



CHAPTER 7



Separate Database, User Interface and Domain Logic

Example

This is a conference management system. The system needs to record the ID, name, telephone of each participant and the region he comes from. The participant ID is a unique integer assigned by the conference organizer. The name must be specified. Telephone can be omitted. All the participants must come from China, US or Europe.

We have created a database and a table to store the information of the participants. The schema is:

```
create table Participants (  
    id int primary key,  
    name varchar(20) not null,  
    telNo varchar(20),  
    region varchar(20)  
);
```

We have also written the code below to let an operator add a new participant. The system will check all the existing participants to find the maximum ID and then adds one to it and use it as the default value for the ID of this new participant. The operator may use this ID or use any other value. Please read the code carefully:

```
class AddParticipantDialog extends JDialog {  
    Connection dbConn;  
    JTextField id;  
    JTextField name;  
    JTextField telNo;  
    JTextField region;  
    AddParticipantDialog() {  
        setupComponents();  
        dbConn = ...;  
    }  
    void setupComponents() {  
        ...  
    }  
    void show() {  
        showDefaultValues();  
        setVisible(true);  
    }  
    void showDefaultValues() {  
        int nextId;  
        PreparedStatement st =
```

```

        dbConn.prepareStatement("select max(id) from participants");
    try {
        ResultSet rs = st.executeQuery();
        try {
            rs.next();
            nextId = rs.getInt(1)+1;
        } finally {
            rs.close();
        }
    } finally {
        st.close();
    }
    id.setText(new Integer(nextId).toString());
    name.setText("");
    region.setText("China");
}
void onOK() {
    if (name.getText().equals("")) {
        JOptionPane.showMessageDialog(this, "Invalid name");
        return;
    }
    if (!region.equals("China") &&
        !region.equals("US") &&
        !region.equals("Europe")) {
        JOptionPane.showMessageDialog(this, "Region is unknown");
        return;
    }
    PreparedStatement st =
        dbConn.prepareStatement("insert into from participants values(?,?,?,?)");
    try {
        st.setInt(1, Integer.parseInt(id.getText()));
        st.setString(2, name.getText());
        st.setString(3, telNo.getText());
        st.setString(4, region.getText());
        st.executeUpdate();
    } finally {
        st.close();
    }
    dispose();
}
}

```

This code looks normal, but it has mixed three different types of code together:

1. User interface: JDialog, JTextField, event handling.
2. Database access: Connection, PreparedStatement, SQL statements, ResultSet and etc.
3. Domain logic: Default value of the participant, that the name must be specified, the restriction on the region and etc. Domain logic is also called "domain model" or "business logic".

That these different types of code are mixed together causes the following problems:

- The code is complicated.
- The code is hard to reuse. If we need to create an `EditParticipantDialog` for the operator to edit a participant, we would like to reuse some domain logic (e.g., the restriction on the region). But currently the code implementing the domain logic is mixed with `AddParticipantDialog` and cannot be reused. It will be more difficult to reuse the domain logic in a web application.
- The code is hard to test. To test it, we have to setup a database and test it through the user interface.
- If the database schema is changed, `AddParticipantDialog` and many other places will have to be changed accordingly.
- It leads us to think in terms of low level concepts such as fields and records in database, instead of classes, objects, methods and attributes.

Therefore, we should separate these three types of code (user interface, database access and domain logic).

Extract the database access code

We will first extract the database access code. We can consider all the participants in the database as a set, i.e., there is no duplicate elements (participants) and there is no particular ordering between them. The set should support the operations of add, delete, update and enumeration:

```
class Participant {
    int id;
    String name;
    String telNo;
    String region;
    ...
}
interface ParticipantIterator {
    boolean next();
    Participant getParticipant();
}
class Participants {
    Connection dbConn;
    Participants() {
        dbConn = ...;
    }
    void addParticipant(Participant part) {
        PreparedStatement st =
            dbConn.prepareStatement("insert into from participants values(?, ?, ?, ?)");
        try {
```

```

        st.setInt(1, part.getId());
        st.setString(2, part.getName());
        st.setString(3, part.getTelNo());
        st.setString(4, part.getRegion());
        st.executeUpdate();
    } finally {
        st.close();
    }
}
}
void removeParticipant(int partId) {
    ...
}
void updateParticipant(Participant part) {
    ...
}
ParticipantIterator getAllParticipantsById() {
    ...
}
ParticipantIterator getParticipantsWithNameById(String name) {
    ...
}
}
}

```

In `AddParticipantDialog` in addition to adding a new participant, we also need to find the maximum participant ID so far. Therefore, we need to define a `getMaxId` method:

```

class Participants {
    Connection dbConn;
    void addParticipant(Participant part) {
        ...
    }
    int getMaxId() {
        PreparedStatement st =
            dbConn.prepareStatement("select max(id) from participants");
        try {
            ResultSet rs = st.executeQuery();
            try {
                rs.next();
                return rs.getInt(1);
            } finally {
                rs.close();
            }
        } finally {
            st.close();
        }
    }
    ...
}
}

```

Now, `AddParticipantDialog` can be simplified as:

```

class AddParticipantDialog extends JDialog {
    Participants participants;
    JTextField id;
    JTextField name;
    JTextField telNo;
    JTextField region;
}

