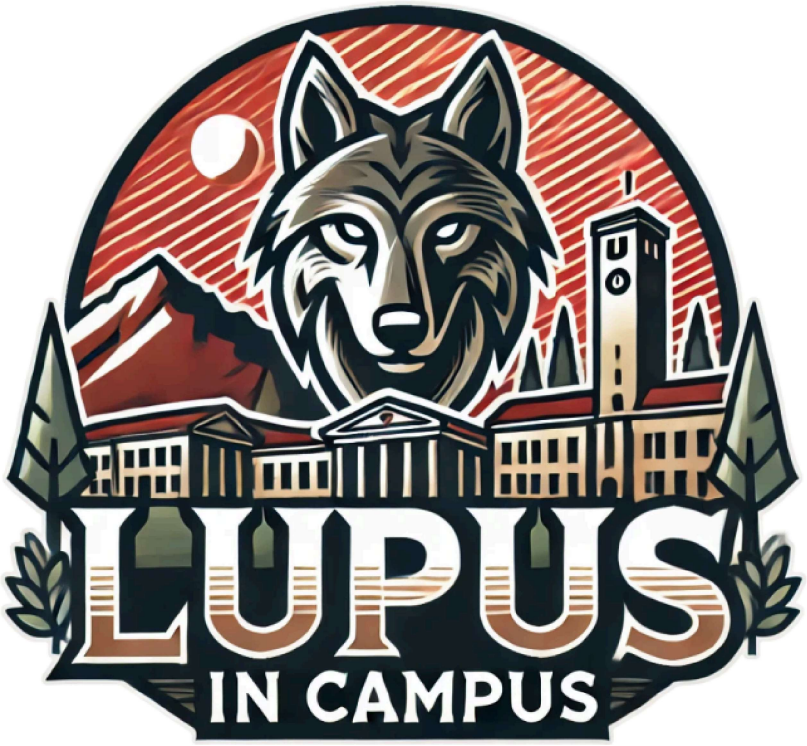



# ODD

# Design Pattern

## LUPUS IN CAMPUS

| | |
|---|---|
| **Riferimento** | NC12_RAD |
| **Versione** | 0.1 |
| **Data** | 12/02/2025 |
| **Destinatario** | Prof Carmine Gravino |
| **Presentato da** | NC12 Team |
| **Approvato da** | |

# Revision History

| Data | Versione | Descrizione | Autori |
|---|---|---|---|
| 13/02/2025 | 0.1 | Prima stesura | S.G |
| 17/02/2025 | 0.1 | Revisione | F.G |

# Membri del team

| Nome | Acronimo | Informazioni di contatto |
|---|---|---|
| Angelo Ascione | A.A | a.ascione19@studenti.unisa.it |
| Federica Graziuso | F.G | f.graziuso1@studenti.unisa.it |
| Stefano Gagliarde | S.G | s.gagliarde@studenti.unisa.it |
| Christian Izzo | C.I | c.izzo43@studenti.unisa.it |

# Sommario

# 1 Introduzione

L'obiettivo di questo documento è descrivere i **Design Pattern** adottati nel sistema, illustrandone il contesto di utilizzo, la motivazione della scelta e il loro impatto sulla progettazione.

Per ogni pattern verranno analizzati i seguenti aspetti:

- **Contesto:** il problema che il pattern risolve e perché è stato scelto
- **Struttura:** come il pattern è stato applicato e il suo ruolo nel sistema
- **Vantaggi e Svantaggi:** una valutazione del pattern in termini di benefici e limitazioni.

# 2 Design Pattern

I design pattern utilizzati nel sistema sono i seguenti:

- **Observer**: Questo pattern è essenziale in un'architettura MVC, poiché consente al modello di notificare automaticamente le viste (i client) di ogni cambiamento.
- **Abstract Factory**: Questo pattern consente di assegnare i ruoli ai giocatori in base al numero di partecipanti alla partita.

## 2.1 Observer

### 2.1.1 Contesto

Nel sistema, il **pattern Observer** viene utilizzato per gestire le comunicazioni in tempo reale tra il server e i client durante la partita. Questo è fondamentale per mantenere aggiornati i giocatori su eventi come l'ingresso nella lobby, l'inizio della partita e le azioni di gioco.

Abbiamo scelto **Observer** perché il sistema segue un'architettura **MVC**, dove i client devono essere notificati in tempo reale di cambiamenti avvenuti nel server. Il pattern ci permette di mantenere un'alta modularità, separando la logica di gestione del gioco dalla propagazione degli eventi ai client.

