PATTERN

Il *PATTERN* è una o un INSIEME DI LINEE GUIDA che sono utili per un'implementazione applicativa o per risolvere un problema ricorrente.

[Gangs of fours design pattern]
Sono un insieme di 24 pattern creati da 4 designers e si suddividono in 3 categorie:

1. **CREATIONAL**:
   rappresentano le linee guida relative alla creazione di oggetti;
2. **STRUCTURAL**:
   rappresentano le linee guida relative alla struttura da dare ad una applicazione e forniscono modelli per organizzare Oggetti in modo flessibile;
3. **BEHAVIORAL**:
   linee guida relative a come gli oggetti devono interagire tra di loro (come si devono comportare gli oggetti).

Elenchiamo una parte di pattern appartenenti alla Gof (i più usati e famosi):

***STRUCTURAL**

- **Facade**

CREATIONAL

- **Singleton**
- **Factory**
- **Builder**

BEHAVIOUAL

- **Strategy**

> ***FACADE*** **= Appartiene alla categoria Structural.**

Secondo questo Pattern, è possibile strutturare un'applicazione invocata, in maniera tale che si esponga alle applicazioni invocate, tramite un unico e centralizzato punto di ingresso agnostico rispetto al sistema invocante.

[In parole più semplici, secondo questo pattern, è possibile fornire un unico punto di accesso centralizzato all'applicazione invocante, per accedere alle funzionalità dell'applicazione invocata.]

MODALITA' DI **IMPLEMENTAZIONE PATTERN FACADE** :
1. **SI CREA UN' INTERFACCIA CHE CONTIENE TUTTE LE FUNZIONALITA' DEL SISTEMA;**
2. **SI CREA UNA CLASSE CHE INVOCHI TUTTI I METODI DELLA STRUTTURA APPLICATIVA SOTTOSTANTE.**

> ***FACTORY*** **= Appartiene alla categoria Creational, e la sua traduzione è "Delega"**

Si considera utile applicare questo Pattern, in contesti dove non si vuole dare all'**Applicazione Invocante** alcuna responsabilità di creazione di **Oggetti**, ne tantomeno di **BUSINESS LOGIC**.
Ma si vuole dare la possibilità all'**Applicazione Invocante**, in base a diversi input (scelte), si delega **l'Applicazione Invocata** per la Logica Di Creazione degli **Oggetti** e qualsiasi **Logica Di Business**.

[In parole più semplici, Invece di creare Oggetti direttamente nel codice, si crea una "fabbrica" (la classe Factory) che si occupa di creare Oggetti specifici in base alle richieste.]

EREDITARIETA' FRA CLASSI : (non è collegata al pattern, è una cosa generale il java)

OGNI VOLTA CHE SI ISTANZIA UNA CLASSE FIGLIA,
VIENE AUTOMATICAMENTE INVOCATO PRIMA IL COSTRUTTORE DELLA CLASSE PADRE,
E POI IL COSTRUTTORE DELLA CLASSE FIGLIA.

[ASSIOMA] : NEL CASO IN CUI LA CLASSE PADRE DICHIARI UN COSTRUTTORE CUSTOM, LA CLASSE FIGLIA E' OBBLIGATA A DICHIARARE UN COSTRUTTORE CUSTOM CHE ABBIA ALMENO LE PROPRIETA' GIA' UTILIZZATE NEL COSTRUTTORE CUSTOM DELLA CLASSE PADRE ,
(POTENDO EVENTUALMENTE AGGIUNGERNE ALTRE)
E INVOCARE ESPLICITAMENTE TRAMITE LA PAROLA CHIAVE **["super"]** IL COSTRUTTORE DELLA CLASSE PADRE.

> ***SINGLETON*** **= Appartiene alla categoria Creational, e la sua traduzione è "Singolo"**

[E' utile applicare questo Pattern, quando si vuole che un oggetto, una volta creato sia irripetibile, ovvero, non si possono creare successivi oggetti aventi esattamente lo stesso contenuto.]
Creare più Oggetti a partire dalla stessa classe, allocando diverse aree di memoria per ogni Oggetto creato, vuol dire creare Oggetti in modalità ["prototype"].

Si usa questo pattern per:

1. Risparmiare memoria = se una classe ha molte risorse, la creazione di un unica istanza, evita sprechi;
2. Migliora la coerenza;
3. Semplifica l'accesso globale.

MODALITA' DI **IMPLEMENTAZIONE PATTERN SINGLETON** (JAVA SE):

1. **Dichiarazione di una variabile di classe (static) dello stesso tipo della classe rispetto alla quale si intende implementare il pattern Singleton;**

2. **Implementazione del costruttore private;**

3. **Implementazione di un metodo di classe (static) che ritorni una reference dello stesso tipo della classe e costruisca una logica per istanziare una sola volta un oggetto della classe tramite Costruttore private.**

> ***BUILDER*** **= Appartiene alla categoria Creational, e la sua traduzione è "costruttore".**

[Si considera utile applicare questo Pattern, nei casi in cui sia necessario/utile SEPARARE la creazione di un oggetto dalla sua rappresentazione.]