```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```

AddParticipantDialog(Participants participants) {
    this.participants = participants;
    setupComponents();
}
void setupComponents() {
    ...
}
void show() {
    showDefaultValues();
    setVisible(true);
}
void showDefaultValues() {
    int nextId = participants.getMaxId()+1;
    id.setText(new Integer(nextId).toString());
    name.setText("");
    telNo.setText("");
    region.setText("China");
}
void onOK() {
    if (name.getText().equals("")) {
        JOptionPane.showMessageDialog(this, "Invalid name");
        return;
    }
    if (!region.equals("China") &&
        !region.equals("US") &&
        !region.equals("Europe")) {
        JOptionPane.showMessageDialog(this, "Region is unknown");
        return;
    }
    Participant part = new Participant(
        Integer.parseInt(id.getText()),
        name.getText(),
        telNo.getText(),
        region.getText());
    participants.addParticipant(part);
    dispose();
}
}
}

```

Now, AddParticipantDialog is much simpler. From its code we can't see that it is accessing the database at all!

Flexibility created by extracting the database access code

Because AddParticipantDialog now only deals with a set of participants instead of a database table, you may well use an XML file or a regular text file to store these participants. You only need to modify the Participants class, without changing AddParticipantDialog:

```

class Participants {
    void addParticipant(Participant part) {
        //save the participant into an XML file.
    }
    void getMaxId() {

```

```

        //find the maximum Id from the XML file.
    }
}

```

Even more, you may in some cases use a database and in some other cases use an XML file. All you need is to turn the Participants class into an interface and use the database in one implementation class and use an XML file in another:

```

interface Participants {
    void addParticipant(Participant part);
    int getMaxId();
    ...
}
class ParticipantsInDB implements Participants {
    void addParticipant(Participant part) {
        ...
    }
    int getMaxId() {
        ...
    }
}
class ParticipantsInXMLFile implements Participants {
    void addParticipant(Participant part) {
        //save the participant into an XML file.
    }
    int getMaxId() {
        //find the maximum Id from the XML file.
    }
}
class AddParticipantDialog extends JDialog {
    AddParticipantDialog(Participants participants) {
        ...
    }
    ...
}

```

Separate domain logic and user interface

Now, we will separate the domain logic and the user interface:

```

class Participant {
    int id;
    String name;
    String telNo;
    String region;
    ...
    static Participant makeDefaultParticipant() {
        return new Participant(0, "", "", "China");
    }
    void assertValid() throws ParticipantException {
        if (name.equals("")) {
            throw new ParticipantException("Invalid name");
        }
    }
}

```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```

    }
    if (!region.equals("China") &&
        !region.equals("US") &&
        !region.equals("Europe")) {
        throw new ParticipantException("Region is unknown");
    }
}
}
class ParticipantException extends Exception {
    ParticipantException(String msg) {
        super(msg);
    }
}
}

```

Now, AddParticipantDialog can be simplified as:

```

class AddParticipantDialog extends JDialog {
    Participants participants;
    JTextField id;
    JTextField name;
    JTextField telNo;
    JTextField region;
    AddParticipantDialog(Participants participants) {
        this.participants = participants;
        setupComponents();
    }
    void setupComponents() {
        ...
    }
    void show() {
        showDefaultValues();
        setVisible(true);
    }
    void showDefaultValues() {
        Participant newPart = Participant.makeDefaultParticipant();
        newPart.setId(participants.getMaxId()+1);
        showParticipant(newPart);
    }
    void showParticipant(Participant part) {
        id.setText(new Integer(part.getId()).toString());
        name.setText(part.getName());
        telNo.setText(part.getTelNo());
        region.setText(part.getRegion());
    }
    Participant makeParticipant() throws ParticipantException {
        Participant part = new Participant(
            Integer.parseInt(id.getText()),
            name.getText(),
            telNo.getText(),
            region.getText());
        part.assertValid();
        return part;
    }
    void onOK() {
        try {
            participants.addParticipant(makeParticipant());
            dispose();
        } catch (Exception e) {

```

```

        JOptionPane.showMessageDialog(this, e.getMessage());
    }
}
}

```

Now, AddParticipantDialog is even simpler. In particular, what it basically does is: show the data in a Participant object through the various UI components (showParticipant) and use the data in the UI components to make a Participant object (makeParticipant). Right, these are exactly the things that a user interface should do.

The improved code

The improved code is shown below:

```

class Participant {
    int id;
    String name;
    String telNo;
    String region;
    ...
    static Participant makeDefaultParticipant() {
        return new Participant(0, "", "", "China");
    }
    void assertValid() throws ParticipantException {
        if (name.equals("")) {
            throw new ParticipantException("Invalid name");
        }
        if (!region.equals("China") &&
            !region.equals("US") &&
            !region.equals("Europe")) {
            throw new ParticipantException("Region is unknown");
        }
    }
}
}
class ParticipantException extends Exception {
    ParticipantException(String msg) {
        super(msg);
    }
}
}
class Participants {
    Connection dbConn;
    Participants() {
        dbConn = ...;
    }
    void addParticipant(Participant part) {
        PreparedStatement st =
            dbConn.prepareStatement("insert into from participants values(?, ?, ?, ?)");
        try {
            st.setInt(1, part.getId());
            st.setString(2, part.getName());
            st.setString(3, part.getTelNo());
            st.setString(4, part.getRegion());
            st.executeUpdate();
        }
    }
}

```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

```
        } finally {
            st.close();
        }
    }
}

int getMaxId() {
    PreparedStatement st =
        dbConn.prepareStatement("select max(id) from participants");
    try {
        ResultSet rs = st.executeQuery();
        try {
            rs.next();
            return rs.getInt(1);
        } finally {
            rs.close();
        }
    } finally {
        st.close();
    }
}

}

class AddParticipantDialog extends JDialog {
    Participants participants;
    JTextField id;
    JTextField name;
    JTextField telNo;
    JTextField region;
    AddParticipantDialog(Participants participants) {
        this.participants = participants;
        setupComponents();
    }
    void setupComponents() {
        ...
    }
    void show() {
        showDefaultValues();
        setVisible(true);
    }
    void showDefaultValues() {
        Participant newPart = Participant.makeDefaultParticipant();
        newPart.setId(participants.getMaxId()+1);
        showParticipant(newPart);
    }
    void showParticipant(Participant part) {
        id.setText(new Integer(part.getId()).toString());
        name.setText(part.getName());
        telNo.setText(part.getTelNo());
        region.setText(part.getRegion());
    }
    Participant makeParticipant() throws ParticipantException {
        Participant part = new Participant(
            Integer.parseInt(id.getText()),
            name.getText(),
            telNo.getText(),
            region.getText());
        part.assertValid();
        return part;
    }
    void onOK() {
        try {
            participants.addParticipant(makeParticipant());
        }
    }
}
```

```
        dispose();
    } catch (Exception e) {
        JOptionPane.showMessageDialog(this, e.getMessage());
    }
}
}
```

