Relazione progetto

BANKSYSTEM

CORSO DI LAUREA IN INFORMATICA

Programmazione 3 – A.A. 2022/2023
Prof. Ciaramella, Prof. Di Nardo

Componenti del gruppo:
Simone Palladino - 0124002316
Luca Tartaglia - 0124002294
Mattia Di Palma – 0124002448

INTRODUZIONE E ANALISI DEI REQUISITI

La traccia a noi assegnata riguarda l'implementazione di un sistema bancario per la gestione di conti correnti con le relative operazioni.
Il sistema bancario prevede un accesso in due modalità:

- Amministratore (admin)
- Correntista (Holder).

L'amministratore può effettuare le seguenti operazioni:

- Inserire un nuovo correntista (ricerca per nome e cognome o per numero di conto)
- Eliminare un correntista (ricerca per nome e cognome o per numero di conto)

Il correntista (Holder) può effettuare le seguenti operazioni:

- Visualizzare i movimenti di tutte le operazioni che ha fatto (sia per carta sia per il totale delle carte)
- Fare acquisti tramite una carta a esso associata (bancomat oppure carta di credito)
- Richiedere di annullare l'ultima operazione effettuata (ad esempio l'acquisto di un prodotto) e il costo verrà integrato nuovamente al saldo totale
- Possiede dei servizi in base alla tipologia di conto (Basic, Premium o Enterprise)
- Possiede la capacità di prelevare o depositare sul conto e scegliere inoltre la carta su cui depositare.

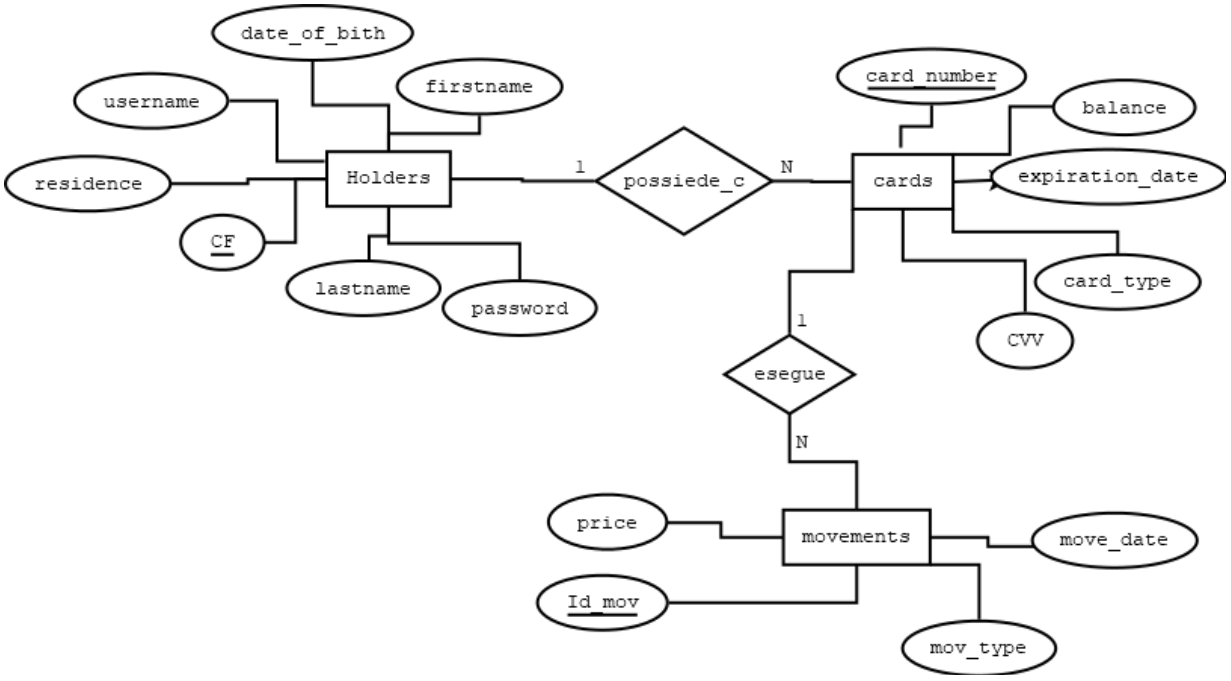
Regole generali in **BankSystem:**

- Una persona con contratto Basic può depositare massimo 5.000€ al giorno.
- Una persona con contratto Premium può depositare massimo 50.000€ al giorno.
- Una persona con contratto Enterprise può depositare 1.000.000€ al giorno.
- Una carta Bancomat non può andare sotto lo zero.
- Una persona con contratto Basic e carta di credito può avere un saldo negativo di massimo -10€.
- Una persona con contratto Premium e carta di credito può avere un saldo negativo di massimo -100€.
- Una persona con contratto Enterprise e carta di credito può avere un saldo negativo di massimo -1000€.
- Una persona con contratto Basic può spendere massimo 2000€ per carta al giorno ed effettuare massimo tre operazioni
- Una persona con contratto Premium può spenderne massimo 10.000€ per carta al giorno ed effettuare massimo dieci operazioni
- Una persona con contratto Enterprise può spendere 100.000€ per carta al giorno con un numero indefinito di operazioni

Il progetto è stato sviluppato come una applicazione web tramite l'utilizzo di apache tomcat 10.1.5 e le java servlet, inoltre per una conservazione coerente dei dati ci siamo serviti di un database esterno che in seguito verrà analizzato nel dettaglio insieme a tutti i metodi, design pattern associati e le eccezioni create per il rispetto dei criteri sopracitati.

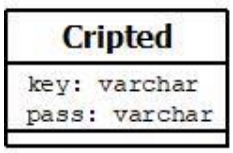
IMPLEMENTAZIONE DEL DATABASE

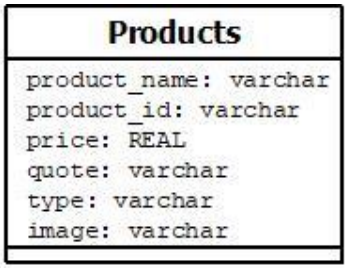
Per quanto riguarda l'implementazione del database abbiamo scelto di utilizzare SQLite per la conservazione dei dati.
Di seguito il diagramma EER.

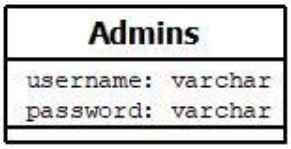


Utilizziamo inoltre tre tabelle "esterne" CRIPTED, PRODUCTS e ADMINS

- CRIPTED : per salvare le coppie chiave valore in riferimento alle password criptate
- PRODUCTS: per salvare le tipologie di prodotto che è possibile acquistare direttamente dal nostro sistema bancario
- ADMINS: per salvare gli amministratori del sistema







Le implementazioni di queste operazioni verranno successivamente analizzate nel dettaglio.
La connessione al database viene realizzata mediante lo sfruttamento della tecnologia **JDBC** (Java database connectivity) tramite un'astrazione software che permette la connessione alle applicazioni java.
Per una corretta gestione di tutte le operazioni sul database abbiamo utilizzato il **Pattern DAO**.

NOTA: Inserire il database (banksystem.sqlite) nella cartella di apache-tomcat-10.1.5/bin e richiamare il percorso nella configurazione di tomcat su Intellij Idea.

