

Programmazione II

appunti + riassunto libro, a.a. 2021/2022

Introduzione	2
Encapsulation and information hiding	4
Design by contract	7
Types and Interfaces	8
Iterator	15
Strategy	16
Chain Responsibility	17
Interface Segregation Principle	18
Object State	19
NullObject	21
Singleton	24
Flyweight	26
Nested Classes	27
Unit Testing	30
Associazione, aggregazione e composizione	33
Composite	34
Iterator deck e diagrammi	39
Decorator	41
Law of Demeter	46
Inheritance	49
Adapter	53
Command	55
Rivisitazione pattern Decorator	57
Template	59
Principi SOLID	60
Observer	62
Factory Method	67
MV*	70
MVC	70

Introduzione

software design process = The construction of **abstractions** of data and computation and the organization of these abstractions into a working software application.

Il processo di progettazione del software è la costruzione di dati (nuove entità) e computazioni/elaborazioni (metodi e funzioni) e l'organizzazione di queste astrazioni in un applicativo software funzionante.

astrazione = l'amplificazione dell'essenziale e l'eliminazione del irrilevante, nasconde i particolari per far cogliere ciò che è importante al livello in cui ci interessa [Robert Martin].

object orientation = mezzo per organizzare questa astrazione in un linguaggio di programmazione.

Es.: codice sorgente in C "The Twelve Days of Christmas" di Ian Phillips. E' stato uno dei vincitori del concorso annuale "International Obfuscated C Code" nel 1988.

```
main(t,_,a ) char* a;{return!0<t?t<3?main(-79,-13,a+main(-87,
1-_,main(-86, 0,a+1 )+a)):1,t<_?main( t+1, _, a ):3,main(-94,
-27+t, a )&&t == 2 ?_<13 ? main ( 2, _+1,"%s %d %d\n" ):9:16:
t<0?t<-72?main( _, t,"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{**+\\
,/w{q+,/w#q#n+,/#{l,+,/n{n+,/+n+,/#;#q#n+,/+k#;*,/'r : 'd*'\
3,}{w+K w'K: '+'}e#';dq# 'l q#'+d'K#!/+k#;q# 'r}eKK#}w' r}eKK{n\\
'/#;#q#n')}{#}w')}{n\\ '/+ #n';d}rw' i;# )n\\ !/n{n#'; r{#w' r\\
nc{n\\ '/#{l,+ 'K {rw' iK{;[{n\\ '/w#q#n'wk nw' iwk{KK{n\\ !/w{\\
% 'l##w# ' i; :{n\\ '/*{q# 'ld;r'}{nlwb!/*de}'c ; ;{nl' -{ }rw' '/+,\\
}## ' }#nc, '#nw' '/+kd'+e}+;# 'rdq#w! nr/ ' ) }+}{rl# '{n' ' )# \\
}'+}##(!!/"):t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main
(( *a == '/' )+t, _,a+1):0<t?main ( 2, 2 , "%s"):*a=='/'||main
(0,main(-61,*a,"!ek;dc i@bK'(q)-[w]*%n+r3#l,{ }:\nuwloca-0;m\\
.vpbks,fxntdCeghiry"),a+1);}
```

Un codice di questo tipo con la giusta indentazione e aggiunta di metodi diventa chiaro e di facile lettura, la qualità viene aumentata (qualità come: efficienza, modificabilità, eleganza, chiarezza, correttezza e verificabilità).

Qualità del software

In pratica il design process è un processo di decision making.

design space = immaginiamo uno spazio a n dimensioni in cui ogni dimensione corrisponde a una **qualità** del software (efficienza, verificabilità, usabilità, facilità di implementazione, etc...). Ogni **soluzione** a un problema può essere rappresentata come un punto in questo spazio → difficile valutare una soluzione come la migliore, è soggettivo.

Dentro questo design space ogni specifica decisione di design (o un coerente set di decisioni) corrisponde a una coordinata nello spazio che rappresenta le conseguenze delle decisioni. In pratica, qualsiasi design decision è probabile che sia buona in una dimensione ma meno in un'altra, e questo viene chiamato **design trade-off**.

Obiettivo comune a tutte le soluzioni è la **semplicità** (reduce the complexity).

design context = è uno specifico set di requisiti e vincoli all'interno di un dominio in cui una soluzione di design deve essere fondata e integrata.

refactoring = rimettere mano a del codice che funziona senza aggiungere funzionalità al fine di migliorare il design, preparare il codice a nuove funzionalità o eliminare **debito tecnico** (finché il

codice non è a posto ogni intervento successivo “costa” di più, si accumula un debito di complicazioni).

design knowledge = **come fare** una cosa e le **ragioni**/intenzioni che hanno portato a tale scelta. L'idea di avere le design decision evidenti automaticamente nel codice viene chiamata **sustainability**.

Le scelte possono essere salvate:

- nella memoria del programmatore
- in documenti scritti di design in linguaggio naturale e diagrammi
- tramite tool come forum
- tramite modelli specializzati UML (=Unified Model Language)
- in una documentazione nel codice, grazie a una serie di test (in questa modalità è difficile spiegare le ragioni delle scelte di design)

come si deve progettare? non un design specifico, ma il *know how* di come si deve progettare bisogna lavorare su:

- il metodo, lavorare sul processo di lavoro
- il design pattern, trovare situazioni ricorrenti
- i principi (es.: SOLID)

//Lezione di introduzione a Java e a IntelliJ IDEA

Encapsulation and information hiding

The construction of **abstraction** of data and computation and the organization of these abstractions into a working software application.

L'**incapsulamento** rende l'astrazione **opaca** (non posso vedere cosa c'è dentro) e **chiusa** (nascondo tutti i particolari). L'astrazione è perciò legata all'identificazione e scomposizione (dividi-et-impera) di queste "scatole".

Una buona astrazione deve godere di alcune proprietà:

- **coesione interna** = ci deve essere dentro **solo** quello che serve.
- **disaccoppiamento esterno** = tutto quello che c'è, che serve, è dentro la "scatola", la classe stessa.

Qualche collegamento esterno esiste, però il buon design impone che siano pochi e chiari.

Esempio di encapsulation: rappresentazione di un Deck

Un mazzo di 52 carte è composto da:

- 4 semi
- 13 valori

Cerchiamo di ideare una **struttura dati** capace di rappresentare un mazzo o una mano (sequenza) di carte tratte dal mazzo.

Una prima idea per definire l'oggetto mazzo è fare una astrazione come tipi (in questo caso il tipo intero e il tipo contenitore array/lista). Si può rappresentare il mazzo come una classe contenente un array cards di interi, ovvero mappiamo le carte su 52 interi (0-12 cuori, 13-25 quadri,...).

```
class Deck {  
    int[] cards;  
}
```

Si può anche pensare che non serva creare una classe, ovvero si astrae il mazzo come array di interi.

```
int[] deck;
```

Questa astrazione però non è applicabile a tutti i problemi, non esiste sempre un array con il tipo adatto al nostro problema.

Per conoscere una carta bisogna fare delle semplici operazioni.

```
int card = 15;  
int suit = card/13;    // 1 => Quadri  
int rank = card%13;    // 2 => Tre (si comincia da 0)
```

Questa rappresentazione è scomoda infatti:

- fare operazioni per capire il seme e il valore è laborioso
- non si nasconde, non si rende limitato (quanto è il valore massimo che una carta può assumere? non è stabilito), fare operazioni sulla variabile card permette di fare operazioni illecite (sommare una carta con l'altra)
- poco comunicativo

Si potrebbe fare una struct con un array di due interi (array di array) che rappresentano le informazioni che caratterizzano una carta (seme e valore).

```
int[] card = {1,3};    // 1 = Quadri, 3=Tre (comincia da 1)
```


Ma anche in questa matrice è difficile interpretare in che direzione bisogna andare.

Si può rappresentare un mazzo di carte in diversi modi ma tutte queste strutture dati non stanno facendo una buona **astrazione**.

information hiding = solo ciò che è nascosto può essere cambiato liberamente e senza pericoli. I due scopi fondamentali sono:

- facilitare la comprensione del codice
- rendere più facile modificarne una parte senza fare danni

Tornando all'esempio del mazzo, mappare carte su interni non è una buona idea, infatti il significato di tali numeri non è chiaro ed è facile dare valori impossibili (es.: card con valore 100) e fare operazioni illecite.

Si può dare più chiarezza implementando dei tipi (diamo un nome ai tipi):

```
class Card {
    int value; //0-51
}

class Deck {
    Card[] cards;
}

Card card = new Card();
card.value = 28;
```

Si può migliorare la rappresentazione interna usando una tecnica **enumerativa**, si usa un tipo enum dove si cita **esplicitamente** quali valori può assumere una variabile, nel caso del mazzo di carte nome del seme e valore della carta. Da una parte rende la struttura semplice e dall'altra rende il codice di facile lettura, non è ambiguo e risolviamo il problema dei valori impossibili, ma non la combinazione di carte non valide, come null of null. Per risolvere questo problema si usano i costruttori e le asserzioni.

```
enum Suit {CLUBS, DIAMONDS, SPADES, HEARTS}
enum Rank {ACE, TWO, ..., KING}

class Card {
    Suit suit;
    Rank rank;
}

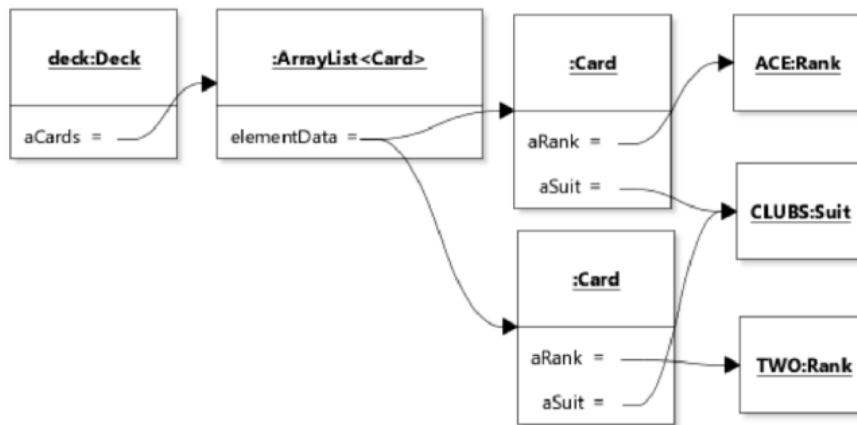
Card card = new Card();
card.suit = Suit.CLUBS;
card.rank = Rank.ACE;
```

Come si fa a nascondere la rappresentazione interna? In java ogni membro di una classe, e quindi anche gli attributi, ha un modificatore di accesso che può essere:

- **public** (public Suit suit è un campo pubblico accessibile anche da oggetti e parti di codice che non sono all'interno della classe Card).
- **private** (private Suit suit è un campo privato e non vi si può più accedere liberamente, bisogna passare dalle operazioni/metodi che la classe mette a disposizione).

I metodi più semplici da implementare per accedere a campi privati sono:

- **getter** = serve per accedere in lettura a un valore (non corrisponde per forza alla natura interna del valore).
- **setter** = significa che si vuole modificare un valore (accesso in scrittura).



Principio Tell-Don't-Ask = non chiedere i dati, ma di cosa vuoi che faccia sui dati.

UML = Unified Model Language, linguaggio composto da diversi diagrammi.

Una classe è rappresentabile, astraendola dal linguaggio, tramite i diagrammi delle classi UML, ovvero gli object diagrams. Si tratta di uno standard ISO. Aiutano il programmatore ad avere un'idea di cosa succede in memoria.

Ogni "scatola" è composta da:

- nome oggetto . classe di appartenenza
- attributi (ci può essere un riferimento/puntatore ad un oggetto)

escaping references = i riferimenti che dovrebbero essere segreti vengono persi.

oggetti con immutabilità = la classe string è un esempio di classe composta da oggetti immutabili, non c'è maniera di cambiare lo stato dell'oggetto dopo la sua inizializzazione. Un oggetto è immutabile quando:

- non ci sono dei metodi che modificano l'oggetto.
- gli attributi sono `private` e `final` (ovvero il contenuto di quell'identificatore non può cambiare).
- si assicura accesso esclusivo a tutte le parti non immutabili (no escaping references).

Questa proprietà semplifica la comprensione del codice e protegge la segretezza.

exposing internal data = a volte può essere necessario esporre parte dello stato interno, per esempio se si vuole sapere quali carte sono nel mazzo. Per fare exposing si può:

- **estendere l'interfaccia**, aggiungendo metodi in modo da poter controllare le operazioni.
- **restituire delle copie**, invece di ritornare un riferimento diretto allo stato ne ritorna una copia, questo viene detto **shallow copy**. Questa è un'operazione più pesante rispetto a estendere l'interfaccia, inoltre se si dovesse fare riferimento a oggetti non immutabili bisognerebbe fare una copia in profondità, **deep copy** (esempio di `ArrayList` di carte, non solo si fa la copia dell'`ArrayList` ma si deve fare anche la copia delle sue carte originali).
- **usare iteratori**, tramite il design pattern ITERATOR.

Design by contract

Design By Contract = is an approach to designing robust yet simple software. It provides methodological guidelines to achieve these goals without resorting to defensive programming. Instead, Design by Contract builds class invariants and pre/post validation of methods arguments and return values into the code itself.

The term was coined by Bertrand Meyer in connection with his design of the Eiffel programming language and first described in various articles starting in 1986 and the two successive editions (1988, 1997) of his book Object-Oriented Software Construction.

Il Design By Contract è una metodologia per progettare software.

La sua idea centrale è che le entità software hanno degli obblighi nei confronti di altre entità in base a regole formalizzate fra di essi. Una specifica funzionale, o contratto, viene creato per ogni modulo nel sistema prima che sia codificato. L'esecuzione del programma è quindi vista come l'interazione fra i vari moduli vincolati da questi contratti.

In generale, i metodi hanno:

- **precondizioni** esplicite che il chiamante deve soddisfare prima di chiamare il metodo
- **postcondizioni** esplicite che descrivono le condizioni che il metodo garantirà essere vere nel momento in cui il metodo finisce.

Per segnalare la presenza di precondizioni in java si usano dei **commenti** strutturati caratterizzati da `@pre`.

```
/**
 *@pre pRank!=null && pSuit!=null
 */
public Card (Rank pRank, Suit pSuit){
    rank = r;
    suit = s;
}
```

Questo commento non fa nulla sul codice, ma può essere dato al programmatore ed essere integrato alla documentazione.

La keyword `assert` ha la stessa funzione dei commenti `@pre` ma, se la condizione non viene rispettata, genera un errore se il programma viene compilato con l'opzione `-ea`, ovvero enable assertion.

```
public Card (Rank pRank, Suit pSuit){
    assert r!=null && s!=null;
    rank = r;
    suit = s;
}
```

Types and Interfaces

interfaccia di una classe = insieme di operazioni accessibili ad altre classi (oggetti, codice esterno alla classe), ovvero i metodi `public` (senza implementazione o attributi).

Aggiungiamo una classe `Client` che usi `Deck` e `Card`.

```
public class Deck {
    public void shuffle() {...};
    public Card draw () {...};
    public boolean isEmpty() {...};
    public void sort() {...};
}
public class Client {
    private Deck deck = new Deck();
    ...
    deck.shuffle();
}
```

Le due classi però sono fortemente accoppiate: `Client` conosce `Deck` e può lavorare solo con quello. Non potrebbe ad esempio mescolare un mazzo con due serie di 52 carte per giocare a ramino. C'è un problema al livello di dipendenze (dependency inversion of SOLID principles).

i [SOLID principles](#) sono:

- **S** - Single responsibility
- **O** - Open/closed = aperto ai cambiamenti/le evoluzioni delle funzionalità ma chiuso alle modifiche del codice.
- **L** - Liskov substitution
- **I** - Interface segregation
- **D** - Dependency inversion

L'ultimo principio, la dependency inversion, dice di invertire le dipendenze classiche:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

Per giocare con un numero di carte >52 bisogna creare una **astrazione** del mazzo.

Dato il seguente codice si può estrarre una interfaccia

```
public static List<Card> drawCards(Deck deck, int number) {
    List<Card> result = new ArrayList<>();
    for (int i = 0; i < number && !deck.isEmpty(); i++) {
        result.add(deck.draw());
    }
    return result;
}
```

Questo metodo funziona solo per gli oggetti di tipo `Deck`, in particolare gli interessa che l'oggetto abbia i due metodi `draw` ed `isEmpty`.

Si può definire una **interfaccia** `CardSource` che definisce proprio quei due metodi.

```

public interface CardSource {
    /**
     * @return The next available card.
     * @pre !isEmpty()
     */
    Card draw();

    /**
     * @return True if there is no card in the source.
     */
    boolean isEmpty();
}

public class Deck implements CardSource {...}

```

Bisogna dichiarare che la classe Deck **implementa** l'interfaccia CardSource, ciò significa che Deck deve implementare i metodi draw() ed isEmpty().
Ovunque compaia un identificatore di tipo cardSource gli si può associare un oggetto di tipo compatibile, come Deck.

```
CardSource source = new Deck();
```

Il concetto di **polimorfismo** e di **dinamicità** permette di definire una parte di codice che invoca un'altra parte di codice che non è ancora stata scritta.

Questa capacità di creare uno scheletro fisso è collegata al **OpenClosed principle**, il principio di extensibility. Diverse classi possono aderire alla stessa interfaccia.

Esempio nelle librerie standard di Java

```

public class Deck {
    private List<Card> cards = new ArrayList<>();
    public void shuffle() {
        Collections.shuffle(cards);
    }
}

```

- Cards è un oggetto di tipo ArrayList
- che implementa interfaccia List
- che è passabile come parametro a shuffle() di Collections
- visto che accetta List di qualunque elemento

```
public static void shuffle(List<?> list)
```

Infatti per disordinare una lista di elementi non c'è bisogno di guardarli, ma se volessimo riordinare il mazzo di carte, potremmo usare la seguente funzione?

```

public void sort() {
    Collections.sort(cards);
}

```

comparable interface = è presente nelle librerie. Si può implementare quando si vuole comparare un oggetto di una certa classe con un oggetto di tipo T.

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

Definisce un singolo metodo che restituisce un valore minore, uguale o maggiore a 0, a seconda del risultato del confronto. Questo metodo è quello che userà la sort di Collections. Quindi l'interfaccia di Collection.sort è:

```
public static <T extends Comparable<? super T>> void sort(List<T> list);
```

Questa è la definizione di una sort generica rispetto al tipo generico di un suo parametro. Cioè sort ha bisogno che gli elementi della lista da ordinare implementino tutti Comparable, e quindi siano confrontabili tra di loro.

```
class Card implements Comparable<Card> {  
    private CardOrderType orderType; // con un setter  
  
    public int compareTo(Card other) {  
        switch (orderType) {  
            case SUITFIRST:  
                return suit.compareTo(other.suit);  
            case RANKFIRST:  
                return rank.compareTo(other.rank);  
        }  
    }  
}
```

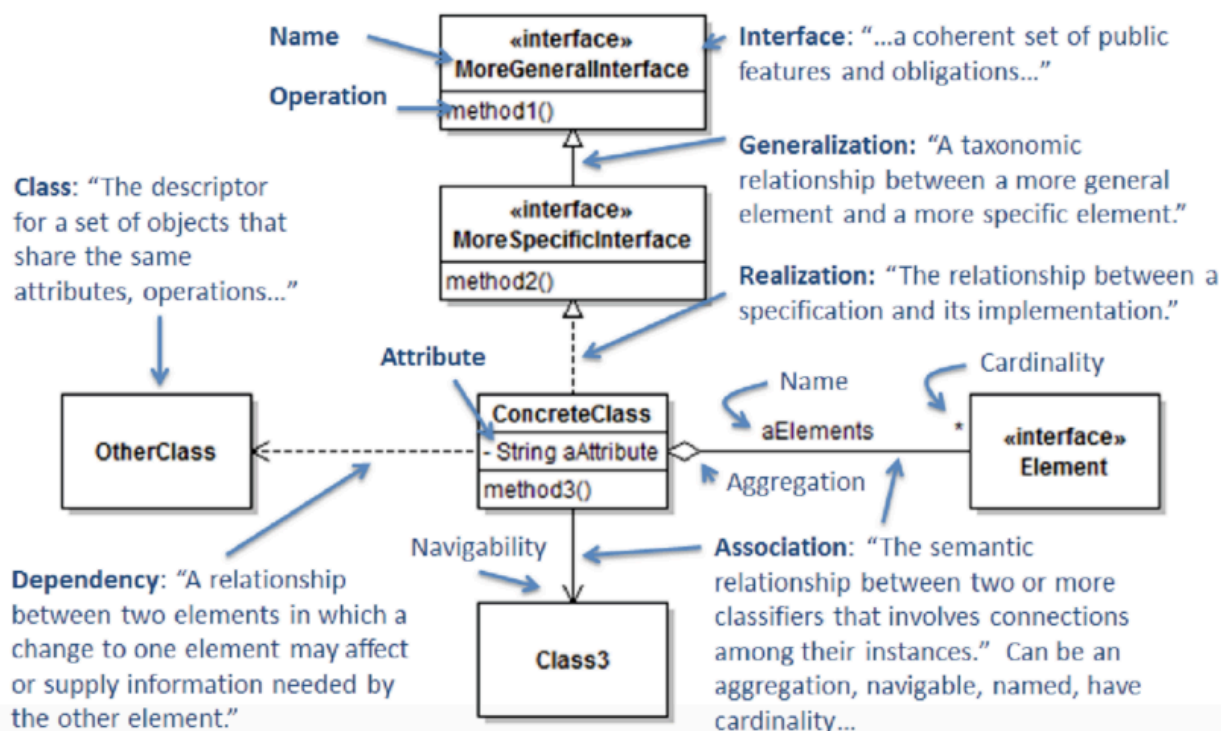


Diagramma delle classi = i box che rappresentano le classi sono composti da:

- nome
- tipo degli attributi
- assegnare dei metodi

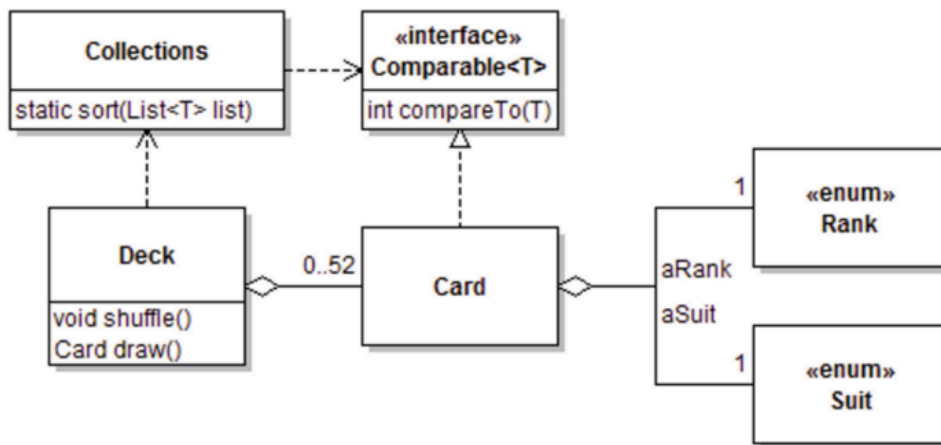
La **navigabilità**, ovvero la direzione della freccia, esprime in quale **direzione** esiste la conoscenza. L'**implementazione** di un'interfaccia è espressa con una linea tratteggiata con la punta bianca.