SQLException is giving us away

In fact, there is still one problem in the code. Let's read the code the Participants class carefully:

```
class Participants {
    ...
    int getMaxId() {
        PreparedStatement st =
            dbConn.prepareStatement("select max(id) from participants");
        try {
            ResultSet rs = st.executeQuery();
            try {
                rs.next();
                return rs.getInt(1);
            } finally {
                rs.close();
            }
        } finally {
            st.close();
        }
    }
}
```

When calling methods like `prepareStatement`, `executeQuery`, `next` and `close`, JDBC may throw an `SQLException`. Because `getMaxId` by itself doesn't know how to handle this exception, we can only let the caller decide how to handle it. So, we need to change the code to:

```
class Participants {
    ...
    void getMaxId() throws SQLException {
        PreparedStatement st =
            dbConn.prepareStatement("select max(id) from participants");
        try {
            ResultSet rs = st.executeQuery();
            try {
                rs.next();
                return st.getInt(1);
            } finally {
                rs.close();
            }
        } finally {
            st.close();
        }
    }
}
```

Here comes the problem: When AddParticipantDialog calls the getMaxId method, it must be prepared to handle an SQLException (catch it or re-throw it), e.g.:

```
class AddParticipantDialog extends JDialog {
    ...
    void showDefaultValues() {
        Participant newPart = Participant.makeDefaultParticipant();
        try {
            newPart.setId(participants.getMaxId()+1);
        } catch (SQLException e) {
            ...
        }
        showParticipant(newPart);
    }
}
```

Now, because AddParticipantDialog needs to handle SQLException, it is obviously dealing with a database instead of a simple set of participants or an XML file. That is, SQLException is giving us away, exposing the fact that the Participants class uses a database. It forces all the code using the Participants class such as AddParticipantDialog to be used along with a database. All this client code cannot be reused in any environment where there is no database.

To solve this problem, when the Participants class encounters an error, it shouldn't throw an SQLException (which is related to a database). Instead, it should throw an exception related to a set of participants:

```
class ParticipantsException extends Exception {
    ParticipantsException(Throwable cause) {
        super(cause);
    }
}
class Participants {
    ...
    int getMaxId() throws ParticipantsException {
        try {
            PreparedStatement st =
                dbConn.prepareStatement("select max(id) from participants");
            try {
                ResultSet rs = st.executeQuery();
                try {
                    rs.next();
                    return rs.getInt(1);
                } finally {
                    rs.close();
                }
            } finally {
                st.close();
            }
        } catch (SQLException e) {
            //use e as the cause. It means the SQLException is the cause
            //for this ParticipantsException.
            throw new ParticipantsException(e);
        }
    }
}
```

```
}
```

Now, `AddParticipantDialog` only needs to handle a `ParticipantsException`, not an `SQLException`, e.g.:

```
class AddParticipantDialog extends JDialog {
    ...
    void showDefaultValues() {
        Participant newPart = Participant.makeDefaultParticipant();
        try {
            newPart.setId(participants.getMaxId()+1);
        } catch (ParticipantsException e) {
            ...
        }
        showParticipant(newPart);
    }
}
```

Divide the system into layers

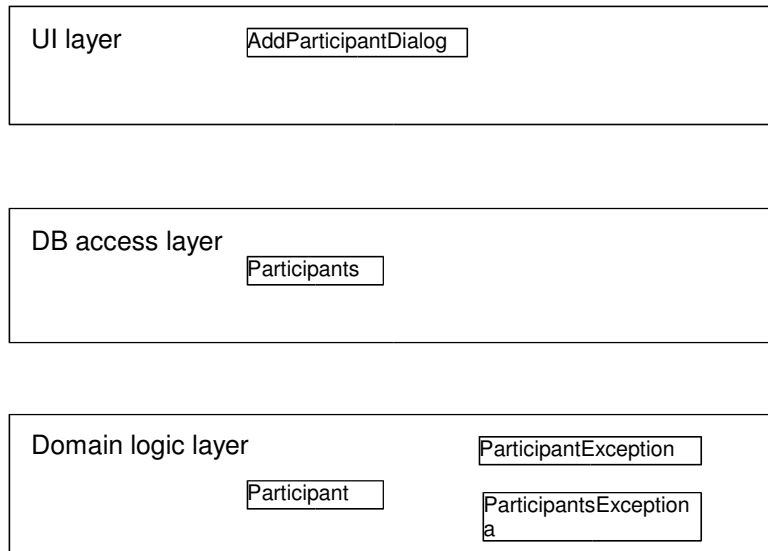
We can divide the code in the above conference system into three layers: "database access layer", "domain layer" and "user interface layer":

