Testing

Test Plan:

Oltre a validare con l'esecuzione del codice sorgente, testando le varie funzionalità ai fini di verificarne la validità, e che le funzionalità così testate siano quelle che il committente ci aveva richiesto, abbiamo usato JUnit 4 per testare due metodi della classe : '*GestoreQueryCerca'.*

Metodi:

- `Impiegato eseguiQueryRicercaImpiegatoConnesso(String username,String password){}`
- `List<Cliente> eseguiQueryRicercaEventiArtista(String Artista) {}`

Plan Test metodo *eseguiQueryRicercaImpiegatoConnesso:*

Questo metodo ha due parametri di ingresso, ancora nel range del numero di parametri accettabili per un test JUnit, essendo di tipo *String* e non essendo limitati procedere con i Plan che per un tipo testing BlackBox sarebbe stato improponibile, abbiamo optato per svolgere un Testing di tipo **White-Box** avendo il codice sorgente a disposizione abbiamo analizzato le varie vie da seguire.

Abbiamo quindi diviso le possibili combinazioni di input, considerando due possibili valori di ingresso : **String == null** oppure **String !=null,** per entrambe le stringhe in input.
Portandoci alla seguente :

1) Username != null	Password !=null
2) Username == null	Password !=null
3) Username != null	Password == null
4) Username == null	Password == null

Quindi questi sono stati i primi test cases che volevamo testare, ma sappiamo che un utente è SEMPRE identificato da entrambi quindi i casi 2,3 e 4 sono stati fusi in unico caso, che è quello del mancato inserimento di almeno uno dei valori da inserire.

Codice Sorgente:

```
public Impiegato eseguiQueryRicercaImpiegatoConnesso(String username, String
password) throws SQLException{

        if( username == null || password == null || username.equals("") ||
password.equals("")) return null;
        PreparedStatement preparedStatement = null;
        ResultSet resultSet = null;
        String query = " SELECT * FROM IMPIEGATO WHERE USERNAME = ? AND PASSWORD
= ? ";

        preparedStatement =
UtilityDB.getConnessioneDB().prepareStatement(query);
        preparedStatement.setString(1,username);
        preparedStatement.setString(2,password);
        resultSet = preparedStatement.executeQuery();
        Impiegato imp = null;

        resultSet.next();
        String usernameI = (resultSet.getString("USERNAME"));
        String passwordI = (resultSet.getString("PASSWORD"));
        String amministratore = (resultSet.getString("ADMIN"));
        int id = (resultSet.getInt("ID"));
        imp = new Impiegato(usernameI,passwordI,amministratore,id);

        return imp;

    }
```

Ora ci siamo chiesti come potevamo coprire tutto il codice del metodo, da avere così il **TER = 100%**.

Abbiamo optato per il **Branch Coverage :**
Il primo if sarà superato solo con l'inserimento di due valori diversi da null e dalla stringa vuota, ma abbiamo dovuto aggiungere altri casi di test per coprire tutto il codice, poiché l'istruzione 'resulSet.getString("Username"), lancerà un eccezione se il resulSet è effettivamente vuoto, quindi non è stato trovato nessun Impiegato con questi dati, abbiamo deciso di aggiungere dei test case che assicurassero che l'eccezione sarebbe stata lancia con input non presendi nel DB.

