UNIVERSITA' DEGLI STUDI
DI BARI

PROGETTO REALIZZATO DA:

PASQUALE TRAETTA – MATRICOLA 450428

LUCA MIGNOGNA – MATRICOLA 467644

CORSO DI LAUREA: INFORMATICA

ANNO ACCADEMICO: 2008/2009

ESAME: METODI AVANZATI DI PROGRAMMAZIONE

DOCENTI: PROF. DONATO MALERBA

PROF. MICHELANGELO CECI

SMGAME

GIOCO ITALIANO DEL SETTE E MEZZO

Reference Guide

INDICE

1. Regolamento “Gioco Italiano del Sette e Mezzo”

In questo Paragrafo si riprendono le Regole del “Gioco Italiano del Sette e Mezzo” facendo riferimento al sito web http://it.wikipedia.org/wiki/Sette_e_mezzo evidenziando i casi in cui l'implementazione di progetto si discosta da tali regole in senso restrittivo o estensivo.

1.1 - Definizione del Gioco

Il *Gioco Italiano del Sette e Mezzo* è un popolare gioco di carte italiano classificato fra i giochi d'azzardo. Per giocare si usa un mazzo di quaranta carte da gioco che possono essere napoletane, regionali italiane o francesi (da queste ultime, vanno tolti gli 8, i 9 e i 10). Nel mazzo ci sono, quindi, quattro carte di semi diversi per ciascun valore fra asso e 7, più dodici figure.

Il numero ideale di giocatori nel sette e mezzo è di 4-6, ma si può giocare già con 2 giocatori fino ad un massimo di 10-12. Uno dei giocatori è deputato alla funzione di Mazziere e di Banco. Il ruolo del mazziere non è comunque fisso, ma cambia durante il gioco. Tutti gli altri giocatori non competono fra di loro ma esclusivamente contro il mazziere di turno.

L'implementazione di Progetto ha adottato il Mazzo di Carte Napoletane e ha assimilato il numero minimo e massimo di Giocatori rispettivamente nei valori 2 e 12.

1.2 - Scopo del Gioco

Lo scopo del gioco è quello di realizzare il punteggio più alto possibile senza mai “*sballare*”, vale a dire senza superare il 7 e mezzo. Il mazziere deve cercare di eguagliare o superare il punteggio del maggior numero possibile di giocatori senza sballare. Il mazziere riscuote la somma puntata da tutti i giocatori che sballano, o che totalizzano un punteggio inferiore o uguale al suo, e paga l'equivalente della puntata ai giocatori che superano il suo punteggio. Il punteggio di ciascun giocatore si calcola sommando i punti di tutte le carte che possiede.

- **Le carte dall'asso al 7 valgono tanti punti quanto è il loro valore numerico. L'asso vale 1 punto, il 2 vale 2 punti, ecc.**
- **Le figure valgono mezzo punto.**
- **Il re di denari o di cuori ha la funzione di *matta* o *jolly*, può cioè assumere il punteggio di una qualsiasi altra carta a discrezione del giocatore che la possiede. Sono comuni anche varianti del gioco che prevedono una matta diversa dal re di denari o di quadri, o che stabiliscano che la matta debba assumere un punteggio intero da 1 a 7, escludendo dunque il mezzo punto.**

Il 7 e mezzo realizzato con due sole carte (un sette e una figura, oppure la matta e

una figura oppure la matta e un sette) è detto *sette e mezzo reale* o *sette e mezzo legittimo* o anche *sette e mezzo d'emblée*. Il giocatore che realizza questo punteggio riceve dal banco una somma pari al doppio della posta e diventa mazziere alla mano successiva.

Se più giocatori fanno sette e mezzo reale nella stessa mano, tutti loro saranno pagati doppio, e il banco passerà al giocatore che siede più vicino alla destra del mazziere.

Se è il mazziere a fare sette e mezzo reale, egli riscuote una posta doppia da tutti i giocatori, tranne da quelli che hanno fatto a loro volta sette e mezzo reale o che hanno sballato, dai quali riscuote una posta semplice, e non deve cedere il banco.

Se uno dei giocatori realizza il sette e mezzo reale avendo la Matta e una figura (es. re di quadri + una figura qualunque) il mazziere vince solo se realizza a sua volta un sette e mezzo reale usando solo due carte, quindi un sette e una figura; se il mazziere realizza un sette e mezzo utilizzando più di due carte si dice che è illegittimo e può battere tutti gli altri e sette e mezzo anche se fatti con due carte ad esclusione di quello sopra elencato realizzato con la matta e figura.

L'implementazione di Progetto ha assimilato completamente tali regole.

1.3 - Svolgimento

Prima di iniziare si sorteggia quale dei giocatori farà il mazziere nella prima mano. All'inizio della mano il mazziere distribuisce una carta coperta a ciascun giocatore; il gioco inizia, quindi, dal giocatore seduto alla destra del mazziere e procede in senso antiorario. A turno, ciascun giocatore effettua le seguenti operazioni:

1. Guarda la propria carta.
2. Effettua la puntata ponendo la somma corrispondente sopra la carta coperta o davanti a sé. Si può decidere di chiamare con una cifra e stare con un'altra. Di solito si fa con le carte di alta cifra. Se si sballa si danno i soldi e la carta chiamata ma non la carta coperta.
3. Se lo vuole, può richiedere altre carte per migliorare il proprio punteggio. Tutte le carte successive alla prima vengono date scoperte una per volta, fino a quando il giocatore continua a richiederne.
4. Se un giocatore sballa o realizza 7 e mezzo deve farlo notare immediatamente scoprendo anche la prima carta ricevuta. Quando si sballa, la posta viene subito ritirata dal banco, in caso contrario il gioco procede e il turno passa al giocatore successivo.

Dopo che tutti i giocatori hanno fatto il loro gioco, il mazziere scopre la propria carta e decide se prenderne altre. Se il mazziere sballa, paga alla pari tutte le puntate dei giocatori ancora in gioco. In caso contrario, tutti scoprono le proprie carte e confrontano il proprio punteggio con quello del mazziere. Ciascun giocatore cede, quindi, la propria posta al banco o ne incassa l'equivalente secondo le regole del gioco. Con il sette e mezzo reale la posta viene raddoppiata.

Se il sette e mezzo viene pagato doppio, dà diritto di diventare mazziere al primo giocatore che lo realizza; in caso contrario il mazziere tiene il banco fino

all'esaurimento delle carte nel mazzo, dopodiché il banco passa al giocatore seduto alla sua destra.

L'implementazione di Progetto ha assimilato il processo che determina lo svolgimento del Gioco ad eccezione di parte del precedente punto 2 allorquando si cita la possibilità di “**... chiamare con una cifra e stare con un'altra ...” i**n quanto da ricerche eseguite sembra essere più una variante che una regola standard del Gioco.

1.4 - Varianti implementative introdotte

La traduzione in un'Applicazione Software del Gioco comporta la necessità di dover introdurre alcune varianti implementative per gestire particolari situazioni. Di seguito si indicano le varianti introdotte:

- E' stato definito un Credito Iniziale per ogni Giocatore pari a 1000,00 Euro piuttosto che una cifra arbitraria per ogni Giocatore. Peraltro le regole di Wikipedia non dicono nulla in merito.
- E' stato stabilito che ogni Partita sia costituito da un minimo di una Manche e da un massimo di 10. Tale criterio è del tutto arbitrario in quanto le regole tratte da Wikipedia non asseriscono nulla in merito ma ai fini del progetto può risultare una utile limitazione al fine di verificare il corretto funzionamento delle varie parti dell'applicazione. Tale limite sarebbe comunque in futuro facilmente rimovibile.
- E' stato stabilito un criterio che decida quando una intera Partita al Gioco Italiano del Sette e Mezzo sia terminata e tale criterio stabilisce che una Partita termina allorquando si è raggiunto il numero di Manches prestabilito o allorquando uno dei Giocatori è in *bancarotta* ossia ha un credito residuo inferiore o uguale a 0.00 Euro.
- L'implementazione del Gioco, come si vedrà prevede la presenza non solo di Giocatori Umani ma anche Artificiali e per questi ultimi si è dovuto procedere necessariamente all'implementazione di una qualche forma di *comportamento* anch'essa Artificiale per simulare le azioni di Gioco.

2. Scelte Progettuali

In questo paragrafo si intendono descrivere informalmente le scelte progettuali che sono state intraprese per realizzare il progetto. Le decisioni hanno cercato di far convergere le linee guida del progetto con la modellazione che è stata con l'obiettivo di implementare tutti i punti richiesti seppure attraverso una implementazione differente.

2.1 - Giocatore

Il Giocatore è stato modellato considerando che sostanzialmente esistono due tipi di Giocatori: il Giocatore Umano e il Giocatore Artificiale denominato d'ora innanzi CPU.

Come si può facilmente intuire la differenza consiste essenzialmente nel fatto che il primo prevede un'interazione effettiva col programma laddove per il secondo sarà simulata. Oltre a questo evidentemente sarà il comportamento durante il gioco l'elemento distintivo tra le due tipologie in quanto nel primo caso le decisioni intraprese saranno il frutto di un ragionamento umano e comunque completamente delegate all'Interazione Uomo-Macchina mentre nel secondo saranno la conclusione di un ragionamento basato su un minimo di intelligenza artificiale che occorrerà implementare nell'applicazione.

Dunque la modellazione del Giocatore prevede una classe HumanPlayer (Giocatore Umano) e una sottoclasse di questa ossia CPUPlayer (CPU).

Occorre peraltro far notare che nel gioco del Sette e Mezzo, secondo una regola prestabilita, uno dei Giocatori diventa Mazziere; ciò gli conferisce un comportamento differente da quello del Giocatore classico indipendentemente dal fatto che sia un Giocatore Umano o Artificiale.

Si è deciso di modellare questo comportamento come un Ruolo che il Giocatore assume in un dato istante piuttosto che come un ulteriore tipo di Giocatore perché nella realtà il tipo di un Giocatore è costante per tutta la Partita mentre quello del Mazziere è effettivamente un Ruolo che a turno può assumere uno dei Giocatori della Partita.

2.2 - Componenti di una Partita

Il concetto di Partita è stato modellato separandone gli aspetti caratteristici:

- Regole del Gioco
- Componenti del Gioco
- Motore del Gioco

Sebbene possa sembrare un'anomalia, occorre fissare le Regole del Gioco prima di iniziare una Partita e questo per il semplice motivo che non esiste una

versione standard del gioco del Sette e Mezzo. Al contrario esistono diverse varianti e nessuna garantisce quale sia quella originale ed ufficiale del gioco.

Peraltro occorre evidenziare che l'insieme delle Regole costituenti una particolare variante del gioco sono incompatibili con un'altra variante tanto da non poterne fondere le singole opzioni al fine di creare una variante universale del gioco.

Ciò, come sarà ribadito in seguito, costituisce uno dei motivi fondamentali per considerare tali varianti da implementare come una utile estensione dle contesto che renderebbe così il gioco adattabile alle diverse abitudini e consuetudini consolidate nelle varie regioni d'Italia.

Ad ogni modo, ai fini del progetto le Regole del Gioco che sono state implementate rispecchiano quelle del già citato sito di Wikipedia.

Per componenti di una Partita si intendono gli elementi imprescindibili che concorrono a far si che una Partita al gioco del Sette e Mezzo possa sussistere.

Tali componenti sono stati individuati nei Giocatori, nel Mazzo di Carte e nelle Regole del Gioco. Dunque le Regole del Gioco sono al tempo stesso sia la base comune con cui si gioca una Partita che un componente stesso di una Partita.

Per la precisione una Partita è stata modellata attraverso la classe **`Game`**.

L'insieme dei Giocatori è stato modellato con la classe **`PlayerList`** che contiene lista dei Giocatori della Partita e che dunque sono unici e non modificabili per tutta quella partita.

Il Mazzo di Carte è stato modellato con la classe **`Deck`** che è l'insieme delle carte da gioco per il Sette e Mezzo. Si sarebbe potuto implementare la classe **`Deck`** utilizzando il pattern Singleton ma ciò non è stato fatto perché sebbene un Mazzo di Carte sia immutabile nel corso di tutta una Partita e anche tra partite diverse, come si vedrà c'è la necessità che il mazzo iniziale di 40 carte termini e che in sua vece si utilizzi un mazzo derivante dalle carte recuperate da tutti i Giocatori che hanno sballato. Tale insieme di Carte sarà usato alla stregua del Mazzo iniziale vero e proprio pur essendone soltanto un sottoinsieme. Ad ogni modo la mutabilità dei componenti del Mazzo di Carte fa propendere per la non adozione del pattern Singleton.

Infine è stata introdotta una classe denominata **`GameEngine`** la quale non ha lo scopo di modellare una particolare entità del mondo reale; al contrario essa è utile per concretizzare l'idea di un arbitro del gioco o comunque di una sorta di orchestratore che ha il compito di moderare tutte le interazioni fra le componenti del gioco che hanno la possibilità di interagire tra di loro. L'adozione di una simile classe seguendo il pattern Mediator fornisce uno strumento centralizzato per gestire tutte le interazioni evitando l'interazione diretta tra coppie di oggetti che richiederebbe la conoscenza da parte di entrambi delle interfacce esposte dalle controparti.

2.3 – Oggetti di confine

Per come sono state modellati gli oggetti è possibile raggrupparli per insiemi

disgiunti ossia:

- classi relative all'interfaccia grafica
- classi relative al nucleo del gioco
- classi relative alla interazione client-server
- classi relative alla gestione della persistenza

A seguito del raggruppamento eseguito occorre che siano individuati specifici “Oggetti di confine” che consentano l'interazione fra tali insiemi. Allo scopo sono state individuate le classi **`CoreProxy`** e **`ClientProxy`**.

Questa scelta è stata dettata sostanzialmente dall'evidenza che avere degli oggetti di confine accentra in tali oggetti le richieste provenienti da oggetti appartenenti ad uno degli insiemi individuati i quali sono a conoscenza solo del proprio contesto. Ciò ha come diretta conseguenza una maggiore semplicità nella gestione dei riferimenti ad oggetti tra la totalità degli oggetti stessi e quindi una più semplice manutenibilità del codice.

In altri termini se ad esempio l'interfaccia grafica in test successivi dimostra in qualche caso di visualizzare informazioni corrette ed in altri casi no allora è molto semplice individuare il problema perché esso è sicuramente residente nelle classi che vanno da quella di confine fino a quelle dell'altro insieme per cui questa fa da proxy.

Tutto ciò è possibile perché l'interfaccia grafica è a conoscenza solo del proprio contesto ossia le altre interfacce grafiche realizzate e i componenti grafici di cui è composta e null'altro relativo alla logica applicativa.

Alla stregua di quanto detto sin'ora vale lo stesso tipo di ragionamento anche per i restanti insiemi individuati.

2.4 - Motore del Gioco

Si consideri la situazione tipica in pieno svolgimento del gioco in cui il Mazziere sta interagendo con il giocatore di turno. L'idea è che l'applicazione sia dotata di una GUI e che attraverso tale GUI il giocatore di turno che ha già avuto una carta coperta ne chieda un'altra puntando una corrispettiva somma di danaro.

L'utente esegue la sua richiesta attraverso la GUI con un pulsante opportunamente predisposto. Il click sul pulsante genera un evento gestito dalla classe specifica della GUI la quale saprà che alla pressione di quello specifico pulsante deve essere invocato un opportuno metodo della classe **`ClientProxy`** supponiamo **`ClientProxy.requestCard(Player, bet)`**. Dove il metodo **`requestCard`** rappresenta la richiesta di una nuova Carta da parte del Giocatore. Ovviamente la classe della GUI deve invocare il metodo **`requestCard`** passando anche il **`Player`** (Giocatore) che ha eseguito la richiesta e la somma (**`bet`**) puntata.

A questo punto la classe **`ClientProxy`** interagirà opportunamente con la classe **`CoreProxy`** che a sua volta delegherà alla classe **`GameEngine`** invoca l'opportuno

metodo della classe **`Deck`** per ricevere la carta successiva dal mazzo che è stato mescolato all'inizio.

Se il mazzo non è terminato allora al Giocatore viene data un'altra Carta e contestualmente la classe **`GameEngine`** verifica che il Giocatore non abbia sballato. Se così non è il metodo **`requestCard`** termina la sua esecuzione altrimenti prende le contromisure necessarie ossia:

- Toglie al Giocatore tutte le Carte che gli sono state data e le raggruppa in un contenitore provvisorio della classe **`Deck`**.
- Calcola il valore complessivo della puntata del Giocatore nella mano in questione e sottrae l'importo dal suo credito totale.
- L'importo complessivo della puntata del Giocatore che ha sballato viene immediatamente portata ad aumento del credito del Mazziere.

Nel caso in cui il mazzo sia terminato la classe **`GameEngine`** provvederà ad utilizzare l'insieme delle carte raggruppate nella stessa classe **`GameEngine`** derivanti da tutti quei Giocatori che hanno già sballato nella stessa mano.

