Design Pattern

January 28, 2024

Contents

1 Design Pattern Creazionali

Forniscono soluzioni per istanziare oggetti o gruppi di oggetti in modo flessibile e riutilizzabile.

1.1 Singleton

1.1.1 Scopo

Evitare di dover creare piu istanze di una classe, garantendo che esista una sola istanza e fornendo un punto di accesso globale a tale istanza.

1.1.2 Struttura

- campo privato e statico che contiene l'istanza
- costruttore privato
- metodo pubblico e statico che restituisce l'istanza (getInstance())

1.1.3 Problematiche

Nasconde le dipendenze tra classi client e il metodo statico, i metodi statici danno problemi con il testing.

1.2 Builder

1.2.1 Scopo

Separare la costruzione di un oggetto complesso (ha tante dipendenze) dalla sua rappresentazione

1.2.2 Struttura

- Rendo protetto il costruttore della classe complessa
- Costruisco la classe Builder che ha gli stessi attributi della classe complessa
- Builder ha metodi accomulatori (definiti con with) per settare gli attributi e ritorna se stesso
- Builder ha un metodo build che ritorna un'istanza della classe complessa
- Da fuori in una funzione che resituisce l'oggetto complesso creo un'istanza di Builder e setto gli attributi con i metodi with, infine chiamo il metodo build e ritorno la classe complessa

1.2.3 Problematiche

Mi aspetto un accoppiamento forte ma va bene

1.3 Abstract Factory

1.3.1 Scopo

I client richiedono la costruzione non solo di un oggetto ma di un gruppo di oggetti che lavorano insieme

1.3.2 Struttura

- Interfaccia astratta Factory che definisce le operazioni che creano i prodotti astratti
- Classe concreta che implementa Factory che definisce le operazioni che creano i prodotti concreti

1.3.3 Problematiche

Complesso supportare un nuovo prodotto, richiede di modificare la Factory e le sue implementazioni

2 Design Pattern Strutturali

Sono focalizzati sulle dipendenze degli aggetti affichè abbiano certe caratteristiche

2.1 Adapter

2.1.1 Scopo

Consente di adattare l'interfaccia di una libreria esterna (Adaptee) ad un'interfaccia che mi serve (Target) per poterla usare all'interno del mio sistema

2.1.2 Struttura

Due modi per implementarlo:

- **Adapter di classe**: eredito dalla libreria (Adaptee) e implemento l'interfaccia che mi serve (Target) in una classe ClassAdapter
- **Adapter di oggetto**: creo un oggetto che implementa l'interfaccia (Target) che mi serve e uso Adaptee come attributo

2.1.3 Problematiche

Piu metodi da implementare, piu complessità

2.2 Decorator

2.2.1 Scopo

Aggiungere responsabilità ad un oggetto dinamicamente (senza utilizzo di ereditarietà). Permette di aggiungere funzioni e funzionalità ad una funzionalità base.

2.2.2 Struttura

- **Component**: interfaccia che definisce l'oggetto base (Pizza)
- **ConcreteComponent**: implementazione dell'interfaccia Component (BasePizza)
- **Decorator**: classe astratta che implementa l'interfaccia Component e ha un attributo di tipo Component (Topped Pizza)
- **ConcreteDecorator**: implementazione di Decorator (Pizze concrete con aggiunte)

2.2.3 Problematiche

Tante classi molto simili e piccole possono risultare complesse da testare in isolamento

2.3 Facade

2.3.1 Scopo

Permette di fornire un'interfaccia unica (accesso semplice) per un sottosistema complesso, ma non nosconde completamente le funzionalità del sottosistema. Accentra le dipendenze

2.3.2 Struttura

- **Facade**: classe che fornisce l'interfaccia unica
- **Subsystem**: classi che implementano le funzionalità del sottosistema

2.3.3 Problematiche

Da gestire molte dipendenze. Rischio di introdurre bug. Rischio di classe grande. Deve fare un'interfaccia stabile.

3 Design Pattern Comportamentali

Table 1: Design Pattern Software

Nome	Descrizione