



CHAPTER 4

Keeping Code Fit



Example

This is a conference management system. It is used to manage the information of the conference participants. At the beginning, we only need to record the ID (assigned by the conference organizer), name, telephone and address of each participant. For this, we write the following code:

```
class Participant {
    String id;
    String name;
    String telNo;
    String address;
}
class ConferenceSystem {
    Participant participants[];
}
```

Later, there comes a new requirement: We need to record whether a participant requests the conference organizer to book a hotel for him. If yes, we also need to record the name of the hotel he selected, the check-in date, check-out date, room type (single or double). For this, we expand the above Participant class:

```
class Participant {
    String id;
    String name;
    String telNo;
    String address;
    boolean bookHotelForHim;
    String hotelName;
    Date checkInDate;
    Date checkOutDate;
    boolean isSingleRoom;
    void setHotelBooking(String hotelName, Date checkInDate, ...) {
        ...
    }
}
```

Later, there comes yet another new requirement: Record which seminars each participant is going to attend. For each seminar that he is going to attend, we need to record the registration date and whether he needs a set of simultaneous interpretation device. For this, we expand the Participant class again:

```
class Participant {
```

```
String id;
String name;
String telNo;
String address;
boolean bookHotelForHim;
String hotelName;
Date checkInDate;
Date checkOutDate;
boolean isSingleRoom;
String idOfSeminarsRegistered[];
Date seminarRegistrationDates[];
boolean needSIDeviceForEachSeminar[];
void setHotelBooking(String hotelName, Date checkInDate, ...) {
    ...
}
void registerForSeminar(String seminarId, Date regDate, boolean needSIDevice) {
    //add seminarId to idOfSeminarsRegistered.
    //add regDate to seminarRegistrationDates.
    //add needSIDevice to needSIDeviceForEachSeminar.
}
boolean isRegisteredForSeminar(String seminarId) {
    ...
}
Date getSeminarRegistrationDate(String seminarId) {
    ...
}
boolean needSIDeviceForSeminar(String seminarId) {
    ...
}
String [] getAllSeminarsRegistered() {
    return idOfSeminarsRegistered;
}
}
```

The code is getting bloated

Note that this is already the second time that we expand the Participant class. Every time we expand it, it includes more code (instance variables and methods) and more functionality. Originally it had only 4 instance variables. Now it has 12! In addition, the number of concepts it deals with has increased greatly. Originally it only dealt with the basic information of a participant (name, address and etc.), now it also deals with the concepts of hotel, hotel booking, seminar, simultaneous interpretation and etc. If in the future more requirements come, we will expand the Participant class again, then how complicated and bloated it will become!

How to trim the Participant class? Or how to keep it fit from day one? Before answering these two questions, let's consider another question that must be answered first: Given a class, how to check whether it needs trimming?

How to check if a class needs trimming

To check if a class needs trimming, a subjective method is: After reading its code, do we feel that it is "too long" (a common code smell), "too complicated" or involves "too many" concepts? If yes, it needs trimming.

A simpler and objective method is, when we find that we are already expanding a class for the second or third time, we consider it needs trimming. This is a "lazy", "passive" method, but it is effective.

Now let's see how to trim the Participant class.

Extract the functionality about hotel booking

First we consider how to extract the functionality about hotel booking. A possible method is:

```
class Participant {
    String id;
    String name;
    String telNo;
    String address;
}
class HotelBooking {
    String participantId;
    String hotelName;
    Date checkInDate;
    Date checkOutDate;
    boolean isSingleRoom;
}
class HotelBookings {
    HotelBooking hotelBookings[];
    void addBooking(HotelBooking booking) {
        ...
    }
}
class ConferenceSystem {
    Participant participants[];
    HotelBookings hotelBookings;
}
```

Now, the Participant class knows absolutely nothing about hotel booking. Of course, we don't have to use an array to store the hotel bookings. For example, we may as well use a Map:

```
class Participant {
    String id;
    String name;
    String telNo;
    String address;
}
```

```

class HotelBooking {
    String participantId;
    String hotelName;
    Date checkInDate;
    Date checkOutDate;
    boolean isSingleRoom;
}
class HotelBookings {
    HashMap mapFromPartIdToHotelBooking;
    //Must provide participant's ID.
    void addBooking(String participantId, HotelBooking booking) {
        ...
    }
}
class ConferenceSystem {
    Participant participants[];
    HotelBookings hotelBookings;
}

```

The advantage of this method is that we can quickly find the hotel booking of a participant. Another possible method is:

```

class Participant {
    String id;
    String name;
    String telNo;
    String address;
    HotelBooking hotelBooking;
}
class HotelBooking {
    String hotelName;
    Date checkInDate;
    Date checkOutDate;
    boolean isSingleRoom;
}
class ConferenceSystem {
    Participant participants[];
}

```

Note that in this method the Participant class still has the concept of hotel booking. Just that the actual functionality has been extracted into the HotelBooking class. It is more straight forward to find the hotel booking of a participant. The cost we pay is that Participant still has the concept of hotel booking.

Of course, in addition to the methods above, there are many other possible methods.

Extract the functionality about seminar

Now we consider how to extract the functionality about seminar. A possible method is:

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

```
class Participant {
    String id;
    String name;
    String telNo;
    String address;
}
class SeminarRegistration {
    String participantId;
    String seminarId;
    Date registrationDate;
    boolean needSIDevice;
}
class SeminarRegistry {
    SeminarRegistration registrations[];
    void registerForSeminar(SeminarRegistration registration) {
        //add registration to registrations.
    }
    boolean isRegisteredForSeminar(String participantId, String seminarId) {
        ...
    }
    Date getSeminarRegistrationDate(String participantId, String seminarId) {
        ...
    }
    boolean needSIDeviceForSeminar(String participantId, String seminarId) {
        ...
    }
    SeminarRegistration[] getAllRegistrations(String participantId) {
        ...
    }
}
class ConferenceSystem {
    Participant participants[];
    SeminarRegistry seminarRegistry;
}
```

Of course, in addition to the methods above, there are many other possible methods.

The improved code

The improved code is shown below:

```
class Participant {
    String id;
    String name;
    String telNo;
    String address;
}
class HotelBooking {
    String participantId;
    String hotelName;
    Date checkInDate;
    Date checkOutDate;
    boolean isSingleRoom;
```

```
}
class HotelBookings {
    HotelBooking hotelBookings[];
    void addBooking(HotelBooking booking) {
        ...
    }
}
class SeminarRegistration {
    String participantId;
    String seminarId;
    Date registrationDate;
    boolean needSIDevice;
}
class SeminarRegistry {
    SeminarRegistration registrations[];
    void registerForSeminar(SeminarRegistration registration) {
        //add registration to registrations.
    }
    boolean isRegistered (String participantId, String seminarId) {
        ...
    }
    Date getRegistrationDate(String participantId, String seminarId) {
        ...
    }
    boolean needSIDevice(String participantId, String seminarId) {
        ...
    }
    SeminarRegistration[] getAllRegistrations(String participantId) {
        ...
    }
}
class ConferenceSystem {
    Participant participants[];
    HotelBookings hotelBookings;
    SeminarRegistry seminarRegistry;
}
```

References

The Single Responsibility Principle states: We should make a class that will change for one reason only. When a class includes many different types of functionality, we are obviously violating the Single Responsibility Principle. For more details, please see:

- <http://www.objectmentor.com/resources/articles/srp>.
- <http://c2.com/cgi/wiki?OneResponsibilityRule>.