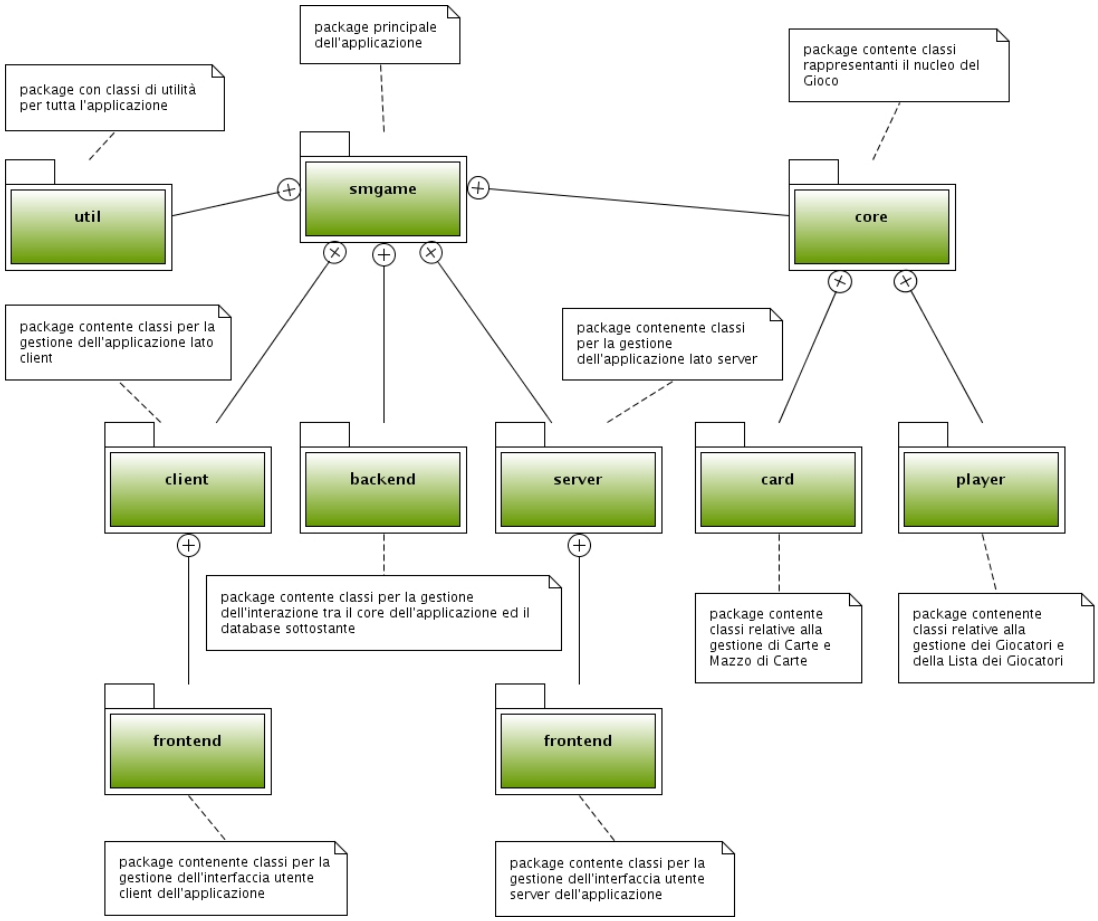
3. Riferimento Tecnico

In questo paragrafo si intende descrivere gli obiettivi dei singoli package realizzati per il progetto, delle classi più importanti contenute e dei relativi metodi e attributi.

Tale descrizione sarà accompagnata da un diagramma UML delle classi contenute nel package tranne che per il package **`org.smgame.client.frontend`** a causa dell'alto numero delle classi e dunque la difficoltà di rendere ben visibili gli oggetti nel diagramma.

Inoltre saranno inclusi i codici sorgente delle classi più significative e anche in questo caso la scelta è dettata dal buon senso ossia si è preferito inserire il codice relativo alle classi che più hanno attinenza con la logica del Gioco tralasciando quelle di contorno che hanno lo scopo di offrire funzioni di utilità o sono relative esclusivamente all'interfaccia grafica dell'applicazione.

Mentre nei paragrafi successivi si procederà ad esaminare ogni singolo package, di seguito si illustra il diagramma UML dei package inclusi nel progetto.



3.1 - package `org.smgame.backend`

Il package **`org.smgame.backend`** contiene tutte le classi necessarie all'interazione col database, e quindi ad effettuare operazioni di lettura e/o scrittura in esso. Il pattern Valued Object viene usato per l'oggetto **`DBPropertiesVO`**, dove vengono memorizzati i dati di accesso al database per essere riusati nel server. Di seguito si analizzano le classi contenute:

- **`DBAccess`**: è la classe che si occupa dell'accesso al database, oltre al caricamento dei dati di accesso, effettua anche il test di connessione e la chiusura connessione; infatti i metodi principali sono:
  - **`getConnection`**: verifica se esiste già la connessione al database e restituisce l'oggetto connessione, in caso contrario lo crea richiamando un metodo privato.
  - **`testConnection`**: testa la connessione restituendo un booleano true/false.
  - **`closeConnection`**: chiude la connessione al database.
- **`DBTransactions`**: è la classe che si occupa di gestire le transazioni sul database in scrittura e lettura, le principali operazioni sono la scrittura della transazione per ogni giocatore con annesse carte giocate, e la lettura dello storico partite. Ecco alcuni metodi in esso contenuti:
  - **`executeSingleTransaction`**: esegue la scrittura della transazione sul database, ricevendo i dati del giocatore, nome, numero partita, numero mano, carte giocate, punteggio e relativa vincita/perdita.
  - **`GetStoryGame`**: restituisce lo storico di tutte le partite.

classe **`DBAccess`** – package **`org.smgame.backend`**

```
package org.smgame.backend;

import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;
import org.smgame.util.ResourceLocator;
import org.smgame.util.Logging;

/** Classe DbAccess/accesso database
 *
 * @author Traetta  Pasquale 450428
 * @author Mignogna Luca     467644
 */
public class DBAccess {
    final private String DRIVER_CLASS_NAME_MYSQL = "org.gjt.mm.mysql.Driver";
    private static Connection conn = null;

    /**Costruttore privato
     *
```

```
     * @throws java.lang.ClassNotFoundException
     * @throws java.sql.SQLException
     * @throws java.io.IOException
     */
    private DBAccess() throws ClassNotFoundException, SQLException, IOException
{
        DBPropertiesVO dbpropVO = requestDBPropertiesVO();
        //carico il driver JDBC MYSQL
        Class.forName(DRIVER_CLASS_NAME_MYSQL);
        //creo l'url JDBC per la connessione
             String url = dbpropVO.getUri() + dbpropVO.getServer() + ":" +
dbpropVO.getPort() + "/" +
                dbpropVO.getDatabase();
                 conn = DriverManager.getConnection(url, dbpropVO.getUser(),
dbpropVO.getPassword());
        Logging.logInfo("Connessione effettuata con: " + url);
    }

    /**inizializza e restituisce l'oggetto connessione
     *
     * @return connessione
     */
      public static Connection getConnection() throws ClassNotFoundException,
SQLException, IOException {
        if ((conn == null) || (!conn.isValid(0))) {
            DBAccess dba = new DBAccess();
        }
        return conn;
    }

    /**Chiude la connessione
     *
     */
    public static void closeConnection() throws SQLException {
        conn.close();
        conn = null;
    }

    /**Verifica lo stato della connessione
     *
     * @return true se attiva, false se null
     */
    public static boolean testConnection() throws Exception {
        Connection tempConn = getConnection();
        if (tempConn.isValid(0))
            return true;
        return false;
    }

      /**legge il file contenente le informazioni sul database e setta le
variabili per la connessione
     *
     * @throws java.io.IOException
     */
    public static DBPropertiesVO requestDBPropertiesVO() throws IOException{
        Properties properties = new Properties();

         properties.load(DBAccess.class.getResourceAsStream(ResourceLocator.getR
esource()+"database.properties"));

        Logging.logInfo("Caricamento database.properties effettuato");
```

```
        DBPropertiesVO dbPropVO = new DBPropertiesVO();
        dbPropVO.setUri(properties.getProperty("URI"));
        dbPropVO.setServer(properties.getProperty("SERVER"));
        dbPropVO.setPort(properties.getProperty("PORT"));
        dbPropVO.setDatabase(properties.getProperty("DATABASE"));
        dbPropVO.setUser(properties.getProperty("USER"));
        dbPropVO.setPassword(properties.getProperty("PASSWORD"));

        return dbPropVO;
    }

}//end class
```

classe **DBTransactions** – package **org.smgame.backend**

```
package org.smgame.backend;

import java.io.IOException;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Types;
import java.util.ArrayList;
import java.util.LinkedHashMap;
import java.util.List;

import org.smgame.core.card.Card;
import org.smgame.util.Logging;

/** Classe DBTransactions/gestisce le transazioni col database
 *
 * @author Traetta  Pasquale 450428
 * @author Mignogna Luca     467644
 */
public class DBTransactions {

    private final String tableTrans = "TRANSACTION"; //nome tabella
       private final String[] columnTrans = new String[]{"transaction_id",
"game_id", "manche",
                                                          "player_name", "score",
"win_lose_amount"};
    private final String tableCard = "CARD c";
      private final String[] columnCard = new String[]{"c.card_id", "c.suit",
"c.point"};
    private final String tableRelation = "TRANSACTION_CARD";
      private final String[] columnRelation = new String[]{"transaction_id",
"card_id"};
    private long idT; //id transazione
    private long id_game; //id partita
    private int manche; //numero manche
    private String player; //nome giocatore
    private double score; //punteggio
    private double win; //vincita
    private ArrayList<Card> cardAL;
    //private ArrayList<DBTransactions> transactionsAL;

    /**Costruttore vuoto
```

```
     *
     */
    public DBTransactions() {}

    /**Costruttore
     *
     * @param id_game id partita
     * @param manche numero di manche nel gioco
     * @param player giocatore
     * @param score punteggio
     * @param win vincita
     * @param cardal arraylist di carte
     */
      public DBTransactions(long id_game, int manche, String player, double
score,
            double win, List<Card> cardal) {
        this.id_game = id_game;
        this.manche = manche;
        this.player = player;
        this.score = score;
        this.win = win;
        this.cardAL = (ArrayList<Card>) cardal;
    }

    /**Restituisce id gioco
     *
     * @return idgioco
     */
    public long getId_game() {
        return id_game;
    }

    /**Restituisce nome giocatore
     *
     * @return nome
     */
    public String getPlayer() {
        return player;
    }

    /**Restituisce punteggio
     *
     * @return punteggio
     */
    public double getScore() {
        return score;
    }

    /**restituisce vincita/perdita
     *
     * @return vincita o pertita
     */
    public double getWin() {
        return win;
    }

    /**Restituisce la manche
     *
     * @return numero di manche
     */
    public int getManche() {
```

```
        return manche;
    }

    /**imposta l'id della transazione
     *
     * @param id numero transazione
     */
    public void setIdTransaction(long id) {
        this.idT = id;
    }

    /**Restituisce l'id transazione
     *
     * @return id transazione
     */
    public long getIdTransaction() {
        return idT;
    }

    /**registra su db la singola transazione
     *
     * @throws java.lang.ClassNotFoundException
     * @throws java.sql.SQLException
     * @throws java.io.IOException
     * @throws java.lang.Exception
     */
       public void executeSingleTransaction() throws ClassNotFoundException,
SQLException,
            IOException, Exception {
        Connection conn = DBAccess.getConnection();
        String sql = "INSERT INTO " + tableTrans + "(" + columnTrans[1] + "," +
columnTrans[2] + "," +
                columnTrans[3] + "," + columnTrans[4] + "," + columnTrans[5] +
") VALUES (?,?,?,?,?)";
        PreparedStatement prpstmt = conn.prepareStatement(sql);
        setParameter(prpstmt, 1, getId_game(), Types.BIGINT);
        setParameter(prpstmt, 2, getManche(), Types.INTEGER);
        setParameter(prpstmt, 3, getPlayer(), Types.VARCHAR);
        setParameter(prpstmt, 4, getScore(), Types.DOUBLE);
        setParameter(prpstmt, 5, getWin(), Types.DOUBLE);
        Logging.logInfo(prpstmt.toString());
        prpstmt.execute();
        setIdTransaction(getLastInsertId(conn));
        executeArraylistCardTransaction(conn);
    }

    /**Scrive su db le carte presenti nell'arraylist delle transazioni
     *
     * @param conn connessione db
     * @throws java.sql.SQLException
     * @throws java.lang.Exception
     */
       private void executeArraylistCardTransaction(Connection conn) throws
SQLException, Exception {
         String sql = "INSERT INTO " + tableRelation + " VALUES (?,(SELECT " +
columnCard[0] +
                " FROM " + tableCard + " WHERE " + columnCard[1] + "=? AND " +
columnCard[2] + "=?));";
        for (int i = 0; i < cardAL.size(); i++) {
            PreparedStatement prpstmt = conn.prepareStatement(sql);
            setParameter(prpstmt, 1, getIdTransaction(), Types.BIGINT);
```

```
                    setParameter(prpstmt, 2, cardAL.get(i).getSuit().toString(),
Types.VARCHAR);
                  setParameter(prpstmt, 3, cardAL.get(i).getPoint().toString(),
Types.VARCHAR);
            Logging.logInfo(prpstmt.toString());
            prpstmt.execute();
        }
    }

    /**Cerca l'ultimo id appena inserito della transazione
     *
     */
                private    long    getLastInsertId(Connection    conn)    throws
ClassNotFoundException, SQLException, IOException {
        long last_id = -1;
        String sql = "SELECT LAST_INSERT_ID();";
        PreparedStatement prpstmt = conn.prepareStatement(sql);
        ResultSet rs = prpstmt.executeQuery();
        if (rs.next()) {
            last_id = rs.getLong(1);
        }
        return last_id;
    }

    /**Restituisce un map (long, matrice oggetti) ordinato per inserimento
     *
     * @return oggetto maps
     *
     * @throws java.lang.ClassNotFoundException
     * @throws java.sql.SQLException
     * @throws java.io.IOException
     * @throws java.lang.Exception
     */
    public LinkedHashMap<Long, Object[][]> getStoryGame() throws
            ClassNotFoundException, SQLException, IOException, Exception {

         LinkedHashMap<Long, Object[][]> map = new LinkedHashMap<Long, Object[]
[]>();

         String sqlDistinct = "SELECT DISTINCT(" + columnTrans[1] + ") FROM " +
tableTrans;
          String sqlCount = "SELECT count(*) FROM " + tableTrans + " WHERE " +
columnTrans[1] + "= ?;";
              String  sqlSelect  =  "SELECT  t."  +  columnTrans[2]  +  ",  t."  +
columnTrans[3] + ", t." + columnTrans[4] + ", t."
                     + columnTrans[5] + ", GROUP_CONCAT(LOWER("+ columnCard[2]
+"), \' \', LEFT(" + columnCard[1] +",1)) AS group_card" +
                       " FROM " + tableTrans + " t, " + tableCard + ", " +
tableRelation +
                  " r WHERE t." + columnTrans[1] + "= ? AND " + columnCard[0] +
"=r." + columnRelation[1] +
                    " AND r." + columnRelation[0] + "=t." + columnTrans[0] + "
GROUP BY t." + columnTrans[0] +
                " ORDER BY t." + columnTrans[2] + " ASC, t." + columnTrans[5] +
" DESC;";

        Connection conn = DBAccess.getConnection();
        PreparedStatement prpstmtDistinct = conn.prepareStatement(sqlDistinct);
        Logging.logInfo(prpstmtDistinct.toString());
        ResultSet rsDistinct = prpstmtDistinct.executeQuery();
```

```
        while (rsDistinct.next()) {
            Object[][] matrix = null;
            long id = rsDistinct.getLong(1);
            PreparedStatement prpstmtCount = conn.prepareStatement(sqlCount);
            setParameter(prpstmtCount, 1, id, Types.BIGINT);
            Logging.logInfo(prpstmtCount.toString());
            ResultSet rsCount = prpstmtCount.executeQuery();
            rsCount.next();
            int rows = rsCount.getInt(1);
            if (rows > 0) {
                matrix = new Object[rows][5];
                int r = 0;
                                            PreparedStatement   prpstmtSelect   =
conn.prepareStatement(sqlSelect);
                setParameter(prpstmtSelect, 1, id, Types.BIGINT);
                Logging.logInfo(prpstmtSelect.toString());
                ResultSet rsSelect = prpstmtSelect.executeQuery();
                while (rsSelect.next()) {
                    matrix[r][0] = rsSelect.getInt(1);
                    matrix[r][1] = rsSelect.getString(2);
                    matrix[r][2] = rsSelect.getDouble(3);
                    matrix[r][3] = rsSelect.getDouble(4);
                    matrix[r][4] = rsSelect.getString(5);
                    r++;
                }
            } //end if
            map.put(new Long(id), matrix);
        }//end while rsDistinct
        return map;
    }


    /**imposta i tipi di valore da usare per la preparedStatement
     *
     * @param stmt preparedstatement
     * @param index indice
     * @param value valore
     * @param type tipo
     * @throws java.sql.SQLException
     * @throws java.lang.Exception
     */
    private void setParameter(PreparedStatement stmt, int index,
            Object value, int type) throws SQLException, Exception {
        if (value == null) {
            stmt.setNull(index, type);
        } else {
            if (type == Types.VARCHAR) {
                stmt.setString(index, (String) value);
            } else if (type == Types.INTEGER) {
                stmt.setInt(index, ((Integer) value).intValue());
            } else if (type == Types.BIGINT) {
                stmt.setLong(index, ((Long) value).longValue());
            } else if (type == Types.DOUBLE) {
                stmt.setDouble(index, ((Double) value).doubleValue());
            } else {
                throw new Exception("Tipo di dato non gestito");
            }
        } //end
    } //end setParameter
} //end class
```

3.2 - package `org.smgame.client`

Il package **`org.smgame.client`** contiene tutte le classi determinano l'attività lato client dell'applicazione. Tali classi sostanzialmente si occupano di dialogare con il server RMI e di ricevere le richieste proveniente dalla GUI di cui sono l'unica interfaccia.

Di seguito invece si espone il significato delle classi contenute in questo package:

- **`ClientProxy`**: è la classe che come si diceva poc'anzi fa da interfaccia fra la GUI e i componenti core dell'applicazione. Per la precisione, essa di pone come filtro per le richieste girandole verso la classe **`CoreProxy`** del package **`org.smgame.core`** di cui si discuterà più avanti. La necessità di questa classe è dunque quella di ricevere le richieste ma soprattutto quella di smistarle. Tale smistamento avviene verso la classe **`CoreProxy`** se si tratta di una partita offline oppure verso l'oggetto **`stub`** remoto se si tratta di giocare una partita online.
- **`RMIClient`**: è la classe che fa da client RMI e dunque è quella che contiene i metodi per recuperare l'oggetto **`stub`** remoto ed invocarne i metodi opportuni. Da notare che questa classe è realizzata con il pattern singleton perchè è lecito attendersi che per tutta la durata dell'applicazione esista un unico oggetto client che si interfaccia col server. Tuttavia rispetto alla classica implementazione del pattern singleton c'è una variazione sul tema nel metodo **`getStub`** che si descrive di seguito:
    - **`getStub`**: questo metodo si occupa si di restituire un oggetto **`stub`** remoto ma la natura client-server dell'applicazione implicano qualche accorgimento necessario nel senso che può accedere che il server RMI sia giù e dunque non si possa recuperare l'oggetto **`stub`** o ancor peggio che l'oggetto **`stub`** già recuperato in precedenza non sia più valido e ciò si verifica facilmente provando ad invocare il metodo **`test`** nell'interfaccia **`IStub`**. Tutto ciò implica che in qualche modo l'oggetto **`stub`** sia ricreato e sia gestita l'eccezione che l'invocazione del metodo **`test`** può generare.
- **`SMGameClient`**: è la classe che contiene il metodo **`main`** dell'applicazione qualora si lanci l'applicazione come normale Application e la ridefinizione del metodo **`start`** nel caso si avvii l'applicazione come Applet.

