**Visitor Pattern**

Saluti, colleghi sviluppatori! Bentornati a .Net Core Central. L'esplorazione di oggi ci porta nell'intricato mondo dei design pattern, con un focus sul Visitor Design Pattern, una gemma all'interno dei design pattern Gang of Four.

**Unveiling the Visitor Pattern**

Il Visitor Design Pattern, incastonato nel regno dei behavioral design pattern, è un potente strumento per rappresentare le operazioni eseguite sugli elementi di una struttura di oggetti. Questo pattern porta flessibilità al tavolo consentendoci di definire nuove operazioni senza alterare le classi degli elementi su cui opera.

**Navigating the Intent of Visitor Pattern**

L'obiettivo primario del Visitor Design Pattern è quello di fornire un mezzo per rappresentare le operazioni da eseguire sugli elementi di una struttura di oggetti. Ciò avviene introducendo un visitatore che definisce queste operazioni, separando così le preoccupazioni e promuovendo un design più modulare ed estensibile. Ora, immergiamoci in un esempio pratico per cogliere l'essenza del Visitor Design Pattern.

**The Notification Conundrum**

Immagina uno scenario in cui un sistema deve inviare vari tipi di notifiche: fatture, materiali di marketing o avvisi di scadenza di abbonamenti. Ogni tipo di notifica richiede contenuti e logiche differenti. Inizialmente, il sistema supporta notifiche via e-mail e SMS, ciascuna implementata tramite un'interfaccia denominata INotificationSender.

namespace PatternDemo.VisitorPattern.Demo

{

internal interface INotificationSender

{

void Send(string message);

}

}

Tuttavia, sorge un problema evidente quando si contempla l'aggiunta di nuovi protocolli di notifica, come WebSocket o notifiche mobili tramite Firebase. L'implementazione esistente richiede modifiche per accogliere questi nuovi protocolli, violando il principio di progettazione aperta-chiusa

**Refining the Solution with Visitor Pattern**

Per risolvere questo problema, ristrutturiamo la nostra implementazione utilizzando il Visitor Design Pattern. L'idea di base è quella di estrarre le responsabilità di configurazione per ogni protocollo di notifica in classi di visitatori separate. Ciò garantisce che l'aggiunta di nuovi protocolli non avrà alcun impatto sui mittenti di notifiche esistenti. Introduciamo un nuovo metodo, Accept, nell'interfaccia INotificationSender, che accetta un IVisitor. Ciò prepara il terreno per i visitatori per definire le operazioni di configurazione.

namespace PatternDemo.VisitorPattern.Demo

{

internal interface INotificationSender

{

void Send(string message);

void Accept(IVisitor visitor);

}

}

**Crafting the Visitors**

**Emergono due classi di visitatori: EmailVisitor e TextVisitor, ciascuna responsabile della configurazione rispettivamente dei protocolli email e di testo. Ciò suddivide ordinatamente la logica di configurazione. Ora, aggiungere un nuovo protocollo di notifica, come WebSocket, diventa un gioco da ragazzi. Creiamo una classe WebsocketVisitor senza modificare i mittenti di notifica esistenti.**

|  |
| --- |
| **namespace VisitorPattern.Demo;**    **internal interface IVisitor**  **{**  **void Visit(INotificationSender notificationSender);**  **}**    **internal class EmailVisitor : IVisitor**  **{**  **public void Visit(INotificationSender notificationSender)**  **{**  **Console.WriteLine("Setup email");**  **}**  **}**    **internal class TextVisitor : IVisitor**  **{**  **public void Visit(INotificationSender notificationSender)**  **{**  **Console.WriteLine("Setup text");**  **}**  **}**    **internal class WebsocketVisitor : IVisitor**  **{**  **public void Visit(INotificationSender notificationSender)**  **{**  **Console.WriteLine("Setup websocket");**  **}**  **}** |

**Embracing Flexibility with Visitors**

**Nella dimostrazione finale, abbiamo visto con quanta facilità possiamo adattare il nostro sistema per supportare diversi protocolli di notifica. Il metodo Accept si integra perfettamente con diversi visitatori, consentendo a ogni mittente di notifica di essere configurato in modo indipendente. Per i mittenti di notifica, possiamo averne uno per Invioce e uno per Marketing.**

**namespace VisitorPattern.Demo;**

**internal class InvoiceNotificationSender : INotificationSender**

**{**

**public void Accept(IVisitor visitor)**

**{**

**visitor.Visit(this);**

**}**

**public void Send(string message)**

**{**

**Console.WriteLine($"Notification sent: {message}");**

**}**

**}**

**internal class MarketingNotificationSender : INotificationSender**

**{**

**public void Accept(IVisitor visitor)**

**{**

**visitor.Visit(this);**

**}**

**public void Send(string message)**

**{**

**Console.WriteLine($"Notification sent: {message}");**

**}**

**}**

**Conclusion: Empowering Designs with Visitors**

**Il Visitor Design Pattern dimostra la sua abilità nel migliorare la flessibilità e la manutenibilità del nostro codice. Delegando le responsabilità di configurazione ai visitatori, aderiamo al principio di separazione delle preoccupazioni, rendendo il nostro sistema più adattabile ai cambiamenti futuri. E questo conclude il nostro viaggio nel Visitor Design Pattern nello sviluppo .NET. Se hai trovato questa esplorazione illuminante, dai un'occhiata al resto dei miei post del blog**