In questo caso ConcreteClass implementa l'interfaccia MoreSpecificInterface, e allo stesso tempo eredita anche l'implementazione di MoreGeneralInterface, l'estensione di MoreSpecificInterface.

Una classe è **associata** a un'altra classe se le istanze della prima classe devono conoscere (a causa di un attributo o di un metodo) istanze della seconda classe. Una istanza di ConcreteClass è associata ad una istanza della Class3 (conosce le istanze di Class3).

L'**aggregazione** è un particolare tipo di **associazione** in cui viene identificato un ruolo di contenimento, in uno dei due estremi, dell'altro. ConcreteClass non solo è associata a Elements ma li contiene, fanno parte della sua definizione.

Quando viene espressa una relazione non si va a citare l'attributo.



Deck è la classe principale che contiene delle carte, il mazzo ha una relazione di **aggregazione** con le sue carte. La relazione è da Deck verso Card, non il contrario, anche se in questo schema non è segnata la navigabilità.

I metodi citati sotto la classe Deck non sono tutti, ma un sottoinsieme, sarebbe meglio citarli tutti o non citarne nessuno.

Card **implementa** Comparable.

Collections è una classe in cui esistono solo metodi statici e neanche un attributo. Fornisce funzionalità a oggetti astratti, come le interfacce. Serve per esempio quando bisogna definire un metodo di sorting che lavori su un qualunque oggetto lista di elementi comparabili.

Deck **dipende** dalla classe Collections e allo stesso modo Collections **dipende** dalle istanze aderenti l'interfaccia Comparable.

Rank e Suit esistono quando nasce una carta, si forza il fatto che non possono assumere valore null.

L'ordinamento può essere:

- **naturale** = ordine unico con cui trattare gli elementi (ho le carte e si confronta prima il numero e poi il rank)
- **multiplo** = quando si vuole avere più forme di confronto (ordinamento di stringhe che non tiene conto del case/ case insensitive). Se però si aggiungono tanti criteri di ordinamento si può cadere nell'antipattern SWITCH STATEMENT† e si potrebbe andare contro i principi SOLID.

oggetti funzione = oggetto che contiene solo una funzione. Per darne una versione astratta si definisce un'interfaccia. Collections.sort ha due versioni:

- la versione in cui viene passato un container che ha elementi comparabili
- la versione in cui va bene come parametro un container qualunque, a patto che come secondo elemento venga passato un oggetto comparatore (istanza di qualcosa aderente a Comparator) che permette di confrontare i due elementi.

```

public interface Comparator<T> {
    int compare(T o1, T o2);
    ... }
// usata da
Collections.sort(List<T> list, Comparator<? super T> c)
  
```


per dare una definizione esterna di questa comparazione ci sono diversi approcci:

1. classi innestate
2. classi anonime
3. lambda expression

classi innestate = è il metodo più semplice per non rischiare di rompere l'incapsulamento, qualunque cosa esterna alla classe `Card` non potrà accedere direttamente a questi campi. Per citarlo bisogna citare anche la classe che contiene la classe innestata come `Card.SuitComparator`. Risolve il problema che il confronto sarà probabilmente basato su campi privati non più accessibili. Le classi innestate possono accedere ai campi privati della classe che li contiene. Questo però viola il principio SOLID open/closed.

```
public class Card {
    private Suit suit;
    public static class SuitComparator implements Comparator<Card> {
        @Override
        public int compare(Card card1, Card card2) {
            return card1.suit.compareTo(card2.suit);
        }
    }
}
Collections.sort(cards, new Card.SuitComparator());
```

classi anonime = si può non dare un nome a una classe, soprattutto se viene istanziata una sola volta. Si crea un oggetto di una classe di cui si dà una definizione ma non se ne cita il nome. La definizione è dentro il chiamante `Deck` e non all'oggetto chiamato `Card`.

```
public class Deck {
    public void sort() {
        Collections.sort(cards, new Comparator<Card>() {
            public int compare(Card card1, Card card2) {
                // non ha accesso diretto a campi privati
                return card1.getSuit().compareTo(card2.getSuit());
            }
        });
    }
}
```

lambda expression = se la classe anonima necessita della definizione di un solo metodo si può semplificare la sintassi.

```
public class Deck {
    public void sort() {
        Collections.sort(cards, (card1, card2) ->
            card1.getSuit().compareTo(card2.getSuit()));
    }
}
```

Iterators = li avevamo già citati come mezzo per dare accesso a parti private **proteggendo** encapsulation.

Diventa anche un pattern che dà accesso alle parti private di un contenitore proteggendo l'incapsulamento.

Rispetto alla soluzione che fornisce una copia della lista, ha il vantaggio che:

- nasconde dettagli sulla struttura contenitrice
- la struttura risulta (di default) read-only, ma prevede anche versione estesa

L'astrazione avviene tramite le funzioni `hasNext()` e `next()`.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();

    default void remove() {
        throw new UnsupportedOperationException("remove");
    }

    default void forEachRemaining(Consumer<? super E> action) {
        Objects.requireNonNull(action);
        while (hasNext())
            action.accept(next());
    }
}
```

iterare con iteratori = se `getCards()` ritorna un `Iterator`

```
for(Iterator<Card> cardIterator = deck.getCards(); cardIterator.hasNext();){
    Card card = cardIterator.next();
    System.out.println(card.getSuit());
}
```

ma se invece di chiamarlo `getCards()` lo chiamiamo `iterator()` in modo da aderire alla interfaccia `Iterable`

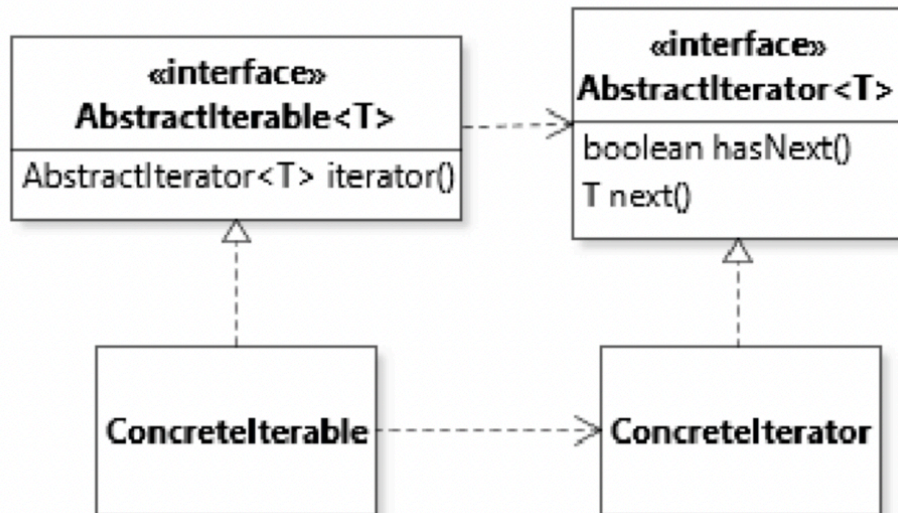
```
Public interface Iterable<>{
    Iterator<T> iterator();
    ...
}
```

allora possiamo usare il costrutto `for` esteso

```
for(Card card : deck)
    System.out.println(card.getSuit());
```

ITERATOR

ITERATOR = Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

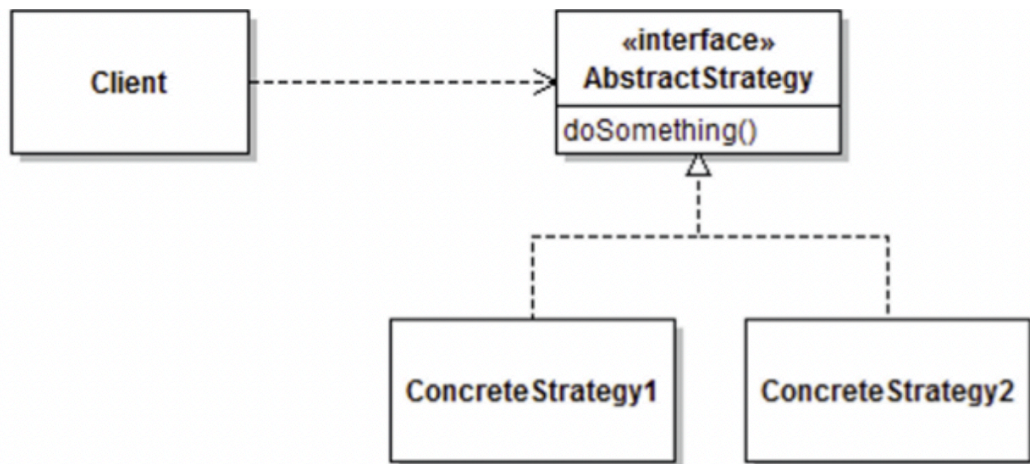


Java fornisce nella libreria in `java.util` le definizioni delle interfacce. :

- `AbstractIterable` → `Iterable` = è una interfaccia e stabilisce che una classe ha un metodo `iterator` che ritorna un iteratore sugli elementi della classe stessa.
- `AbstractIterator` → `Iterator`

STRATEGY

STRATEGY = Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithms are independently from clients that use it.



//Lo abbiamo visto con `sort()` e `Comparator`.

È uno dei pattern più semplici e utili. Consente di isolare un algoritmo al di fuori di una classe, per far sì che quest'ultima possa variare dinamicamente il suo comportamento, rendendo così gli algoritmi intercambiabili a runtime.

Prendendo in esame una famiglia di algoritmi (es. algoritmi di ordinamento, di crittografia, di validazione, etc) grazie allo STRATEGY pattern è possibile utilizzare una qualsiasi implementazione (che viene genericamente chiamata Strategy o Strategia), scegliendo fra quelle disponibili, che si rende più opportuna in un determinato contesto, in quanto tutte le implementazioni facenti parte della stessa famiglia hanno interfaccia comune.

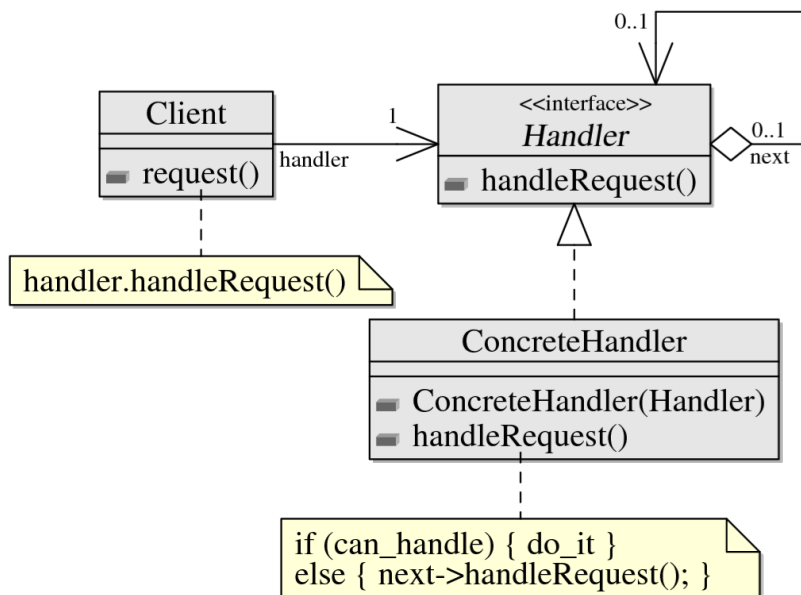
Questo design pattern consiste nell'**incapsulare un algoritmo all'interno di una classe**, mantenendo un'interfaccia generica.

Più nel dettaglio:

- **Strategy**, dichiara l'interfaccia della classe di algoritmi, interfaccia che viene utilizzata da **Client** per invocare un algoritmo concreto.
- **ConcreteStrategy1** e **ConcreteStrategy2**, sono delle classi al cui interno hanno uno specifico algoritmo, implementano l'interfaccia **Strategy**.
- **Client**, classe che invoca/utilizza la **ConcreteStrategy**.

CHAIN RESPONSIBILITY

CHAIN RESPONSIBILITY = is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.



In questo schema si può osservare:

- **Client**, colui che interagisce con il pattern. Deve essere capace di rispondere a una certa richiesta, infatti ha il metodo `request()` (es.: dire la composizione della mano di carte, ovvero se è poker, tris, full, etc.). Normalmente risolviamo queste richieste con un metodo **atomico**, dentro il metodo andiamo a svolgere compiti seguendo un certo flusso condizionato (tramite costrutti `if` o `switch` si controllano alcune condizioni, si hanno tante diramazione). Tutti questi compiti sono ben distinti, si fa un controllo su cose diverse (colore, rank, suit). Si può **separare la responsabilità**, creando una **catena di comando**, infatti l'ordine in cui presento questa sequenza di operazioni è importante. Questo permette di poter modificare questa catena con più facilità e allo stesso tempo far sì che chi chiama questo servizio, il **Client**, sia a conoscenza di questa modifica.
- **Handler**, il gestore è una interfaccia, attributo di **Client** reso dalla freccia di **associazione** da **Client** verso **Handler**. C'è anche un puntatore che punta a se stesso, ovvero un handler può avere sia un successore che un predecessore.
- **ConcreteHandler**, è un gestore concreto che implementa l'interfaccia. Quando si vuole aggiungere un nuovo modo di filtrare le carte, si crea una nuova classe e la si concatena.

Interface Segregation Principle

Interface Segregation Principle = Client code should not be forced to depend on interfaces it does not need.

Perché mai un utilizzatore (client) di un componente dovrebbe dipendere da qualcosa che non usa?

Il problema è che l'interfaccia di un componente (cioè l'insieme dei metodi pubblici che un utilizzatore può sfruttare) potrebbe essere molto più ampia di ciò che ha davvero bisogno il client.

In particolare, poi, potrebbe succedere che nell'interfaccia ci sono molti metodi, diciamo 10, di cui 6 servono a certi tipi di client e gli altri 4 ad altri tipi di client. In questo caso il principio non sarebbe rispettato: il client oltre a dipendere dai 6 che usa, **dipende** anche dagli altri 4, che non usa (sono pensati per una "clientela" differente..), ma che fanno parte della stessa interfaccia. In altre parole, se cambio uno dei 4 non utilizzate dal client che usa gli altri 6, il cambiamento influisce comunque sul client, perché la dipendenza sintattica è con tutti e 10.

I clienti non dovrebbero essere costretti a dipendere da interfacce che non usano. Questo principio nasce dalla necessità di evitare che una classe debba incorporare metodi non necessari al proprio funzionamento, come diretta conseguenza di un fenomeno che Robert C. Martin denomina **Interface Pollution**, inquinamento da interfacce.

In sostanza? Bisogna **separare le interfacce in gruppi coerenti**. É un passo aggiuntivo rispetto alla Dependency Inversion.

Object State

Le viste possono essere statiche o dinamiche. Sono due aspetti complementari da considerare quando progettiamo un sistema.

In UML abbiamo diversi diagrammi:

- Classi (statico)
- Oggetti (dinamico)
- Stati (dinamico)
- Sequenze (dinamico)
- etc...

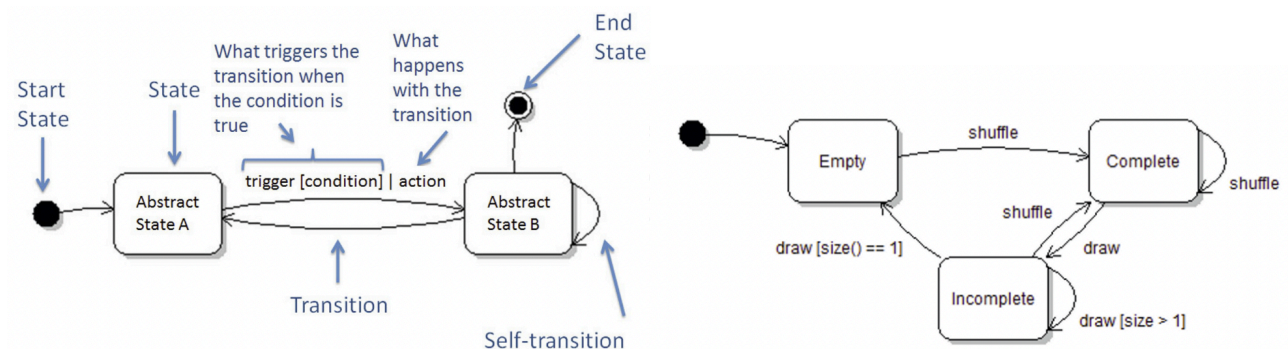
Lo stato di un oggetto può essere:

- **concentrato** = può essere visto come la combinazione di tutti i valori dei suoi attributi. Dipende dalla sua implementazione. La cardinalità dell'insieme degli stati possibili (state space) "esplode" molto facilmente. Ad esempio in un mazzo, tutte le permutazioni di qualunque numero delle 52 carte, circa $2 \cdot 10^{68}$.
- **astratto** = è un sottoinsieme arbitrario degli stati concreti, meglio se significativo.

Ci sono anche dei **casi particolari** come:

- **oggetti senza stato** = stateless object, come ad esempio gli oggetti funzione
- **oggetti che hanno un solo stato** = oggetti immutabili. E allora perché abbiamo detto che String (che sappiamo essere immutabile) ha un numero di stati quasi infinito? Una classe immutabile non ha un solo stato, un oggetto una volta che lo sceglie non può più cambiarlo.

Diagramma degli stati



Un principio legato a **Separation of Concerns** e Singola Responsabilità è quello di **minimizzare lo spazio degli stati degli oggetti** in qualche modo.

Aiuta anche a evitare di perdere il controllo e generare combinazioni di stati invalidi o inutili. Il concetto di mazzo di carte non mischiato è utile? o posso chiamare shuffle direttamente nell'init?

antipattern code smells:

- SPECULATIVE GENERALITY†
- TEMPORARY FIELD†

Nullability = ad una variabile che indica un riferimento a un oggetto (quindi in Java sempre a parte i tipi primitivi) possiamo assegnare il valore speciale `null`, per dire che non punta a nulla.

```
Card card = null; //oppure Card.rank = null;
```

Il problema si manifesta quando proviamo a differenziare la variabile. Non puntando a nulla si riceve un errore, una `NullPointerException`.

Il problema è che `null` potrebbe significare (a livello più astratto) diverse cose:

- errore
- stato temporaneamente inconsistente
- valore assente

Un codice chiaro non dovrebbe far uso di `null`.

NULLOBJECT

NULLOBJECT = is a design pattern that simplifies the use of dependencies that can be undefined. This is achieved by using instances of a concrete class that implements a known interface, instead of null references.

Vogliamo creare un oggetto che corrisponda al concetto di "nessun valore". Si cerca di risolvere il problema del valore nullo dal punto di vista dell'object orientation.

Il problema del `null` è che solleverà un `NullPointerException` nel caso in cui cerchi di usare quell'oggetto. Bisogna ogni volta chiedersi se un oggetto è `null`. Se lo è si devono usare un certo tipo di operazioni e viceversa.

Il **polimorfismo** è il meccanismo con cui andiamo ad associare un concetto generico a diverse realizzazioni. Attraverso il **collegamento dinamico** (o extensibility) possiamo chiamare la realizzazione corretta.

Il collegamento dinamico/extensibility permette di chiamare codice non ancora scritto perché:

1. posso scrivere del codice che chiama funzioni citandole come definite su una interfaccia,
2. scrivere classi che implementano tale interfaccia in momenti successivi.

Quindi ho del codice (1) che compila e funziona a un certo punto, e che senza modificarlo, è capace di chiamare nuovo codice quando aggiungo le classi (2).

Il compilatore sa i metodi dell'interfaccia che verranno chiamati, ma potrebbe non sapere quali oggetti (e quindi quali classi che implementano tale interfaccia) saranno in effetti puntati.

Il **polimorfismo** è quello che molto spesso viene in aiuto in questi problemi di scelta, si promuove questa assenza di valore ad oggetto. Si crea un oggetto che incarna il valore nullo di quella classe, e per quell'oggetto vado, mediante l'operazione di overriding, ad esplicitare come si devono comportare i vari metodi.

```
public interface CardSource {  
  
    public static CardSource NULL = new CardSource() {  
        public boolean isEmpty() { return true; }  
        public Card draw() {  
            assert !isEmpty();  
            return null;  
        }  
        public boolean isNull() { return true; }  
    };  
  
    Card draw();  
    boolean isEmpty();  
    default boolean isNull() { return false; }  
}
```

`CardSource.NULL` è un oggetto valido di un tipo anonimo che aderisce alla interfaccia `CardSource` ma che ha particolari implementazioni per i vari metodi.

optional types = Optional is primarily intended for use as a method return type where there is a clear need to represent "no result," and where using null is likely to cause errors. A variable whose type is Optional should never itself be null; it should always point to an Optional instance.

Il tipo opzionale è un nuovo tipo in cui si cerca di eliminare l'uso del `null` come comunicazione dell'errore. Infatti se:

- c'è un valore viene comunicato il valore dell'optional type
- non c'è il valore lo si sa già, senza bisogno di test o assegnando un particolare valore del dominio a "valore non valido", è una variabile booleana separata

Per renderlo il più usabile possibile è definito come un tipo generico, ovvero parametrico rispetto a un altro tipo.

Al posto del costruttore, siccome parte da un interfaccia, definisce 3 metodi statici:

- `static Optional empty()` Returns an empty Optional instance.
- `static Optional of(T value)` Returns an Optional describing the given non-null value.
- `static Optional ofNullable(T value)` Returns an Optional describing the given value, if non-null, otherwise returns an empty Optional.

Chi prende questo oggetto optional può chiedere (ovvero l'interfaccia poi fornisce i metodi):

- `T get()`
- `boolean isEmpty()`
- `boolean isPresent()`
- `T orElse(T)`
- `void ifPresent(Consumer)`
- etc...

Prima $\Rightarrow$ si chiede di prendere una stringa da un oggetto, la si assegna a `value` e prima di poterne stampare la lunghezza bisogna controllare se `value` è diverso da `null`.