Di seguito si indica il Diagramma UML delle classi del package **`org.smgame.client`**.

Di seguito si indica inoltre il codice sorgente delle classi del package **`org.smgame.client`**.

3.3 - package `org.smgame.client.frontend`

Il package **`org.smgame.client.frontend`** contiene tutte le classi necessarie a visualizzare la GUI che consente di poter giocare una partita al Gioco Italiano del Sette e Mezzo. Non si entrerà nel dettaglio dell'implementazione della GUI in quanto si tratterebbe solo di elencare metodi e Classi che dispongono oggetti a video senza evidenziare elementi di particolare interesse.

Ci si soffermerà invece solo su un aspetto importante che è poi stato la linea guida che si è intrapresa nello sviluppo dell'interfaccia.

L'idea sin dall'inizio è stata quella di realizzare dei componenti grafici che in generale sapessero poco o nulla della dinamica del gioco e addirittura di che gioco si trattasse o che si trattasse dello sviluppo di un gioco. In questo senso si è voluto produrre una interfaccia il più “stupida” possibile rispetto alla logica dei restanti componenti “core”.

Questa scelta implementativa è stata dettata dalla volontà di gestire centralmente in un'unica classe tutte le interazioni tra la GUI e l'”engine” del gioco. Tale classe centralizzante è **`CoreProxy`**. In questo modo ogni interazione dell'utente con l'interfaccia si traduce si in una chiamata specifica ad un metodo della classe **`CoreProxy`** ma la conseguenza di tale chiamata non è un valore restituito o un oggetto da gestire; semplicemente l'interfaccia è stata progettata in modo tale che ad ogni interazione con la classe **`CoreProxy`** ne segua un'altra che ha l'obiettivo di recuperare un ben specifico oggetto denominato **`GameVO`** al cui interno ci sono tutti i valori necessari a valorizzare le proprietà degli oggetti presenti nella GUI.

L'alternativa sarebbe stata quella di dotare tutte o quasi le interfacce della GUI di riferimenti alle classi core del gioco distribuendo in questo modo i riferimenti a tali oggetti e causando quindi una più difficile manutenzione del software a lungo andare.

Pertanto tutte le classi presenti nel package **`org.smgame.client.frontend`** sono classi che estendono qualche classe del framework Swing di Java e tutte quelle classi il cui nome termina con VO che sta per Valued Object è da intersi come un oggetto di ritorno restituito dalla classe **`ClientProxy`** all'interfaccia di competenza.

Di seguito invece si espone il significato di alcune classi che meritano qualche attenzione in più rispetto alle altre:

- **`ICustomDM`**: è l'interfaccia che consente di poter marcare un oggetto grafico di tipo **`JInternalFrame`** come di tipo **`ICustomDM`** in modo tale che la gestione del dragging di tale oggetto non sia possibile.
- **`CustomDM`**: è una classe che estende la classe **`DefaultDesktopManager`** del framework Swing di Java. Lo scopo di questa classe è fare l'overriding del metodo **`dragFrame`** per far si che gli oggetti grafici definiti come istanze della classe **`JInternalFrame`** non abbia la possibilità di essere spostati all'interno del frame contenitore. Infatti è possibile notare che

tutte le classi del package **org.smgame.client.frontend** che estendono la classe **JInternalFrame** implementano anche l'interfaccia **ICustomDM**.

- **NewGameEvent**: è una classe che estende la classe **EventObject** del framework Swing di Java allo scopo di poter rendere disponibile all'applicazione uno specifico evento da poter lanciare allorquando si verifica una specifica situazione ossia quella di creazione di una nuova Partita. Ovviamente associato a tale evento c'è anche una interfaccia listener denominta **NewGameListener** predisposta con un metodo ad-hoc per cogliere e gestire tale evento. La classe concreta che poi implementerà l'azione corrispondente da intraprendere a seguito del verificarsi di tale evento è la classe **MainJF** che infatti implementa l'interfaccia **NewGameListener**.
- **NewGameListener**: è l'interfaccia che estende **EventListener** allo svopo di definire il metodo **newGameCreating** che sarà utilizzato per gestire l'evento di creazione di una nuova Partita.
- **PDVViewerJP**: è una classe che predispone un JPanel adatto a contenere un testo in formato PDF. Si è utilizzata questa implementazione derivata direttamente dal sito www.jpedal.org come demo per l'utilizzo della libreria JPedal. Tale implementazione poi è stata modificata opportunamente per gli scopi di questo progetto.

Di seguito si indica il Diagramma UML delle classi del package **org.smgame.client.frontend**.

Di seguito si indica inoltre il codice sorgente di alcune classi e interfacce del package **org.smgame.core.card** ed in particolare quelle indicate nella descrizione di cui sopra.

interfaccia **ICustomDM** - package **org.smgame.client.frontend**

```
package org.smgame.client.frontend;

/**Interfaccia che permette agli oggetti che la implementano di poter essere
gestiti dalla customDM
 *
 * @author Traetta  Pasquale 450428
 * @author Mignogna Luca     467644
 */
public interface ICustomDM {}
```

classe **CustomDM** - package **org.smgame.client.frontend**

```
package org.smgame.client.frontend;

import javax.swing.DefaultDesktopManager;
import javax.swing.JComponent;

/**Classe DesktopManager personalizzato
 *
```

```
 * @author Traetta  Pasquale 450428
 * @author Mignogna Luca     467644
 */
public class CustomDM extends DefaultDesktopManager {

    @Override
    public void dragFrame(JComponent f, int x, int y) {
        if (!(f instanceof ICustomDM)) {
            super.dragFrame(f, x, y);
        }
    }
}
```

classe **NewGameEvent** - package **org.smgame.client.frontend**

```
package org.smgame.client.frontend;

import java.util.EventObject;

/**Evento nuovo gioco
 *
 * @author Traetta  Pasquale 450428
 * @author Mignogna Luca     467644
 */
public class NewGameEvent extends EventObject {

    /**Costruttore
     *
     * @param source provenienza
     */
    public NewGameEvent(Object source) {
        super(source);
    }
}
```

classe **NewGameListener** - package **org.smgame.client.frontend**

```
package org.smgame.client.frontend;

import java.util.EventListener;

/**interrfaccia di ascolto nuova partita
 *
 * @author Traetta  Pasquale 450428
 * @author Mignogna Luca     467644
 */
public interface NewGameListener extends EventListener {

    /**Creazione nuova partita
     *
     * @param e evento nuova partita
     */
    public void newGameCreating(NewGameEvent e);
}
```

classe **PDFViewerJP** - package **org.smgame.client.frontend**

```
package org.smgame.client.frontend;

import java.awt.BorderLayout;
import java.awt.Component;
import java.awt.FlowLayout;
import java.awt.Point;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.net.URI;
import java.net.URL;

import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextField;

import org.jpedal.PdfDecoder;
import org.jpedal.exception.PdfException;

/**Pannello per visualizzare i pdf
 *
 * @author Traetta  Pasquale 450428
 * @author Mignogna Luca     467644
 */
public class PDFViewerJP extends JPanel {

    /**the actual JPanel/decoder object*/
    private PdfDecoder pdfDecoder;
    /**name of current PDF file*/
    private String currentFile = null;
    /**current page number (first page is 1)*/
    private int currentPage = 1;
    private final JLabel pageCounter1 = new JLabel("Pagina ");
      private JTextField pageCounter2 = new JTextField(4);//000 used to set
prefered size
     private JLabel pageCounter3 = new JLabel("di");//000 used to set prefered
size

    /**Costruttore
     *
     * @param name nome del file da aprire
     */
    public PDFViewerJP(String name) {
        pdfDecoder = new PdfDecoder(true);
        //ensure non-embedded font map to sensible replacements
        PdfDecoder.setFontReplacements(pdfDecoder);
        currentFile = name;//store file name for use in page changer
        try {
            //this opens the PDF and reads its internal details
            pdfDecoder.openPdfFile(new URI(currentFile).getPath());

            //these 2 lines opens page 1 at 100% scaling
            pdfDecoder.decodePage(currentPage);
            pdfDecoder.setPageParameters(1, 1); //values scaling (1=100%). page
number
        } catch (Exception e) {
        }
        //setup our GUI display
```

```
        initializeViewer();

        //set page number display
        pageCounter2.setText(String.valueOf(currentPage));
        pageCounter3.setText("di " + pdfDecoder.getPageCount());
    }

    /**Verifica se il criptaggio è presente ed un'eventuale passsword
     *
     * @return true = pdf accessibile, false = password o criptazione presente
     */
    private boolean checkEncryption() {
//    check if file is encrypted
        if (pdfDecoder.isEncrypted()) {
                //if file has a null password it will have been decoded and
isFileViewable will return true
            while (!pdfDecoder.isFileViewable()) {
                /** popup window if password needed */
                   String password = JOptionPane.showInputDialog(this, "Please
enter password");
                /** try and reopen with new password */
                if (password != null) {
                    try {
                        pdfDecoder.setEncryptionPassword(password);
                    } catch (PdfException e) {
                    }
                    //pdfDecoder.verifyAccess();
                }
            }
            return true;
        }
        //if not encrypted return true
        return true;
    }

    /**Inizializza la visione e i suoi componenti
     *
     */
    private void initializeViewer() {
        setLayout(new BorderLayout());

          Component[] itemsToAdd = initChangerPanel();//setup page display and
changer

        JPanel topBar = new JPanel();
        topBar.setLayout(new FlowLayout(FlowLayout.CENTER, 2, 2));

        for (int i = 0; i < itemsToAdd.length; i++) {
            topBar.add(itemsToAdd[i]);
        }

        add(topBar, BorderLayout.NORTH);

           JScrollPane display = initPDFDisplay();//setup scrollpane with pdf
display inside
        add(display, BorderLayout.CENTER);
    }

    /**
     * returns the scrollpane with pdfDecoder set as the viewport
     */
```

```
    private JScrollPane initPDFDisplay() {
        JScrollPane currentScroll = new JScrollPane();
         currentScroll.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROL
LBAR_AS_NEEDED);
         currentScroll.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR
_AS_NEEDED);
        currentScroll.getVerticalScrollBar().setUnitIncrement(20);
        currentScroll.setViewportView(pdfDecoder);

        return currentScroll;
    }

    /**Inizializza le componenti e le restituisce
     *
     * @return array di componenti
     */
    private Component[] initChangerPanel() {
        Component[] list = new Component[11];

        /**back to page 1*/
        JButton start = new JButton();
        start.setBorderPainted(false);
                                                    URL      startImage      =
getClass().getResource("/org/jpedal/examples/simpleviewer/res/start.gif");
        start.setIcon(new ImageIcon(startImage));
        start.setToolTipText("Rewind to page 1");
//    currentBar1.add(start);
        list[0] = start;
        start.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {
                if (currentFile != null && currentPage != 1) {
                    currentPage = 1;
                    try {
                        pdfDecoder.decodePage(currentPage);
                        pdfDecoder.invalidate();
                        repaint();
                    } catch (Exception e1) {
                    }
                    //set page number display
                    pageCounter2.setText(String.valueOf(currentPage));
                }
            }
        });

        /**back 10 icon*/
        JButton fback = new JButton();
        fback.setBorderPainted(false);
                                                    URL      fbackImage      =
getClass().getResource("/org/jpedal/examples/simpleviewer/res/fback.gif");
        fback.setIcon(new ImageIcon(fbackImage));
        fback.setToolTipText("Rewind 10 pages");
//    currentBar1.add(fback);
        list[1] = fback;
        fback.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {
                if (currentFile != null && currentPage > 10) {
                    currentPage -= 10;
```

```
                    try {
                        pdfDecoder.decodePage(currentPage);
                        pdfDecoder.invalidate();
                        repaint();
                    } catch (Exception e1) {
                    }

//             set page number display
                        pageCounter2.setText(String.valueOf(currentPage));
                  }
              }
          });

          /**back icon*/
          JButton back = new JButton();
          back.setBorderPainted(false);
                                                          URL       backImage       =
getClass().getResource("/org/jpedal/examples/simpleviewer/res/back.gif");
          back.setIcon(new ImageIcon(backImage));
          back.setToolTipText("Rewind one page");
//    currentBar1.add(back);
          list[2] = back;
          back.addActionListener(new ActionListener() {

              @Override
              public void actionPerformed(ActionEvent e) {
                  if (currentFile != null && currentPage > 1) {
                      currentPage -= 1;
                      try {
                          pdfDecoder.decodePage(currentPage);
                          pdfDecoder.invalidate();
                          repaint();
                      } catch (Exception e1) {
                      }

//          set page number display
                          pageCounter2.setText(String.valueOf(currentPage));
                  }
              }
          });

          pageCounter2.setEditable(true);
          pageCounter2.addActionListener(new ActionListener() {

              @Override
              public void actionPerformed(ActionEvent a) {

                  String value = pageCounter2.getText().trim();
                  int newPage;

                  //allow for bum values
                  try {
                      newPage = Integer.parseInt(value);

                       if ((newPage > pdfDecoder.getPageCount()) | (newPage < 1))
{
                          return;
                      }

                      currentPage = newPage;
                      try {
```

```
                        pdfDecoder.decodePage(currentPage);
                        pdfDecoder.invalidate();
                        repaint();
                    } catch (Exception e) {
                    }

                } catch (Exception e) {
                    JOptionPane.showMessageDialog(null, '>' + value + "< is Not
a   valid   Value.\nPlease   enter   a   number   between   1   and   "   +
pdfDecoder.getPageCount());
                }

            }
        });

        /**put page count in middle of forward and back*/
//    currentBar1.add(pageCounter1);
//    currentBar1.add(new JPanel());//add gap
//    currentBar1.add(pageCounter2);
//    currentBar1.add(new JPanel());//add gap
//    currentBar1.add(pageCounter3);
        list[3] = pageCounter1;
        list[4] = new JPanel();
        list[5] = pageCounter2;
        list[6] = new JPanel();
        list[7] = pageCounter3;

        /**forward icon*/
        JButton forward = new JButton();
        forward.setBorderPainted(false);
                                                URL      fowardImage      =
getClass().getResource("/org/jpedal/examples/simpleviewer/res/forward.gif");
        forward.setIcon(new ImageIcon(fowardImage));
        forward.setToolTipText("forward 1 page");
//    currentBar1.add(forward);
        list[8] = forward;
        forward.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {
                            if  (currentFile  !=  null  &&  currentPage  <
pdfDecoder.getPageCount()) {
                    currentPage += 1;
                    try {
                        pdfDecoder.decodePage(currentPage);
                        pdfDecoder.invalidate();
                        repaint();
                    } catch (Exception e1) {
                    }

//        set page number display
                    pageCounter2.setText(String.valueOf(currentPage));
                }
            }
        });

        /**fast forward icon*/
        JButton fforward = new JButton();
        fforward.setBorderPainted(false);
                                               URL      ffowardImage      =
getClass().getResource("/org/jpedal/examples/simpleviewer/res/fforward.gif");
```

```
        fforward.setIcon(new ImageIcon(ffowardImage));
        fforward.setToolTipText("Fast forward 10 pages");
//    currentBar1.add(fforward);
        list[9] = fforward;
        fforward.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {
                                if (currentFile != null && currentPage <
pdfDecoder.getPageCount() - 9) {
                    currentPage += 10;
                    try {
                        pdfDecoder.decodePage(currentPage);
                        pdfDecoder.invalidate();
                        repaint();
                    } catch (Exception e1) {
                    }

//        set page number display
                    pageCounter2.setText(String.valueOf(currentPage));
                }
            }
        });

        /**goto last page*/
        JButton end = new JButton();
        end.setBorderPainted(false);
                                                        URL       endImage       =
getClass().getResource("/org/jpedal/examples/simpleviewer/res/end.gif");
        end.setIcon(new ImageIcon(endImage));
        end.setToolTipText("Fast forward to last page");
//    currentBar1.add(end);
        list[10] = end;
        end.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {
                                if (currentFile != null && currentPage <
pdfDecoder.getPageCount()) {
                    currentPage = pdfDecoder.getPageCount();
                    try {
                        pdfDecoder.decodePage(currentPage);
                        pdfDecoder.invalidate();
                        repaint();
                    } catch (Exception e1) {
                    }

//        set page number display
                    pageCounter2.setText(String.valueOf(currentPage));
                }
            }
        });

        return list;
    }
}
```

3.4 - package `org.smgame.core`

Il package **org.smgame.core** contiene le classi necessarie a modellare i componenti core dell'applicazione le cui classi sono descritte di seguito:

- **GameMode**: è una classe di tipo **Enum** allo scopo di individuare le due modalità di svolgimento di una Partita ossia online e offline.
- **Game**: è la classe che modella una Partita con i suoi componenti che sono oggetti rispettivamente delle classi **GameEngine**, **Deck**, **PlayerList**. Inoltre esiste anche il metodo **generateGameID** che ha lo scopo di generare un identificativo univoco per ogni Partita che si crea.

  Si noti infine che la classe implementa anche l'interfaccia **Serializable** che è fondamentale per rendere possibile il salvataggio su file di una Partita.
- **GameEngine**: questa è probabilmente la classe più importante di tutta l'applicazione perché implementa tutti i meccanismi necessari affinché si svolgano correttamente tutte le fasi del gioco pur delegando ai componenti di cui è composta il compito di recuperare certe informazioni.

  Si noti inoltre come anche in questo caso la classe implementa l'interfaccia **Serializable** e ciò è un obbligo oltre che una necessità in quanto la serializzazione di un oggetto di classe **Game** implica la serializzazione anche degli oggetti cui punta e dunque nel caso specifico anche l'oggetto di classe **GameEngine** deve essere serializzato e dunque deve implementare l'interfaccia **Serializable**.

