

```
1 <?xml version='1.0' encoding='utf-8'?>
2 <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
3 <hibernate-configuration>
4   <session-factory>
5     <property name="hibernate.connection.driver_class">
6       com.mysql.jdbc.Driver
7     </property>
8     <property name="hibernate.connection.url">
9       jdbc:mysql://localhost:3306/haim
10    </property>
11    <property name="hibernate.connection.username">haim</property>
12    <property name="hibernate.connection.password">michael</property>
13    <property name="show_sql">true</property>
14    <property name="dialect">
```

```
15 org.hibernate.dialect.  
    MySQLDialect</property>  
16          <property name="hibernate  
    .hbm2ddl.auto">update</property>  
17          <!-- Mapping files -->  
18          <mapping resource="  
    storeinventory.hbm.xml"/>  
19      </session-factory>  
20 </hibernate-configuration>
```

```
1 <?xml version="1.0"?>
2 <!DOCTYPE hibernate-mapping
PUBLIC
3           "-//Hibernate/Hibernate
Mapping DTD 3.0//EN"
4           "http://hibernate.
sourceforge.net/hibernate-mapping
-3.0.dtd">
5 <hibernate-mapping>
6   <class name="il.ac.hit.model.
Item" table="ITEMS">
7     <id name="
incrementalValue" type="int"
column="INCREMENTALVALUE" >
8       <generator class="
increment"/>
9     </id>
10
11   <property name="id">
12     <column name="ID"/>
13   </property>
14   <property name="userId">
15     <column name="USERID"
/>
16   </property>
17   <property name="cost">
18     <column name="COST"/>
19   </property>
```

```
20          <property name="date">
21              <column name="DATE"/>
22          </property>
23          <property name="type">
24              <column name="TYPE"/>
25          </property>
26      </class>
27
28      <class name="il.ac.hit.model.
User" table="USERS">
29          <id name="userId" type="
int" column="USERID" >
30              <generator class="
assigned"/>
31          </id>
32
33          <property name="userName"
>
34              <column name="
USERNAME"/>
35          </property>
36          <property name="lastName"
>
37              <column name="
LASTNAME"/>
38          </property>
39          <property name="password"
>
```

```
40 <column name="
41     PASSWORD"/>
42 </property>
43 </class>
44 </hibernate-mapping>
```

```
1 package il.ac.hit.model;
2
3 /**
4  * this class represents the
5  * items in our project Cost Manager
6  */
7
8 public class Item {
9
10    private int id;
11    private int userId;
12    private int cost;
13    private String date;
14    private String type;
15    private int incrementalValue;
16
17    /**
18     * Empty constructor
19     */
20
21    /**
22     * This is our program's Item
23     * constructor
24     * @param id item's id
25     * @param userId item's user
26     * id
```

```
25      * @param cost item's cost
26      * @param date item's date
27      * @param type item's type
28      * @throws ItemDAOException
29      if database is corrupted
30      */
31      public Item(int id, int
32          userId, int cost, String date,
33          String type) throws
34          ItemDAOException{
35          setId(id);
36          setUserId(userId);
37          setCost(cost);
38          setDate(date);
39          setType(type);
40      }
41
42      /**
43      * This is our program's Item
44      * constructor
45      * @param id item's id
46      * @param userId item of user
47      * id
48      * @param cost item's cost
49      * @param date item's date
50      * @param type item's type
51      * @param incrementalValue
52      * item's incremental value for
```

```
45 database
46     * @throws ItemDAOException
47     if database is corrupted
48     */
49     public Item(int id, int
50 userId, int cost, String date,
51 String type, int incrementalValue
52 ) throws ItemDAOException{
53     this(id,userId,cost,date,
54 type);
55     setIncrementalValue(
56 incrementalValue);
57 }
58 /**
59 * get incremental value of
60 the item
61 * @return An int type
62 containing the item's incremental
63 Value.
64 */
65 public int
66 getIncrementalValue() {
67     return incrementalValue;
68 }
69 /**
70 * Sets the item's
```

```
62 incrementalValue and checking validation
63     * @param incrementalValue An int type containing the item's incremental Value.
64     * @throws ItemDAOException if database is corrupted
65     */
66     public void
67         setIncrementalValue(int
68             incrementalValue) throws
69             ItemDAOException {
70             String temp =
71                 incrementalValue + "";
72             try {
73                 int checkedId =
74                     Integer.parseInt(temp); //if the exception was not thrown
75                 this.incrementalValue
76                     = incrementalValue;
77             } catch (
78                 NumberFormatException ex) {
79                 throw new
80                 ItemDAOException("wrong input
81                     type: incrementalValue");
82             }
83         }
84     }
85 }
```

```
76     /**
77      * get id of the item
78      * @return An int type
    containing the item's id.
79     */
80     public int getId() {
81         return id;
82     }
83
84     /**
85      * Sets the item's id and
    checking validation
86      * @param id An int type
    containing the item's id.