UI layer

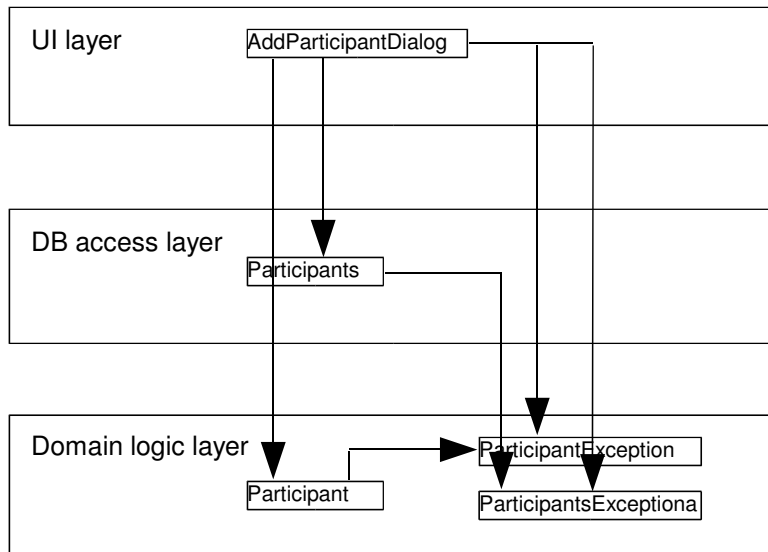
DB access layer

Domain logic layer

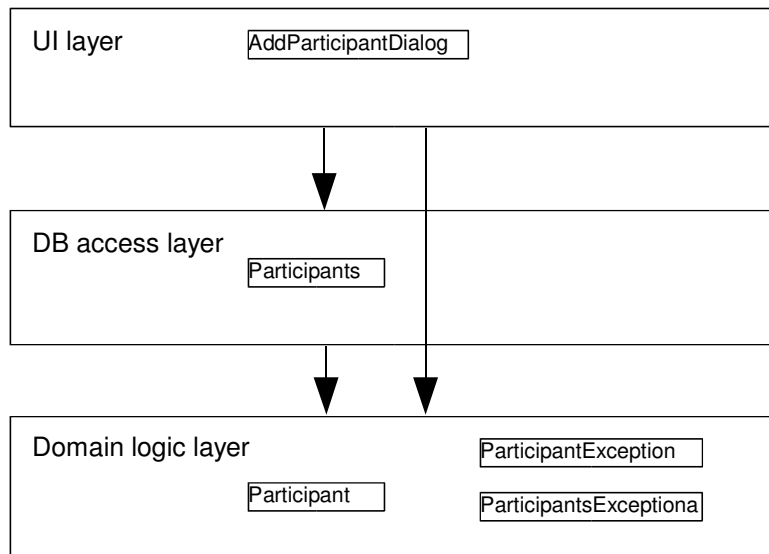
Because Participants accesses the database, we put it into the database access layer. Because Participant, ParticipantException, ParticipantsException deals with the domain logic, we put them into the domain logic layer. Because AddParticipantDialog interfaces with the user, we put it into the user interface layer:



If class A references class B, in the picture we draw an arrow from A to B:



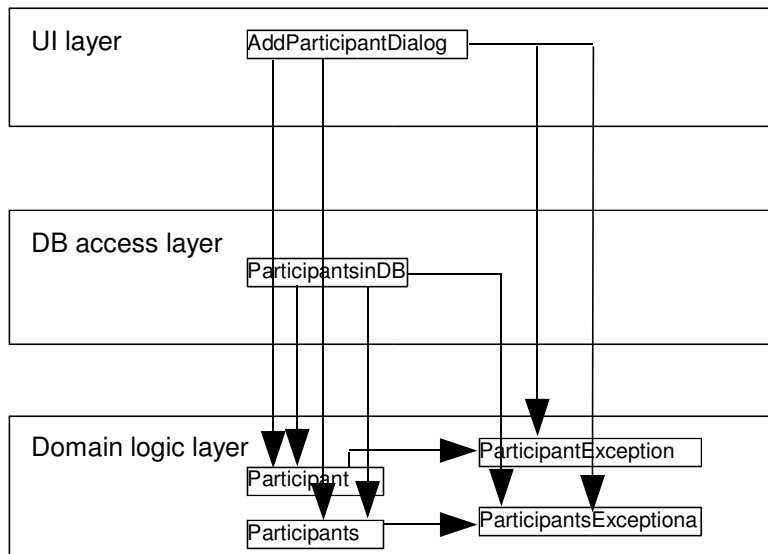
If we only show the references between the layers, the picture above becomes:



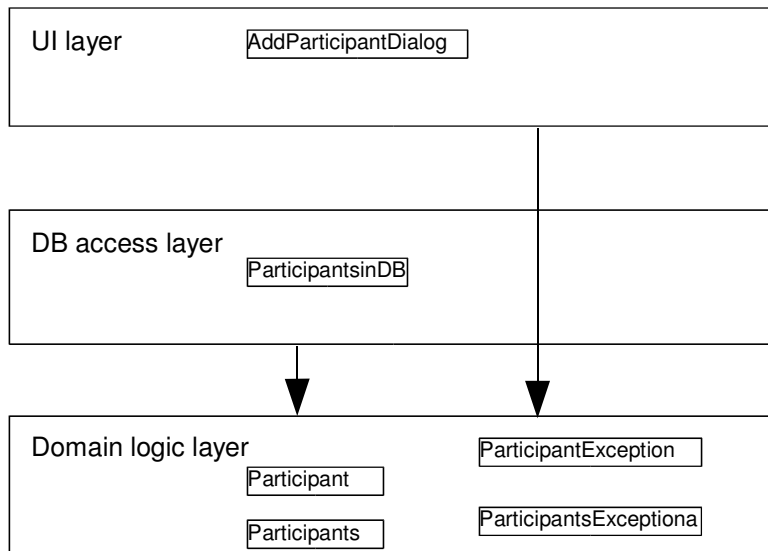
Note some points in the picture:

1. The domain logic layer is referenced by the other two layers but not the other way around. Therefore, this layer is easiest to reuse.
2. The database access layer references the domain logic layer but not the user interface layer. Therefore, no matter the system is text-based, GUI-based or web-based, this layer can be reused.
3. The user interface layer references the other two layers. Therefore, this layer is hardest to reuse.