  Di seguito si espongono i metodi di maggiore importanza della classe **GameEngine**:

  - **applyPaymentRule**: è il metodo che si occupa a fine manche di applicare le regole di pagamento per accreditare o addebitare una certa somma di danaro rispettivamente al giocatore che ha vinto e a quello che ha perso. Ovviamente l'implementazione tiene conto delle regole di Gioco già descritte nel Capitolo 1 di questo documento.
  - **closeManche**: è il metodo che si occupa di intraprendere tutte le attività necessarie a chiudere una Manche di una Partita. Tali attività prevedono l'applicazione delle regole di pagamento invocando il metodo **applyPaymentRule**, determinare il primo giocatore a partire dal Mazziere ad aver eventualmente realizzato un Sette e Mezzo Reale e recuperare tutte le carte che sono state distribuite ai Giocatori per accodarle in una lista opportuna.
  - **compareScore**: è il metodo che consente di poter stabilire tra due Giocatori chi ha vinto e chi ha perso. La corretta implementazione di questo metodo secondo le regole di Gioco già descritte nel Capitolo 1 di questo documento è fondamentale. Infatti è questo il metodo invocato dal metodo iterativamente dal metodo **applyPaymentRule**.
  - **declareGoodScore**: questo metodo si occupa di gestire opportunamente la volontà di un Giocatore Umano o Artificiale che

sia di voler “dichiarare” di possedere un “buon punteggio”. Tuttavia la gestione prevede il controllo che la puntata eseguita sia corretta in caso contrario solleva l'eccezione specifica **`BetOverflowException`**.

- **`distributeFirstCard`**: questo metodo si occupa di distribuire le prime carte ad ogni Giocatore ad inizio di una nuova Manche. La distribuzione prevede di fatto l'aggiunta alla lista delle carte di un Giocatore della carta ricevuta.
- **`isEndGame`**: questo metodo stabilisce un criterio di buon senso per determinare la fine di una Partita e tale criterio prevede che una partita sia considerata conclusa allorquando si è raggiunto il numero massimo di Manche previsto che è fissato a 10 oppure evntualmente è possibile terminare prima una Partita se un Giocatore è in “bancarotta” ossia ha credito inferiore o uguale a zero.
- **`isEndManche`**: questo metodo stabilisce un criterio per determinare la fine di una Manche e tale criterio prevede che una Manche sia considerata conclusa allorquando il giocatore di turno è il Mazziere e che il Mazziere abbia raggiunto uno tra gli stati di **`GoodScore`** o **`ScoreOverflow`** e ciò per la banale osservazione che se lo stato del Mazziere è **`null`** vuol dire che ancora non ha concluso il suo turno.
- **`NextPlayer`**: questo metodo si occupa di determinare quale sia il prossimo Giocatore di turno.
- **`playCPU`**: questo metodo si occupa di simulare il gioco di un Giocatore Artificiale.
- **`requestCard`**: questo metodo si occupa di gestire la richiesta di una Carta proveniente da un Giocatore. La richiesta viene esaudita se la puntata è corretta, in caso contrario si solleva una opportuna eccezione **`BetOverflowException`**. Tuttavia è possibile che chiedendo una carta il Giocatore concluda il gioco con uno “sballo” e ciò va gestito opportunamente accreditando immediatamente la somma puntata al Mazziere e addebitandola al Giocatore che ha “sballato” ed allo stesso tempo si recuperano le carte che hanno condotto alla giocata “sballata”; la gestione di questa eventualità si conclude sollevando l'eccezione opportuna **`ScoreOverflowException`**. Le carte recuperate sono evidenziate nell'interfaccia grafica in scala di grigio.

  Se non si verifica alcun evento eccezionale allora semplicemente la carta viene assegnata al Giocatore.
- **`selectBankPlayer`**: questo metodo si occupa di determinare chi sia il Mazziere per una determinata Manche. Anche questo metodo come altri descritti in precedenza tiene evidentemente conto delle regole di Gioco già descritte nel Capitolo 1 di questo documento.
- **`selectFirstRandomBankPlayer`**: questo metodo si occupa di

determinare chi sia il primo Mazziere della Partita.

- **startManche**: questo metodo imposta correttamente il contesto di una Partita in modo tale da poter cominciare una nuova Manche. Le attività eseguite in questo metodo prevedono:
  - lo svuotamento della lista delle puntate e delle carte per ogni Giocatore ed il reset dello status;
  - l'aggiornamento del numero di Manche corrente;
  - la selezione del Mazziere per la Manche corrente;
  - il mischiare il Mazzo di Carte;
  - la determinazione del primo Giocatore di turno per la Manche corrente.

- **CoreProxy**: è la classe che si occupa di ricevere le richieste provenienti dalla classe **ClientProxy** ed invocare i metodi opportuni della classe **GameEngine** se la richiesta ha a che fare con le fasi di sviluppo di una partita oppure svolge attività di persistenza su file o su database.

  Di seguito si espongono i metodi di maggiore importanza della classe **CoreProxy**:

  - **createGame**: questo metodo si occupa di creare una partita indipendentemente che sia in modalità online o offline.
  - **loadGame**: questo metodo si occupa di caricare una specifica partita.
  - **loadGames**: questo metodo si occupa di caricare tutte le partite salvate su file in una **HashMap** in memoria.
  - **saveGame**: questo metodo si occupa di salvare una partita ma come si vede tale metodo invoca il metodo saveGames e ciò vuol dire in realtà il salvataggio di una partita implica il salvataggio dell'intera **HashMap** caricata in memoria.
  - **saveGames**: questo metodo come accennato poc'anzi esegue il salvataggio su file dell'intera **HashMap** caricata in memoria.
  - **requestCard**: questo metodo è l'interfaccia per la richiesta di una carta tra la GUI è le classi core dell'applicazione e infatti la richiesta è smistata alla classe **GameEngine**. Per contro la classe **CoreProxy** verificherà l'esito della richiesta valorizzando opportunamente l'oggetto **gameVO**.
  - **declareGoodScore**: questo metodo è l'interfaccia per la dichiarazione di “buon punteggio” tra la GUI è le classi core dell'applicazione e infatti la richiesta è smistata alla classe **GameEngine**. Per contro la classe **CoreProxy** verificherà l'esito della richiesta valorizzando opportunamente l'oggetto **gameVO**.
  - **requestDataReport**: questo metodo si occupa di generare i dati che saranno visualizzati nella scoreboard al termine di ogni

Manche.

- **requestStoryGames**: questo metodo si occupa di recuperare i dati relativi alle Partite online precedenti dal database MySql.
- **saveTransaction**: questo metodo si occupa di salvare su database le transazioni di una intera Manche di una Partita online.
- **testDBConnection**: questo metodo si occupa di verificare se sia possibile collegarsi al DataBase MySql.
- **requestGameVO**: questo metodo è sicuramente quello più importante dell'intera classe in quanto genera un oggetto della classe **GameVO** che la scopo di fornire tutti gli elementi indispensabili alla GUI che visualizza la partita in corso. Come accennato nel Capitolo 2 di questo documento la classe **GameVO** è una classe progettata secondo il pattern Valued Object. In questo modo è possibile valorizzare opportuni componenti della classe **GameVO** che possono essere facilmente interpretati dalla GUI, senza che questa sia a conoscenza del significato intrinseco degli oggetti che essa stessa visualizza a video.
- **colorPlayerCredit**: questo metodo è utilizzato per poter visualizzare evidenziare in modo corretto quei Giocatori che per ogni Manche possiedono il maggior credito.

3.5 - package `org.smgame.core.card`

Il package **org.smgame.core.card** contiene tutte le classi utili a modellare opportunamente la Carta da Gioco e il Mazzo di Carte da Gioco. Di seguito si analizzano le classi contenute:

- **Point** (Punto): è una classe di tipo **Enum** allo scopo di enumerare in modo prefissato tutti i 10 Punteggi possibili di una Carta da Gioco di un Mazzo di Carte qualunque delle Regioni Italiane.
- **Suit** (Seme): è una classe di tipo **Enum** allo scopo di enumerare in modo prefissato tutti i 4 Semi possibili di una Carta da Gioco di un Mazzo di Carte qualunque delle Regioni Italiane.
- **Card** (Carta da Gioco): è la classe che modella una Carta da Gioco attribuendole un Punto, un Seme, un Valore tenendo conto che quest'ultimo è attribuito specificatamente per il Gioco Italiano del Sette e Mezzo.

  Occorre in particolare rilevare le seguenti due proprietà della classe **Card**:

  - **frontImage**: variabile che memorizza un'immagine distintiva di

quella specifica Carta da Gioco che per gli scopi del progetto è tratta dal Mazzo di Carte Napoletane.

- **backImage**: variabile che memorizza un'immagine del dorso di una generica Carta da Gioco tratta dalla Casa Editrice Modiano. Si è resa tale proprietà statica e dunque variabile di classe trattandosi di un'immagine che è comune a tutte le Carte da Gioco di un certo Mazzo.

- **JollyCard** (Matta): è la sottoclasse di Card utile a contraddistinguere il ruolo particolarmente importante che riveste la Matta nel Gioco Italiano del Sette e Mezzo. Essendo questa una sottoclasse di Card ne eredita le proprietà e i metodi ma in più è dotata della seguente proprietà:

  - **VALUES**: array di 8 valori che rappresenta tutti i possibili valori che la Matta può assumere nel Gioco Italiano del Sette e Mezzo. Si noti che tale array sia una costante in quanto elemento invariabile del Gioco ed è resa variabile di classe in quanto specifica di questa particolare Carta da Gioco

  Inoltre è dotata del seguente metodo:

  - **getBestValue**: metodo che acquisisce come parametro un punteggio e restituisce come valore il miglior valore che la Matta può assumere in modo tale da non superare il valore massimo di 7.5. Si farà largo uso di questo metodo per stabilire quale sia il punteggio ottenuto da un Giocatore che possegga tra le sue Carte la Matta. Questo metodo è reso statico in modo tale da poter essere sempre invocato e fornire un risultato senza la necessità di conoscere l'istanza di una Carta da Gioco di tipo **JollyCard**.

- **Deck** (Mazzo di Carte da Gioco): è la classe che modella un generico Mazzo di Carte da Gioco. Di seguito si analizzano in dettaglio le proprietà:

  - **ALL_VALUE**: array di 10 valori che rappresenta tutti i possibili valori che una Carta di un certo Seme può assumere. Si noti che tale array è una costante in quanto elemento invariabile del Gioco.

  - **CARDS**: **ArrayList** di oggetti con parametro di tipo **Card** che contiene tutte le Carte di un Mazzo. Anche questa variabile è costante in quanto un Mazzo di Carte (una volta definito) costituisce elemento invariabile del Gioco. Si noti come si sia fatto uso dell'Astrazione sui Dati nascondendo di fatto l'implementazione del Container di oggetti di tipo **Card** ed esponendo nella classe **Deck** i metodi utili agli scopi del Gioco senza consentire in alcun modo accesso diretto alla variabile **CARDS**.

  - **onGameCardList**: **ArrayList** di oggetti con parametro di tipo **Card** che contiene tutte le Carte del Mazzo “in uso” per una specifica partita. Con il termine “in uso” si intende riferirsi a tutte quelle Carte del Mazzo che non sono ancora state distribuite ai Giocatori o che lo sono già e dunque sono in possesso dei Giocatori stessi. Sarà molto comodo utilizzare questa Lista per per verificare quando il Mazzo di Carte è terminato e ripristinarlo evitando dunque di

utilizzare direttamente **CARDS** come Mazzo di Carte. In effetti all'inizio di una Partita la lista **onGameCardList** contiene gli stessi elementi di **CARDS**.

- **offGameCardList**: **ArrayList** di oggetti con parametro di tipo **Card** che contiene tutte le Carte del Mazzo "non in uso" per una specifica partita. Con il termine "non in uso" si intende riferirsi a tutte quelle Carte del Mazzo che sono già state distribuite ai Giocatori ma non più in possesso degli stessi a causa del fatto che sono Carte relative ad una Mano già conclusa oppure perché relativa ad una giocata conclusasi con uno "sballo".

  Sarà molto comodo utilizzare questa Lista per poter attingere ad una nuova Carta quando la Lista **onGameCardList** si esaurisce.

- **onGameCardsIterator**: **Iterator** su collezioni di oggetti di tipo **Card** che consente di poter iterare per scorre la Lista delle Carte "in uso" (**onGameCardList**). Si noti come questa variabile sia marcata con il modificatore ***transient*** in quanto non è possibile serializzare un **Iterator**.

- **nextCard**: variabile che memorizza la più recente Carta estratta dal Mazzo e dunque, nei termini dell'astrazione che si è data, si tratta della carta più recente estratta dalla Lista delle Carte "in uso" (**onGameCardList**).

- **emptyDeck**: variabile che marca lo stato di Mazzo esaurito.

La Classe **Deck** dispone inoltre di specifici metodi per eseguire particolari operazioni sul Mazzo e di seguito si analizzano i più significativi:

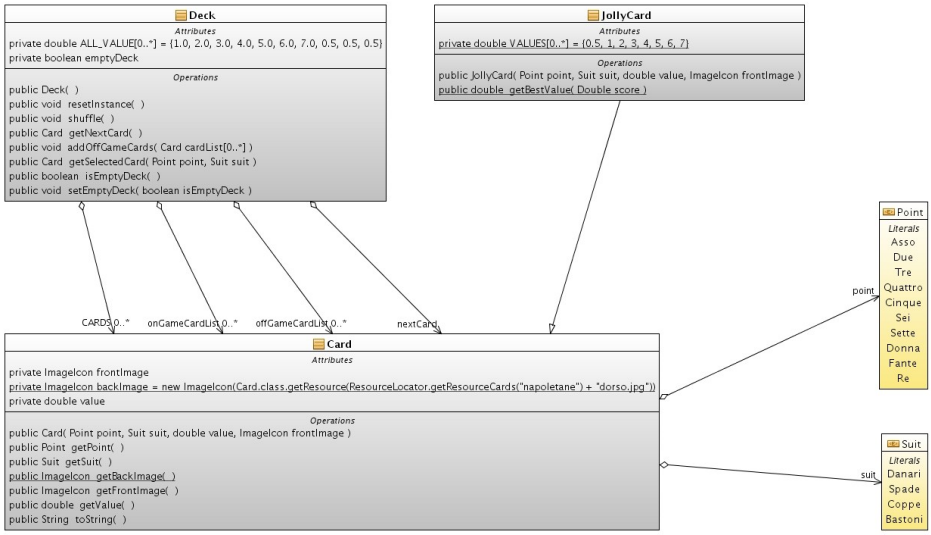
- **Deck**: costruttore utilizzato per istanziare oggetti della classe **Deck**. Si occupa sostanzialmente di creare tutti gli oggetti di tipo Card, di istanziare un oggetto **JollyCard** in corrispondenza della Carta ***Re di Denari***, di riempire un tantum e in modo definitivo il container **CARDS**, di copiare tutto il contenuto di **CARDS** nella Lista delle Carte "in uso" (**onGameCardList**) e di generare il primo **Iterator** su tale Lista ossia **onGameCardsIterator**.

- **resetInstance**: metodo che ha lo scopo di "resettare" un Mazzo di Carte ossia un'istanza di **Deck** dover col termine "resettare" si intende la capacità di ripristinare le proprietà viste poc'anzi ai valori iniziali che assumono quando si ha a che fare con un oggetto **Deck** appena istanziato. Ciò evita la necessità di creare un nuovo oggetto della classe **Deck** ma semplicemente si può utilizzare lo stesso oggetto ripristinando al valore di default le sue proprietà.

- **shuffle**: metodo che ha lo scopo di implementare il concetto di "mischiare le carte del mazzo", ricorrendo all'omonimo metodo shuffle della classe Java **Collections**. Si noti come in effetti la lista sulla quale si invoca il metodo **shuffle** è **onGameCardList** e non **CARDS**.

- **`getNextCard`**: metodo che ha lo scopo di restituire la prossima Carta dal Mazzo. Si noti come questo metodo non generi mai un'eccezione perché il Mazzo, infatti è sempre in grado di restituire una Carta; ciò in virtù del fatto che il metodo è predisposto a cogliere l'evento in cui il Mazzo è terminato, ossia la Lista delle Carte “in uso” (**`onGameCardList`**) è esaurita e provvedere a ripristinarla attingendo alla Lista delle Carte “non in uso” (**`offGameCardList`**) prima di fornire una nuova Carta.
- **`addOffGameCards`**: metodo che ha lo scopo di aggiungere una Lista di Carte alla Lista delle Carte “non in uso” (**`offGameCardList`**). Come si vedrà successivamente sarà la classe **`GameEngine`** ad invocare questo metodo allorquando una Mano termina oppure il Giocatore produce una giocata “sballata”.

Di seguito si indica il Diagramma UML delle classi del package **`org.smgame.core.card`**.