87      * @throws ItemDAOException
    if database is corrupted
88     */
89     public void setId(int id)
90         throws ItemDAOException {
91         String temp = id + "";
92         try {
93             int checkedId =
    Integer.parseInt(temp); //if the
    exception was not thrown
94             this.id = id;
95         } catch (
    NumberFormatException ex) {
96             throw new
```

```
95 ItemDAOException("wrong input
96 type: id");
97 }
98 /**
99  * get userId of the item
100 * @return An int type
101 containing the item's userId.
102 */
103 public int getUserId() {
104     return userId;
105 }
106
107 /**
108  * Sets the item's userId
109 and checking validation
110 * @param userId An int type
111 containing the item's userId.
112 * @throws ItemDAOException
113 if database is corrupted
114 */
115 public void setId(int
116 userId) throws ItemDAOException
117 {
118     String temp = userId +
119     "";
120     try {
```

```
115             int checkedId =
116                 Integer.parseInt(temp); //if the
117                 exception was not thrown
118             this.userId = userId
119             ;
120         }
121
122         /**
123          * get cost of the item
124          * @return An int type
125          * containing the item's cost.
126      */
127     public int getCost() {
128         return cost;
129     }
130
131     /**
132         * Sets the item's cost and
133         * checking validation
134         * @param cost An int type
135         * containing the item's cost.
136         * @throws ItemDAOException
```

```
133 if database is corrupted
134     */
135     public void setCost(int cost
136         ) throws ItemDAOException {
137         String temp = cost + "";
138         try {
139             int checkedCost =
140                 Integer.parseInt(temp); //if the
141                 exception was not thrown
142                 this.cost = cost;
143             } catch (
144                 NumberFormatException ex) {
145                 throw new
146                 ItemDAOException("wrong input
147                 type: id");
148             }
149         /**
150             * get date of the item
151             * @return A string type
152             containing the item's date.
153             */
154         public String getDate() {
155             return date;
156         }
157         /**
158             */
```

```
154     * Sets the item's date and  
155     * checking validation  
156     * @param date A string type  
157     * containing the item's date.  
158     * @throws ItemDAOException  
159     * if database is corrupted  
160     */  
161     public void setDate(String  
162         date) throws ItemDAOException {  
163         if (date.equals(" ")) ||  
164             date.length() == 0) {  
165             throw new  
166                 ItemDAOException("Empty string:  
167                 date");  
168         } else this.date = date;  
169     }  
170  
171     /**  
172     * get type of the item  
173     * @return A string type  
174     * containing the item's type.  
175     */  
176     public String getType() {  
177         return type;  
178     }  
179  
180     /**  
181     * Sets the item's type and  
182     * checking validation  
183     * @param type A string type  
184     * containing the item's type.  
185     * @throws ItemDAOException  
186     * if database is corrupted  
187     */  
188     public void setType(String  
189         type) throws ItemDAOException {  
190         if (type.equals(" ")) ||  
191             type.length() == 0) {  
192             throw new  
193                 ItemDAOException("Empty string:  
194                 type");  
195         } else this.type = type;  
196     }
```

```
173 checking validation
174     * @param type A string type
175     * @throws ItemDAOException
176     if database is corrupted
177     */
178     public void setType(String
179     type) throws ItemDAOException {
180         if (type.equals(" ")) ||
181         type.length() == 0) {
182             throw new
183             ItemDAOException("Empty string:
184             type");
185         } else this.type = type;
186     }
187 }
```

```
1 package il.ac.hit.model;
2
3 /**
4  * this class represents the
5  * users in our project Cost Manager
6  *
7
8  private int userId;
9  private String userName;
10 private String lastName;
11 private String password;
12
13 /**
14  * Empty constructor
15 */
16 public User() {
17 }
18
19 /**
20  * This is our program's User
21  * constructor
22  * @param userId user's
23  * @param userName user's
24  * @param lastName user's
```

```
23 lastName
24     * @param password user's
25     * @throws UserDAOException
26     if database is corrupted
27     */
28     public User(int userId,
29     String userName, String lastName
30     , String password) throws
31     UserDAOException {
32         setId(userId);
33         setUserName(userName);
34         setLastName(lastName);
35         setPassword(password);
36     }
37
38     /**
39      * get userId of the user
40      * @return An int type
41      containing the item's userId.
42      */
43     /**
44      * Sets the user's userId and
```

```
44     checking validation
45     * @param userId An int type
46     containing the user's userId.
47     * @throws UserDAOException
48     if database is corrupted
49     */
50     public void setUserId(int
51     userId) throws UserDAOException {
52         String temp = userId + ""
53         ;
54         try {
55             int checkedId =
56             Integer.parseInt(temp); //if the
57             exception was not thrown
58             this.userId = userId;
59         } catch (
60             NumberFormatException ex) {
61             throw new
62             UserDAOException("wrong input
63             type: incrementalValue");
64         }
65     }
66
67     /**
68      * get userName of the user
69      * @return A string type
70      containing the user's userName.
