

# Design Patterns

# Design Patterns

## Pattern Adapter

PATTERN  
STRUTTURALE

Il pattern Adapter risulta utile quando interfacce di classi diverse devono comunque poter comunicare tra loro. Possiamo avere diversi casi:

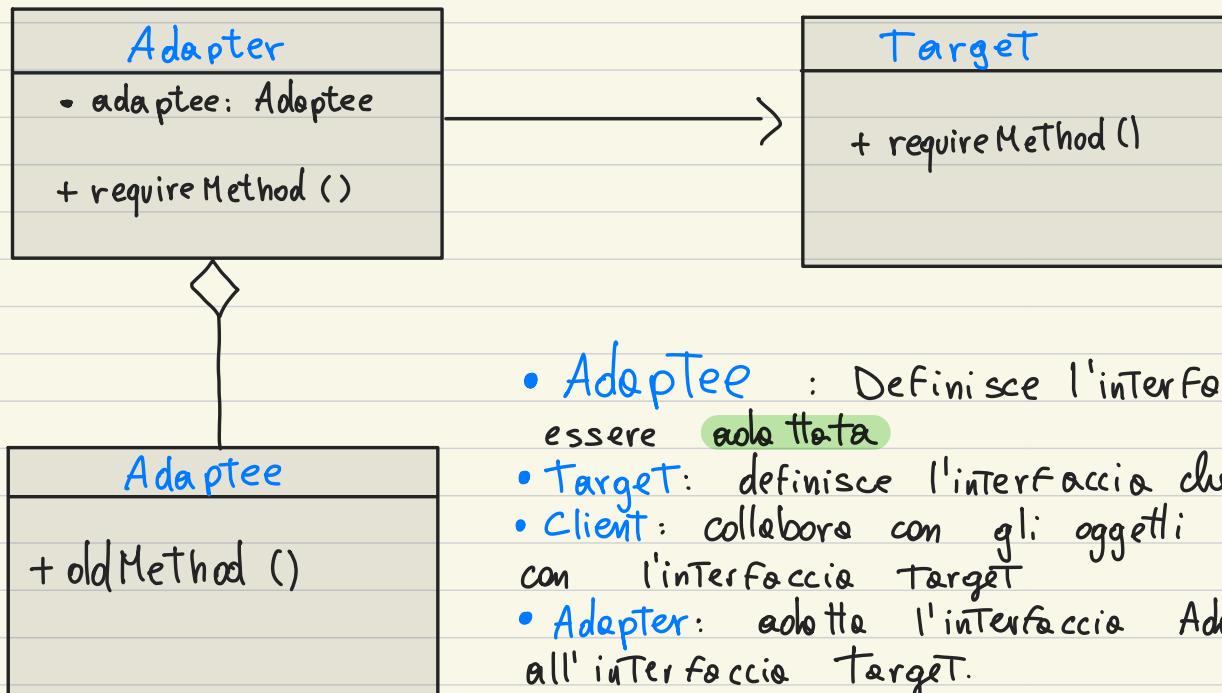
- L'uso di una classe esistente che presenta un'interfaccia diversa da quella desiderata.
- Potremmo dover scrivere una classe senza poter conoscere a priori le altre classi con cui essa dovrà operare.

Questo pattern può inoltre essere usato quando vogliamo che l'invocazione di un metodo di un oggetto da parte del client avvenga solo in maniera indiretta.

## Struttura

Questo pattern può essere basato su classi, usando l'ereditarietà multipla per ereditare diverse interfacce con il meccanismo dell'ereditarietà.

Si può anche includere l'oggetto sorgente nell'implementazione dell'adpter.



- **Adaptee**: Define l'interfaccia che deve essere adattata.
- **Target**: definisce l'interfaccia che usa il client.
- **Client**: collabora con gli oggetti in conformità con l'interfaccia Target.
- **Adapter**: adatta l'interfaccia Adaptee all'interfaccia Target.

# Pattern Adapter nell'applicazione

Nell'app è stato usato il pattern adapter diverse volte in modo da far comunicare un database (Firebase o MongoDB) con le varie views, in particolare con le **RecyclerView**.

Prendiamo come esempio il Pattern Adapter che gestisce i Reports.

In questo caso, la classe **Report Adapter** funge da classe **Adapter**.

Questa classe (seguendo il pattern) serve a "far comunicare" due componenti che solitamente non sarebbero compatibili.

Nel caso dell'app, però, non siamo in presenza dell'applicazione letterale del pattern Adapter. Quello che le classi Adapter fanno in Android è semplicemente prendere i dati da **un model** e metterli all'interno di una **View**.

Questo tipo di comportamento è più simile al pattern **MVP - Model View Presenter**.

## Model View Presenter Pattern

MVP è una derivazione del pattern **MVC**, ed è maggiormente usato per costruire interfacce utente. In questo pattern, il **Presenter** assume la funzionalità del "middle man".

### Vantaggi

Pur lasciando invariata la logica, è possibile cambiare lo **Strato di presentazione**; questo potrebbe essere utile per passare da sistemi ad UI Desktop ad Applicazioni mobile.

### Struttura

#### Componenti

- **Model**: è lo strato di data access per la gestione dei dati. Potrebbe essere visto come un'interfaccia responsabile di accedere alle API connesse con un database locale o remoto.
- **View**: Solitamente non ha logica applicativa, ma solo visuale; lavora con il **Presenter**.
- **Presenter**: Lavora come intermediario tra View e Model: prende i dati dal Model li elabora e li restituisce alla View.

# Pattern Singleton

PATTERN  
CREAZIONALE

## Key Features

Il pattern Singleton fornisce una sola istanza di una determinata classe singleton, e fornisce per essa un solo punto di accesso.

## Come Funziona?

Dobbiamo strutturare la classe in maniera tale che essa possa essere istanziata arbitrariamente, e soprattutto in modo che essa possa essere acceduta da qualunque contesto.

Per ottenere questo tipo di risultato, dobbiamo tenere a mente due aspetti fondamentali del pattern: Il costruttore della classe deve essere **privato**, e che la classe venga resa **non ereditabile**.

Possiamo quindi creare un metodo **statico** che ci permetta di recuperare l'istanza della classe se presente, o istanziarla per la prima volta. Da questo momento in poi, verrà sempre usata la stessa istanza della classe.

## Pattern MVVM - Model View Viewmodel

E' un pattern software architettonale; questo pattern è una variante del pattern **Presentation Model Design**. Il pattern astrae una vista usando una classe **View model**.