Di seguito si indica inoltre il codice sorgente di alcune classi del package **`org.smgame.core.card`**.

classe **`Card`** – package **`org.smgame.core.card`**

```
package org.smgame.core.card;

import java.io.Serializable;
import javax.swing.ImageIcon;
import org.smgame.util.ResourceLocator;
```

```
/**Classe Carta
 * rappresenta la singola carta del mazzo
 *
 * @author Traetta  Pasquale 450428
 * @author Mignogna Luca     467644
 */
public class Card implements Serializable {
    //variabili

    private Point point; //punto
    private Suit suit; //seme
    private ImageIcon frontImage;
                private    static    final    ImageIcon    backImage    =    new
ImageIcon(Card.class.getResource(ResourceLocator.getResourceCards("napoletane")
+ "dorso.jpg"));
    private double value; //valore

    /**Costruttore con tre parametri
     *
     * @param point descrizione del punto
     * @param suit seme
     * @param value valore
     * @param frontImage immagine della carta
     */
    public Card(Point point, Suit suit, double value, ImageIcon frontImage) {
        this.point = point;
        this.suit = suit;
        this.value = value;
        this.frontImage = frontImage;
    }

    /**Restituisce il punto
     *
     * @return stringa punto
     */
    public Point getPoint() {
        return this.point;
    }

    /**Restituisce il seme
     *
     * @return stringa seme
     */
    public Suit getSuit() {
        return this.suit;
    }

    /**Restituisce il dorso della carta sotto forma di imageicon
     *
     * @return dorso
     */
    public static ImageIcon getBackImage() {
        return backImage;
    }

    /**restituisce l'immagine della carta
     *
     * @return carta frontale
     */
    public ImageIcon getFrontImage() {
```

```
        return frontImage;
    }

    /**Restituisce il valore della carta
     *
     * @return valore
     */
    public double getValue() {
        return this.value;
    }

    /**Stampa la carta
     * descrizione, seme, valore
     *
     * @return carta stampata
     */
    @Override
    public String toString() {
        return point + " di " + suit + " => " + value;
    }
}//end class
```

classe **JollyCard** – package **org.smgame.core.card**

```
package org.smgame.core.card;

import javax.swing.ImageIcon;

/**Classe jollycard, Gestisce il valore della carta jolly detta anche matta
 *
 * @author Traetta  Pasquale 450428
 * @author Mignogna Luca     467644
 */
public class JollyCard extends Card {

    private static final double[] VALUES = {0.5, 1, 2, 3, 4, 5, 6, 7};

    /**Costruttore
     *
     * @param point punto
     * @param suit seme
     * @param value valore
     * @param frontImage immagine
     */
        public  JollyCard(Point  point,  Suit  suit,  double  value,  ImageIcon
frontImage) {
        super(point, suit, value, frontImage);
    }

    /**Calcola e restituisce il massimo valore della carta jolly
     *
     * @param score punteggio
     * @return valore jolly
     */
    public static double getBestValue(Double score) {
        Double value=VALUES[0], max = score;

        if (score > 7.5) {
            return 0.0;
```

```
        } else {
            for (int i = 0; i < VALUES.length; i++) {
                if (score + VALUES[i] <= 7.5) {
                    value = VALUES[i];
                }
            }
            return value;
        }
    }
}
```

classe **Deck** - package **org.smgame.core.card**

```
package org.smgame.core.card;

import java.io.Serializable;
import java.net.URL;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;
import java.util.Random;
import javax.swing.ImageIcon;
import org.smgame.util.ResourceLocator;

/**Classe Mazzo
 * contiene le 40carte da gioco
 *
 * @author Traetta  Pasquale 450428
 * @author Mignogna Luca     467644
 */
public class Deck implements Serializable {

    private final double[] ALL_VALUE = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 0.5,
0.5, 0.5};//tutti i valori
    private final ArrayList<Card> CARDS = new ArrayList<Card>();
    private ArrayList<Card> onGameCardList = new ArrayList<Card>();
    private ArrayList<Card> offGameCardList = new ArrayList<Card>();
    private transient Iterator<Card> onGameCardsIterator;
    private Card nextCard; //prossima carta
    private boolean emptyDeck;

    /**Costruttore
     *
     */
    public Deck() {
        String img = "";
        ImageIcon frontImage;
        int i;
        Card c;

        for (Suit suit : Suit.values()) {
            i = 0;
            for (Point point : Point.values()) {
                if (suit == Suit.Bastoni) {
                    img = "B";
                } else if (suit == Suit.Coppe) {
                    img = "C";
                } else if (suit == Suit.Danari) {
```

```
                    img = "D";
                } else if (suit == Suit.Spade) {
                    img = "S";
                }

                if (i < 9) {
                    img += "0";
                }
                img += i + 1 + ".jpg";

                                                        String resource =
ResourceLocator.getResourceCards("napoletane") + img;
                URL image = ResourceLocator.convertStringToURL(resource);
                frontImage = new ImageIcon(image);
                if (point == Point.Re && suit == Suit.Danari) {
                    c = new JollyCard(point, suit, ALL_VALUE[i], frontImage);
                } else {
                    c = new Card(point, suit, ALL_VALUE[i], frontImage);
                }
                CARDS.add(c);

                i++;
            }
        }

        onGameCardList.addAll(CARDS);
        onGameCardsIterator = onGameCardList.iterator();
    } //end costruttore

    /**Resetta l'istanza del mazzo
     *
     */
    public void resetInstance() {
        onGameCardList.clear();
        offGameCardList.clear();
        onGameCardList.addAll(CARDS);
        onGameCardsIterator = onGameCardList.iterator();
        emptyDeck = false;
    }

    /**Mischia il mazzo
     *
     */
    public void shuffle() {
                                        Collections.shuffle(onGameCardList,     new
Random(System.currentTimeMillis()));
    }

    /**Restituisce la prossima carta
     *
     * @return prossima carta
     */
    public Card getNextCard() {

        if (onGameCardsIterator == null) {
            onGameCardsIterator = onGameCardList.iterator();
        }

        if (!onGameCardsIterator.hasNext()) {
            onGameCardList.addAll(offGameCardList);
            offGameCardList.clear();
```

```
                                        Collections.shuffle(onGameCardList,   new
Random(System.currentTimeMillis()));
            onGameCardsIterator = onGameCardList.iterator();
            emptyDeck = true;
        }

        nextCard = (Card) onGameCardsIterator.next();
        onGameCardsIterator.remove();
        return nextCard;
    }//end getNextCard

    /**Aggiunge la lista di carte alla lista di carte fuori gioco
     *
     * @param cardList lista di carte
     */
    public void addOffGameCards(List<Card> cardList) {
        offGameCardList.addAll(cardList);
    }

    /**Restituisce la carta selezionata
     *
     * @param point punto
     * @param suit seme
     * @return carta selezionata
     */
    public Card getSelectedCard(Point point, Suit suit) {
        for (Card c : CARDS) {
            if (c.getPoint() == point && c.getSuit() == suit) {
                return c;
            }
        }
        return null;
    }

    /**Restituisce lo stato del mazzo vuoto
     *
     * @return booleano di risposta
     */
    public boolean isEmptyDeck() {
        return emptyDeck;
    }

    /**Imposta lo stato del mazzo vuoto
     *
     * @param isEmptyDeck booleano
     */
    public void setEmptyDeck(boolean isEmptyDeck) {
        this.emptyDeck = isEmptyDeck;
    }
} //end class
```

3.6 - package org.smgame.core.player

Il package **`org.smgame.core.player`** contiene tutte le classi utili a modellare opportunamente un Giocatore e l'insieme dei Giocatori di una Partita. Di seguito si analizzano le classi contenute nel package:

- **`PlayerStatus`**: classe di tipo **`Enum`** che raccoglie i possibili stati in cui può trovarsi un Giocatore durante una Mano.
- **`PlayerRole`**: classe di tipo **`Enum`** che raccoglie i possibili ruoli che può assumere un giocatore.
- **`Player`**: classe che modella un Giocatore Umano che partecipa ad una partita. Di seguito si analizzano le principali proprietà di cui è dotata questa classe:
    - **`name`** (nome): variabile relativa al nome del Giocatore.
    - **`credit`** (diponibilità): variabile relativa alla disponibilità economica residua in un dato istante per il Giocatore.
    - **`lastWinLoseAmount`** (ultima vincita/perdita): variabile relativa all'ammontare della vincita/perdita conseguita nell'ultima mano giocata
    - **`cardList`** (Lista di Carte): **`ArrayList`** di oggetti con parametro di tipo **`Card`** che rappresenta la Lista delle Carte che sono state date dal Giocatore in una specifica Mano
    - **`betList`** (Lista di puntate): **`ArrayList`** di oggetti con parametro di tipo della classe wrapper **`Double`** che rappresenta la Lista delle Puntate che il Giocatore ha effettuato nell'ultima Mano. Si noti come in effetti nell'implementazione fornita ogni Giocatore può puntare solo una volta; costituisce infatti una possibile estensione di progetto la possibilità di consentire più di una puntata in considerazione delle diverse varianti esistenti del Gioco Italiano del Sette e Mezzo.
    - **`role`** (ruolo): variabile di tipo **`PlayerRole`** allo scopo di individuare il Ruolo che ha il Giocatore in una specifica Mano.
    - **`status`** (stato di gioco): variabile di tipo **`PlayerStatus`** allo scopo di individuare lo Status che ha assunto il Giocatore in una specifica Mano.
    - **`playerList`** (Lista di Giocatori): variabile di tipo **`PlayerList`** allo scopo di individuare la Lista dei Giocatori cui il Giocatore appartiene in quanto ogni Giocatore può e deve avere visibilità su tale Lista di cui si è fornita l'astrazione con la classe **`PlayerList`**.

  La Classe **`Player`** dispone inoltre di specifici metodi e di seguito si analizzano i più significativi:

    - **`Player`**: costruttore utilizzato per istanziare oggetti della classe **`Player`** per i quali setta in automatico **`name`** e **`status`**.
    - **`getScore`**: metodo che restituisce il punteggio ottenuto in una specifica Mano dal Giocatore calcolato eseguendo la somma dei valori delle singole Carte presenti nella **`cardList`** e valutando opportunamente il valore della Matta qualora se ne sia in possesso.

      Questo metodo è ovviamente di fondamentale importanza per

poter stabilire, alla fine di una Mano, attraverso il confronto dei punteggi ottenuti, se il Giocatore ha conseguito una vincita o una perdita.

- **`getVisibleScore`**: metodo che restituisce il punteggio ottenuto dal Giocatore relativo esclusivamente alle Carte Visibili. Questo metodo è importante per poter implementare un meccanismo sensato e ragionevole di Ragionamento Automatico per un Giocatore che non sia Umano e che ricopra il ruolo di Mazziere; infatti è attraverso l'analisi del punteggio visibile di ogni **`Player`** che il Mazziere non Umano può prendere delle decisioni relative alla sua condotta di gioco al fine di evitare quanto più possibile una perdita.
- **`hasSM`**: metodo che verifica se il Giocatore in una specifica Mano ha realizzato il punteggio massimo ossia 7.5.
- **`hasKingSM`**: metodo che verifica se il Giocatore in una specifica Mano ha realizzato cosiddetto “Sette e Mezzo Reale” ossia punteggio massimo con sole due Carte.
- **`hasJollyKingSM`**: metodo che verifica se il Giocatore in una specifica Mano ha realizzato cosiddetto “Sette e Mezzo Reale con la Matta” ossia punteggio massimo con sole due Carte di cui una è la Matta.
- **`getStake`**: metodo che restituisce la puntata complessiva del Giocatore in una specifica Mano. Ancora una volta si fa notare come in effetti nell'implementazione fornita ogni Giocatore può puntare solo una volta; costituisce infatti una possibile estensione di progetto la possibilità di consentire più di una puntata in considerazione delle diverse varianti esistenti del Gioco Italiano del Sette e Mezzo.

- **`CPUPlayer`**: è la sottoclasse di Player utile a contraddistinguere un comportamento particolare attribuito al Giocatore non Umano ossia Artificiale. Tale comportamento peraltro è stato modellato affinché sia distinto allorquando riveste il Ruolo di Banco rispetto a quando riveste il Ruolo di Giocatore Normale. Di seguito si analizzano le principali proprietà di cui è dotata questa classe:
  - **`MIN_SCORE`** (punteggio minimo): variabile che individua il punteggio minimo che un Giocatore Artificiale deve conseguire prima di poter considerare conclusa la sua giocata. Tale valore è stato attribuito in modo del tutto arbitrario considerandolo ragionevole in quando è un punteggio né troppo basso né troppo alto e tale per cui chiedere una ulteriore Carta rischierebbe di condurre il Giocatore ad una giocata “sballata”. Si noti che questa variabile assume valore costante in quanto non si è ritenuto di doverla mai modificare durante tutta la partita.
  - **`valueWeightMap`** (Mapping Valore/Peso): **`HashMap`** di coppie di valori **`Double`** dove il primo rappresenta uno degli 8 possibili valori che una Carta può avere e l'altro è un peso su base 1000 attribuito a

tale valore. Questo metodo è utilizzato per stabilire quanto deve essere la puntata del Giocatore Artificiale sulla base della prima Carta che riceve considerando che la puntata sarà unica. In questo senso il valore del peso associato al valore della Carta è da considerarsi su base 1000 rispetto al Credito residuo del Giocatore Artificiale. In altri termini se il Giocatore avesse come prima carta il “6 di Spade” e avesse 850 Euro come Credito residuo, la sua puntata corrisponderà a (20.00 * 850.00) / 1000.00 dove appunto il valore 20.00 è ottenuto dalla HashMap.

L'attribuzione dei pesi in anche in questo caso è stata fatta su valutazioni di ragionevolezza sul punteggio posseduto.

- **`isGoodScore`**: metodo che valuta se la giocata del Giocatore Artificiale deve terminare o meno. L'implementazione di tale valutazione si differenzia sulla base del Ruolo assunto in una certa Mano da parte del Giocatore Artificiale.
- **`requestBet`**: metodo che richiede al Giocatore Artificiale di fornire la sua Puntata per una giocata di una specifica Mano. Anche in questo caso si valuta il caso che il Giocatore Artificiale ricopra il ruolo di Banco nel qual caso non deve eseguire alcuna Puntata come da Regolamento.

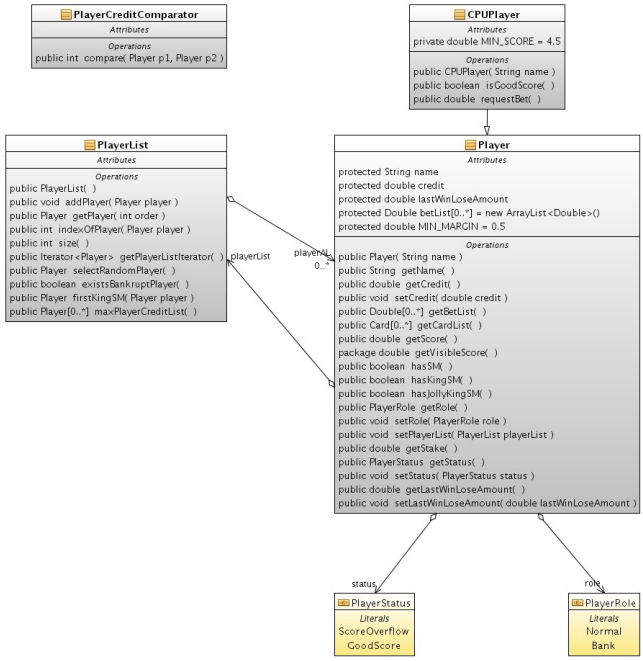
- **`PlayerList`**: è la classe che modella una generica Lista di Giocatori offrendo un'astrazione su come effettivamente tale Lista sia implementata e fornendo dei metodi per accedere alle informazioni che tale Lista può dare. Di seguito si analizzano in dettaglio le principali proprietà di questa classe:
  - **`playerAL`**: **`ArrayList`** di oggetti **`Player`** che rappresenta la vera implementazione della Lista dei Giocatori.

  La Classe **`PlayerList`** dispone di specifici metodi per eseguire particolari operazioni sulla Lista quali aggiunta di un Giocatore Umano o Artificiale, restituizione di un Giocatore sulla base dell'indice che occupa nella Lista, restituzione della dimensione della Lista ossia del numero dei Giocatori. Inoltre sono presenti ulteriori metodi che si analizzano di seguito:
  - **`getPlayerListIterator`**: metodo che fornisce un iteratore sui Giocatori della Lista in modo da poterla scorrere dall'inizio alla fine.
  - **`selectRandomPlayer`**: metodo che fornisce un Giocatore a caso dalla Lista. Questo metodo è di fondamentale importanza per attribuire all'inizio di una partita il Ruolo di Banco.
  - **`existsBankruptPlayer`**: metodo che fornisce un valore boolean se esiste o meno un Giocatore nella Lista che abbia un credito residuo negativo il che, come si vedrà è motivo sufficiente a far concludere una Partita.
  - **`firstKingSM`**: metodo che fornisce il primo Giocatore della Lista a partire dal Giocatore fornito come parametro ad aver ottenuto un “Sette e Mezzo Reale”. Questo metodo è importante per stabilire se

esista a quale sia tale Giocatore al fine di poter attribuire il Ruolo di Banco.

- **`PlayerCreditComparator`**: è la classe che specifica un criterio di ordinamento tra oggetti di tipo Player che sarà utilizzato dal metodo **`maxPlayerCreditList`** della classe **`PlayerList`** per poter determinare quei Giocatori col massimo credito residuo alla fine di ogni manche di una Partita

Di seguito si indica il Diagramma UML delle classi del package **`org.smgame.core.player`**.