IMPLEMENTAZIONE DEI PATTERN

I **pattern** sono dei modelli di risoluzioni a problemi già noti, vengono utilizzati per avere una solida soluzione progettuale e possiamo classificarli in tre macrocategorie:

- CREAZIONALI (Singleton e Factory)
- STRUTTURALI (MVC, DAO)
- COMPORTAMENTALI (Observer)

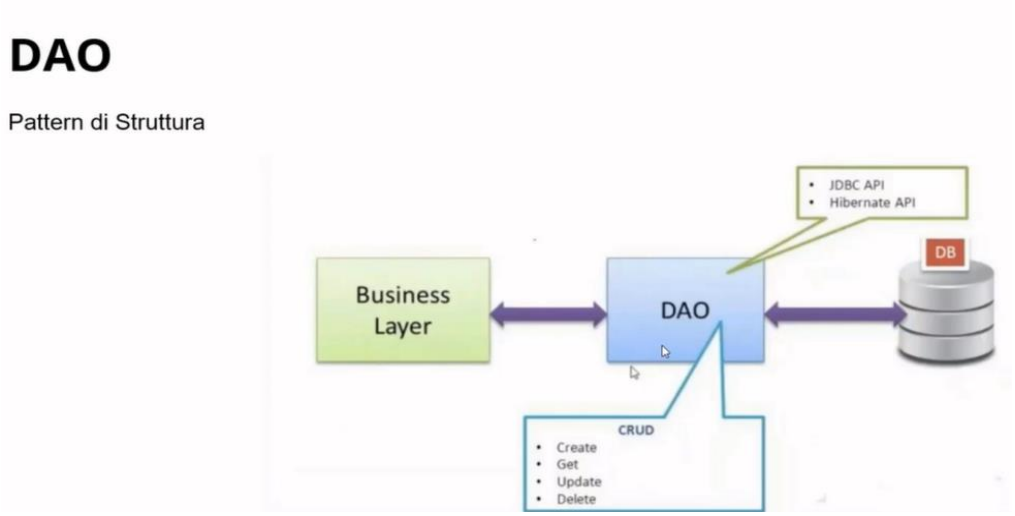
I **DESIGN PATTERN** da noi utilizzati sono i seguenti:

- **Pattern DAO (Data Access Object)**
- **Pattern Observer**
- **Pattern MVC (Model View Controller)**
- **Pattern Factory**
- **Pattern Iterator**

DAO

Il Pattern Dao viene utilizzato per separare la logica di business da quella di persistenza tramite l'utilizzo di un'interfaccia di collegamento al database.
Sfruttando questa implementazione il software ha vari vantaggi, come ad esempio l'incapsulamento dei dati in oggetti dedicati e un alto grado di manutenibilità.

STRUTTURA DEL PATTERN DAO:



IMPLEMENTAZIONE :

Nel nostro progetto abbiamo implementato questo pattern attraverso l'utilizzo di tre strutture:

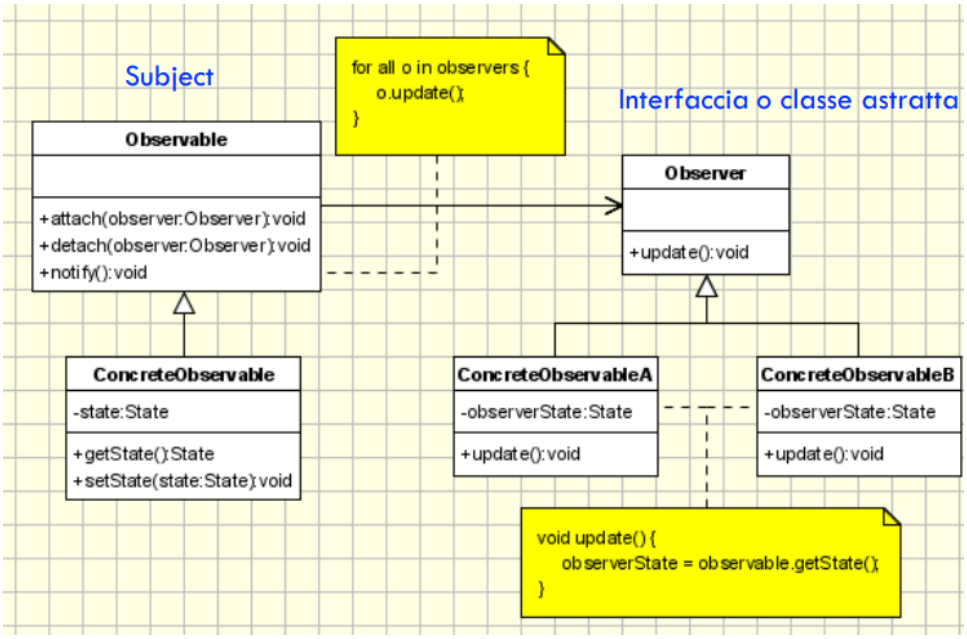
- **Classe Model** (Es. Holders)
- **Classe ModelOperation** (Es. HolderOperation)
- **Interfaccia Operation**

La classe Model, ad esempio Holders, rappresenta il modello dei dati.
Nell'interfaccia Operation vengono dichiarati i metodi **CRUD** per le operazioni sul database.
Nella classe HolderOperation viene effettuato l'override dei metodi dichiarati nella nostra interfaccia di collegamento al database e viene istanziata una lista di oggetti specifica all'implementazione.

OBSERVER

Il pattern Observer è molto utile per la gestione degli eventi e in generale dei cambi di stato da voler notificare. Definisce infatti una dipendenza tra oggetti, tale che se un oggetto cambia stato, tutte le sue dipendenze sono notificate e aggiornate automaticamente.

STRUTTURA DEL PATTERN OBSERVER:



IMPLEMENTAZIONE:

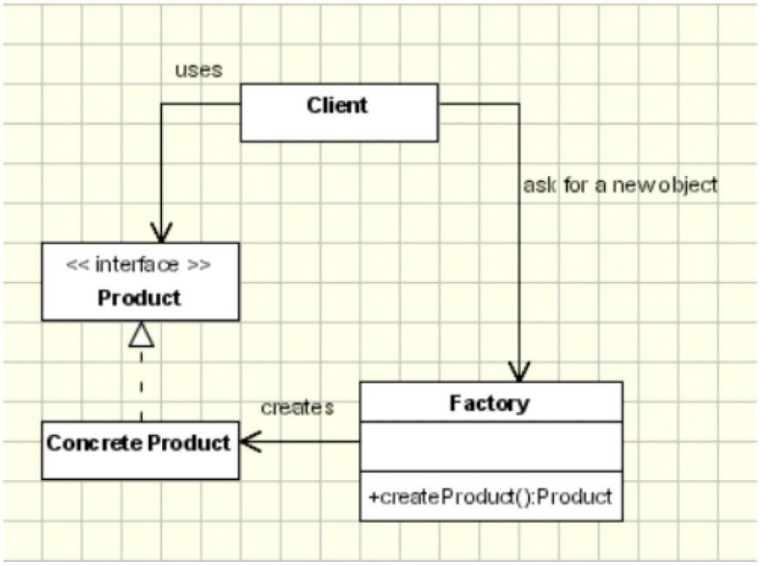
Nell'implementazione del pattern Observer riconosciamo due "attori" principali, ovvero l'osservato e gli osservatori, questi ultimi vengono notificati degli eventuali cambi di stato dell'osservato. Nel nostro caso utilizziamo la classe Holders e la classe Cards che implementano la classe Observer.
Quando ad esempio generiamo un movimento (deposito, prelievo, acquisto o rimborso) tramite l'utilizzo della classe MovementObserver notifichiamo attraverso il metodo notifyObservers il cambiamento di stato alla carta associata, inoltre Cards che implementa Observer farà l'override del metodo Update per "ricevere" la notifica del movimento generato.