## Componenti del Pattern

**Model**: Rappresenta il punto di accesso ai dati; potremmo infatti avere bisogno di avere una o più classi che leggono dal Database o da una qualsiasi sorgente.

**View**: la View rappresenta la vista dell'applicazione, la UI

**ViewModel**: è il punto di incontro tra la View e la Model: I dati ricevuti da Model sono elaborati per essere presentati e passati alla View.

## Interazione Tra i Livelli

1. L'utente interagisce con la View

2. La **variazione di Stato** è comunicata alla ViewModel, oppure viene eseguito un metodo specifico; in android possiamo associare dei metodi specifici a seguito di un evento su un elemento visuale.

3. Come risposta al cambio di stato, la **ViewModel** esegue della logica tramite **Model** ed aggiorna il proprio stato.

4. Il nuovo stato di ViewModel si riflette sulla **View**.

## Pattern MVC

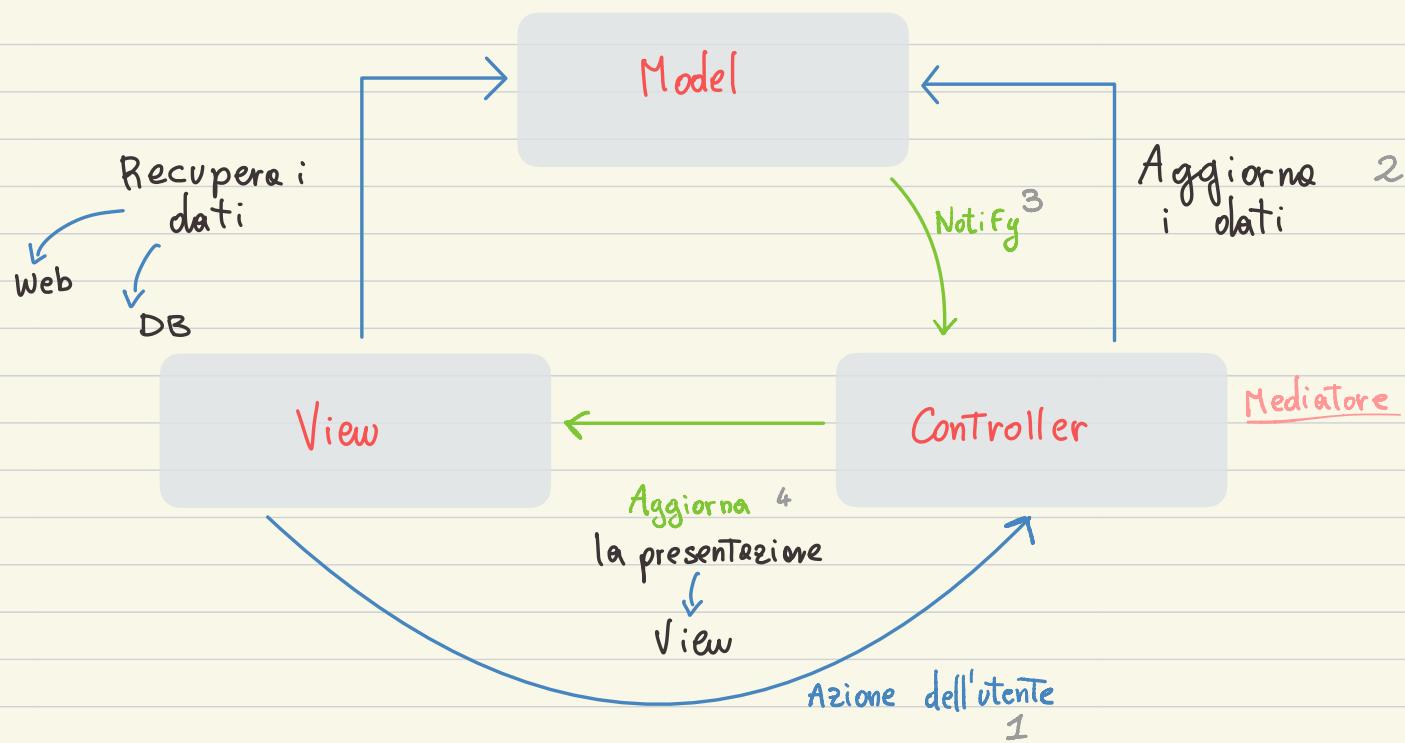
Questo pattern suggerisce di dividere il codice in 3; quando si crea una nuova classe, lo si deve identificare (e salvare nel package corrispondente) in uno di questi 3 layers:

**Model**: Questo componente salva i dati dell'applicazione. Non è a conoscenza dell'interfaccia. Questo componente è incaricato di comunicare con il database e network layers.

**View**: È la UI che contiene i componenti che sono visibili sullo schermo. Più precisamente, fornisce la visualizzazione dei dati salvati nello classe model ed offre interazione all'utente.

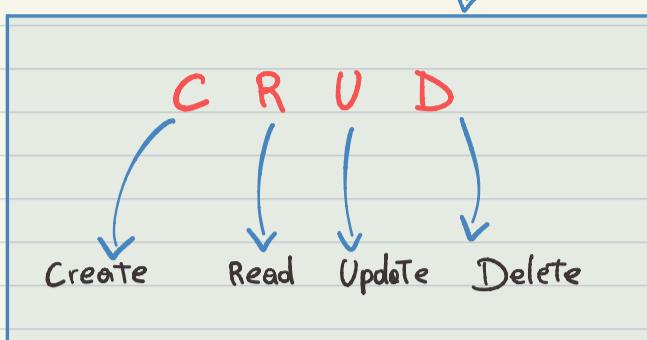
**Controller**: questo componente stabilisce la relazione tra la View e Model; contiene il nocciolo della logica dell'applicazione e viene informata del comportamento dell'utente ed aggiorna la Model quando serve.

Possibile ordine



## Pattern DAO - Data Access Object

Il pattern DAO è un pattern **Strutturale** che ci permette di isolare l'applicazione dallo **Strato di persistenza**. Per fare questo usiamo una **API Astratta**, che nasconde all'applicazione tutta la complessità dell'esecuzione di **Operazioni CRUD**.



Le operazioni CRUD vengono eseguite nei meccanismi sottostanti di storage; il loro occultamento permette ad entrambi i layers di evolversi **separatamente** senza che l'uno sia a conoscenza dell'altro.

