Prima di tutto facciamo distinzione tra i due strumenti:

1. Maven: Ti permette di automatizzare la gestione di una struttura di un progetto, gestire dipendenze, compilazione e packaging (tipo creare un .jar). Il file principale dalla quale buildare il progetto è il pom.xml, dove appunto sono automaticamente inserite tutte le dipendenze ecc. Permette di dichiarare librerie, compilare codice, eseguire test ed eseguire comandi mvn clean install.

2. Spring: E un framework Java, cioè una piattaforma di sviluppo con lo scopo di scrivere applicazioni Java moderne, sicure, scalabili e modulari. Spring Boot è una versione semplificata e automatica di spring per creare più facilmente app web e REST API.

# SPRING

Il metodo di approccio usato da spring è un approccio "Convention Over Configuration", grazie a questo approccio, seguendo le convenzioni, spring sa già cosa fare e noi non dobbiamo configurare nulla manualmente. I vantaggi sono ovviamente:

1. meno configurazione
2. avvio rapido
3. meno codice
4. più produttività

Inoltre, nonostante le convenzioni, possiamo sovrascriverle per avere un comportamento personalizzato.

2 Concetti Chiave

(Inversion Of Control & Dependency Injection)

Inversion Of Control: è un principio chiave dell'ingegneria del software che trasferisce il controllo di oggetti o parti di un programma a un contenitore o framework. Concediamo quindi al framework di assumere il controllo del flusso di un programma e di effettuare chiamate al nostro codice, utilizzando delle astrazioni con comportamenti personalizzati. Abbiamo dei Pattern che si occupano di fare ciò e sono:

1. Strategy
2. Service
3. Factory
4. Dependency Injection

Ad esempio, se noi abbiamo una classe Veicolo che all'interno ha un’istanza motore scritta come sappiamo abbiamo creato una forte dipendenza, automaticamente non possiamo utilizzare la nostra classe Veicolo senza portarci dietro anche il motore. E proprio queste dipendenze sono gestite da Spring tramite l'Inversion of Control. Analizziamo ora la Dependency Injection che, come abbiamo capito, è una possibile implementazione dell'Inversion, una modalità che ci permette di utilizzare una classe esterna. Il Framework crea l'istanza motore e la "Inietta" all'interno della classe veicolo senza richiamare nessuno metodo ma solo specificando la richiesta. Questa Injection può avvenire attraverso un costruttore, un metodo setter, un field ecc questo ci permette di disaccoppiare le classi (possiamo quindi portarci la classe macchina senza la classe motore).

|  |  |  |
| --- | --- | --- |
|  | Uses | Service |
|  |



@Bean

Sono oggetti gestiti dal container di Spring. Quindi quando annoti una classe con @component, @service, @repository, @controller ecc stai dicendo a Spring di occuparsene lui per eventuali Injection ecc. Abbiamo visto che un esempio di @bean è l'oggetto emailservice creato automaticamente appena parte l'app.

# CONTAINER DI SPRING

Il container di Spring è il cuore del framework. Usa i metadati di configurazione per gestire i bean. Infatti, il container gestisce i cicli di vita degli oggetti, appunto chiamati beans e le loro dipendenze.

|  |
| --- |
| Spring 10C Container Java Concept Of The Day |
| |  | | --- | | Configuration Metadata |   XML—basedAnnotation-based  Java-based |

# Analizziamo ora la struttura del progetto:

* .idea contiene un sacco di configurazione che non dovremmo mai toccare
* .mvn contiene la configurazione mvn, compreso il wrapper che contiene la versione di mvn. Abbiamo anche la 2 shell
* Pom che è il cuore praticamente, qui abbiamo tutti gli attributi e i dati riguardanti il progetto
* In java abbiamo 2 cose importanti 1°application.properties che sono delle coppie 2° Springbootapp che è il punto d’ingresso della nostra applicazione

Nel pom dobbiamo stare attenti alle dipendenze che sono molte importanti

@controller è il tag per identificare un gestore del traffico web

@Configuration: contrassegna la classe come origine delle definizioni di bean per il contesto dell'applicazione

@EnableAutoConfiguration: Indica a Spring Boot di iniziare ad aggiungere bean in base alle impostazioni del classpath, ad altri bean e a varie impostazioni delle proprietà.

@ComponentScan: Indica a Spring di cercare altri componenti, configurazioni e servizi nel pacchetto com/example, consentendogli di trovare i controller.

@springapplicationrun è il metodo che lancia l’applicazione

# APPLICATION SERVER E WEB SERVER

Lo scopo principale del web server è quello di gestire le richieste HTTP, servire contenuti statici quali (HTML, CSS, JS) (un esempio è Apache). Lo scopo principale invece dell'application server è quello di eseguire la logica applicatica e gestire contenuti dinamici (Servlet, Jsp, business logic) (un esempio è Jboss o WildFly). Il Web server non supporta framework, mentre l'application server supporta framework come Spring o Jakarta EE. Un esempio pratico di questi casi può essere nel caso del web server mostrare un sito statico, mentre nel caso di un application server può essere un login. Lavorano spesso insieme: il web server riceve la richiesta, che verrà girata all'application server se servono operazioni logiche, a questo punto eseguirà il codice e restituirà la risposta al web server che poi la invierà al client. ln ambienti di programmazioni moderni abbiamo la presenza di TOMCAT che funge da entrambi.

