

Problem in the code above: The code will keep changing

There is a problem in the code above: if we need to support one more shape (e.g., triangles), the Shape class needs to change, and the drawShapes method in CADApp class also needs to change, e.g.:

```
class Shape {
    final static int TYPELINE = 0;
    final static int TYPERECTANGLE = 1;
    final static int TYPECIRCLE = 2;
    final static int TYPETRIANGLE = 3;
    int shapeType;
    Point p1;
    Point p2;
    //third point of the triangle.
    Point p3;
    int radius;
}
class CADApp {
    void drawShapes(Graphics graphics, Shape shapes[]) {
        for (int i = 0; i < shapes.length; i++) {
            switch (shapes[i].getType()) {
                case Shape.TYPELINE:
                    graphics.drawLine(shapes[i].getP1(), shapes[i].getP2());
                    break;
                case Shape.TYPERECTANGLE:
                    //draw the four edges.
                    graphics.drawLine(...);
                    graphics.drawLine(...);
                    graphics.drawLine(...);
                    graphics.drawLine(...);
                    break;
                case Shape.TYPECIRCLE:
                    graphics.drawCircle(shapes[i].getP1(), shapes[i].getRadius());
                    break;
                case Shape.TYPETRIANGLE:
                    graphics.drawLine(shapes[i].getP1(), shapes[i].getP2());
                    graphics.drawLine(shapes[i].getP2(), shapes[i].getP3());
                    graphics.drawLine(shapes[i].getP3(), shapes[i].getP1());
                    break;
            }
        }
    }
}
```

If in the future one more shape is added, they will have to change again. This is no good thing. Ideally, we hope that a class, a method and etc. do not need to change after they are written. They should be readily available for reuse without change. But the current case is the opposite. When we keep adding new shapes, we will keep changing the Shape class and the drawShapes method in the CADApp class.

How to stabilize this code (no need to change again)? Before answering this question, let's consider another question first: Given a piece of code, how to check if it is stable?

How to check if some code is stable

To check if some given code is stable, we may do it like this: Assume if some certain situations arise or some certain new requirements come, then check if the code needs to change. However, it is very hard to do because there are too many possible situations that could arise.

A simpler method is, when we find that this is already the third time that we change this code, we consider it unstable. This is a "lazy" and "passive" method. We start to handle to problem only when we feel the pain. However, it is a very effective method.

In addition, there is another simple but more "active" method: If the code is unstable or contains some other potential problems, the code will often contain some obvious traces. We call these traces "code smells". Code smells are not always a bad thing, but most often they are indeed a bad thing. Therefore, whenever we find code smells in the code, we must become alerted and check the code carefully.

Now, let's see the code smells in the example code above.

Code smells in the example code

The first code smell: The code uses type code:

```
class Shape {
    final int TYPELINE = 0;
    final int TYPERECTANGLE = 1;
    final int TYPECIRCLE = 2;
    int shapeType;
    ...
}
```

This is a severe warning that our code may have problems.

The second code smell: The Shape class contains variables that are not always used. For example, the variable radius is used only when the Shape is a circle:

```
class Shape {
    ...
    Point p1;
    Point p2;
    int radius; //not always used!
}
```

The third code smell: We cannot give the variable p1 (or p2) a better name, because it has different meanings in different situations:

```
class Shape {
    ...
    Point p1; //should we call it startPoint, lowerLeftCorner or center?
    Point p2;
}
```

The fourth code smell: The switch statement in drawShapes. When we use switch (or a long list of if-then-else-if), get alerted. The switch statement commonly appears with type code at the same time.

Now, let's see how to remove the code smells in the example code above.