### DAO in Android

Quando usiamo la libreria **Room** per salvare i dati della nostra applicazione, interagiamo con i dati salvati definendo dei **DAO**. Ogni DAO include dei metodi che offrono un **accesso astratto** al database. A tempo di compilazione, Room genera automaticamente **implementazioni** dei DAO che abbiamo definito.

Usando i DAO per accedere al database, possiamo preservare la **Separazione degli interessi**.

I DAO, inoltre, rendono più semplice la **simulazione** di accessi al database durante le fasi di **Testing** dell'applicazione.

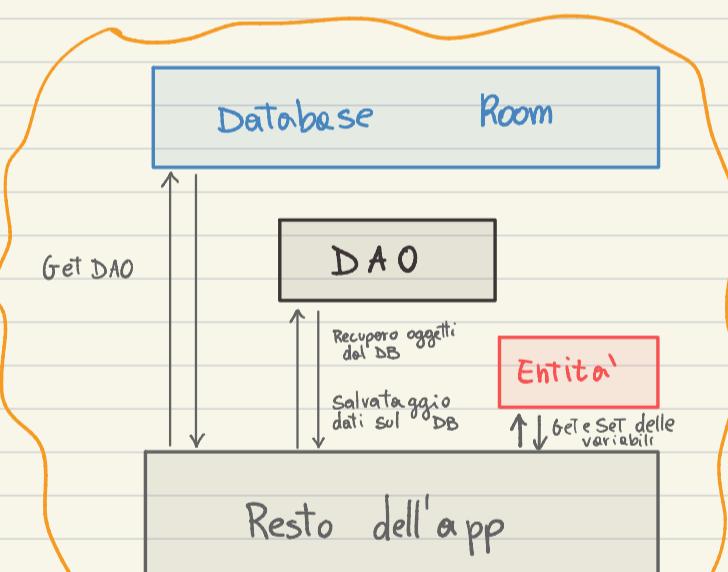
### Separation of concerns

La separazione degli interessi è il principio più importante da seguire. Non dobbiamo scrivere tutto il codice all'interno di una **Activity** o **Fragment**; queste classi devono contenere solo la logica che gestisce la UI e le interazioni con il SO.

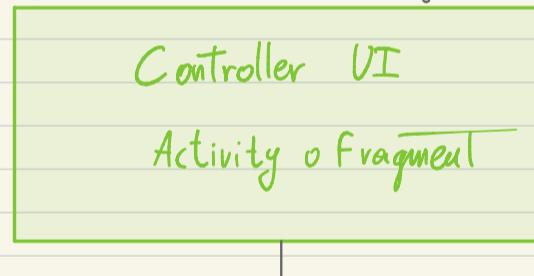
### Anatomia di un DAO

Possiamo definire un DAO sia come classe che come interfaccia. Per utilizzarli basiliari, è **meglio usare un'interfaccia**. La cosa principale da fare, è annotare il dao con il tag **@Dao**. Questo tipo di oggetti **Non hanno proprietà**, ma definiscono dei metodi per interagire i nostri dati al Database.

### Come funziona?



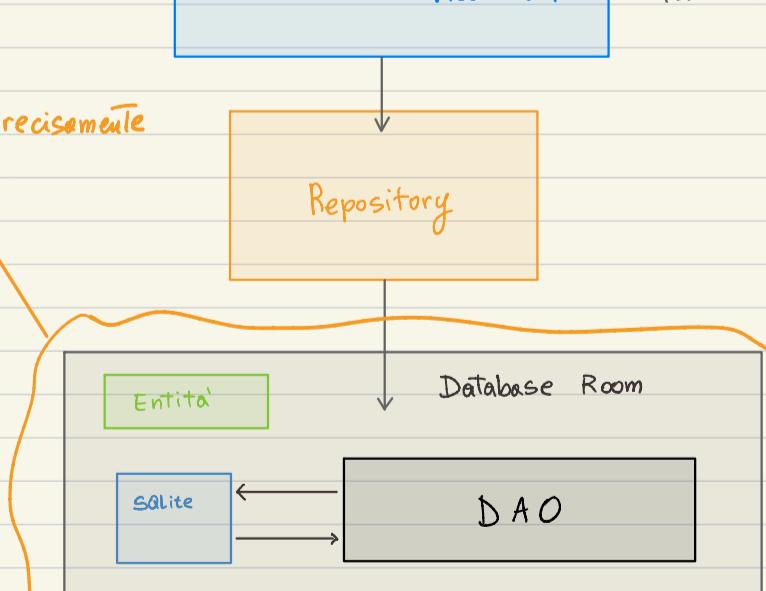
Rappresenta tutti i dati ed inoltre gli eventi (tacchi) che si verificano agli handler.



### Persistenza

**Caratteristica** dei dati di un programma di sopravvivere al programma che li ha creati.

Contiene tutti i dati necessari per la UI



Gestisce il database locale usando gli oggetti definiti

## Pattern creazionali

I pattern creazionali effettuano un'astrazione del processo di creazione ed istanziamento di un oggetto. permettono di rendere un oggetto indipendente da come esso è creato, composto e rappresentato.

Ci sono due temi ricorrenti in questo tipo di pattern: In primo luogo **incapsulano la conoscenza di quale classe concreta il sistema deve usare**, in secondo luogo **nascondono come le istanze di queste classi vengono create**. I pattern creazionali ci concedono molta flessibilità in cosa viene creato ed in che modo.

PATTERN  
CREAZIONALE

### Abstract Factory - AKA KIT

Questo pattern consiste nel creare Famiglie di oggetti dipendenti o con una relazione senza specificare le loro classi specifiche possiamo insomma includere un gruppo di **fabbriche individuali** che hanno un tema comune.

