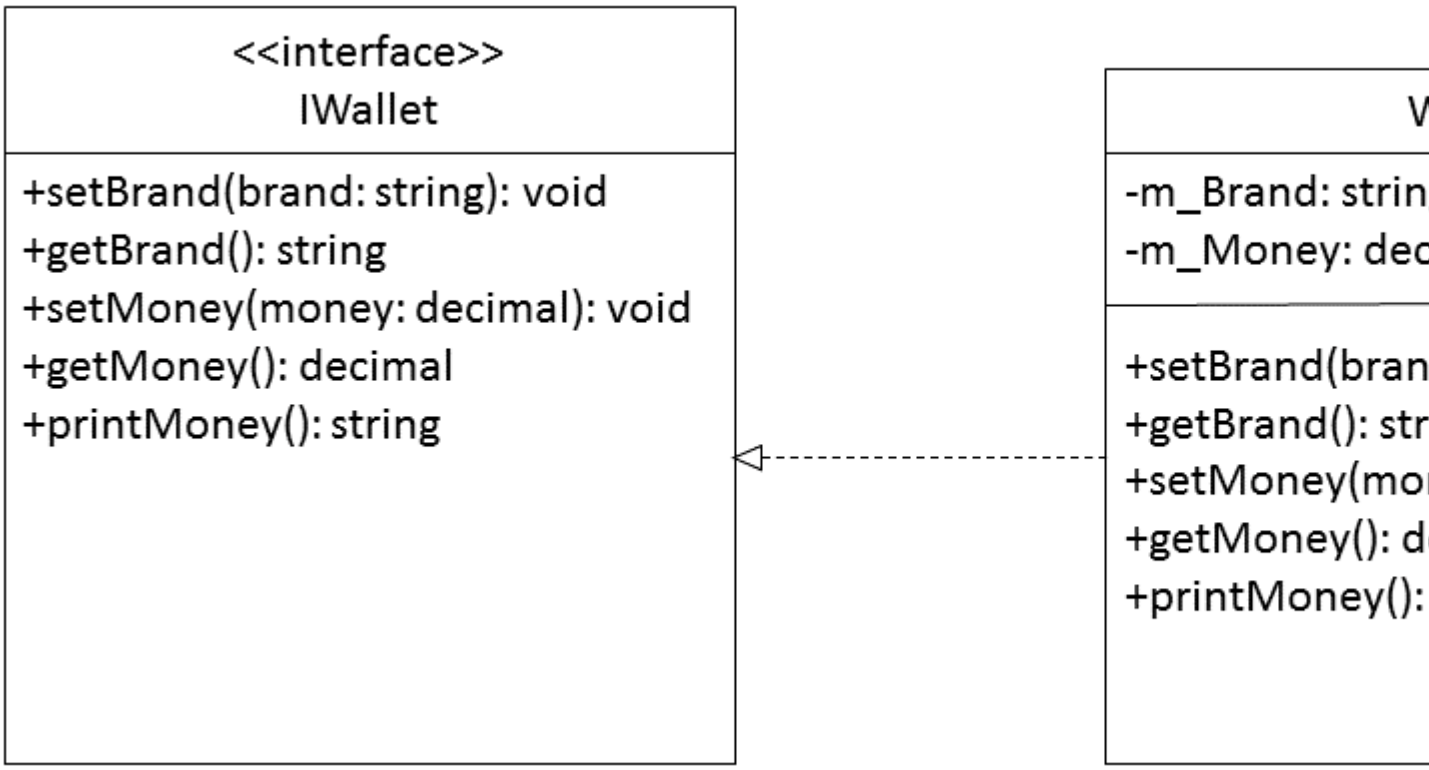
Motivo per applicare l'SRP:

Quando si modifica una classe, si potrebbe influire sulla funzionalità correlata ad altre responsabilità della classe. Mantenere le responsabilità a un livello basso riduce al minimo il rischio di effetti collaterali.