71      */
```

```
62     public String getUserName() {
63         return userName;
64     }
65
66     /**
67      * Sets the user's userName
68      * and checking validation
69      * @param userName A string
70      * type containing the user's
71      * userName.
72      * @throws UserDAOException
73      * if database is corrupted
74      */
75     public void setUserName(
76         String userName) throws
77         UserDAOException {
78         if (userName.equals(" "))
79         || userName.length() == 0) {
80             throw new
81             UserDAOException("Empty string:
82 date");
83         } else this.userName =
84         userName;
85     }
86
87     /**
88      * get lastName of the user
89      * @return A string type
```

```
79 containing the user's lastName.  
80 */  
81     public String getLastName  
82     {  
83         return lastName;  
84     }  
85     /**  
86      * Sets the user's lastName  
87      * and checking validation  
88      * @param lastName A string  
89      * type containing the user's  
90      * lastName.  
91      * @throws UserDAOException  
92      * if database is corrupted  
93      */  
94     public void setLastName(  
95         String lastName) throws  
UserDAOException {  
91         if (lastName.equals(" ")  
92             || lastName.length() == 0) {  
92             throw new  
UserDAOException("Empty string:  
date");  
93         } else this.lastName =  
lastName;  
94     }  
95 }
```

```
96     /**
97      * get password of the user
98      * @return A string type
99      * containing the user's password.
100     public String getPassword
101    () {
102      return password;
103    }
104    /**
105     * Sets the user's password
106     * and checking validation
107     * @param password A string
108     * type containing the user's
109     * password.
110     * @throws UserDAOException
111     * if database is corrupted
112     */
113    public void setPassword(
114      String password) throws
115      UserDAOException {
116      if (password.equals(" "))
117      || password.length() == 0) {
118        throw new
119        UserDAOException("Empty string:
120        date");
121      } else this.password =
```

```
112 password;  
113     }  
114 }  
115
```

```
1  /**
2  *
3  */
4  package il.ac.hit.model;
5
6  import java.util.List;
7
8  /**
9   * Represents the interface of
10  * HibernateCostManagerDAO class
11  * gives more accessibility and
12  * flexibility to the code
13  */
14  public interface ICostManagerDAO
15  {
16
17      /**
18       * getting instance of
19       * session ,saving the object of the
20       * specific item to our database
21       * if not succeed a rollback
22       * will be committed and an error
23       * message will be sent
24       * finally closing the
25       * session upon completion
26       * @param ob Item's object
27       * @throws ItemDAOException
28       * if database is corrupted
29  }
```

```
20      */
21      public void addItem(Item ob)
22          throws ItemDAOException;
23
24      /**
25      * getting instance of
26      session.
27      * getting all items from
28      database for this specific user's
29      id,
30      * if not succeed a rollback
31      will be committed and an error
32      message will be sent
33      *
34      * finally will close the
35      session upon completion
36      *
37      * @param userId getting user
38      's id
39      *
40      * @return list of items of
41      this specific user's id
42      *
43      * @throws ItemDAOException
44      if database is corrupted
45      */
46
47      public List<Item> getItems(
48          int userId) throws
49          ItemDAOException;
50
51      /**
52      * getting instance of
```

```
35 session ,saving the object of the
      specific user to our database
36           * if not succeed a rollback
      will be committed and an error
      message will be sent
37           * finally closing the
      session upon completion
38           * param ob User's object
39           * throws UserDAOException
      if database is corrupted
40           */
41     public void addUser(il.ac.hit
      .model.User ob) throws
UserDAOException;
42
43     /**
44      * getting instance of
      session.
45      * getting all users from
      database,
46      * if not succeed a rollback
      will be committed and an error
      message will be sent
47      * finally will close the
      session upon completion
48      * return list of items of
      this specific user's id
49      * throws UserDAOException
```

```
49 if database is corrupted
50 */
51     public List<User> getUsers()
52         throws UserDAOException;
53     /**
54         * getting instance of
55         * session.
56         * getting all the items that
57         * match the month and year from
58         * database,
59         * if not succeed a rollback
60         * will be committed and an error
61         * message will be sent
62         * finally will close the
63         * session upon completion
64         * param month specific
65         * month of bought items that the
66         * user want to view
67         * param year specific year
68         * of bought items that the user
69         * want to view
70         * return list of items that
71         * corresponding to the input
72         * parameters
73         * throws ItemDAOException
74         * if database is corrupted
75         */
76 }
```

```
63     public List<Item> getReport(  
64         String month ,String year,int  
65         userId) throws ItemDAOException;  
66  
67          * getting instance of  
68      session.  