Removing code smells: How to remove a type code

Usually, in order to remove a type code, we can create a sub-class for each type code value, e.g.:

```
class Shape {
}
class Line extends Shape {
    Point startPoint;
    Point endPoint;
}
class Rectangle extends Shape {
    Point lowerLeftCorner;
    Point upperRightCorner;
}
class Circle extends Shape {
    Point center;
    int radius;
}
```

Because there is no type code, drawShapes has to use instanceof to check the type of the Shape object. Therefore, it can't use switch any more. It must change to use if-then-else such as:

```
class CADApp {
    void drawShapes(Graphics graphics, Shape shapes[]) {
        for (int i = 0; i < shapes.length; i++) {
            if (shapes[i] instanceof Line) {
                Line line = (Line)shapes[i];
                graphics.drawLine(line.getStartPoint(), line.getEndPoint());
            } else if (shapes[i] instanceof Rectangle) {
                Rectangle rect = (Rectangle)shapes[i];
                graphics.drawLine(...);
                graphics.drawLine(...);
                graphics.drawLine(...);
                graphics.drawLine(...);
            } else if (shapes[i] instanceof Circle) {
                Circle circle = (Circle)shapes[i];
            }
        }
    }
}
```

```

        graphics.drawCircle(circle.getCenter(), circle.getRadius());
    }
}
}

```

Now, without the type code, the variables in each class (Shape, Line, Rectangle, Circle) are always useful. They also have much better names (p1 can now be called startPoint). The only code smell left is the long if-then-else-if in drawShapes. Our next step is to remove this long if-then-else-if.

Removing code smells: How to remove a long if-then-else-if

Usually, in order to remove a long if-then-else-if or a switch, we need to try to make the code in each conditional branch identical. For drawShapes, we will first write the code in each branch in a more abstract way:

```

class CADApp {
    void drawShapes(Graphics graphics, Shape shapes[]) {
        for (int i = 0; i < shapes.length; i++) {
            if (shapes[i] instanceof Line) {
                draw the line;
            } else if (shapes[i] instanceof Rectangle) {
                draw the rectangle;
            } else if (shapes[i] instanceof Circle) {
                draw the circle;
            }
        }
    }
}

```

However, the code in these three branches is still different. So, make it even more abstract:

```

class CADApp {
    void drawShapes(Graphics graphics, Shape shapes[]) {
        for (int i = 0; i < shapes.length; i++) {
            if (shapes[i] instanceof Line) {
                draw the shape;
            } else if (shapes[i] instanceof Rectangle) {
                draw the shape;
            } else if (shapes[i] instanceof Circle) {
                draw the shape;
            }
        }
    }
}

```

Now, the code in these three branches is identical. Therefore, we no longer need the different branches:

```

class CADApp {

```

```
void drawShapes(Graphics graphics, Shape shapes[]) {
    for (int i = 0; i < shapes.length; i++) {
        draw the shape;
    }
}
```

Finally, turn "draw the shape" into a method of the Shape class:

```
class CADApp {
    void drawShapes(Graphics graphics, Shape shapes[]) {
        for (int i = 0; i < shapes.length; i++) {
            shapes[i].draw(graphics);
        }
    }
}
```

Of course, we need to provide this draw method:

```
abstract class Shape {
    abstract void draw(Graphics graphics);
}
class Line extends Shape {
    Point startPoint;
    Point endPoint;
    void draw(Graphics graphics) {
        graphics.drawLine(getStartPoint(), getEndPoint());
    }
}
class Rectangle extends Shape {
    Point lowerLeftCorner;
    Point upperRightCorner;
    void draw(Graphics graphics) {
        graphics.drawLine(...);
        graphics.drawLine(...);
        graphics.drawLine(...);
        graphics.drawLine(...);
    }
}
class Circle extends Shape {
    Point center;
    int radius;
    void draw(Graphics graphics) {
        graphics.drawCircle(getCenter(), getRadius());
    }
}
```

Turning an abstract class into an interface

Now, Shape is an abstract class without any executable code. It seems more appropriate to turn it into an interface:

```
interface Shape {
    void draw(Graphics graphics);
}
class Line implements Shape {
    ...
}
class Rectangle implements Shape {
    ...
}
class Circle implements Shape {
    ...
}
```

The improved code

The improved code is shown below:

```
interface Shape {
    void draw(Graphics graphics);
}
class Line implements Shape {
    Point startPoint;
    Point endPoint;
    void draw(Graphics graphics) {
        graphics.drawLine(getStartPoint(), getEndPoint());
    }
}
class Rectangle implements Shape {
    Point lowerLeftCorner;
    Point upperRightCorner;
    void draw(Graphics graphics) {
        graphics.drawLine(...);
        graphics.drawLine(...);
        graphics.drawLine(...);
        graphics.drawLine(...);
    }
}
class Circle implements Shape {
    Point center;
    int radius;
    void draw(Graphics graphics) {
        graphics.drawCircle(getCenter(), getRadius());
    }
}
class CADApp {
    void drawShapes(Graphics graphics, Shape shapes[]) {
        for (int i = 0; i < shapes.length; i++) {
            shapes[i].draw(graphics);
        }
    }
}
```

If we need to support one more shape (e.g., triangle), none of classes needs to change. All it takes is to create a new Triangle class.