Chapter exercises

Introduction

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

Problems

1. The code below still contains duplicate code: we create a prepared statement, set it up, execute it and then close it at several places. If you need to remove this duplication at any cost, how do you do it?

```
class ParticipantsInDB {
    Connection conn;
    final String tableName="participants";
    void addParticipant(Participant part){
        PreparedStatement st = conn.prepareStatement("INSERT INTO "+tableName+"
VALUES (?, ?, ?, ?, ?)");
        try {
            st.setString(1, part.getId());
            st.setString(2, part.getEFirstName());
            st.setString(3, part.getELastName());
            //...
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    void deleteAllParticipants(){
        PreparedStatement st = conn.prepareStatement("DELETE FROM "+tableName);
        try {
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    void deleteParticipant(String participantId){
        PreparedStatement st = conn.prepareStatement("DELETE FROM "+tableName+"
WHERE id=?");
        try {
            st.setString(1, participantId);
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}
```

2. This application lets the network administrators document their server configurations. There are several types of servers as shown below. Design the code that implements a Swing GUI allowing the user to edit a server object. It must allow the user to change the

type of the server. It must use a CardLayout to display the components just right for the type of the server.

```
class Server {
    String id;
    String CPUModel;
}
class DNSServer extends Server {
    String domainName;
}
class WINSServer extends Server {
    String replicationPartner;
    int replicationInterval;
}
class DomainController extends Server {
    boolean remainNT4Compatible;
}
```

3. Point out and remove the code smells in the code below (this example is adapted from the one contributed by Eugen):

```
public static String executeUpdateSQLStmt(String sqlStmt, String mode) {
    // mode value --> ADD, MOD, DEL
    String outResult = "";
    final String connName = "tmp";
    DBConnectionManager connMgr = DBConnectionManager.getInstance();
    Connection conP = connMgr.getConnection(connName);
    Statement stmt = conP.createStatement();
    int upd = stmt.executeUpdate(sqlStmt);
    switch(upd) {
        case 1:
            if (mode.equalsIgnoreCase("ADD")) { outResult = "SUC Add"; }
            else if (mode.equalsIgnoreCase("MOD")) { outResult = "SUC Mod"; }
            else if (mode.equalsIgnoreCase("DEL")) { outResult = "SUC Del"; }
            break;
        case 0:
            if (mode.equalsIgnoreCase("ADD")) { outResult = "Err Add"; }
            else if (mode.equalsIgnoreCase("MOD")) { outResult = "Err Mod"; }
            else if (mode.equalsIgnoreCase("DEL")) { outResult = "Err Del"; }
            break;
        default:
            if (mode.equalsIgnoreCase("ADD")) { outResult = "Err Add"; }
            else if (mode.equalsIgnoreCase("MOD")) { outResult = "Err Mod"; }
            else if (mode.equalsIgnoreCase("DEL")) { outResult = "Err Del"; }
            break;
    }
    stmt.close();
    conP.close();
    return outResult;
}
```

4. Point out and remove the code smells in the code below (this example is adapted from the one contributed by Carol):

```
class Account {
    final public int SAVING=0;
    final public int CHEQUE=1;
```

```
final public int FIXED=2; //Portuguese currency
private int accountType;
private double balance;
public double getInterestRate(...) { // Some method;
    ...
}
public Account(int accountType) {
    this.accountType=accountType;
}
public double calcInterest() {
    switch (accountType) {
        case SAVING:
            return balance*getInterestRate();
        case CHEQUE:
            return 0;
        case FIXED:
            return balance*(getInterestRate()+0.02);
    }
}
}
```

5. Point out and remove the code smells in the code below (this example is contributed by Antonio):

```
class Department{
    final public int Account =0;
    final public int Marketing = 1;
    final public int CustomerServices = 2;
    protected int departmentCode;
    public Department(int departmentCode){
        this.departmentCode = departmentCode;
    }
    public String getDepartmentName(){
        switch (departmentCode){
            case Account:
                return "Account";
            case Marketing:
                return "Marketing";
            case CustomerServices:
                return "Customer Services";
        }
    }
}
```

6. Point out and remove the code smells in the code below (this example is contributed by Carita):

```
class NormalPayment {
    int units;
    double rate;
    final double TAX_RATE = 0.1;
    double getBillableAmount() {
        double baseAmt = units * rate;
        double tax = baseAmt * TAX_RATE;
        return baseAmt + tax;
    }
}
class PaymentForSeniorCitizen {
```

```

int units;
double rate;
final double TAX_RATE = 0.1;
double getBillableAmount() {
    double baseAmt = units * rate * 0.8;
    double tax = baseAmt * (TAX_RATE - 0.5) ;
    return baseAmt + tax;
}
}

```

7. Point out and remove the code smells in the code below (this example is contributed by Malaquias):

```

class PianoKey {
    final static int key0 = 0;
    final static int key1 = 1;
    final static int key2 = 2;
    int keyNumber;
    public void playSound() {
        if (keyNumber == 0) {
            //play the frequency for key0
        }
        else if (keyNumber == 1) {
            //play the frequency for key1
        }
        else if (keyNumber == 2) {
            //play the frequency for key2
        }
    }
}
class Piano {
    Vector rythmn;
    public void play() {
        for(int i=0;i<rythmn.size();i++) {
            rythmn.elementAt(i).playSound();
        }
    }
}
}

```

8. Point out and remove the code smells in the code below (this example is contributed by Frankie):

```

class Account {
    final static int LEVEL_USER = 1;
    final static int LEVEL_ADMIN = 2;
    int accountLevel;
    Date expiredDate; // for user account only
    boolean hasLogin; // for admin account only
}
class ERPApp {
    public boolean checkLoginIssue(Account account) {
        if (account.getLevel() == Account.LEVEL_USER) {
            // Check the account expired date
            Date now = new Date();
            if (account.getExpiredDate().before(now))
                return false;
            return true;
        }
    }
}

```

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

```
else if (account.getLevel() == Account.LEVEL_ADMIN) {
    // No expired date for admin account
    // Check multilogin
    if (account.hasLogin())
        return false;
    return true;
}
return false;
}
```