69      * checking if the user  
70      exists in the database,  
71      * if not succeed a rollback  
72      will be committed and an error  
73      message will be sent  
74      * finally will close the  
75      session upon completion  
76      * @param ob User's object  
77      * @return a boolean  
78      parameter that is true if user  
79      exists and false otherwise  
80      * @throws UserDAOException  
81      if database is corrupted  
82      */  
83     public boolean isUserExist(il  
84         .ac.hit.model.User ob) throws  
85         UserDAOException;  
86     }  
87 
```

```
1 package il.ac.hit.model;
2
3
4 /**
5  * Represents the exception class
6  * which is handling exceptions
7  * related to
8  * HibernateCostManagerDAO that
9  * possible to happen.
10 */
11 /**
12  * Exception constructor with
13  * @param message Exception's
14  * message
15  */
16 public ItemDAOException(
17     String message) {
18     super(message);
19 }
20 /**
21  * Exception constructor with
22  * message and cause
```

```
21      * @param message Exception's
22      * @param cause Exception's
23      */
24      public ItemDAOException(
25          String message, Throwable cause
26      ) {
27
28      /**
29      * Empty exception
30      * constructor
31      */
32      public ItemDAOException() {
33          super();
34
35
36
37      }
38
```

```
1 package il.ac.hit.model;
2
3
4 /**
5  * Represents the exception class
6  * which is handling exceptions
7  * related to
8  * HibernateCostManagerDAO that
9  * possible to happen.
10 */
11 /**
12  * Exception constructor with
13  * @param message Exception's
14  * message
15  */
16 public UserDAOException(
17     String message) {
18     super(message);
19 }
20 /**
21  * Exception constructor with
22  * message and cause
```

```
21      * @param message Exception's
22      * @param cause Exception's
23      */
24  public UserDAOException(
25      String message, Throwable cause
26  ) {
27
28      /**
29      * Empty exception
30      * constructor
31      */
32  public UserDAOException() {
33      super();
34
35
36 }
```

```
1 package il.ac.hit.model;
2
3 import org.hibernate.*;
4 import org.hibernate.cfg.
5 AnnotationConfiguration;
6 import java.util.ArrayList;
7 import java.util.List;
8
9 /**
10  * a class representing model in
11  * our project
12  * which is managing our database
13  */
14 public class
15 HibernateCostManagerDAO
16 implements ICostManagerDAO {
17
18     private static SessionFactory
19         factory;
20     private static
21     HibernateCostManagerDAO
22     costManagerDAO=null;
23
24     /**
25      * Empty constructor
26      */
27     private
28     HibernateCostManagerDAO() {
```

```
21    }
22
23    /**
24     * A design pattern for our
25     * database
26     * Singleton implementation
27     * for HibernateCostManagerDAO
28     * @return static class
29     * member
30     */
31
32     public static
33     HibernateCostManagerDAO
34     getHibernateCostManagerDAO(){
35         if(costManagerDAO==null){
36             costManagerDAO=new
37             HibernateCostManagerDAO();
38         }
39         return costManagerDAO;
40     }
41
42     /**
43      * A design pattern for our
44      * database
45      * for getting a session per
46      * user
47      * @return return static
48      * class member ,session factory
49      */
50
```

```
40     public static SessionFactory  
41         getSessionFactory(){  
42             if(factory==null){  
43                 factory=new  
44                     AnnotationConfiguration().  
45                     configure().buildSessionFactory()  
46                     ();  
47             }  
48             return factory;  
49         }  
50     }  
51  
52     /**  
53      * getting instance of  
54      session ,saving the object of the  
55      specific item to our database  
56      * if not succeed a rollback  
57      will be committed and an error  
58      message will be sent  
59      * finally closing the  
60      session upon completion  
61      * @param ob Item's object  
62      * @throws ItemDAOException  
63      if database is corrupted  
64      */  
65     @Override  
66     public void addItem(Item ob)  
67     throws ItemDAOException {  
68         Transaction tx=null;
```

```
57         Session session =
58             getSessionFactory().openSession
59             ();
60             try {
61                 tx = session.
62                 beginTransaction();
63                 session.save(ob);
64                 session.
65                 getTransaction().commit();
66             } catch (
67                 HibernateException e) {
68                 if (tx != null)
69                     tx.rollback();
70                 throw new
71                 ItemDAOException("Can't add item"
72                 , e);
73             } finally {
74                 try {
75                     session.close
76                 () ;
77                 } catch (
78                 HibernateException e) {
79
80                 }
81             }
82         }
83
84     /**
85      *
```

```
75      * getting instance of
    session.
76      * getting all items from
    database for this specific user'
    s id,
77      * if not succeed a rollback
    will be committed and an error
    message will be sent
78      * finally will close the
    session upon completion
79      * @param userId getting
    user's id
80      * @return list of items of
    this specific user's id
81      * @throws ItemDAOException
    if database is corrupted
82      */
83 @Override
84 public List<Item> getItems(
    int userId) throws
    ItemDAOException {
85     Transaction tx=null;
86     List<Item> list =new
    ArrayList<>();
87
88     Session session =
    getSessionFactory().openSession
    ();
```

```
89         try {
90             tx = session.