Partecipanti:

- **Abstract Factory**: dichiara una interfaccia per operazioni che creano prodotti astratti (oggetti astratti)
- **Concrete Factory**: Implementa le operazioni in modo da creare oggetti
- **Abstract Product**: dichiara un'interfaccia per un tipo di oggetto prodotto
- **Concrete Product**: Implementa l'interfaccia Abstract Product

PATTERN  
CREAZIONALE

### Builder

Semplifica la creazione di oggetti in una maniera pulita ed affidabile. È particolarmente utile quando abbiamo delle **classi model** aventi molti parametri: possiamo rendere alcuni di questi **optional o richiesti** in modo da non forzare l'utente ad usare un **ordine specifico**. Come quello presente nel costruttore della classe. Usando questo pattern possiamo creare una elegante **catena di metodi** per la creazione di un oggetto:

```
1 new AlertDialog.Builder(this)
2   .setTitle("Design Patterns")
3   .setMessage("Builder is awesome")
4   .create();
```

Per creare l'oggetto usando il pattern, ci basta creare diversi metodi Set (per ogni variabile), dove ogni metodo Set ritorna il **context** della classe che stiamo costruendo; ci basta quindi ritornare **this** per ogni metodo. Infine, ci serve un metodo **creat()** che non fa altro che ritornare una nuova istanza della classe appena costruita: **return new User(this);**

PATTERN  
CREAZIONALE

### Factory Method Pattern

Definisce un'interfaccia per creare oggetti, ma lascia decidere alle sottoclassi quali classi istanziare.

#### Esempio

Poniamo il caso di avere degli oggetti **Baguette**, **Brioche**, ...

##### 1) Interfaccia Bread

Le baguette sono ovviamente Pane

+ name()  
+ Calories()

##### 2) Classe Baguette, Brioche...

la classe implementa l'interfaccia

Bread e sovrscrive name() e calories()

##### 3) Classe BreadFactory

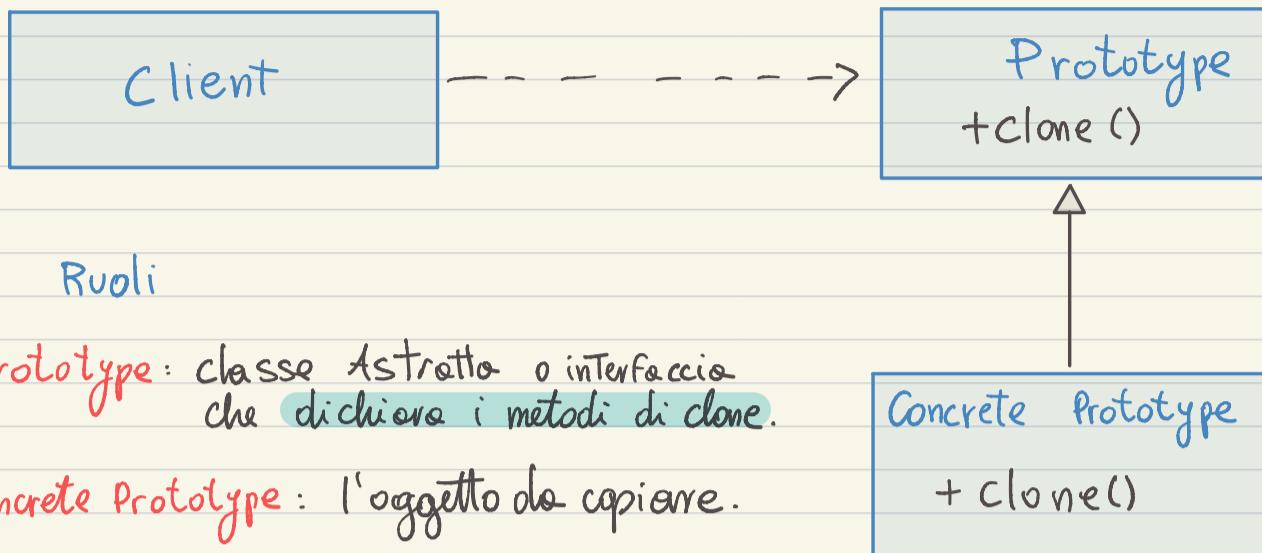
Questa classe ritorna un oggetto di tipo **Bread** (interfaccia), e tramite una stringa passata come parametro puo' creare diversi tipi di pane come Brioche, Baguette, ecc.

Il pattern Factory Method è chiaramente comodo quando dobbiamo creare degli oggetti accomunati dalle stesse caratteristiche, che vanno gestiti allo stesso modo.

Nella ci vieta, poi, di definire un comportamento diverso per ogni classe.

## Prototype Pattern

Specifica i tipi di oggetti creati con istanze prototipo e crea nuovi oggetti copiando questi prototipi. L'oggetto esistente (prototipo) crea un nuovo oggetto con le stesse proprietà interne del prototipo, replicando il prototipo stesso. Il nocciolo di questo pattern è il metodo di clonazione, che realizza le copie degli oggetti.



**Prototype**: classe astratta o interfaccia che dichiara i metodi di clone.

**Concrete Prototype**: l'oggetto da copiare.

**Client**: Dove il pattern verrà usato.

### Utilizzo nel Client

Per usare il pattern nel client e creare oggetti a partire da un oggetto prototipo, ci basterà creare l'oggetto prototipo iniziale, e chiamare il suo metodo `clone()` quando vorremo creare un altro oggetto uguale al primo.

### Possibili problemi

Quando usiamo questo Pattern dobbiamo stare attenti alle modifiche che facciamo agli oggetti derivati dal prototipo: poniamo il caso di avere, all'interno del prototipo, un ulteriore oggetto come variabile di istanza; se modifichiamo questo oggetto nell'oggetto derivato dal prototipo, avremo un cambiamento anche nel prototipo stesso! Se invece modifichiamo una variabile basileare come un intero, questa viene modificata solo sul nuovo oggetto.

E' in questo caso che parliamo di copia per riferimento e copia per valore. Possiamo copiare un valore intero direttamente, ma un oggetto è un oggetto di un tipo specifico, e di conseguenza è semplicemente un riferimento ad un oggetto reale `Object`!

Possiamo quindi copiare semplicemente il riferimento del valore dell'oggetto, ed in questo caso abbiamo una **Copia Superficiale**.

Per avere una **Deep copy** dobbiamo creare un nuovo oggetto identico ed assegnare il riferimento del nuovo oggetto nel campo `object` (variabile di istanza) nell'oggetto appena creato dal prototipo.

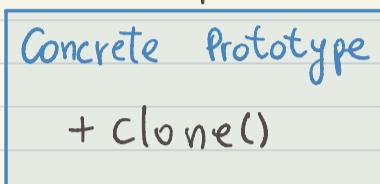
### Come risolviamo?

Come suggerisce il nome "Deep Copy" ci basta "andare in profondità". Tutti gli oggetti usati dal prototipo, devono anch'essi implementare `Cloneable`. In questo modo possiamo modificare i campi del prototipo senza preoccuparci che questi cambiamenti si propaghino ad altri oggetti.