Another example

Let's see another example. In this system there are three types of users: regular users, administrators and guests. Regular users must change their password once every 90 days (or sooner). Administrators must change their password once every 30 days. Guests don't need to change passwords. Only regular users and administrators can print reports. Please read the current code:

```
class UserAccount {
    final static int USERTYPE_NORMAL = 0;
    final static int USERTYPE_ADMIN = 1;
    final static int USERTYPE_GUEST = 2;
    int userType;
    String id;
    String name;
    String password;
    Date dateOfLastPasswdChange;
    public boolean checkPassword(String password) {
        ...
    }
}

class InventoryApp {
    void login(UserAccount userLoggingIn, String password) {
        if (userLoggingIn.checkPassword(password)) {
            GregorianCalendar today = new GregorianCalendar();
            GregorianCalendar expiryDate = getAccountExpiryDate(userLoggingIn);
            if (today.after(expiryDate)) {
                //prompt the user to change password
                ...
            }
        }
    }

    GregorianCalendar getAccountExpiryDate(UserAccount account) {
        int passwordMaxAgeInDays = getPasswordMaxAgeInDays(account);
        GregorianCalendar expiryDate = new GregorianCalendar();
        expiryDate.setTime(account.dateOfLastPasswdChange);
        expiryDate.add(Calendar.DAY_OF_MONTH, passwordMaxAgeInDays);
        return expiryDate;
    }

    int getPasswordMaxAgeInDays(UserAccount account) {
        switch (account.getType()) {
            case UserAccount.USERTYPE_NORMAL:
                return 90;
            case UserAccount.USERTYPE_ADMIN:
                return 30;
            case UserAccount.USERTYPE_GUEST:
                return Integer.MAX_VALUE;
        }
    }

    void printReport(UserAccount currentUser) {
        boolean canPrint;
        switch (currentUser.getType()) {
            case UserAccount.USERTYPE_NORMAL:
                canPrint = true;
                break;
            case UserAccount.USERTYPE_ADMIN:
                canPrint = true;
        }
    }
}
```

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

```
        break;
    case UserAccount.USERTYPE_GUEST:
        canPrint = false;
    }
    if (!canPrint) {
        throw new SecurityException("You have no right");
    }
    //print the report.
}
}
```

Use an object to represent a type code value

According to the method described before, to remove a type code, we only need to create a sub-class for each type code value, e.g.:

```
abstract class UserAccount {
    String id;
    String name;
    String password;
    Date dateOfLastPasswdChange;
    abstract int getPasswordMaxAgeInDays();
    abstract boolean canPrintReport();
}
class NormalUserAccount extends UserAccount {
    int getPasswordMaxAgeInDays() {
        return 90;
    }
    boolean canPrintReport() {
        return true;
    }
}
class AdminUserAccount extends UserAccount {
    int getPasswordMaxAgeInDays() {
        return 30;
    }
    boolean canPrintReport() {
        return true;
    }
}
class GuestUserAccount extends UserAccount {
    int getPasswordMaxAgeInDays() {
        return Integer.MAX_VALUE;
    }
    boolean canPrintReport() {
        return false;
    }
}
```

However, the behavior (code) of these three sub-classes is too similar: Their `getPasswordMaxAgeInDays` methods only differ in a value (30, 90 or `Integer.MAX_VALUE`). Their `canPrintReport` methods also only differ in a value (true or false). Therefore, these three