91             beginTransaction();
92             List items = session
93               .createQuery("from Item").list
94             ();
95             List userIdPerUser
96               = session.createQuery("select
97                 userId from Item").list();
98             for (int i = 0; i <
99                 items.size(); i++)
100               if (
101                 userIdPerUser.get(i).equals(
102                   userId))
103                 list.add((
104                   Item) items.get(i));
105             session.
106             getTransaction().commit();
107         }
108         catch (
109           HibernateException e){
110             if(tx!=null)
111               tx.rollback();
112             throw new
113               ItemDAOException("Can't get list
```

```
104    of items",e);
105    }
106    finally{
107        try{
108            session.close();
109        }
110        catch (
111            HibernateException e){
112        }
113    }
114    return list;
115}
116
117 /**
118     * getting instance of
119     * session ,saving the object of
120     * the specific user to our
121     * database
122     * if not succeed a rollback
123     * will be committed and an error
124     * message will be sent
125     * finally closing the
126     * session upon completion
127     * @param ob User's object
128     * @throws UserDAOException
129     * if database is corrupted
130     */
131 }
```

```
124     @Override
125     public void addUser(User ob
126 ) throws UserDAOException {
127         Transaction tx=null;
128         if(!isUserExist(ob)){
129
130             Session session =
131                 getSessionFactory().openSession
132                 ();
133
134             try {
135                 tx = session.
136                 beginTransaction();
137                 session.save(ob
138             );
139             session.
140             getTransaction().commit();
141         } catch (
142             HibernateException e) {
143             if (tx != null)
144                 tx.rollback();
145             throw new
146             UserDAOException("Can't add user
147                 ", e);
148         } finally {
149             try {
150                 session.
151                 close();
152             }
153         }
154     }
155 }
```

```
141 } catch (HibernateException e) {
142 }
143 }
144 }
145 }
146 }
147 /**
148 * getting instance of session.
149 * getting all users from database,
150 * if not succeed a rollback will be committed and an error message will be sent
151 * finally will close the session upon completion
152 * @return list of items of this specific user's id
153 * @throws UserDAOException if database is corrupted
154 */
155 @Override
156 public List<User> getUsers()
157     throws UserDAOException {
158
159     Transaction tx=null;
```

```
160         List<User> list =new
161             ArrayList<>();
162         Session session =
163             getSessionFactory().openSession
164             ();
165         try {
166             tx = session.
167             beginTransaction();
168             List users = session
169             .createQuery("from User").list
170             ();
171             for (int i = 0; i <
172                 users.size(); i++)
173                 list.add((User)
174                     users.get(i));
175             session.
176             getTransaction().commit();
177         }
178         catch (
179             HibernateException e){
180             if(tx!=null)
181                 tx.rollback();
182             throw new
183             UserDAOException("Can't get list
184             of users",e);
185         }
```

```
176         }
177     }  
178     finally{
179         try{
180             session.close();
181         }
182         catch (
183             HibernateException e){
184         }
185         return list;
186     }
187
188     /**
189      * getting instance of
190      * session.
191      * getting all the items
192      * that match the month and year
193      * from database,
194      * if not succeed a rollback
195      * will be committed and an error
196      * message will be sent
197      * finally will close the
198      * session upon completion
199      * @param month specific
200      * month of bought items that the
201      * user want to view
202      * @param year specific year
```

```
194    of bought items that the user  
want to view  
195        * @return list of items  
that corresponding to the input  
parameters  
196        * @throws ItemDAOException  
if database is corrupted  
197        */  
198    @Override  
199    public List<Item> getReport(  
String month, String year ,int  
userId) throws ItemDAOException  
{  
200        Transaction tx=null;  
201        List<Item> list =new  
ArrayList<>();  
202  
203        Session session =  
getSessionFactory().openSession  
();  
204        try {  
205            tx = session.  
beginTransaction();  
206            List items = session  
.createQuery("from Item").list  
();  
207            List dateFromItem =  
session.createQuery("select date
```

```
207     from Item").list();
208             List userIdFromItem
209             = session.createQuery("select
210               userId from Item").list();
211
212             for (int i = 0; i <
213               items.size(); i++) {
214                 String str = (
215                   String) dateFromItem.get(i);
216                 String[]
217                 afterSplit = str.split("-");
218                 int userIdFromDB
219                 =(Integer)userIdFromItem.get(i);
220
221                 if (afterSplit[0]
222                   .equals(year) && afterSplit[1].
223                     equals(month) && userId ==
224                     userIdFromDB)
225
226                   list.add((
227                     Item) items.get(i));
228
229                 }
230             session.
