

ESAME

PROGETTO + ORALE

3 CFU TEORIA

3 CFU MONTELLA

LIBRI

MANUALIS JAVA 9

DESIGN PATTERNS E. GAMA

JAVA

Hello World.java

```
public class HelloWorld {  
    public static void main() {  
        System.out.println("Hello world")  
    }  
}
```

Annotations:

- public class HelloWorld {
 ^ nome file
- public static void main() {
 ^ ci permette di utilizzare la classe
 senza istanziare un oggetto
- System.out.println("Hello world")
 ^ classe di sistema

CONCETTI FONDAMENTALI

- CLASSE
- OGGETTO
- MEMBRO - ATTRIBUTO
 - ^ Metodo
- COSTRUTTORE
- PACKAGE

CLASSE INSIEME DI OGGETTI CHE
CONDIVIDONO LE STESSSE CARATTERISTICHE

UNA CLASSE HA UN **NAME**

PUO' CONTENERE DUE TIPI DI MEMBRI **CAMPIONI** E **METODI**

OGLGETO → Istanza di una classe

PER CREARE UN NUOVO OGGETTO SI USA **new oggetto(...)**

Classe oggetto = new Classe(var);

VARIABILI Istanza → appartengono ad un oggetto

VARIABILI LOCALI → appartengono ad un metodo

VARIABILE STATICA → variabile di classe

→ come se fosse globale

VARIABILE FINAL → variabile costante

TIPI DI METODO

• ACCESSORE → non modificano la variabile esplorata

• MODIFICATORE → possono modificare

• STATICO → possono interagire solo con var. statiche

ASTRAZIONE

concentrarsi solo sui dettagli essenziali nella descrizione di una classe

INCAPSULAMENTO

Accesso controllato ai dati della classe rendendo robusto, indipendente e utilizzabile.

Modificatori di accesso

MODIFICATORE	STESSA CLASSE	STESO PACKAGE	SOTTOCLASSE	OVUNQUE
public	SI	SI	SI	SI
protected ,	SI	SI	SI	NO
nessun modificatore	SI	SI	NO	NO
private	SI	NO	NO	NO

Input → JOptionPane.showInputDialog("xx");

InputStreamReader - = InputStreamReader(system

EREDITARIETÀ

sottoclasse Extends superclasse

Una sottoclasse eredita tutte le variabili e i metodi non private della superclasse.

METODI

Si possono definire:

Sovrascrivendo i metodi della superclasse

Ereditando quelli della superclasse

Dichiarandoli come nuovi metodi

creatore → deve necessariamente riferirsi a quello super

super → per riferirsi alla superclasse

CLASSE OBJECT

È la superclasse di ogn. classe

Contiene metodi base come .toString()

METODO ABSTRACT Es. abstract void doSmt();

Non definisco l'implementazione nella superclasse e lo faccio nella sottoclasse.

CLASSE ASTRATTA → non istanziabile in oggetto

Contiene: metodi astratti, variabili mon static e final, costruttori

INTERFACCIA Es. interface Interfaccia {

Non sono istanziabili → si usa implements nelle sottoclassi

Posso implementare più interfacce

Tutti i metodi sono astratti, pubblici e non ha variabili istanza

PRINCIPI SOLID

Single responsibility principle

Una classe o un modulo deve avere un singolo motivo per il quale debba cambiare, ovvero una sola responsabilità.

Open-closed principle

Una classe deve essere aperta per le estensioni ma chiusa per le modifiche. Dobbiamo garantire che per l'estensione di una classe non ci sia bisogno di modificarla.

Liskov's substitution principle

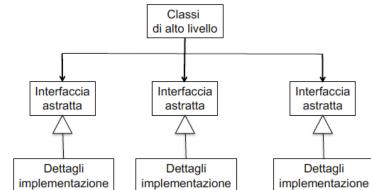
Le classi derivate devono essere completamente sostituibili alle classi base e non devono cambiarne il comportamento.

Interface segregation principle

Un client non deve dipendere da metodi che non usa. Le interfacce devono essere piccole, specifiche e molte.

Dependency Inversion principle

Le classi devono dipendere da astrazioni, non da dettagli concreti.



DESIGN PATTERNS

- CREATIONAL PATTERNS

Nascondono i costruttori e forniscono metodi alternativi per la creazione degli oggetti.

- BEHAVIORAL PATTERNS

Gestiscono il modo in cui interagiscono i diversi oggetti, tra loro.

- STRUCTURAL PATTERNS

Consentono di riutilizzare degli oggetti esistenti, fornendo agli utillizzatori un'interfaccia più adatta alle loro esigenze.

SINGLETION PATTERN

Si assicura che la classe abbia una singola istanza e un unico punto globale di accesso.