Modalità Implementazione Pattern Builder
MODALITA' DI **IMPLEMENTAZIONE PATTERN BUILDER** :

1. **Creazione di una classe rispetto alla quale si vogliono costruire gli Oggetti (OUTER);**

2. **Dichiarazione all'interno della classe OUTER di tutte le variabili di istanza di cui si potrebbe comporre l'oggetto, sia per la struttura fissa, sia per modifiche future della struttura (OUTER);**

3. **Implementazione di tanti metodi di Get() (e solo di get), per quante sono le variabili di istanza (OUTER);**

4. **Creazione di una classe INNER static;**

5. **Dichiarazione all'interno della classe INNER delle stesse variabili di istanza dichiarate nella OUTER;**

6. **Implementazione di un metodo di Set() che modifichi le variabili di istanza rappresentanti la struttura non fissa, e che ritorni una reference di tipo INNER (nella INNER class);**

7. **Implementazione all'interno della INNER Class di un Costruttore che riceva in input una reference di tipo INNER;**

8. **Implementazione nella INNER class di un metodo (best practice build) che ritorni un oggetto OUTER passando "this" in input al Costruttore dell'OUTER;**

9. **Implementazione di un Costruttore Custom all'interno della classe INNER, che setti le variabili di istanza rappresentanti la struttura fissa.**

> ***STRATEGY*** **= Appartiene alla categoria Behavioural**

[Il Pattern Strategy, si applica in base all'invio di diversi tipi di informazioni da parte dell'APPLICAZIONE INVOCANTE.
[Occorre che l'APPLICAZIONE INVOCATA attui una strategia differente di implementazione in base alle informazioni passate dall'APPLICAZIONE INVOCANTE.

IL PATTERN STRATEGY A DIFFERENZA DEL PATTERN FACTORY DELEGA ALL'APPLICAZIONE INVOCANTE LA CREAZIONE DEGLI OGGETTI

(GLI OGGETTI RAPPRESENTANO LE INFORMAZIONI IN BASE ALLE QUALI L'APPLICAZIONE INVOCATA ATTUA UNA DIVERSA STRATEGIA DI RISPOSTA ALL'APPLICAZIONE INVOCANTE).

> ***LOOSE COUPLING*** **= Il Pattern Loose Coupling NON appartiene alla Gof**

[Questo Pattern dice che è preferibile disaccoppiare un'Applicazione Invocante da un'Applicazione Invocata.
[Ovvero fare in modo che l'Applicazione Invocante possa colloquiare con l'Applicazione Invocata SENZA conoscere i DETTAGLI IMPLEMENTATIVI.

DAO **= Data Access Object = Anche il Pattern "DAO" non fa parte dei Gof**
(CONCETTUALMENTE PUO' ESSERE CONSIDERATO COME "STRUCTURAL" ANCHE SE NON FA PARTE DEI "Gof")

[E' un Pattern che serve per Applicazioni che interagiscono con Database (qualsiasi sia il tipo di database).
[E' preferibile separare la Logica Di Persistenza (detta anche "logica di C.R.U.D.")
[in uno STRATO APPLICATIVO dedicato (insieme di componenti applicative), che è composto da un'INTERFACCIA e una CLASSE che espongono le Operazioni di CRUD per ogni ENTITA' CONCETTUALE del database, in maniera tale che lo STRATO APPLICATIVO sia agnostico rispetto alle applicazioni invocanti.

Il termine **agnostico** qui significa che lo **strato applicativo** è progettato per essere indipendente dalle applicazioni che lo utilizzano,
offrendo una maggiore flessibilità e una facilità di integrazione con diverse applicazioni senza richiedere conoscenze specifiche o dipendenze da queste ultime.

VO = Value Object
all'interno di un'applicazione che esegue operazioni di CRUD su un database occorre creare una classe per ogni entità del database, che contenga tante variabili di istanza per ogni campo dell'entità, tanti metodi di Set e Get per ogni variabile d'istanza e almeno il costruttore vuoto/implicito (ci ricorda il java bean).

[E' CONSIGLIABILE L'UTILIZZO, ALL'INTERNO DI UNA APPLICAZIONE CHE DEBBA IMPLEMENTARE LE OPERAZIONI DI CRUD DI LETTURA.]
[SI CREA PER OGNI TABELLA DEL DATABASE, UNA CORRISPONDENTE CLASSE CHE CONTIENE TANTE VARIABILI D'ISTANZA PER QUANTE SONO LE COLONNE DELLA TABELLA CORRISPONDENTE, E TANTI METODI DI SET E GET PER QUANTE SONO LE VARIABILI DI ISTANZA, E ALMENO IL COSTRUTTORE DI DEFAULT.]

Per operazioni di lettura dal database, si considera importante poter recuperare l'informazione integrale, ed anche eventuali campi AUTO-INCREMENT perchè possono essere funzionali ad altre operazioni.