### Quando usarlo?

- Se abbiamo bisogno di molte risorse per inizializzare una classe, possiamo usare il pattern per evitare questi consumi
- Se per creare un oggetto abbiamo bisogno di effettuare operazioni tediose che possono essere evitate usando il pattern

Non deve essere definito, siccome la copia è un'operazione comune, l'`interfaccia Cloneable` è fornita da Java per supportare l'operazione di cloning.



Ci basterà quindi creare la nostra classe concreta ed implementare l'interfaccia `Cloneable`. Dovremo quindi implementare il metodo `clone()` dove chiamiamo `Super` per la clonazione.

`Super.clone()`

### Vantaggi

- Possiamo risolvere problemi di eccessivo consumo di risorse durante la creazione di oggetti pesanti
- La **copia protettiva** può evitare che chiunque esterno modifichi l'oggetto, assicurando che l'oggetto sia di sola lettura

### Ed in Android?

In Android `Intent` implementa `Cloneable`, anche se invece di chiamare `super`, implementa il metodo `clone()` semplicemente con `new Intent(this)`

# Patterns Strutturali

I pattern strutturali riguardano il modo in cui le classi ed oggetti sono composti in modo da creare strutture più grandi. Le classi che adottano questi patterns usano l'**'ereditarietà'** per comporre interfacce o implementazioni.

PATTERN  
STRUTTURALE

## Pattern Adapter

Questo pattern è già stato visto all'inizio di questi appunti, ma in questo lozione farò un esempio per comprendere meglio il pattern.

Questo pattern viene usato quando si ha la necessità di rendere compatibili due interfacce che non lo sono, senza modificare il loro codice.

Una tipica soluzione è quella in cui un **client** usa oggetti che implementano una determinata interfaccia, ma questa interfaccia è diversa tra i vari oggetti che devono comunicare.

Consideriamo il caso in cui abbiamo una classe **ShapeCalculator** che riceve in input oggetti che implementano l'interfaccia **Shape**, e ne calcola area e perimetro.

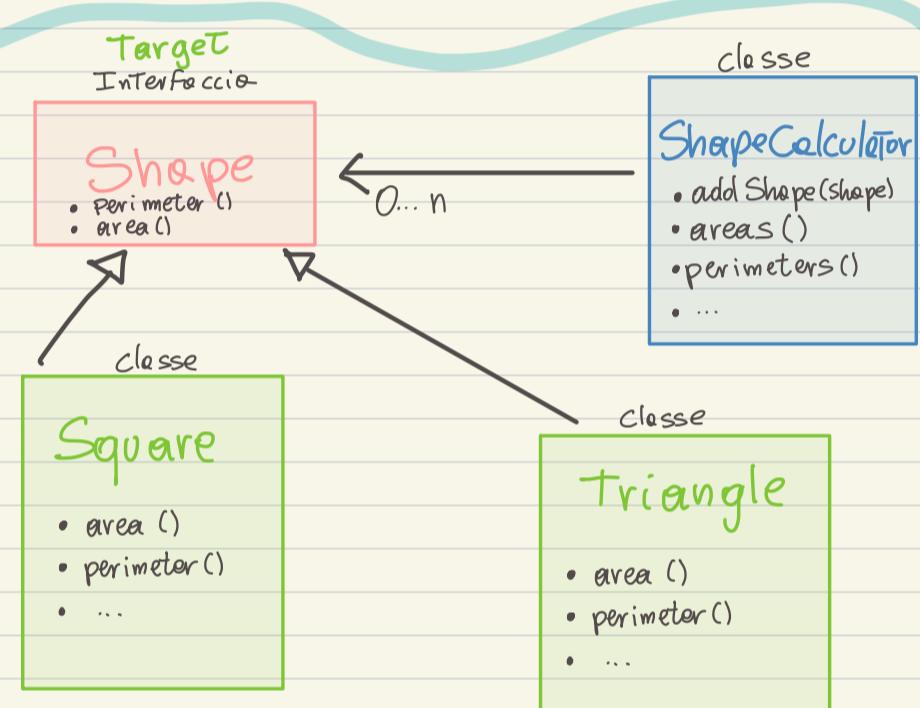
```
public class ShapeCalculator {
    1 2
    3 List<Shape> shapes = new ArrayList<Shape>();
    4 public void addShape(Shape shape) {
        shapes.add(shape);
    }
    5
    6
    7
    8 public void areas() {
        shapes.stream().forEach(shape -> System.out.println( shape.getClass().getSimpleName() + " : " + shape.area() ) );
    }
    9
    10 public void perimeters() {
        shapes.stream().forEach(shape -> System.out.println( shape.getClass().getSimpleName() + " : " + shape.perimeter() ) );
    }
    11
    12
    13 public static void main(String[] args) {
        ShapeCalculator calculator = new ShapeCalculator();
    14
        calculator.addShape( new Triangle() );
        calculator.addShape( new Square() );
    15
        calculator.perimeters();
        calculator.areas();
    16    }
}
```

le classi **Triangle** e **Square** implementano l'interfaccia **Shape**, la quale espone due metodi che servono a calcolare **Area** e **perimetro**.

Struttura Iniziale

Supponiamo poi che ci venga fornita una API con altre figure geometriche già implementate e che vogliamo usare nel nostro programma.

Questa API contiene delle figure che però implementano un'interfaccia diversa, seppur con la stessa struttura: essa espone sempre i metodi **area()** e **perimeter()**.



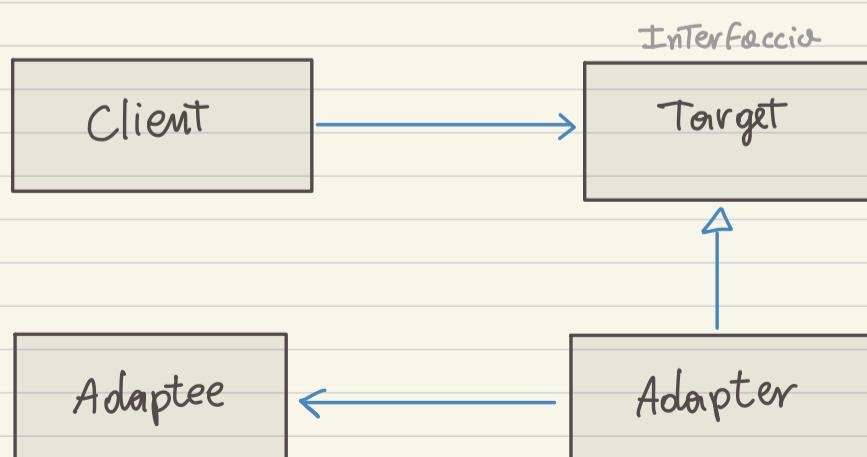
## Ruoli nel pattern

**Target**: Specifica interfaccio usato dal client. In questo esempio viene usato **Shape**.

**Adaptee**: È l'interfaccia esistente che necessita di essere adattata e resa compatibile con l'interfaccia Target. In questo esempio è **GeometricShape**.