Di seguito si indica inoltre il codice sorgente di alcune classi del package **`org.smgame.core.player`**.

classe **`player`** - package **`org.smgame.core.player`**

```
package org.smgame.core.player;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import org.smgame.core.card.Card;
import org.smgame.core.card.JollyCard;
import org.smgame.core.card.Point;
import org.smgame.core.card.Suit;

/**Classe astratta Giocatore
 *
```

```
 * @author Traetta  Pasquale 450428
 * @author Mignogna Luca     467644
 */
public class Player implements Serializable {

    protected String name; //nome giocatore
    protected double credit; //credito
       protected double lastWinLoseAmount; //ammontare della vincita/perdita
relativa all'ultima giocata
    //protected boolean hasJollyCard = false;
     protected ArrayList<Card> cardList = new ArrayList<Card>(); //lista delle
carte possedute in una specifica manche
      protected ArrayList<Double> betList = new ArrayList<Double>(); //lista
delle puntate in una specifica manche
    protected double MIN_MARGIN = 0.5;
    protected PlayerRole role; //ruolo del giocatore in una specifica manche
     protected PlayerStatus status; //status del giocatore dopo la sua giocata
in una specifica manche
    protected PlayerList playerList; //lista dei giocatori cui questo giocatore
appartiene

    /**Costruttore
     *
     * @param name
     */
    public Player(String name) {
        this.name = name;
        this.role = PlayerRole.Normal;
    }

    /**Restituisce il nome giocatore
     *
     * @return nome
     */
    public String getName() {
        return name;
    }

    /**Restituisce il credito corrente
     *
     * @return credito
     */
    public double getCredit() {
        return credit;
    }

    /**imposta il credito corrente
     *
     * @param credit credito
     */
    public void setCredit(double credit) {
        this.credit = credit;
    }

    /**Restituisce  la lista delle puntate
     *
     * @return lista puntate
     */
    public List<Double> getBetList() {
        return betList;
    }
```

```
    /**Restituisce la lista delle carte
     *
     * @return lista di card
     */
    public List<Card> getCardList() {
        return cardList;
    }

    /**Restituisce il punteggio
     *
     * @return punteggio
     */
    public double getScore() {
        double bestValue;
        double score = 0.00;
        boolean hasJollyCard = false;

        for (Card c : cardList) {
            if (c instanceof JollyCard) {
                hasJollyCard = true;
            } else {
                score += c.getValue();
            }
        }

        if (hasJollyCard) {
            bestValue = JollyCard.getBestValue(score);
            score += bestValue;
        }

        return score;
    }

    /**restituisce il punteggio visibile
     *
     * @return punteggio
     */
    double getVisibleScore() {
        Card firstCard = cardList.get(0);
        double score = 0.00;

        if (status == PlayerStatus.ScoreOverflow || hasSM()) {
            return getScore();
        } else {
            if (firstCard instanceof JollyCard) {
                for (Card c : cardList) {
                    if (!c.equals(firstCard)) {
                        score += c.getValue();
                    }
                }
                return score;
            } else {
                return getScore() - firstCard.getValue();
            }
        }
    }

    /**verifica se c'è sette e mezzo
     *
     * @return true = sette e mezzo
```

```
     */
    public boolean hasSM() {
        if (getScore() == 7.5) {
            return true;
        }
        return false;
    }

    /**verifica se c'è sette e mezzo reale
     *
     * @return true = sm reale
     */
    public boolean hasKingSM() {
        if (hasSM() && cardList.size() == 2) {
            return true;
        }
        return false;
    }

    /**Verifica se c'è sette e mezzo reale con matta
     *
     * @return true = sm reale matta
     */
    public boolean hasJollyKingSM() {
        if (hasKingSM() &&
                                ((getCardList().get(0).getPoint()  ==  Point.Re  &&
getCardList().get(0).getSuit() == Suit.Danari) ||
                                 (getCardList().get(1).getPoint()  ==  Point.Re  &&
getCardList().get(1).getSuit() == Suit.Danari))) {
            return true;
        }
        return false;
    }

    /**Restituisce il ruolo corrente
     * se mazziere o giocatore
     *
     * @return ruolo
     */
    public PlayerRole getRole() {
        return role;
    }

    /**imposta il ruolo corrente
     *
     * @param role ruolo
     */
    public void setRole(PlayerRole role) {
        this.role = role;
    }

    /**imposta la lista dei giocatori
     *
     * @param playerList lista
     */
    public void setPlayerList(PlayerList playerList) {
        this.playerList = playerList;
    }

    /**Restituisce la puntata
     *
```

```
     * @return puntata
     */
    public double getStake() {
        double stake = 0;

        for (Double b : betList) {
            stake += b.doubleValue();
        }

        return stake;
    }

    /**restituisce lo stato
     *
     * @return stato
     */
    public PlayerStatus getStatus() {
        return status;
    }

    /**imposta lo stato
     *
     * @param status stato
     */
    public void setStatus(PlayerStatus status) {
        this.status = status;
    }

    /**Restituisce l'ammontare dell'ultima vincita/perdita
     *
     * @return vincita/perdita
     */
    public double getLastWinLoseAmount() {
        return lastWinLoseAmount;
    }

    /**Imposta l'ammontare dell'ultima vincita/perdita
     *
     * @param lastWinLoseAmount ultimo ammontare
     */
    public void setLastWinLoseAmount(double lastWinLoseAmount) {
        this.lastWinLoseAmount = lastWinLoseAmount;
    }

} //end class
```

classe **PlayerCreditComparator** - package **org.smgame.core.player**

```
package org.smgame.core.player;

import java.util.Comparator;

/**Classe Comparatore di crediti giocatori
 * lo scopo è quello di trovare il massimo credito
 *
 * @author Traetta  Pasquale 450428
 * @author Mignogna Luca     467644
 */
public class PlayerCreditComparator implements Comparator<Player> {
```

```
    @Override
    public int compare(Player p1, Player p2) {
        if (p1.getCredit() > p2.getCredit()) {
            return -1;
        } else if (p1.getCredit() < p2.getCredit()) {
            return +1;
        } else {
            return 0;
        }
    }
}
```

classe **PlayerList** - package **org.smgame.core.player**

```
package org.smgame.core.player;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
import java.util.Random;

/**Classe lista giocatori
 *
 * @author Traetta  Pasquale 450428
 * @author Mignogna Luca     467644
 */
public class PlayerList implements Serializable {

    private LinkedList<Player> playerAL = new LinkedList<Player>();

    /**Costruttore vuoto
     *
     */
    public PlayerList() {
    }

    /**Aggiunge giocatore alla lista
     *
     * @param player giocatore
     */
    public void addPlayer(Player player) {
        playerAL.add(player);
    }

    /**REstituisce giocatore n della lista
     *
     * @param order numero lista
     * @return giocatore
     */
    public Player getPlayer(int order) {
        return playerAL.get(order);
    }

    /**restituisce posizione nella lista
     *
```

```
     * @param player giocatore
     * @return posizione
     */
    public int indexOfPlayer(Player player) {
        return playerAL.indexOf(player);
    }

    /**Restituisce dimensione lista
     *
     * @return dimensione
     */
    public int size() {
        return playerAL.size();
    }

    /**Retituisce la lista iterata dei giocatori
     *
     * @return lista iterata
     */
    public Iterator<Player> getPlayerListIterator() {
        return playerAL.iterator();
    }

    /**Restituisce un player casuale
     *
     * @return player random
     */
    public Player selectRandomPlayer() {
        List<Player> tempList = new ArrayList<Player>(playerAL);
        Collections.shuffle(tempList, new Random(System.currentTimeMillis()));
        return tempList.get(0);
    }

    /**Verifica se esiste almeno un player in bancarotta
     *
     * @return true = player bancarotta
     */
    public boolean existsBankruptPlayer() {
        for (Player p : playerAL) {
            if (p.getCredit() <= 0.00) {
                return true;
            }
        }
        return false;
    }

     /**Restituisce il primo giocatore ad aver fatto SMreale per il cambio di
mazziere
     *
     * @param player vecchio mazziere
     * @return nuovo mazziere
     */
    public Player firstKingSM(Player player) {
        ArrayList<Player> tempList = new ArrayList(playerAL);
        int playerIndex = playerAL.indexOf(player);

        Collections.rotate(tempList, -playerIndex);

        for (Player p : tempList) {
            if (p.hasKingSM()) {
                return p;
```

```
            }
        }
        return null;
    }

    /**Restituisce la lista di giocatori ordinata per il massimo credito
     *
     * @return lista giocatori
     */
    public List<Player> maxPlayerCreditList() {
        int i;
        boolean isMaxCredit = true;
        ArrayList<Player> playerSubList = new ArrayList<Player>();
        ArrayList<Player> tempList = new ArrayList<Player>(playerAL);

        Collections.sort(tempList, new PlayerCreditComparator());

        i = 0;
        while (isMaxCredit && i < tempList.size()) {
            if (tempList.get(i).getCredit() != tempList.get(0).getCredit()) {
                isMaxCredit = false;
            } else {
                playerSubList.add(tempList.get(i));
                i++;
            }
        }

        return playerSubList;
    }
}
```

3.7 - package `org.smgame.resource.cardimage.napoletane`

Il package **`org.smgame.resource.cardimage.napoletane`** ospita le Immagini delle Carte Napoletane utilizzate per il progetto. Dato che le Carte del Mazzo sono differenti per Regioni, è ragionevole supporre di poter accogliere diverse versioni di tali Carte e di consentire all'utente di poter scegliere il Mazzo di Carte Regionale che preferisce.

3.8 - package `org.smgame.resource.authorimage`

Il package **`org.smgame.resource.authorimage`** ospita le Immagini degli avatar degli autori del progetto.

3.9 - package `org.smgame.server`

Il package **org.smgame.server** contiene tutte le classi utili a realizzare un server RMI e consentire dunque ad un client di potersi connettere.

Il server RMI ammette connessioni multiple dunque è possibile giocare più

Partite contemporaneamente.

L'implementazione del server è stata fatta considerando le nuove feature introdotte già nella versione Java 5 e di seguito si analizzano le classi contenute nel package:

- **`IStub`**: è l'interfaccia che rappresenta il tipo di oggetto **`Stub`** che sarà passato dal server al client nella chiamata di metodi remoti.
- **`RMIServer`**: è la classe che implementa il server vero e proprio implementato secondo il pattern Singleton. Di seguito si analizzano in dettaglio le principali proprietà di questa classe:
  - **`RMIServer()`**: è il costruttore della classe **RMIServer** e si occupa di determinare qual è l'ambiente di Sistema Operativo su cui il server sta girando per poter settare opportunamente il comando da lanciare per avviare il Registry RMI.

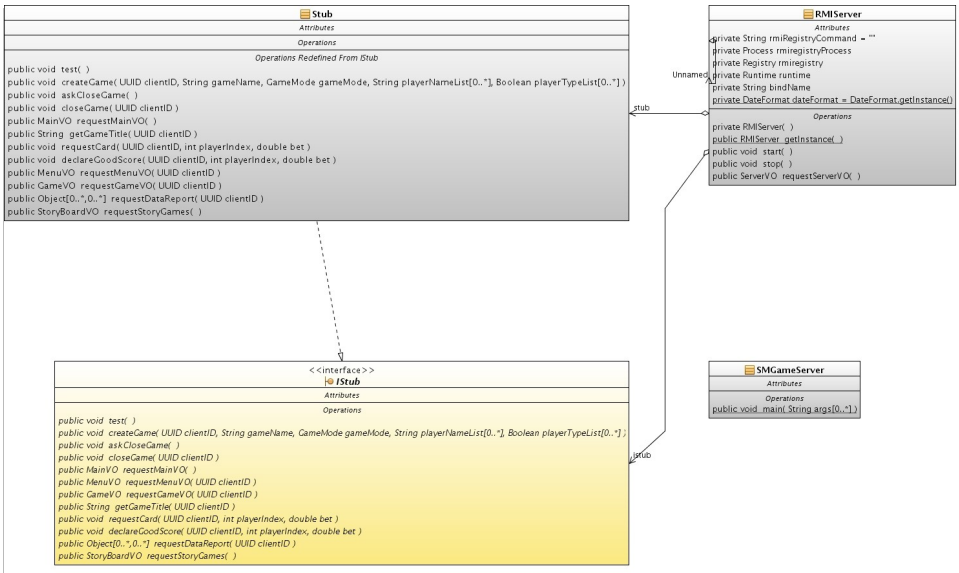
    Utilizza quindi la classe Java Runtime per avvire il Registry RMI.

    Definisce inoltre un nome per il servizio di cui sarà eseguito il binding e che sarà utilizzato dal client per ottenere un oggetto remoto.
  - **`start()`**: è il metodo che consente di avviare il server infatti si avvia il processo relativo al Registry RMI, si crea l'oggetto di tipo **`Stub`** e lo si esporta in modo tale da avere un oggetto di tipo **`IStub`**.

    Inoltre il metodo esegue anche del logging minimale che è possibile visualizzare nell'interfaccia di frontend del Server.
  - **`stop()`**: è il metodo che consente di arrestare il server infatti verifica che non esista un processo associato al Registry RMI e se così è lo distrugge eseguendo anche l'unbinding dell'oggetto **`IStub`**.

    Inoltre il metodo esegue anche del logging minimale che è possibile visualizzare nell'interfaccia di frontend del Server.
- **`Stub`**: è la classe concreta che implementa i metodi definiti nell'interfaccia **`IStub`**.
- **`SMGameServer`**: è la classe che contiene il metodo **`main`** che ha lo scopo di creare e quindi mostrare la GUI con cui è possibile avviare o fermare il server

Di seguito si indica il Diagramma UML delle classi del package **`org.smgame.server`**.




```
    private IStub istub;
    private Stub stub;
    private static ServerVO serverVO = new ServerVO();
    private String rmiRegistryCommand = "";
    private Process rmiregistryProcess;
    private Registry rmiregistry;
    private Runtime runtime;
    private String bindName;
    private static final DateFormat dateFormat = DateFormat.getInstance();

    /**Costruttore privato
     *
     */
    private RMIServer() {
        // Must implement constructor to throw RemoteException:
        runtime = Runtime.getRuntime();

        if (ResourceLocator.isWindows()) {
            rmiRegistryCommand = "rmiregistry.exe";
        } else {
            rmiRegistryCommand = "rmiregistry";
        }
        bindName = "rmi://localhost/ServerMediator";
    }

    /**restituisce l'istanza del server
     *
     * @return istanza server, se nulla, viene creata
     */
    public static RMIServer getInstance() {
        if (server == null) {
            server = new RMIServer();
        }
```

```
        return server;
    }

    /**Lancia il server
     *
     */
    public void start() {
        serverVO.clear();

        try {
            rmiregistryProcess = runtime.exec(rmiRegistryCommand);
            Thread.sleep(3000);

            rmiregistry = LocateRegistry.getRegistry();

            stub = new Stub();

            istub = (IStub) UnicastRemoteObject.exportObject(stub, 0);
            rmiregistry.rebind(bindName, istub);

              serverVO.setMessage(dateFormat.format(new Date()) + "- RMI Server
Avviato su localhost");
            serverVO.setMessageType(MessageType.INFO);
                Logging.logInfo(dateFormat.format(new Date()) + "- RMI Server
Avviato su localhost");
        } catch (Exception e) {
            serverVO.setMessage("Impossibile avviare il server");
            serverVO.setMessageType(MessageType.ERROR);
            Logging.logExceptionSevere(this.getClass(), e);
        }
    }

    /**Arresta il server
     *
     */
    public void stop() {
        if (rmiregistryProcess != null) {
            try {
                UnicastRemoteObject.unexportObject(stub, true);
                rmiregistryProcess.destroy();
                stub = null;
                rmiregistryProcess = null;
                        serverVO.setMessage(dateFormat.format(new Date()) + "-
RMIServer Interrotto su localhost");
                serverVO.setMessageType(MessageType.INFO);
                Logging.logInfo("RMIServer Interrotto su localhost");
            } catch (Exception e) {
            }
        }
    }

    /**Richiede il value objects server
     *
     * @return serverVO
     */
    public ServerVO requestServerVO() {
        return serverVO;
    }
}
```

classe **IStub** - package **org.smgame.server**

```
package org.smgame.server;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.List;

import java.util.UUID;
import org.smgame.client.frontend.MenuVO;
import org.smgame.client.frontend.GameVO;
import org.smgame.client.frontend.MainVO;
import org.smgame.client.frontend.StoryBoardVO;
import org.smgame.core.GameMode;

/**Interfaccia mediatore di gioco
 *
 * @author Traetta  Pasquale 450428
 * @author Mignogna Luca     467644
 */
public interface IStub extends Remote {

    /**testa la connessione
     *
     * @throws java.rmi.RemoteException
     */
    public void test() throws RemoteException;

    /**Crea partita
     *
     * @param clientID identificativo unico
     * @param gameName nome partita
     * @param gameMode tipo partita
     * @param playerNameList lista nomi
     * @param playerTypeList lista tipo
     *
     * @throws java.rmi.RemoteException
     */
    public void createGame(UUID clientID, String gameName, GameMode gameMode,
              List<String> playerNameList, List<Boolean> playerTypeList) throws
RemoteException;

    /**chiedi chiusura partita
     *
     * @throws java.rmi.RemoteException
     */
    public void askCloseGame() throws RemoteException;

    /**Chiudi partita
     *
     * @param clientID identificativo unico
     *
     * @throws java.rmi.RemoteException
     */
    public void closeGame(UUID clientID) throws RemoteException;

    /**richiede e restituisce l'oggetto mainVO
     *
     * @return mainVO
```

```
     *
     * @throws java.rmi.RemoteException
     */
    public MainVO requestMainVO() throws RemoteException;

    /**Richiede e restituisce l'oggetto menuVO per il menù
     *
     * @param clientID identificativo unico
     * @return menuVO
     *
     * @throws java.rmi.RemoteException
     */
    public MenuVO requestMenuVO(UUID clientID) throws RemoteException;

    /**richide e restituisce l'oggetto gameVO per la partita
     *
     * @param clientID identificativo unico
     * @return gamevo
     *
     * @throws java.rmi.RemoteException
     */
    public GameVO requestGameVO(UUID clientID) throws RemoteException;

    /**restituisce il titolo
     *
     * @param clientID identificativo unico
     * @return titolo
     *
     * @throws java.rmi.RemoteException
     */
    public String getGameTitle(UUID clientID) throws RemoteException;

    /**richiede carta
     *
     * @param clientID identificativo unico
     * @param playerIndex indice giocatore
     * @param bet puntata
     *
     * @throws java.rmi.RemoteException
     */
     public void requestCard(UUID clientID, int playerIndex, double bet) throws
RemoteException;

    /**dichiara di star bene
     *
     * @param clientID identificativo unico
     * @param playerIndex indice giocatore
     * @param bet puntata
     *
     * @throws java.rmi.RemoteException
     */
      public void declareGoodScore(UUID clientID, int playerIndex, double bet)
throws RemoteException;

    /**richiede e restituisce la matrice report
     *
     * @param clientID identificativo unico
     * @return matrice
     *
     * @throws java.rmi.RemoteException
     */
```

```
    public Object[][] requestDataReport(UUID clientID) throws RemoteException;

    /**richiede e restituisce l'oggetto storico partite
     *
     * @return storyboardVO
     *
     * @throws java.rmi.RemoteException
     */
    public StoryBoardVO requestStoryGames() throws RemoteException;
}
```

classe **Stub** - package **org.smgame.server**

```
package org.smgame.server;

import java.rmi.RemoteException;
import java.util.List;

import java.util.UUID;
import org.smgame.core.CoreProxy;
import org.smgame.client.frontend.GameVO;
import org.smgame.client.frontend.MainVO;
import org.smgame.client.frontend.MenuVO;
import org.smgame.client.frontend.StoryBoardVO;
import org.smgame.core.GameMode;

/**Oggetto remoto
 *
 * @author Traetta  Pasquale 450428
 * @author Mignogna Luca     467644
 */
public class Stub implements IStub {

    @Override
    public void test() throws RemoteException { }

    @Override
     public void createGame(UUID clientID, String gameName, GameMode gameMode,
List<String> playerNameList, List<Boolean> playerTypeList) {
            CoreProxy.createGame(clientID, gameName, gameMode, playerNameList,
playerTypeList);
    }

    @Override
    public void askCloseGame() {
        CoreProxy.askCloseGame();
    }

    @Override
    public void closeGame(UUID clientID) {
        CoreProxy.closeGame(clientID);
    }

    @Override
    public MainVO requestMainVO() {
        return CoreProxy.requestMainVO();
    }

    @Override
```

```
    public String getGameTitle(UUID clientID) {
        return CoreProxy.getGameTitle(clientID);
    }

    @Override
    public void requestCard(UUID clientID, int playerIndex, double bet) {
        CoreProxy.requestCard(clientID, playerIndex, bet);
    }

    @Override
    public void declareGoodScore(UUID clientID, int playerIndex, double bet) {
        CoreProxy.declareGoodScore(clientID, playerIndex, bet);
    }

    @Override
    public MenuVO requestMenuVO(UUID clientID) {
        return CoreProxy.requestMenuVO(clientID);
    }

    @Override
    public GameVO requestGameVO(UUID clientID) {
        return CoreProxy.requestGameVO(clientID);
    }

    @Override
    public Object[][] requestDataReport(UUID clientID) {
        return CoreProxy.requestDataReport(clientID);
    }

    @Override
    public StoryBoardVO requestStoryGames() throws RemoteException {
        try {
            return CoreProxy.requestStoryGames();
        } catch (Exception ex) {
            throw new RemoteException();
        }
    }
}
```