9. Point out and remove the code smells in the code below (this example is adapted from the one contributed by YK):

```
class Form1 extends JDialog {
    JComboBox comboBoxReportType;
    Form1() {
        comboBoxReportType = new JComboBox();
        comboBoxReportType.addItem("r1");
        comboBoxReportType.addItem("r2");
        ...
        comboBoxReportType.addItem("r31c");
    }
    void processReport1() {
        //print some fancy report...
    }
    void processReport2() {
        //print another totally different fancy report...
    }
    ...
    void processReport31c() {
        //print yet another totally different fancy report...
    }
    void printReport(String repNo) {
        if (repNo.equals("r1"))
            processReport1();
        else if (repNo.equals("r2"))
            processReport2();
        ...
        else if (repNo.equals("r31c"))
            processReport31c();
    }
    void onPrintClick() {
        printReport((String) comboBoxReportType.getSelectedItem());
    }
}
```

10. This application is about restaurants. Initially we created a Restaurant class to represent a restaurant account and includes information such as its name, access password, tel and fax number, address. The class is like:

```
class Restaurant {
    String name;
    String password;
    String telNo;
    String faxNo;
    String address;
}
```

Later, the following requirements are added in sequence:

1. After initial registration, the restaurant account is assigned an activation code by the system. Only after the user enters the activation code, the account will become activated. Until then, the account is inactive and login is not allowed.
2. If the user would like to change the fax number of the account, the new fax number will not take effect immediately (the existing fax number will remain in effect). Instead, the account is assigned an activation code by the system. Only after the user enters the activation code, the new fax number will take effect.
3. A restaurant can be marked as in a certain category (e.g., Chinese restaurant, Portuguese restaurant and etc.). A category is identified by a category ID.
4. The user can input the holidays for the restaurant.
5. The user can input the business hours for the restaurant.

After implementing all these five requirements, the class has grown significantly and become quite complicated as shown below:

```
class Restaurant {
    String name;
    String password;
    String telNo;
    String faxNo;
    String address;
    String verificationCode;
    boolean isActivated;
    String faxNoToBeConfirmed;
    boolean isThereFaxNoToBeConfirmed = false;
    String catId;
    Vector holidays;
    Vector businessSessions;

    void activate(String verificationCode) {
        isActivated = (this.verificationCode.equals(verificationCode));
        if (isActivated && isThereFaxNoToBeConfirmed) {
            faxNo = faxNoToBeConfirmed;
            isThereFaxNoToBeConfirmed = false;
        }
    }

    void setFaxNo(String newFaxNo) {
        faxNoToBeConfirmed = newFaxNo;
        isThereFaxNoToBeConfirmed = true;
        isActivated = false;
    }

    boolean isInCategory(String catId) {
        return this.catId.equals(catId);
    }

    void addHoliday(int year, int month, int day) {
        ...
    }
}
```

```

void removeHoliday(int year, int month, int day) {
    ...
}
boolean addBusinessSession(int fromHour, int fromMin, int toHour, int toMin)
{
    ...
}
boolean isInBusinessHour(Calendar time) {
    ...
}
Vector getAllHolidays() {
    return holidays;
}
Vector getAllBusinessSessions() {
    return businessSessions;
}
}

```

Your task is to implement the five requirements above in separate classes, leaving the original simple Restaurant class unchanged.

11. This application is about students. Originally we had a simple Student class as shown in the code:

```

class StudentManagementSystem {
    Student students[];
}
class Student {
    String studentId;
    String name;
    Date dateOfBirth;
}

```

Later, in order to record what courses the student has enrolled in, on which dates he enrolled and how he paid for them, we modified the code as:

```

class StudentManagementSystem {
    Student students[];
}
class Student {
    String studentId;
    String name;
    Date dateOfBirth;
    String courseCodes[]; //the student has enrolled in these courses.
    Date enrollDates[]; //for each enrolled course, the date he enrolled.
    Payment payments[]; //for each enrolled course, how he paid.

    void enroll(String courseCode, Date enrollDate, Payment payment) {
        //add courseCode to courseCodes
        //add enrollDate to enrollDates
        //add payment to Payments
    }
    void unenroll(String courseCode) {
        ...
    }
}

```

Your task is to implement this requirement without modifying the Student class.

12. This application lets the network administrators document their server configurations. Originally we had a simple Server class as shown in the code:

```
class Server {
    String name;
    String CPUModel;
    int RAMSizeInMB;
    int diskSizeInMB;
    InetAddress ipAddress;
}
class ServerConfigSystem {
    Server servers[];
}
```

Now, there are four new requirements. You are required to implement these requirements without modifying the Server class:

1. An administrator (identified by an admin ID) can be assigned to be responsible for a server.
2. We can check if a server is a DHCP server or not. If yes, we can record the address scope it manages (e.g., from 192.168.0.21 to 192.168.0.254).
3. We can check if a server is a file server or not. If yes, we can set and check the disk space quota allocated for each user (identified by a user ID).
4. A server can be a DHCP server and a file server at the same time.

13. At the moment the code of a system is shown below:

```
class Customer {
    String id;
    String name;
    String address;
}
class Supplier {
    String id;
    String name;
    String telNo;
    String address;
}
class SalesSystem {
    Vector customers;
    Vector suppliers;
}
```

Implement the following new requirements while keeping the code fit:

1. Customers can place orders. Each order has a unique ID. It records the date of the order, the ID of the customer and the name and quantity of each item being

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

ordered.

2. The system can list all the orders placed by a particular customer.
 3. The system can list all the orders placed for a particular supplier.
 4. A supplier can provide a certain discount to some of the customers it selects (valid for some selected items only). The discount for different customers or different items may be different.
14. Come up with a class that was originally slim but becomes bloated as new requirements are implemented. Then separate the code into different classes and restore the class to its original slim form.

Hints

1. It is similar to the RentalProcessor example. First, make the code look identical:

```
class ParticipantsInDB
{
    ...
    void addParticipant(Participant part) {
        PreparedStatement st = conn.prepareStatement(some SQL);
        try {
            set the parameters of st;
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    void deleteAllParticipants() {
        PreparedStatement st = conn.prepareStatement(some SQL);
        try {
            set the parameters of st;
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}
```

2. Refer to the EditPaymentDialog.
3. The different values of "mode" do NOT result in significantly different behaviors. Therefore you should NOT create a sub-class to represent each value (if you do, you will find these classes are pretty much the same). The same is true for "upd".
4. No hint for this one.
5. The difference values of "departmentCode" do NOT result in significantly different

behaviors. Therefore you should NOT create a sub-class to represent each value.

6. No hint for this one.
7. The difference between the keys is just the frequency. Therefore, use an object containing a int (the frequency) to represent each key. Do NOT create a class to represent each key.
8. No hint for this one.
9. No hint for this one.
10. The system should keep a list of restaurant objects. They have all been activated. If a new restaurant account is created and waiting to be activated, do not put a new restaurant object onto the list yet. Instead, create another object in the system. This object should contain all the information of a restaurant and have a method like activate(restaurant ID, verification code). When the user enters the verification code, the system should call this method (directly or indirectly). If everything is correct, this object should put the restaurant object (create it if required) onto the list. It should also remove itself from the system because it has finished its duty.