Motivazione

- Per alcune classi è importante avere una sola istanza
 - e.g., un singolo spooler per diverse stampanti
- La classe assicura che non possono essere create altre istanze e prevede un modo per accedere all'istanza

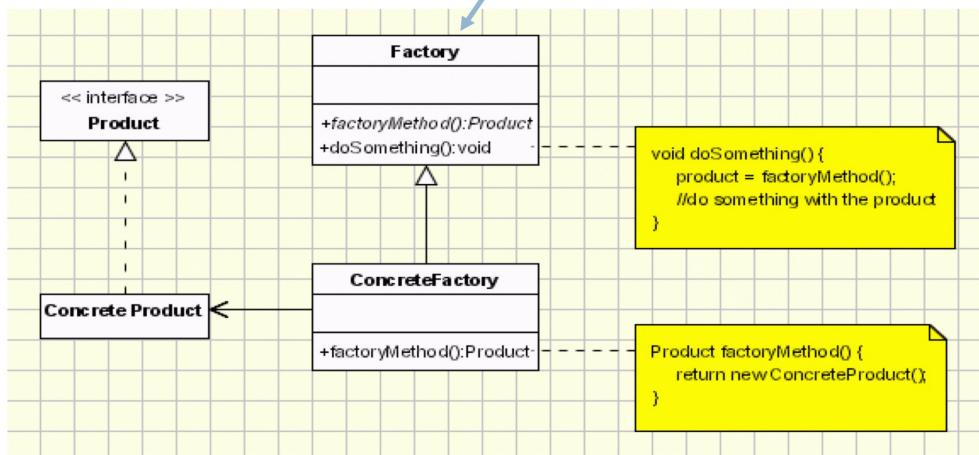
STRUTURA

Costuttore privato → non accessibile

metodo getInstance che restituisce l'istanza o la crea se non è presente

FACTORY METHOD

Definisce un'interfaccia per creare l'oggetto ma lascia la scelta del tipo alla sottoclasse.



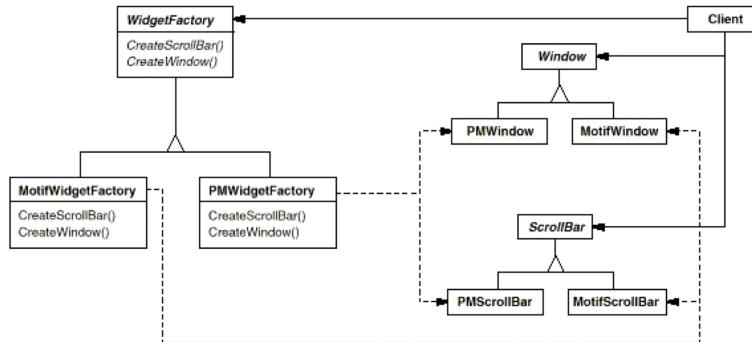
```
public Document CreateDocument(String type) {
    if (type.isEqual("html"))
        return new HtmlDocument();
    if (type.isEqual("proprietary"))
        return new MyDocument();
    if (type.isEqual("pdf"))
        return new PdfDocument ();
}
```

```
public void NewDocument(String type) {
    Document doc=CreateDocument(type);
    Docs.add(doc);
    Doc.open();
}
```

E' molto usato per la separazione tra applicazioni e famiglie di classi e permette modifiche con minimi cambiamenti nel codice.

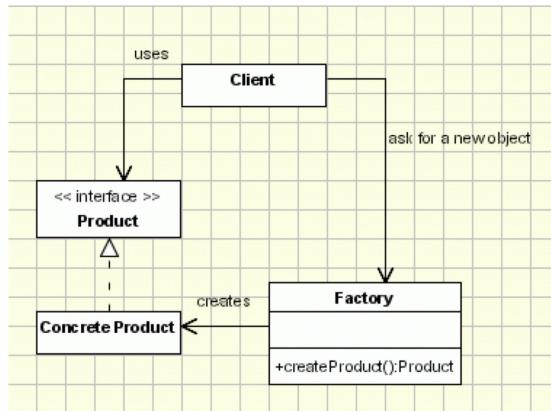
ABSTRACT FACTORY

Dispone di un'interfaccia per creare una famiglia di oggetti, senza specificare le loro classi concrete.



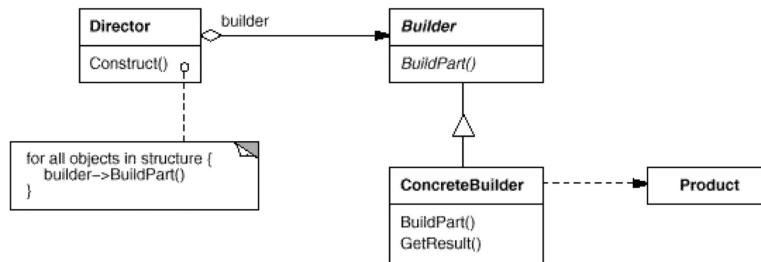
FACTORY PATTERN

Permette di creare oggetti, tramite un'interfaccia comune nascondendo la logica di istanziazione al client.