In maniera analoga, la stessa operazione viene effettuata quando ad un correntista viene assegnata una nuova carta: sarà aggiornata la lista di carte che il singolo possiede per facilitare le operazioni di inserimento/cancellazione.

FACTORY

Lo scopo del Factory Pattern è creare oggetti senza esporre la logica di istanziazione al client, definendo oggetti attraverso un'interfaccia comune.

STRUTTURA DEL PATTERN FACTORY:



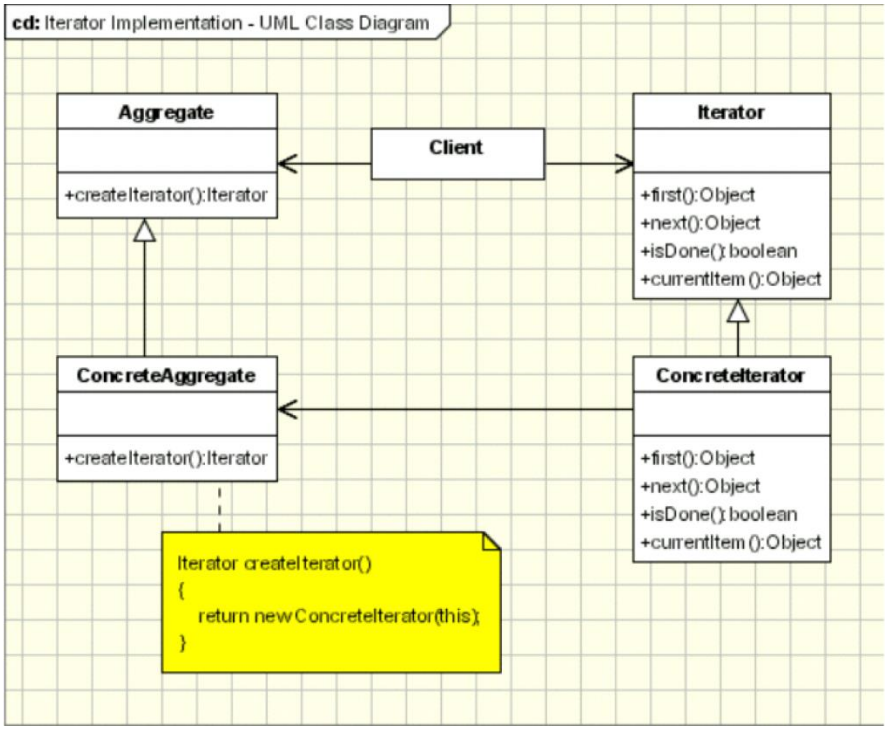
IMPLEMENTAZIONE:

Nel nostro progetto c'è una classe chiamata Factory che serve a richiamare le altre classi, la classe possiede metodi statici in modo tale da non essere istanziata. Noi le abbiamo utilizzate per istanziare iterator e operazioni in generale. In particolar modo richiama l'interfaccia Operation (che implementa il pattern DAO) e in base al tipo di OperationType passato come parametro restituisce l'operazione richiesta tramite uno switch.

ITERATOR

Il pattern Iterator è un pattern di progettazione di software che consente di accedere agli elementi di una collezione in modo sequenziale senza conoscere la sua rappresentazione interna. In pratica, esso fornisce un modo per accedere agli elementi di una collezione in modo semplice e generico, indipendentemente dal tipo di collezione.

STRUTTURA DEL PATTERN ITERATOR:



IMPLEMENTAZIONE:

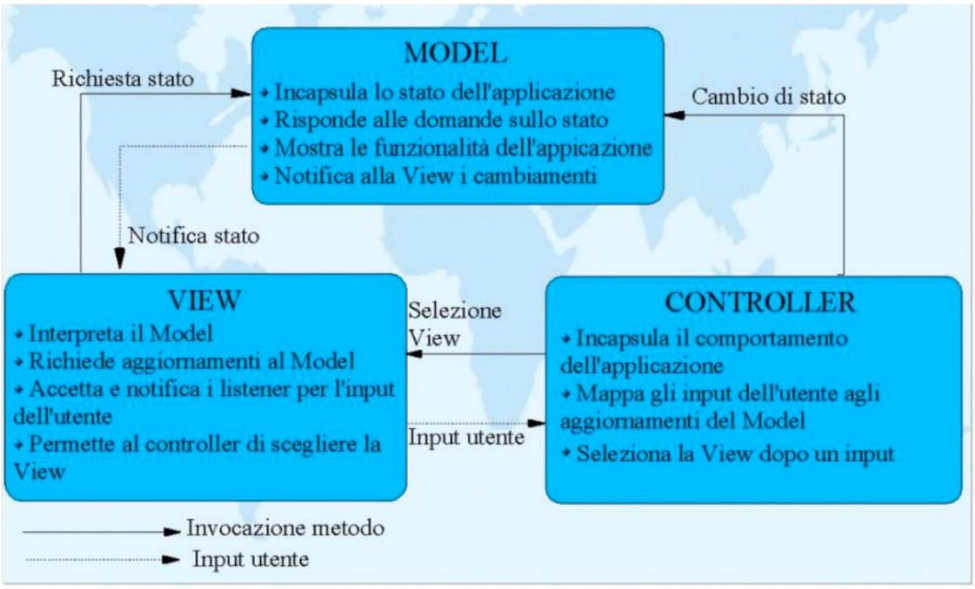
Nel nostro caso abbiamo utilizzato tre iterator: HolderIterator, MovementIterator e CardIterator. Tutti e tre implementano l'interfaccia Iterator e forniscono metodi per verificare se ci sono ancora elementi nella collezione (hasNext()); quest'ultimo è necessario ad ottenere il prossimo elemento (next()). La classe HolderIterator itera su una lista di Holder, la classe MovementIterator su una lista di Movement e la classe CardIterator su una lista di Card.
Inoltre, la classe MovementIterator ha un'opzione per iterare la lista in ordine inverso, utilizzando la proprietà 'reverse'; in questo modo i movimenti sulla schermata principale verranno visualizzati correttamente (dal più recente).