231             getTransaction().commit();
232         }
233         catch (
234           HibernateException e){
235
236             if(tx!=null)
237               tx.rollback();
238         }
239       }
240     }
241   }
242 }
```

```
223         throw new
224             ItemDAOException("Can't get list
225                 of items",e);
226         }
227         finally{
228             try{
229                 session.close();
230             }
231             catch (
232                 HibernateException e){
233                 return list;
234             }
235         }
236     /**
237      * getting instance of
238      * checking if the user
239      * exists in the database,
240      * if not succeed a rollback
241      * will be committed and an error
242      * message will be sent
243      * finally will close the
244      * session upon completion
245      * param ob User's object
246      * return a boolean
```

```
242 parameter that is true if user  
exists and false otherwise  
243     * @throws UserDAOException  
     if database is corrupted  
244     */  
245     @Override  
246     public boolean isUserExist(  
         User ob) throws UserDAOException  
    {  
247         Transaction tx=null;  
248         Session session =  
             getSessionFactory().openSession()  
();  
249         boolean flag = false;  
250         try{  
251             tx=session.  
beginTransaction();  
252             List users = session  
                 .createQuery("select userId from  
User").list();  
253             for (Object x:users  
) {  
254                 if(x.equals(ob.  
getUserId())) {  
255                     flag=true;  
256                     break;  
257                 }  
258             }  
    }
```

```
259             session.  
260         getTransaction().commit();  
261     }  
262     catch (br/>263         HibernateException e){  
264         if(tx!=null)  
265             tx.rollback();  
266         throw new  
267             UserDaoException("User not exist  
268             ",e);  
269     }  
270     finally{  
271         try{  
272             session.close();  
273         }  
274         catch (br/>275             HibernateException e){  
276         }  
277     }  
278 }
```

```
1 package il.ac.hit.controller;
2
3 import javax.servlet.
4 RequestDispatcher;
5 import javax.servlet.
6 ServletException;
7 import javax.servlet.annotation.
8 WebServlet;
9 import javax.servlet.http.
10 HttpServlet;
11 import javax.servlet.http.
12 HttpServletRequest;
13 import javax.servlet.http.
14 HttpServletResponse;
15 import java.io.IOException;
16 import java.io.PrintWriter;
17 import java.lang.reflect.
18 InvocationTargetException;
19 import java.lang.reflect.Method;
20
21 /**
22  * Servlet implementation class
23  * RouterServlet
24  * directs the incoming requests
25  * to the requested Controller
26  */
27 @WebServlet("/controller/*")
28 public class Router extends
```

```
19 HttpServlet {  
20     private static final long  
21     serialVersionUID = 1L;  
22     public static String  
23     CONTROLLERS_PACKAGE = "il.ac.hit.  
24     controller";  
25     /**  
26     * @see HttpServlet#  
27     HttpServlet()  
28     */  
29     public Router() {  
30         super();  
31         // TODO Auto-generated  
32         constructor stub  
33     }  
34     protected void doGet(  
35     HttpServletRequest request,  
36     HttpServletResponse response)  
37     throws
```

```
35  ServletException, IOException {  
36  
37      try {  
38          response.  
39          setContentType("text/html");  
40          // TODO Auto-  
41          generated method stub  
42          // Returns the part  
43          // of this request's URL from the  
44          // protocol name up to  
45          // the query string  
46          // in the first line of the HTTP  
47          // request.  
48          String text = request  
49          .getRequestURI();  
50          //Splits this string  
51          // around matches of the given  
52          // regular expression.  
53          String[] texts = text  
54          .split("/");  
55          // extracting  
56          // controller and action  
57          String controller =  
58          texts[3];  
59          String action = texts  
60          [4];  
61
```

```
50          // building the full
      qualified name of the controller
51          String temp =
      controller + "Controller";
52          String
      controllerClassName =
      CONTROLLERS_PACKAGE + "."
      + temp.
      substring(0, 1).toUpperCase() +
      temp.substring(1);
54
55          // instantiating the
      controller class and calling
56          // the action method
      on the controller object
57          Class clazz = Class.
      forName(controllerClassName);
58          Method method = clazz
      .getMethod(action,
      HttpServletRequest.class,
      HttpServletResponse.class);
59
60          if(request.getSession
      ().getAttribute("userid")==null
      && action.equals("additem")) {
61              PrintWriter out
      = response.getWriter();
      out.println("User
```

```
62     is not logged in");
63     }
64     else{
65         //activate the
66         //action method by reflection
67         method.invoke(
68             clazz.getDeclaredConstructor().
69             newInstance(), request, response
70         );
71         // creating a
72         // RequestDispatcher object that
73         // points at the JSP document
74         // which is view
75         // of our action
76         RequestDispatcher
77         dispatcher = getServletContext
78         ().getRequestDispatcher("/views/"
79         + action + ".jsp");
80         dispatcher.
81         include(request, response);
82     }
83     }
84     }catch (
85     ClassNotFoundException |
86     InstantiationException |
87     IllegalAccessException |
88     NoSuchMethodException |
```

```
74 SecurityException |  
    IllegalArgumentException |  
    InvocationTargetException e) {  
75         //Prints this  
        throwable and its backtrace to  
        the standard error stream.  