user types can be represented using three objects instead of three sub-classes:

```

class UserAccount {
    UserType userType;
    String id;
    String name;
    String password;
    Date dateOfLastPasswdChange;
    UserType getType() {
        return userType;
    }
}

class UserType {
    int passwordMaxAgeInDays;
    boolean allowedToPrintReport;
    UserType(int passwordMaxAgeInDays, boolean allowedToPrintReport) {
        this.passwordMaxAgeInDays = passwordMaxAgeInDays;
        this.allowedToPrintReport = allowedToPrintReport;
    }
    int getPasswordMaxAgeInDays() {
        return passwordMaxAgeInDays;
    }
    boolean canPrintReport() {
        return allowedToPrintReport;
    }
    static UserType normalUserType = new UserType(90, true);
    static UserType adminUserType = new UserType(30, true);
    static UserType guestUserType = new UserType(Integer.MAX_VALUE, false);
}

class InventoryApp {
    void login(UserAccount userLoggingIn, String password) {
        if (userLoggingIn.checkPassword(password)) {
            GregorianCalendar today = new GregorianCalendar();
            GregorianCalendar expiryDate = getAccountExpiryDate(userLoggingIn);
            if (today.after(expiryDate)) {
                //prompt the user to change password
                ...
            }
        }
    }

    GregorianCalendar getAccountExpiryDate(UserAccount account) {
        int passwordMaxAgeInDays = getPasswordMaxAgeInDays(account);
        GregorianCalendar expiryDate = new GregorianCalendar();
        expiryDate.setTime(account.dateOfLastPasswdChange);
        expiryDate.add(Calendar.DAY_OF_MONTH, passwordMaxAgeInDays);
        return expiryDate;
    }

    int getPasswordMaxAgeInDays(UserAccount account) {
        return account.getType().getPasswordMaxAgeInDays();
    }

    void printReport(UserAccount currentUser) {
        boolean canPrint;
        canPrint = currentUser.getType().canPrintReport();
        if (!canPrint) {
            throw new SecurityException("You have no right");
        }
        //print the report.
    }
}

```

```
}
```

Note that using an object to represent a type code value can also remove the switch statement.

Summary on type code removal

To remove type codes and switch statements, there are two methods:

1. Use different sub-classes of a base class to represent the different type code values.
2. Use different objects of a class to represent the different type code values.

When different type code values need very different behaviors, use method 1. When their behaviors are mostly the same and only differ in values, use method 2.

Common code smells

Type codes and switch statements are common code smells. In addition, there are quite some other code smells that are also very common. Below is a summary list:

- Duplicate code.
- Too many comments.
- Type code.
- Switch or a long if-then-else-if.
- Cannot give a good name to a variable, method or class.
- Use names like XXXUtil, XXXManager, XXXController and etc.
- Use the words "And", "Or" and etc. in the name of a variable, method or class.
- Some instance variables are not always used.
- A method contains too much code.

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

- A class contains too much code.
- A method takes too many parameters.
- Two classes are using each other.

References

- <http://c2.com/cgi/wiki?CodeSmell>.
- The Open Closed Principle states: If we need to add more functionality, we should only need to add new code, but not change the existing code. Removing type codes and switch statements is a common way to achieve Open Closed Principle. For more information about Open Closed Principle, please see <http://www.objectmentor.com/publications/ocp.pdf>.
- Robert C. Martin, Agile Software Development, Principles, Patterns, and Practices, Prentice Hall, 2002.
- Bertrand Meyer, Object-Oriented Software Construction, Pearson Higher Education, 1988.
- Martin Fowler, Kent Beck, John Brant, William Opdyke and Don Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999. This book lists many code smells and the refactoring methods.
- Martin Fowler's web site about refactoring: <http://www.refactoring.com/catalog/index.html>.



Chapter exercises

Introduction

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

Problems

1. Point out and remove the code smells in the code (this example was inspired by the one given by YK):

```
class FoodSalesReport {
    int q0; //how many items of rice sold?
    int q1; //how many items of noodle sold?
    int q2; //how many items of drink sold?
    int q3; //how many items of dessert sold?
    void LoadData(Connection conn) {
        PreparedStatement st = conn.prepareStatement("select "+
            "sum(case when foodType=0 then qty else 0 end) as totalQty0, "+
            "sum(case when foodType=1 then qty else 0 end) as totalQty1, "+
            "sum(case when foodType=2 then qty else 0 end) as totalQty2, "+
            "sum(case when foodType=3 then qty else 0 end) as totalQty3 "+
            "from foodSalesTable group by foodType");
        try {
            ResultSet rs = st.executeQuery();
            try {
                rs.next();
                q0 = rs.getInt("totalQty0");
                q1 = rs.getInt("totalQty1");
                q2 = rs.getInt("totalQty2");
                q3 = rs.getInt("totalQty3");
            } finally {
                rs.close();
            }
        } finally {
            st.close();
        }
    }
}
```