MVC (MODEL VIEW CONTROLLER)

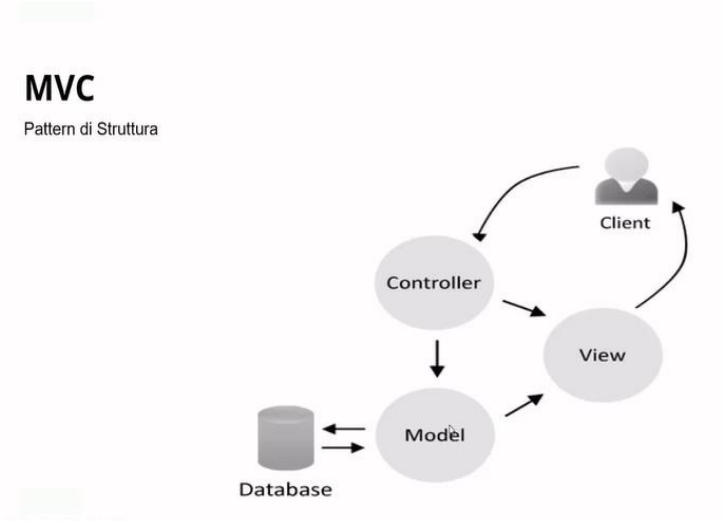
Il pattern MVC (Model-View-Controller) è un pattern di progettazione architetturale utilizzato per la creazione di interfacce utente. Il pattern separa l'applicazione in tre componenti principali:

- Model: rappresenta i dati e la logica di business dell'applicazione. Si occupa di gestire la manipolazione dei dati e di notificare i cambiamenti ai componenti interessati.
- View: rappresenta l'interfaccia utente dell'applicazione. Si occupa di visualizzare i dati forniti dal Model e di generare eventi in risposta alle azioni dell'utente.
- Controller: si occupa di gestire gli eventi generati dalla View e di modificare il Model o la View in base a essi.

STRUTTURA DEL PATTERN MVC:



IMPLEMENTAZIONE:



Il pattern MVC viene implementato nella nostra applicazione attraverso tre strutture fondamentali.

- **CLASSI MODEL/OPERATION**
- **FILE JSP (VIEW)**
- **CONTROLLER (SERVLET)**

Le classi **Model** insieme alle classi **Operation** (Pattern DAO) gestiscono l'interfacciamento al database per una corretta conservazione dei dati.
I file **JSP/HTML** invece si occupano delle **View** lato utente, con la gestione dei dati immessi a seconda delle operazioni scelte da quest'ultimo.
Infine, i **Controller** rappresentati dalle **Servlet** ricevono comandi dall'utente attraverso le view con il compito di eseguirli.
Le **Servlet (Controller)** inoltre interagiscono con le classi incaricate di interfacciarsi con il database per salvare i dati e/o operazioni effettuate dall'utente.

Dettagli aggiuntivi ed esempi di codice verranno forniti nel capitolo successivo.

ANALISI E IMPLEMENTAZIONE DELLE SERVLET

Come già anticipato nella sezione riguardante i pattern e in particolare nella sezione implementazione del **pattern MVC**, le **Servlet** rappresentano il **Controller** della nostra applicazione web.
Le funzionalità principali riguardano l'elaborazione o memorizzazione di dati provenienti da form HTML/JSP utilizzando una comunicazione Client-Server con protocollo HTTP, si occupano inoltre della generazione di contenuti dinamici a seconda dei parametri della richiesta effettuata dall'utente.
Di seguito alcuni esempi di codice per il login (user e admin) e l'aggiunta di un nuovo utente.

SERVLET ADMIN-ADD-ACCOUNT

```
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        response.setContentType("text/html");

        //parametri presi in input dal form di inserimento nella sezione admin
        //la prima password dell'utente viene
        String cf = request.getParameter("cf");
        String firstname = request.getParameter("firstname");
        String lastname = request.getParameter("lastname");
        String account_type = request.getParameter("accounttype");
        String residence = request.getParameter("address");
        String username =  request.getParameter("username");
        String card_type = request.getParameter("cardtype");

        String password = cf;
        byte[] salt = new String("12345678").getBytes();
        int iterationCount = 40000;
        int keyLength = 128;
        SecretKeySpec key ;
        //creazione della chiave associta alla password da criptare
        try {
            key = createSecretKey(password.toCharArray(), salt, iterationCount,
keyLength);
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        } catch (InvalidKeySpecException e) {
            throw new RuntimeException(e);
        }

        String originalPassword = password;

        String encryptedPassword = null;
        try {
            encryptedPassword = encrypt(originalPassword, key);
        } catch (GeneralSecurityException e) {
            throw new RuntimeException(e);
        }

        Holder h = new Holder(username, firstname, lastname, cf, null, account_type,
residence, 0, encryptedPassword);
```

```
        Actions.getInstance().holderOperation.add(h);
        k.put(key,encryptedPassword);

        addP(key, encryptedPassword); //funzione per l'aggiunta della coppia chiave
valore alla tabella cripted

        //generazione della prima carta da associare alla creazione
        String[] randomCard = generateCard();
        LocalDate carddeadline = LocalDate.now().plusYears(10);
        CardObserver.getInstance().add(new Card(card_type + " of " + firstname,
randomCard[0], cf, Integer.valueOf(randomCard[1]),card_type, carddeadline, 0));

        switch (account_type) {
            case "Premium" :
                MovementObserver.getInstance().add(new Movement("welcomepremium",
LocalDate.now(), randomCard[0]));
                break;
            case "Enterprise":
                MovementObserver.getInstance().add(new Movement("welcomeenterprise",
LocalDate.now(), randomCard[0]));
                break;
        }

        request.getRequestDispatcher("admin-dashboard.jsp").forward(request, response);
    }

}
```

SERVLET LOGIN

```
public void doGet(HttpServletRequest request, HttpServletResponse response) throws
IOException, ServletException {
    response.setContentType("text/html");
    HttpSession session = request.getSession();

    logintype = request.getParameter("logintype");
    error = request.getParameter("error");

    request.getRequestDispatcher("login.jsp").forward(request, response);
}

/** Metodo all'interno del quale viene controllato come è stato settato il logintype,
se quest'ultimo
 * è uguale a holder viene caricato il form per accedere alla sua area personale.
 * Si effettuano successivamente i relativi controlli sulla correttezza dei dati
inseriti
 * utilizzando la tabella "cripted" e l'attributo password nella tabella "Holders".
 * Se il logintype risulta uguale ad admin si effettuano le stesse operazioni sulla
tabella "admins"
 *
 * @param request  an {@link HttpServletRequest} oggetto che contiene la richiesta che
il client ha fatto alla servlet
 * @param response an {@link HttpServletResponse} oggetto che contiene la risposta che
la servlet invia al client
 * @throws ServletException
 * @throws IOException
 */
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
```