```
String value = someObject.getSomeString();  
if (value != null) System.out.println(value.length());
```

Dopo $\Rightarrow$ con `ifPresent()`

```
Optional<String> value = someObject.getSomeString(); //diverso valore ritornato  
value.ifPresent( valore -> System.out.println(valore.length()) );
```

attributo final = può essere assegnato solo una volta e dentro il costruttore (o come parte della dichiarazione). Si ha un errore di compilazione se si cerca di fare un secondo assegnamento.

Sono perciò delle costanti che non possono cambiare valore?

- Se è un tipo primitivo sì, diventa una costante perché sono dei **tipi immutabili** e non un riferimento a un valore.
- Nel caso in cui sia un riferimento a un oggetto non si può puntare a un altro oggetto. Quello che diventa costante è il **riferimento all'oggetto** non il valore/stato di quell'oggetto (es.: `final list` è garantito che sia immutabile? no, l'oggetto `list` non è immutabile, non è sufficiente che sia scritto prima `final`, devono essere immutabili anche tutti i suoi elementi, non deve essere possibile aggiungere elementi alla lista. Ciò che è reso immutabile è a quale lista si fa riferimento).

```
public class Card {  
    final private Rank rank;  
    final private Suit suit;  
    public Card(Rank r, Suit s) {  
        rank = r;  
    }  
}
```

```

    suit = s;
  }
}

```

Identity, Equality, Uniqueness → sono concetti legati tra di loro

- **identità** = **sono lo stesso oggetto**, la stessa zona di memoria. Viene valutata nel caso degli oggetti tramite l'operatore `==`.
- **uguaglianza** = **rappresentano uno stesso valore**. Dovrebbe essere valutata tramite il metodo `equals(Object)`, che però di default la rimappa su identità.
- **unicità** = **quando identità e uguaglianza coincidono**. Cioè non possono esistere due oggetti distinti che sono considerati uguali.

La unicità, oltre ad essere richiesta in certi casi, porta comunque diversi vantaggi sia di prestazioni (minore occupazione di memoria), sia di chiarezza e leggibilità. Due pattern sono legati al concetto di unicità: SINGLETON e FLYWEIGHT.

La uguaglianza è perciò più "astratta" e per questo deve essere definita da programmatore (che potrebbe decidere che due oggetti sono uguali anche se non hanno tutti gli attributi uguali) ma è sufficientemente "standard" da permetterne una costruzione semi automatica:

```

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Card card = (Card) o;
    return rank == card.rank && suit == card.suit;
}

@Override
public int hashCode() {
    return Objects.hash(rank, suit);
}

```

Se garantiamo unicità, non serve definire equals perché equivalente a `==`

SINGLETON

SINGLETON = is a design pattern that restricts the instantiation of a class to one "single" instance. This is useful when exactly one object is needed to coordinate actions across the system.

Ha lo scopo di assicurarsi che una classe abbia solo un'istanza e che la classe fornisca un punto di accesso globale a essa.

Con il termine SINGLETON si indica quindi una modalità di dichiarazione delle classi che ha lo scopo di **garantire** la creazione di **una sola istanza della classe**. La creazione e l'accesso a questa istanza è gestito direttamente dalla classe dichiarata in modalità SINGLETON.

Gli elementi che caratterizzano un SINGLETON sono:

- il **costruttore privato**, per evitare la creazione di oggetti da classi esterne;
- un **metodo statico**, per accedere all'unica istanza dell'oggetto;
- un **attributo statico**, dove memorizzare l'unica istanza.

Si può dire che va oltre il concetto di classe: non ha identità e stato.

```
public class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == null) instance = new Singleton();
        return instance;
    };
    public void sampleOp() {...};
}
```

Questa soluzione è ottima finché il programma non è **multithread**. Nel caso invece in cui lo sia si ha il rischio che due thread arrivino contemporaneamente al `getInstance()`, entrando così in **race condition** per la prima esecuzione di quel metodo. La JVM potrebbe però togliere il controllo a un thread e passarlo ad altri, i quali una volta terminato, ridaranno il controllo al thread principale da dove era rimasto e ci ritroviamo così due istanze del singleton. Questo problema è risolvibile scrivendo la firma del metodo come `public synchronized static Singleton getInstance()`. Il problema viene risolto ma risulta inefficiente a livello computazionale. Tuttavia questo problema lo si ha solo all'inizio quindi si può trasformare nuovamente l'implementazione come:

```
java public class Singleton {
    protected Singleton() {}
    private static Singleton instance = null;
    public static Singleton getInstance(){
        if (instance == null){
            synchronized(Singleton.class){
                if (instance == null)
                    instance = new Singleton();
            }
        }
        return instance;
    }
}
```

Stiamo applicando in questo caso il pattern di concorrenza double check. Crediamo di aver risolto così ma in realtà con le versioni di java precedenti alla 5 si verificava un bug (veniva settata la memoria, veniva assegnata all'identificatore e poi invocato il costruttore, comportando così un return instance nullo).

Quindi la soluzione finale, per quanto riguarda Java, è quella di sfruttare il tipo **enumerativo**, particolarità di Java stesso. Per questo motivo, il SINGLETON implementato in questo modo non è

più un pattern ma un idiomma (modo che rende molto facile la creazione di un singleton). Nello specifico si tratta di uno uso improprio del costrutto enum, tuttavia funziona. Infatti i campi degli enum sono costanti calcolate a tempo di compilazione, che corrispondono a istanze della classe, ma `final static`, e la prima volta che mi riferisco all'enumerativo viene invocato il costruttore privato e finale dell'oggetto `instance`,

```
public enum Singleton {  
    INSTANCE;  
    public void sampleOp() {...};  
}
```

Alla fine il singleton è una classe statica, allora perché si usa? È più flessibile: è possibile sfruttare il dynamic binding, si può ereditare il costruttore se `protected`, si può specializzare la classe e fissare anche un numero di istanze consentite.

- invece di `getInstance` riferiamo direttamente l'unico valore possibile `INSTANCE`
- è un approccio implicitamente Lazy, l'oggetto statico viene creato la prima volta che lo cito, finché non lo cito non occupo spazio.
- thread safe

FLYWEIGHT

FLYWEIGHT = refers to an object that minimizes memory usage by sharing some of its data with other similar objects.

È un pattern che serve a gestire una collezione di oggetti immutabili assicurandone l'unicità.

Quando le istanze equivalenti sono fortemente condivise all'interno del programma diventano auspicabili sia immutabilità che unicità. Ad esempio le nostre Card.

è simile al SINGLETON infatti:

- il costruttore è privato
- viene fornito un metodo statico per creare i vari oggetti i numero fissato

ma a differenza del SINGLETON devo contenere **più istanze** e spesso non è definibile a priori quante sono (non viene restituito sempre lo stesso oggetto e possono citarne diversi per qualche caratteristica).

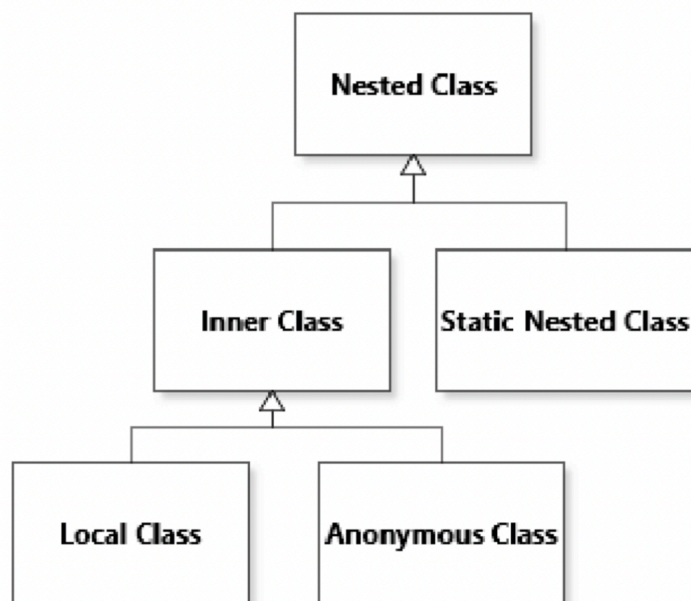
```
public class Card{
    private static final Card[][] CARDS =
        new Card[Suit.value().length][Rank.value().length];
    static{
        for (Suit suit : Suit.value()){
            for (Rank rank : Rank.value()){
                CARDS[suit.ordinal()][rank.ordinal()] = new Card(rank,suit);
            }
        }
    }
    public static Card get(Rank pRank, Suit pSuit){
        assert pRank != null && pSuit != null;
        return CARDS[pSuit.ordinal()][pRank.ordinal()];
    }
}
```

Si tratta di un pattern strutturale basato su oggetti che viene utilizzato per ottimizzare l'utilizzo delle risorse ed evitare la presenza di oggetti duplicati, spesso viene associato a richieste di performance in quanto il suo utilizzo può migliorare le prestazioni di una applicazione. In particolare in Java, ogni qual volta che viene utilizzato l'operatore new, la JVM alloca nell'heap uno spazio in memoria di 32 bit ed una eccessiva generazione di oggetti può saturare le risorse del sistema che, anche se elevate, sono pur sempre limitate. Molto spesso la generazione di nuovi oggetti non è motivata da reali esigenze mentre invece è dovuta a superficialità oppure a errata analisi delle prestazioni. Da qui l'esigenza di riutilizzare gli oggetti precedentemente creati ai fini del loro riutilizzo.

L'esigenza di adottare questo pattern può nascere da vari fattori:

- evitare la generazione di un elevato numero di oggetti
- evitare di creare oggetti che richiedono molta memoria
- evitare di creare oggetti che richiedono molto impegno computazionale
- Disponibilità limitate di risorse: di calcolo, di memoria, di spazio ecc

Nested Classes



static nested class = sono la versione più semplice delle classi innestate. Un attributo privato è visibile in tutte quelle parti di codice che sono contenute dentro le parentesi graffe della definizione della classe.

```
class A{
    private int secret;
    public int method(){
        return secret;
    }
    public static class B{
        private int innerSecret;

        int sumSecrets(){
            A ext = new A();
            return innerSecret + e.secret;
        }
    }
}
```

```
public class Main{
    public static void main(String[] args){
        A external = new A();
        A.B internal = new A.B();
    }
}
```

- Siccome B è pubblico il main riesce a vederlo e posso creare un oggetto di tipo B citando esplicitamente A.B.
- Se B vuole accedere a un campo privato non statico di A bisogna creare un oggetto di tipo A. (La classe B non può vedere i segreti della classe A perché B è una classe statica e quindi non collegata a nessuna istanza e `private int secret` è un attributo di istanza. Se invece fosse stato `private static int secret` avrei potuto citarlo dentro B).
- B può accedere ai segreti della classe A e viceversa, i campi privati della classe interna B sono visibili alla classe esterna A.

inner class = Differisce dalla static nested class in quanto la classe interna B non è statica. È obbligatorio che gli oggetti definiti come classi innestate non statiche devono essere create all'interno della classe contenitrice (non posso creare un oggetto nel main ma in A, solo gli oggetti di tipo A possono creare oggetti di tipo B).
Le inner class mantengono un riferimento all'oggetto contenente e possono essere create solo all'interno di un metodo di istanza della classe contenitore perché altrimenti non saprei qual è il riferimento a cui si aggancia automaticamente.

```
class A{
    private int secret;
    public int getSecret(){
        B i = new B();
        return i.innerSecret;
    }
    public class B{
        private int innerSecret;
        int sumSecrets(){
            return this.innerSecret + A.this.secret;
        }
    }
}
```

anonymous inner classes = è un particolare tipo di inner class. All'interno della classe A viene implementato un comparator. Il metodo compare serve per definire delle istanze della classe. Le classi anonime non sono statiche.

```
class A{
    private int secret;
    private Comparator<A> cmp = new Comparator<A>(){
        @Override
        public int compare(A o1, A o2){
            return o1.Secret-o2.secret;
        }
    };
}
```

```
class A{
    private int secret;
    private Comparator<A> cmp = (o1, o2) -> o1.Secret-o2.secret;
}
```

local inner classes = è un particolare tipo di inner class. La classe, invece di essere un attributo, diventa come una variabile locale.

```
class A{
    private int secret;
    public int getSecret(){
        class I{
            private int innerSecret;
        }
        I i = new I();
        return i.innerSecret;
    }
}
```

Dentro la classe I si possono citare variabili e parametri che sono definiti nella funzione getSecret solo se le variabili sono final (viene assegnato un valore e poi non viene più cambiato).

anonymous inner classes static

```
public class Deck{
    public static Comparator<Deck> createByRankComparator(Rank pRank){
        return new Comparator<Deck>(){
            public int compare(Deck pDeck1, Deck pDeck2){
                return contCards (pDeck1, pRank) - contCards (pDeck2, pRank);
            }
            private int countCards(Deck pDeck, Rank pRank)
            { /*returns the number of cards in pDeck with pRank*/ }
        };
    }
}
```

Unit Testing

*"Program testing can be used to show the **presence** of bugs, but never to show their **absence**!"*
- Edsger W. Dijkstra

Ci possono essere 3 livelli di analisi di un errore:

- Malfunzionamento (**failure**), è la manifestazione esterna che qualcosa non funziona rispetto alle attese.
- Difetto (**fault**), il motivo per cui si può presentare un malfunzionamento.
- Sbaglio (**mistake**), il motivo dell'errore, come un errore di battitura, mancanza di conoscenze, incomprensioni, etc..

Il testing può essere:

- **Unit** = si va a testare la **singola unità di sviluppo**.
- **Functional** = non guarda il codice della singola funzionalità ma partendo dalle specifiche. Si fa un test sull'**intero programma**.

La struttura di un test si può suddividere in tre momenti:

1. **setup**, a volte non presente/necessario
2. **exercise**, lanciare l'operazione che si sta testando
3. **verify**, verifica per controllare se l'esecuzione di quello che è stato esercitato ha prodotto ciò che ci si aspettava

Di seguito un esempio su pokerHand. Non è presente un setup. Si vuole controllare il costruttore che parte da una stringa.

```
public class PokerHandTest {
    @Test
    public void controllaCostruttoreDaStringa() {

        //EXERCISE
        PokerHand mia = new PokerHand("JC 3H AC 4C 8H");

        //VERIFY
        List<Card> cards = new ArrayList<>();
        for (Card card : mia) {
            cards.add(card);
        }
        assertEquals(5, cards.size());
        assertTrue(cards.contains(Card.get(Rank.JACK, Suit.CLUBS)));
        assertTrue(cards.contains(Card.get(Rank.THREE, Suit.HEARTS)));
        assertTrue(cards.contains(Card.get(Rank.ACE, Suit.CLUBS)));
        assertTrue(cards.contains(Card.get(Rank.FOUR, Suit.CLUBS)));
        assertTrue(cards.contains(Card.get(Rank.EIGHT, Suit.HEARTS)));
    }
}
```

Si assume che la implementazione di `Iterable` sia corretta. Si può mettere tutto questo in un `main`, però pensare di fare un `main` per ognuno dei test che ci interessa diventa molto scomodo. Più comunemente si definisce una classe che sfrutta un framework di JUnit (Java Unit Test) che permette di marcare con la keyword/annotazione `@Test` un metodo.

Un unit test di qualità deve essere:

- **Veloce**
- **Ripetibile**, deve essere chiaro e determinato il risultato che ci si aspetta.
- **Indipendente**, quando si fa un test su una parte del codice, il test non deve fallire per colpa del codice ma a causa di altri componenti usati.
- **Centrato**, nel test ci deve essere solo la parte necessaria per identificare un singolo problema.

- **Leggibile**, il codice che fa il controllo deve essere autoesplicativo. Le tre fasi del test devono essere chiare e non ci devono essere if-else o cicli (dato che si sta andando a eseguire una particolare situazione di cui si sa già l'input e l'output).

Creare test è un compito diverso rispetto a scrivere codice, infatti nelle aziende più importanti ci sono due figure distinte che si occupano dei test, i test engineer, e del software, i software engineer.

Eccezioni = Non dobbiamo controllare che le assert sollevino eccezione alla violazione del contratto. Il modo corretto per controllare se una eccezione è sollevata da un metodo è con un try/catch.

```
try {
    methodThatShouldThrow();
    fail( "My method didn't throw when I expected it to" );
} catch (MyException expectedException) {}
```

Il problema dell'encapsulation è: come facciamo se lo stato su cui dobbiamo fare asserzioni è privato e non accessibile?

- controlliamo che non ci sia un metodo da cui controllarlo indirettamente (ad esempio una toString).
- aggiungiamo un metodo per avere accesso all'informazione.
- creiamo dentro al test un metodo (helper) in grado di ricavare tale informazione o una sua derivata.
- usiamo la reflection (o introspezione in italiano), è la possibilità di scrivere un programma che ragiona sul programma stesso. Si interroga la classe, manipolando il codice della classe.

Qua sotto si può osservare un esempio di introspezione del codice.

```
public class TestFoundationPile {
    private FoundationPile aPile;
    @Before
    public void setUp() { aPile = new FoundationPile(); }
    private Optional<Card> getPreviousCard(Card pCard) {
        try {
            Method method = FoundationPile.class.
                getDeclaredMethod("getPreviousCard", Card.class);
            method.setAccessible(true);
            return (Optional<Card>) method.invoke(aPile, pCard);
        }
        catch( ReflectiveOperationException exception ) {
            fail();
            return Optional.empty();
        }
    }
    @Test public void testGetPreviousCard_empty() {
        assertFalse(getPreviousCard(Card.get(Rank.ACE, Suit.CLUBS)).isPresent());
    }
}
```

L'attività di testing è **ottimistica**, l'assenza di errori nei test non mi assicura la correttezza del codice ma la non presenza di errori. Quando è sufficiente? Quando mi fermo? Quando ritengo di aver fatto abbastanza attività di testing?

Gli unici test che assicurano la correttezza del codice sono i **test esaustivi**, quello in cui si provano tutte le situazioni possibili.

Data questo semplice codice si vuole fare un test esaustivo.

```
class Trivial {  
    static int sum(int a, int b)  
    { return a + b; }  
}
```

In Java un int è espresso su 32 bit. Il dominio è quindi di cardinalità è

$$2^{32} \cdot 2^{32} = 2^{64} \approx 2 \cdot 10^{19}$$

Considerando un tempo di 1 ns per ogni test. Il tempo totale per testare ogni somma è di

$$2 \cdot 10^{10}$$

secondi che corrisponde a più di 600 anni.

Testing coverage = Quando ritengo di aver fatto abbastanza attività di testing? Non è utile stabilire un numero di test. Bisogna chiedersi “quale è la **prima condizione necessaria** affinché un test possa evidenziare un malfunzionamento dovuto a un difetto?”

... che la riga di codice che contiene il difetto venga eseguita dal test. Bisogna eseguire la riga di codice per evidenziare il difetto.

Questo ci fa capire che ci sono alcune richieste che possiamo fare a un test per poterlo giudicare.

Ad esempio un test che non copre **tutti i comandi** è un test che possiamo giudicare insufficiente.

Ci sono moltissimi **criteri di copertura** definiti in letteratura:

- strutturali (che identificano una qualche caratteristica del codice)
- funzionali (che identificano una qualche caratteristica delle specifiche)

//IDEA rende semplice almeno il controllo della copertura dei comandi

Stubs = o **Test Double** è un termine generico per un qualunque tipo di oggetto che **viene messo al posto di quello reale** al fine di permettere una attività di testing. Questo perché:

- non esiste ancora o non è facilmente accessibile
- l'oggetto fornirebbe dei dati non deterministici/non prevedibili: il tempo corrente, la temperatura corrente
- può presentare delle situazioni difficilmente riproducibili (errori di trasmissione, esaurimento della memoria)
- è lento, si costruisce un fantoccio del componente vero e ottengo rapidamente e in maniera deterministica un risultato per il test

In java si usa `Mockito.mock()` = allows us to create a mock object of a class or an interface.

Associazione, aggregazione e composizione

Composizione = comporre diverse astrazioni.

//Lavorare per composizione di astrazioni può essere fatto arbitrariamente male.

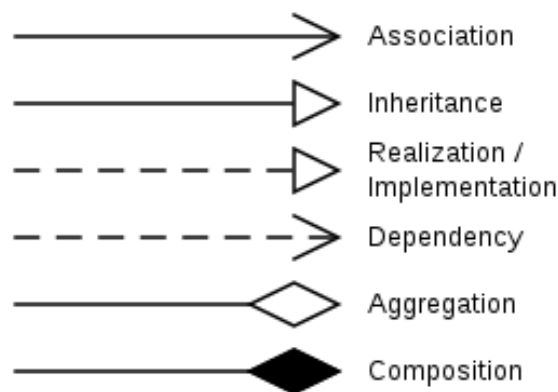
Robillard distingue tra composizione con scopo di:

- **rappresentazione** (composizione per eccellenza) = quando si compongono oggetti che ci identificano (la definizione di stato è composta da vari elementi, un mazzo è una composizione di carte).
- **delega** (che chiama aggregazione) = quando ci si scompone in parti a cui si delegano dei compiti (attributo che dice a chi devo delegare il compito di valutazione di una mano di poker).

ma non sono sempre in mutua esclusione tra loro.

Robillard stesso ammette che questa terminologia si scontra con quella più standard di UML che abbiamo già anticipato nel diagramma delle classi:

- **associazione** = la classe A è associata alla classe B quando la classe A usa elementi della classe B. La classe A conosce istanze della classe B su cui può invocare dei metodi o accedere a degli attributi. La classe A ha dei riferimenti alla classe B a cui è associata (sono scollegato ma interagisco, come professore e studenti). In UML l'associazione viene rappresentata come una linea semplice.
- **aggregazione** = assomiglia alla composizione con scopo di rappresentazione. La classe A aggrega elementi di altre classi e che fanno parte della definizione della classe stessa, contribuiscono a definire lo stato di A (una sessione d'esame e studenti iscritti a quell'esame, un esame è definito dagli studenti che vogliono partecipare a quell'esame, oltre che dall'aula, la data, il tema d'esame, etc.). In UML l'aggregazione viene rappresentata come una linea con rombo bianco.
- **composizione** = un contenuto può appartenere a un solo oggetto contenente. In UML la composizione viene rappresentata come una linea con rombo nero.



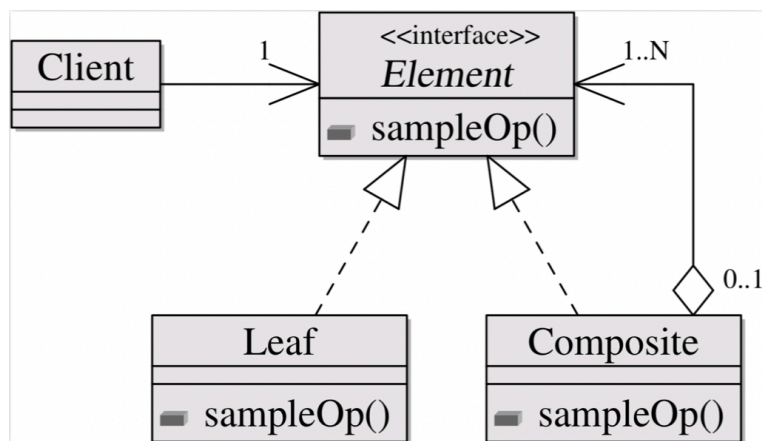
COMPOSITE

COMPOSITE = is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

È un pattern che ha il compito di gestire dei gruppi di oggetti, facendoli interagire come se fossero dei singoli oggetti. Il cliente non si deve accorgere che sta lavorando con un gruppo o con un singolo oggetto.

Questo pattern permette di trattare un gruppo di oggetti come se fossero l'istanza di un oggetto singolo. Il design pattern COMPOSITE organizza gli oggetti in una **struttura ad albero**, nella quale i **nodì sono delle composite** e le **foglie sono oggetti semplici**.

È utilizzato per dare la possibilità ai clienti di manipolare oggetti singoli e composizioni in modo uniforme.



- Client, manipola gli oggetti attraverso l'interfaccia Element.
- Element, dichiara l'interfaccia per gli oggetti nella composizione, per l'accesso e la manipolazione di questi, imposta un comportamento di default per l'interfaccia comune a tutte le classi e può definire un'interfaccia per l'accesso al padre del componente e la implementa se è appropriata.
- Composite, definisce il comportamento per i componenti aventi figli, salva i figli e implementa le operazioni ad essi connesse nell'interfaccia Element.
- Leaf, definisce il comportamento degli oggetti primitivi, cioè delle foglie.

Element è ciò che il cliente conosce di questa struttura. Può corrispondere sia a delle foglie (degli elementi finali) sia a degli elementi composti, che sono l'aggregazione da 1 a N elementi. Un componente può appartenere a 0 o 1 oggetto composto.

Essendo Element a sua volta un oggetto composto può essere visto come un'aggregazione di aggregazioni.

Quando i programmatori trattano dati strutturati ad albero devono spesso discriminare se stanno visitando un nodo o una foglia. Questa differenza è una possibile fonte di complessità per il codice e, se non trattata a dovere, rende il programma facilmente soggetto a errori. La soluzione è adottare un'interfaccia che permetta di trattare oggetti complessi e primitivi in modo uniforme. Nella programmazione orientata agli oggetti un COMPOSITE è un oggetto (per esempio una figura geometrica) progettato per composizione di uno o più oggetti simili (una figura geometrica come un trapezio può essere vista a sua volta come formata da due triangoli rettangoli e un rettangolo) che offrono tutte le medesime funzionalità. Questa caratteristica è conosciuta anche come relazione "has-a" tra gli oggetti.

Il concetto fondamentale è che il programmatore manipola ogni oggetto dell'insieme dato nello stesso modo: sia esso un "raggruppamento" o un oggetto singolo.

Le operazioni che possono essere effettuate sul COMPOSITE hanno spesso un denominatore **comune**, per esempio: se il programmatore deve visualizzare a schermo un insieme di figure

potrebbe essergli utile definirne il ridimensionamento in modo tale da avere lo stesso effetto (in senso lato) del ridimensionamento di una singola figura.

Multideck

COMPOSITE pattern applicato all'esempio delle carte. Voglio creare un oggetto composito CardSourceComp aderente a CardSource, ovvero un oggetto che al suo interno contiene diversi mazzi ma si comporta come un normale CardSource.