BUILDER PATTERN

Serve a separare la creazione di un oggetto complesso dalla sua rappresentazione



PROTOTYPE

Specifica il tipo di oggetti da creare utilizzando un'istanza prototipale e crea nuovi oggetti copiando le prototipi. Questo permette agli oggetti di poter creare nuovi oggetti, senza conoscere i dettagli.

ESEMPIO

- un **labirinto** con diversi **oggetti visuali**
- per generare **diverse mappe** del **labirinto**
 - **Muri, porte, passaggi, stanze, ...**
- **diversi prototipi** per i componenti

CHAIN OF RESPONSABILITY

Consente di separare il mittente di una richiesta dal destinatario in modo da consentire al più ad un oggetto di gestire la richiesta.

Gli oggetti destinatari vengono messi in catena e la richiesta viene trasmessa fino a trovare un oggetto che la gestisce.

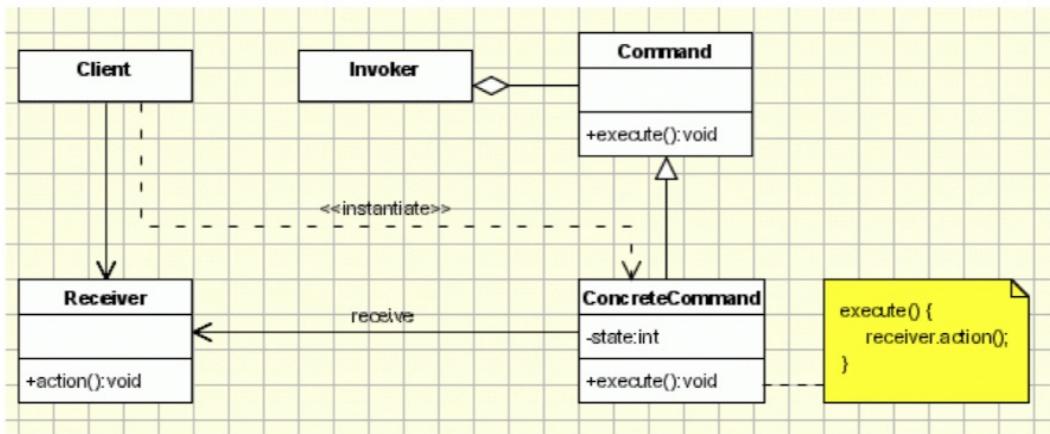
Il pattern CoR è usato quando

- non conosciamo a priori quale oggetto è in grado di gestire una determinata richiesta
- l'oggetto può essere sconosciuto staticamente
- l'insieme degli oggetti in grado di gestire richieste cambia dinamicamente a runtime

COMMAND

Incapsula la richiesta di un oggetto consentendo di parameterizzare il client con richieste diverse, accodare o mantenere uno storico delle richieste e gestire richieste cancellabili.

Si usa quando è necessario inviare richieste a oggetti senza conoscere nulla dell'operazione richiesta o del destinatario.



INTERPRETER

Dato un linguaggio, definisce una rappresentazione per la sua grammatica e un interprete che ne interpreta il linguaggio.

Se un tipo di problema si presenta spesso si esprimono le istanze del problema come proposizioni di un linguaggio

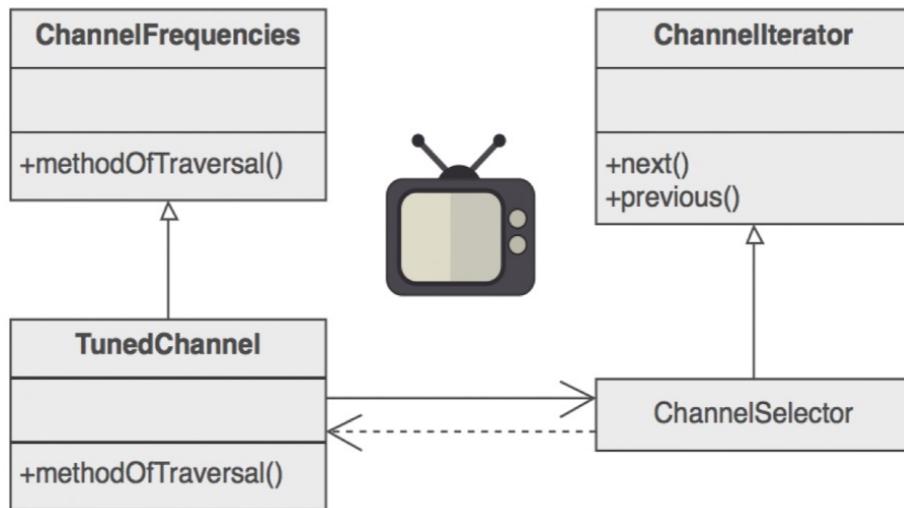
Es.

- Google Translate
- Compilator.

ITERATOR

Fornece un modo di accesso sequenziale agli elementi che formano un oggetto composto senza esporre la struttura interna.