To implement the new fax number activation, do something similar. Create an object in the system. When it is activated, it should set the fax number of the restaurant object and remove itself from the system.

You will note that there is quite some duplicate code after implementing the two requirements above. Remove the duplication.

The requirements about categories, holidays and business hours should be easy to implement.

11. No hint for this one.
12. To meet the 4th requirement, you can't use inheritance. Instead of a DHCP Server class, try creating a DHCP Configuration class.
13. Create classes like Orders and Discounts.

Sample solutions

1. The code below still contains duplicate code: they all create a prepared statement, set it up, execute it and then close it. If you need to remove this duplication at any cost, how do you do it?

```
class ParticipantsInDB {
    Connection conn;
    final String tableName="participants";
    void addParticipant(Participant part){
        PreparedStatement st = conn.prepareStatement("INSERT INTO "+tableName+"
VALUES (?, ?, ?, ?, ?)");
        try {
            st.setString(1, part.getId());
            st.setString(2, part.getEFirstName());
            st.setString(3, part.getELastName());
            //...
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    void deleteAllParticipants(){
        PreparedStatement st = conn.prepareStatement("DELETE FROM "+tableName);
        try {
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    void deleteParticipant(String participantId){
        PreparedStatement st = conn.prepareStatement("DELETE FROM "+tableName+"
WHERE id=?");
        try {
            st.setString(1, participantId);
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}
```

First, make the code look identical:

```
class ParticipantsInDB {
    Connection conn;
    final String tableName="participants";
    void addParticipant(Participant part){
        PreparedStatement st = conn.prepareStatement(some SQL);
        try {
            set the parameters of st;
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    void deleteAllParticipants(){
```

```

        PreparedStatement st = conn.prepareStatement(some SQL);
        try {
            set the parameters of st;
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}

void deleteParticipant(String participantId){
    PreparedStatement st = conn.prepareStatement(some SQL);
    try {
        set the parameters of st;
        st.executeUpdate();
    } finally {
        st.close();
    }
}
}
}

```

Then extract the duplicate code into a method:

```

class ParticipantsInDB {
    ...
    void ???() {
        PreparedStatement st = conn.prepareStatement(some SQL);
        try {
            set the parameters of st;
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}

```

To find a good name for this method, see what the code of the method actually does. In this case, it creates, executes and closes an SQL statement. So "executeSQL" should be a good name:

```

class ParticipantsInDB {
    ...
    void executeSQL() {
        PreparedStatement st = conn.prepareStatement(some SQL);
        try {
            set the parameters of st;
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}

```

As there are several ways to implement "set the parameters of st", we need to create an interface so that later we can create one implementation class for each implementation:

```

interface ??? {
    void setParametersOf(PreparedStatement st);
}

```

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

```

}
class ParticipantsInDB {
    ...
    void executeSQL(??? someObject) {
        PreparedStatement st = conn.prepareStatement(some SQL);
        try {
            someObject.setParametersOf(st);
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}

```

To find a good name for this interface, see what this interface does. As it has only one method (setParametersOf) which sets the parameters of a prepared statement, "StatementParamsFiller" should be a good name:

```

interface StatementParamsSetter{
    void setParametersOf(PreparedStatement st);
}
class ParticipantsInDB {
    ...
    void executeSQL(StatementParamsSetter paramsSetter) {
        PreparedStatement st = conn.prepareStatement(some SQL);
        try {
            paramsSetter.setParametersOf(st);
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}

```

The "some SQL" can be represented by different data values (strings):

```

interface StatementParamsSetter{
    void setParametersOf(PreparedStatement st);
}
class ParticipantsInDB {
    ...
    void executeSQL(String SQL, StatementParamsSetter paramsSetter) {
        PreparedStatement st = conn.prepareStatement(SQL);
        try {
            paramsSetter.setParametersOf(st);
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
}

```

Finally, provide different implementation classes. Make them inner classes if possible:

```

interface StatementParamsSetter{
    void setParametersOf(PreparedStatement st);
}

```

```

}
class ParticipantsInDB {
    ...
    void executeSQL(String SQL, StatementParamsSetter paramsSetter) {
        PreparedStatement st = conn.prepareStatement(SQL);
        try {
            paramsSetter.setParametersOf(st);
            st.executeUpdate();
        } finally {
            st.close();
        }
    }
    void addParticipant(final Participant part){
        executeSQL("INSERT INTO "+tableName+" VALUES (?, ?, ?, ?, ?)",
            new StatementParamsSetter() {
                void setParametersOf(PreparedStatement st) {
                    st.setString(1, part.getId());
                    st.setString(2, part.getEFirstName());
                    st.setString(3, part.getELastName());
                }
            });
    }
    void deleteAllParticipants(){
        executeSQL("DELETE FROM "+tableName,
            new StatementParamsSetter() {
                void setParametersOf(PreparedStatement st) {
                }
            });
    }
    void deleteParticipant(final String participantId){
        executeSQL("DELETE FROM "+tableName+" WHERE id=?",
            new StatementParamsSetter() {
                void setParametersOf(PreparedStatement st) {
                    st.setString(1, participantId);
                }
            });
    }
}
}

```

2. This application lets the network administrators document their server configurations. There are several types of servers as shown below. Design the code that implements a Swing GUI allowing the user to edit a server object. It must allow the user to change the type of the server. It must use a CardLayout to display the components just right for the type of the server.

```

class Server {
    String id;
    String CPUModel;
}
class DNSServer extends Server {
    String domainName;
}
class WINSServer extends Server {
    String replicationPartner;
    int replicationInterval;
}

```

```
class DomainController extends Server {
    boolean remainNT4Compatible;
}
```

