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

OBJECT DESIGN DOCUMENT

JustInTime

Riferimento	
Versione	0.2
Data	7/12/2024
Destinatario	Studenti di Ingegneria del Software 2024/25
Presentato da	Dashchuk Yulia, Fernandez Ferdinando Gregorio, Ferraro Giulia, Genovese Vincenzo
Approvato da	

Revision History

Data	Versione	Descrizione	Autori
23/12/2024	0.1	Completa definizione dei design pattern	Ferdinando Gregorio Fernandez Giulia Ferraro
29/12/2024	0.2	Revisione finale	Team

INDICE

Design Patterns

In questa sezione vengono descritti i principali design pattern utilizzati nello sviluppo dell'applicativo **JustInTime**. Per ogni pattern verranno forniti:

- Una breve introduzione teorica.
- Il problema risolto nel progetto.
- Dettagli sull'implementazione adottata

1.0 Bridge Pattern

Il **Bridge Pattern** è un design pattern strutturale il cui scopo principale è separare l'interfaccia di una classe dalla sua implementazione.

La divisione avviene creando due gerarchie principali:

- **Astrazione:** rappresenta l'interfaccia principale utilizzata dal livello client contiene una o più operazioni che delegano il lavoro effettivo all'implementazione.
- **Implementazione**: definisce i dettagli concreti su come le operazioni sono effettivamente eseguite.

Queste due gerarchie comunicano tramite un'interfaccia comune, ma possono evolversi separatamente senza influire l'una sull'altra.

1.1 Esempio d'uso nel progetto

Nel nostro progetto, il Bridge Pattern viene utilizzato per dividere la logica tra il **Player** e l'**Utente**.

Ogni Player corrisponde ad un Utente in fase di partita, di conseguenza questo approccio ci consente di gestire in modo più flessibile le diverse implementazioni e proprietà richieste per ciascuno di essi.

- **abstractPlayer e abstractUser**: sono due interfacce che rappresentano un player e un utente generico.
- **Player**: è una classe che implementa l'interfaccia dell'**abstractPlayer**, questa rappresenterà il nostro giocatore durante la partita.

- **Utente:** è una classe che implementa l'interfaccia dell'**abstractUtente**, questa rappresenta un utente qualunque. All'interno di questa andiamo anche a definire un campo di tipo **abstractPlayer,** considerando che ad ogni Utente corrisponde un Player.

Grazie a questa divisione sarà molto più semplice estendere il concetto di Player e Utente nell'eventuale aggiunta di ruoli.

2.0 Factory Pattern

Il **Factory Pattern** definisce un'interfaccia o una classe astratta per creare un oggetto, delegando alle sottoclassi la responsabilità di decidere quale classe concreta istanziare.

Questo approccio consente di centralizzare la logica di creazione degli oggetti, garantendo flessibilità e separazione delle responsabilità.

2.1 Esempio d'uso nel progetto

Nel nostro progetto, il Factory Pattern è stato utilizzato per gestire la creazione dei diversi tipi di mazzi, come **MazzoPesca** e **MazzoScarto**. La classe **MazzoFactory** è responsabile di istanziare dinamicamente le classi concrete basandosi su un parametro testuale.

- Il metodo **createMazzo(String tipo)** utilizza una struttura switch per scegliere il tipo di mazzo richiesto:
  - "**pesca**": crea un'istanza di **MazzoPesca**, utilizzato per pescare le carte.
  - "**scarto**": crea un'istanza di **MazzoScarto**, utilizzato per raccogliere le carte scartate.

Grazie all'adozione del Factory Pattern consente di aggiungere facilmente nuovi tipi di mazzi, migliorando l'estensibilità del progetto senza dover modificare il codice esistente.

3.0 State Pattern

Lo State Pattern è un design pattern comportamentale che consente a un oggetto di modificare il proprio comportamento quando il suo stato interno cambia.

In pratica, lo State Pattern permette di rappresentare gli stati di un oggetto come classi separate, ognuna con il proprio comportamento, rendendo più semplice gestire i cambiamenti di stato e separare le logiche associate.

3.1 Esempio d'uso nel progetto

Nel progetto abbiamo implementato lo State Pattern per gestire i diversi stati della classe **Partita**.

L'interfaccia **GameState** definisce il metodo **execute**, che contiene le operazioni specifiche di ogni stato.

Le classi concrete, come **StartGameState, TurnState, PauseState** e **EndGameState,** implementano questa interfaccia, fornendo comportamenti specifici per ogni fase del gioco.

La gestione dello stato è affidata alla classe **PartitaService**, che funge da contesto e permette il passaggio dinamico da uno stato all'altro in base alla logica della partita.

- **StartGameState**: distribuisce le carte e imposta il primo giocatore.
- **TurnState**: gestisce il turno del giocatore corrente, controllando il tempo e verificando la fine del turno.
- **PauseState**: gestisce la pausa tra un turno e l'altro, preparando il prossimo giocatore.
- **EndGameState**: determina il vincitore e chiude la partita.