76         e.printStackTrace();  
77     }  
78 }  
79  
80 /**  
81     * Called by the server (via  
     * the service method) to allow a  
     * servlet to handle a POST request  
     * .  
82     * @see HttpServlet#doPost(  
     *      HttpServletRequest request,  
     *      HttpServletResponse response)  
83     */  
84     protected void doPost(  
     *      HttpServletRequest request,  
     *      HttpServletResponse response)  
85     throws  
     *      ServletException, IOException {  
86         // TODO Auto-generated  
         method stub  
87         doGet(request, response  
     );
```

```
88     }
89 }
90
```

```
1 package il.ac.hit.controller;  
2  
3 /**  
4  * Represents the interface of  
5  * the UserController and  
6  * ItemController  
7 */  
8 public interface IController {  
9 }
```

```
1 package il.ac.hit.controller;
2
3 import il.ac.hit.model.*;
4 import org.hibernate.Session;
5
6 import javax.servlet.http.
7     HttpServletRequest;
8 import javax.servlet.http.
9     HttpServletResponse;
10 import javax.servlet.http.
11     HttpSession;
12 import java.io.IOException;
13 import java.io.PrintWriter;
14 import java.util.List;
15
16 /**
17 * This class receives Item
18 * requests coming from the router,
19 * handles them according to the
20 * client's requests
21 */
22 public class ItemController
23     implements IController {
24
25     private static
26     HibernateCostManagerDAO dao;
27     private static int incval=5;
```

```
22     /**
23      * action that getting the
24      requested items from User
25      Interface
26      * @param request the
27      session is a reference to the
28      request per user
29      * @param response the
30      session is a reference to the
31      response per user
32      * @throws ItemDAOException
33      if database is corrupted
34      */
35
36     public void additem(
37         HttpServletRequest request,
38         HttpServletResponse response)
39     throws ItemDAOException,
40     IOException {
41         try{
42             dao=
43             HibernateCostManagerDAO.
44             getHibernateCostManagerDAO();
45
46             //checking if
47             parameters not null and getting
48             parameters from the received
49             request
50             if(request.getSession
```

```
33    ().getAttribute("userid")!=null  
     &&  
34          request.  
        getParameter("id") !=null &&  
35          request.  
        getParameter("cost")!=null &&  
36          request.  
        getParameter("date")!=null &&  
37          request.  
        getParameter("type")!=null ) {  
38            int userIdPerUser  
            = (Integer) request.getSession()  
              .getAttribute("userid");  
39            String temp =  
            request.getParameter("id");  
40            int idPerUser =  
            Integer.parseInt(temp);  
41            int costPerUser  
            = Integer.parseInt(request.  
              getParameter("cost"));  
42            String  
            datePerUser = request.  
              getParameter("date");  
43            String  
            typePerUser = request.  
              getParameter("type");  
44  
45          //creating a new
```

```
45 Item object
46             Item item = new
47             Item(idPerUser, userIdPerUser,
48             costPerUser, datePerUser,
49             typePerUser, incval);
50             //add item to db
51             dao.addItem(item
52             );
53             //getting items
54             //according to input userId
55             List<Item>
56             itemList = dao.getItems(
57             userIdPerUser);
58             if (itemList !=
59             null)
60             request.
61             getSession().setAttribute("
62             tableitems", itemList);
63             }
64         } catch (ItemDAOException
65             e){
66             throw new
67             ItemDAOException("Error with
68             adding item",e);
69         }
70     }
71
72     /**
73      * @param id
74      * @return
75      */
76     public Item getItem(int id)
77     {
78         Item item = null;
79         try
80         {
81             item = (Item)getSession()
82             .get(Item.class, id);
83         }
84         catch (Exception e)
85         {
86             e.printStackTrace();
87         }
88         return item;
89     }
90 }
```

```
60      * action that getting all  
61      * the items that match the month  
62      * and year from User Interface  
63      * @param request the  
64      * session is a reference to the  
65      * response per user  
66      * @param response the  
67      * session is a reference to the  
68      * response per user  
69      * @throws ItemDAOException  
70      * if database is corrupted  
71      */  
72      public void report(  
73          HttpServletRequest request,  
74          HttpServletResponse response)  
75      throws ItemDAOException {  
76          List<Item> reportList;  
77          try{  
78              dao=  
79                  HibernateCostManagerDAO.  
80                  getHibernateCostManagerDAO();  
81              int userId = (Integer  
82                  ) request.getSession().  
83                  getAttribute("userid");  
84              //getting parameters  
85              //from the received request  
86              String month=request.  
87              getParameter("month");  
88          } catch (ItemDAOException e) {  
89              e.printStackTrace();  
90          }  
91      }
```

```
72             String year=request.  
73                 getParameter("year");  
74             reportList=dao.  
75                 getReport(month,year,userId);  
76             request.getSession().  
77                 setAttribute("report", reportList  
78             );  
79         }  
80     }  
81 }
```

```
1 package il.ac.hit.controller;  
2  
3 import il.ac.hit.model.  