2. Point out and remove the code smells in the code (this example was inspired by the one given by YK):

```
class SurveyData {
    String path; //save the data to this file.
    boolean hidden; //should the file be hidden?
    //set the path to save the data according to the type of data (t).
    void setSavePath(int t) {
        if (t==0) { //raw data.
            path = "c:/application/data/raw.dat";
            hidden = true;
        }
    }
}
```

```

    } else if (t==1) { //cleaned up data.
        path = "c:/application/data/cleanedUp.dat";
        hidden = true;
    } else if (t==2) { //processed data.
        path = "c:/application/data/processed.dat";
        hidden = true;
    } else if (t==3) { //data ready for publication.
        path = "c:/application/data/publication.dat";
        hidden = false;
    }
}
}
}

```

3. Point out and remove the code smells in the code (this example was inspired by the one given by Eugen):

```

class CustomersInDB {
    Connection conn;
    Customer getCustomer(String IDNumber) {
        PreparedStatement st = conn.prepareStatement(
            "select * from customer where ID=?");
        try {
            st.setString(1,
                IDNumber.replace('-', ' ').
                    replace('(', ' ').
                    replace(')', ' ').
                    replace('/', ' '));
            ResultSet rs = st.executeQuery();
            ...
        } finally {
            st.close();
        }
    }
    void addCustomer(Customer customer) {
        PreparedStatement st = conn.prepareStatement(
            "insert into customer values(?,?,?,?)");
        try {
            st.setString(1,
                customer.getIDNumber().replace('-', ' ').
                    replace('(', ' ').
                    replace(')', ' ').
                    replace('/', ' '));
            st.setString(2, customer.getName());
            ...
            st.executeUpdate();
            ...
        } finally {
            st.close();
        }
    }
}
}

```

4. The code below contains duplicate code: the loop in `printOverdueRentals` and in `countOverdueRentals`. If you need to remove this duplication at any cost, how do you do it?

```

class BookRentals {
    Vector rentals;
    int countRentals() {

```

```
        return rentals.size();
    }
    BookRental getRentalAt(int i) {
        return (BookRental)rentals.elementAt(i);
    }
    void printOverdueRentals() {
        int i;
        for (i=0; i<countRentals(); i++) {
            BookRental rental = getRentalAt(i);
            if (rental.isOverdue())
                System.out.println(rental.toString());
        }
    }
    int countOverdueRentals() {
        int i, count;
        count=0;
        for (i=0; i<countRentals(); i++)
            if (getRentalAt(i).isOverdue())
                count++;
        return count;
    }
}
```

5. Point out and remove the code smells in the code:

```
class Currency {
    final public int USD=0;
    final public int RMB=1;
    final public int ESCUDO=2; //Portuguese currency
    private int currencyCode;
    public Currency(int currencyCode) {
        this.currencyCode=currencyCode;
    }
    public String format(int amount) {
        switch (currencyCode) {
            case USD:
                //return something like $1,200
            case RMB:
                //return something like RMB1,200
            case ESCUDO:
                //return something like $1.200
        }
    }
}
```

6. Point out and remove the code smells in the code:

```
class Payment {
    final static String FOC = "FOC"; //free of charge.
    final static String TT = "TT"; //paid by telegraphic transfer.
    final static String CHEQUE = "Cheque"; //paid by cheque.
    final static String CREDIT_CARD = "CreditCard"; //paid by credit card.
    final static String CASH = "Cash"; //paid by cash.
    //type of payment. Must be one of the above constant.
    String paymentType;
    Date paymentDate; //if FOC, the date the fee is waived.
    int actualPayment; //if FOC, it is 0.
    int discount; //if FOC, the amount that is waived.
}
```

```

String bankName; //if it is by TT, cheque or credit card.
String chequeNumber; //if it is by cheque.
//if it is by credit card.
String creditCardType;
String creditCardHolderName;
String creditCardNumber;
Date creditCardExpiryDate;
int getNominalPayment() {
    return actualPayment+discount;
}
String getBankName() {
    if (paymentType.equals(TT) ||
        paymentType.equals(CHEQUE) ||
        paymentType.equals(CREDIT_CARD)) {
        return bankName;
    }
    else {
        throw new Exception("bank name is undefined for this payment type");
    }
}
}

