Design Pattern

January 30, 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.

2.4 Proxy

2.4.1 Scopo

Fornire un oggetto (un altro tipo) sul quale si vogliono fornire caratteristiche ulteriore per l'accesso dei suoi metodi.

2.4.2 Struttura

Virtual Proxy: crea oggetti complessi on-demand (quando servono, caricamento in memoria lazy)
Remote Proxy: si fa vedere locale un oggetto che in realtà è remoto (sfruttando la rete)
Protection Proxy: permetto l'accesso ad alcune funzionalità di alcuni tipi solo a solo a certi utenti (client), fa il controllo dei diritti di accesso
Puntatori intelligenti: tengono traccia degli oggetti istanziati

- **Subject**: interfaccia che definisce l'oggetto base
- **RealSubject**: implementazione dell'interfaccia Subject
- **Proxy**: classe che implementa l'interfaccia Subject e ha un attributo di tipo Subject (RealSubject)

3 Design Pattern Comportamentali

Forniscono dipendenze tra le classi in modo da strutturare il comportamento di tipi (la parte di business)

3.1 Command

3.2 Observer

3.2.1 Scopo

Aggiornare o far reagire automaticamente un insieme di oggetti quando un altro oggetto cambia stato o viene modificato. Quindi vuole:

- Avere un sistema di notifica di eventuali modifiche
- Non replicare questo sistema di notifica in ogni classe
- definire una dipendenza uno a molti tra oggetti
- creare massimo disaccoppiamento tra colore che cambiano il loro stato (Subject) e colore che devono essere notificati (Observer)

3.2.2 Struttura

- Subject: classe astratta (mai interfaccia!) che contiene come attributo una lista di Observer, definisce le operazioni per aggiungere e rimuovere Observer (quindi permette ad un Observer di registrarsi e di cancellarsi alle notifiche di questa classe). Contiene poi un metodo notify() (implementato) che notifica tutti gli Observer registrati
- ConcreteSubject: implementa le operazioni di Subject
- Observer: interfaccia (deve esserlo) che definisce l'operazione di aggiornamento con il metodo update() astratto
- ConcreteObserver: implementa l'operazione di aggiornamento implementa update() chiedendo lo stato del ConcreteSubject

3.2.3 Problematiche

Non tutti gli Observer sono interessati alle medesime informazioni (si usano i Topic). Si può notificare ogni tot aggiornamenti dello stato per non dover notificare ogni volta.

3.3 Strategy

3.3.1 Scopo

Definisce una famiglia di algoritmi che operano tutti allo stesso modo, esponendo alla medesima interfaccia. In questo modo è possibile cambiare l'algoritmo senza dover cambiare la classe che lo usa. Differenti varianti dello stesso algoritmo.

3.3.2 Struttura

- Strategy: interfaccia che definisce l'operazione comune a tutti gli algoritmi, modo unico per accedere all'algoritmo
- ConcreteStrategy: implementa l'operazione comune a tutti gli algoritmi
- Context: classe che usa l'algoritmo, contiene un riferimento a Strategy e definisce un metodo per impostare la Strategy (rappresenta gli input dell'algoritmo), di solito è un interfaccia

3.4 Template Method

3.4.1 Scopo

Riutilizzo il workflow (lo scheletro) dell'algoritmo ma l'implementazione delle singole operazioni può variare in qualche modo.

3.4.2 Struttura

- AbstractClass: classe astratta che definisce il template method (che contiene il workflow dell'algoritmo) e le operazioni primitive astratte (che sono le operazioni che variano)
- ConcreteClass: implementa le operazioni primitive

3.4.3 Problematiche

Usa l'ereditarietà anzichè la composizione

Table 1: Design Pattern Software

Nome	**Descrizione**