Cattivo esempio

Abbiamo un'interfaccia IWallet e una classe Wallet che implementa l'IWallet. Il Portafoglio tiene i nostri soldi e il marchio, inoltre deve stampare i nostri soldi come rappresentazione delle stringhe. La classe è usata da

1. un servizio web
2. un autore di testi che stampa i soldi in euro in un file di testo.



L'SRP è qui violato perché abbiamo due preoccupazioni:

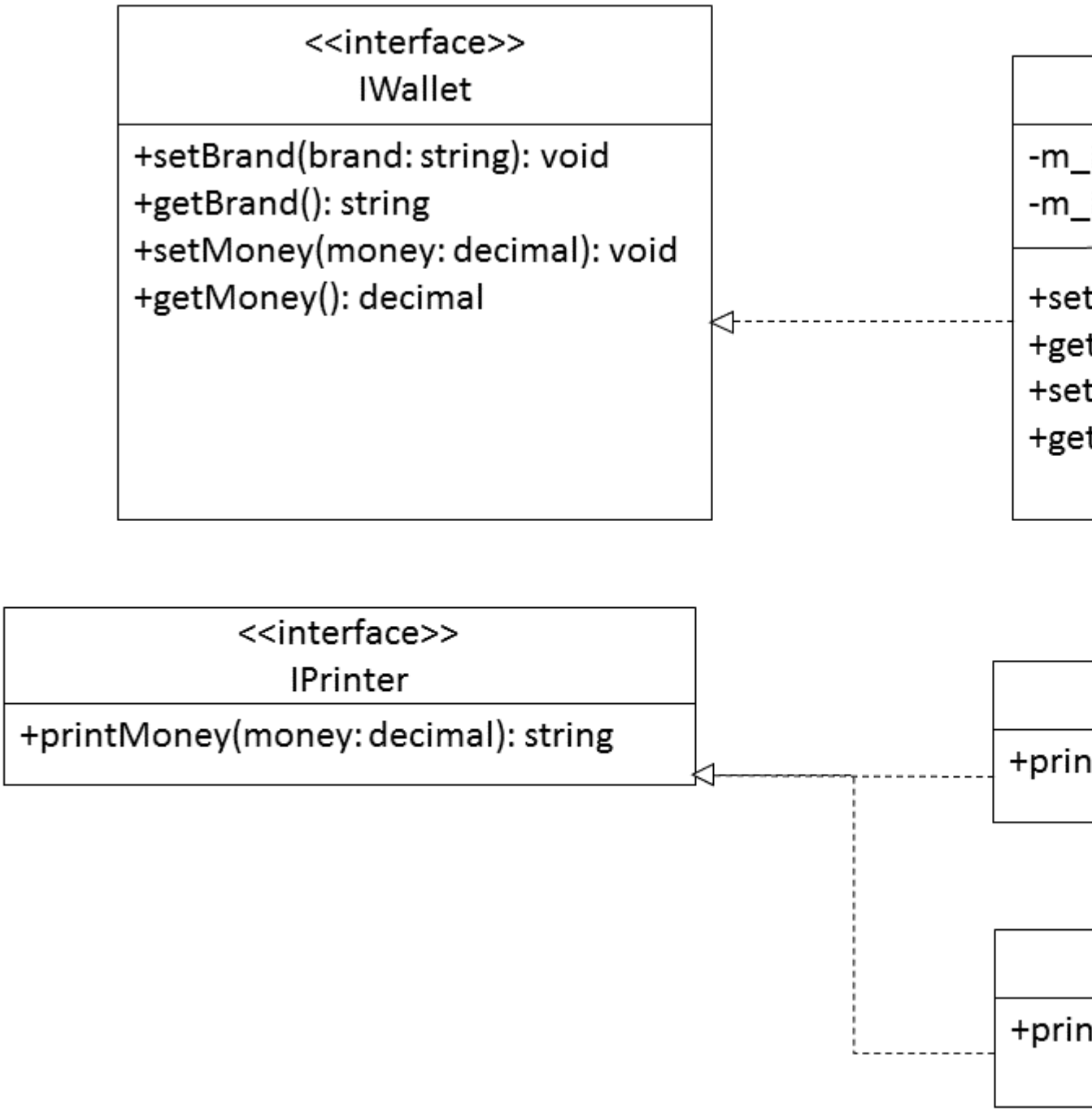
1. La memorizzazione dei soldi e del marchio
2. La rappresentazione del denaro.