3.10 - package `org.smgame.server.frontend`

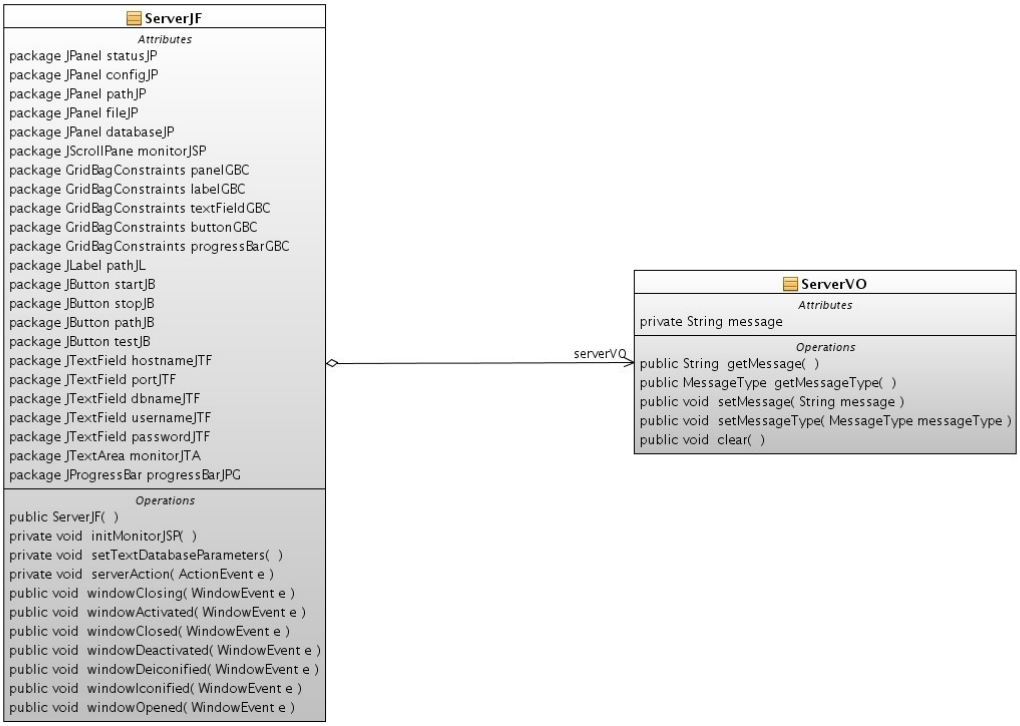
Il Package **`org.smgame.server.frontend`** contiene una classe utile a realizzare una GUI minimale che faccia da frontend per avviare e arrestare il server RMI.

La logica con cui è stata realizzata è la stessa con cui sono state realizzate le altre classi di frontend del package **`org.smgame.cient.frontend`**. Tale interfaccia fornisce inoltre una parte di Test della connessione verso il backend costituito dal Database MySQL ed una sezione di monitoraggio dell'attività del server.

La GUI anche in questo caso non ha conoscenza di dover interagire con un server RMI esegue solo delle chiamate a metodi in risposta a certi eventi che si verificano sull'interfaccia Swing.

La GUI ricevere sempre e comunque un oggetto ServerVO che è un Valued Object che contiene un messaggio.

Di seguito si indica il Diagramma UML delle classi del package **`org.smgame.server.frontend`**:



3.11 - package `org.smgame.util`

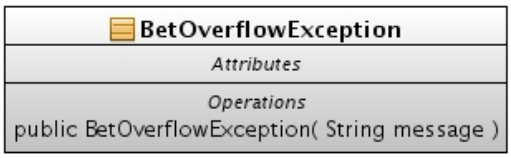
Il package **`org.smgame.util`** contiene classi di utilità che possono essere utilizzate da tutte le altre classi di altri package.

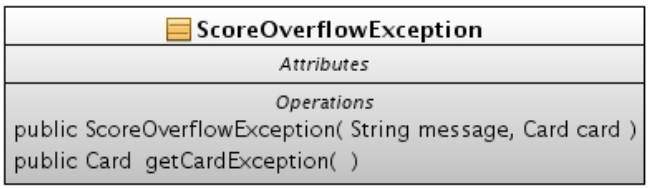
Di seguito si analizzano le classi contenute nel package:

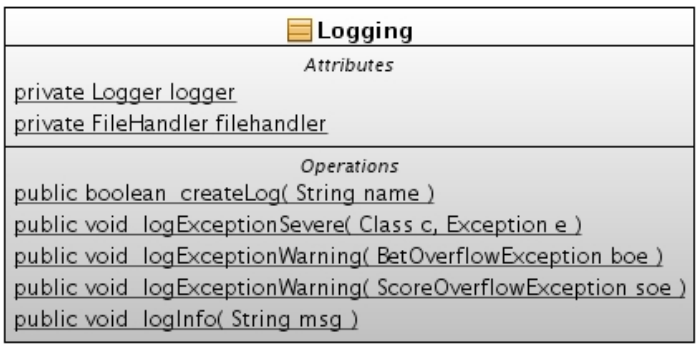
- **`BetOverFlowException`**: è l'eccezione che viene sollevata non appena si verifica che un Giocatore ha eseguito una puntata superiore al suo credito residuo.
- **`ScoreOverFlowException`**: è l'eccezione che viene sollevata non appena si verifica che un Giocatore ha “sballato”.
- **`NoGameException`**: è l'eccezione che viene sollevata non appena si verifica che non esiste alcuna partita da poter caricare.
- **`ImageEdit`**: è una classe di utilità per l'elaborazione delle immagini infatti è dotata di due metodi statici che rispettivamente si occupano di eseguire il resize di un'immagine alle dimensioni opportune e di eseguire una trasformazione di colore per ottenere una immagine in scala di grigio.
- **`Logging`**: è una classe di utilità per eseguire il logging di qualche attività.
- **`ResourceLocator`**: è una classe di utilità per eseguire correttamente la

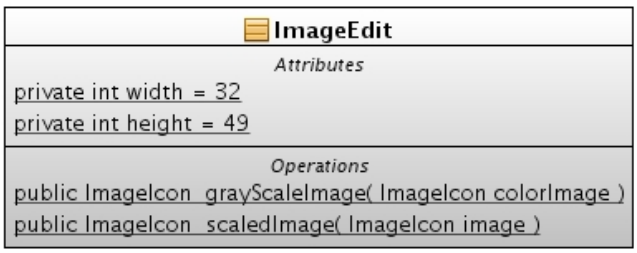
ricerca di file all'interno del sistema che ospita l'applicazione.

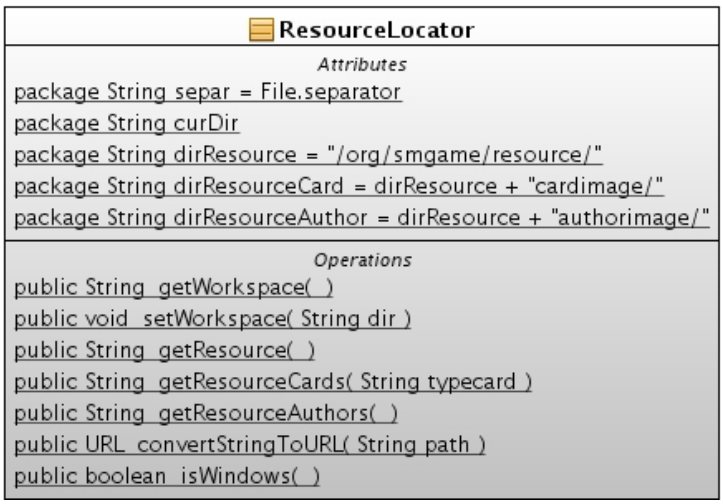
Di seguito si indica il Diagramma UML delle classi del package **`org.smgame.util`**.











4. Progettazione del DATABASE

E’ necessario premettere che per gli scopi del progetto, la progettazione del Database è minimale in quanto molte informazioni sono rese persistenti attraverso gli strumenti di serializzazione su file messi a disposizione da Java. Tuttavia pur tenendo conto delle specifiche originali di progetto si è proceduto ad estendere l'implementazione come dettagliato in seguito.

Lo strumento utilizzato per la modellazione è MySQL WorkBench 5.1.1.6 strumento prodotto da MySQL e dunque ovviamente pienamente integrato con il Database MySQL.

4.1 – Analisi delle Specifiche Progettuali

L’analisi delle specifiche progettuali ha portato ad evidenziare le Entità fondamentali e le associazioni tra esse modellando un quadro abbastanza fedele relativamente alla gestione delle transazioni per ogni Manche di ogni Partita ad SMGame.

4.2 – Schema E/R delle Specifiche Progettuali

L’analisi condotta sulle specifiche di progetto ha consentito di isolare le due seguenti Entità principali che derivano dalle specifiche di progetto originali:

- ***TRANSAZIONE***: una Transazione rappresenta una traccia storica dell'esito di una specifica Manche in una specifica Partita per uno specifico Giocatore. Di conseguenza c'è l'evidenza del punteggio ottenuto dal Giocatore in tale Transazione con l'ammontare della vincita/perdita ed il suo credito residuo.
- ***CARTA***: una Carta rappresenta l'Entità di una Carta del Mazzo. Tale Entità è presumibilmente invariante nel corso del tempo.

Le due Entità poc'anzi illustrate mettono in luce la seguente unica relazione:

- ***carte nella transazione***: una Transazione coinvolge un Giocatore in una specifica Manche di una specifica Partita per il quale vengono pescate un certo numero di Carte. Viceversa una Carta in una Partita può appartenere a più Giocatori anche nella stessa Manche nell'ipotesi che un Giocatore abbia “sballato” e dunque le sue Carte vengono rimescolate.

Tutto ciò si traduce nello schema logico evidenziato in Figura 1.

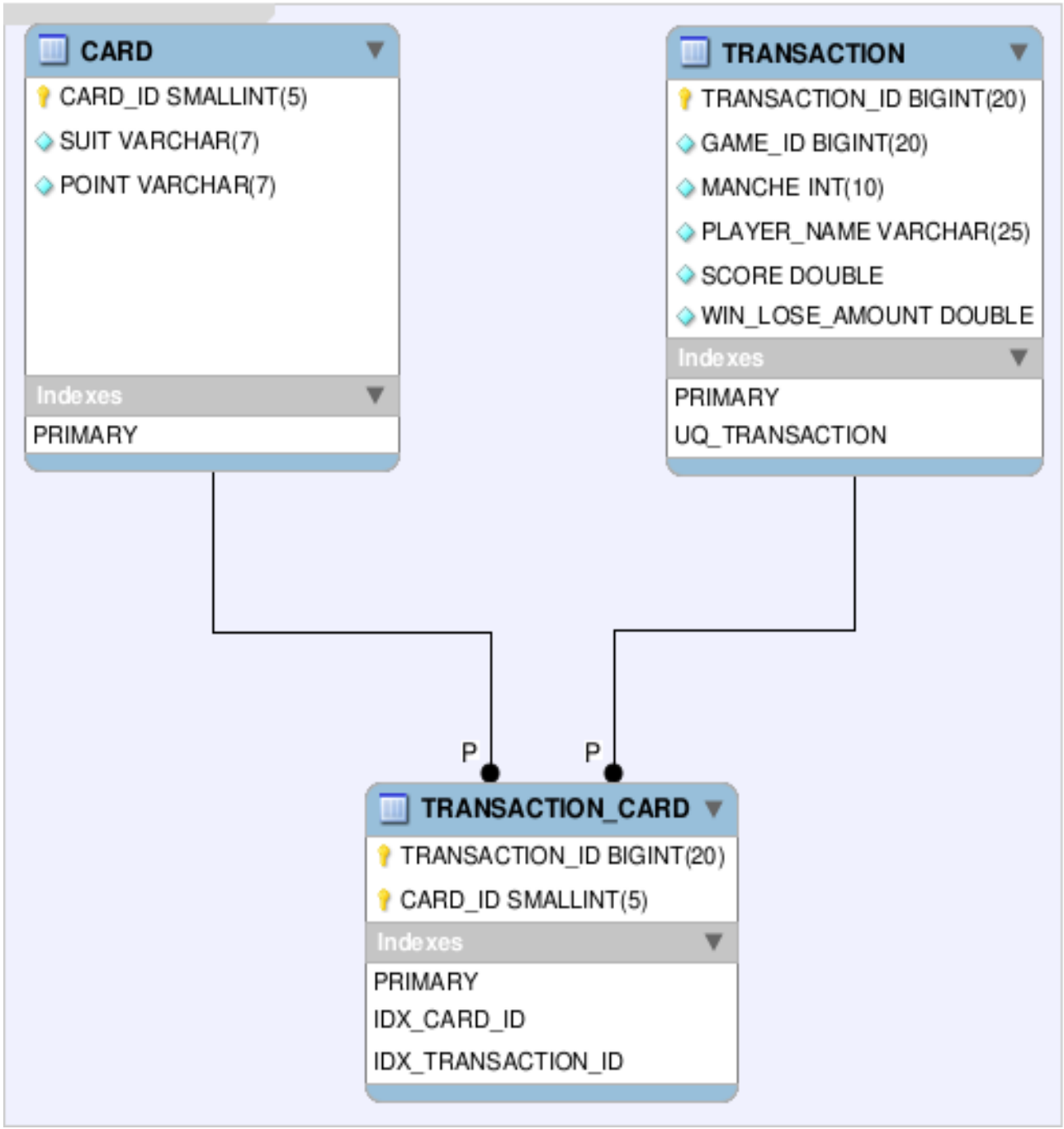


Figura 1: Schema Logico E/R

4.3 - Progettazione dello Schema Logico del Database.

Sono stati utilizzati nomi in inglese ed al singolare per ognuna delle Relazioni in gioco e per i relativi Attributi allo scopo di non incorrere in tipici inconvenienti quali Entità dotate di nomi accentati ecc.... Di seguito si evidenzia la traduzione (peraltro evidente) tra Italiano e Inglese per le relazioni utilizzate e la relativa semantica. Si noti, inoltre, come tale traduzione sia utile solo per comprendere meglio il significato delle relazioni in quanto la versione italiana delle relazioni non è mai stata usata neanche né nel corrispondente schema fisico né nell'implementazione nel corrispondente schema fisico.

Relazione (Italiano)	Relazione (Inglese)	Semantica
TRANSAZIONE	TRANSACTION	Relazione che modella le Transazioni per ogni Manche di una specifica Partita.
CARTA	CARD	Relazione che modella tutte le 40 Carte di un Mazzo.
TRANSAZIONE_CARTA	TRANSACTION_CARD	Relazione introdotta per modellare le informazioni selle Carte pescate per ogni Transazione.

Relazione TRANSACTION

La relazione TRANSACTION è dotata di Attributi minimali per modellare una Transazione relativa ad una Partita ad SMGame.

Attributo	TIPO	Null	Commento
TRANSACTION_ID	BIGINT(20) UNSIGNED	N	Identificativo univoco della Transazione
GAME_ID	BIGINT(20) UNSIGNED	N	Riferimento all''Identificativo univoco di una Partita. Tale Identificativo deve essere presente nel file di salvataggio delle Partite
MANCHE	INT(10) UNSIGNED	N	Numero della Manche per una specifica Partita
PLAYER_NAME	VARCHAR(25)	N	Nome del Giocatore. I Giocatori non sono memorizzati nel Database ma sono serializzati nel file di salvataggio della Partite
SCORE	DOUBLE	N	Punteggio realizzato dal Giocatore in una specifica Manche di una specifica Partita
WIN_LOSE_AMOUNT	DOUBLE	N	Ammontare della vincita/perdita realizzata dal Giocatore in una specifica Manche di una specifica Partita

L’attributo TRANSACTION_ID è Primary Key per la relazione TRANSACTION.

Relazione CARD

La relazione CARD è dotata di Attributi minimali per modellare una Carta nel Mazzo.

Attributo	TIPO	Null	Commento
CARD_ID	SMALLINT(5) UNSIGNED	N	Identificativo Univoco di una Carta nel Mazzo
SUIT	VARCHAR(7)	N	Seme corrispondente della Carta
POINT	VARCHAR(7)	N	Punto corrispondente della Carta

L’attributo CARD_ID è Primary Key per la relazione CARD.

Relazione TRANSACTION_CARD

La relazione TRANSACTION_CARD è la modellazione logica di una relazione M:N tra la relazione TRANSACTION e la relazione CARD e contiene Attributi relativi a collegare una singola Transazione con le Carte pescate per quella Transazione.

Attributo	Tipo	Null	Commento
TRANSACTION_ID	BIGINT(20) UNSIGNED	N	Riferimento all''Identificativo univoco di una Transazione
CARD_ID	SMALLINT(5) UNSIGNED	N	Riferimento all''Identificativo univoco di una Carta

La coppia di attributi TRANSACTION_ID e CARD_ID costituiscono la Primary Key per la relazione TRANSACTION_CARD.

L’attributo TRANSACTION_ID inoltre è Foreign Key che referenzia l’attributo TRANSACTION_ID della relazione TRANSACTION.

L’attributo CARD_ID è inoltre Foreign Key che referenzia l’attributo CARD_ID della relazione CARD.

4.4 - Realizzazione del Database in MySQL

La realizzazione della base di dati può essere fatta importando il dump del Database MySQL del progetto SMGame.

Ovviamente tale dump contiene gli oggetti evidenziati nel paragrafo precedente ed è stato creato con il seguente comando:

```
mysqldump --database smgame > smgame.sql -u root -p
```

Il File smgame.sql creato può essere importato in un qualunque Database Server MySQL con il seguente comando:

```
mysql < smgame.sql –u root -p
```

Si noti che tale script creerà lo Schema **`smgame`** che è quello utilizzato per il progetto. Tale Schema conterrà oggetti creati con l'engine **InnoDB** e con Character Set **cp1252 West European**.

A questo punto sarebbe opportuno creare l'utente **`smgameuser`** con password **`smgamepassword`** attraverso il tool MySQL Administrator per potersi connettere correttamente al Database dall'applicazione.