## JUnit

E I un framework molto usato per testare codice Java, fondamentale per scrivere piccoli testi di verifica per classi o metodi. Questo ci consente di individuare errori prima che vadano in produzione e avere un codice migliore. Con maven basta inserire all'interno del pom.xml una dipendenza specifica e poi eseguire da terminale mvn test. Ci sono vari comandi per eseguire test più specifici.

Cos’è un BATCH

Un batch è un programma pensato per essere eseguito senza interazione dell’utente, spesso per elaborare grandi quantità di dati, come:

* import/export da file
* aggiornamento di record nel database
* elaborazioni programmate (tipo cron)

Può essere un semplice programma main, oppure usare framework come Spring Batch se la logica è più complessa.

API FIRST

OpenAPI è una specifica per definire API REST in un formato leggibile da macchine, tipicamente in YAML o JSON. Un documento OpenAPI descrive l'API, inclusi i suoi endpoint, metodi, parametri, schemi di risposta, e dettagli di autenticazione. Questo permette la generazione automatica di documentazione, client e stub del server, facilitando lo sviluppo e l'integrazione.

In dettaglio, OpenAPI, definisce il formato:

Utilizza YAML o JSON per descrivere l'API, rendendo il documento leggibile sia per gli umani che per le macchine.

Differenza tra SOAP e REST

I servizi SOAP (Simple Object Access Protocol) sono un tipo di servizio web che permette a due applicazioni diverse (anche scritte in linguaggi differenti e su sistemi diversi) di comunicare tra loro tramite il protocollo HTTP, usando messaggi XML strutturati., mentre il REST è uno stile architetturale per progettare servizi web leggeri e facilmente accessibili tramite il protocollo HTTP (lo stesso usato dai browser).

Differenze:

SOAP REST

XML obbligatorio JSON o XML

Più formale e verboso. Più leggero e flessibile

Usa WSDL Usa URL e metodi HTTP

Usato in ambienti enterprise // nel web moderno/microservizi

GIT

Git: permette di poter lavorare sul codice sorgente senza la necessità di un server centralizzato, permettendo agli sviluppatori di scaricare una copia integrale del progetto per poter lavorare e condividere le modifiche anche su diverse versioni (o meglio, rami) di quest’ultimo.

Terminologia:

REPOSITORY: altro non è che la copia sincronizzabile della directory principale del progetto, mettono a disposizione un hosting per i repository, permettendo di controllare gli accessi e agevolando il lavoro in team.

COMMIT: (tradotto letteralmente “consegnare”) è l’elenco delle modifiche effettuate ad un progetto, validate e “affidate” a git per essere inviate al repository principale.

PUSH E PULL: (tradotti letteralmente “spingere” e “tirare”) sono due comandi di git che servono per inviare e ricevere gli aggiornamenti. Il push invierà i tuoi commit al repository principale, il pull chiederà a

quest'ultimo di poter scaricare le modifiche sul tuo repository locale e unirle ai tuoi file. Il pull è altresì l’unione del comando fetch e del comando merge.

FETCH: Il fetch (tradotto letteralmente “raggiungere, andare a prendere”) è il comando che dice a Git di recuperare il contenuto di un repository. Generalmente viene utilizzato il comando pull, che combina appunto il fetch (“andare a prendere”) e il comando MERGE (“unione”, cioè l'unione tra commit di brach differenti).

CD: change directory

Directory -> folder

CLI-> command line interface

github è il sito web che fa da host ai nostri git

Clone-> è il comando per copiare un repository che è presente online ma che non è presente sulla nostra macchina

ADD -> per caricare

Commit -> salva i nostri cambiamenti

Riassunto comandi:

repo -> repository

clone -> bring a repo down from the internet (remote repository like Github) to your local machine

add -> track your files and changes with Git

commit -> save your changes into Git

push -> push your changes to your remote repo on Github (or another website)

pull -> pull changes down from the remote repo to your local machine

status -> check to see which files are being tracked or need to be commited

init -> use this command inside of your project to turn it into a Git repository and start using Git with that codebase.

PATTERN

Cos’è un design pattern? È un modello che può essere personalizzato per risolvere ricorrenti problemi di design all’interno del codice. I design pattern sono un insieme di soluzioni collaudate a problemi comuni nella progettazione del software. Essi definiscono un linguaggio comune che i programmatori possono usare per comunicare in modo più efficiente.

I pattern a basso livello vengono definiti idiomi, mentre quelli ad alto livello, per lo più, architetturali. In aggiunta tutti i pattern vengono categorizzati in base ai loro intenti, principalmente 3 gruppi principali:

* Pattern creazionali: forniscono meccanismi di creazione di oggetti che aumentano la flessibilità e il riutilizzo del codice esistente.
* Pattern strutturali: aiutano ad unire classi e oggetti in strutture più grandi mantenendola flessibile ed efficiente.
* Pattern comportamentali: garantiscono l’efficacia della comunicazione e l’assegnazione di responsabilità tra oggetti.