Create a UI class for each server class:

```
abstract class ServerUI {
    JPanel panelForThisServerType;
    JTextField id;
    JTextField CPUModel;
    ServerUI(JTextField id, JTextField CPUModel) {
        this.id = id;
        this.CPUModel = CPUModel;
        this.panelForThisServerType = new JPanel(...);
    }
    void showServerCommonInfo(Server server) {
        id.setText(server.getId());
        CPUModel.setText(server.getCPUModel());
    }
    String getIdFromUI() {
        return id.getText();
    }
    String getCPUModelFromUI() {
        return CPUModel.getText();
    }
    JPanel getPanel() {
        return panelForThisServerType;
    }
    abstract boolean tryToDisplayServer(Server server);
    abstract Server makeServer();
}
class DNSServerUI extends ServerUI {
    JTextField domainName;
    DNSServerUI(JTextField id, JTextField CPUModel) {
        super(id, CPUModel);
        domainName = new JTextField(...);
        panelForThisServerType.add(domainName);
    }
    String toString() {
        return "DNS";
    }
    boolean tryToDisplayServer(Server server) {
        if (server instanceof DNSServer) {
            DNSServer dnsServer = (DNSServer)server;
            showServerCommonInfo(server);
            domainName.setText(dnsServer.getDomainName());
            return true;
        }
        return false;
    }
    Server makeServer() {
        return new DNSServer(
            getIdFromUI(),
            getCPUModelFromUI(),
            domainName.getText());
    }
}
```

```

}
class WINSServerUI extends ServerUI {
    JTextField replicationPartner;
    JTextField replicationInterval;
    WINSServerUI(JTextField id, JTextField CPUModel) {
        super(id, CPUModel);
        replicationPartner = new JTextField(...);
        replicationInterval = new JTextField(...);
        panelForThisServerType.add(replicationPartner);
        panelForThisServerType.add(replicationInterval);
    }
    String toString() {
        return "WINS";
    }
    boolean tryToDisplayServer(Server server) {
        ...
    }
    Server makeServer() {
        ...
    }
}
class DomainControllerUI extends ServerUI {
    ...
}
class ServerDialog extends JDialog {
    Server newServerToReturn;
    JPanel panelForCommonComponents;
    JPanel panelForServerTypeDependentComponents;
    JTextField id;
    JTextField CPUModel;
    JComboBox serverType;
    ServerUI serverUIs[];
    ServerDialog() {
        setupCommonComponents();
        setupServerUIs();
        setupServerTypeCombo();
        setupServerTypeDependentComponents();
    }
    void setupCommonComponents() {
        panelForCommonComponents = new JPanel(...);
        id = new JTextField(...);
        CPUModel = new JTextField(...);
        panelForCommonComponents.add(id);
        panelForCommonComponents.add(CPUModel);
        Container contentPane = getContentPane();
        contentPane.add(panelForCommonComponents, BorderLayout.NORTH);
    }
    void setupServerUIs() {
        ServerUI serverUIs[] = {
            new DNSServerUI(id, CPUModel),
            new WINSServerUI(id, CPUModel),
            new DomainControllerUI(id, CPUModel)
        };
        this.serverUIs = serverUIs;
    }
    void setupServerTypeCombo() {
        serverType = new JComboBox(serverUIs);

```

```

        panelForCommonComponents.add(serverType);
    }
    void setupServerTypeDependentComponents() {
        panelForServerTypeDependentComponents = new JPanel(...);
        panelForServerTypeDependentComponents.setLayout(new CardLayout());
        for (int i = 0; i < serverUIs.length; i++) {
            ServerUI UI = serverUIs[i];
            panelForServerTypeDependentComponents.add(
                UI.getPanel(),
                UI.toString());
        }
        Container contentPane = getContentPane();
        contentPane.add(
            panelForServerTypeDependentComponents,
            BorderLayout.CENTER);
    }
    Server editServer(Server server) {
        displayServer(server);
        setVisible(true);
        return newServerToReturn;
    }
    void displayServer(Server server) {
        for (int i = 0; i < serverUIs.length; i++) {
            ServerUI serverUI = serverUIs[i];
            if (serverUI.tryToDisplayServer(server)) {
                serverType.setSelectedItem(serverUI);
            }
        }
    }
    void onOK() {
        newServerToReturn = makeServer();
        dispose();
    }
    Server makeServer() {
        ServerUI serverUI = (ServerUI)serverType.getSelectedItem();
        return serverUI.makeServer();
    }
}

```

3. Point out and remove the code smells in the code below (this example is adapted from the one contributed by Eugen):

```

public static String executeUpdateSQLStmt(String sqlStmt, String mode) {
    // mode value --> ADD, MOD, DEL
    String outResult = "";
    final String connName = "tmp";
    DBConnectionManager connMgr = DBConnectionManager.getInstance();
    Connection conP = connMgr.getConnection(connName);
    Statement stmt = conP.createStatement();
    int upd = stmt.executeUpdate(sqlStmt);
    switch(upd) {
    case 1:
        if (mode.equalsIgnoreCase("ADD")) { outResult = "SUC Add"; }
        else if (mode.equalsIgnoreCase("MOD")) { outResult = "SUC Mod"; }
        else if (mode.equalsIgnoreCase("DEL")) { outResult = "SUC Del"; }
        break;
    case 0:
        if (mode.equalsIgnoreCase("ADD")) { outResult = "Err Add"; }
    }
}

```

```

        else if (mode.equalsIgnoreCase("MOD")) { outResult = "Err Mod"; }
        else if (mode.equalsIgnoreCase("DEL")) { outResult = "Err Del"; }
        break;
    default:
        if (mode.equalsIgnoreCase("ADD")) { outResult = "Err Add"; }
        else if (mode.equalsIgnoreCase("MOD")) { outResult = "Err Mod"; }
        else if (mode.equalsIgnoreCase("DEL")) { outResult = "Err Del"; }
        break;
    }
    stmt.close();
    conP.close();
    return outResult;
}

```

The three branches contain a lot of duplication. They are not much different and their differences can be represented using different data values (no need for different classes).

```

public static String executeUpdateSQLStmt(String sqlStmt, String mode) {
    // mode value --> ADD, MOD, DEL
    String outResult = "";
    final String connName = "tmp";
    DBConnectionManager connMgr = DBConnectionManager.getInstance();
    Connection conP = connMgr.getConnection(connName);
    Statement stmt = conP.createStatement();
    int upd = stmt.executeUpdate(sqlStmt);
    outResult = (upd==0 ? "SUC" : "Err")+
        " "+
        capitalizeString(mode);
    stmt.close();
    conP.close();
    return outResult;
}
public static String capitalizeString(String string) {
    ...
}

```

4. Point out and remove the code smells in the code below (this example is adapted from the one contributed by Carol):

```

class Account {
    final public int SAVING=0;
    final public int CHEQUE=1;
    final public int FIXED=2; //Portuguese currency
    private int accountType;
    private double balance;
    public double getInterestRate(...) { // Some method;
        ...
    }
    public Account(int accountType) {
        this.accountType=accountType;
    }
    public double calcInterest() {
        switch (accountType) {
            case SAVING:
                return balance*getInterestRate();
            case CHEQUE:

```

```
        return 0;
    case FIXED:
        return balance*(getInterestRate()+0.02);
    }
}
```

Remove the type code and switch using different classes. There is a wrong comment ("Portuguese currency") that should be removed.

```
abstract class Account {
    private double balance;
    abstract public double calcInterest();
    public double getInterestRate(...) { // Some method;
        ...
    }
}
class SavingAccount extends Account {
    public double calcInterest() {
        return getBalance()*getInterestRate();
    }
}
class ChequeAccount extends Account {
    public double calcInterest() {
        return 0;
    }
}
class FixedAccount extends Account {
    public double calcInterest() {
        return getBalance()*(getInterestRate()+0.02);
    }
}
```

