Design: Sala Giochi Virtuale

Progetto: Sala Giochi Virtuale
Data: 11 Gennaio 2026
Team di Progetto: Team UNI
Corso: Ingegneria del Software – Università degli Studi di Bergamo

Contenuto:

- Modelling
- Software Architecture
- Software Design Pattern

1. Modelling (Modellazione UML)

La fase di modellazione ha prodotto i seguenti diagrammi UML, utilizzati per guidare l'implementazione del codice:

- **Class Diagram:** Rappresenta la struttura statica del sistema, le relazioni di ereditarietà tra la classe astratta `VideoGames` e i giochi concreti, e le associazioni con le entità del database (`User`, `Category`).
- **Sequence Diagram / Activity Diagram:** Descrivono il flusso dinamico delle interazioni. Sono stati formalizzati in particolare gli scenari di:
    - *Login Utente:* Interazione tra GUI, Controller e Database.
    - *Partita:* Flusso di avvio gioco, aggiornamento stato e salvataggio punteggio.
- **Use Case Diagram:** Descrive le funzionalità offerte agli attori.
- **State Diagram:** Mostra i possibili stati dell'applicazione.
- **Comunication Diagram / Component Diagram / Package Diagram:** Utili a descrivere le relazioni intra/inter modulari, a rappresentare come si interfacciano le componenti del sistema e a descrivere come vengono scambiati dati e informazioni tra partner di iterazione.

(Nota: I diagrammi UML sono allegati nella cartella `documenti/DiagrammiUML` del repository).

2. Software Architecture (Architettura Software)

2.1 Stile Architetturale

Il sistema adotta lo stile **Layered Architecture** (Architettura a Strati) combinato con il pattern **MVC** (Model-View-Controller). Questa scelta garantisce una netta separazione delle responsabilità (Separation of Concerns), facilitando lo sviluppo, la manutenzione e il testing.

Il sistema è strutturato in tre livelli logici:

1. **Presentation Layer (View/Controller):** Gestisce l'interazione con l'utente e la visualizzazione grafica. Implementato tramite *JavaFX*.
2. **Business Logic Layer (Model):** Contiene le regole del dominio, la logica dei giochi e le entità.
3. **Data Access Layer (Data):** Gestisce la persistenza e il recupero delle informazioni dal database.

2.2 Architectural Views (Viste Architetturali)

Vista Logica (Package View)

Questa vista descrive come il codice è organizzato in pacchetti Java, riflettendo la suddivisione a strati.

- `com.mondsciomegn.salagiochi.gui` **(Presentation)**: Contiene la classe `Room.java` e le logiche di gestione delle finestre (Stage) e degli eventi (Login, Navigazione).
- `com.mondsciomegn.salagiochi.videogame` **(Logic)**: Contiene le implementazioni delle classi di ogni gioco (`Tris`, `BattleShip`, `Roulette e DiceGuesser`).
- `com.mondsciomegn.salagiochi.db` **(Data)**: Contiene le classi per la connessione al DB (`DataBaseConnection`), l'inizializzazione (`DataBaseInitializer`) e le entità (`User`, `Category`, `VideoGames`).

Vista Componenti e Connettori (C&C View)

Questa vista descrive i componenti *run time* e le loro dipendenze esterne.

- **Componente ArcadeHub (Application):** Il core applicativo sviluppato dal team.
- **Componente H2 DBMS (External Library):** Il motore di *database embedded* che viene invocato dall'applicazione tramite driver JDBC.
- **Componente JavaFX (Runtime):** Il *framework* grafico che gestisce il *rendering* della *UI*.

Descrizione del Connettore: L'applicazione comunica con il componente *H2 DataBaseManagementSystem* tramite protocollo **JavaDBConnection** su file locale (`jdbc:h2:./database/salagiochi`). Questo connettore è sincrono e gestito dalla classe `DataBaseConnection`.