FACTORY METHOD

Il Metodo Factory separa il codice di costruzione del prodotto dal codice che effettivamente utilizza il prodotto. Pertanto, è più facile estendere il codice di costruzione del prodotto in modo indipendente dal resto del codice. È utile quando vuoi consentire a chiunque di sovrascrivere questo metodo, oltre a estendere il componente stesso. Utilizzare il metodo Factory quando si desidera risparmiare risorse di sistema riutilizzando oggetti esistenti anziché ricostruirli ogni volta. Prendiamo ora un esempio: Se creo un programma per le spedizioni di soli camion, ma mi rendo conto che non solo i camion la vogliono usare io faccio un’interfaccia principale del mezzo di trasporto e poi lascio alle sue sottoclassi la creazione di navi, auto ecc. Così ogni qual volta dovrò aggiungere un mezzo, non dovrò più modificare l’interfaccia principale ma mi basterà aggiungere una sottoclasse. Questo potrebbe essere anche l’unico problema in quanto si potrebbe avere la necessità di avere tante sottoclassi.

Abstract Factory

Factory è un modello di progettazione creazionale che consente di produrre famiglie di oggetti correlati senza specificarne le classi concrete. Utilizza l'Abstract Factory quando il tuo codice deve funzionare con diverse famiglie di prodotti correlati, ma non vuoi che dipenda dalle classi concrete di tali prodotti: potrebbero essere sconosciute in precedenza o semplicemente vuoi consentire un'estensibilità futura. L'Abstract Factory fornisce un'interfaccia per creare oggetti da ciascuna classe della famiglia di prodotti. Finché il tuo codice crea oggetti tramite questa interfaccia, non devi preoccuparti di creare la variante sbagliata di un prodotto che non corrisponde ai prodotti già creati dalla tua app. L’esempio che abbiamo visto è un negozio che ha diverse tipologie dello stesso prodotto, tipo un sofà che può essere antico, moderno, futuristico. Si crea l’interfaccia principale e si lascia alle sottoclassi la creazione delle varie tipologie. Anche in questo caso l’unico lato negativo potrebbe essere la necessità di avere innumerevoli sottoclassi. La differenza che notiamo è che il Factory pensa ad essere utile per un oggetto, mentre l’abstract per famigli di oggetti simili.

Builder

Immaginiamo di dover costruire una casa, e di doverlo fare più volte, dovremmo ogni volta creare i vari componenti della casa. Con questo pattern invece creiamo un’interfaccia, che contiene tutti i possibili componenti della casa, che sarà a disposizione di un director che si occuperà della creazione step by step. Anche in questo caso bisogna stare attenti a non aumentare la complessità inserendo troppe sottoclassi.

Prototype

Utilizza il pattern Prototype quando il tuo codice non deve dipendere dalle classi concrete degli oggetti che devi copiare. Questo accade spesso quando il codice lavora con oggetti passati da codice di terze parti tramite un'interfaccia. Le classi concrete di questi oggetti sono sconosciute e non potresti dipendere da esse nemmeno volendo. Il pattern Prototype fornisce al codice client un'interfaccia generale per lavorare con tutti gli oggetti che supportano la clonazione. Questa interfaccia rende il codice client indipendente dalle classi concrete degli oggetti che clona. Così facendo andiamo a ridurre anche il numero di sottoclassi necessarie per la creazione dello stesso oggetto. Crea un’interfaccia prototype e dichiara il metodo clone, oppure inserisce il metodo in tutte le classi della gerarchia. Il problema è che può essere complicato creare oggetti complessi con queste referenze circolari.

Singleton

Singleton è un modello di progettazione creazionale che consente di garantire che una classe abbia una sola istanza, fornendo al contempo un punto di accesso globale a tale istanza. Questo pattern risolve 2 problemi contemporaneamente 1° si assicura che quell’oggetto abbia una singola istanza 2° provvede a fornire un punto di accesso globale a tale istanza proteggendola da eventuali sovrascrizioni. Quindi si rende privato il costruttore predefinito, per impedire ad altri oggetti di utilizzare l'operatore new con la classe Singleton. Si crea un metodo di creazione statico che funga da costruttore. In pratica, questo metodo chiama il costruttore privato per creare un oggetto e salvarlo in un campo statico. Tutte le chiamate successive a questo metodo restituiscono l'oggetto memorizzato nella cache. L’esempio che abbiamo visto è il governo, un solo governo ufficiale, un solo punto di accesso per identificare una serie di persona al comando. A differenza degli altri però questo pattern ha dei contro:

* Viola il principio di singola responsabilità
* Può mascherare un brutto design, in cui tutti “sanno tutti troppo”
* La creazione in ambiente multi thread è più particolare al fine di non far ricreare l’oggetto singleton più volte.
* Potrebbe essere difficile eseguire dei test in quanto molti framework di test si basano sull’ereditarietà quando producono oggetti fittizi.

Passiamo ora ai pattern STRUTTURALI

Adapter

È un pattern strutturale che consente di adattare un oggetto con un’interfaccia incompatibile con cui collaborare. Problema che abbiamo visto: se ad esempio noi creiamo una lista di prodotti destinati alla vendita in xml e poi in un secondo momento la vuoi far collaborare con una lista di terze parti che però usa Json, invece di cambiare la libreria si crea appunto un adapter che è un oggetto speciale che converte l'interfaccia di un oggetto in modo che un altro oggetto possa interpretarla. L’esempio che abbiamo visto è il caricatore del pc che da uno stato all’altro potrebbe non avere la giusta presa e quindi si usa un oggetto adapter per permette l’utilizzo.