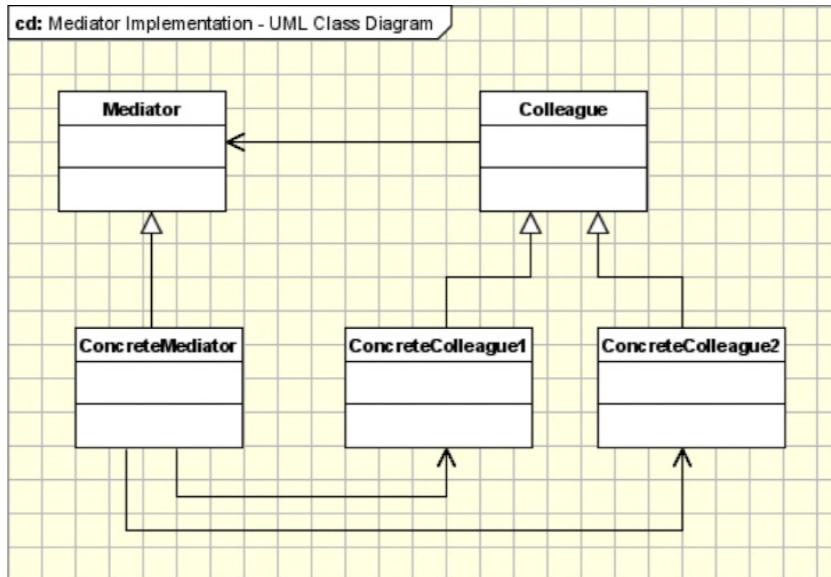
Pone la responsabilità di accesso e di attraversamento in un oggetto ausiliario detto iteratore.



MEDIATOR

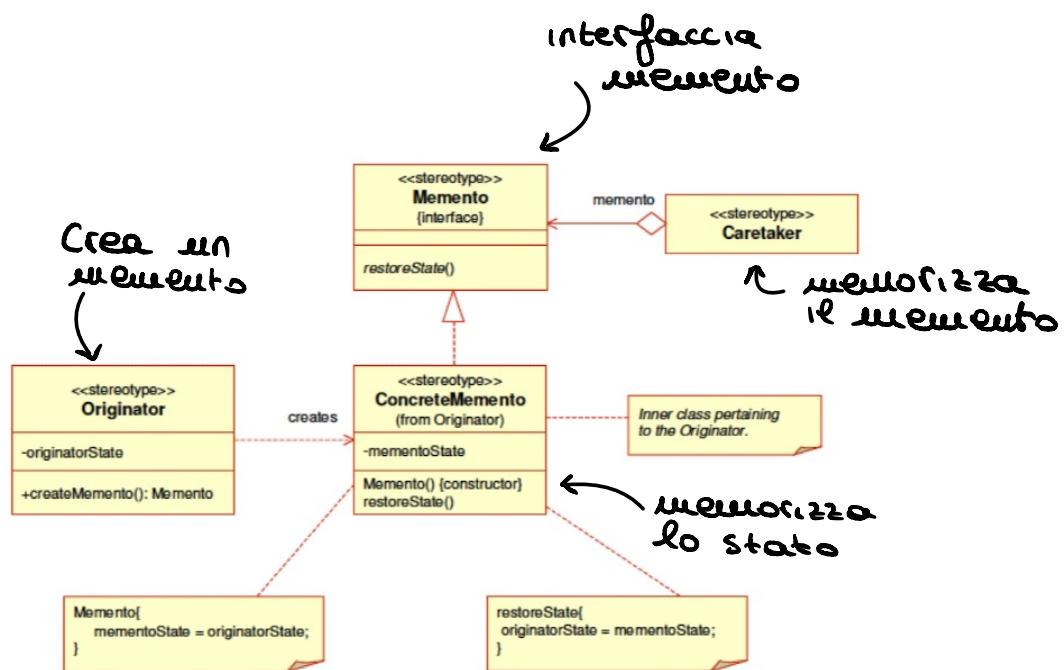
Definisce un oggetto che incapsula le modalità di interazione di altri oggetti. favorisce un basso accoppiamento evitando che gli oggetti facciano riferimento l'uno all'altro esplicitamente.

Permette di modificare agevolmente le politiche di interazione perché le entità fanno riferimento solo al mediatore