**Adapter**: È la classe che realizza il pattern e rende le due interfacce compatibili.

**Client**: Il client che usa esclusivamente gli oggetti che implementano Target. Nel nostro caso usiamo la classe **ShapeCalculator**.

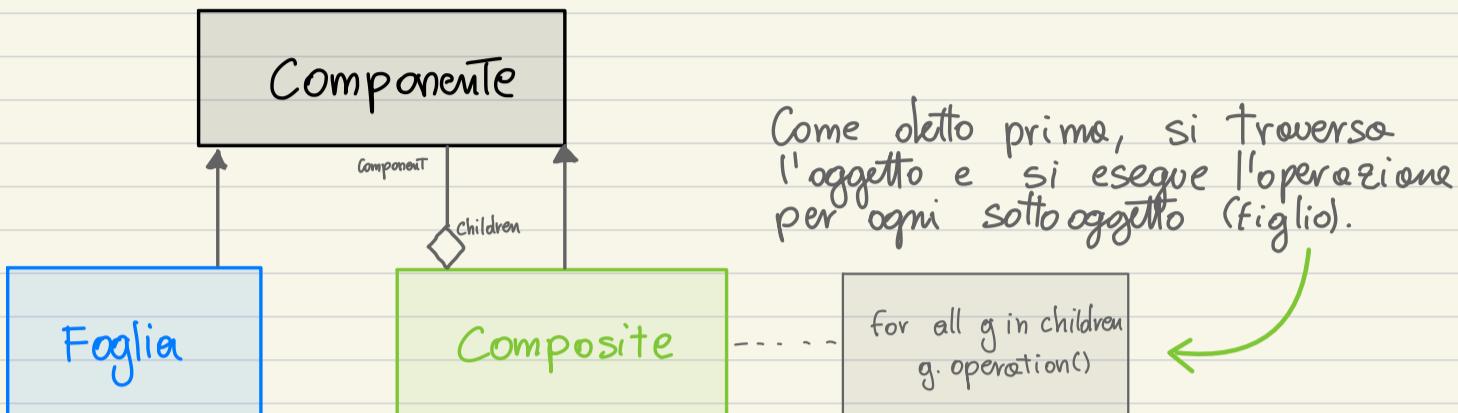


## Realizzazione

La classe **Adapter** implementa l'interfaccia **Target** (**Shape**) e fa riferimento ad un oggetto di tipo **Adaptee** (**GeometricShape**). Implementa tutti i metodi dell'interfaccia e seguendo la conversione necessaria per soddisfare i requisiti dell'interfaccia usando i metodi esposti da **Adaptee**.

## Pattern Composite

Supponiamo di avere un oggetto chiamato **Picture**, che è un oggetto grafico. Questo obj potrebbe consistere di altre immagini in modo ricorsivo così come oggetti primitivi come Line o Rectangle. tutti questi oggetti che **compongono** l'immagine, conformano alla stessa interfaccia grafica. Per questo motivo, al **client**, una parte di oggetto **appare allo stesso modo** dell'immagine completa che consiste di altri oggetti. Per "dipingere" l'oggetto, il client semplicemente traversa l'intera immagine e **disegna le varie parti**.



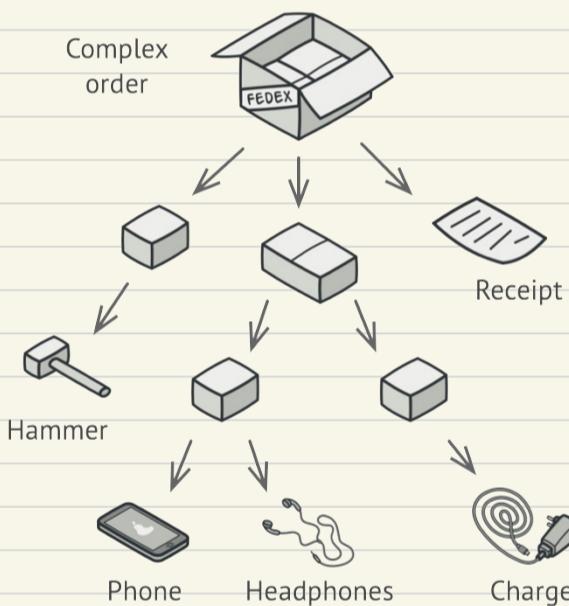
### Partecipanti

**Component**: Di chiara l'interfaccia per gli oggetti ed aiuta a gestire gli oggetti con operazioni come add, remove, ecc.

**Foglia**: rappresenta gli oggetti foglia non aventi figli, ovvero gli elementi primitivi come int, long...

**Composite**: Definisce il comportamento per componenti aventi figli; inoltre salva il children.

**Client**: Con l'aiuto dell'interfaccia Component per manipolare i diversi oggetti.



Supponiamo di avere un Sistema di ordini che usa le classi:

- **Products**: Dei prodotti, articoli

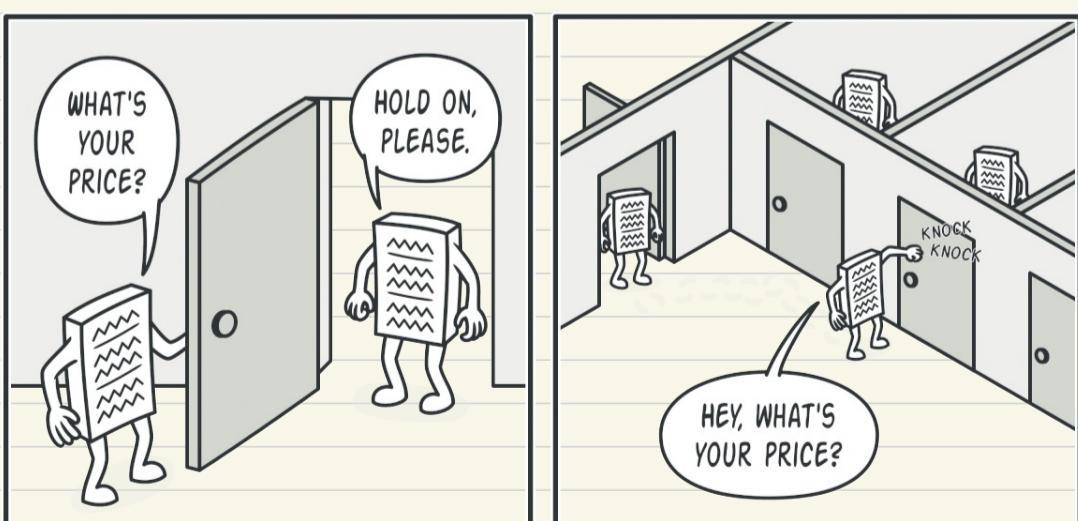
- **Boxes**: Scatole che possono contenere prodotti o anche altre scatole.