The third point (the user interface layer uses the other two layers) needs some attention: Haven't we made `AddParticipantDialog` (user interface layer) unaware of the database? Why in the picture it is still referencing the database access layer? The problem comes from the `Participants` class. The interface of the `Participants` class has nothing to do with the database. Therefore, its interface belongs to the domain logic layer. However, the implementation of the `Participants` class is indeed about the database. Therefore, its implementation belongs to the database access layer. Therefore, putting the whole `Participants` class into the database access layer is not entirely correct. If we need to make it clearer, as mentioned before, we should make `Participants` an interface and put the current implementation of `Participants` into a `ParticipantsInDB` class. Then, the `Participants` interface will belong to the domain logic layer, while the `ParticipantsInDB` class will belong to the database access layer:



If we only show the references between the layers, the picture above becomes:



Now, the user interface layer only references the domain logic layer, but not the database access layer. Therefore, no matter the system uses a database, a text file or an XML file to store its data, this layer can be reused.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

Now, there are only two the references between the layers:

1. The user interface layer references the domain logic layer.
2. The database access layer references the domain logic layer.

Not all systems can be divided into three layers like this, with only two references between the layers. For example, the original AddParticipantDialog cannot be classified into any one of these layers. Therefore, the original system cannot be divided like this. This is possible only with systems that are well structured. Therefore, we should set this as our target (in particular, for larger systems).

Many things are user interfaces

Many things are user interfaces. It not only includes windows, buttons and etc., but also includes reports, servlet, jsp, text console and etc. Therefore, do not implement domain logic or database access in servlets (In particular, watch out for those servlets containing lots of code. They are the prime suspects); Nor should you call System.out.print in domain logic.

Other methods

Treating the database as a set of objects is only one way to hide the database. For the other methods, please see the references.

References

- <http://c2.com/cgi/wiki?ScatterSqlEverywhere>.
- <http://c2.com/cgi/wiki?ModelFirst>.
- <http://c2.com/cgi/wiki?ObjectRelationalMapping>.
- Scott Ambler's article at <http://www.agiledata.org/essays/mappingObjects.html> discusses how to map objects of complicated relationship onto the database.
- Martin Fowler's article at <http://www.martinfowler.com/articles/dblogic.html> discusses the respective advantages and disadvantages of using a programming language and SQL to express domain logic.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agileskills.org

- Martin Fowler has put forth several patterns to access a database (Table Data Gateway, Row Data Gateway, Active Record and Data Mapper): <http://www.martinfowler.com/eaCatalog>.
- Robert C. Martin's article at <http://www.objectmentor.com/resources/articles/Proxy.pdf> describes how to use patterns like Proxy and Stairway to Heaven to encapsulate the access to a database.



Chapter exercises

Introduction

Some of the problems will test you on the topics in the previous chapters.

Problems

1. Implement the `updateParticipant` and `getAllParticipantsById` methods in the `Participants` class:

```
class Participants {
    ...
    void updateParticipant(Participant part) {
        //write code here.
    }
    ParticipantIterator getAllParticipantsById() {
        //write code here.
    }
}
```

2. This application is about restaurants. An order has an order ID, a restaurant ID and a customer ID and contains some order items. Each order item contains the food ID, the quantity and unit price. You have created the tables and classes below.

```
create table Orders (
    orderId varchar(20) primary key,
    customerId varchar(20) not null,
    restId varchar(20) not null
);
create table OrderItems (
    orderId varchar(20),
    itemId int,
    foodId varchar(20) not null,
    quantity int not null,
    unitPrice float not null,
    primary key(orderId, itemId)
);
class Order {
    String orderId;
    String customerId;
    String restId;
    OrderItem items[];
}
class OrderItem {
    String foodId;
    int quantity;
    double unitPrice;
}
```

Your tasks are:

1. Create an interface for accessing the orders while hiding the database.
 2. Create a class to implement that interface and show how to implement its method for adding an order to the database.
 3. Identify which layers they belong to.
3. In the application above, you would like to keep record of the orders that have been delivered and if so, the delivery time. As you have attended the CPTTM Object Oriented Design course, you know it is good to keep the Order class slim, so you decide to create a new class, while keeping the Order class unchanged. So you have written the code below:

```
class OrderDelivery {
    String orderId;
    Date deliveryTime;
}
interface OrderDeliveries {
    boolean isDelivered(String orderId);
    Date getDeliveryTime(String orderId);
    void markAsDelivered(String orderId, Date deliveryTime);
}
```

To store this information in the database, you ask the database administrator. He says that it is most efficient to just add a datetime field to the Orders table. If that field is NULL, it means the order has not been delivered:

```
create table Orders (
    orderId varchar(20) primary key,
    customerId varchar(20) not null,
    restId varchar(20) not null,
    deliveryTime datetime
);
```

Your tasks are:

1. Create a class to implement OrderDeliveries and show how to implement its method for the isDelivered and markAsDelivered methods.
 2. Determine if you need to modify your code done in the previous question. If so, where?
 3. Determine what happens to the order delivery if an order is deleted?
4. This program below implements a game called "Hangman". The game is played like this: the computer "comes up with" a secret such as "banana". The task of the player is to try to find out this secret. Every turn the player can input one english letter such as "a", then the computer will show the "a" letters in the secret (if any), while the other letters will be shown

as a dash. For example, in this case, the computer will show "-a-a-a". In the next turn if the player inputs "b", the computer will show "ba-a-a". In the next turn if the player inputs "c", the computer will still show "ba-a-a" because there is no "c" in the secret. The player has at most 7 turns. If he can find out the secret within 7 turns, he wins. Otherwise he loses. If the player inputs say "b" again, the computer will tell him that it has been guessed before and this guess is not counted (not included in the 7 turns).