2.3 Gestione Dipendenze (Maven)

L'uso di Maven ha permesso di dichiarare formalmente le dipendenze (*JavaFX, H2 Driver*) nel file *pom.xml*, rendendo il progetto indipendente dalla configurazione locale della macchina dello sviluppatore. Un esempio di dipendenza è quella per il database:

```
<dependency>
        <groupId> com.h2database </groupId>
        <artifactId> h2 </artifactId>
        <version> 2.2.224 </version>
</dependency>
```

La gestione delle librerie esterne e del ciclo di vita del software è affidata per questo motivo ad **Apache Maven**, che permette un allineamento delle verisioni al Team

3. Software Design Pattern

3.1 Pattern Applicati

Per risolvere problemi ricorrenti di progettazione, sono stati applicati i seguenti Design Pattern:

A. Delegation Pattern

Questo Pattern è stato utilizzato per modellare il comportamento del gioco `BattleShip` in modo da non eseguire direttamente determinate funzioni, delegando l'implementazione ad altre istanze.

- La classe `BattleShip` utilizza due istanze di `Player` per gestire la logica di giocatore e avversario.
- `Player` delega a sua volta la gestione della griglia e dei colpi a `Field.`
- `Field` utilizza i metodi della grafica della classe `GuiGrid` per la rappresentazione della *UI*.

B. Façade Pattern

Questo è un Pattern strutturale che fornisce un'interfaccia semplice e unificata per gestire un insieme complesso di metodi e classi, nascondendo in una *black box* la complessità interna e semplificandone l'uso.

- La classe `Player` è interessata esclusivamente al piazzamento delle navi e agli attacchi, senza occuparsi di gestire le coordinate o il funzionamento dei bottoni.
- `Field` fornisce metodi semplificati ad alto livello mentre la logica viene gestita da altre classi

C. Strategy Pattern

Questo pattern comportamentale è stato utilizzato nel gioco **Roulette** per gestire la complessità delle regole di scommessa e il calcolo delle vincite.

- L'interfaccia `GameRules` definisce il contratto per una regola di scommessa generica.
- Classi concrete come `RangeRule`, `ColorRule` e `RowRule` incapsulano la logica specifica (algoritmo) per determinare se una scommessa è vinta.
- Questo permette di aggiungere nuove tipologie di puntata senza modificare la logica principale di verifica nella classe `RouletteRules`, riducendo drasticamente la complessità ciclomatica

D. DAO (Data Access Object) Pattern

Utilizzato per separare la logica di business dai dettagli di persistenza dei dati.

