



Laurea Triennale in informatica-Università di Salerno
Corso di *Ingegneria del Software*- Prof. C. Gravino

Design Pattern



Riferimento	
Versione	0.1
Data	5/12/2025
Destinatario	Docente Ingegneria del Software Prof. Carmine Gravino
Presentato da	Gaetano Aprile, Luigi Artuso, Alessandro De Bonis & Marco Galdi
Approvato da	



Design Patterns

Per garantire un'architettura scalabile e coerente, il sistema non adotta soluzioni ad hoc per ogni singola schermata, ma implementa Design Pattern consolidati. Questi agiscono come veri e propri template di soluzioni: modelli astratti progettati per risolvere problemi di interazione ricorrenti in modo efficace e standardizzato.

Di seguito vengono presentati due esempi pratici di pattern integrati nel progetto, analizzando come il template astratto viene istanziato nell'interfaccia reale e come le sue componenti sono strutturate per il riutilizzo:

1. Observer Pattern

Verrà utilizzato nel sottosistema Gestione Eventi in comunicazione con Gestione Notifiche. Serve a gestire la reazione del sistema quando un evento cambia stato, in particolare quando viene annullato dall'organizzatore.

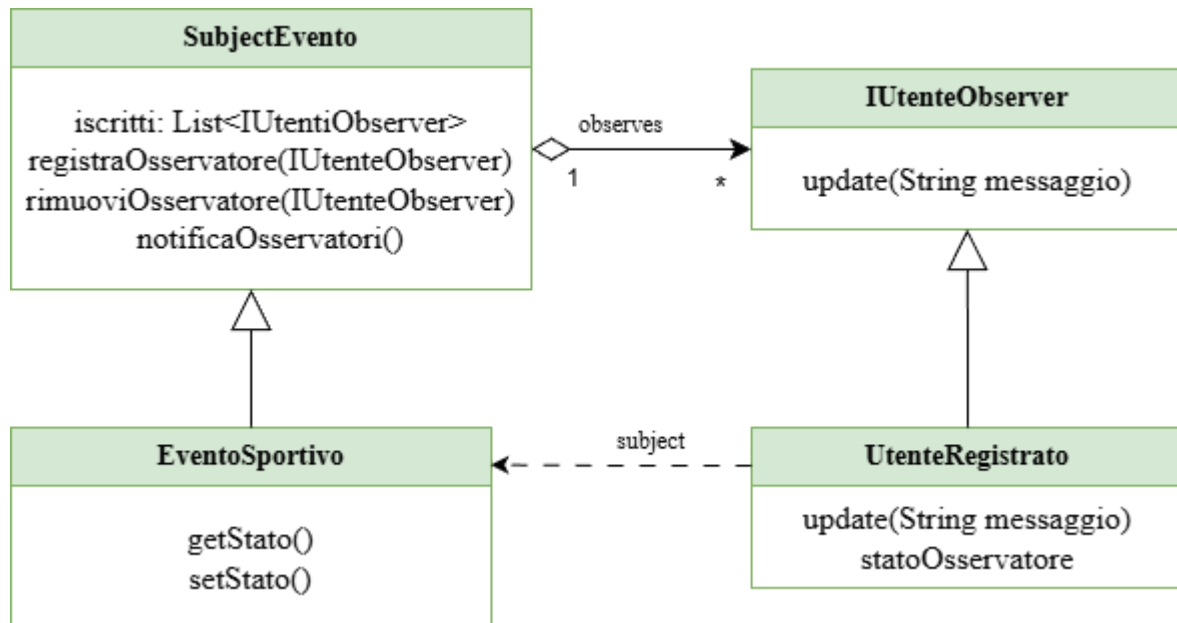
Obiettivo: Definire una dipendenza "uno-a-molti" tra l'oggetto EventoSportivo (Subject) e gli utenti iscritti (Observers). L'obiettivo è disaccoppiare la logica di business dell'evento dalla logica di notifica: l'evento non deve sapere come o via quale canale (email, push, sms) avvisare gli utenti, deve solo notificare il cambiamento di stato.

Implementazione:

- **Subject (Superclasse):** La classe astratta `SubjectEvento`. Mantiene la lista degli osservatori (`iscritti`) e fornisce i metodi pubblici `registraOsservatore` e `rimuoviOsservatore`, oltre al metodo `notificaOsservatori`.
- **ConcreteSubject (Soggetto Concreto):** La classe `EventoSportivo` estende `SubjectEvento`. Contiene lo stato dell'evento (es. `ANNULLATO`) e i metodi getter/setter. Quando lo stato cambia, invoca il metodo ereditato `notificaOsservatori()`.
- **Observer (Interfaccia):** un'interfaccia `IUtenteObserver` che espone il metodo `update(String messaggio)`.
- **ConcreteObserver (Osservatore Concreto):** la classe `UtenteRegistrato` implementa l'interfaccia `IUtenteObserver`.

Flusso:

1. L'Organizzatore preme "Annulla Evento" sulla GUI.
2. Il metodo `setStato(ANNULLATO)` di `EventoSportivo` viene invocato.
3. All'interno di `setStato`, viene chiamato automaticamente il metodo `notificaOsservatori()`.
4. Il metodo cicla su tutti gli iscritti e chiama `update()`, che delega al sottosistema Notifiche l'invio della mail di avviso



2. Facade Pattern

Verrà utilizzato nel sottosistema Servizi Geografici. Questo sottosistema deve interagire con API esterne complesse (es. Google Maps, OpenStreetMap) per la geolocalizzazione e il calcolo delle distanze per i filtri di ricerca.

Obiettivo: fornire un'interfaccia unificata e semplificata a un insieme di interfacce di un sottosistema complesso. Il Facade nasconde la complessità delle chiamate HTTP, della gestione delle API Key, del parsing dei JSON e delle conversioni di coordinate, offrendo ai controller di MatchPoint metodi semplici e diretti.

Implementazione:

- **Facade Class:** Una classe **MappeServiceFacade**.
- **Metodi esposti:** Metodi semplici come **getDistanza(Indirizzo A, Indirizzo B)** o **getCoordinate(Indirizzo)**.
- **Sottosistema Nascosto:** Le classi che gestiscono la connessione HTTP, la libreria JSON (come GSON) e la logica specifica del fornitore di mappe (Google/Bing/OSM).
- **Client:** I controller **GestioneEventoControl** (per salvare la posizione) e **SezioneEventiControl** (per filtrare per raggio KM) chiamano solo il Facade.

Vantaggio: Se si decide di passare da Google Maps a OpenStreetMap per risparmiare, bisogna modificare solo il codice dentro la classe **MappeServiceFacade**, senza toccare nemmeno una riga del resto dell'applicazione.