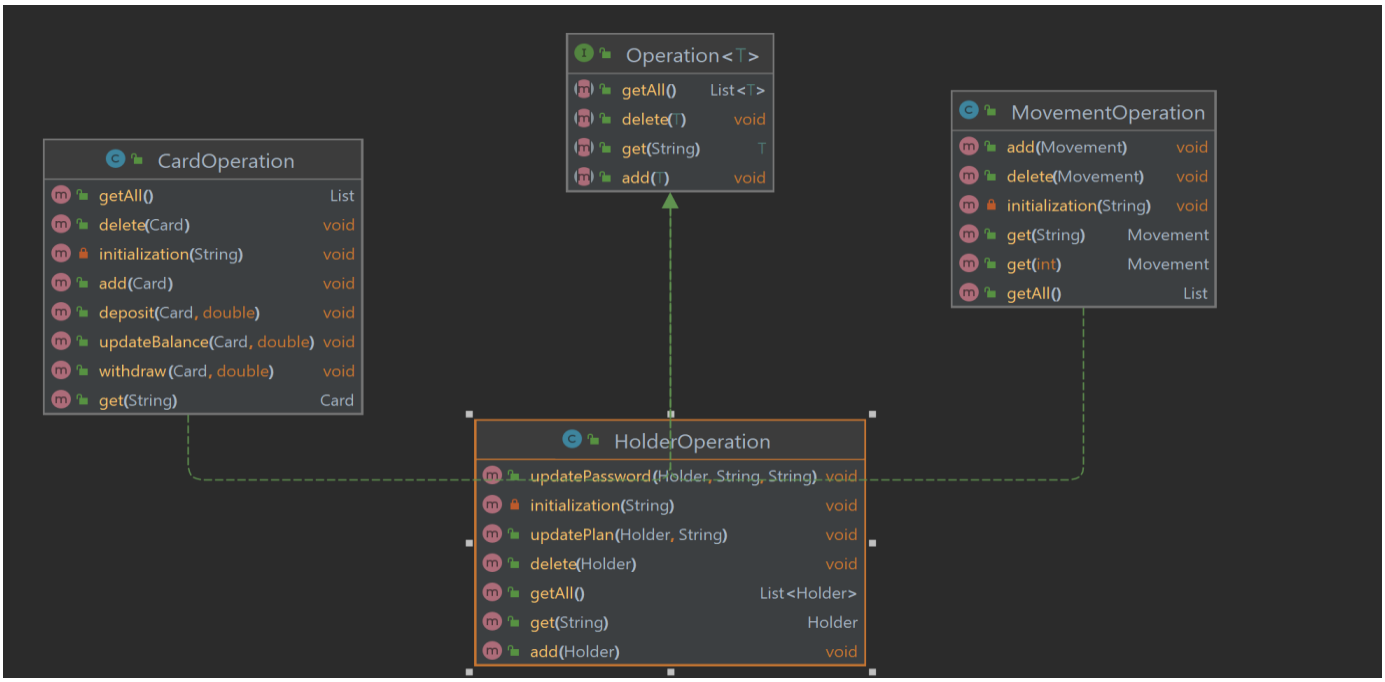
```
    response.setContentType("text/html");
    Boolean done = false;

    //Imposta l'username immesso come attributo di sessione per visualizzarlo nella
barra in alto
    HttpSession session = request.getSession();
    session.setAttribute("usertext", request.getParameter("username").toString());
    String username = null, password = null, cf = null;
    String depassword =null;


    //Codice relativo al login del correntista
    //Controlla se l'username e la password sono presenti nella tabella "Holders"
    if (logintype.equals("holder")) {
        try {
            con = DriverManager.getConnection("jdbc:sqlite:banksystem.sqlite");
            Statement stmt = con.createStatement();
            ResultSet rs;

            String key = null;
            SecretKeySpec keytemp ;
            getK(); //caricamento della struttura dati hashmap(chiave,valore)
            username = request.getParameter("username");
            password = request.getParameter("password");
            rs = stmt.executeQuery("SELECT * FROM Holders WHERE username='" + username
+"'");
            String decryptedPassword = null;
            //controllo della correttezza della password
            if (rs.next()) {
                depassword = rs.getString("password");
                cf = rs.getString("cf");
                for (SecretKeySpec keyy : k.keySet()) {
                    if (k.get(keyy).equals(depassword)){
                        keytemp = keyy;
                        decryptedPassword =  decrypt(depassword,keytemp);
                        System.out.println(decryptedPassword);
                        if (decryptedPassword.equals(password))
                            done = true;
                    }
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } catch (GeneralSecurityException e) {
            throw new RuntimeException(e);
        } finally {
            try {
                if (con != null)
                    con.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }

        //se il controllo avviene con successo l'utente viene reindirizzto sulla
servlet "holder-set-pass"
        if (done) {
            session.setAttribute("selectedHolder",
Actions.getInstance().holderOperation.get(cf));

            if (password.equals(cf))
                response.sendRedirect("holder-setpassword?cf=" + cf +"&depassword="+
```

```
depassword);
            else
                response.sendRedirect("dashboard?logintype=holder");
        } else {
            response.sendRedirect("login?logintype=holder&error=errore");
        }
    } else {
        //Codice relativo al login admin
        //Controlla se l'username e la password sono presenti nella tabella "Admins"
        try {
            con = DriverManager.getConnection("jdbc:sqlite:banksystem.sqlite");
            Statement stmt = con.createStatement();
            ResultSet rs;

            username = request.getParameter("username");
            password = request.getParameter("password");
            rs = stmt.executeQuery("SELECT * FROM Admins WHERE username='" + username +
"' AND password='" + password + "'");

            if (rs.next()) {
                done = true;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                if (con != null)
                    con.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }

        if (done) {
            response.sendRedirect("dashboard?logintype=admin");
        } else {
            response.sendRedirect("login?logintype=admin&error=errore");
        }
    }
```

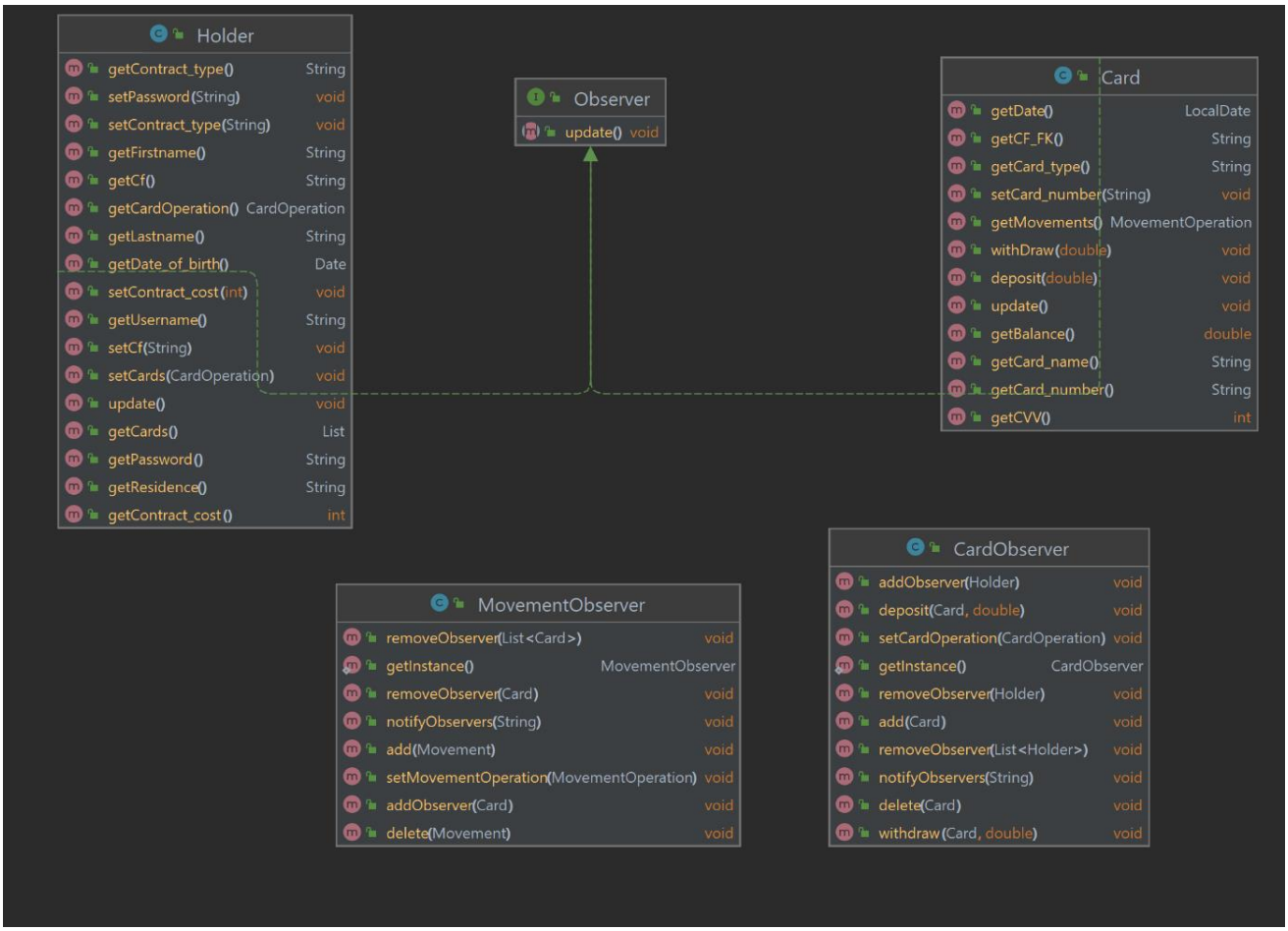
Gli esempi di codice riportarti in questo capitolo sono solo alcune delle operazioni implementate nel progetto, ulteriori spiegazioni verranno poi oralmente fornite dagli sviluppatori che compongono il gruppo.