MEMENTO

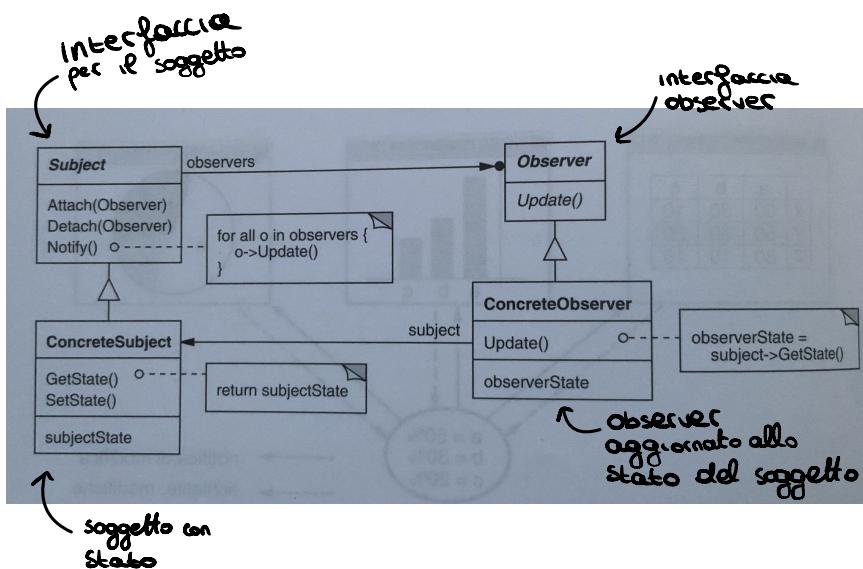
Serve a catturare ed esternare lo stato interno di un oggetto senza violare l'incapsulamento in modo tale che sia possibile, in un secondo momento ripristinare un oggetto nello stato esportato. Il processo è particolarmente utile per evitare errori, esempio calcolatrice.



OBSERVER

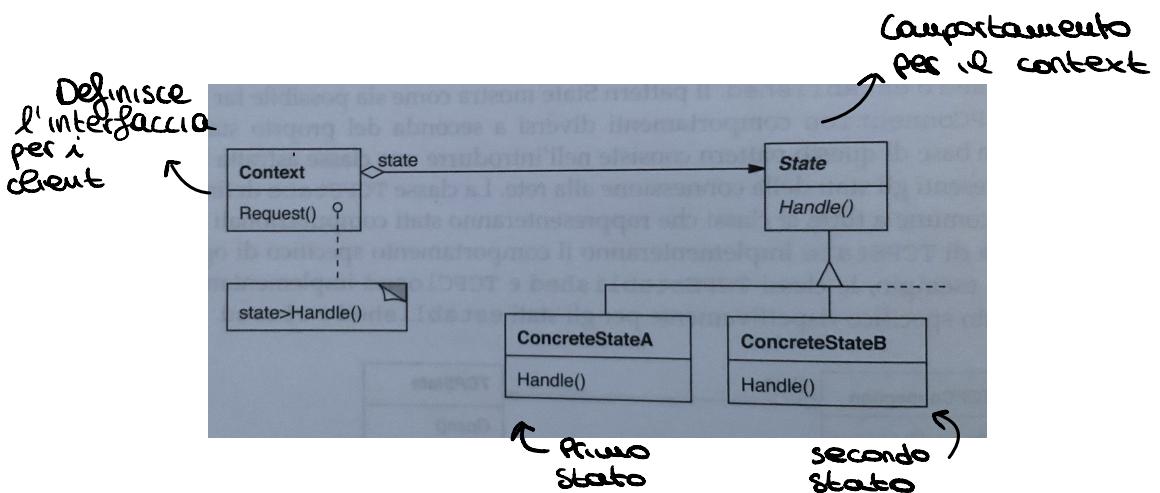
Definisce una dipendenza uno a molti fra oggetti, in modo tale che se un oggetto cambia il suo stato, tutti gli oggetti dependenti siano aggiornati automaticamente.

Serve a mantenere la consistenza tra gli oggetti dependenti.



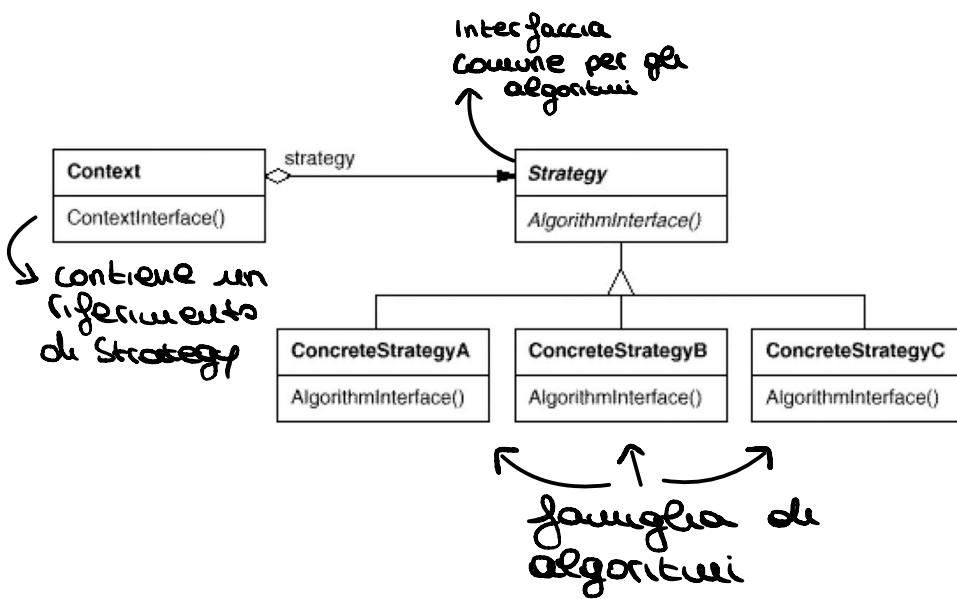
STATE

Permette ad un oggetto di cambiare il suo comportamento quando cambia il suo stato interno.
Serve per poter cambiare un oggetto a run-time in base allo stato nel quale si trova.