```
public interface CardSource {
    Card draw();
    boolean isEmpty();
}
```

All'interno ho un attributo lista di CardSource che corrisponde alla aggregazione UML vista in precedenza. Come prima modifica, per forzare il legame da 1 a N, aggiungeremo un assert che controlla se CardSource ha almeno una sorgente quando si passa una lista di CardSource. (Altra modifica opportuna sarebbe quella di mettere uno shuffle() dei mazzi a ogni draw() di carta, dato che in questo momento si prendono le carte da un deck alla volta finché non è vuoto.)

```
public class CardSourceComp implements CardSource {
    private List<CardSource> sources;

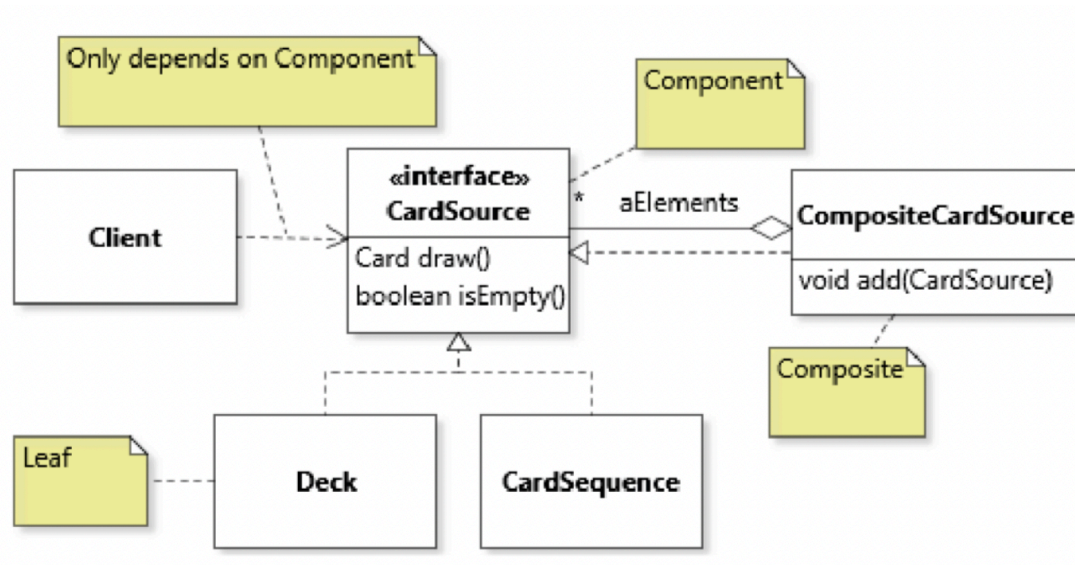
    public MultiDeck(int num){
        mazzi = new ArrayList<>(num);
        for(int i = 0; i<num; i++)
            mazzi.add(new Deck());
    }

    public boolean isEmpty() {
        for( CardSource source : sources )
            if( !source.isEmpty() ) return false;
        return true;
    }

    public Card draw() {
        assert !isEmpty();
        //Collection.shuffle(sources);
        for( CardSource source : sources )
            if( !source.isEmpty() )
                return source.draw();
        assert false;
        return null;
    }

    public CardSourceComp(List<CardSource> cardSources) {
        sources = new ArrayList<>(cardSources);
    }
}
```

Diagramma delle classi UML del COMPOSITE di CardSouce.



La differenza fondamentale tra questo schema e quello visto in precedenza è nel come si differenzia la classe composta dalle classi foglie. Nello schema precedente abbiamo visto che **Leaf** e **Composite** sono esattamente equivalenti, cambiano solo nel tipo di relazioni, ma non nel tipo di funzioni che esportano, che sono esattamente quelle fornite dalla classe **Element**.

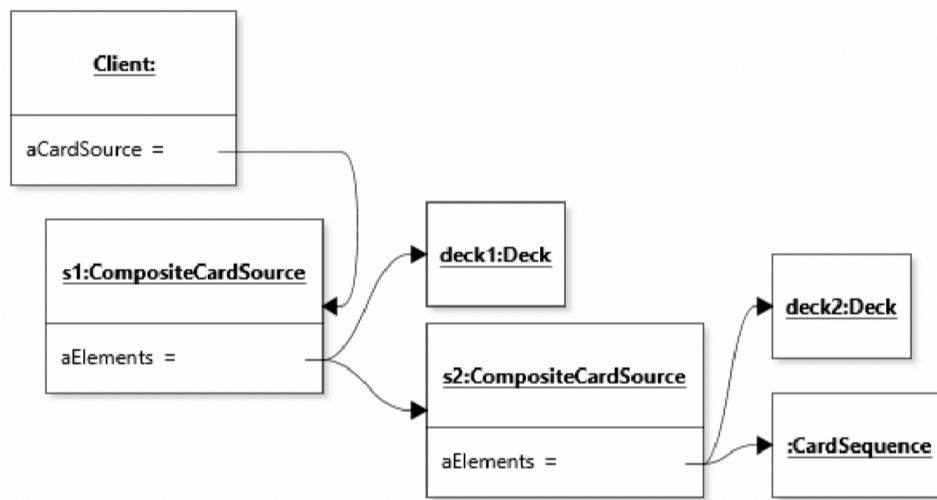
In questo caso invece abbiamo **Deck** e **CardSequence** che sono due esempi di foglie, in particolare di sorgenti di carta perché entrambi implementano l'interfaccia **CardSource**, e quindi fanno override dei due metodi `draw()` e `isEmpty()`.

Inoltre **CompositeCardSource** definisce un altro metodo. Si può notare una disomogeneità tra la parte composta e la parte foglia, è uno schema poco preciso.

Il metodo `add()` di **CompositeCardSource** serve a far vedere che ci sono più diverse versioni del pattern **COMPOSITE**, infatti bisogna capire se la visione del gruppo può essere decisa una volta per tutte quando creo l'oggetto composto oppure se gli elementi che lo compongono possono cambiare nel tempo.

Client è l'utilizzatore del pattern.

Diagramma degli oggetti di COMPOSITE di CardSource.



Questo diagramma non è totalmente coerente con quello precedente. Questo diagramma degli oggetti dice che i clienti sono caratterizzati da un attributo CardSource che punterà ad un elemento CompositeCardSource in questo caso l'oggetto composito.

Se client ha un attributo allora la relazione dello schema precedente non è di dipendenza ma di associazione.

diagramma degli oggetti = object diagram, ho una scatola non per ogni classe ma è ogni oggetto che compone in un certo istante la situazione degli oggetti del mio programma. Si può notare una struttura ad **albero**.

La radice riferita da Client è CompositeCardSource che punta a due nodi, il primo un nodo foglia, un Deck, e il secondo è un oggetto composito.

CompositeCardSource può essere un:

- Deck
- CardSequence
- CompositeCardSource

Infatti queste tre classi implementano l'interfaccia CardSource.

Il secondo CompositeCardSource è composto da due foglie. Si ha un totale di 3 mazzi, uno di primo livello è due di secondo livello.

Il pattern COMPOSITE è un pattern di tipo strutturale, cioè in cui viene riconosciuto che grossa parte del problema da risolvere è il legame statico tra i vari oggetti. Questo pattern però può portare a **problemi di incomprensione** dovuti alla ricorsione di aggregazione di oggetti che possono essere anche uguali fra loro, in questo caso CompositeCardSource può aggregare CompositeCardSource.

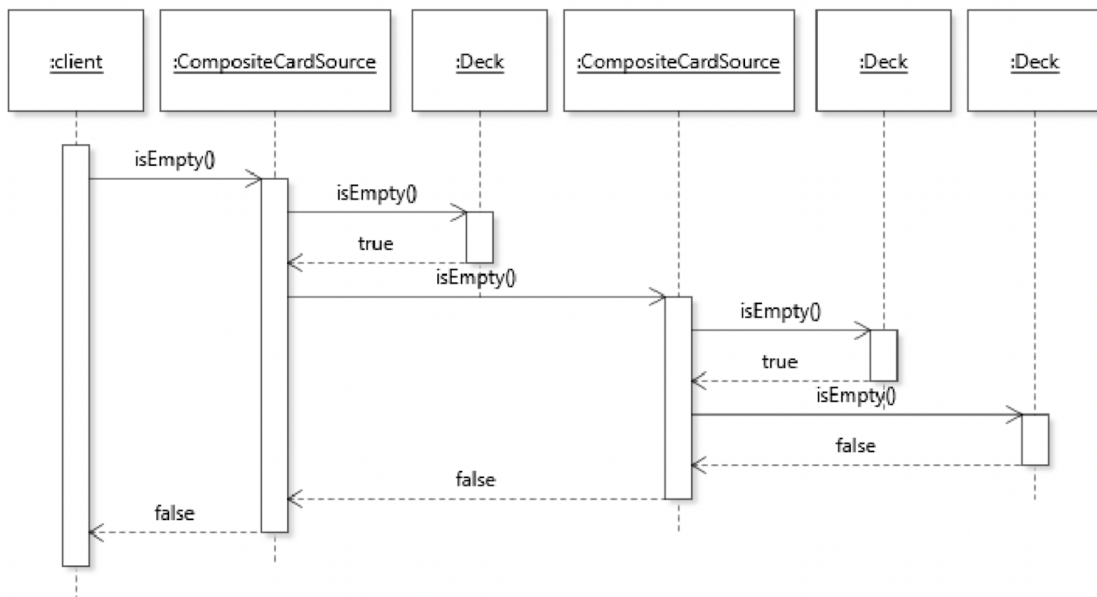
Un'altro problema è capire che cosa sta succedendo, come si svolge un certo compito. Ad esempio quando Client chiama isEmpty() su CompositeCardSource cosa accade? Non abbiamo una rappresentazione chiara del flusso dei messaggi, dei comandi, delle invocazione e di metodi che vanno a susseguirsi. Esiste in UML un'altra tipologia di diagrammi chiamati diagrammi di sequenza.

diagrammi di sequenza = sequence diagram, mostra la sequenza con cui vengono viste ed eseguite alcune operazioni, alcuni scambi di messaggi, tra vari oggetti.

Identifica sulla prima linea in alto del diagramma quali sono gli oggetti su cui vuole lavorare.

Di solito viene dato sia il nome che la classe con il formato :NomeClasse.

Potrebbe anche esistere una colonna in cui non si ha un oggetto quando, ad esempio, di quella classe si invocano solo metodi statici e quindi non c'è bisogno che esista un oggetto di quella classe per poterla usare.

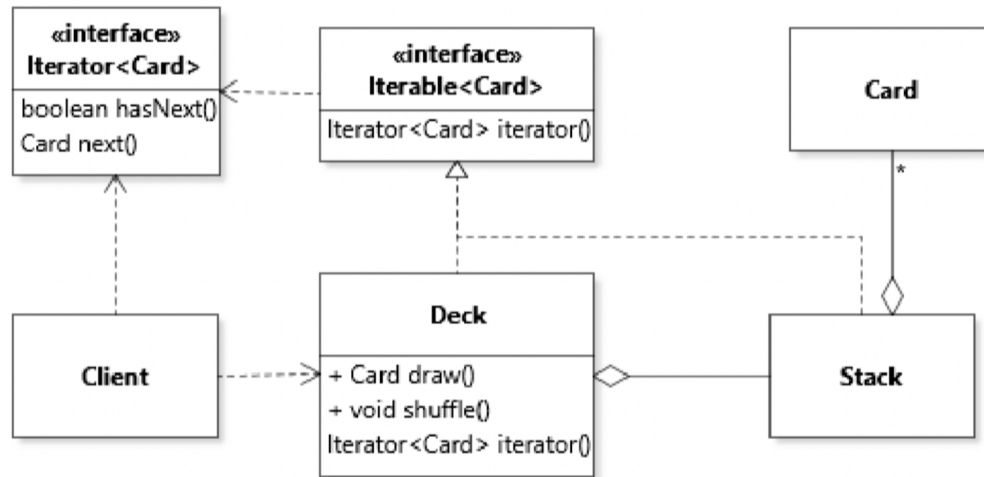


In questo caso abbiamo Client che manda un messaggio `isEmpty()` a un oggetto di tipo `CompositeCardSource`. Dallo schema degli oggetti precedente possiamo notare infatti che `CompositeCardSource` è l'unico oggetto che Client conosce e con cui può dialogare. Per rispondere al messaggio `isEmpty()` `CompositeCardSource` va a chiedere ai suoi componenti se a loro volta sono vuoti. Finché la risposta è `true` si chiede se l'elemento successivo è vuoto.

Qui non abbiamo dato una definizione generica di quello che è il funzionamento della funzione `isEmpty()`, ma in una specifica situazione.

ITERATOR deck e diagrammi

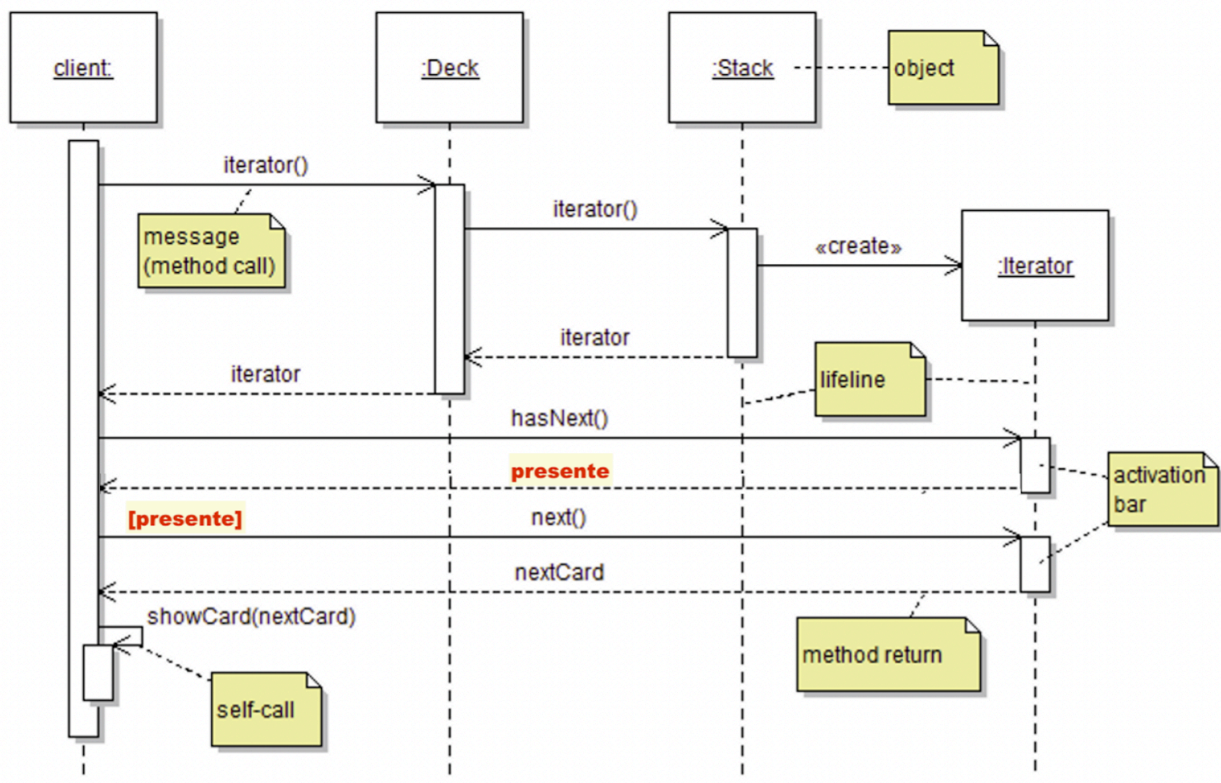
ITERATOR deck class diagram



Abbiamo:

- il concetto di **Iterator**
- **Client**, che dipende da **Iterator** che verrà ritornato da un metodo `iterator()` della classe **Deck** che viene probabilmente usato implicitamente da un `for`. Per cui è vero che non ha un attributo di tipo `iterator` ma dipende dalla interfaccia **Iterator** perché direttamente o indirettamente sta usando `next()` e `hasNext()`.
- **Deck**, che implementa l'interfaccia **Iterable<Card>** e infatti implementa il metodo `Iterator<Card> iterator()`, che è ciò che invocherà **Client** (**Client** è associato a **Deck**).
- **Stack**, contenuto dentro **Deck**, implementa l'interfaccia **Iterable<Card>**. È la implementazione di **Deck**. **Stack** può essere visto come un aggregazione usato per la **delega** di compiti, in questo caso il compito è gestire le carte. Può anche **rappresentare** lo stato in cui si trova il **Deck**.
- **Card** è contenuto dentro **Stack**.

ITERATOR deck sequence diagram



Possiamo notare che non tutti gli oggetti sono in cima.

`Client` siccome dialoga con `Deck` significa che ha un riferimento a `Deck`, quindi come minimo c'è una relazione di dipendenza.

La freccia continua indica la chiamata di metodo singolo fatta da un oggetto su un'altro oggetto.

`Client` chiama `Deck.iterator()`, che inoltra la chiamata a `Stack` che crea l'oggetto `Iterator`.

lifeline = la linea tratteggiata verticale, rappresenta timeline di vita dell'oggetto.

Lo stack di attivazione è uno spazio di memoria particolare, che viene gestito direttamente dallo heap, in cui si vanno a inserire le informazioni specifiche e peculiari solo dell'esecuzione della funzione, per cui ad esempio i valori delle sue variabili locali e chi mi ha chiamato.

activation bar = le box che si trovano sulle lifeline. Significa che l'esecuzione della funzione si trova sullo stack di attivazione.

DECORATOR

DECORATOR = is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

È un pattern che aggiungere dinamicamente funzionalità ad oggetti.

Simile al pattern COMPOSITE come diagramma delle classi ma non come compito. DECORATOR vuole creare degli oggetti che hanno delle caratteristiche in più o che possono cambiare il comportamento rispetto a certe caratteristiche.

DECORATOR vuole trattare casi in cui le decorazioni sono ortogonali tra di loro e quindi possono essere composte e combinate a piacere.

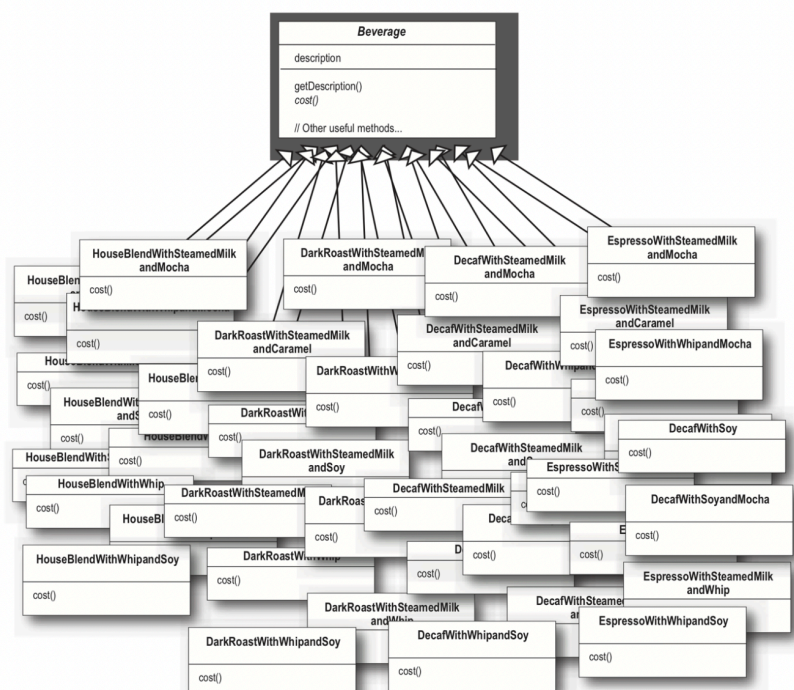
Ad esempio vogliamo aggiungere il fatto che a ogni invocazione di `draw()` venga scritta su console la carta estratta.