In caso contrario occorrerà modificare il file di properties **`database.properties`** presente nella directory **`./config`** ed inserire le credenziali dell'utente root per connettersi al Database.

5. Estensioni Progettuali

Il progetto è meritevole di ulteriori estensioni sia in termini di funzionalità sia in termini di maggiore flessibilità rispetto alle tante varianti che il Gioco Italiano del Sette e Mezzo mette a disposizione.

Di seguito si propone una panoramica non esaustiva di tutte le possibili implementazioni suddividendole tra tecniche e di contesto con l'attribuzione a queste ultime dell'accezione data poc'anzi quando si faceva riferimento alle varianti del Gioco esistenti.

5.1 Estensioni Tecniche

L'applicazione consente oggi di poter memorizzare, in caso di partita Offline, le partite salvate su file, tuttavia tale salvataggio offre una fotografia della situazione attuale in un dato istante ma non offre informazioni storiche che sono parzialmente presenti qualora si esegua una partita in modalità Online (Client Server).

L'applicazione nella sua modalità di funzionamento Client/Server consente di poter eseguire una partita scegliendo a priori il numero di Giocatori e fissando a priori la loro tipologia ossia si scelgono N giocatori sempre rispettando il limite di N compreso tra 2 e 12 e di cui N-1 sono obbligatoriamente Giocatori Artificiali. Una possibile estensione potrebbe essere quella che consente di avere un numero variabile di Giocatori tra Umani e Artificiali.

La modalità di gioco Client/Server seppure funzionale non è però completamente rispondente ai requisito di un gioco MultiPlayer propriamente detto. Una importante estensione del Gioco potrebbe essere la realizzazione del vero Gioco Italiano del Sette e Mezzo in modalità Multiplayer; ciò comporterebbe che un Giocatore che vuole eseguire una Partita OnLine potrebbe connettersi al Server e scegliere di partecipare ad una Partita creata da qualche altro utente o dal Server stesso oppure può creare egli stesso una partita nel qual caso indicherà solo il proprio nome e il nome della Partita che vuole creare.

Le tecnologie utilizzate per realizzare l'applicazione consentono di poterne fruire come una classica applicazione Client o via browser web scaricando l'equivalente Applet. Una possibile estensione potrebbe essere quella di rendere l'applicazione di livello Enterprise adottando tecnologie messe a disposizione dal framework J2EE quali Servlet, JSP, JSF, ecc....

Qualora l'applicazione fosse fruita da una gran quantità di utenti e di conseguenza fossero create diverse Partite sarebbe utile e necessario offrire uno strato di persistenza estremamente affidabile. In tal senso si potrebbe pensare di estendere l'applicazione in modo tale da predisporla a salvare i dati su database diversi da MySQL o addirittura si potrebbe pensare di astrarre il livello di persistenza adottando framework di persistenza quali Hibernate che offrono tale possibilità evitando che lo sviluppatore integri nella propria

applicazione frammenti di codice che caricano specifici Driver relativi a specifici Database e quant'altro.

5.2 Estensioni di Contesto

Le ricerche condotte su varie fonti in internet e su qualche testo ci hanno condotto a ritenere che una ufficializzazione e standardizzazione vera e propria delle regole e dello svolgimento del Gioco Italiano del Sette e Mezzo non esista.

Di sicuro quelle tratte dal sito Wikipedia sono le più comuni ma esistono diverse varianti che spesso si diversificano a seconda della regione italiana.

Ciò comporta il fatto che evidentemente è impensabile poter stabilire in autonomia delle regole standard da implementare ma è più sensato progettare l'applicazione affinché sia più flessibile nell'assimilare in modo corretto le varianti esistenti a patto di aver indicato in precedenza, all'atto di creazione di una nuova partita, quali siano le regole condivise che si applicheranno per quella Partita.

Inoltre si da per ovvio il fatto che una volta creata una Partita con certe regole tali regole non cambino per tutta la durata della Partita.

Di seguito si indicano le possibili estensioni di contesto per poter recepire tutte le varianti conosciute del Gioco:

- Esiste una variante per cui il numero minimo e massimo di Giocatori è stabilito in 3 e 9.
- Esiste una variante per cui la regola di determinazione del mazziere all'inizio della partita potrebbe essere a sorte oppure attribuendola a chi pesca la carta più alta nel mazzo di 40 carte.
- Esiste una variante per cui la distribuzione delle carte potrebbe essere in senso antiorario o orario e di conseguenza il primo giocatore di turno sapere rispettivamente quello a destra del Mazziere o quello a sinistra.
- Esiste una variante in cui è data facoltà al Giocatore di “chiamare” una carta con una puntata e di “stare bene” con un'altra. Nel caso in cui la sua giocata conducesse ad uno sballo il Giocatore restituirà immediatamente la carta che ha determinato lo sballo al Mazziere e corrisponderà anche a quest'ultimo la puntata relativa alla carta chiamata. Si noti che fino a questo punto, questa variante è anche citata da Wikipedia come già accennato nel Capitolo 1. Oltre a questa variante definita anche come “gioco doppio” è possibile eseguire un “gioco triplo” o “gioco quadruplo” ecc... a seconda di quanto (ricorsivamente) si applichi la regola del “gioco doppio” ad ogni carta chiamata. In ogni caso tale tipo di giocata ha comunque termine o quando il Giocatore decide di fermarsi o qualora si concluda con uno sballo.
- Esiste una variante che prevede il rimescolamento del Mazzo recuperando tutte le carte giocate non appena si concluda una Manche

che ha visto la presenza della Matta. Nell'implementazione di progetto si ricorda che il rimescolamento del Mazzo è previsto quando questo termina e quando avviene il cambio del ruolo del Mazziere tra due Giocatori.

- Esiste una variante per cui il valore della Matta può assumere un valore qualunque discreto tra 0.5 e 7.5 ad intervallo di 0.5 punti per un totale di 14 valori possibili. Nell'implementazione di progetto si ricorda che la Matta può assumere complessivamente solo 8 valori compresi tra 0.5 e 7 inclusi.
- Esiste una variante che consente di stabilire a priori quale sia la carta tra le 40 del mazzo che assurgerà al ruolo di Matta.
- Esiste una variante che consente ad un Giocatore che riceve come carta un Quattro di poterla scartare e di richiedere al Mazziere una nuova carta prima ancora di eseguire la sua puntata.
- Esiste una variante per cui il Giocatore che abbia realizzato un Sette e Mezzo Reale senza la Matta perde solo se il Mazziere ha realizzato un Sette e Mezzo Reale. Nell'implementazione di progetto al contrario, il Mazziere che abbia realizzato Sette e Mezzo batte tutti i Sette e Mezzo anche se realizzati con due carte a meno che non sia Un Sette e Mezzo Reale con Matta.
- Esiste una variante che non considera legittimo o reale il Sette e Mezzo con 2 carte realizzato con la Matta.
- Esiste una variante che contempla la realizzazione del cosiddetto Sette e Mezzo Triplèe che si ottiene quando un Giocatore riceve come prima carta un Sette e chiamando un'altra riceve ancora una volta il Sette. Nella fase di verifica dei punteggi ad esso si applicano le stesse regole del Sette e Mezzo Reale ma a differenza di esso la vincita attribuita è pari al triplo. Il Sette e Mezzo Triplèe inoltre da diritto al Giocatore che lo realizza di diventare Mazziere.
- Esiste una variante che assimila il Sette e Mezzo Reale con Matta al Sette e Mezzo Triplèe nella modalità di pagamento per cui a cui lo realizza sarà corrisposto il triplo della puntata.
- Esiste una variante che impone che i Giocatori preliminarmente stabiliscano il tetto massimo della puntata.
- Esiste una variante che stabilisce che il numero dei Giocatori non sia mai inferiore a 3.
- Esiste una variante che stabilisce che ogni Giocatore deve eseguire la sua puntata prima di guardare la carta coperta.
- Esiste una variante che da al Giocatore la facoltà facoltà di comportarsi in due modi differenti: può tenere coperta la carta che ha in suo possesso e chiedere una carta aggiuntiva scoperta; se poi, chiederà altre carte anche queste gli saranno date sempre scoperte; se, invece, ha inizialmente scoperto la carta in suo possesso, la carta aggiuntiva richiesta sarà sempre coperta ma, una volta che il giocatore l’avrà

guardata e deciderà di chiederne un’altra o più di una, il giocatore dovrà in ogni caso scoprire quella che gli è stata data precedentemente. Ciò in virtù della regola che solo una carta, qualunque sia il modo di gioco scelto, può restare coperta.

I riferimenti per tutte le varianti del Gioco Italiano del Sette e Mezzo sono state tratte dai seguenti siti web:

- http://www.correrenelverde.com/giochi/carte/setteemezzo.htm
- http://cavallore.interfree.it/giochi/sette_e_mezzo.html
- http://giochi-di-carte.lotoflaughs.com/giochi-di-carte-sette-e-mezzo.php

6. Glossario dei Termini

Partita: è per definizione un ciclo di mani che inizia allorquando esplicitamente si vuol creare una nuova partita.

Mano/Manche: è assimilabile ad una serie sequenziale di interazioni tra il mazziere e ciascun giocatore. Durante una mano il mazziere interagisce con un giocatore e l'interazione passa al giocatore successivo quando è terminata l'interazione col giocatore precedente. L'interazione è strettamente sequenziale e procede in unico senso: inizia col primo giocatore di Turno e procede sino al penultimo considerando che l'ultimo giocatore di Turno è sempre il Mazziere di quella specifica mano.

Turno: è il periodo durante il quale il focus del gioco di una specifica mano è concentrata tutta su un unico giocatore e infatti il mazziere interagisce solo con lui, ossia appunto con il “giocatore di turno”.

Appendice A – Struttura delle directory del progetto

Il presente progetto si consegna con la seguente struttura delle directory in modo tale da rendere omogenea la dislocazione di tutti i file inerenti il progetto stesso e conseguentemente la loro ricerca.

E' utile precisare sin d'ora che il l'applicazione è stata sviluppata realizzando due progetti che sono identici relativamente ai file sorgenti. L'unica differenza è nella specifica della classe che contiene il metodo **`main`** che deve essere lanciata. In un caso infatti la classe è **`SMGameClient`** nell'altro è **`SMGameServer`**.

La scelta progettuale è stata dettata dall'impossibilità di scindere completamente l'applicazione in due gruppi di classi tali da formare un progetto client ed uno server propriamente detto in quanto la necessità di poter eseguire anche una partita in modalità OffLine implica il fatto che talune classi che dovrebbe risiedere solo nel progetto server in realtà servirebbero anche per il progetto client.

Di conseguenza i due progetti come già detto sono identici e sono identici anche i due jar-file prodotti a meno del file di Manifest che tra le altre informazioni contiene appunto quella relativa al **`Main-Class`**.

In ogni caso la ridondanza tra i due progetti è stata opportunamente gestita ricorrendo all'uso di strumenti di Versioning Control ed infatti nel caso specifico è stato utilizzato Google Code come repository per il progetto.

La directory principale è la **`root`** directory nella quale sono presenti i seguenti file:

- **`smgame-client.jar`**: versione client dell'applicazione java.
- **`smgame-server.jar`**: versione server dell'applicazione java.
- **`smgame-client.sh`**, **`smgame-client.bat`**: file script che possono essere rispettivamente lanciati su sistema Unix-like o Windows per poter avviare la versione client dell'applicazione. Può essere comodo utilizzare questi script perché occorre passare alcuni parametri alla JVM relativi ad esempio ai file di policy che saranno definiti nel seguito.
- **`smgame-server.sh`**, **`smgame-server.bat`**: file script che possono essere rispettivamente lanciati su sistema Unix-like o Windows per poter avviare la versione server dell'applicazione. Anche in questo caso essere comodo utilizzare questi script perchè occorre passare alcuni parametri alla JVM relativi ad esempio ai file di policy che saranno definiti nel seguito.
- **`signer.sh`**, **`signer.bat`**: file script che possono essere rispettivamente lanciati su sistema Unix-like o Windows per firmare i file jar utilizzando il file **`.keystore`** precedentemente creato.
- **`smgame.html`**: file in formato HTML utile per lanciare la versione client dell'applicazione java come Applet.
- **`games.dat`**: file delle partite salvate.

- **smgame.sql**: file di testo prodotto in quale altro non è che il dump del database di MySql che può essere opportunamente importato secondo le indicazioni fornite nel precedente Capito 4 di questo documento.
- **server.log**: file di log delle attività svolte dalla versione server dell'applicazione java.

A partire dalla **root** directory questa vi sono ulteriori sottodirectory che ospitano file di tipologia e scopo differente il cui significato si indica di seguito:

- **./config**: ospita i file di configurazione necessari all'applicazione:
  - **database.properties**: file di configurazione relativo al Database MySql. Successivamente questa directory potrebbe ospitare ulteriori file di configurazione tra cui quello del server RMI, quello delle directory su cui salvare la partita ecc...
- **./doc**: ospita i documenti in formato PDF relativi a UserGuide e ReferenceGuide.
- **./javadoc**: ospita i file in formato HTML relativi alla generazione della documentazione JavaDoc del progetto;
- **./diagram**: ospita i diagrammi UML realizzati per il progetto.
- **./eclipse-project**: ospita tutti i file del progetto Eclipse relativo all'applicazione.
- **./lib**: ospita le librerie necessarie al corretto funzionamento dell'applicazione:
  - **synthetica.jar**: libreria utilizzata per il Look&Feel in modo tale da rendere omogenea la visualizzazione in ambienti differenti tra loro. Tale libreria è una Trial Version ed il sito di riferimento è http://www.javasoft.de/jsf/public/start.
  - **jpedal_lgpl.jar**: libreria utilizzata per poter visualizzare correttamente i file in formato PDF dall'interno di una applicazione Java. Tale libreria è liberamente utilizzabile sotto licenza LGPL ed il sito di riferimento è http://www.jpedal.org/.
  - **mysql-connector-java-5.1.7-bin.jar**: libreria utilizzata per potersi connettere ad un Database MySQL. Tale libreria è liberamente utilizzabile sotto licenza GPL ed il sito di riferimento è http://dev.mysql.com/downloads/connector/j/5.1.html.
- **./security**: ospite diversi file che concorrono alla gestione della security dell'applicazione java:
  - **client.policy**: file di testo formattato secondo le regole definite da Java Security attraverso cui si consente alla versione client dell'applicazione Java di avere accesso a talune risorse che nel caso specifico sono oggetti remoti via RMI o l'utilizzo di librerie esterne.
  - **server.policy**: file di testo formattato secondo le regole definite da Java Security attraverso cui si consente alla versione server

dell'applicazione Java di avere accesso a talune risorse che nel caso specifico sono file eseguibili del Sistema Operativo o l'utilizzo di librerie esterne.

- **`.keystore`**: file che contiene la coppia di chiavi pubbliche e private generate attraverso il tool java keytool e necessarie per firmare digitalmente la versione client dell'applicazione Java per l'utilizzo come Applet e le restanti librerie.

Appendice B – Firma Digitale dei jar-files

B.1 – Firma Digitale jar-files

Ai fini dell'esecuzione dell'applicazione come applet, è stata eseguita la procedura di firma dei jar-files nel seguente modo:

1) Creazione di una coppia di chiavi (pubblica e privata). Con la chiave privata saranno firmati i file scaricati all'Applet. Mentre la chiave pubblica serve per poter verificare la firma eseguita con la chiave privata. Per creare una coppia di chiavi si esegue il seguente comando:

```
keytool -genkey -alias smgamekey -keypass smgamekeypassword
        -keystore ./security/smgamekeystore -storepass smgamestorepassword
```

   con il quale crea si crea un alias (**smgamekey**) e si inserisce una password (**smgamekeypassword**) per usare la propria chiave privata. Il comando **keytool** memorizzerà le chiavi e gli eventuali certificati in un oggetto chiamato KeyStore (implementazione della Sun della classe astratta **java.security.KeyStore**), memorizzandolo in un file (**smgamekeystore**) e proteggendolo tale KeyStore con una password (**smgamestorepassword**). Dopo l'inserimento di alcune informazioni personali (nome e cognome, società, città, etc..) verranno create le chiavi da usare per firmare i propri codici e farli autenticare. Il KeyStore verrà salvato come nella directory (**./security/**).

2) Firma dell'Applet (contenuta nel jar-file **smgame-client.jar**) e della libreria **synthetica.jar** con la linea di comando:

```
jarsigner -keystore ./security/smgamekeystore
          -storepass smgamestorepassword -keypass smgamekeypassword
          ./smgame-client.jar smgamekey
```

   Questa istruzione genera il file **smgame-client.jar** derivante dall'aggiunta dei file **SMGAMEKEY.SF** e **SMGAMEKEY.DSA** al precedente **smgame-client.jar**. Si noti l'uso delle password (**-storepass smgamestorepassword -keypass smgamekeypassword**) per accedere al KeyStore e alla chiave privata memorizzata all'interno del KeyStore.

Sono stati inoltre predisposti gli script **jarsigner.sh** e **jarsigner.bat** rispettivamente per ambiente Unix-like e Windows per poter firmare i jar-file dell'Applet in sostituzione del comando al precedente punto 2.

B.2 – Configurazione lato Client e lato Server

Per poter lanciare correttamente l'applicazione lato client e lato server, come accennato in Appendice A sono stati messi a disposizione due script rispettivamente per ambiente Linux e Windows.

Per quel che riguarda il file **`smgame-server.sh`** e **`smgame-server.bat`** occorre precisare che all'interno di tali file script, è indicata la codebase RMI da cui la JVM lato server ricercherà la classe Stub. Tale parametro relativo alla codebase deve essere valorizzato con una URL al filesystem locale che deve essere un path assoluto. Di conseguenza è necessario cambiare il parametro codebase in modo opportuno affinché punti al file **`smgame-server.jar`**.

Se non si vuole utilizzare i file di script predisposti allora è anche possibile lanciare le applicazioni lato client come segue:

```
java -Djava.security.policy=./security/client.policy -jar ./smgame-client.jar
```

e lato server come segue:

```
java -Djava.rmi.server.codebase=<PATH>/smgame-server.jar
     -Djava.security.policy=./security/server.policy -jar ./smgame-server.jar
```

dove <PATH> ancora una volta deve essere valorizzato al path assoluto sul filesystem locale dove risiede il file **`smgame-server.jar`** .