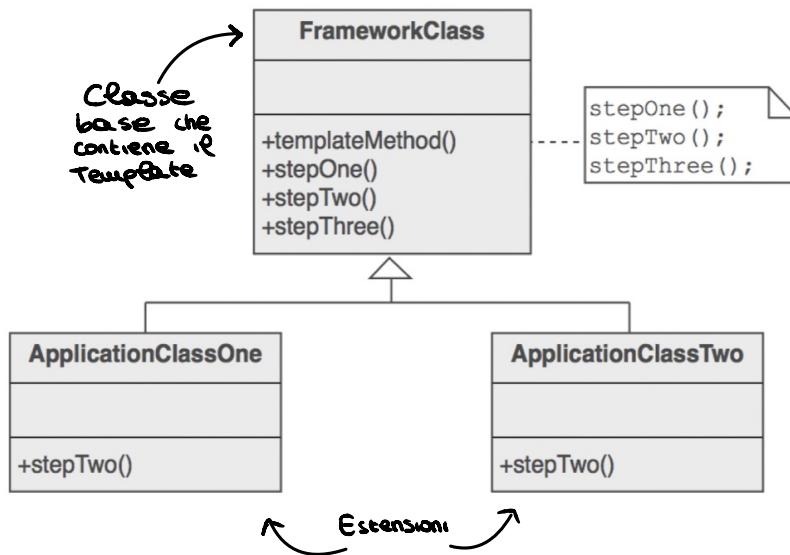
STRATEGY

Definisce una famiglia di algoritmi, incapsulati e intercambiabile. Permette agli algoritmi di cambiare indipendentemente dal client che ne fa uso.



TEMPLATE METHOD

Definisce lo scheletro di un algoritmo in un metodo riinviano alcuni passi alle sottoclassi client.
Permette alle sottoclassi di ridefinire alcuni passi senza dover implementare la struttura dell'algoritmo.



VISOR

Si applica agli elementi composti. Consente la definizione di nuove operazioni senza modificare le classi degli elementi su quali opera.

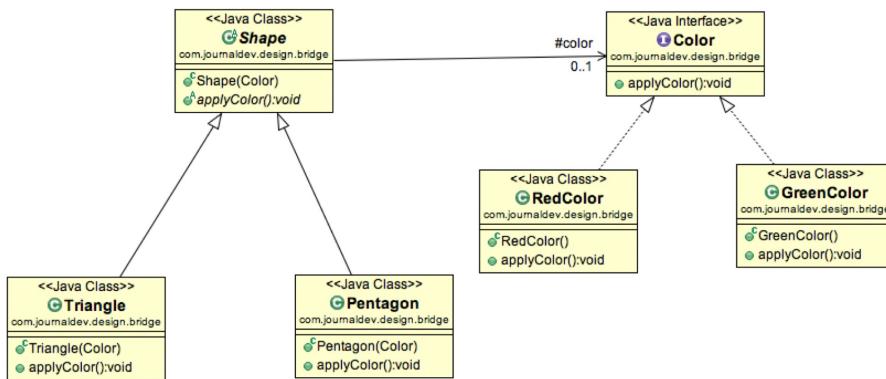
ADAPTER

Consente di convertire l'interfaccia di una classe in un'altra richiesta dal client.

Fornisce una soluzione al problema dell'interoperabilità tra diverse interfacce.

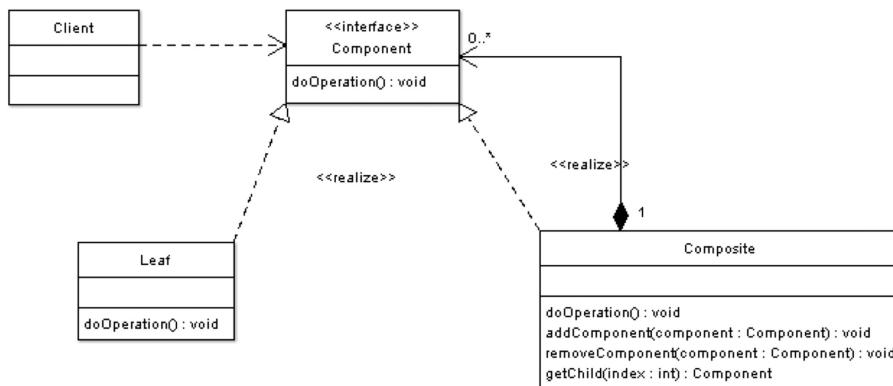
BRIDGE

Separa un'astrazione dalla sua implementazione in modo che entrambe possono funzionare autonomamente