```

7. In the above application, there is a dialog to allow the user edit a payment object. The user can choose the payment type in a combo box. Then the related components for that payment type will be displayed. This function is provided by the CardLayout in Java. The code is shown below. Update the code accordingly.

```

class EditPaymentDialog extends JDialog {
    Payment newPayment; //new payment to be returned.
    JPanel sharedPaymentDetails;
    JPanel uniquePaymentDetails;
    JTextField paymentDate;
    JComboBox paymentType;
    JTextField discountForFOC;
    JTextField bankNameForTT;
    JTextField actualAmountForTT;
    JTextField discountForTT;
    JTextField bankNameForCheque;
    JTextField chequeNumberForCheque;
    JTextField actualAmountForCheque;
    JTextField discountForCheque;
    ...

    EditPaymentDialog() {
        //setup the components.
        Container contentPane = getContentPane();
        String comboBoxItems[] = { //available payment types.
            Payment.FOC,
            Payment.TT,
            Payment.CHEQUE,
            Payment.CREDIT_CARD,
            Payment.CASH
        };
        //setup the components for the details shared by all types of payment.
        sharedPaymentDetails = new JPanel();
        paymentDate = new JTextField();
        paymentType = new JComboBox(comboBoxItems);
        sharedPaymentDetails.add(paymentDate);
    }
}

```

```

        sharedPaymentDetails.add(paymentType);
        contentPane.add(sharedPaymentDetails, BorderLayout.NORTH);
        //setup the unique components for each type of payment.
        uniquePaymentDetails = new JPanel();
        uniquePaymentDetails.setLayout(new CardLayout());
        //setup a panel for FOC payment.
        JPanel panelForFOC = new JPanel();
        discountForFOC = new JTextField();
        panelForFOC.add(discountForFOC);
        uniquePaymentDetails.add(panelForFOC, Payment.FOC);
        //setup a panel for TT payment.
        JPanel panelForTT = new JPanel();
        bankNameForTT = new JTextField();
        actualAmountForTT = new JTextField();
        discountForTT = new JTextField();
        panelForTT.add(bankNameForTT);
        panelForTT.add(actualAmountForTT);
        panelForTT.add(discountForTT);
        uniquePaymentDetails.add(panelForTT, Payment.TT);
        //setup a panel for cheque payment.
        JPanel panelForCheque = new JPanel();
        bankNameForCheque = new JTextField();
        chequeNumberForCheque = new JTextField();
        actualAmountForCheque = new JTextField();
        discountForCheque = new JTextField();
        panelForCheque.add(bankNameForCheque);
        panelForCheque.add(chequeNumberForCheque);
        panelForCheque.add(actualAmountForCheque);
        panelForCheque.add(discountForCheque);
        uniquePaymentDetails.add(panelForCheque, Payment.CHEQUE);
        //setup a panel for credit card payment.
        ...
        //setup a panel for cash payment.
        ...
        contentPane.add(uniquePaymentDetails, BorderLayout.CENTER);
    }
}
Payment editPayment(Payment payment) {
    displayPayment(payment);
    setVisible(true);
    return newPayment;
}
void displayPayment(Payment payment) {
    paymentDate.setText(payment.getDateAsString());
    paymentType.setSelectedItem(payment.getType());
    if (payment.getType().equals(Payment.FOC)) {
        discountForFOC.setText(Integer.toString(payment.getDiscount()));
    }
    else if (payment.getType().equals(Payment.TT)) {
        bankNameForTT.setText(payment.getBankName());
        actualAmountForTT.setText(
            Integer.toString(payment.getActualAmount()));
        discountForTT.setText(Integer.toString(payment.getDiscount()));
    }
    else if (payment.getType().equals(Payment.CHEQUE)) {
        bankNameForCheque.setText(payment.getBankName());
        chequeNumberForCheque.setText(payment.getChequeNumber());
        actualAmountForCheque.setText(
            Integer.toString(payment.getActualAmount()));
        discountForCheque.setText(Integer.toString(payment.getDiscount()));
    }
}