DTO **= Data Transfer Object = CONCETTUALMENTE PUO' ESSERE CONSIDERATO COME "CREATIONAL" ANCHE SE NON FA PARTE DEI "Gof"**
ed è un Pattern antecedente alla Gof

[E' CONSIGLIABILE L'UTILIZZO, ALL'INTERNO DI UNA APPLICAZIONE CHE DEBBA IMPLEMENTARE LE OPERAZIONI DI CRUD.]
[SI CREA PER OGNI TABELLA DEL DATABASE, UNA CORRISPONDENTE CLASSE CHE CONTIENE TANTE VARIABILI DI ISTANZA QUANTE SONO LE COLONNE DELLA CORRISPONDENTE TABELLA, (AD ECCEZION FATTA DI EVENTUALI COLONNE AUTO-INCREMENT) E TANTI METODI DI SET E GET PER QUANTE SONO LE VARIABILI DI ISTANZA, ED ALMENO IL COSTRUTTORE DI DEFAULT (CI RICORDA IL JAVA BEAN).]

In pratica una classe DTO è UN JAVA BEAN SPECULARE ad una tabella (se parliamo di database relazionali) ad eccezion fatta di eventuali campi AUTO-INCREMENT.

Il pattern DTO ha un senso di applicabilità nelle casistiche in cui si richiede il trasferimento dall'applicazione al database (quindi le operazioni C.U.D.).
Serve alla creazione di oggetti per trasferire dei dati.

(AGGIUNGERE APPUNTI DELL' 01/07)
Quando si applica il pattern DTO con classi java, viene ritenuto importante aggiungere una **best practice**, ovvero ogni classe DTO implementi una interfaccia java dal nome Serializable;

SERIALIZABLE **è un'interfaccia API Java SE, facente parte di una categoria di interfacce API, chiamata Marker Interfaces;**

Una *MARKER INTERFACE*, è un'interfaccia che se implementata da una classe, conferisce a tutti gli oggetti creati da quella classe, uno "speciale comportamento" a run time
(comportamento che gli oggetti non avrebbero nativamente);

Se **Serializable** viene implementato da una classe, conferisce a tutti gli oggetti che verranno cerati da quella classe, il comportamento di diventare **Serializable a run time**;

OGGETTO DTO NON SERIALIZABLE:
In caso di mal funzionamento della JVM, un oggetto java serializable, viene deallocato dall'HEAP per cui verifica perdita di qualsiasi informazione in esso contenuta (quindi la GC spazza e pulisce tutto), quindi in pratica l'oggetto viene perso.

OGGETTO DTO SERIALIZABLE:
In caso di mal funzionamento della JVM, un oggetto java serializable, viene deallocato dall'HEAP, ma il contenuto viene salvato sul disco fisico della macchina su cui è installata la

JVM, e viene ripristinato all'interno dell'oggetto stesso che viene riallocato nell'HEAP al ripristino della JVM

> **(probabilmente chiedono sto DTO)**

> **MVC** = Model View Object
> E' un Pattern applicabile solo per applicazioni web o mobile.

(CONCETTUALMENTE PUO' ESSERE CONSIDERATO COME "STRUCTURAL" ANCHE SE NON FA PARTE DEI "Gof")

[Questo pattern dice che è consigliabile scomporre una qualsiasi Applicazione Enterprise in tre strati applicativi differenti, ognuno dei quali ha delle specifiche responsabilità:]

MODEL (Anche chiamato **Back End Tier**)
Le responsabilità del model sono la gestione dei dati / operazioni di CRUD relativi al Database;

DTO, VO, DAO (dovrebbero essere i 3 strati applicativi differenti)

VIEW (Anche chiamato **Front End Tier**)
Per la presentazione dei dati (li fa vedere) ;

html(oggi abbiamo fatto una piccola pagina html)

CONTROLLER (Anche chiamato **Middle Layer** o **Middle Tier**, dato che permette alla **Model** e la **View** di colloquiare)
Il CONTROLLER per noi è la Servlet, quindi si occupa della gestione delle richieste e delle risposte HTTP e della Business Logic.
(oggi abbiamo fatto un operazione di business logic, convertendo una stringa in intero) (quindi eventuali controlli, dati, calcoli, algoritmi, ecc...)

IoC
[Questo pattern dice che è possibile chiedere la creazione delle istanze ad una componente infrastrutturale (IoC Container).]

Dependency Injection è una implementazione del pattern IoC, ovvero è uno strumento di Sintassi.

Ioc Container che è un contenitore (vedi appunti Giuseppe)

HATEOAS (HYPERMEDIA AS THE ENGINE OF APPLICATION STATE)
Secondo questo pattern, è preferibile restituire ai web service consumer(quelli che fanno le richieste), risposte che mostrino l'intero stato dell'applicazione web service provider, ovvero restituire risposte che contengano sia il contenuto informativo sulla base della richiesta specifica effettuata, sia un link che rimandi a tutte le altre possibili operazioni consumabili. in pratica il pattern hateoas consiglia di rendere navigabili le risposte fornite ai web service consumer

lo stato di un'applicazione web service provider = è un insieme delle operazioni rest che un web service provider espone.