COMPOSITE

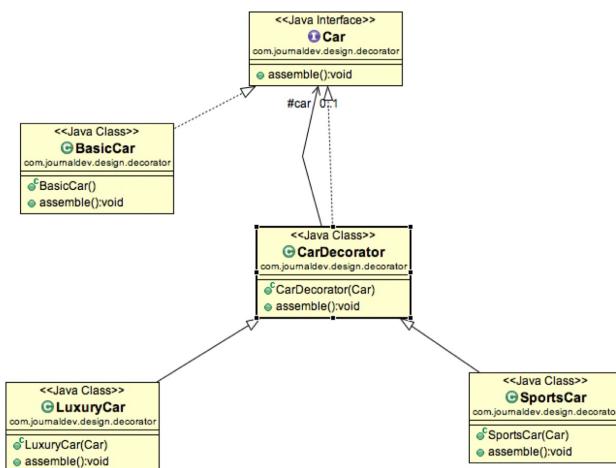
Comporre oggetti con una struttura ad albero per rappresentare gerarchie.
Permette di trattare oggetti singoli e composizioni di oggetti uniformi.



DECORATOR

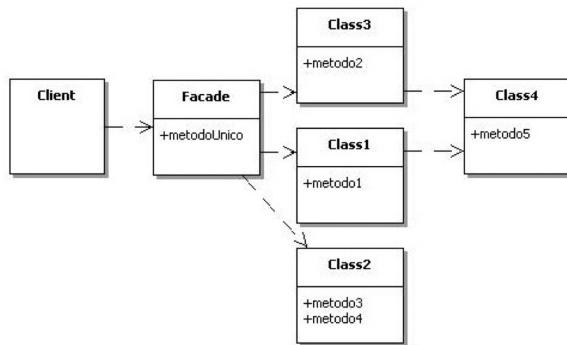
Aggiunge responsabilità aggiuntive ad un oggetto dinamicamente. Fornisce un'alternativa flessibile alla costruzione di sottoclassi per estendere delle funzionalità.

Consente di aggiungere durante il runtime nuove funzionalità.



FAÇADE

Fornisce un'interfaccia unificata ad un insieme di interfacce in un sottosistema.
Fa da facciata che si comporta come adattatore per le interfacce del sottosistema.



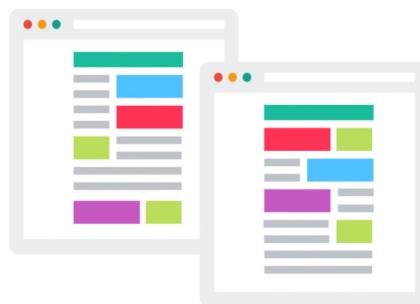
Facade

FLYWEIGHT

Serve a separare la parte variabile di una classe dalla parte che puo' essere riutilizzata.

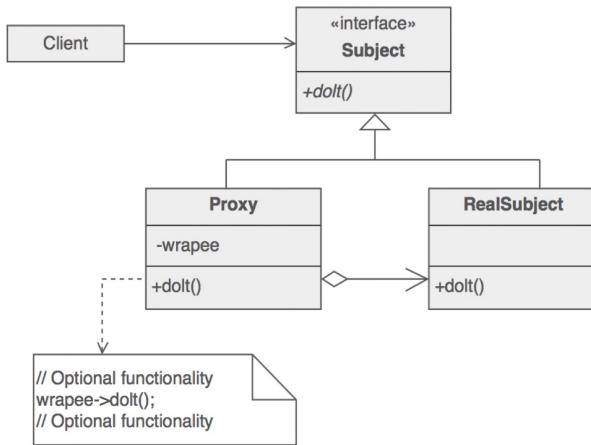
ES.

Browser loads images just once and then reuses them from pool:



PROXY

Fornisce un surrogato per un altro oggetto per controllare l'accesso ad esso.



ECCEZIONI

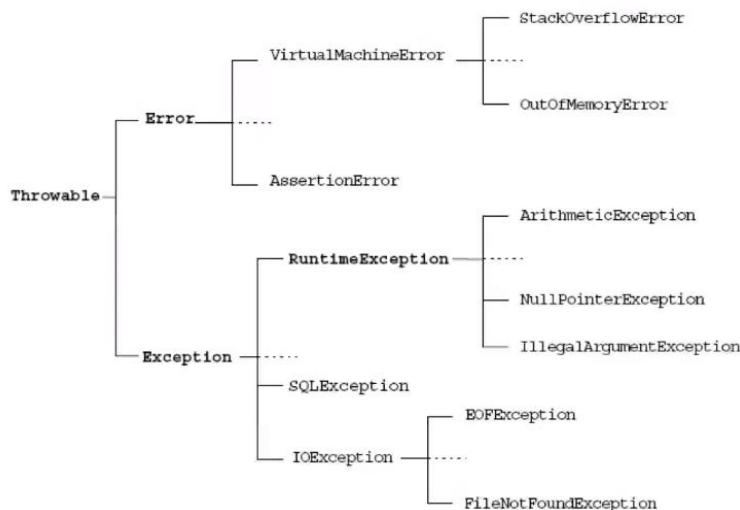
Sono delle condizioni che si possono verificare in un codice corretto semanticamente e logicamente.