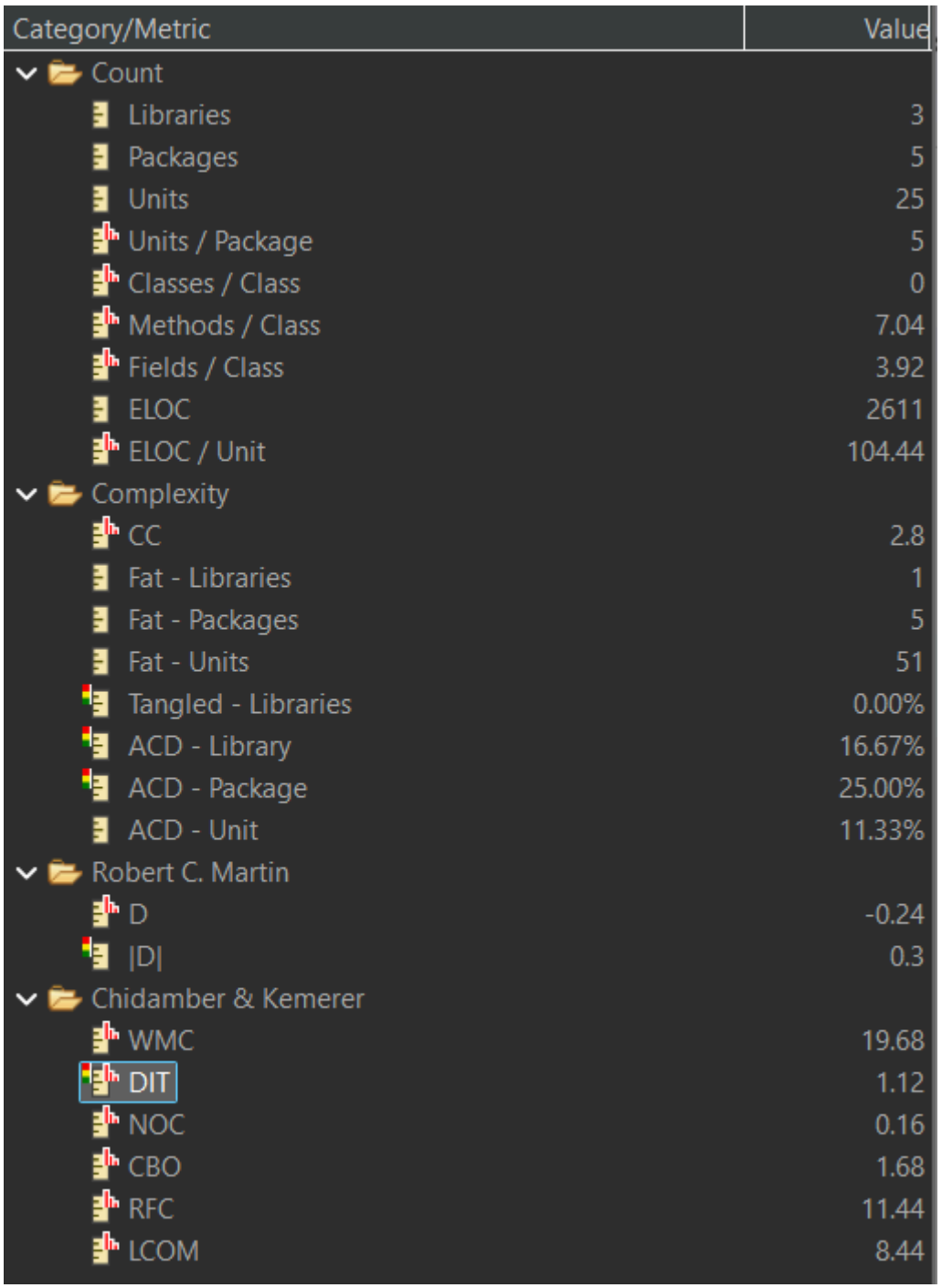
- La classe `RouletteDB` funge da intermediario tra l'applicazione e il database.
- La classe `Roulette` non contiene query SQL, ma invoca metodi astratti (es. `getScoreFromDB`), garantendo che modifiche alla struttura del database non impattino la logica del videogioco.

4. Misurazione del Codice e Complessità:

Di seguito elencate le metriche generali del progetto (dati presi da software di analisi statica)

- **Linee di Codice (LOC):** ~ 2.600
- **Numero di Classi:** ~ 25
- **Numero di Packages:** ~ 5

- **Weighted Methods per Class:** ~ 20
- **Depth of Inheritance Tree:** ~ 2
- **Number of Children:** ~ 1
- **Coupling Between Objects:** ~ 2
- **Response For Classes**: ~ 12
- **Lack of Cohesion in Methods:** ~ 9


Category/Metric
Value
Count
Libraries 3
Packages 5
Units 25
Units / Package 5
Classes / Class 0
Methods / Class 7.04
Fields / Class 3.92
ELOC 2611
ELOC / Unit 104.44
Complexity
CC 2.8
Fat - Libraries 1
Fat - Packages 5
Fat - Units 51
Tangled - Libraries 0.00%
ACD - Library 16.67%
ACD - Package 25.00%
ACD - Unit 11.33%
Robert C. Martin
D -0.24
|D| 0.3
Chidamber & Kemerer
WMC 19.68
DIT 1.12
NOC 0.16
CBO 1.68
RFC 11.44
LCOM 8.44