5. Point out and remove the code smells in the code below (this example is contributed by Antonio):

```
class Department{
    final public int Account =0;
    final public int Marketing = 1;
    final public int CustomerServices = 2;
    protected int departmentCode;
    public Department(int departmentCode){
        this.departmentCode = departmentCode;
    }
    public String getDepartmentName(){
        switch (departmentCode){
            case Account:
                return "Account";
            case Marketing:
                return "Marketing";
            case CustomerServices:
                return "Customer Services";
        }
    }
}
```

Remove the type code and the switch. The differences between the types can be

represented using different data values (no need for different classes).

```
class Department {
    final public Department Account = new Department("Account");
    final public Department Marketing = new Department("Marketing");
    final public Department CustomerServices =
        new Department("Customer Services ");
    private String departmentName;
    private Department(String departmentName) {
        this.departmentName = departmentName;
    }
    public String getDepartmentName(){
        return departmentName;
    }
}
```

6. Point out and remove the code smells in the code below (this example is contributed by Carita):

```
class NormalPayment {
    int units;
    double rate;
    final double TAX_RATE = 0.1;
    double getBillableAmount() {
        double baseAmt = units * rate;
        double tax = baseAmt * TAX_RATE;
        return baseAmt + tax;
    }
}
class PaymentForSeniorCitizen {
    int units;
    double rate;
    final double TAX_RATE = 0.1;
    double getBillableAmount() {
        double baseAmt = units * rate * 0.8;
        double tax = baseAmt * (TAX_RATE - 0.5) ;
        return baseAmt + tax;
    }
}
```

The two classes contain quite some duplicate code. Extract the code to form a parent class:

```
abstract class Payment {
    int units;
    double rate;
    final double TAX_RATE = 0.1;
    abstract double getPreTaxedAmount();
    abstract double getTaxRate();
    double getBillableAmount() {
        return getPreTaxedAmount() * (1 + getTaxRate());
    }
    double getNormalAmount() {
        return units * rate;
    }
}
```

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

```
class NormalPayment extends Payment {
    double getPreTaxedAmount() {
        return getNormalAmount();
    }
    double getTaxRate() {
        return TAX_RATE;
    }
}
class PaymentForSeniorCitizen extends Payment {
    double getPreTaxedAmount() {
        return getNormalAmount()*0.8;
    }
    double getTaxRate() {
        return TAX_RATE - 0.5;
    }
}
```

7. Point out and remove the code smells in the code below (this example is contributed by Malaquias):

```
class PianoKey {
    final static int key0 = 0;
    final static int key1 = 1;
    final static int key2 = 2;
    int keyNumber;
    public void playSound() {
        if (keyNumber == 0) {
            //play the frequency for key0
        }
        else if (keyNumber == 1) {
            //play the frequency for key1
        }
        else if (keyNumber == 2) {
            //play the frequency for key2
        }
    }
}
class Piano {
    Vector rythmn;
    public void play() {
        for(int i=0;i<rythmn.size();i++) {
            ((PianoKey) rythmn.elementAt(i)).playSound();
        }
    }
}
```

Remove the type code and the long if-then-else-if. The differences between the types can be represented using different data values (no need for different classes).

```
class PianoKey {
    final static PianoKey key0 = new PianoKey(frequency for key0);
    final static PianoKey key1 = new PianoKey(frequency for key1);
    final static PianoKey key2 = new PianoKey(frequency for key2);
    ...
    int frequency;
    private PianoKey(int frequency) {
```

```

    this.frequency = frequency;
}
public void playSound() {
    //play the frequency.
}
}
class Piano {
    Vector rythmn;
    public void play() {
        for(int i=0;i<rythmn.size();i++) {
            ((PianoKey) rythmn.elementAt(i)).playSound();
        }
    }
}
}

```

8. Point out and remove the code smells in the code below (this example is contributed by Frankie):

```

class Account {
    final static int LEVEL_USER = 1;
    final static int LEVEL_ADMIN = 2;
    int accountLevel;
    Date expiredDate; // for user account only
    boolean hasLogin; // for admin account only
}
class ERPApp {
    public boolean checkLoginIssue(Account account) {
        if (account.getLevel() == Account.LEVEL_USER) {
            // Check the account expired date
            Date now = new Date();
            if (account.getExpiredDate().before(now))
                return false;
            return true;
        }
        else if (account.getLevel() == Account.LEVEL_ADMIN) {
            // No expired date for admin account
            // Check multillogin
            if (account.hasLogin())
                return false;
            return true;
        }
        return false;
    }
}
}

```

Remove the type code and switch using different classes. Remove the comments into code ("Check the account expired date", "Check multillogin" and etc.):

```

interface Account {
    boolean canLogin();
}
class UserAccount implements Account {
    Date expiredDate;
    boolean canLogin() {
        return isAccountExpired();
    }
    boolean isAccountExpired() {

```

```
        Date now = new Date();
        return !getExpiredDate().before(now);
    }
}
class AdminAccount implements Account {
    boolean hasLogin;
    boolean canLogin() {
        return !isTryingMultiLogin();
    }
    boolean isTryingMultiLogin() {
        return hasLogin;
    }
}
class ERPApp {
    public boolean checkLoginIssue(Account account) {
        return account.canLogin();
    }
}
```

9. Point out and remove the code smells in the code below (this example is adapted from the one contributed by YK):

```
class Form1 extends JDialog {
    JComboBox comboBoxReportType;
    Form1() {
        comboBoxReportType = new JComboBox();
        comboBoxReportType.addItem("r1");
        comboBoxReportType.addItem("r2");
        ...
        comboBoxReportType.addItem("r31c");
    }
    void processReport1() {
        //print some fancy report...
    }
    void processReport2() {
        //print another totally different fancy report...
    }
    ...
    void processReport31c() {
        //print yet another totally different fancy report...
    }
    void printReport(String repNo) {
        if (repNo.equals("r1"))
            processReport1();
        else if (repNo.equals("r2"))
            processReport2();
        ...
        else if (repNo.equals("r31c"))
            processReport31c();
    }
    void onPrintClick() {
        printReport((String) comboBoxReportType.getSelectedItem());
    }
}
```

Remove the type code and switch using different classes:

```

interface Report {
    void print();
}
class Report1 implements Report {
    String toString() {
        return "r1";
    }
    void print() {
        //print some fancy report...
    }
}
class Report2 implements Report {
    String toString() {
        return "r2";
    }
    void print() {
        //print another totally different fancy report...
    }
}
...
class Report31c implements Report {
    String toString() {
        return "31c";
    }
    void print() {
        //print yet another totally different fancy report...
    }
}
}
class Form1 extends JDialog {
    JComboBox comboBoxReportType;
    Report reports[] = {
        new Report1(),
        new Report2(),
        ...
        new Report31c()
    };
    Form1() {
        comboBoxReportType = new JComboBox();
        for (int i = 0; i < reports.length; i++) {
            comboBoxReportType.addItem(reports[i]);
        }
    }
    void onPrintClick() {
        Report report = (Report) comboBoxReportType.getSelectedItem();
        report.print();
    }
}
}