UML DELLE CLASSI E DEI PATTERN

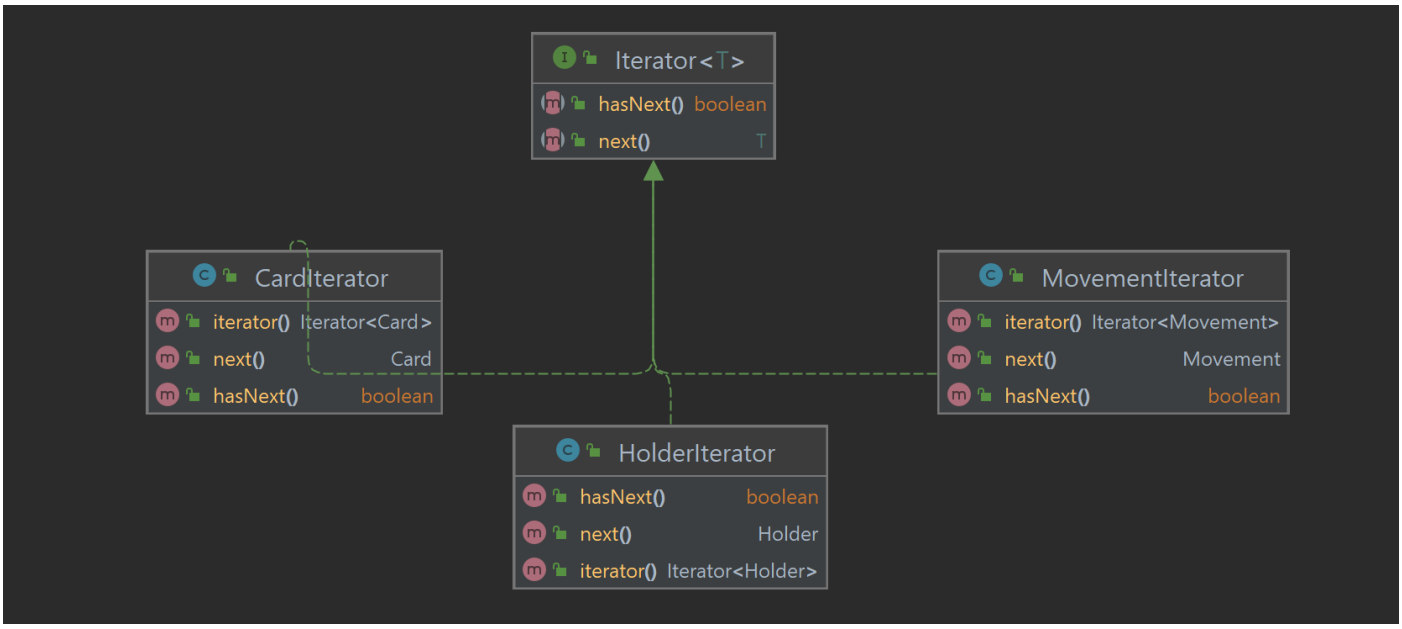
UML DEL PATTERN DAO:



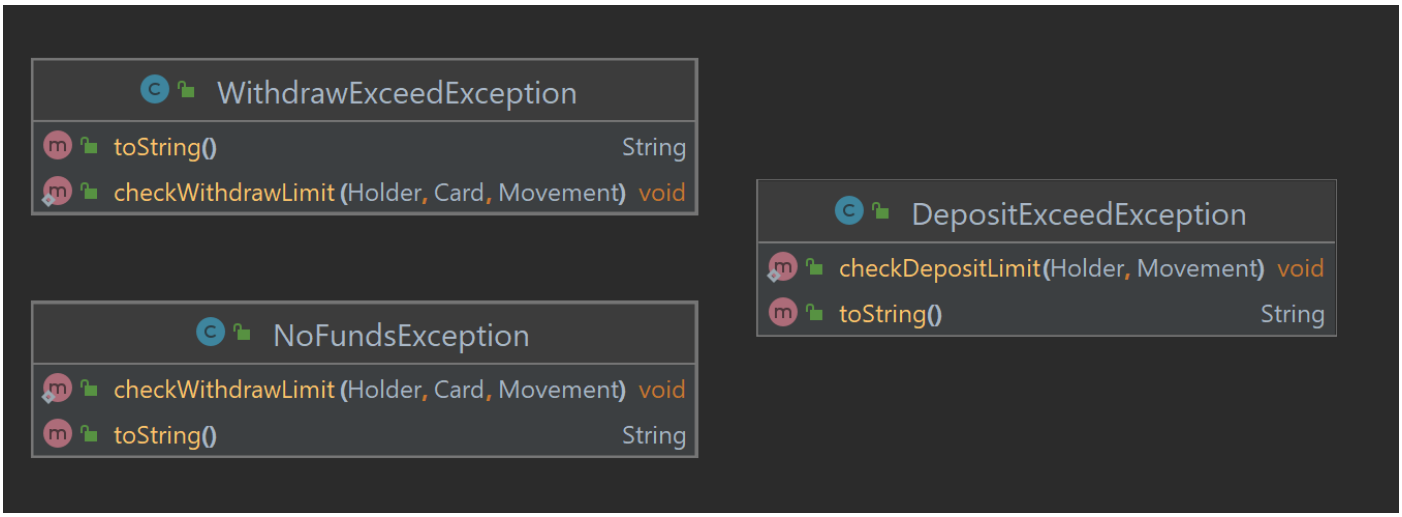
UML DEL PATTERN OBSERVER:



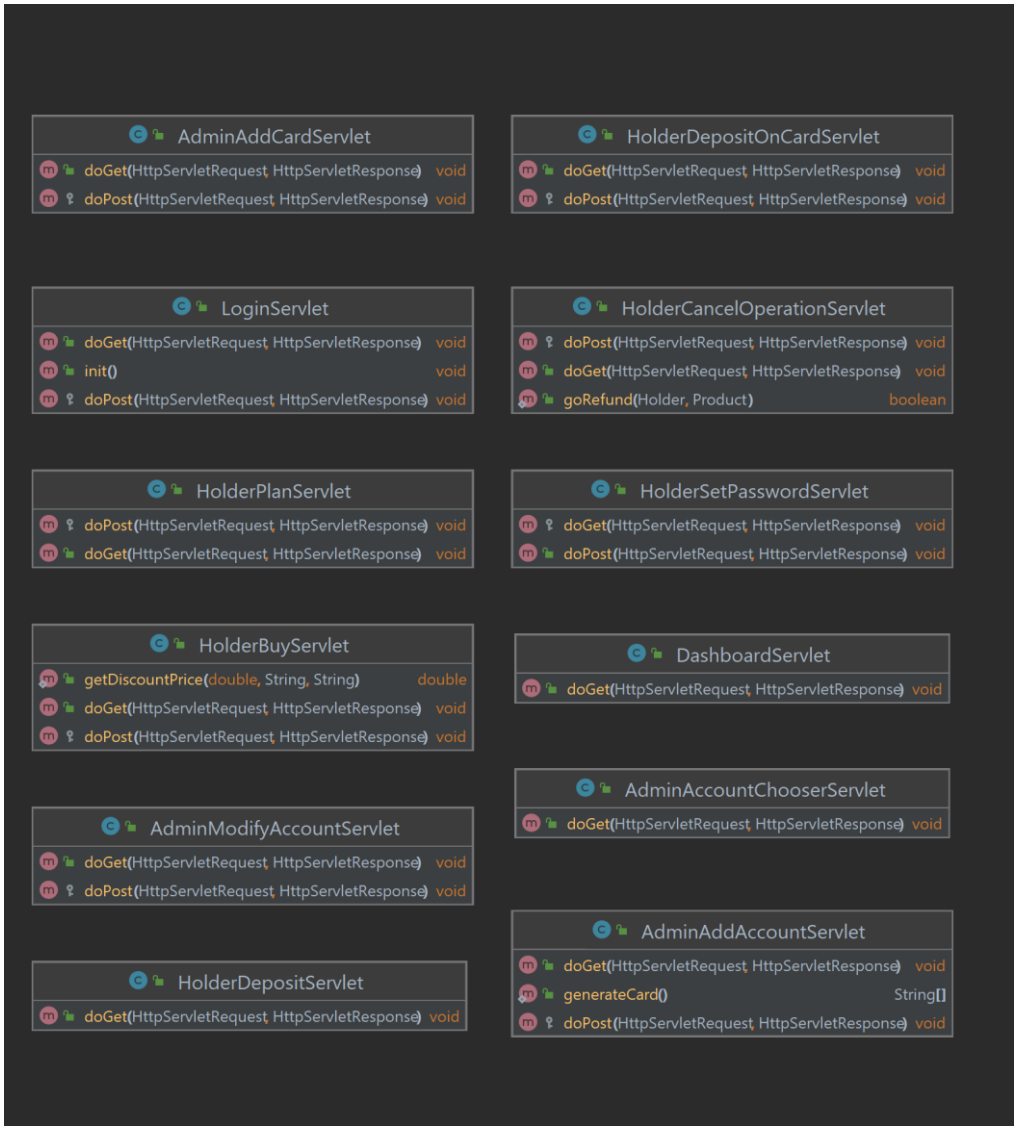
UML PATTERN ITERATOR:



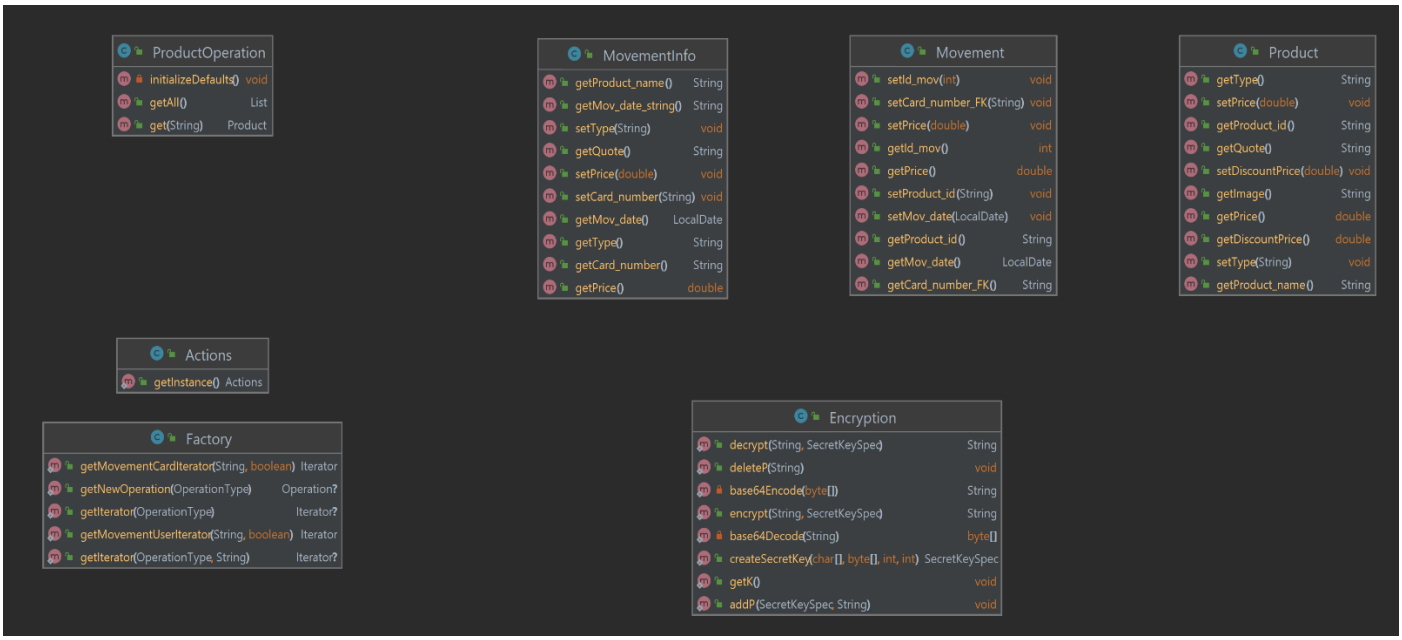
UML ECCEZIONI:



UML DELLE SERVLET:



UML DI ULTERIORI CLASSI:



GESTIONE DELLE ECCEZIONI

Un'eccezione in Java è un evento anomalo che si verifica durante l'esecuzione di un programma. Le eccezioni sono utilizzate per gestire gli errori e le situazioni impreviste, nel nostro caso abbiamo utilizzato tre eccezioni:

- **DepositExceedException:** questa eccezione permette di impostare delle limitazioni sui depositi.
- **NoFundsException:** questa eccezione consente di impostare delle limitazioni in base ai tipi di contratti.
- **WithdrawExceedException:** questa eccezione consente d'impostare delle limitazioni sui prelievi.

