**Template Method Pattern**

Saluti, colleghi sviluppatori! Bentornati a .Net Core Central. Nel post del blog di oggi, ci imbarchiamo in un viaggio nel regno dei design pattern, concentrando la nostra attenzione sul Template Method Design Pattern, un potente membro dei design pattern Gang of Four.

Understanding the Essence of Template Method Pattern

Il Template Method Design Pattern, incastonato nei behavioral design pattern, è uno strumento versatile per la creazione di algoritmi con vari passaggi. In sostanza, questo pattern cerca di definire lo scheletro di un algoritmo all'interno di un'operazione, consentendo alle sottoclassi di implementare o ridefinire passaggi specifici senza alterare la struttura complessiva dell'algoritmo. Il template method crea essenzialmente un progetto per un algoritmo, con le sottoclassi che forniscono le loro esclusive modifiche a determinati passaggi.

**Demystifying the Template Method Pattern**

Per illustrare questo concetto in un contesto di programmazione, approfondiamo un esempio che coinvolge un semplice carrello della spesa con un processo di pagamento. Il processo di pagamento comprende più passaggi e, all'interno di questo, un passaggio si distingue come metodo modello. In questo scenario, stiamo implementando il processo di pagamento utilizzando sia il Template Method Design Pattern sia aderendo ai principi di progettazione SOLID.

**Template Method Pattern Implementation**

**Nell'approccio tradizionale del metodo template, creiamo una classe astratta denominata Cart con un metodo denominato Checkout. Questo metodo consiste di tre passaggi: convalida degli articoli, esecuzione del pagamento e invio di una ricevuta. Mentre i primi due passaggi sono implementati direttamente nella classe Cart, il terzo passaggio, l'invio di una ricevuta, è contrassegnato come metodo astratto, in attesa di implementazione da parte delle sottoclassi**

**internal abstract class Cart**

**{**

**public void Checkout()**

**{**

**ValidateItem();**

**ExecutePayment();**

**SendReceipt();**

**}**

**protected abstract void SendReceipt();**

**private void ExecutePayment()**

**{**

**Console.WriteLine("Payment executed");**

**}**

**private void ValidateItem()**

**{**

**Console.WriteLine("Item validated");**

**}**

**}**

**Successivamente, creiamo due sottoclassi: EmailCart e TextCart. Ognuna di queste sottoclassi fornisce la sua implementazione unica per l'invio di ricevute tramite e-mail e messaggi di testo, rispettivamente.**

**internal class EmailCart : Cart**

**{**

**protected override void SendReceipt()**

**{**

**Console.WriteLine("Email Sent");**

**}**

**}**

**internal class TextCart : Cart**

**{**

**protected override void SendReceipt()**

**{**

**Console.WriteLine("Text Sent");**

**}**

**}**

**Implementation of using the Carts**

**var cart = new EmailCart();**

**cart.Checkout();**

**var cart1 = new TextCart();**

**cart1.Checkout();**

**SOLID Design Principles Approach**

**Nell'approccio dei principi di progettazione SOLID, introduciamo tre interfacce: IItemValidator, IPaymentExecutor e IReceiptSender. La classe Cart ora prende istanze di queste interfacce tramite il suo costruttore, consentendo una progettazione più modulare e flessibile.**

**namespace PatternDemo.TemplateMethodPattern.Demo.SOLID**

**{**

**public interface IItemValidator**

**{**

**void ValidateItem();**

**}**

**}**

**namespace PatternDemo.TemplateMethodPattern.Demo.SOLID**

**{**

**public class ItemValidator : IItemValidator**

**{**

**public void ValidateItem()**

**{**

**Console.WriteLine("Item validated");**

**}**

**}**

**}**

**namespace TemplateMethodPattern.Demo.SOLID;**

**public interface IPaymentExecutor**

**{**

**void ExecutePayment();**

**}**

**public class PaymentExecutor : IPaymentExecutor**

**{**

**public void ExecutePayment()**

**{**

**Console.WriteLine("Payment executed");**

**}**

**}**

**public interface IReceiptSender**

**{**

**void SendReceipt();**

**}**

**public class EmailReceiptSender : IReceiptSender**

**{**

**public void SendReceipt()**

**{**

**Console.WriteLine("Email Sent");**

**}**

**}**

**public class TextReceiptSender : IReceiptSender**

**{**

**public void SendReceipt()**

**{**

**Console.WriteLine("Text Sent");**

**}**

**}**

**Ogni interfaccia ha un singolo metodo che rappresenta la sua responsabilità: ValidateItem, ExecutePayment e SendReceipt. Creiamo quindi classi concrete che implementano queste interfacce: ItemValidator, PaymentExecutor, EmailReceiptSender e TextReceiptSender. Nella classe Cart, il metodo Checkout ora richiama i metodi da queste interfacce, fornendo un'implementazione diversa del processo di invio della ricevuta in base all'istanza IReceiptSender iniettata.**

**internal class Cart**

**{**

**private readonly IItemValidator itemValidator;**

**private readonly IPaymentExecutor paymentExecutor;**

**private readonly IReceiptSender receiptSender;**

**public Cart(IItemValidator itemValidator,**

**IPaymentExecutor paymentExecutor,**

**IReceiptSender receiptSender)**

**{**

**this.itemValidator = itemValidator;**

**this.paymentExecutor = paymentExecutor;**

**this.receiptSender = receiptSender;**

**}**

**public void Checkout()**

**{**

**itemValidator.ValidateItem();**

**paymentExecutor.ExecutePayment();**

**receiptSender.SendReceipt();**

**}**

**var itemValidator = new ItemValidator();**

**var paymentExecutor = new PaymentExecutor();**

**var emailReciptSender = new EmailReceiptSender();**

**var textReceiptSender = new TextReceiptSender();**

**var cart = new Cart(itemValidator, paymentExecutor, emailReciptSender);**

**cart.Checkout();**

**cart = new Cart(itemValidator, paymentExecutor, textReceiptSender);**

**cart.Checkout();**

**Comparing Approaches**

**Entrambi gli approcci producono lo stesso risultato: un processo di checkout flessibile con meccanismi di invio di ricevute variabili. La scelta tra il metodo tradizionale del modello e l'approccio dei principi di progettazione SOLID dipende dai requisiti specifici del progetto e dal livello di flessibilità desiderato.**

**Conclusion: Tailoring Algorithms with Templates**

**Il Template Method Design Pattern consente agli sviluppatori di creare algoritmi robusti, adattandosi alla variabilità in determinati passaggi. Sia che tu scelga il tradizionale metodo template o che tu adotti i principi di progettazione SOLID, la chiave sta nel creare soluzioni flessibili e gestibili. Con questo si conclude la nostra esplorazione del Template Method Design Pattern in .NET Core. Se hai trovato questo viaggio illuminante, non dimenticare di leggere il resto dei miei post del blog.**