Codice Sorgente Test Cases:

```
 private GestoreQueryCerca gestoreQueryCerca;

    @Before public void setUp()  throws Exception {
        gestoreQueryCerca = new GestoreQueryCerca();
        assertNotNull(gestoreQueryCerca);
    }

    @After public void tearDown() throws  Exception {
        gestoreQueryCerca = null;
        assertNull(gestoreQueryCerca);
    }

    /**
Primo caso:
```

```
Username e password inseriti e presente nel DB
*/
    @Test public void testeseguiQueryRicercaImpiegatoConnesso1() {
        Impiegato testAdmin = new Impiegato("admin","admin","V",43);
        Impiegato testResult = null;
        try {
            testResult =
gestoreQueryCerca.eseguiQueryRicercaImpiegatoConnesso("admin", "admin");
        } catch ( SQLException e ) {
            e.printStackTrace();
        }

        assertEquals(testAdmin,testResult);
    }
    /**
Secondo caso:
Username &&  password non inseriti
*/
    @Test public void testeseguiQueryRicercaImpiegatoConnesso2() {
        Impiegato testResult = null;
        try {
            testResult =
gestoreQueryCerca.eseguiQueryRicercaImpiegatoConnesso("","");
        } catch ( SQLException e ) { e.printStackTrace(); }

        assertNull(testResult);
    }
    /**
Terzo caso:
Username e password inseriti ma non presenti nel DB
*/
    @Test public void testeeseguiQueryRicercaImpiegatoConnesso3() {

        Impiegato testResult = null;
        try {
            // Queste credienziali non sono presenti nel DB
            testResult =
gestoreQueryCerca.eseguiQueryRicercaImpiegatoConnesso("prova","prova");
            fail();// Se non lancia un eccezione il test fallisce,
( resultSet.next() deve lanciarla precisamente  )
        } catch ( SQLException e ) { assertTrue(true); }

    }
```

Test Case metodo *eseguiQueryRicercaEventiArtista(String artista)*

In questo caso avendo come input solo una Stringa, abbiamo due casi di test, simili al test descritti sopra in **White-Box** con **Branch Coverage:**

Il test sarà effettuando confrontando il risultato della chiamata del metodo con un artista che sappiamo già a quali eventi ha partecipanto.

Codice Sorgente:

```
  public List<Evento> eseguiQueryRicercaEventiArtista(String Artista)  throws
SQLException {

        String query = " SELECT ID_EVENTO FROM PARTECIPANTI_EVENTO WHERE
PARTECIPANTE = ? ";
        PreparedStatement preparedStatement =
UtilityDB.getConnessioneDB().prepareStatement(query);
```

```
        preparedStatement.setString(1,Artista);
        ResultSet resultSet = preparedStatement.executeQuery();
        List<Evento> listaEvento = new ArrayList<>();

        resultSet.next()
            listaEvento.add(eseguiQueryRicercaEventiId(resultSet.getInt(1)));

        UtilityDB.closeDB(preparedStatement);
        return listaEvento;
    }
```

Codice Sorgente Test cases:

```
      /**
      * Primo Caso:
      * testiamo il numero di partecipanti nel Set<String>,
      * con un Set<String> dell'evento che sappiamo già quanti eventi
      */

    @Test public void testeseguiQueryRicercaEventiArtista1() {

        // L'artista in questione partecipa a 2 Eventi
        // con id 243 e 245, controlliamo quindi la lista di ritorno
        // che contenga questi due eventi
        String artistaInserito = "Ghali";
        List<Evento> testList = new ArraList<>();
        try {
            testList =
gestoreQueryCerca.eseguiQueryRicercaEventiArtista(artistaInserito);
        } catch (SQLException e ) { e.printStackTrace(); }
        // Se ha esattamente due elementi
        assertTrue(testList.size() == 2);
        // Ordino la lista per criterio di ordimaento numero crescendi di eventi
        Collections.sort(testList, new Comparator<Evento>() {
            @Override
            public int compare(Evento o1, Evento o2) {
                if (o1.getIdEvento()<o2.getIdEvento()) return -1;
                else if(o1.getIdEvento() > o2.getIdEvento()) return 1;
                return 0;
            }
        });
        assertTrue(testList.get(0).getId()==243);
        assertTrue(testList.get(1).getId()==245);
    }

    /**
      * Secondo caso:
      * Controlliamo se inserendo un artista non presente nel DB
      * questo metodo lanci un eccezione
*     /
    @Test public void testeseguiQueryRicercaEventiArtista2() {
        String artistaInserito = "Ghali";
        List<Evento> testList = new ArraList<>();
        try {
            testList =
gestoreQueryCerca.eseguiQueryRicercaEventiArtista(artistaInserito);
            fail();
        } catch (SQLException e ) { e.printStackTrace(); }
        assertTrue(true);
    }
```