Bridge

Bridge è un modello di progettazione strutturale che consente di suddividere una classe di grandi dimensioni o un insieme di classi strettamente correlate in due gerarchie distinte, astrazione e implementazione, che possono essere sviluppate indipendentemente l'una dall'altra. Il pattern Bridge consente di suddividere una classe monolitica (che è un male in termini di gestione) in diverse gerarchie di classi. Successivamente, è possibile modificare le classi in ciascuna gerarchia indipendentemente dalle classi nelle altre. Questo approccio semplifica la manutenzione del codice e riduce al minimo il rischio di interruzione del codice esistente. Si utilizza il Bridge se hai bisogno di poter cambiare implementazione in fase di esecuzione. Bridge consente di sostituire l'oggetto di implementazione all'interno dell'astrazione. È semplice come assegnare un nuovo valore a un campo. Il rischio è quello di rendere il codice molto complicato applicando questo pattern per far coesistere molte classi.

Composite

Composite è un modello di progettazione strutturale che consente di comporre oggetti in strutture ad albero e quindi di lavorare con queste strutture come se fossero oggetti individuali. Ad esempio, quando hai una scatola con all’interno degli oggetti. La struttura gerarchica delle forze dell’ordine è l’esempio di questo pattern

* Component: interfaccia comune (es. visualizza(), aggiungi())
* Leaf (foglia): oggetto "semplice", non contiene altri oggetti
* Composite (contenitore): può contenere altri componenti (foglie o altri composite)
* Client: usa i componenti senza distinguere foglie e contenitori

Un esempio pratico è un menù di un app o simili con i vari sottomenù annidati.

Decorator

Decorator è un modello di progettazione strutturale che consente di associare nuovi comportamenti agli oggetti inserendo tali oggetti all'interno di speciali oggetti wrapper che contengono i comportamenti. Nella maggior parte dei linguaggi, l'ereditarietà non consente a una classe di ereditare i comportamenti di più classi contemporaneamente. Un modo per superare queste limitazioni è utilizzare l'aggregazione o la composizione al posto dell'ereditarietà. Entrambe le alternative funzionano pressoché allo stesso modo: un oggetto ha un riferimento a un altro e gli delega una parte del lavoro, mentre con l'ereditarietà, l'oggetto stesso è in grado di svolgere tale lavoro, ereditando il comportamento dal suo superclasse. Con questo nuovo approccio è possibile sostituire facilmente l'oggetto "helper" collegato con un altro, modificando il comportamento del contenitore a runtime. Un oggetto può utilizzare il comportamento di diverse classi, avendo riferimenti a più oggetti e delegando loro ogni tipo di lavoro. Aggregazione/composizione è il principio chiave alla base di molti design pattern, incluso Decorator. A questo proposito, torniamo alla discussione sui pattern. il wrapper implementa la stessa interfaccia dell'oggetto incapsulato. Ecco perché dal punto di vista del client questi oggetti sono identici. Fate in modo che il campo di riferimento del wrapper accetti qualsiasi oggetto che segua tale interfaccia. Questo vi permetterà di coprire un oggetto con più wrapper, aggiungendovi il comportamento combinato di tutti i wrapper. Estendono il comportamento ma non fanno parte dell’oggetto in sé, come i vestiti per noi. I contro: la difficoltà di rimuovere un wrapper specifico dallo stack dei wrapper, e la

difficoltà di implementare un decoratore in modo che il suo comportamento non dipenda dall'ordine nello stack dei decoratori.

Facade

Facade è un pattern di progettazione strutturale che fornisce un'interfaccia semplificata a una libreria, un framework o qualsiasi altro set complesso di classi. Facade è una classe che fornisce un'interfaccia semplice a un sottosistema complesso contenente numerose parti mobili. Una facciata potrebbe offrire funzionalità limitate rispetto all'utilizzo diretto del sottosistema. Tuttavia, include solo le funzionalità che interessano realmente ai clienti. Quindi è utile quando è necessario integrare la propria app con una libreria sofisticata con decine di funzionalità, ma si necessita solo di una minima parte delle sue funzionalità. L’esempio nel mondo reale è quando un cliente ordina attraverso una telefonata un prodotto, fa una chiamata semplice per utilizzare una funzione più complessa. Il contro di questo pattern è che potrebbe rendere l’oggetto coinvolto in quasi tutto il comportamento del programma.

Questo pattern è molto simile al Mediator.

Flyweight

Flyweight è un modello di progettazione strutturale che consente di adattare più oggetti alla quantità di RAM disponibile condividendo parti comuni di stato tra più oggetti anziché mantenere tutti i dati in ciascun oggetto. Utilizzare il pattern Flyweight solo quando il programma deve supportare un numero enorme di oggetti che difficilmente trovano posto nella RAM disponibile. Con questo pattern però si sacrifica un po’ di cpu e si rende il codice particolarmente complicato. Diventa difficile la comprensione con i nuovi membri poi. Un esempio di questo pattern può essere la creazione di un albero ripetuto all’interno di un ambiente per un videogioco.