Ogni ordine potrebbe contenere dei prodotti semplici o scatole piene di altre cose (prodotti o altre scatole); come facciamo a determinare il prezzo complessivo dell'ordine?

### Soluzione

Siamo ingegneri quindi scartiamo subito la possibilità di usare un approccio iterativo. Il pattern ci suggerisce di lavorare con **Products** e **Boxes** tramite un'interfaccia comune che dichiara un metodo per calcolare il prezzo Totale.

Per calcolare il prezzo di un prodotto il gioco è semplice, basta ritornare il prezzo del prodotto. Per un Box, invece, dobbiamo visitare ogni elemento, chiedere il suo prezzo e ritornare il prezzo totale per quel Box.

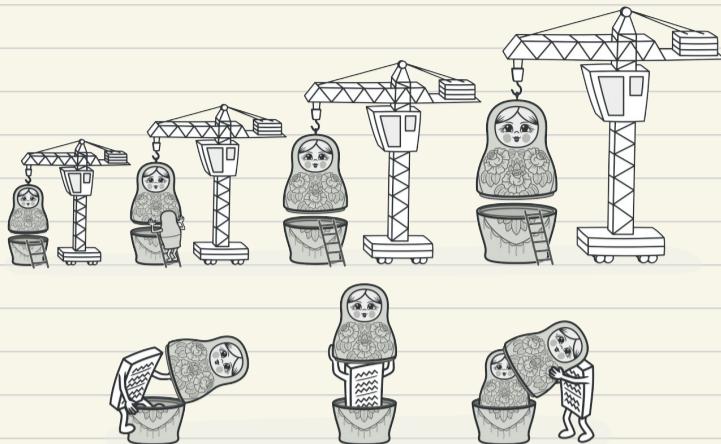


Il bello di questo pattern, è che non abbiamo bisogno di sapere se un determinato item è un Box o Product, ma possiamo trattarli tutti allo stesso modo grazie all'interfaccia comune.

Quando chiamiamo il metodo, gli oggetti stessi passano la richiesta in fondo all'albero.

## Pattern Decorator AKA Wrapper

Questo è un pattern strutturale che permette di aggiungere nuovi comportamenti ad oggetti, ponendo questi oggetti all'interno di speciali **classi wrapper** contenenti i comportamenti.



Immaginiamo di lavorare su una libreria di notifiche che permette ad altri programmi di notificare gli utenti di eventi importanti.

La versione iniziale della libreria era basata sulla classe **Notifier** che aveva solo pochi campi, un costruttore ed un solo metodo. L'implementazione base prevedeva l'invio di una mail quando bisognava notificare qualcosa all'utente.

Ad un certo punto ci accorgiamo che l'utente potrebbe volere anche altri tipi di notifiche, come SMS.

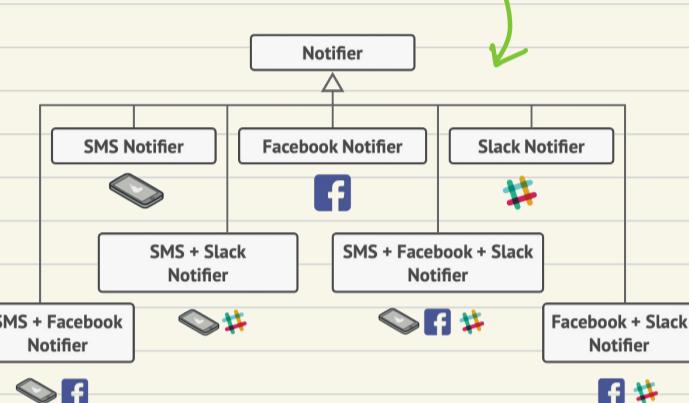
Per fare questo potremmo estendere **Notifier** e porre il nuovo codice in **sottoclasse**. Il problema con questo approccio, è che ora il client deve **istanziare una classe diversa per ogni tipo di notifica**.

### Soluzione

L'ereditarietà non è la risposta. Questo sia perché è statico, non possono quindi alterarne il comportamento, sia perché le sottoclassi possono ereditare solo da una superclasse.

Possiamo usare la **composizione**: un oggetto ha un riferimento ad un altro, e gli delega del "lavoro".

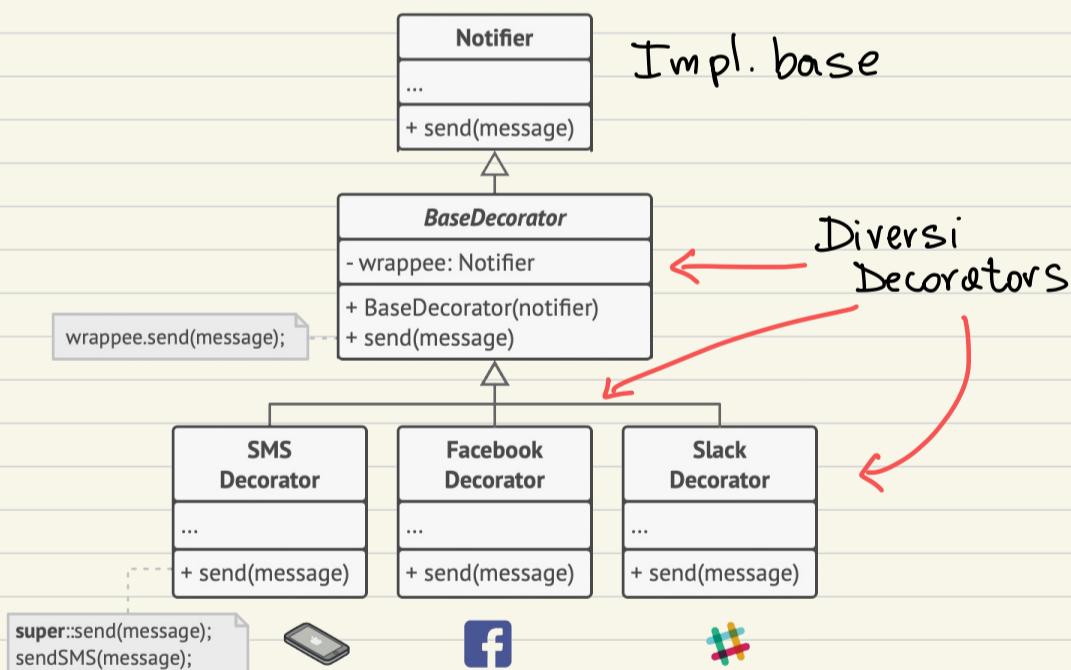
Implementa la stessa interfaccia del target