Your tasks are:

1. Point out and remove the problems in the code below.
2. Divide your revised code into appropriate layers.

```
class Hangman {
    String secret = "banana";
    String guessedChars = "";
    static public void main(String args[]) {
        new Hangman();
    }
    Hangman() {
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));
        for (int k = 0; k < 7;) { //can guess at most 7 times
            String s = ""; //partially found secret
            for (int i=0; i<secret.length(); i++) {
                char ch = secret.charAt(i);
                if (guessedChars.indexOf(ch)<0) //has it been guessed?
                    ch = '-'; //no, hide it. just show a dash.
                s = s+ch;
            }
            System.out.println("Secret: "+s);
            System.out.print("Guess letter: ");
            char ch = br.readLine().charAt(0); //read just one char
            if (guessedChars.indexOf(ch)>=0) { //already guessed?
                System.out.println("You have guessed this char!");
                continue;
            }
            int n = numberOfFoundChars();
            guessedChars = guessedChars+ch;
            int m = numberOfFoundChars();
            if (m>n) {
                System.out.println("Success, you have found letter "+ch);
                System.out.println("Letters found: "+m);
            }
            if (m==secret.length()) {
                System.out.println("You won!");
                return;
            }
            k++;
        }
        System.out.println("You lost!");
    }
    int numberOfFoundChars() {
        int n = 0;
        for (int i=0; i< secret.length(); i++) {
            char ch = secret.charAt(i);
```

```

        if (guessedChars.indexOf(ch)>=0)
            n++;
    }
    return n;
}
}

```

5. This is a web-based application concerned with training courses. A user can choose a subject area ("IT", "Management", "Language", "Apparel") in a form and click submit. Then the application will display all the courses in that subject area. The servlet doing that is shown below.

```

public class ShowCoursesServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML><TITLE>Course listing</TITLE><BODY>");
        Connection dbConn = ...;
        PreparedStatement st =
            dbConn.prepareStatement("select * from Courses where subjArea=?");
        try {
            st.setString(1, request.getParameter("SubjArea"));
            ResultSet rs = st.executeQuery();
            try {
                out.println("<TABLE>");
                while (rs.next()) {
                    out.println("<TR>");
                    out.println("<TD>");
                    out.println(rs.getString(1)); //course code
                    out.println("</TD>");
                    out.println("<TD>");
                    out.println(rs.getString(2)); //course name
                    out.println("</TD>");
                    out.println("<TD>");
                    out.println(""+rs.getInt(3)); //course fee in MOP
                    out.println("</TD>");
                    out.println("</TR>");
                }
                out.println("</TABLE>");
            } finally {
                rs.close();
            }
        } finally {
            st.close();
        }
        out.println("</BODY></HTML>");
    }
}

```

Your tasks are:

1. Point out and remove the problems in the code.
2. Divide your revised code into appropriate layers.

Hints

1. To implement `getAllParticipantsById`, you need to create a class implementing `ParticipantIterator` that reads the data from a `ResultSet`.
2. You should have methods to add, delete or update an order and retrieve a selection of orders. For example, in the add method, you need to add one record to the `Orders` table and add multiple records to the `OrderItems` table.
3. No hint for this one.
4. UI is mixed with the domain logic. Extract the domain logic into a class like `Hangman` and rename the original one as `HangmanApp`. Make sure that `HangmanApp` uses `Hangman` but not the reverse.
5. No hint for this one.

Sample solutions

1. Implement the `updateParticipant` and `getAllParticipantsById` methods in the `Participants` class:

```
class Participants {
    ...
    void updateParticipant(Participant part) {
        //write code here.
    }
    ParticipantIterator getAllParticipantsById() {
        //write code here.
    }
}
```

The code may be like:

```
class Participants {
    ...
    void updateParticipant(Participant part) {
        PreparedStatement st =
            dbConn.prepareStatement(
                "update participants set name=?, telNo=?, region=? where id=?");
        try {
            st.setString(1, part.getName());
            st.setString(2, part.getTelNo());
            st.setString(3, part.getRegion());
            st.setInt(4, part.getId());
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    ParticipantIterator getAllParticipantsById() {
        PreparedStatement st =
            dbConn.prepareStatement(
                "select * from participants order by id");
        return new ParticipantResultSetIterator(st);
    }
}

class ParticipantResultSetIterator implements ParticipantIterator {
    PreparedStatement st;
    ResultSet rs;
    ParticipantResultSetIterator(PreparedStatement st) {
        this.st = st;
        this.rs=st.executeQuery();
    }
    boolean next() {
        return rs.next();
    }
    Participant getParticipant() {
        new Participant(
            rs.getInt(1),
            rs.getString(2),
            rs.getString(3),
            rs.getString(4));
    }
}
```

```

void finalize() {
    rs.close();
    st.close();
}
}

```

2. This application is about restaurants. An order has an order ID, a restaurant ID and a customer ID and contains some order items. Each order item contains the food ID, the quantity and unit price. You have created the tables and classes below.

```

create table Orders (
    orderId varchar(20) primary key,
    customerId varchar(20) not null,
    restId varchar(20) not null
);
create table OrderItems (
    orderId varchar(20),
    itemId int,
    foodId varchar(20) not null,
    quantity int not null,
    unitPrice float not null,
    primary key(orderId, itemId)
);
class Order {
    String orderId;
    String customerId;
    String restId;
    OrderItem items[];
}
class OrderItem {
    String foodId;
    int quantity;
    double unitPrice;
}

```

Your tasks are:

1. Create an interface for accessing the orders while hiding the database.

```

interface Orders {
    void addOrder(Order order);
    void deleteOrder(String orderId);
    void updateOrder(Order order);
    OrderIterator getOrdersById();
}

```

2. Create a class to implement that interface and show how to implement its method for adding an order to the database.

```

class OrdersInDB implements Orders {
    void addOrder(Order order) {
        PreparedStatement st =
            dbConn.prepareStatement("insert into from Orders values(?,?,?)");
        try {
            st.setString(1, order.getId());
            st.setString(2, order.getCustomerId());

```

```

        st.setString(3, order.getRestId());
        st.executeUpdate();
    } finally {
        st.close();
    }
}
st = dbConn.prepareStatement(
    "insert into from OrderItems values(?,?,?,?,?)");
try {
    for (int i = 0; i < order.items.length; i++) {
        OrderItem orderItem = order.items[i];
        st.setString(1, order.getId());
        st.setInt(2, i);
        st.setString(3, orderItem.getFoodId());
        st.setInt(4, orderItem.getQuantity());
        st.setDouble(5, orderItem.getUnitPrice());
        st.executeUpdate();
    }
} finally {
    st.close();
}
}
}
}

```

3. Identify which layers they belong to.

Domain: Orders.

Database access: OrdersInDB.

3. In the application above, you would like to keep record of the orders that have been delivered and if so, the delivery time. As you have attended the CPTTM Object Oriented Design course, you know it is good to keep the Order class slim, so you decide to create a new class, while keeping the Order class unchanged. So you have written the code below:

```

class OrderDelivery {
    String orderId;
    Date deliveryTime;
}
interface OrderDeliveries {
    boolean isDelivered(String orderId);
    Date getDeliveryTime(String orderId);
    void markAsDelivered(String orderId, Date deliveryTime);
}

```

To store this information in the database, you ask the database administrator. He says that it is most efficient to just add a datetime field to the Orders table. If that field is NULL, it means the order has not been delivered:

```

create table Orders (
    orderId varchar(20) primary key,
    customerId varchar(20) not null,
    restId varchar(20) not null,
    deliveryTime datetime
);

```

Your tasks are:

1. Create a class to implement OrderDeliveries and show how to implement its method for the isDelivered and markAsDelivered methods.

```
class OrderDeliveriesInDB implements OrderDeliveries {
    boolean isDelivered(String orderId) {
        PreparedStatement st = dbConn.prepareStatement(
            "select deliveryTime from Orders where orderId=?");
        try {
            st.setString(1, orderId);
            ResultSet rs=st.executeQuery();
            try {
                rs.next();
                return rs.getDate(1)!=null;
            } finally {
                rs.close();
            }
        } finally {
            st.close();
        }
    }
    void markAsDelivered(String orderId, Date deliveryTime) {
        PreparedStatement st = dbConn.prepareStatement(
            "update Orders set deliveryTime=? where orderId=?");
        try {
            st.setDate(1, new java.sql.Date(deliveryTime.getTime()));
            st.setString(2, orderId);
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}
```

2. Determine if you need to modify your code done in the previous question. If so, where?

When adding an order record to the Orders table, there is one more field (deliveryTime). It must be NULL to indicate that the order has not been delivered. Some DBMS may allow you to use the previous code unchanged, while still having this effect. But to be safe than sorry, do it explicitly:

```
class OrdersInDB implements Orders {
    void addOrder(Order order) {
        PreparedStatement st =
            dbConn.prepareStatement("insert into from Orders values(?,?,?,?)");
        try {
            st.setString(1, order.getId());
            st.setString(2, order.getCustomerId());
            st.setString(3, order.getRestId());
            st.setNull(4, java.sql.Types.TIME);
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}
```

```

    }
    ...
}

```

3. Determine what happens to the order delivery if an order is deleted?

Then the order delivery is also deleted. This sounds fine.

4. This program below implements a game called "Hangman". The game is played like this: the computer "comes up with" a secret such as "banana". The task of the player is to try to find out this secret. Every turn the player can input one English letter such as "a", then the computer will show the "a" letters in the secret (if any), while the other letters will be shown as a dash. For example, in this case, the computer will show "-a-a-a". In the next turn if the player inputs "b", the computer will show "ba-a-a". In the next turn if the player inputs "c", the computer will still show "ba-a-a" because there is no "c" in the secret. The player has at most 7 turns. If he can find out the secret within 7 turns, he wins. Otherwise he loses. If the player inputs say "b" again, the computer will tell him that it has been guessed before and this guess is not counted (not included in the 7 turns).

Your tasks are:

1. Point out and remove the problems in the code below.
2. Divide your revised code into appropriate layers.

```

class Hangman {
    String secret = "banana";
    String guessedChars = "";
    static public void main(String args[] ) {
        new Hangman();
    }
    Hangman() {
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));
        for (int k = 0; k < 7; ) { //can guess at most 7 times
            String s = ""; //partially found secret
            for (int i=0; i<secret.length(); i++) {
                char ch = secret.charAt(i);
                if (guessedChars.indexOf(ch)<0) //has it been guessed?
                    ch = '-'; //no, hide it. just show a dash.
                s = s+ch;
            }
            System.out.println("Secret: "+s);
            System.out.print("Guess letter: ");
            char ch = br.readLine().charAt(0); //read just one char
            if (guessedChars.indexOf(ch)>=0) { //already guessed?
                System.out.println("You have guessed this char!");
                continue;
            }
            int n = numberOfFoundChars();
            guessedChars = guessedChars+ch;
            int m = numberOfFoundChars();
            if (m>n) {

```

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from www.agilekills.org

```
        System.out.println("Success, you have found letter "+ch);
        System.out.println("Letters found: "+m);
    }
    if (m==secret.length()) {
        System.out.println("You won!");
        return;
    }
    k++;
}
System.out.println("You lost!");
}
int numberOfFoundChars() {
    int n = 0;
    for (int i=0; i< secret.length(); i++) {
        char ch = secret.charAt(i);
        if (guessedChars.indexOf(ch)>=0)
            n++;
    }
    return n;
}
}
```

The code mixes model logic with UI. The comments in the code can be removed by making the code as clear as the comments.