* TreeType: è il Flyweight (condivide le info comuni).
* Tree: ha una posizione unica, ma usa un TreeType condiviso.
* TreeFactory: è la Flyweight Factory, crea e gestisce i TreeType.

Proxy

Un proxy è un pattern di progettazione strutturale che consente di fornire un sostituto o un segnaposto per un altro oggetto, magari anche molto pesante con un uso non frequente. Un proxy controlla l'accesso all'oggetto originale, consentendo di eseguire un'operazione prima o dopo che la richiesta sia stata inoltrata all'oggetto originale. Il pattern Proxy suggerisce di creare una nuova classe proxy con la stessa interfaccia dell'oggetto servizio originale in maniera tale da passere lui e delegarli tutto. Per fare un esempio concreto: Una carta di credito è un sostituto di un conto bancario, che a sua volta rappresenta un cumulo di contanti. Entrambe implementano la stessa interfaccia: possono essere utilizzate per effettuare un pagamento. I contro di questo pattern sono principalmente: 1° il codice potrebbe iniziare ad essere complicato con la presenza di troppe classi 2° la risposta del/al servizio potrebbe essere ritardata.

Ora passiamo ai pattern COMPORTAMENTALI

Chain of Responsibility

Chain of Responsibility è un modello di progettazione comportamentale che consente di inoltrare le richieste lungo una catena di gestori. Alla ricezione di una richiesta, ogni gestore decide se elaborarla o passarla al gestore successivo nella catena (handler), così facendo evita accoppiamenti rigidi tra chi invia la richiesta e chi la gestisce. Il problema di questo pattern è che alcune richieste potrebbero non essere gestite. Un esempio è una richiesta di assistenza, in quanto, se sarà semplice verrà passata ad un handler, se sarà media ad un altro, se sarà difficile ad un altro ancora. Ogni Handler di questa catena valuta se è in grado di risolvere il problema.

* Handler (astratto): definisce l'interfaccia per gestire la richiesta e collegarsi al successivo.
* ConcreteHandler: implementa la logica per gestire (o passare) la richiesta.
* Client: invia la richiesta a un handler della catena.

Command

Command è un pattern di progettazione comportamentale che trasforma una richiesta in un oggetto autonomo contenente tutte le informazioni sulla richiesta. Questa trasformazione consente di passare le richieste come argomenti di un metodo, ritardare o mettere in coda l'esecuzione di una richiesta e supportare operazioni annullabili. L’esempio pratico che abbiamo visto è il caso in cui una persona vada a mangiare, il cameriere prende l’ordine e lo appena in cucina, lo chef aspetterà di leggere i dettagli per iniziare a cucinare invece di venire da te a chiedertele.

1. Command (interfaccia astratta): definisce il metodo execute().
2. ConcreteCommand: implementa execute() e chiama metodi su un ricevitore.
3. Receiver: chi fa davvero il lavoro (es. una lampada, un documento...).
4. Invoker: invoca il comando (es. un bottone o un menu).
5. Client: crea i comandi e li passa all’invoker.

Utilizza il pattern Command quando desideri parametrizzare oggetti con operazioni, mettere in coda operazioni, pianificarne l'esecuzione o eseguirle in remoto. Il codice potrebbe essere molto complicato nella gestione della richiesta tra senders e receivers.

Iterator

Iterator è un modello di progettazione comportamentale che consente di attraversare gli elementi di una raccolta senza esporre la rappresentazione sottostante (elenco, pila, albero, ecc.), cioè io lo posso esplorare sequenzialmente senza però saper com’è fatto internamente. Un esempio reale è la scelta del metodo di esplorazione di roma per un turista che può scegliere tra mini-mappa, tour guidato, tour nelle cuffiette ecc.

Utilizza il pattern Iterator quando la tua collezione ha una struttura dati complessa, ma vuoi nasconderne la complessità ai client (per comodità o per motivi di sicurezza), oppure quando vuoi che il tuo codice sia in grado di attraversare diverse strutture dati o quando i tipi di queste strutture sono sconosciuti a priori.

* Iterator: interfaccia che definisce i metodi per scorrere (es. next(), has\_next()).
* Concrete Iterator: implementa la logica di scorrimento.
* Aggregate (collezione): interfaccia o classe che fornisce il metodo create\_iterator().
* Concrete Aggregate: implementa la collezione vera e propria.

Applicare il pattern può essere eccessivo se la tua app funziona solo con raccolte semplici. Mentre potrebbe essere meno efficiente rispetto all'analisi diretta degli elementi di alcune raccolte specializzate.

Mediator

Usato per **semplificare la comunicazione tra oggetti**, evitando che si riferiscano direttamente tra loro con dipendenze complicate e impedendone la comunicazione diretta utilizzando, appunto, un’interfaccia mediator che definisce il metodo di comunicazione. Questo consente di rendere il sistema più modulare e facile da modificare/testare.

1. Mediator (interfaccia): definisce un metodo per la comunicazione.
2. ConcreteMediator: implementa la logica di comunicazione tra gli oggetti.
3. Colleague: oggetti che usano il mediatore per comunicare tra loro.