```

10. This application is about restaurants. Initially we created a Restaurant class to represent a restaurant account and includes information such as its name, access password, tel and fax number, address. The class is like:

```

class Restaurant {
    String name;
    String password;
    String telNo;
}

```

```
String faxNo;  
String address;  
}
```

Later, the following requirements are added in sequence:

1. After initial registration, the restaurant account is assigned an activation code by the system. Only after the user enters the activation code, the account will become activated. Until then, the account is inactive and login is not allowed.
2. If the user would like to change the fax number of the account, the new fax number will not take effect immediately (the existing fax number will remain in effect). Instead, the account is assigned an activation code by the system. Only after the user enters the activation code, the new fax number will take effect.
3. A restaurant can be marked as in a certain category (e.g., Chinese restaurant, Portuguese restaurant and etc.). A category is identified by a category ID.
4. The user can input the holidays for the restaurant.
5. The user can input the business hours for the restaurant.

After implementing all these five requirements, the class has grown significantly and become quite complicated as shown below:

```
class Restaurant {  
    String name;  
    String password;  
    String telNo;  
    String faxNo;  
    String address;  
    String verificationCode;  
    boolean isActivated;  
    String faxNoToBeConfirmed;  
    boolean isThereFaxNoToBeConfirmed = false;  
    String catId;  
    Vector holidays;  
    Vector businessSessions;  
  
    void activate(String verificationCode) {  
        isActivated = (this.verificationCode.equals(verificationCode));  
        if (isActivated && isThereFaxNoToBeConfirmed){  
            faxNo = faxNoToBeConfirmed;  
            isThereFaxNoToBeConfirmed = false;  
        }  
    }  
    void setFaxNo(String newFaxNo) {  
        faxNoToBeConfirmed = newFaxNo;  
        isThereFaxNoToBeConfirmed = true;  
        isActivated = false;  
    }  
    boolean isInCategory(String catId) {  
        return this.catId.equals(catId);  
    }  
}
```

```

}
void addHoliday(int year, int month, int day) {
    ...
}
void removeHoliday(int year, int month, int day) {
    ...
}
boolean addBusinessSession(int fromHour, int fromMin, int toHour, int toMin) {
    ...
}
boolean isInBusinessHour(Calendar time) {
    ...
}
Vector getAllHolidays() {
    return holidays;
}
Vector getAllBusinessSessions() {
    return businessSessions;
}
}

```

Your task is to implement the five requirements above in separate classes, leaving the original simple Restaurant class unchanged.

First, restore the Restaurant class to the original simple form:

```

class Restaurant {
    String name;
    String password;
    String telNo;
    String faxNo;
    String address;
}

```

Implement the initial account activation:

```

class RestaurantActivator {
    String verificationCode;
    Restaurant restaurantToAdd;
    boolean tryToActivate(String restName, String verificationCode) {
        if (restName.equals(restaurantToAdd.getName()) &&
            this.verificationCode.equals(verificationCode)) {
            //add restaurantToAdd to the system;
            return true;
        }
        return false;
    }
}

class RestaurantActivators {
    RestaurantActivator activators[];
    void activate(String restName, String verificationCode) {
        for (int i = 0; i < activators.length; i++) {
            if (activators[i].tryToActivate(restName, verificationCode)) {
                //remove activator[i] from activators;
                return;
            }
        }
    }
}

```

```

    }
}
class RestaurantSystem {
    Restaurant restaurants[];
    RestaurantActivators restaurantActivators;
}

```

Implement the set fax number activation:

```

class FaxNoActivator {
    String verificationCode;
    String newFaxNo;
    Restaurant restaurantToEdit;
    boolean tryToActivate(String restName, String verificationCode) {
        if (restName.equals(restaurantToEdit.getName()) &&
            this.verificationCode.equals(verificationCode)) {
            restaurantToEdit.setFaxNo(newFaxNo);
            return true;
        }
        return false;
    }
}
class FaxNoActivators {
    FaxNoActivator activators[];
    void activate(String restName, String verificationCode) {
        for (int i = 0; i < activators.length; i++) {
            if (activators[i].tryToActivate(restName, verificationCode)) {
                //remove activator[i] from activators;
                return;
            }
        }
    }
}
class RestaurantSystem {
    Restaurant restaurants[];
    RestaurantActivators restaurantActivators;
    FaxNoActivators faxNoActivators;
}

```

However, there is quite some duplicate code between the RestaurantActivator class and the FaxNoActivator class. The RestaurantActivators class is almost identical to the FaxNoActivators class. So, we make RestaurantActivator look identical to FaxNoActivator:

```

class RestaurantActivator {
    String verificationCode;
    Restaurant restaurant;
    boolean tryToActivate(String restName, String verificationCode) {
        if (restName.equals(restaurant.getName()) &&
            this.verificationCode.equals(verificationCode)) {
            //do something to the restaurant;
            return true;
        }
        return false;
    }
}

```

```

    }
}
class FaxNoActivator {
    String verificationCode;
    String newFaxNo;
    Restaurant restaurant;
    boolean tryToActivate(String restName, String verificationCode) {
        if (restName.equals(restaurant.getName()) &&
            this.verificationCode.equals(verificationCode)) {
            //do something to the restaurant;
            return true;
        }
        return false;
    }
}
}

```

Now, extract the common code to form a parent class and let the two classes extend it:

```

abstract class RestaurantTaskActivator {
    String verificationCode;
    Restaurant restaurant;
    boolean tryToActivate(String restName, String verificationCode) {
        if (restName.equals(restaurant.getName()) &&
            this.verificationCode.equals(verificationCode)) {
            doSomethingToRestaurant();
            return true;
        }
        return false;
    }
    abstract void doSomethingToRestaurant();
}
class RestaurantActivator extends RestaurantTaskActivator {
    void doSomethingToRestaurant() {
        //add restaurant to the system;
    }
}
class FaxNoActivator extends RestaurantTaskActivator {
    String newFaxNo;
    void doSomethingToRestaurant() {
        restaurant.setFaxNo(newFaxNo);
    }
}
class RestaurantTaskActivators {
    RestaurantTaskActivator activators[];
    void activate(String restName, String verificationCode) {
        for (int i = 0; i < activators.length; i++) {
            if (activators[i].tryToActivate(restName, verificationCode)) {
                //remove activator[i] from activators;
                return;
            }
        }
    }
}
}
class RestaurantSystem {
    Restaurant restaurants[];
    RestaurantTaskActivators restaurantTaskActivators;
}