```
public class LoggingDeck implements CardSource {  
    ...  
    public Card draw(){  
        Card card = card.pop();  
        System.out.println(card);  
        return card;  
    }  
    ...  
}
```

Si definisce una nuova classe `LoggingDeck` che implementa `CardSource` e che definisce `draw()` e copia dalle altre classi i metodi di `CardSource` senza cambiarli.

Il termine DECORATOR deriva dall'aggiunta di decorazioni grafiche ad oggetti esistenti. In questo caso le decorazioni sono funzionalità aggiuntive.

L'approccio di andare a creare una nuova classe a fronte della necessità di avere una nuova decorazione è una soluzione semplice ma da evitare. Si potrebbe cadere in uno schema di questo tipo.



gerarchia di classi = creo una classe per ogni possibile combinazione di decorazioni.

la gerarchia di classi è statica, e per aggiungere una funzionalità con questo approccio bisogna:

- creare nuovo oggetto,
- copiare nel nuovo oggetto la parte di stato già esistente nel vecchio,
- distruggere l'oggetto vecchio,
- aggiornare tutti i riferimenti agli oggetti.

Se nel pattern abbiamo m oggetti base e n oggetti decorator (quindi m+n classi in tutto) di quante classi abbiamo bisogno con l'approccio con gerarchia di classi per trattare tutte le combinazioni (pur non considerando di poter usare per due volte una stessa decorazione, tipo doppia mozzarella della pizza o doppio bordo intorno a un testo)?

Avremmo bisogno di

$$m \cdot 2^n$$

classi.

Cioè il numero di classi necessarie esplode molto rapidamente.

Se una pizzeria volesse scrivere un menù in cui offre le possibili combinazioni di tre pizze base

(normale, integrale, senza glutine) e 10 decorazioni liberamente scegliibili e combinabili

(pomodoro, mozzarella, acciughe, olive, funghi, prosciutto, capperi, gorgonzola, tonno, ananas[!]) dovrebbe scrivere

$$3 \cdot 2^{10} = 3072$$

pizze possibili ognuna con il suo nome.

Quando la situazione da risolvere è quella prospettata dal pattern DECORATOR, l'approccio del pattern è probabilmente il migliore.

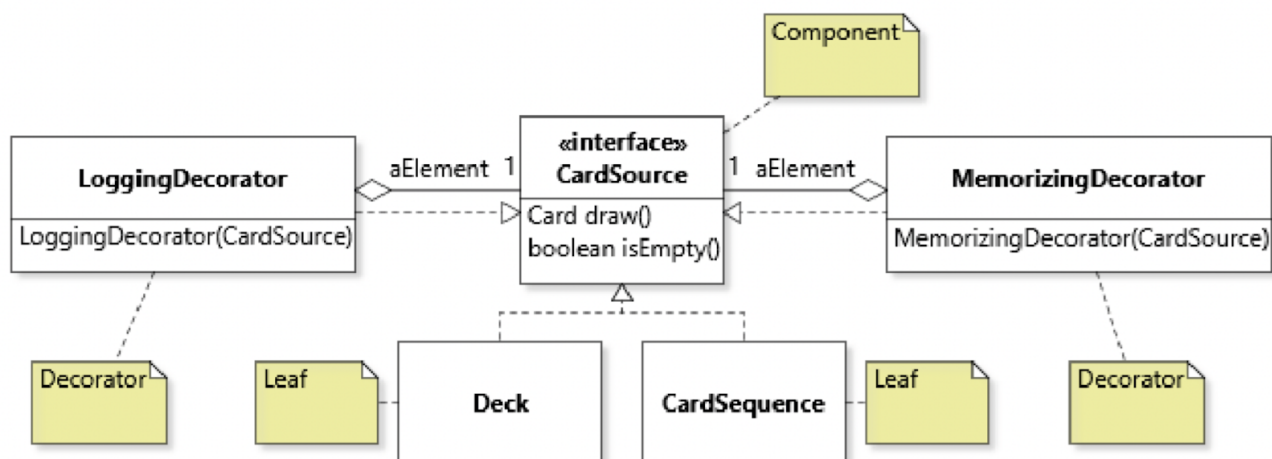
Diverso sarebbe se ad esempio ci trovassimo in un caso in cui il numero di decorazioni fosse limitato e sicuramente non espandibile, magari con non tutte le combinazioni delle decorazioni possibili (ananas, acciughe e gorgonzola,..) ed in cui non devo cambiare le caratteristiche degli oggetti durante la loro vita ma decidere le caratteristiche nel momento in cui creo l'oggetto (ad esempio non posso fare eccezioni delle poche combinazioni di pizze del menu né cambiare idea durante la loro preparazione). Allora in questo caso un approccio con gerarchia di classi potrebbe essere più chiaro e semplice.

GOD CLASS† = creo una classe unica in cui tramite degli attributi booleani e switch attualizzo le varie decorazioni. Nel libro esemplifica tramite la classe `MultiModeDeck`.

```
public class MultiModeDeck implements CardSouce{
    enum Mode {SIMPLE, LOGGING, MEMORIZING, LOGGING_MEMORIZING};
    private Mode aMode = Mode.SIMPLE;
    public void setMode(Mode pMode){...}
    public Card draw(){
        if(aMode==Mode.SIMPLE){...}
        else if(aMode==Mode.LOGGING){...}
    }
}
```

Invece di avere un unico attributo per ognuno delle decorazioni esiste un enumerativo enum `Mode`.

DECORATOR CardSource



LoggingDecorator aggrega e al tempo stesso implementa l'interfaccia CardSource. Simile a CompositeCardSource. Differisce per la cardinalità della relazione di aggregazione, mentre prima il CompositeCardSource aggregava tanti oggetti di tipo CardSource, l'oggetto decorator ne aggrega uno. Per cui si occupa di prendere uno specifico CardSource e applicare una decorazione.

Questo schema esplicita il costruttore del LoggingDecorator che ha un parametro, infatti bisogna esplicitare il CardSource che si deve decorare.

Sia LoggingDecorator che MemorizingDecorator sono due possibili decorator e condividono la stessa struttura, la cosa che li differenzia è come verrà ridefinito draw().

Nel momento in cui avremo ereditarietà la parte comune di queste due classi si potrà fattorizzare e mettere in una classe astratta.

Ci sono tanti decorator, uno per ogni possibile decorazione.

Questo schema funziona bene finché:

- cosa faccio per una decorazione è indipendente dal fatto se sto decorando un oggetto base o un oggetto decorato con altre decorazioni, non ci sono interazioni tra le varie decorazioni
- una decorazione può aggiungere comportamenti ma non toglierli, gli effetti additivi del DECORATOR corrispondono al principio SOLID di Liskov.

l'oggetto decorato con la soluzione DECORATOR, rispetto alle altre due soluzioni, quella con la GOD CLASS† e la gerarchia di classi, non corrisponde più a un'unica classe ma ad una catena di oggetti che sono la base di decorazione più tutti i vari decorator che si linkano l'uno con l'altro in una lista fino ad arrivare alla base.

Questo significa che, se dal punto di vista di scrittura del codice siamo riusciti a separare il decorator dalle varie parti del codice, dall'altra a run-time (ovvero durante l'esecuzione), ad esempio se dovessimo capire come è fatto in un certo momento un oggetto, non abbiamo tutte le informazioni chiare, bisogna percorrere una catena di oggetti. Questa operazione a run-time è complessa, c'è un po' di overhead.

Se però si progetta in maniera chiara, separata, semplice l'aggravio di complicazioni a run-time gestito dal compilatore è minimo perché non ci sarà bisogno di fare debugging (è meno probabile fare errori se si implementa in maniera corretta il pattern).

Qua sotto si può vedere come implementare il MemorizingDecorator.

```
public class MemorizingDecorator implements CardSource{
    private final CardSource element;
    private final List<Card> aDrawnCards = new ArrayList<>();

    public MemorizingDecorator(CardSource cardSource) {element = cardSource}
```

```

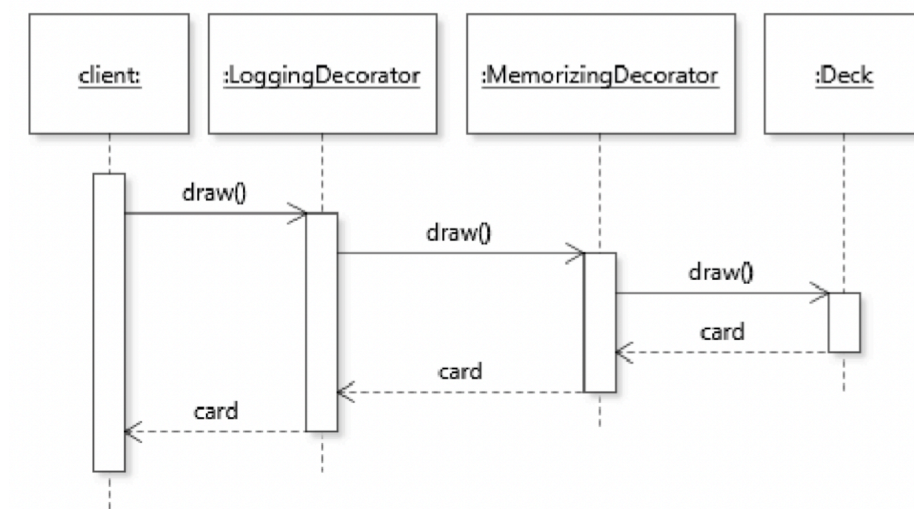
public boolean isEmpty(){ //rimane uguale
    return element.isEmpty();
}

public Card draw(){
    Card card = element.draw();
    aDrawnCards.add(card);
    return card;
}
}

```

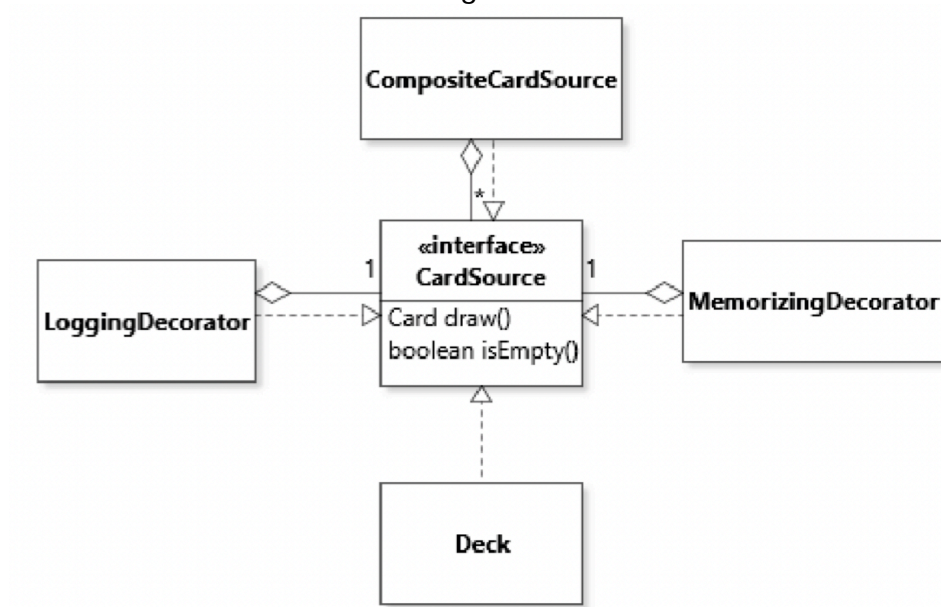
In sintesi con questo design pattern si crea una classe A che può implementare una interfaccia B e al tempo stesso decorare una istanza di tale interfaccia, ovvero A contiene un'istanza di B. La classe A ha quindi un costruttore di B. Inoltre può implementare nuovi metodi.

DECORATOR sequence diagram

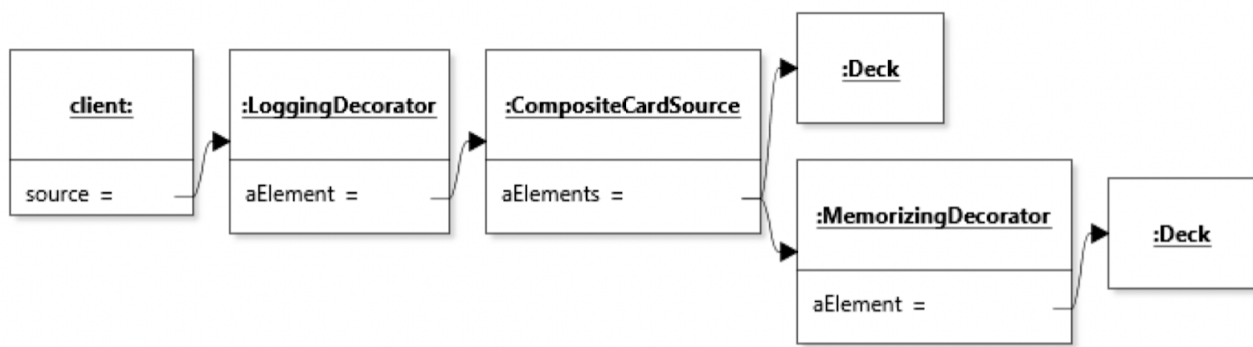


DECORATOR, COMPOSITE e diagrammi

DECORATOR e COMPOSITE: class diagram



DECORATOR e COMPOSITE: object diagram



Law of Demeter

Law of Demeter = states that a module should not have the knowledge on the inner details of the objects it manipulates. In other words, a software component or an object should not have the knowledge of the internal working of other objects or components.

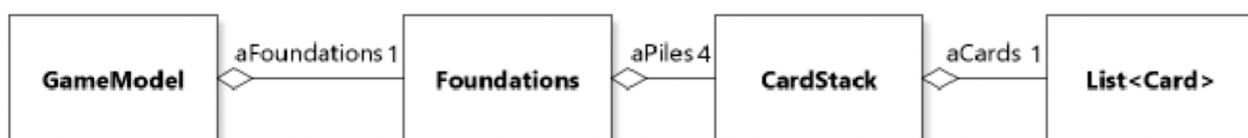
Se si modularizza, disaccoppia il sistema in vari moduli quello che c'è dentro a un modulo deve essere utile (coesione interna), inoltre il modulo deve cercare di dipendere il meno possibile da moduli esterni (disaccoppiamento esterno).

La legge di Demetra, o **principio della conoscenza minima**, esplicita questo concetto dicendo che:

- Ogni unità di programma dovrebbe interagire solo con le unità che conosce direttamente, bisogna minimizzare le dipendenze
- Non parlare agli sconosciuti
- tell don't ask, bisogna delegare

Bisogna conoscere i propri segreti e devo conoscere il meno possibile quello che non c'entra con me, è responsabilità altrui gestire quelli che non sono i miei segreti.

Con l'aggregazione (o composizione) è facile trovarsi con delle catene di questo tipo:



Un `GameModel` contiene una serie di mazzi sovrapposti che sono implementati con delle `CardStack` che sono implementati con delle `List<Card>`.

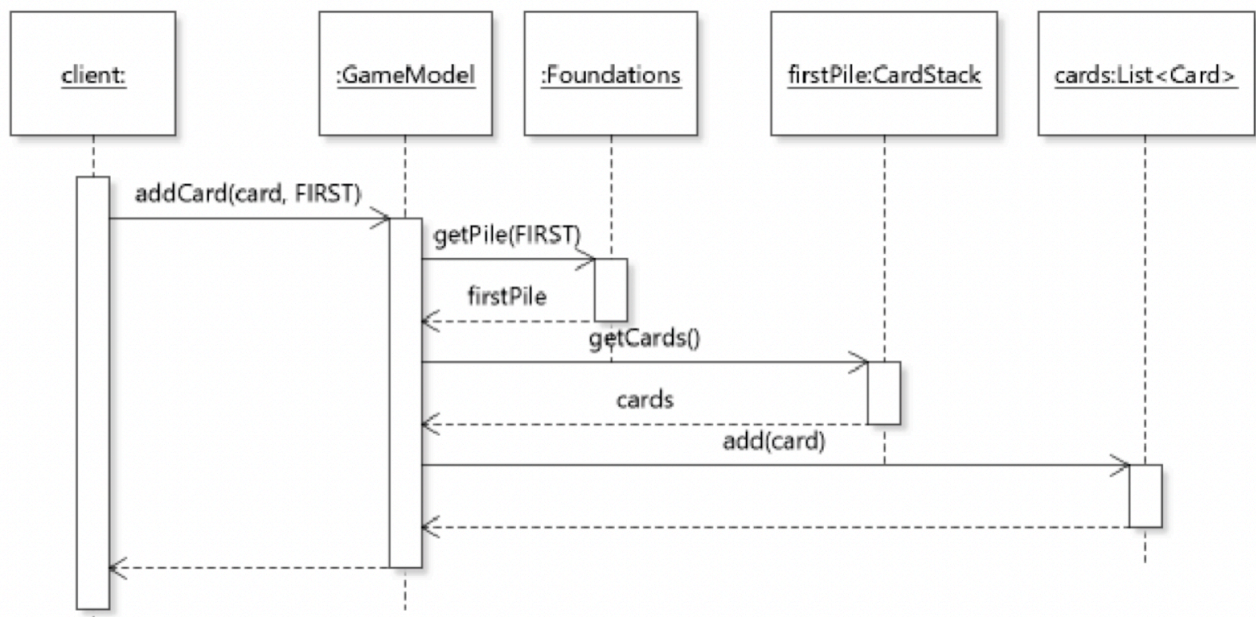
per aggiungere un carta dentro `GameModel` potrebbe venire in mente un codice di questo tipo:

```
void addCard(Card card, TableauPile FIRST){
    this.foundations.getPile(FIRST).getCards().add(card);
}
```

Sembra una scrittura concisa, in realtà si chiede a `Foundations` il suo segreto, ovvero la pila di carte che si trova al suo interno. In questo modo `GameModel` conosce `CardStack`, invece dallo schema UML si può vedere che si conoscevano indirettamente. Allo stesso modo `GameModel` conosce `List<Card>`.

Tutto questo va contro il principio della conoscenza minima.

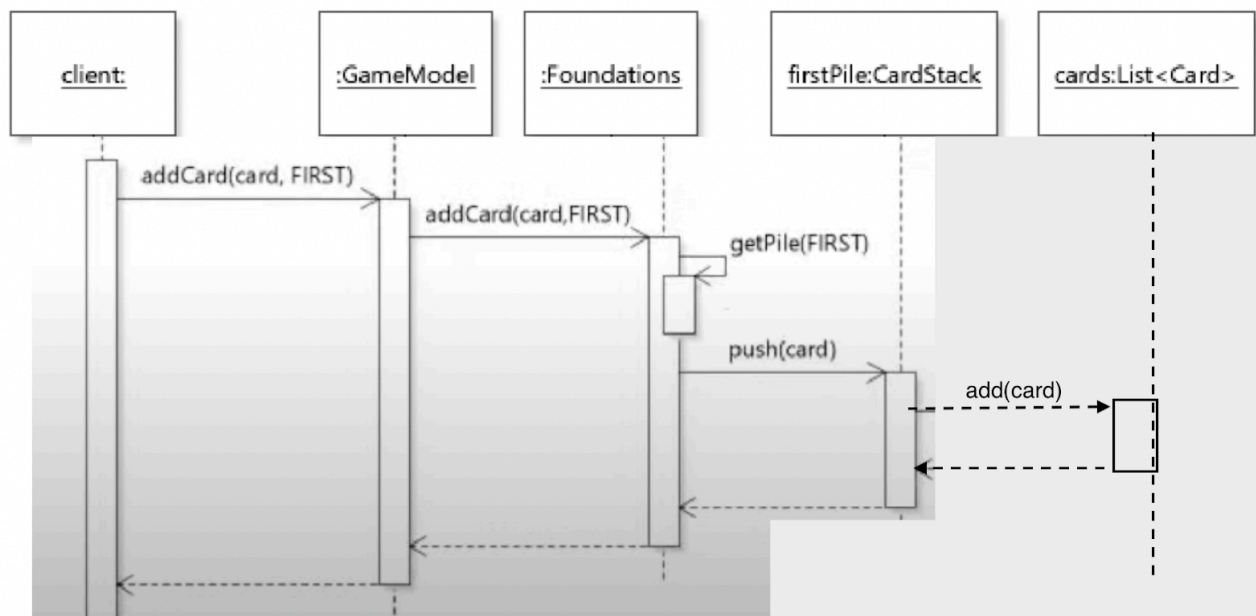
Se noi guardiamo un diagramma di sequenza di questo codice ci si accorge che quando il cliente chiede di aggiungere una carta tutte le interazioni passano da GameMode1. Questa classe toglie la responsabilità alle altre della gestione dei loro segreti.



Un cambiamento potrebbe essere questo, dove ognuno è responsabile dei propri segreti.