Un esempio realistico è il controllo sul traffico aereo, infatti gli aerei non comunicano tra loro per sapere il traffico ma si interfacciano ad una torre di controllo per capire il traffico aereo.

Il problema è che troppe volte un mediator può arrivare ad essere god object (che sa troppo di troppe cose, quindi coinvolto nella maggior parte del codice)

Memento

Memento è un design pattern comportamentale che serve a salvare e ripristinare lo stato di un oggetto senza violarne l'incapsulamento, Permette a un oggetto di salvare uno snapshot del proprio stato interno in un "memento", così da poter tornare a quello stato in seguito (es. operazione di undo, salvataggi, ecc.). L’obiettivo del memento è permettere operazioni di annullamento (undo) o ripristino dello stato, mantenendo la sicurezza e l’incapsulamento.

* Originator: l’oggetto di cui si vuole salvare lo stato.
* Memento: oggetto che contiene lo stato salvato.
* Caretaker: gestisce il memento (li salva, ripristina, ecc.), senza sapere cosa contengono.

Observer

È un pattern che definisce una dipendenza uno-a-molti tra oggetti: quando un oggetto cambia stato, tutti gli oggetti dipendenti vengono notificati automaticamente. Observer è un modello di progettazione comportamentale che consente di definire un meccanismo di sottoscrizione per notificare a più oggetti tutti gli eventi che accadono all'oggetto che stanno osservando. Ha l’obiettivo di evitare l'accoppiamento stretto tra oggetti, permettendo una comunicazione automatica tra chi emette eventi e chi li riceve. Può diventare complesso se ci sono molti observer e bisogna fare attenzione a memory leak se gli osberver non vengono rimossi correttamente. Inoltre, gli iscritti sono notificati in maniera random.

* Subject (osservato): mantiene una lista di observer e li notifica quando cambia.
* Observer (osservatori): oggetti che vogliono essere notificati dei cambiamenti del Subject.
* ConcreteSubject / ConcreteObserver: implementazioni concrete.

Un esempio reale è ad esempio il canale youtube che manda ai suoi iscritti la notifica quando vengono caricati nuovi contenuti.

State

Lo state è un pattern di progettazione comportamentale che consente a un oggetto di modificare il proprio comportamento quando cambia il suo stato interno. Appare come se l'oggetto avesse cambiato classe. L’obiettivo del pattern è quello di sostituire strutture if/switch con classi per ogni stato, migliorando leggibilità, estendibilità e mantenibilità.

* Context: l’oggetto che ha uno stato e delega i comportamenti allo stato attuale.
* State (interfaccia): definisce le operazioni comuni a tutti gli stati.
* ConcreteState: implementazioni specifiche di ogni stato.

Un esempio reale di questo pattern è ad esempio l’utilizzo del tasto di blocco/sblocco nelle varie fasi del cellulare. L'applicazione del modello può essere eccessiva se una macchina a stati ha solo pochi stati o cambia raramente

Strategy

Lo strategy è un modello di progettazione comportamentale che consente di definire una famiglia di algoritmi, di inserire ciascuno di essi in una classe separata e di rendere i loro oggetti intercambiabili. Ha l’obiettivo di evitare codice con if/switch per scegliere algoritmi, favorendo l’estendibilità e il riutilizzo.

* **Strategy (interfaccia)**: definisce l’algoritmo comune.
* **ConcreteStrategy**: implementazioni specifiche dell’algoritmo.
* **Context**: utilizza una Strategy e delega a essa il comportamento.

Un esempio banale è il mezzo di trasporto con cui scelgo di viaggiare che dipende dal budget e dal tempo. I contro di questo pattern sono sostanzialmente il fatto che il codice diventa più complicato soprattutto se abbiamo coppie di algoritmi che raramente cambiano e il fatto che alcuni nuovi linguaggi di programmazione hanno dei supporti per gestire questi casi. Attenzione a non confondere lo state con lo strategy in quanto il primo rappresenta stati e cambia il comportamento in base allo stato, il secondo invece rappresenta algoritmi e cambia comportamento in base alle nostre scelte usato per scelte logiche alternative.

Template Method

Il metodo template è un modello di progettazione comportamentale che definisce lo scheletro di un algoritmo nella superclasse, ma consente alle sottoclassi di sovrascrivere passaggi specifici dell'algoritmo senza modificarne la struttura. Ha l’obiettivo di riutilizzare codice comune ma consentire personalizzazioni, evitando di duplicare la struttura generale dell’algoritmo.

* Classe Astratta (Template): definisce il metodo "template" (l’algoritmo) e alcuni metodi astratti o "hook" da sovrascrivere.
* Sottoclassi Concrete: implementano i passaggi specifici.

I contro di questo pattern sono legati alla rigidità dello scheletro dell’algoritmo, inoltre viola il principio di sostituibilità di Liskov il quale dice che le classi possono essere sostituite tra loro senza alterare la correttezza del programma. Un esempio realistico è la costruzione di alloggi di massa. Il progetto architettonico per la costruzione di una casa standard può prevedere diverse estensioni che consentirebbero al potenziale proprietario di modificare alcuni dettagli della casa risultante. Ogni fase costruttiva, come la posa delle fondamenta, la struttura, la costruzione dei muri, l'installazione di impianti idraulici ed elettrici, ecc., può essere leggermente modificata per rendere la casa risultante leggermente diversa dalle altre.