Vediamole più in dettaglio:

DepositExceedException:

```
        switch (holder.getContract_type()) {
            case "Basic":
                limitDeposit = 5000;
                break;
            case "Premium":
                limitDeposit = 50000;
                break;
            case "Enterprise":
                limitDeposit = 1000000;
                break;
        }

        if (limitDeposit > 0 && operationCount > limitDeposit)
            throw new DepositExceedException();
```

Come possiamo notare in questa eccezione si stabilisce che il correntista ha la possibilità di depositare una quantità di denaro limitata che si differenzia dalla sua tipologia di contratto. Se possiede un contratto Basic allora il suo limite di deposito sarà di 5000 euro al giorno, se possiede un contratto Premium il suo limite di deposito sarà di 50000 euro al giorno e infine se possiede un contratto Enterprise il suo limite di deposito sarà di 1000000 euro al giorno.

NoFundsException:

```
public static void checkWithdrawLimit(Holder holder, Card card, Movement movement)
throws NoFundsException {
        if (card.getBalance() + movement.getPrice() < 0) {
            if (card.getCard_type().equals("Bancomat"))
                throw new NoFundsException();
            else if (card.getCard_type().equals("Credit Card")) {
                switch (holder.getContract_type()) {
                    case "Basic":
                        if (card.getBalance() + movement.getPrice() < 10)
                            throw new NoFundsException();
                    case "Premium":
```

```
                        if (card.getBalance() + movement.getPrice() < 100)
                            throw new NoFundsException();
                    case "Enterprise":
                        if (card.getBalance() + movement.getPrice() < 1000)
                            throw new NoFundsException();
                }
            }
        }
    }
}
```

Come possiamo notare in questa eccezione si stabilisce che il correntista può effettuare il prelievo oppure l'acquisto secondo alcune semplici regole. A seconda della tipologia di pagamento specificata non è possibile scendere al di sotto una determinata soglia di saldo. Se il tipo della carta è Bancomat non può andare sotto lo zero mentre se il tipo della carta è Carta di credito allora a seconda della tipologia di contratto che possiede il correntista avrà la possibilità scendere al di sotto di una determinata soglia, ad esempio se il tipo di contratto del correntista è Basic allora è possibile avere un debito di 10 euro, se il tipo di contratto del correntista è Premium allora è possibile avere un debito di 100 euro e infine se il tipo di contratto del correntista è Enterprise allora è possibile avere un debito di 1000 euro.

WithdrawExceedException:

```
if(movementTemp.getMov_date().toString().equals(movement.getMov_date().toString()) &&
!productTemp.getType().equals("deposit") && !productTemp.getType().equals("upgrade")) {
                operationCount += -movementTemp.getPrice();
                count++;
            }
        }

        switch (holder.getContract_type()) {
            case "Basic":
                limitWithdraw = 2000;
                countLimit = 3;
                break;
            case "Premium":
                limitWithdraw = 10000;
                countLimit = 10;
                break;
            case "Enterprise":
                limitWithdraw = 100000;
                countLimit = -1;
                break;
        }
```

Come possiamo notare in questa eccezione si stabilisce che il correntista può effettuare il prelievo o l'acquisto secondo alcune regole. Se il correntista possiede un contratto Basic allora gli sarà possibile spendere massimo 2000 euro per carta al giorno ed effettuare massimo tre operazioni giornaliere. Se il correntista possiede un contratto Premium il limite di spesa è fissato a 10000 euro al giorno per carta e il massimo numero di operazioni

giornaliere a 10. Infine, se il correntista possiede un contratto Enterprise il limite di spesa è fissato a 100000 euro al giorno, per quanto riguardo invece il numero delle operazioni sono semplicemente illimitate.

Le eccezioni vengono richiamate in questo modo:

```
try {
    //Controlla se l'utente può permettersi di effettuare il prelievo o l'acquisto
    NoFundsException.checkWithdrawLimit(selectedHolder, selectedCard, movement);
    WithdrawExceedException.checkWithdrawLimit(selectedHolder, selectedCard, movement);

    MovementObserver.getInstance().add(movement);
} catch (NoFundsException noFundsException) {
    //Gestione dell'eccezione nel caso non ci sono fondi a sufficienza
    System.out.println(noFundsException.toString());
    response.sendRedirect("errorpage.jsp?error=nofund&backurl=holder-deposit");
    return;
} catch (WithdrawExceedException withdrawExceedException) {
    //Gestione dell'eccezione nel caso l'importo superi i limiti previsti dal piano
    System.out.println(withdrawExceedException.toString());
    response.sendRedirect("errorpage.jsp?error=nowithdraw&backurl=holder-deposit");
    return;
} catch (Exception e) {
    //Gestione dell'eccezione nel caso si sia verificato un errore generico
    response.sendRedirect("errorpage.jsp?backurl=holder-deposit");
    return;
}
```

PRINCIPI DELLA PROGRAMMAZIONE SOLID

I principi SOLID sono un insieme di principi di progettazione di software che mirano a rendere il codice più leggibile, manutenibile e scalabile.
I 5 principi SOLID sono:

1. **Single responsibility principle** (**SRP**): ogni classe dovrebbe avere una sola responsabilità e il suo codice dovrebbe essere focalizzato sull'adempimento di tale responsabilità. Questo principio viene rispettato nel nostro progetto come precedentemente analizzato nell'analisi dei patter.
2. **Open-closed principle** (**OCP**): le classi dovrebbero essere aperte per l'estensione, ma chiuse per la modifica. Ciò significa che le classi dovrebbero essere progettate in modo da poter essere estese per supportare nuove funzionalità, senza dover modificare il codice esistente. All'interno del nostro progetto non sono presenti classi astratte, ma interfacce e classi concrete che le implementano: seguendo l'implementazione originale del Pattern DAO non abbiamo ritenuto necessario complicare ulteriormente la strategia di implementazione.
3. **Liskov substitution principle** (**LSP**): gli oggetti di una sottoclasse dovrebbero essere utilizzabili al posto degli oggetti della superclasse senza causare problemi di funzionamento del programma. Nella struttura del nostro progetto non sono presenti tali problematiche, poiché non è stato necessario ricorrere all'uso di sottoclassi estese.
4. **Interface segregation principle** (**ISP**): le interfacce dovrebbero essere piccole e specializzate in modo che gli oggetti che le implementino non siano costretti a implementare metodi che non utilizzano. Nel nostro caso viene rispettato questo principio in quanto ogni implementazione è specifica e non eccessiva; nel nostro caso vengono utilizzate interfacce per ricorrere all'implementazione dei pattern Iterator, Observer e DAO.
5. **Dependency inversion principle** (**DIP**): i moduli di alto livello (ad esempio, le classi) non dovrebbero dipendere dai moduli di basso livello (ad esempio, le classi di implementazione), ma entrambi dovrebbero dipendere dalle interfacce. Ciò rende il codice più flessibile e facile da modificare. Nel nostro caso viene rispettato questo principio in quanto le nostre classi comunicano soltanto con le classi direttamente collegate ad esse (Law of Demeter): i moduli di alto livello e i moduli di basso livello dipendono entrambi da interfacce appositamente create per esse. Le logiche di implementazione (Factory, DAO, Iterator) sono utilizzate per accedere ai dettagli implementativi.