```

void addCard(Card card, TableauPile FIRST){
    this.foundations.addCard(card, FIRST);
}
  
```



Dato questo esempio si può osservare una procedura di questo tipo:

- mettere tutti gli attributi privati
- se ce ne bisogno mettere getter
- il codice è finito e funziona. Ci si chiede il perché di un getter (dare il segreto), e ci si domanda se sia possibile delegare (fare questa azione sul proprio segreto)
- si cancella il getter e si implementa un nuovo metodo nella classe = **operazione di refactoring**

In questo modo il rispetto della legge di Demetra è incrementale.

In sintesi la legge di Demetra dice che il codice di un metodo dovrebbe accedere solamente a:

- `this` cioè all'oggetto su cui è stato invocato
- agli argomenti passati al metodo
- agli oggetti creati all'interno del metodo
- (se proprio necessario) oggetti disponibili globalmente (ad esempio campi statici di altre classi, oggetti singleton...)

Inheritance

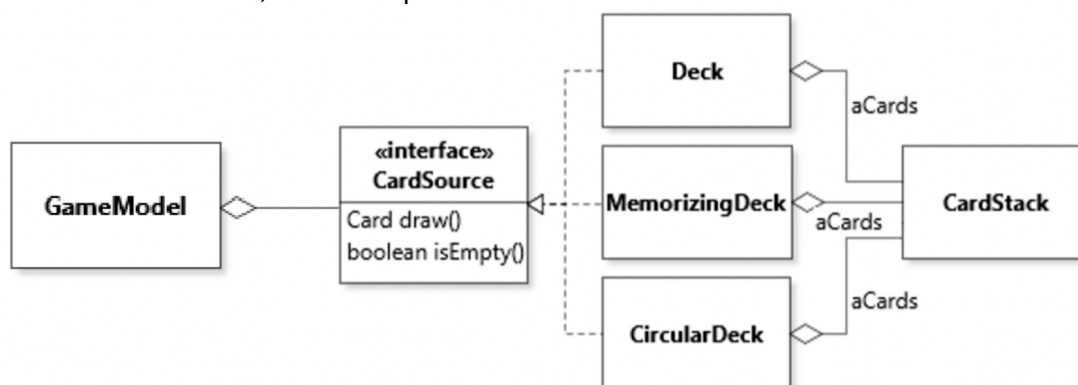
La programmazione orientata agli oggetti è un modo di pensare, progettare e programmare. Un linguaggio orientato agli oggetti si può definire tale solo se prevede:

- Ereditarietà di classi
- Polimorfismo
- Collegamento dinamico

A ciò si uniscono i principi della programmazione modulare:

- Astrazione
- Incapsulamento e Information Hiding.

Abbiamo parlato di ereditarietà con le interfacce, infatti quando si eredita un'interfaccia si ereditano dei doveri, ovvero implementare e definire dei metodi ma non il come.

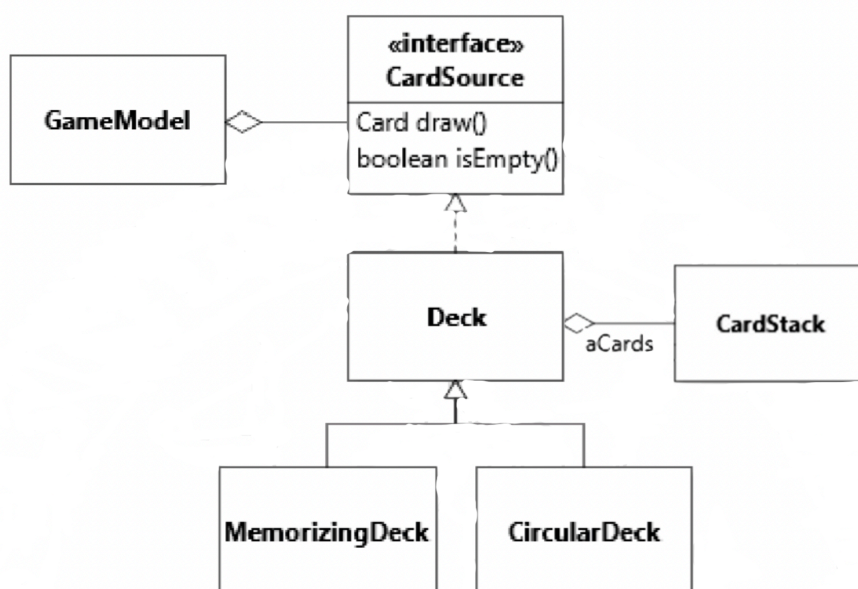


Il vantaggio di implementare un'interfaccia è quello di poter usare il meccanismo del polimorfismo e ad esempio dire che **GameModel** contiene **CardSource**, ovvero oggetti che implementano i metodi `draw()` e `isEmpty()`, e avere diverse classi le cui istanze possono essere assegnate a una lista di **CardSource**.

Le classi **Deck**, **MemorizingDeck** e **CircularDeck** devono avere tutte un **CardStack** al suo interno.

Le classi possono sia implementare un'interfaccia che estendere un'altra classe. Si prende una classe come modello, che può essere un modello completo o incompleto, e la nuova definizione di classe è simile a quella del modello ma con delle aggiunte.

Lo schema precedente può essere espresso in questa maniera:



GameModel contiene CardSource, quest'ultimo è implementato dalla classe Deck che contiene delle carte con una attributo di tipo CardStack. Ci sono due classi che **specializzano** la classe Deck in due modi particolari:

- MemorizingDeck in modo che sia capace di memorizzare
- CircularDeck in modo che sia capace di gestire circolarmente il mazzo

Queste due classi **ereditano** la classe Deck e indirettamente l'implementazione di CardSource e il fatto che stanno aggregando un CardStack.

subtyping = compatibilità dei tipi.

Ovunque ho un attributo, una variabile, un identificatore dichiarato come CardSource posso mettere un oggetto di tipo Deck ma anche un oggetto di tipo CircularDeck perché è un **sottotipo** di Deck che è **compatibile** con CardSource.

Mentre in UML si può dire che una classe eredita da due o più classi, in java

- Non c'è supporto per l'ereditarietà multipla
- Si può estendere al massimo una classe.
- Ogni classe ha al massimo un padre, ma può avere nonni, bisnonni, etc.

La gerarchia delle ereditarietà in java corrisponde ad un albero.

Single rooted = unico albero con un'unica radice. Tutte le classi ereditano per default dalla classe Object (equals, toString).

Cosa si può ereditare?

- si può ereditare attributi (variabili di istanza, campi, fields).
- metodi

Quando si eredita una classe si ereditano tutti gli attributi e:

- non posso cambiarne il tipo
- se sono private non posso accedervi dai metodi della sottoclasse (anche se viene comunque ereditato), esiste nuovo ambito di visibilità, il protected, in questo modo le classi figlie ci possono lavorare sopra

Posso ridefinire i metodi ereditati tramite Override, ma bisogna farlo solo per estenderne il comportamento (LSP) garantendone il corretto funzionamento.

In UML se vengono ridefiniti vanno riscritti nello schema, se vengono solo ereditati non vanno riscritti. Nel codice è utile usare la annotazione @Override per permettere alcuni check al compilatore.

Attenzione a non confondere con Overloading:

- basta cambiare il tipo o il numero dei parametri per avere un nuovo metodo indipendentemente dal fatto che abbia lo stesso nome
- cambiare invece il tipo ritornato a quello di un sottotipo non crea un nuovo metodo

Il subtyping vale anche per gli antenati di grado maggiore di 1.

Quindi posso assegnare una istanza di un sottotipo A a un identificatore di un suo antenato B? Sì... è appunto il polimorfismo

In questo caso, viene cambiato(modificato/copiato/altro) l'oggetto? NO... il suo run-time type (a tempo di esecuzione) non cambia

Posso chiamare i metodi che nel tipo antenato non esistevano? NO... il suo compile-time type cambia

Se chiamo un metodo che nel tipo antenato esisteva ma è stato ridefinito, quale viene chiamato? Viene chiamato quello della classe sottotipo (quello specifico)... è il collegamento dinamico

```
public class Padre{
    public void metodo1 (int pippo){
        System.out.println("metodo del padre");
    }
}
```

```

public class Figlio extends Padre{
    public void metodo2(String a){
        System.out.println(a);
    }

    @Override
    public void metodo1 (int pippo){
        System.out.println("metodo del figlio");
    }

    public static void main(String[] args){
        Figlio f = new Figlio();

        Padre p;
        //polimorfismo
        p = f; //OK
        //ad un identificatore di classe padre posso assegnare un'istanza della
        classe figlio

        f = p; //NO

        p.metodo1(); //OK
        //sul compilatore viene stampato "metodo del figlio"
        //infatti il compile-type decide se una operazione che si può invocare
        //grazie al collegamento dinamico e al run-time-type si accorge che dentro p
        c'è un'istanza di figlio
        //si sceglie il metodo più vicino

        p.metodo2("paperino"); //NO
        //padre non ha il concetto di metodo2
        //è vero che dentro padre c'è un figlio ma questo metodo viene considerato
        dal compilatore
        //e non sa cosa è stato assegnato a padre, sarebbe una informazione dinamica
        //per cui quando si lavora sull'identificativo p, che è dichiarato di tipo
        padre
        //si possono chiamare solo le cose definite a livello di padre, non le cose
        definite nei sottotipi
        //dato che il programmatore sa che in realtà questo metodo è possibile poichè
        dentro p c'è un'istanza di f si può fare un
        //downcasting = si forza l'identificatore a essere visto come qualcosa di
        diverso
        ((Figlio)p).metodo2("pluto");
        //considera questo oggetto come se fosse di tipo Figlio
        //se fosse stato veramente di tipo padre (run-time-type padre) non può
        eseguire questa operazione

        f.metodo2("pippo"); //OK

        f.metodo1(); //OK
        //Figlio eredita il metodo
        //se fosse stato privato NO
        //se fosse stato protected OK, ma non è chiamabile da una classe esterna che
        non figlia di Padre
    }
}

```

(Down)Casting esplicito = mentre per assegnare una istanza a un identificatore di un tipo antenato non c'è bisogno di fare nulla a volte si vorrebbe promuovere un identificatore a essere visto come se fosse quello di un sottotipo.

```

Deck deck = new MemorizingDeck();
MemorizingDeck mdeck;

deck.memorizeSpecificMethos(); //ERRORE

mdeck = deck; //ERRORE

mdeck = (MemorizingDeck) deck; //casting esplicito
mdeck.memorizeSpecificMethos(); //OK

if(deck instanceof MemorizingDeck){
    mdeck = (MemorizingDeck) deck; //casting esplicito controllato
    mdeck.memorizeSpecificMethos(); //OK
}

```

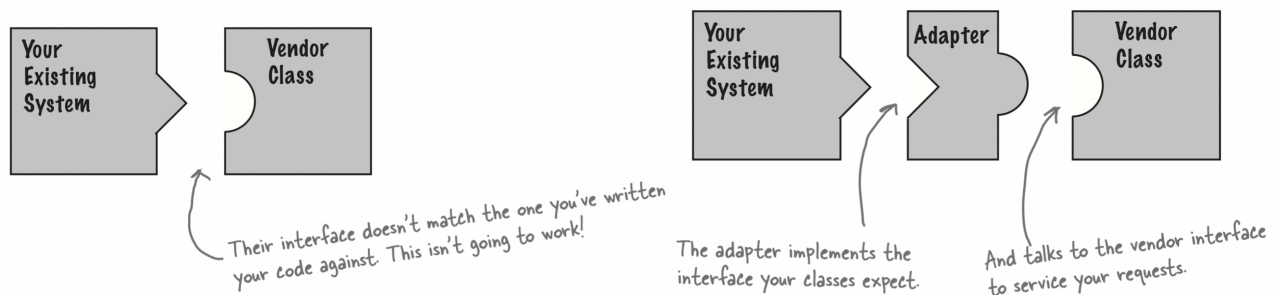
`instanceOf` è una keyword che controlla se il run-time-type di quella variabile è compatibile a un'altra istanza. Il casting e l'uso della `instanceOf` sono comunque cose da evitare il più possibile e spesso sono indicatori di cattivo design (le classi padri vengono a conoscenza delle classi figlio).

Per impedire che una classe abbia figli si aggiunge la keyword `final` alla classe, `String` ad esempio è una classe `final`.

Per evitare di avere altri figli se la classe padre ha già permesso a qualcuno di esserlo si sfrutta la visibilità dei package.

ADAPTER

ADAPTER = is a structural design pattern that allows objects with incompatible interfaces to collaborate.

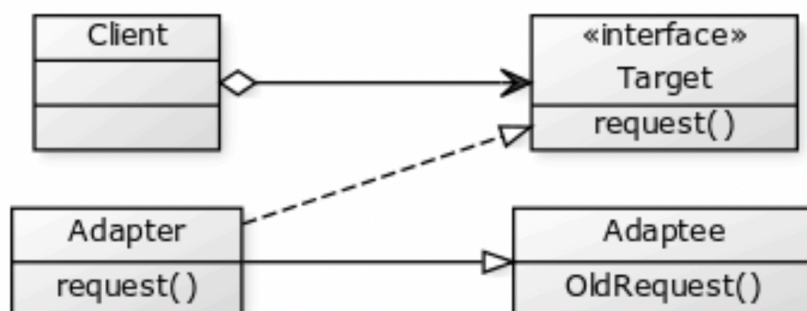


ADAPTER, è un Design Pattern di tipo strutturale che è utilizzato quando si ha la necessità di rendere compatibili due interfacce che di fatto non lo sono, senza modificarne il codice (si rispetta il principio Open-Closed).

Ci sono due alternative:

- **object adapter** = si adatta un singolo oggetto
- **class adapter** = si adatta un'intera classe

class ADAPTER



Abbiamo un **Client** che usa un'interfaccia **Target** e un oggetto **Adaptee** (oggetto da adattare) che usava un'interfaccia diversa. In mezzo a **Client** e **Adaptee** c'è un **Adapter**.

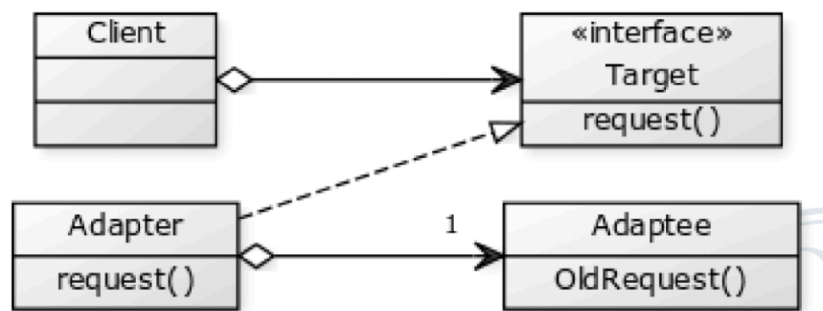
Adapter **estende** (eredita da) **Adaptee**, quest'ultimo aveva un metodo `oldRequest()` mentre **Client** si aspetta di fare questa richiesta semplicemente con il nome `request()`.

Adapter implementa l'interfaccia **Target**. In questo modo **eredita** il dovere di implementare la `request()` di **Target** e allo stesso tempo è capace di rispondere anche a delle richieste `oldRequest()`.

```
public class Adapter extends Adaptee implements Target {  
    @Override  
    public void request(){  
        this.oldRequest();  
    }  
}
```

Per il polimorfismo un oggetto di tipo **Adapter** può essere assegnato a identificatori di tipo **Target** e **Adaptee**.

object ADAPTER



In questo caso cambia la relazione di Adapter e Adaptee. Infatti se prima era un relazione di generalizzazione, ereditarietà (mette in relazioni due classi) ora è una relazione di **aggregazione**. Adapter contiene al suo interno un oggetto di tipo Adaptee (mette in relazione istanze delle classi, infatti si può osservare la cardinalità delle istanze con cui si è in relazione).

```
public class Adapter implements Target {
    private final Adaptee adaptee;

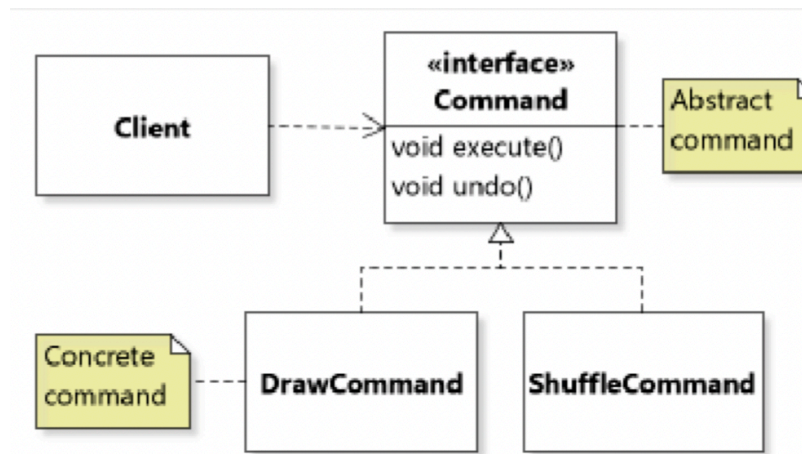
    public Adapter(Adaptee adaptee){
        assert adaptee != null;
        this.adaptee = adaptee;
    }

    @Override
    public void request(){
        this.adaptee.OldRequest();
    }
}
```

Si aggiunge il costruttore in modo da poter istanziare, dire qual'è l'oggetto che si sta adattando.

COMMAND

COMMAND = is a behavioral design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.



È un pattern che definisce una unità di funzionalità. Sembra simile a STRATEGY, infatti abbiamo un cliente che dialoga con una interfaccia che viene implementata da diverse classi.

(Assomiglia ancora di più a quello che abbiamo chiamato un Function Object.)

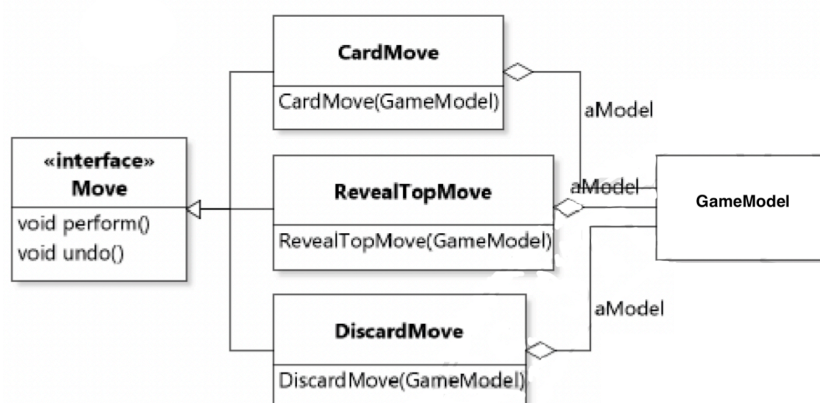
Nel pattern COMMAND viene dato l'accento sul fatto che è presente un'interfaccia di comandi.

Nel caso del diagramma UML qua sopra abbiamo un oggetto di tipo Command, con due funzioni, `execute()` e `undo()`, e poi comandi, quindi si può fare e disfare ogni operazione.

Alcune scelte critiche da compiere sono:

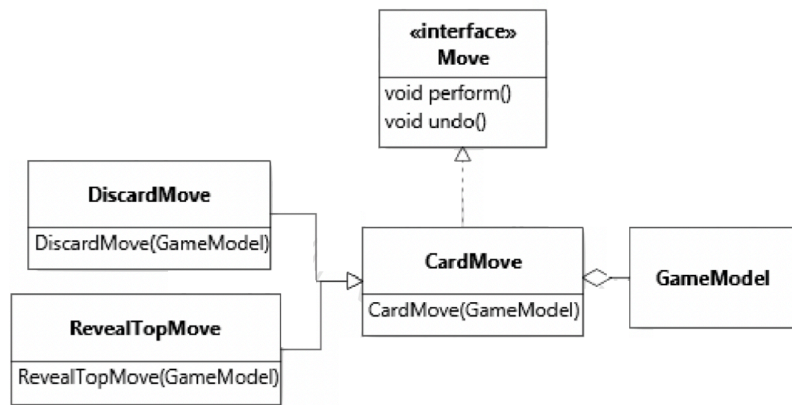
- come rendo disponibile al comando il contesto in cui opera? Fa parte del suo stato (setter o costruttore) o glielo passo come parametro?
- ritorna un valore?
- cosa succede alle precondizioni? Chi è responsabile del loro soddisfacimento?

Vediamo un esempio di COMMAND per il gioco di carte.



Abbiamo **CardMove**, **RevealTopMove** e **DiscardMove**, per ognuna di queste classi passiamo un **GameModel** dentro i costruttori. Queste classi implementano l'interfaccia **Move** e quindi sono capaci di fare `undo()` e `perform()` delle loro operazioni.

Si potrebbe fattorizzare anche l'aggregazione del **GameModel** come nel diagramma UML qua sotto? Sì, ma la situazione qua sotto è sbagliata, **CardMove** non è un tipo generale della classe **DiscardMove**, fanno operazioni completamente diverse.



Fattorizza veramente l'aggregazione in quanto **CardMove** non è una interfaccia ma una classe. Sarebbe opportuno non mettere una vera classe ma solo la **parte comune** alle classi, ovvero una classe astratta.

classi astratte = l'identificazione di metodi e attributi comuni a più classi potrebbe portare a volte a voler definire una superclasse che non ha senso istanziare (ad esempio perché un progetto incompleto).

- Se una classe ha almeno un metodo dichiarato (eventualmente da interfaccia) ma non definito, non può essere istanziata

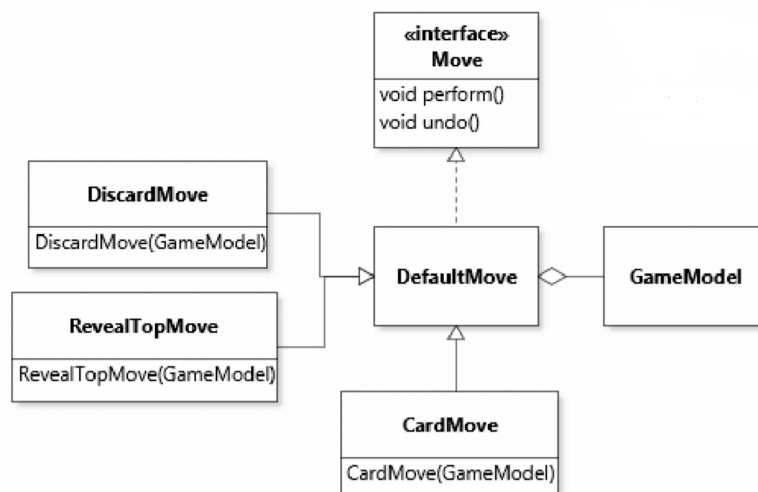
Simili a interfacce, ma meno limiti su attributi e metodi

- utili per fattorizzare parti di codice e responsabilità
- attenzione che continua a valere divieto di eredità multipla

In UML il nome di una classe astratta viene scritto in *italic*.

Costruttori? Sono usati dai costruttori delle sottoclassi.

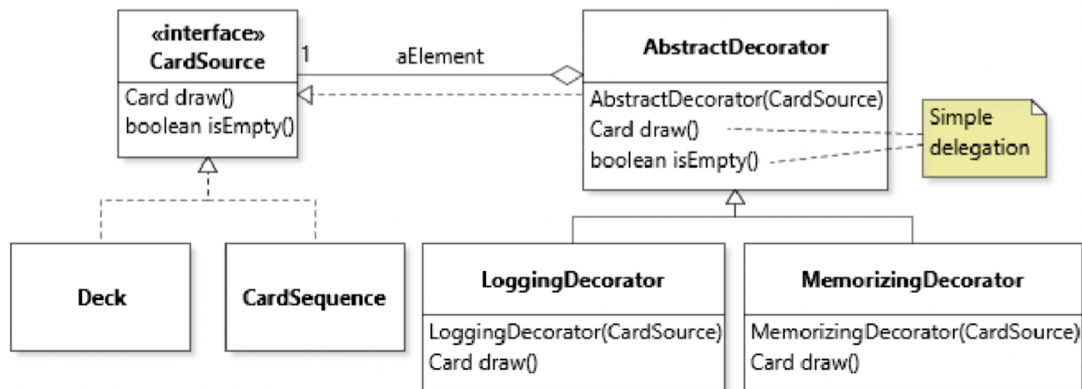
Qua sotto possiamo vedere COMMAND con l'uso di una classe astratta.



Abbiamo messo a fattore comune una classe astratta **DefaultMove**. Le altre classi estendono **DefaultMove**.

C'è però un problema da risolvere: l'attributo (se è privato) come viene visto dalle sottoclassi? Se quell attributo serve va messa la keyword `protected`.

Rivisitazione pattern DECORATOR



Si può osservare che esiste una parte comune a tutti i decorator, quindi si costruisce una classe `AbstractDecorator` tale per cui vado a scrivere quello che c'è in comune alle varie classi. Vengono ridefiniti solo i metodi che servono mentre altri metodi usano l'implementazione di default.