Visitor

Il Visitor è un design pattern comportamentale un po’ più avanzato, ma molto potente, usato quando vuoi eseguire operazioni su una struttura di oggetti eterogenei senza modificare le loro classi. Permette di aggiungere nuove operazioni a oggetti di classi diverse senza modificarli, usando un oggetto "visitatore" che incapsula le operazioni. Si usa appunto quando hai una struttura complesse, non vuoi modificale le classi ma vuoi aggiungere operazioni diverse. Il contro di questo pattern è che ogni volta che aggiungi una nuova classe concreta, devi modificare tutti i visitor ed è più complesso di strategy o template. Un esempio reale può essere un agente assicurativo. Può visitare ogni edificio di un quartiere, cercando di vendere assicurazioni a chiunque incontri. A seconda del tipo di organizzazione che occupa l'edificio, può offrire polizze assicurative specializzate.

* Element: interfaccia o classe con un metodo accept(visitor)
* ConcreteElement: implementa accept () e chiama visitor.visit(this)
* Visitor: interfaccia con un metodo visit () per ogni tipo di elemento
* ConcreteVisitor: implementazione concreta che definisce le operazioni da eseguire sugli oggetti

Postman/Dbeaver/Docker

Postman è un’applicazione che permette di interagire con servizi web in maniera semplificata. DBeaver è uno strumento open source per la gestione di database, infatti consente connettersi a vari db, creare diagrammi ecc. Docker invece è molto usato per creare, distribuire e gestire applicazioni in container. Nel nostro primo approccio abbiamo visto come docker monta l’immagine del db e tramite Dbeaver ci consente le modifiche sul nostro db.

Sia il db e sia le 2 app da fare dovranno essere comunicanti sulla stessa porta in locale, in quanto stiamo rendendo disponibile la nostra macchina per le riscorse necessarie per il db. Nell’init noi troviamo informazioni molto importanti per configurare le app tra loro e rendere possibile la comunicazione tra esse. Vediamolo:

* “image: postgres:15” ci dice che ha caricato il tutto usando un modello pre-esistente con postgres15
* Il nome del progetto, il nome del db, username e pass.
* La porta sulla quale comunicare
* I volumi

I volumi sono molto importanti, infatti quando il db si riavvia ecc perderebbe i dati che lo contengono ma noi ci aspettiamo di trovarli in quei volumi, così da non perdere mai nessun dato che popola il db.

MySQL

È il Database Management System più diffuso. Le sue funzioni sono Creazione-Modifica-Interrogazione

Glossario:

* Dominio: insieme dei valori ammissibili che una colonna (un attributo) può assumere;
* Tabella: insieme ordinato di colonne ed eventuali vincoli relativi ai valori degli attributi stessi;
* Indice: struttura dati ausiliaria, associata ad una tabella, che permette di rendere più efficiente l’esecuzione di interrogazioni sulla tabella mediante l’introduzione di un ordinamento sui valori di uno o più attributi;
* Asserzione: rappresenta dei vincoli direttamente connessi alla base di dati;
* Vista: è una tabella “virtuale”, il cui contenuto è legato al contenuto di altre tabelle della base di dati;
* Privilegio: definizione delle modalità di accesso ai dati da parte di un certo utente.

PRONTUARIO SINTASSI COMANDI SQL

|  |  |
| --- | --- |
| COMANDO | SEMANTICA |
| CREATE DATABASE nome\_database; | Crea un nuovo database |
| DROP DATABASE nome\_database; | Elimina un database esistente |
| CREATE TABLE nome\_tabella | Creare una nuova tabella |
| ALTER TABLE nome\_tabella | Modificare la struttura di una tabella (aggiunge, rimuove, rinomina colonne, cambia tipo di una colonna) |
| DROP TABLE nome\_tabella | Elimina una tabella esistente |
| INSERT INTO nome\_tabella | Inserisce valori in certi campi di una nuova riga |
| UPDATE nome\_tabella | Cambia valori in certi campi di una riga esistente |
| DELETE FROM nome\_tabella | Elimina una riga in base a certi criteri |
| SELECT | Estrapola campi, righe, raggruppa, ordina, calcola |
| GRANT | Garantisce privilegi sulle tabelle agli utenti |
| REVOKE | Revoca privilegi sulle tabelle agli utenti |

NB: gli elementi tra [] sono Opzionali;

CREATE TABLE *nome\_tabella*

(

nome\_campo1 *tipo\_di\_dato*(*size*) [vincoli di campo], nome\_campo1 *2 tipo\_di\_dato*(*size*) [vincoli di campo], nome\_campo1 *3 tipo\_di\_dato*(*size*) [vincoli di campo],

....