Eccezioni controllate:

- Sono rappresentate dalla classe **Exception**
- Possono essere eseguite da un modulo

Eccezioni non controllate:

- situazioni fatali → **Error class**
- bugs o errori → **RuntimeException**



REGOLE DI GESTIONE E DI DICHIARAZIONE

- Gestire le eccezioni tramite il try-catch-finally
- Dichiare che il codice causa una eccezione usando la clausola throws.
- Se l'eccezione non viene gestita correttamente causa crash del programma.

ECCEZIONI E OVERRIDE

Il metodo overriding puo' eseguire:

- No exceptions
- una o piu' eccezioni eseguite dal metodo sovrascritto
- una o piu' sottoclassi di eccezioni eseguite.

Non puo' eseguire:

- Eccezioni aggiuntive
- Superclassi di eccezioni

```
public class TestA {  
    public void methodA() throws IOException {  
        // do some file manipulation  
    }  
}  
public class TestB1 extends TestA {  
    public void methodA() throws EOFException {  
        // do some file manipulation  
    }  
}
```

RIVEDI CREARE ECCEZIONI

GESTIONE DELLE EXCEPTION DEFINITE

THROW:

Si usa con eccezione
che dichiarate tramite
la parola chiave THROWS

```
1 public void connectMe(String serverName)
2     throws ServerTimedOutException {
3     boolean successful;
4     int portToConnect = 80;
5
6     successful = open(serverName, portToConnect);
7
8     if ( ! successful ) {
9         throw new ServerTimedOutException("Could not connect",
10                           portToConnect);
11    }
12 }
```

TRY-CATCH:

Instruzia l'eccezione e
la utilizza direttamente

```
public void findServer() {
    try {
        connectMe(defaultServer);
    } catch (ServerTimedOutException e) {
        System.out.println("Server timed out, trying alternative");
        try {
            connectMe(alternativeServer);
        } catch (ServerTimedOutException e1) {
            System.out.println("Error: " + e1.getMessage() +
                               " connecting to port " + e1.getPort());
        }
    }
}
```

ESEMPIO CREAZIONE:

Questa classe sarà contenuta
in una super che
implementa Exception

```
public MyException(String message, int a, int b) {
    super(message+(a+"+a+", b)+"+b+"));
    this._a=a;
    this._b=b;
}
public int getA() {
    return _a;
}
public int getB() {
    return _b;
}
```

PROPRIETA' DI SISTEMA

Sostituiscono le variabile d'ambiente permettendo a Java di essere eseguito su più sistemi.

CLASSE PROPERTIES

STREAM I/O

Uno stream è un flusso di dati.

InputStream per la lettura

SourceStream

OutputStream per la scrittura

SinkStream

1) Stream basati su caratteri, c'è bisogno di vari adattamenti per la lettura o scrittura dei dati.
La rappresentazione predefinita è UNICODE

- LETTURA → classe Reader
- SCRITTURA → classe Writer

2) Stream basati su byte, i byte saranno sempre considerati in dicitura Big Endian.

- LETTURA → classe StreamReader
- SCRITTURA → classe StreamWriter

Nodi di Stream

Type	Character Streams	Byte Streams
File	FileReader FileWriter	FileInputStream FileOutputStream
Memory: array	CharArrayReader CharArrayWriter	ByteArrayInputStream ByteArrayOutputStream
Memory: string	StringReader StringWriter	N/A
Pipe	PipedReader PipedWriter	PipedInputStream PipedOutputStream

THREADS

Sono una CPU virtuale, si dividono in:

- CPU
- codice
- Dati

Si usano perché:

- migliorano il design della programmazione OO
- semplificano il codice
- migliorano coerenza e singola esecuzione, etc'

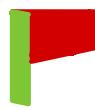
thread.start → fa partire l'esecuzione del thread

thread.join → permette ad un thread di partecipare

thread.sleep → mette in pausa l'attività di un thread

run → metodo dell'interfaccia Runnable, specifica il comportamento del thread

LOCK FLAG



Ogni oggetto ha un lock flag, si usa per proteggere l'accesso ai dati sensibili.

Per interagire con il lock flag si utilizza la parola chiave synchronized.

Il blocca flag viene sbloccato nei seguenti eventi:

- Si sblocca quando un processo passa alla fine del blocco di codice synchronized
- Si blocca automaticamente quando incontra un break, un return, o un'eccezione della synchronized del blocco di codice