Un **Wrapper** è un oggetto che può essere collegato ad un **oggetto target**. Il **Wrapper** contiene lo stesso insieme di metodi del target, e delega ad esso tutte le richieste che riceve.

Tornando all'esempio delle notifiche..

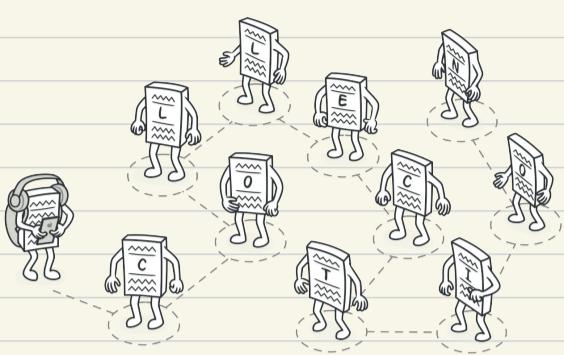
Lo sciamo il comportamento base nello classe basilare **Notifier**, ma spostiamo tutti i metodi di notifica nei **decorators**:



# Pattern Comportamentali

PATTERN  
COMPORTAMENTALI

## Pattern Iterator

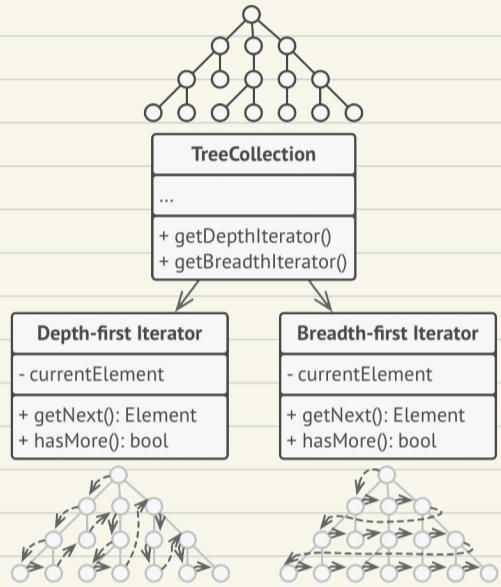


Il pattern iterator ci permette di **traversare** tra gli elementi di una collezione senza esporre la sua rappresentazione sottostante. Questo vuol dire che non ci interessa che tipo di struttura stiamo traversando, ma ognuna di esse verrà trattata allo stesso modo.

Quando ci troviamo davanti ad una collezione basata su di una lista, ci basta iterare su di essa per esaminare ogni elemento. Se, ad esempio, usiamo un Albero, la situazione è più complicata.

### Il pattern

L'idea del pattern è quella di estrarre il comportamento di attraversamento di una collezione, e di porlo all'interno di un oggetto separato chiamato **Iterator**.

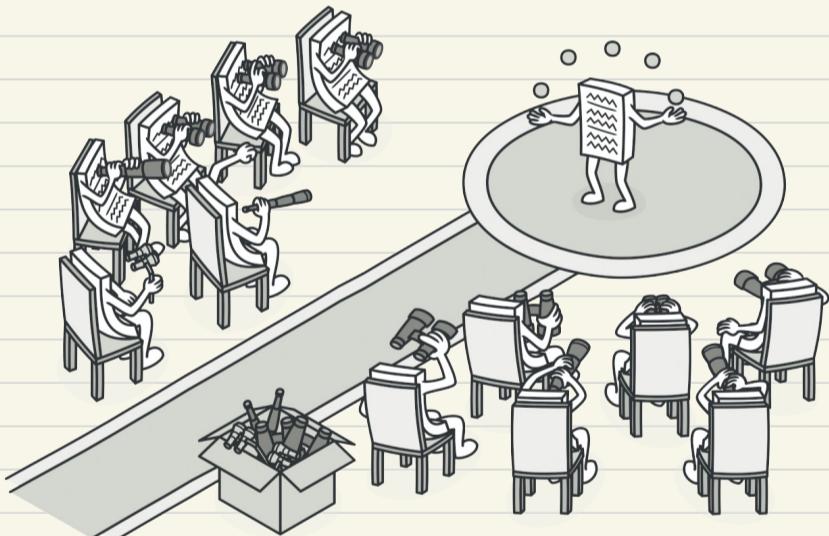


Oltre ad implementare l'algoritmo stesso, un oggetto Iterator incapsula tutti i dettagli di iterazione, come la **posizione corrente** o quanti elementi mancano alla fine.

Tutti gli iterators devono implementare la stessa **interfaccia**. Questo fa sì che il codice scritto sia compatibile con qualsiasi collezione o qualsiasi algoritmo di attraversamento.

Quando abbiamo bisogno di creare un nuovo tipo di attraversamento, ci basta creare una nuova classe iterator, senza dover cambiare il codice della collezione o del client.

## Pattern Observer



Il pattern Observer è un pattern comportamentale che ci permette di definire un **meccanismo di sottoscrizione** in modo da notificare più oggetti un qualsiasi evento che si è verificato all'oggetto che si sta osservando.

L'oggetto avente uno stato di interesse è chiamato **Subject**, ma visto che notificherà altri oggetti riguardo il suo cambiamento di stato, lo chiameremo **Publisher**. Tutti gli altri oggetti che vogliono seguire i cambiamenti del publisher, vengono chiamati **Subscribers**.