We can extract the domain logic into:

```
class Hangman {
    final static int MAXNOGUESSES = 7;
    String secret = "banana";
    String guessedChars = "";
    boolean reachMaxNoGuesses() {
        return getNoGuessedChars() == MAXNOGUESSES;
    }
    int getNoGuessedChars() {
        return guessedChars.length();
    }
    boolean hasBeenGuessed(char ch) {
        return guessedChars.indexOf(ch) >= 0;
    }
    String getPartiallyFoundSecret() {
        String partiallyFoundSecret = "";
        for (int i=0; i< secretLength(); i++) {
            char ch = secret.charAt(i);
            char chToShow = hasBeenGuessed(ch) ? ch : '-';
            partiallyFoundSecret = partiallyFoundSecret+chToShow;
        }
        return partiallyFoundSecret;
    }
    boolean guess(char ch) {
        int n = numberOfFoundChars();
        guessedChars = guessedChars+ch;
        int m = numberOfFoundChars();
        return m>n;
    }
    boolean isSecretFound() {
        return numberOfFoundChars() == secretLength();
    }
}
```

```

int numberOfFoundChars() {
    int n = 0;
    for (int i=0; i< secretLength(); i++) {
        char ch = secret.charAt(i);
        if (guessedChars.indexOf(ch)>=0)
            n++;
    }
    return n;
}
int secretLength() {
    return secret.length();
}
}

```

The UI will use the domain logic:

```

class HangmanApp {
    static public void main(String args[]) {
        new HangmanApp();
    }
    HangmanApp() {
        Hangman hangman = new Hangman();
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        while (!hangman.reachMaxNoGuesses()) {
            System.out.println("Secret: "+hangman.getPartiallyFoundSecret());
            System.out.print("Guess letter: ");
            char ch = readOneChar(br);
            if (hangman.hasBeenGuessed(ch)) {
                System.out.println("You have guessed this char!");
                continue;
            }
            if (hangman.guess(ch)) {
                System.out.println("Success, you have found letter "+ch);
                System.out.println("Letters found: "+
                    hangman.numberOfFoundChars());
            }
            if (hangman.isSecretFound()) {
                System.out.println("You won!");
                return;
            }
        }
        System.out.println("You lost!");
    }
    char readOneChar(BufferedReader br) {
        return br.readLine().charAt(0);
    }
}

```

5. This is a web-based application concerned with training courses. A user can choose a subject area ("IT", "Management", "Language", "Apparel") in a form and click submit. Then the application will display all the courses in that subject area. The servlet doing that is shown below.

```

public class ShowCoursesServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) {
        response.setContentType("text/html");
    }
}

```

```
PrintWriter out = response.getWriter();
out.println("<HTML><TITLE>Course listing</TITLE><BODY>");
Connection dbConn = ...;
PreparedStatement st =
    dbConn.prepareStatement("select * from Courses where subjArea=?");
try {
    st.setString(1, request.getParameter("SubjArea"));
    ResultSet rs = st.executeQuery();
    try {
        out.println("<TABLE>");
        while (rs.next()) {
            out.println("<TR>");
            out.println("<TD>");
            out.println(rs.getString(1)); //course code
            out.println("</TD>");
            out.println("<TD>");
            out.println(rs.getString(2)); //course name
            out.println("</TD>");
            out.println("<TD>");
            out.println(""+rs.getInt(3)); //course fee in MOP
            out.println("</TD>");
            out.println("</TR>");
        }
        out.println("</TABLE>");
    } finally {
        rs.close();
    }
} finally {
    st.close();
}
out.println("</BODY></HTML>");
}
```

Your tasks are:

1. Point out and remove the problems in the code.
2. Divide your revised code into appropriate layers.

The code mixes model logic with database and UI. Remember that servlet is UI. We can extract the database access into:

```
interface Courses {
    Course[] getCoursesInSubject(String subjArea);
}
class CoursesInDB implements Courses {
    Course[] getCoursesInSubject(String subjArea) {
        Connection dbConn = ...;
        PreparedStatement st =
            dbConn.prepareStatement("select * from Courses where subjArea=?");
        try {
            st.setString(1, subjArea);
            ResultSet rs = st.executeQuery();
            try {
                Course courses[];
```

```

        while (rs.next()) {
            Course course = new Course(
                rs.getString(1),
                rs.getString(2),
                rs.getInt(3));
            add course to courses;
        }
        return courses;
    } finally {
        rs.close();
    }
} finally {
    st.close();
}
}
}

```

The UI will use the domain logic (but not the DB) as:

```

public class ShowCoursesServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response) {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML><TITLE>Course listing</TITLE><BODY>");
        Courses courses =
            (Courses) getServletContext().getAttribute("Courses");
        Course coursesInSubject[] =
            courses.getCoursesInSubject(request.getParameter("SubjArea"));
        out.println("<TABLE>");
        for (int i = 0; i < coursesInSubject.length; i++) {
            Course course = coursesInSubject[i];
            out.println("<TR>");
            out.println("<TD>");
            out.println(course.getCourseCode());
            out.println("</TD>");
            out.println("<TD>");
            out.println(course.getCourseName());
            out.println("</TD>");
            out.println("<TD>");
            out.println(course.getCourseFeeInMOP());
            out.println("</TD>");
            out.println("</TR>");
        }
        out.println("</TABLE>");
        out.println("</BODY></HTML>");
    }
}

```