        HibernateCostManagerDAO;  
4 import il.ac.hit.model.  
        ICostManagerDAO;  
5 import il.ac.hit.model.User;  
6 import il.ac.hit.model.  
        UserDAOException;  
7 import javax.servlet.  
        ServletException;  
8 import javax.servlet.http.  
        HttpServletRequest;  
9 import javax.servlet.http.  
        HttpServletResponse;  
10 import javax.servlet.http.  
        HttpSession;  
11 import java.io.IOException;  
12 import java.util.List;  
13  
14 /**  
15  * This class receives User  
16  * requests coming from the router,  
17  * handles them according to the  
18  * client's requests  
19  */  
20 public class UserController  
21     implements IController {
```

```
19
20     private static
21         ICostManagerDAO dao;
22
23     /**
24      * login action handles
25      * request to find and verify the
26      * user with the database.
27      *
28      * @param request the
29      * session is a reference to the
30      * request per user
31      *
32      * @param response the
33      * session is a reference to the
34      * response per user
35      *
36      * @throws UserDAOException
37      * if database is corrupted
38      */
39
40     public void login(
41         HttpServletRequest request,
42         HttpServletResponse response)
43     throws UserDAOException {
44
45         List<User> users;
46
47         try {
48             dao =
49                 HibernateCostManagerDAO.
50                 getHibernateCostManagerDAO();
51
52             users = dao.getUsers
53                 ();
54
55         } catch (UserDAOException e) {
56             throw new UserDAOException("Error
57             occurred while trying to log in
58             user " + user.getLoginName());
59         }
60
61         HttpSession session = request
62             .getSession(true);
63
64         session.setAttribute("user", user);
65
66         response.sendRedirect("index.jsp");
67     }
68
69 }
```

```
33          //get all users from  
34          database  
35          //getting parameters  
36          from the received request  
37          String username =  
38          request.getParameter("username");  
39          String pass = request  
40          .getParameter("password");  
41          for (int i = 0; i <  
42          users.size(); i++) {  
43          if (users.get(i).  
44          getUserName().equals(username) &&  
45          users.get  
46          (i).getPassword().equals(pass)) {  
47          request.  
48          getSession().setAttribute("userid  
49          ", users.get(i).getUserId());  
50          request.  
51          getSession().setAttribute("check"  
52          , "User Logged in successfully");  
53          break;  
54          } else  
55          request.  
56          getSession().setAttribute("check"  
57          , "User is not signed up");  
58      }
```

```
48     } catch (UserDAOException  
49     e) {  
50         throw new  
51             UserDAOException("Error with  
52                 Login", e);  
53     }  
54     /**  
55      * action that handles a  
56      request to add a new user to the  
57      database if no such user exists  
58      create a new user.  
59      * @param request the  
60      session is a reference to the  
61      request per user  
62      * @param response the  
63      session is a reference to the  
64      response per user  
65      * @throws UserDAOException  
66      if database is corrupted  
67      */  
68      public void signup(  
69          HttpServletRequest request,  
70          HttpServletResponse response)  
71      throws UserDAOException {  
72          System.out.println("the
```

```
61 request is " + request.toString()
());
62         List<User> usersList;
63     try {
64         dao =
65             HibernateCostManagerDAO.
66             getHibernateCostManagerDAO();
67         usersList = dao.
68             getUsers();
69         boolean exists =
70             false;
71         if (request.
72             getParameter("userid") != null) {
73             for (int i = 0; i
74                 < usersList.size(); i++) {
75                 if (usersList
76                     .get(i).getUserId() == Integer.
77                     parseInt(request.getParameter(
78                         "userid"))) {
79                     exists =
80                     true;
81                 }
82                 request.
83                 getSession().setAttribute("check1
84                 ", "User already exists");
85             }
86             break;
87         }
88     }
```

```
76                     if (!exists) {  
77                         int userId  
78                             = Integer.parseInt(request.  
79                             getParameter("userid"));  
80                         String  
81                         userName = request.getParameter(  
82                             "username");  
83                         String  
84                         lastName = request.getParameter(  
85                             "lastname");  
86                         String pass  
87                         = request.getParameter("password");  
88                         User user =  
89                         new User(userId, userName,  
90                         lastName, pass);  
91                         dao.addUser(  
92                         user);  
93                         request.  
94                         getSession().setAttribute("check1", "User registered  
95                         successfully");  
96                     }  
97                 }  
98             } catch (  
99                 UserDAOException e) {  
100                 throw new
```

```
88 UserDAOException("Error with
89     Sign-up", e);
90 }
91
92 /**
93     * action that invalidates
94     * the session of the user
95     * @param request the
96     * session is a reference to the
97     * response per user
98     * @param response the
99     * session is a reference to the
100    * response per user
101   */
102
103   public void logout(
104       HttpServletRequest request,
105       HttpServletResponse response) {
106       HttpSession session =
107           request.getSession();
108       session.invalidate();
109   }
110 }
```