```

```

        else if (payment.getType().equals(Payment.CREDIT_CARD)) {
            //...
        }
        else if (payment.getType().equals(Payment.CASH)) {
            //...
        }
    }
    //when the user clicks OK.
    void onOK() {
        newPayment = makePayment();
        dispose();
    }
    //make a payment from the components.
    Payment makePayment() {
        String paymentTypeString = (String) paymentType.getSelectedItem();
        Payment payment = new Payment(paymentTypeString);
        payment.setDateAsText(paymentDate.getText());
        if (paymentTypeString.equals(Payment.FOC)) {
            payment.setDiscount(Integer.parseInt(discountForFOC.getText()));
        }
        else if (paymentTypeString.equals(Payment.TT)) {
            payment.setBankName(bankNameForTT.getText());
            payment.setActualAmount(
                Integer.parseInt(actualAmountForTT.getText()));
            payment.setDiscount(
                Integer.parseInt(discountForTT.getText()));
        }
        else if (paymentTypeString.equals(Payment.CHEQUE)) {
            payment.setBankName(bankNameForCheque.getText());
            payment.setChequeNumber(chequeNumberForCheque.getText());
            payment.setActualAmount(
                Integer.parseInt(actualAmountForCheque.getText()));
            payment.setDiscount(
                Integer.parseInt(discountForCheque.getText()));
        }
        else if (paymentTypeString.equals(Payment.CREDIT_CARD)) {
            //...
        }
        else if (paymentTypeString.equals(Payment.CASH)) {
            //...
        }
        return payment;
    }
}

```

8. This is an embedded application controlling a cooker. In every second, it will check if the cooker is over-heated (e.g., short-circuited). If yes it will cut itself off the power and make an alarm using its built-in speaker. It will also check if the moisture inside is lower than a certain threshold (e.g., the rice is cooked). If yes, it will turn its built-in heater to 50 degree Celsius just to keep the rice warm. In the future, you expect that some more things will need to be done in every second.

Point out and remove the problem in the code.

```

class Scheduler extends Thread {
    Alarm alarm;
    HeatSensor heatSensor;
}

```

```

PowerSupply powerSupply;
MoistureSensor moistureSensor;
Heater heater;
public void run() {
    for (;;) {
        Thread.sleep(1000);
        //check if it is overheated.
        if (heatSensor.isOverHeated()) {
            powerSupply.turnOff();
            alarm.turnOn();
        }
        //check if the rice is cooked.
        if (moistureSensor.getMoisture()<60) {
            heater.setTemperature(50);
        }
    }
}
}
}

```

9. This application is concerned with the training courses. The schedule of a course can be expressed in three ways (as of now): weekly, range or list. A weekly schedule is like "every Tuesday for 5 weeks starting from Oct. 22". A range schedule is like "Every day from Oct. 22 to Nov. 3". A list schedule is like "Oct. 22, Oct. 25, Nov. 3, Nov. 10". In this exercise we will ignore the time and just assume that it is always 7:00pm-10:00pm. It is expected that new ways to express the schedule may be added in the future.

Point out and remove the code smells in the code:

```

class Course {
    static final int WEEKLY=0;
    static final int RANGE=1;
    static final int LIST=2;
    String courseTitle;
    int scheduleType; // WEEKLY, RANGE, or LIST
    int noWeeks; // For WEEKLY.
    Date fromDate; // for WEEKLY and RANGE.
    Date toDate; // for RANGE.
    Date dateList[]; // for LIST.

    int getDurationInDays() {
        switch (scheduleType) {
            case WEEKLY:
                return noWeeks;
            case RANGE:
                int msInOneDay = 24*60*60*1000;
                return (int)((toDate.getTime()-fromDate.getTime())/msInOneDay);
            case LIST:
                return dateList.length;
            default:
                return 0; // unknown schedule type!
        }
    }
}

void printSchedule() {
    switch (scheduleType) {
        case WEEKLY:
            //...
        case RANGE:

```

```
        //...
        case LIST:
            //...
    }
}
```

10. This application is concerned with training courses. A course has a title, a fee and a list of sessions. However, sometimes a course can consist of several modules, each of which is a course. For example, there may be a compound course "Fast track to becoming a web developer" which consists of three modules: a course named "HTML", a course named "FrontPage" and a course named "Flash". It is possible for a module to consist of some other modules. If a course consists of modules, its fee and schedule are totally determined by that of its modules and thus it will not maintain its list of sessions.