```

public abstract class AbstractDecorator implements CardSouce{
    private final CardSouce element;

    public AbstractDecorator(CardSouce element){
        assert element != null;
        this.element = element;
    }

    @Override
    public Card draw(){
        return element.draw();
    }

    @Override
    public boolean isEmpty(){
        return element.isEmpty();
    }
}

```

```

public class LoggingDecorator extends AbstractDecorator{
    public LoggingDecorator(CardSouce element){
        super(element);
    }

    @Override
    public Card draw(){
        Card card = suepr draw();
        System.out.println(card);
        return card;
    }
}

```

Se mi accorgo che c'è uno schema fisso nei decoratori concreti per cui sempre richiamo il padre, "fanno qualcosa sulla carta" e la ritornano, sposto questo schema nella classe astratta:

```

@Override
public Card draw(){
    Card card = element.draw();
    specificDecorationAction(card);
    return card;
}

```

```
}  
abstract protected void specificDecorationAction(Card card);
```

e nella sottoclasse dico semplicemente:

```
public class LoggingDecorator extends AbstractDecorator{  
    public LoggingDecorator(CardSouce element){  
        super(element);  
    }  
  
    @Override  
    protected void specificDecorationAction(Card card){  
        System.out.println(card);  
    }  
}
```

TEMPLATE

TEMPLATE = is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

Quando esiste un schema (template) fattorizzabile a livello di una classe di cui alcune parti dipendono (e verranno definiti) nelle sottoclassi.

Principi SOLID

Lettera	Principio	Acronimo	In breve
S	<i>Single responsibility principle</i> o principio di singola responsabilità	SRP	Afferma che ogni classe dovrebbe avere una ed una sola responsabilità, interamente incapsulata al suo interno.
O	<i>Open/closed principle</i> o principio aperto/chiuso	OCP	Un'entità software dovrebbe essere aperta alle estensioni, ma chiusa alle modifiche.
L	<i>Liskov substitution principle</i> o principio di sostituzione di Liskov	LSP	Gli oggetti dovrebbero poter essere sostituiti con dei loro sottotipi, senza alterare il comportamento del programma che li utilizza.
I	<i>Interface segregation principle</i> o principio di segregazione delle interfacce	ISP	Sarebbero preferibili più interfacce specifiche, che una singola generica.
D	<i>Dependency inversion principle</i> o principio di inversione delle dipendenze	DIP	Una classe dovrebbe dipendere dalle astrazioni, non da classi concrete.

Single Responsibility

Ogni classe dovrebbe avere un singolo incarico. Ciò permette classi più facili da riutilizzare, da comprendere e meno soggette a modifiche illecite.

“Ogni oggetto deve essere responsabile di un singolo aspetto del comportamento che stiamo modellando.”

Open/closed Principle

“Progettare entità che non necessitano di essere modificate a fronte dei naturali cambiamenti delle nostre applicazioni, ma che possano essere estese con estrema facilità.”

Il codice deve essere aperto alle evoluzioni ma chiuso le modifiche: nuove funzionalità non dovrebbero modificare codice già esistente. Infatti ogni cambiamento in un oggetto e ogni aggiunta di un nuovo codice si ripercuote sulla classe stessa. Alla lunga una architettura come questa genera un prodotto i cui componenti sono così interdipendenti da non poter più essere riutilizzati, inoltre mantenere un software costruito in questo modo è veramente difficile, come possiamo sapere in quanti punti una modifica ad uno specifico oggetto deve essere propagata?

La soluzione suggerita dal secondo dei principi SOLID è quella di creare un oggetto che sappia fare da cuscinetto tra chi deve orchestrare l'azione e chi invece possiede le informazioni per eseguirla. Esistono modi diversi per raggiungere questo risultato, si possono ad esempio utilizzare delle interfacce che descrivano i metodi che poi dovranno essere implementati dalle classi che vogliono attivarsi sul quel particolare comportamento.

Interface Segregation Principle

Sarebbero preferibili più interfacce specifiche, che una singola generica. [Rimando](#).

Dependency Inversion

L'Inversione della Dipendenza si basa su due assunzioni:

- *“ I moduli di alto livello non dovrebbero dipendere da moduli di basso livello. Entrambi dovrebbero dipendere da astrazioni. ”*
- *“ Le astrazioni non dovrebbero dipendere dai dettagli. I dettagli dovrebbero dipendere dalle astrazioni ”*

Liskov substitution principle (LSP)

Questo principio fu enunciato per la prima volta da Barbara Liskov nel 1987, asserendo che una sottoclasse non deve restringere ciò che la sua superclasse può fare con una istanza.

Io posso usare un oggetto/un'istanza di una sottoclasse ovunque viene citato un identificatore di una superclasse perché tutte le cose che posso chiedere a quel identificatore della superclasse (o sopra classe) le sanno fare anche tutte le istanze della sotto classe.

Per cui è sostituibile grazie al polimorfismo e grazie al collegamento dinamico che andrà a chiamare l'implementazione corretta.

Se ridefinisco un metodo, la nuova implementazione:

- non può avere precondizioni più strette
- non può avere postcondizioni più larghe
- non può richiedere parametri più specifici
- non può ridurre visibilità dei metodi
- non può lanciare nuovi tipi di eccezioni
- non può avere un tipo di ritorno meno specifico

OBSERVER

OBSERVER = is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

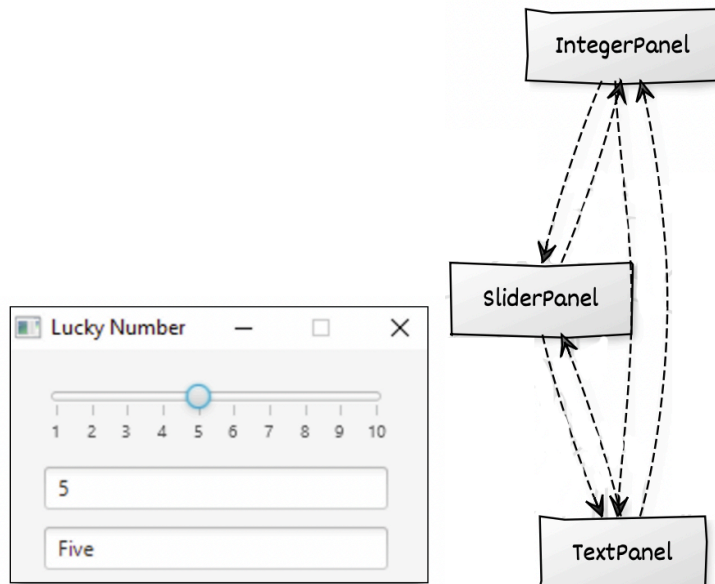
OBSERVER ha molto a che fare con il principio di Inversion of Control, la classe osservata sa che qualcuno la invocherà.

Inversion of Control = (IoC) is a design principle that allows classes to be loosely coupled and, therefore, easier to test and maintain. IoC refers to transferring the control of objects and their dependencies from the main program to a container or framework.

Nei modelli in cui c'è l'Inversion of Control si definiscono tanti punti che possono essere chiamati da qualcun altro, un esempio è la programmazione ad eventi. Nel caso dell'OBSERVER ci sono diversi oggetti che vogliono rimanere sincronizzati e coerenti tra di loro.

Il problema da affrontare è l'antipattern PAIRWISE DEPENDENCIES† dove si hanno delle dipendenze a coppie. Questo porta ad avere due grandi problemi:

- forte accoppiamento = ogni vista deve conoscere le altre viste
- bassa espansibilità = complicato aggiungere e togliere altre viste



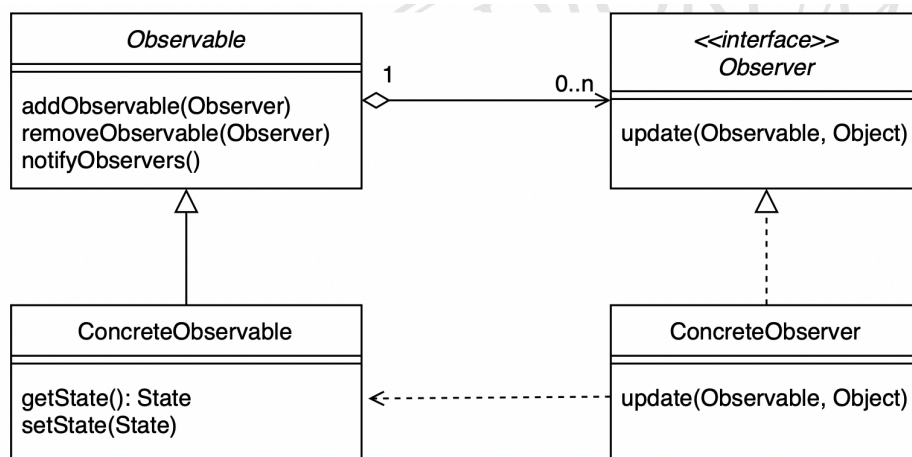
La soluzione in questi casi è creare un'altra classe che modella ed estragga la parte comune in modo che lo **stato** verrà:

- messo in un oggetto a parte (Subject o Observable)
- osservato dagli altri (Observer)

Ognuno degli osservatori si mette in relazione con l'osservabile e non conosce gli altri osservatori, invece l'osservabile deve avere una conoscenza di tutti gli osservatori.

In Java c'erano (ora è deprecato) delle classi/interfacce consigliate per realizzare questo pattern:

- interfaccia `java.util.Observer`
- classe `java.util.Observable`



Come colleghiamo `Observable` e `Observer`? `Observable` (segnata come una classe astratta, ma in generale può anche non esserlo) è collegato con un `Observer` tramite una relazione di aggregazione unidirezionale, ovvero gli `Observable` conoscono gli `Observer` ma non il contrario. La relazione di aggregazione significa che parte della definizione di `Observable` è chi lo sta osservando (ci sono delle istanze di osservatori dentro l'oggetto osservato in modo da tenerne traccia per aggiungerli e aggiornarli). Questi due oggetti, la classe astratta e l'interfaccia, devono essere concretizzati, per cui è presente:

- un oggetto `ConcreteObservable` che estende (è una sottoclasse) di `Observable`
- un oggetto `ConcreteObserver` che implementa l'interfaccia.

Tra `ConcreteObserver` e `ConcreteObservable` c'è una relazione di dipendenza verso quest'ultimo.

Come scoprono gli `Observer` lo stato della `Observable`?

Una possibilità è quella di fare **polling** = is the continuous checking of other programs or devices by one program or device to see what state they are in, usually to see whether they are still connected or want to communicate.

Nel caso del polling l'`Observer` continua a interrogare, con una certa frequenza, l'`Observable`. Questo meccanismo di busy waiting è però inefficiente.

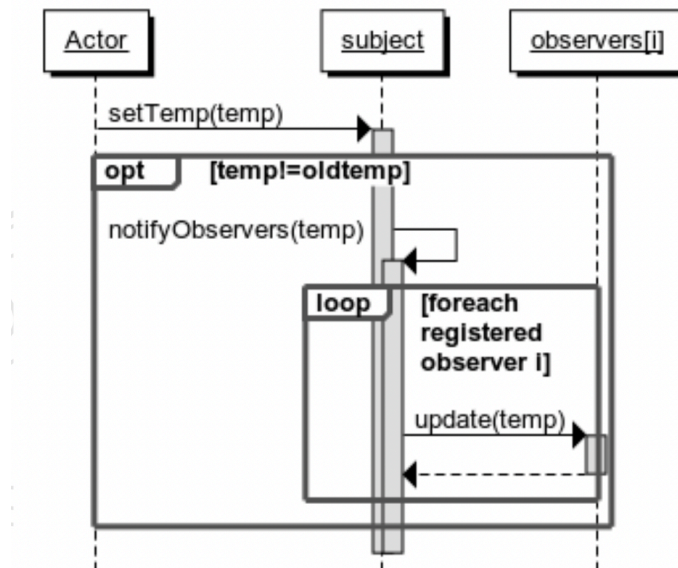
Un'alternativa è l'uso degli `interrupt`, ovvero `Observable` segnala agli `Observer` interessati che è il suo stato è cambiato. Dentro `Observable` c'è un attributo che mantiene un elenco degli osservatori tramite `addObserver()`.

`Observer` viene notificato tramite il metodo `notifyObservers()` di `Observable`. Nel momento in cui vengono notificati gli `Observer` vengono aggiornati tramite `update()`. Ci sono due modalità con cui si può far passare l'informazione di modifica di stato:

- **push** = si spinge lo stato
- **pull** = si chiede di venire a vedere lo stato

OBSERVER: push

Lo stato modificato viene passato come argomento alla callback.



Abbiamo un Actor che chiama il metodo `setTemp()`. Subject, su cui è stato invocato il cambiamento di stato, si chiede se la temperatura passata è diversa da quella che aveva in precedenza, se è esattamente uguale significa che lo stato non è cambiato. Se invece la condizione è vera, Subject chiama su se stesso `notifyObserver()` per cui attiva un'altra barra di attivazione dell'oggetto all'interno del quale viene ripetuto ciclicamente per ognuno degli osservatori registrati il fatto che chiamo `update()` passando il parametro, in questo caso la nuova temperatura.

```
//OBSERVABLE
@Override public void notifyObservers(state){
    for (Observer observer : observers) {
        observer.update(null, state);
    }
};

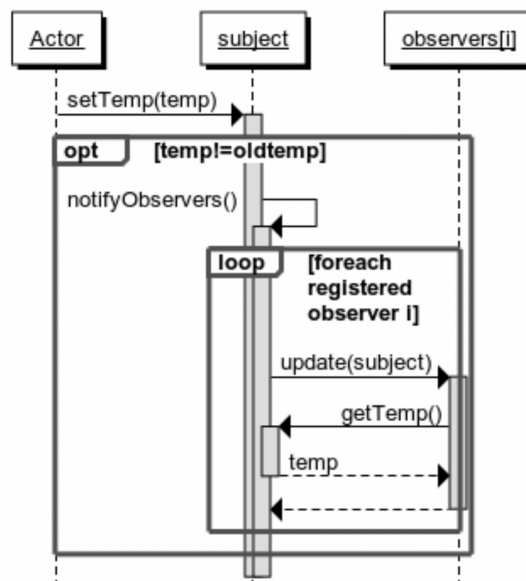
//OBSERVER
@Override public void update(Observable model, Object state){
    if (state instanceof Integer)
        doSomethingOn((Integer) state);
}
```

`doSomethingOn()` è un metodo privato della classe `ConcreteObserver` che si aspetta di ricevere un `Integer`, però quello che sta ricevendo è un `Object`. Allora bisogna fare il casting con l'aggiunta di un controllo.

L'uso della `instanceOf` protegge dal crash del programma ma inserisce una **dipendenza esplicita** dal tipo intero.

OBSERVER: pull

L'argomento della callback è l'oggetto osservato stesso su cui potranno essere invocati metodi per ottenere lo stato (totale o parziale).



C'è un passaggio in più rispetto all'approccio push. Subject notifica agli osservatori che si è aggiornato tramite il metodo `update(subject)`. In seguito l'osservatore richiede la temperatura tramite `getTemp()` a Subject.

Il problema di questo approccio è che si possono accodare più chiamate di `update()` o, nel tempo che intercorre tra una chiamata `update()` e `get()` lo stato può essere stato modificato ulteriormente. Magari si può aspettare un certo numero di chiamate prima di chiamare l'aggiornamento.

```
//OBSERVABLE
@Override public void notifyObservers(){
    for (Observer observer : observers){
        observer.update(this, null);
    }
};

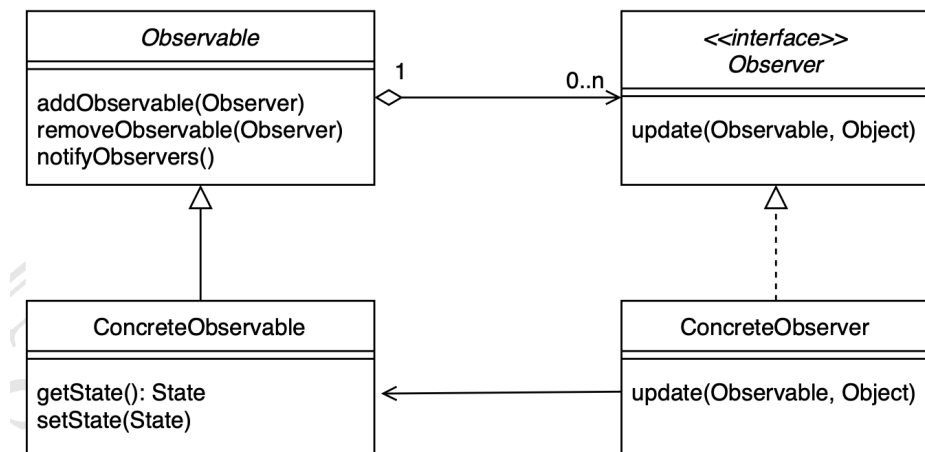
//OBSERVER
@Override public void update(Observable model, Object state){
    if (model instanceof ConcreteObservable)
        doSomethingOn(((ConcreteObservable) model).getState());
}
```

Per eliminare la dipendenza si può esplicitare il legame con un attributo.

```
private ConcreteObservable subject;

public ConcreteObserver(ConcreteObservable co){
    assert co != null;
    co.addObserver(this);
    subject = co;
}