```

Implement the category ID:

```
class Category {
    String catId;
    String IdOfRestaurantsInThisCat[];
    boolean isInCategory(String restName) {
        ...
    }
}
class RestaurantSystem {
    Restaurant restaurants[];
    RestaurantTaskActivators restaurantTaskActivators;
    Category categories[];
}
```

Implement the holidays:

```
class Holidays {
    Vector holidays;

    void addHoliday(int year, int month, int day) {
        ...
    }
    void removeHoliday(int year, int month, int day) {
        ...
    }
    Vector getAllHolidays() {
        return holidays;
    }
}
class RestaurantSystem {
    Restaurant restaurants[];
    RestaurantTaskActivators restaurantTaskActivators;
    Category categories[];
    HashMap mapRestIdToHolidays;
}
```

Implement the business sessions:

```
class BusinessSessions {
    Vector businessSessions;
    boolean addBusinessSession(int fromHour, int fromMin, int toHour, int toMin)
    {
        ...
    }
    boolean isInBusinessHour(Calendar time) {
        ...
    }
    Vector getAllBusinessSessions() {
        return businessSessions;
    }
}
class RestaurantSystem {
    Restaurant restaurants[];
    RestaurantTaskActivators restaurantTaskActivators;
    Category categories[];
    HashMap mapRestIdToHolidays;
}
```

```
HashMap mapRestIdToBusinessSessions;
}
```

11. This application is about students. Originally we had a simple Student class as shown in the code:

```
class StudentManagementSystem {
    Student students[];
}
class Student {
    String studentId;
    String name;
    Date dateOfBirth;
}
```

Later, in order to record what courses the student has enrolled in, on which dates he enrolled and how he paid for them, we modified the code as:

```
class StudentManagementSystem {
    Student students[];
}
class Student {
    String studentId;
    String name;
    Date dateOfBirth;
    String courseCodes[]; //the student has enrolled in these courses.
    Date enrollDates[]; //for each enrolled course, the date he enrolled.
    Payment payments[]; //for each enrolled course, how he paid.

    void enroll(String courseCode, Date enrollDate, Payment payment) {
        //add courseCode to courseCodes
        //add enrollDate to enrollDates
        //add payment to Payments
    }
    void unenroll(String courseCode) {
        ...
    }
}
```

Your task is to implement this requirement without modifying the Student class.

First, restore the Student class to the original simple form:

```
class Student {
    String studentId;
    String name;
    Date dateOfBirth;
}
```

Implement the enrollment requirement:

```
class Enrollment {
    String studentId;
    String courseCode;
    Date enrollDate;
}
```

```

    Payment payment;
}
class Enrollments {
    Enrollment enrollments[];
    void enroll(String studentId, String courseCode, Date enrollDate, Payment
payment) {
        Enrollment enrollment = new Enrollment(studentId, courseCode, ...);
        //add enrollment to enrollments;
    }
    void unenroll(String studentId, String courseCode) {
        ...
    }
}
class StudentManagementSystem {
    Student students[];
    Enrollments enrollments;
}

```

12. This application lets the network administrators document their server configurations. Originally we had a simple Server class as shown in the code:

```

class Server {
    String name;
    String CPUModel;
    int RAMSizeInMB;
    int diskSizeInMB;
    InetAddress ipAddress;
}
class ServerConfigSystem {
    Server servers[];
}

```

Now, there are four new requirements. You are required to implement these requirements without modifying the Server class:

1. An administrator (identified by an admin ID) can be assigned to be responsible for a server.
2. We can check if a server is a DHCP server or not. If yes, we can record the address scope it manages (e.g., from 192.168.0.21 to 192.168.0.254).
3. We can check if a server is a file server or not. If yes, we can set and check the disk space quota allocated for each user (identified by a user ID).
4. A server can be a DHCP server and a file server at the same time.

To allow the assignment of an administrator:

```

class Administrator {
    String adminId;
    Server serversAdminedByHim[];
}
class ServerConfigSystem {

```

```

Server servers[];
Administrator admins[];
}

```

To support DHCP servers:

```

class DHCPConfig {
    InetAddress startIP;
    InetAddress endIP;
}
class ServerConfigSystem {
    Server servers[];
    Administrator admins[];
    HashMap serverToDHCPConfig;
}

```

To support file servers:

```

class FileServerConfig {
    HashMap userIdToQuota;
    int getQuotaForUser(String userId) {
        ...
    }
    void setQuotaForUser(String userId, int quota) {
        ...
    }
}
class ServerConfigSystem {
    Server servers[];
    Administrator admins[];
    HashMap serverToDHCPConfig;
    HashMap serverToFileServerConfig;
}

```

Nothing needs to be done to allow a server to be a DHCP server and a file server at the same time.

13. At the moment the code of a system is shown below:

```

class Customer {
    String id;
    String name;
    String address;
}
class Supplier {
    String id;
    String name;
    String telNo;
    String address;
}
class SalesSystem {
    Vector customers;
    Vector suppliers;
}

```

Implement the following new requirements while keeping the code fit:

1. Customers can place orders. Each order has a unique ID. It records the date of the order, the ID of the customer and the name and quantity of each item being ordered.
2. The system can list all the orders placed by a particular customer.
3. The system can list all the orders placed for a particular supplier.
4. A supplier can provide a certain discount to some of the customers it selects (valid for some selected items only). The discount for different customers or different items may be different.

Implement the requirement of placing orders:

```
class Order {
    String orderId;
    String customerId;
    String supplierId;
    Date orderDate;
    OrderLine orderLines;
}
class OrderLine {
    String productName;
    int quantity;
}
class Orders {
    Order orders[];
    void placeOrder(String customerId, String supplierId, ...) {
        ...
    }
}
class SalesSystem {
    Vector customers;
    Vector suppliers;
    Orders orders;
}
```

Implement the requirement of printing orders placed by a customer:

```
class Orders {
    Order orders[];
    void placeOrder(String customerId, String supplierId, ...) {
        ...
    }
    void printOrdersByCustomer(String customerId) {
        ...
    }
}
```

Implement the requirement of printing orders for a supplier:

```
class Orders {
    Order orders[];
    void placeOrder(String customerId, String supplierId, ...) {
        ...
    }
    void printOrdersByCustomer(String customerId) {
        ...
    }
    void printOrdersForSupplier(String supplierId) {
        ...
    }
}
```

Implement the requirement of providing discounts:

```
class Discount {
    String supplierId;
    String customerId;
    String productName;
    double discountRate;
}
class Discounts {
    Discount discounts[];
    void addDiscount(
        String supplierId,
        String customerId,
        String productName,
        double discountRate) {
        ...
    }
    double findDiscount(
        String supplierId,
        String customerId,
        String productName) {
        ...
    }
}
class SalesSystem {
    Vector customers;
    Vector suppliers;
    Orders orders;
    Discounts discounts;
}
```