[PRIMARY KEY (nome\_campo, [nome\_campo, ..., nome\_campo])]

[FOREIGN KEY (nome\_campo) REFERENCES nome\_tabella[(nome\_campo)]]

[UNIQUE (nome\_campo)]

[CHECK (expr\*)]

);

I vincoli di campo possono essere:

|  |  |
| --- | --- |
| NOT NULL | La colonna non può memorizzare valori nulli |
| AUTO\_INCREMENT | Il valore di tipo numerico si incrementa ad ogni inserimento |
| UNSIGNED | Senza segno e cioè positivo |
| DEFAULT | Valore di Default per la colonna |
| UNIQUE | Ogni riga avrà un valore unico sulla colonna |
| PRIMARY KEY | chiave primaria, UNIQUE e NOT NULL insieme, |
| FOREIGN KEY | Chiave esterna, assicura integrità referenziale a un'altra tabella |
| CHECK | Vincolo sulle colonne |

INSERT INTO nome\_tabella VALUES (valore1, valore2, valore3,...); oppure:

INSERT INTO nome\_tabella (*colonna1*,*colonna2*,*colonna3*,...) VALUES (valore1, valore2, valore3,...);

UPDATE nome\_tabella SET colonna1=valore1,colonna2=valore2,...WHERE condizione\*;

DELETE FROM nome\_tabella WHERE *condizione\**;

SELECT [ALL | DISTINCT] lista\_campi\_separati\_da\_virgola

FROM lista\_nomi\_tabelle\_separati\_da\_virgola [WHERE condizione\*]

[GROUP BY lista\_campi\_separati\_da\_virgola]

[HAVING condizione\*]

[ORDER BY lista\_campi\_separati\_da\_virgola [ASC | DESC]]

[LIMIT numero\_di\_righe [OFFSET numero\_iniziale]]

Nella clausola SELECT è possibile usare al posto di lista\_campi\_separati\_da\_virgola

* \* che vuol dire tutti i campi
* funzioni di aggregazione sui domini separati da virgola (da soli o con campi delle tabelle, vedi sotto)
* alias di campi tramite: AS alias\_per\_il\_campo a destra del campo
* espressione con semplici calcoli aritmetici (+, -, \*, /) tra (valori dei) campi

\*la condizione che si inserisce nel WHERE, nell'HAVING o anche come espressione nel CHECK (vedi CREATE TABLE) deve essere sempre di uno dei seguenti tipi:

* operando operatoreBinario operando
* operatoreUnario operando
* operando operatoreUnario dove:
* operando può essere: un nome di campo, un valore, una condizione (racchiusa tra parentesi)
* operatore può essere:
* di confronto: <, <=, >, >=, =, != (diverso)
* logico: AND, OR, NOT (operatore unario)
* operatore LIKE
* BETWEEN, [NOT] IN, IS [NOT] NULL con le seguenti sintassi:

◦ nome\_campo BETWEEN valore1 AND valore2,

◦ nome\_campo [NOT] IN ('valore1', 'valore2', ...)

◦ nome\_campo IS [NOT] NULL

Funzioni di aggregazione sui domini:

si possono usare nella clausola SELECT singolarmente o separati da virgola al posto di *lista\_campi\_separati\_da\_virgola* se usate assieme a uno o più campi delle tabelle è obbligatorio l'uso del GROUP BY

|  |  |
| --- | --- |
| COUNT (\*)  COUNT (nome\_campo)  COUNT (DISTINCT nome\_campo) | Riporta il numero di righe della tabella  Riporta il numero di valori non nulli del campo  Riporta il numero di valori univoci non nulli del campo |
| SUM (nome\_campo) | Somma i valori della colonna |
| MIN (nome\_campo) | Calcola il minimo valore della colonna |
| MAX (nome\_campo) | Calcola il massimo valore della colonna |
| AVG (nome\_campo) | Calcola la media aritmetica tra tutti i valori della colonna, applicato solo a campi numerici, i valori nulli sono ignorati |

VAADIN

Vaadin è un linguaggio di scripting lato server, scritto in Java, che genererà la maggior parte del codice lato client necessario per un'applicazione web. Utilizza Google Web Toolkit per generare gli oggetti lato client e può essere esteso creando estendendo Google Web Toolkit.

(Praticamente vaadin fornisce tool già fatti per personalizzare il front end)

Vaadin Flow e Vaadin Fusion

Server Side not similar to angular

|  |  |  |
| --- | --- | --- |
| Linguaggio principale | Java (e TypeScript per Hilla) | TypeScript (e HTML/CSS) |
| Livello di astrazione | Astrazione completa da HTML, CSS e JavaScript | Richiede la scrittura di codice HTML, CSS e JavaScript |
| Architettura | Backend Java con frontend (opzionale) in TypeScript per Hilla | Architettura frontend basata su componenti |
| Debugging e Testing | Più semplice in Vaadin (su Java), più complesso in Angular (su browser) | — |
| Comunicazione backend | Accesso diretto ai servizi lato server (Vaadin Flow) | Utilizzo di HttpClient (Angular) |
| Componenti | Ampia libreria di componenti predefiniti | Collezioni di componenti Material Design (Angular) |
| Progettazione | Design system integrato con Vaadin, personalizzabile con CSS | Flessibilità di progettazione con HTML, CSS e componenti customizzabili |
| Curva di apprendimento | Potenzialmente più rapida per sviluppatori Java | Curva di apprendimento più ripida per sviluppatori non familiari con JavaScript e TypeScript |
| Backend | Backend Java con Vaadin Flow o Hilla | Backend separato (es: Spring Boot, Node.js) |