Codice di esempio 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();
    }
}
```

Buon esempio



Per evitare la violazione `printMoney` , abbiamo rimosso il metodo `printMoney` dalla classe Wallet e lo abbiamo inserito in una classe Printer. La classe Printer è ora responsabile della stampa e Wallet è ora responsabile della memorizzazione dei valori.

Codice di esempio 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;
    }
}
```

Leggi SOLIDO online: https://riptutorial.com/it/design-patterns/topic/8651/solido

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con i modelli di progettazione	Community, Ekin, Mateen Ulhaq, meJustAndrew, Sahan Serasinghe, Saurabh Sarode, Stephen C, Tim, ɬolɐəz əɥɬ qoq
2	Adattatore	avojak, Ben Rhys-Lewis, Daniel Käfer, deHaar, Thijs Riezebeek
3	Apri Close Principle	Mher Didaryan
4	carico pigro	Adhikari Bishwash
5	Catena di responsabilità	Ben Rhys-Lewis
6	Fabbrica	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
7	Facciata	Kritner, Makoto, Ravindra babu
8	Iniezione di dipendenza	Kritner, matiaslauriti, user2321864
9	lavagna	Leonidas Menendez
10	Metodo del modello	meJustAndrew, Ravindra babu
11	Metodo di fabbrica statico	abbath, Bongo
12	Modello composito	Krzysztof Branicki
13	Modello costruttore	Alexey Groshev, Arif, Daniel Käfer, fgb, Kyle Morgan, Ravindra babu, Sujit Kamthe, uzilan, Vasiliy Vlasov, yitzih
14	Modello del mediatore	Ravindra babu, Vasiliy Vlasov
15	Modello di comando	matiaslauriti, Ravindra babu, Vasiliy Vlasov
16	Modello di ponte	Mark, Ravindra babu, Vasiliy Vlasov
17	Modello di prototipo	Arif, Jarod42, user2321864

18	Modello di strategia	Aseem Bansal, dimitrisli, fabian, M.S. Dousti, Marek Skiba, matiaslauriti, Ravindra babu, Shog9, SjB, Stephen C, still_learning, uzilan
19	Modello di visitatore	Daniel Käfer, Jarod42, Loki Astari, Ravindra babu, Stephen Leppik, Vasiliy Vlasov
20	Modello oggetto nullo	Jarod42, weston
21	Monostate	skypjack
22	Motivo decorativo	Arif, Krzysztof Branicki, matiaslauriti, Ravindra babu
23	multiton	Kid Binary
24	MVC, MVVM, MVP	Daniel LIn, Jompa234, Stephen C, user1223339
25	Osservatore	Arif, user2321864, uzilan
26	Pattern del repository	bolt19, Leifb
27	Pattern di progettazione Data Access Object (DAO)	Pritam Banerjee
28	Pattern Iterator	bw_üezi, Dave Ranjan, Jeeter, Stephen C
29	Publish-Subscribe	Arif, Jeeter, Stephen C
30	Singleton	Bongo, didiz, DimaSan, Draken, fgb, Gul Md Ershad, hellyale, jefry jacky, Loki Astari, Marek Skiba, Mateen Ulhaq, matiaslauriti, Max, Panther, Prateek, RamenChef, Ravindra babu, S.L. Barth, Stephen C, Tazbir Bhuiyan, Tejus Prasad, Vasiliy Vlasov, volvis
31	SOLIDO	Bongo