@Override public void update() {
    doSomethingOn(subject.getState());
}
```



FACTORY METHOD

FACTORY METHOD = is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

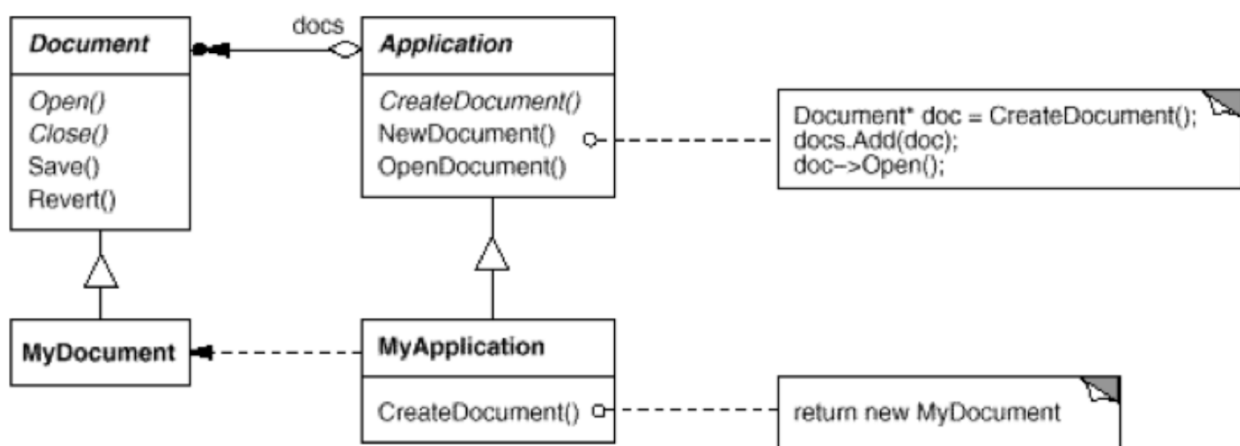
FACTORY METHOD può essere visto come un sinonimo di un costruttore statico, cioè di un metodo statico che è capace di creare un oggetto, come approccio alternativo al definirsi (rendere pubblico) un costruttore.

La differenza tra un costruttore e un metodo che costruisce un oggetto è che un costruttore non può non restituire un oggetto, al massimo solleva una eccezione, mentre un metodo può decidere se creare e un oggetto o restituire un riferimento ad un oggetto.

Inoltre quando si usa un metodo invece del costruttore, soprattutto se non è un metodo statico, lo si può mettere in una interfaccia e si possono specificare dei parametri con cui costruire un oggetto (i costruttori invece non compaiono nelle interfacce).

Se allora si dichiara il metodo come interfaccia può essere implementato da una sottoclasse, mentre il costruttore è specifico di una classe.

FACTORY METHOD definisce un'interfaccia per creare un oggetto, ma delegando la scelta dell'oggetto stesso a sottoclassi. Il pattern può rivelarsi utile quando una classe non è in grado di conoscere a priori il tipo di oggetti da creare o quando si vuole delegare la creazione di un oggetto alle sottoclassi. L'applicazione del pattern consente di eliminare le dipendenze dai tipi concreti utilizzati.



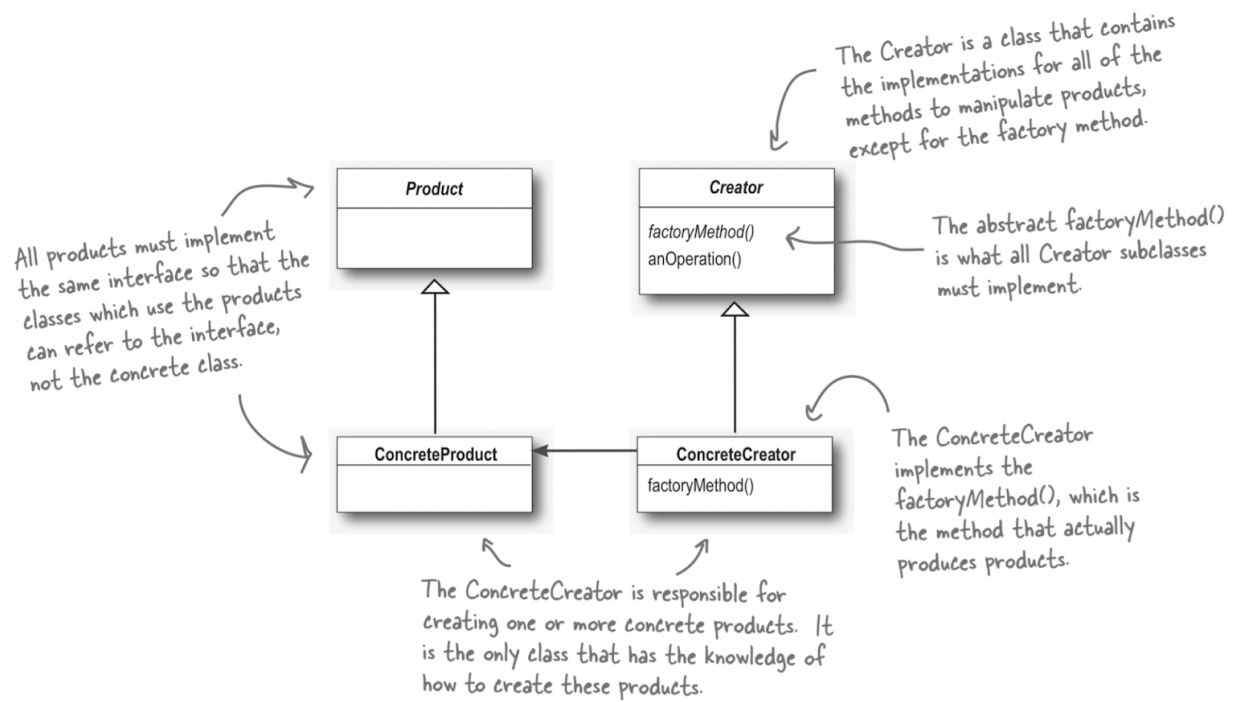
Qua sopra possiamo osservare un diagramma di un'applicazione tipo Word, che può applicare il FACTORY METHOD. L'applicazione è capace di aprire e creare diversi tipi di oggetti. Dentro il concetto del metodo `NewDocument()` si può avere uno schema fisso in cui:

- si crea un'istanza di un documento,
- lo si aggiunge all'insieme dei documenti che l'applicazione gestisce,
- infine si apre il documento.

Application aggrega una serie di **Document**, che è una classe astratta e verrà concretizzata dal tipo **MyDocument**.

Application implementa in qualche modo il template method, la sottoclasse dovrà ridefinire i metodi per farli funzionare.

Quando si crea un nuovo documento si definisce un metodo `CreateDocument()`, al posto di mettere la keyword `new`. Questo metodo è astratto, non si sa ancora come definirlo. La scelta di che tipo di documento creare viene lasciata a **MyApplication**.

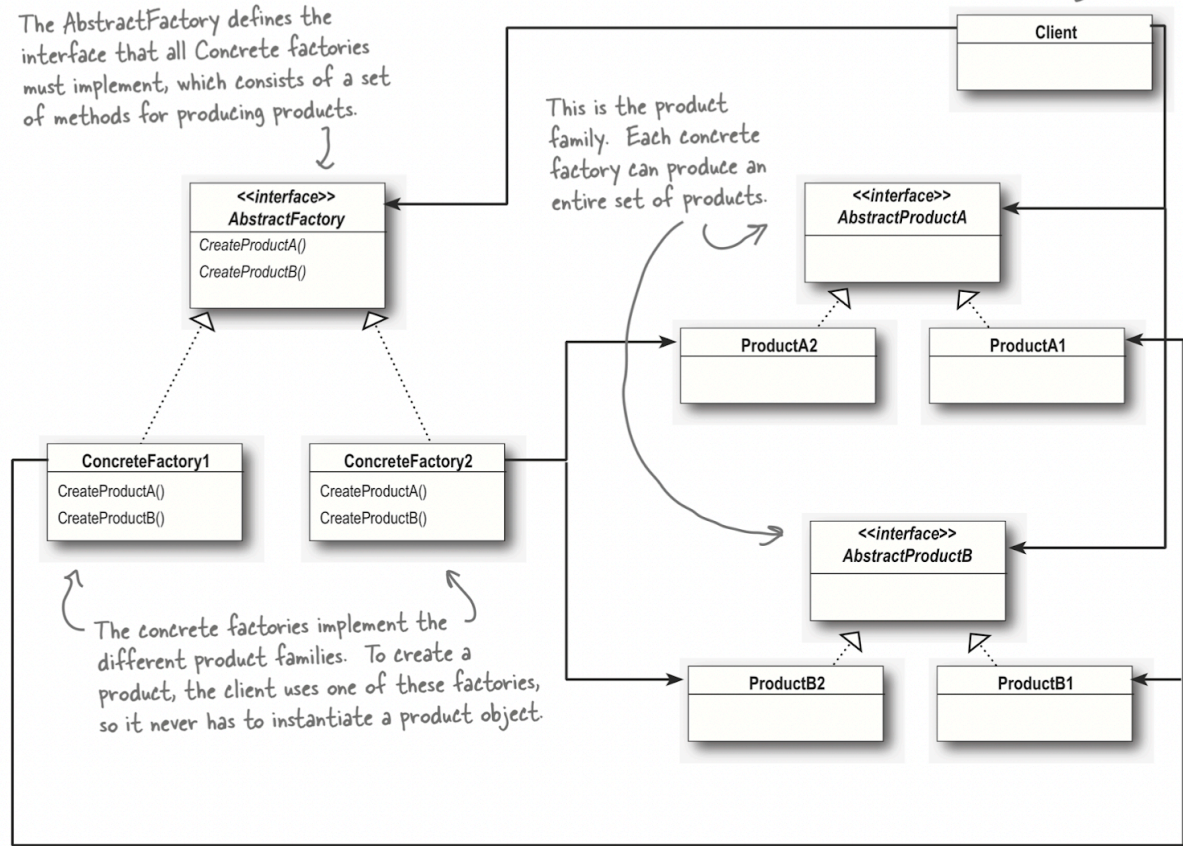


In questo pattern possiamo individuare i seguenti componenti:

- **Creator** dichiara la **Factory** che avrà il compito di ritornare l'oggetto appropriato
- **ConcreteCreator** effettua l'override del metodo della **Factory** al fine di ritornare l'implementazione dell'oggetto
- **Product** definisce l'interfaccia dell'oggetto che deve essere creato dalla **Factory**
- **ConcreteProduct** implementa l'oggetto in base ai metodi definiti dall'interfaccia **Product**.

The AbstractFactory defines the interface that all Concrete factories must implement, which consists of a set of methods for producing products.

The Client is written against the abstract factory and then composed at runtime with an actual factory.

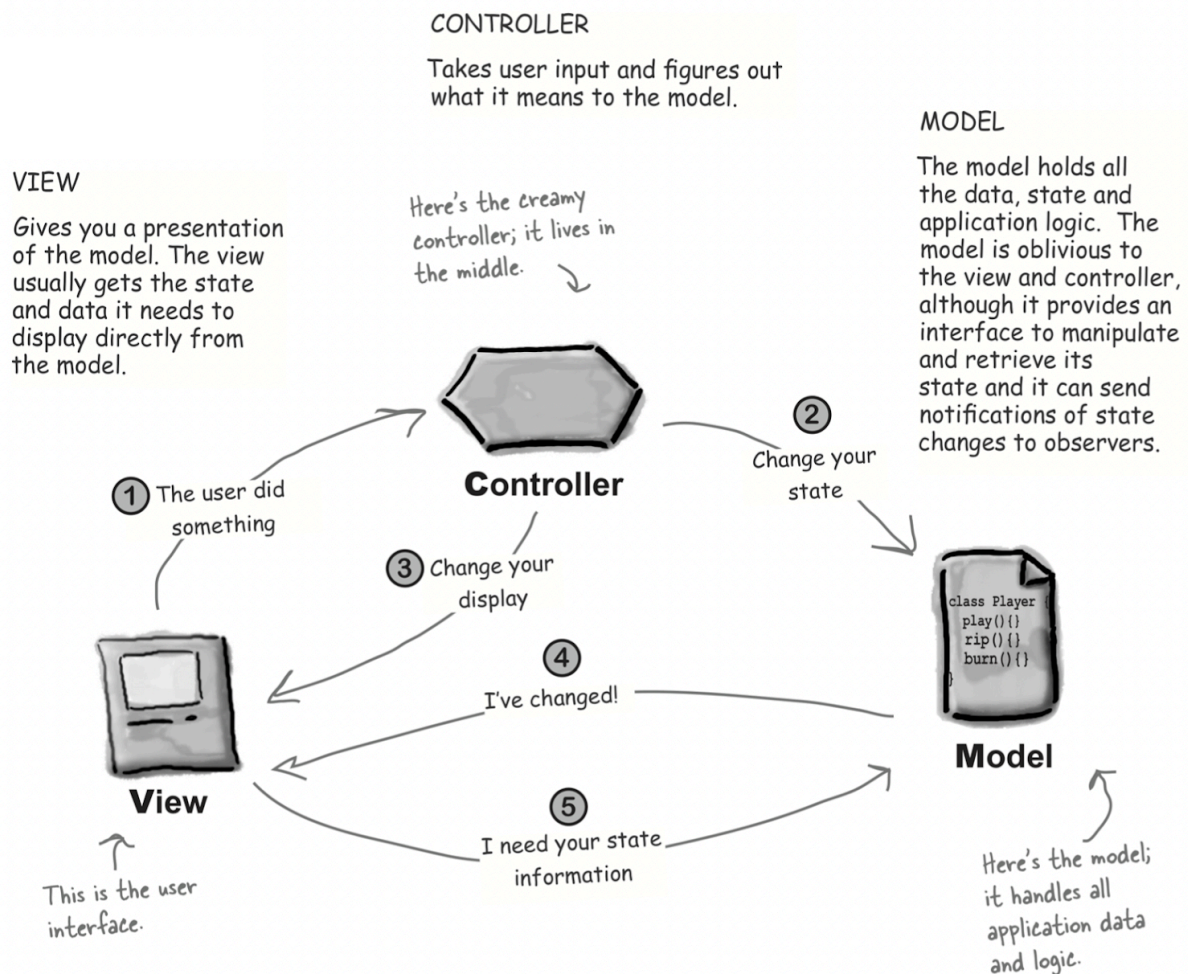


MV*

Il MV* è un macro pattern composto da tre attori, i primi due sono Model e View mentre il terzo può essere: Controller, Presenter o ViewModel. Differiscono su come si collegano le varie parti.

MVC

The **Model View Controller (MVC)** design pattern specifies that an application consists of a data model, presentation information, and control information. The pattern requires that each of these be separated into different objects.

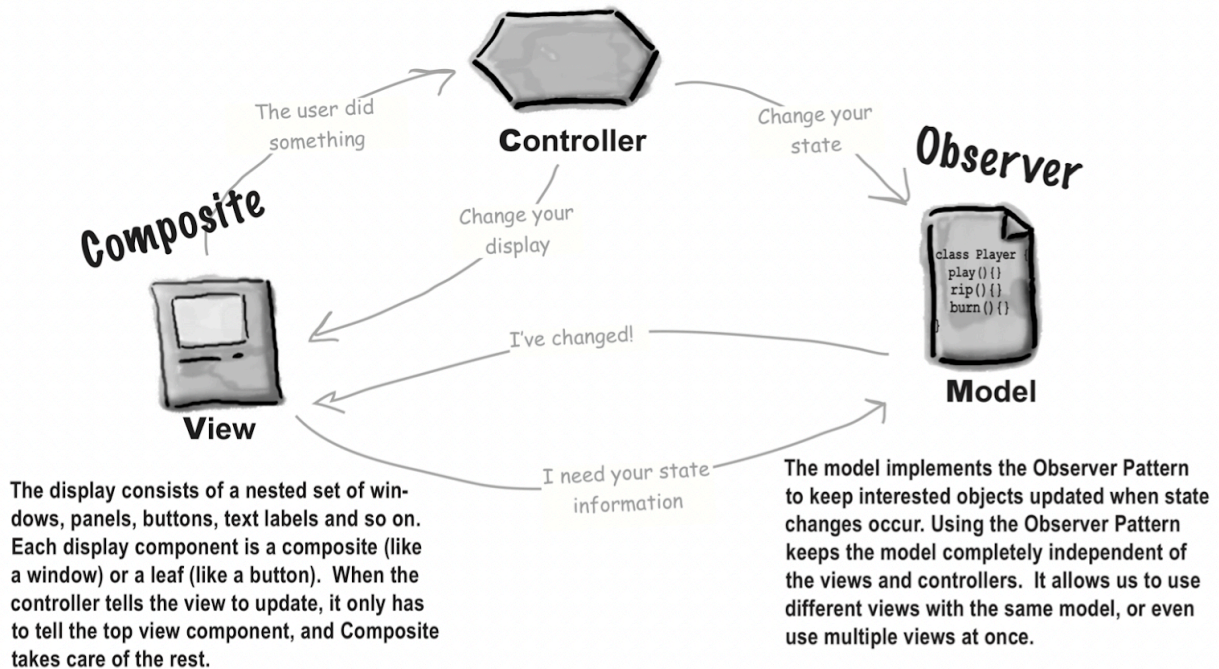


Il Model View Controller (MVC) è composto da tre parti:

- **Model** = Contiene tutti i dati, gli stati e la logica dell'applicazione. Fornisce un'interfaccia per manipolare e raggiungere il proprio stato. Il Model è una classe Subject che viene osservata e deve poter mandare update ai suoi observers, ovvero le Views, spesso infatti in un MVC ci sono diverse viste.
- **View** = Dà una rappresentazione del Model e lo osserva. L'utente interagisce con un MVC attraverso una View.
- **Controller** = Prende l'input dell'utente dalla View e lo manda al Model nel formato opportuni. Per ogni View esiste un Controller.

Strategy

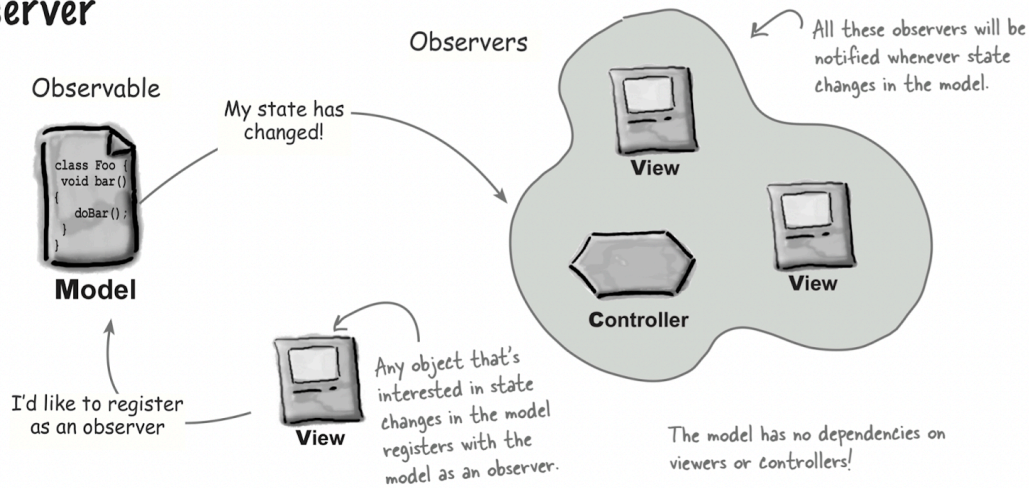
The view and controller implement the classic Strategy Pattern: the view is an object that is configured with a strategy. The controller provides the strategy. The view is concerned only with the visual aspects of the application, and delegates to the controller for any decisions about the interface behavior. Using the Strategy Pattern also keeps the view decoupled from the model because it is the controller that is responsible for interacting with the model to carry out user requests. The view knows nothing about how this gets done.



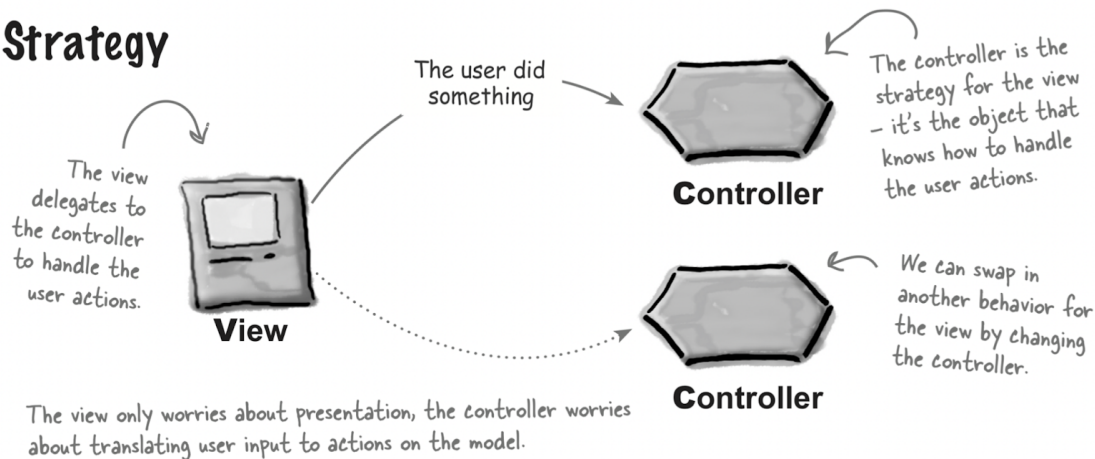
I dialoghi tra i vari attori di questo macro pattern possono essere visti come pattern:

- **STRATEGY**, ovvero si implementa la strategia con cui gestire quel tipo di interazione tramite il controller. Il controller mette a disposizione un metodo standard che la vita chiama, e verranno compiute cose diverse a seconda dell'implementazione del metodo.
- **OBSERVER**, infatti tramite inversione di controllo il controllore si mette in attesa che avvenga un evento da parte dell'utente. In questo caso la vista osserva il model e il controller osserva la vista. La vista può essere collegata al modello secondo lo schema observer. In questo caso la vista osserva il modello.
- **COMPOSITE**, infatti la vista viene definite creando degli oggetti di tipo grafico e questi sono combinabili tra di loro.

Observer



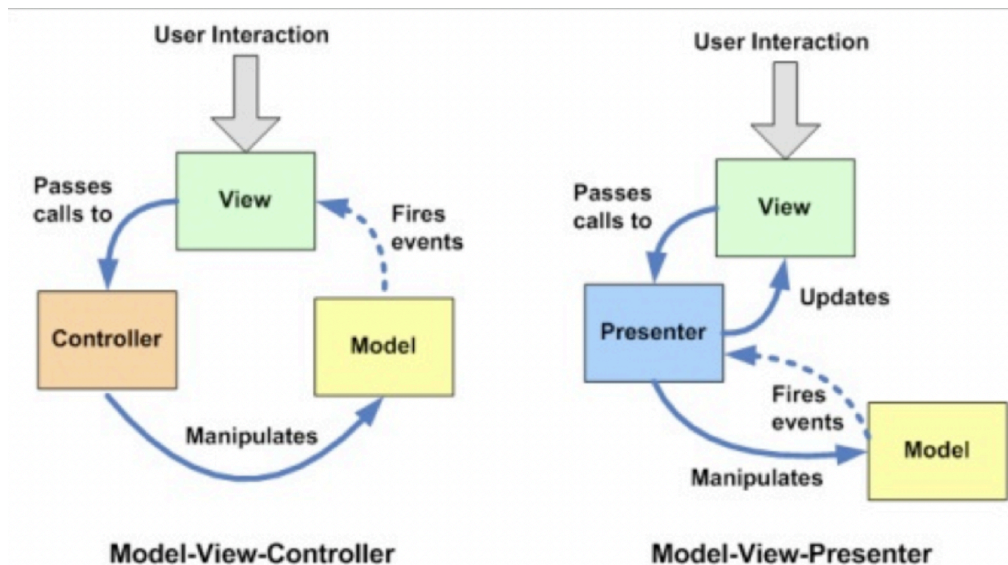
Strategy



Composite



Aver separato il Controller dalla View permette maggior testability. Infatti il testing di interfacce è, inefficiente e scomodo. Isolando quindi la parte di interfaccia si risolvono molti problemi. Per farlo in modo completo esiste una versione alternativa del MVC, il **MVP**, ovvero Model View Presenter. Nel MVP l'interazione tra View e Model non è più diretta: quando il Model cambia comunica il suo cambiamento non alla View, ma al Presenter, rendendo così la testability ancora meno complicata.



Esiste anche un'altra alternativa al MVC il **MVVM**, ovvero Model View ViewModel. In questo macro pattern la View gestisce gli eventi, e il ViewModel dà un valore semantico al Model.

Le **GUI**, ovvero le Graphical User Interface, spesso si basano sul macro pattern MVC. Le GUI sono composte da:

- **framework code**
 - component library → componenti riutilizzabili che implementano elementi delle interfacce (widget): bottoni, menù, slider...
 - application skeleton → controlla gli elementi di sistema, disegna gli elementi dell'interfaccia
- **application code** = il codice che estende lo scheletro e di conseguenza realizza la applicazione

Nelle GUI:

- si lancia il framework
- che richiama funzioni di inizializzazione del codice applicativo
- poi esegue un event loop
- che richiama il codice applicativo che di volta in volta gestisce gli eventi

L'Application Code deve definire:

- Una parte statica → component graph (COMPOSITE, DECORATOR, ...)
 - oggetti visibili (Bottoni, CheckBox, Menù, ...)
 - contenitori (Layout, Region, Pannel, Scene...)
- Una parte dinamica → event handling code (COMMAND, STRATEGY, OBSERVER, ...)
 - mappatura eventi-interazioni-componenti (focus, click, drag, ...)
 - logica applicativa: evoluzione del sistema