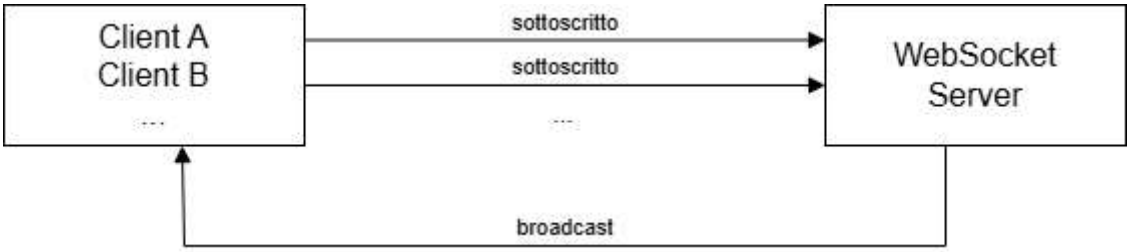
### 2.1.2 Struttura del Pattern nel sistema

L'implementazione dell'Observer nel nostro sistema si basa su **WebSocket** e **STOMP**, permettendo ai client di sottoscriversi a specifici canali (topic) e ricevere aggiornamenti quando si verificano eventi nel gioco.

**Componenti principali:**

- **Observable**: il server, che genera e invia notifiche quando avvengono eventi di gioco.
- **Observer**: i client connessi, che ricevono e gestiscono gli aggiornamenti.

Diagramma semplificato del funzionamento:




I client si connettono ai **topic** (es. `/topic/lobby`, `/topic/game`) e ricevono aggiornamenti quando il server pubblica nuovi messaggi.

### 2.1.3 Vantaggi e Svantaggi

**Vantaggi:**

- **Comunicazione real-time**: i client ricevono aggiornamenti immediati
- **Alta modularità**: il codice rimane separato tra logica di gioco e notifica agli utenti.
- **Scalabilità**: WebSocket permette la gestione di più connessioni contemporaneamente.

**Svantaggi:**

- **Gestione delle connessioni**: bisogna monitorare le disconnessioni e le riconnessioni dei client.

- **Overhead iniziale**: la configurazione WebSocket richiede più risorse rispetto a una semplice API REST.

## 2.2 Abstract Factory

### 2.2.1 Contesto

Nel sistema, il **pattern Abstract Factory** viene utilizzato per gestire l'assegnazione dei ruoli ai giocatori in base al numero di partecipanti. Questo è fondamentale per garantire che la distribuzione dei ruoli sia coerente con il numero di giocatori e segua le regole stabilite per il gioco.

È stato scelto Abstract Factory perché il numero di ruoli varia in base al numero di giocatori, e ogni configurazione di gioco ha un insieme predefinito di ruoli. Questo pattern permette di incapsulare la logica di assegnazione all'interno di specifiche factory, migliorando la modularità e la manutenibilità del codice.

### 2.2.2 Struttura del Pattern nel sistema

L'implementazione di Abstract Factory si basa su tre componenti principali:

- **Interfaccia RoleAssignmentFactory**: definisce il metodo `getRoles(int numPlayers)`, che fornisce la lista di ruoli disponibili in base al numero di giocatori.
- **Implementazioni concrete della factory**: tre classi (`Small: da 6 a 8 giocatori`; `Medium: da 9 a 12 giocatori`; `Large: da 13 a 18 giocatori`) che implementano l'interfaccia e restituiscono ruoli diversi a seconda delle dimensioni della partita.
- **Factory Provider**: la classe `RoleAssignmentFactoryProvider` funge da gestore centrale che seleziona la factory appropriata in base al numero di giocatori.

### 2.2.3 Vantaggi e Svantaggi

**Vantaggi:**

- **Alta modularità**: ogni configurazione di gioco ha una propria factory separata, rendendo il codice più organizzato e facile da estendere.
- **Manutenibilità**: se in futuro si vogliono modificare le regole di assegnazione, è sufficiente aggiornare le factory specifiche senza toccare il codice principale.
- **Separazione delle responsabilità**: la logica di assegnazione dei ruoli è separata dalla logica di gestione del gioco, facilitando i test e la gestione del codice.

**Svantaggi:**

- **Sovraccarico di classi**: per ogni nuova configurazione di gioco è necessario creare una nuova factory, aumentando il numero di classi nel sistema.
- **Maggiore complessità iniziale**: rispetto a un'assegnazione diretta basata su semplici condizioni `if`, il pattern introduce una struttura più articolata, che potrebbe essere eccessiva per scenari semplici.