Point out and remove the code smells in the code:

```
class Session {
    Date date;
    int startHour;
    int endHour;
    int getDuration() {
        return endHour-startHour;
    }
}

class Course {
    String courseTitle;
    Session sessions[];
    double fee;
    Course modules[];

    Course(String courseTitle, double fee, Session sessions[]) {
        //...
    }
    Course(String courseTitle, Course modules[]) {
        //...
    }
    String getTitle() {
        return courseTitle;
    }
    double getDuration() {
        int duration=0;
        if (modules==null)
            for (int i=0; i<sessions.length; i++)
                duration += sessions[i].getDuration();
        else
            for (int i=0; i<modules.length; i++)
                duration += modules[i].getDuration();
        return duration;
    }
    double getFee() {
        if (modules==null)
            return fee;
        else {
            double totalFee = 0;
            for (int i=0; i<modules.length; i++)
                totalFee += modules[i].getFee();
        }
    }
}
```

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

```
        return totalFee;
    }
}
void setFee(int fee) throws Exception {
    if (modules==null)
        this.fee = fee;
    else
        throw new Exception("Please set the fee of each module one by one");
}
}
```

11. Point out and remove the code smells in the code:

```
class BookRental {
    String bookTitle;
    String author;
    Date rentDate;
    Date dueDate;
    double rentalFee;
    boolean isOverdue() {
        Date now=new Date();
        return dueDate.before(now);
    }
    double getTotalFee() {
        return isOverdue() ? 1.2*rentalFee : rentalFee;
    }
}
class MovieRental {
    String movieTitle;
    int classification;
    Date rentDate;
    Date dueDate;
    double rentalFee;
    boolean isOverdue() {
        Date now=new Date();
        return dueDate.before(now);
    }
    double getTotalFee() {
        return isOverdue() ? Math.max(1.3*rentalFee, rentalFee+20) : rentalFee;
    }
}
```

12. Point out and remove the code smells in the code:

```
class Customer {
    String homeAddress;
    String workAddress;
}
class Order {
    String orderId;
    Restaurant restaurantReceivingOrder;
    Customer customerPlacingOrder;
    //"H": deliver to home address of the customer.
    //"W": deliver to work address of the customer.
    //"O": deliver to the address specified here.
    String addressType;
    String otherAddress; //address if addressType is "O".
    HashMap orderItems;
```

```

public String getDeliveryAddress() {
    if (addressType.equals("H")) {
        return customerPlacingOrder.getHomeAddress();
    } else if (addressType.equals("W")) {
        return customerPlacingOrder.getWorkAddress();
    } else if (addressType.equals("O")) {
        return otherAddress;
    } else {
        return null;
    }
}
}

```

13. Come up with some code that uses type code and remove it.

Hints

1. Rename q0-q3, LoadData and etc. The SQL statement contains duplicate code. The field names are duplicated.
2. The data folder is duplicated. The setting of hidden is duplicated. Consider removing the if-then-else-if. Consider renaming some names.
3. The code replacing the few special characters with a space is duplicated. Inside that code, the call to replace is duplicated.
4. You must make the code in the body in each loop look identical (in a certain abstraction level):

```

class BookRentals {
    ...
    void printOverdueRentals() {
        int i;
        for (i=0; i<countRentals(); i++) {
            BookRental rental = getRentalAt(i);
            if (rental.isOverdue())
                process the rental;
        }
    }
    int countOverdueRentals() {
        int i, count;
        count=0;
        for (i=0; i<countRentals(); i++)
            if (getRentalAt(i).isOverdue())
                process the rental;
        return count;
    }
}

```

Extract the common code into a separate method. Let the two existing methods call this new method.

```

class BookRentals {
    ...

```

```

void yyy() {
    for (i=0; i<countRentals(); i++) {
        BookRental rental = getRentalAt(i);
        if (rental.isOverdue())
            process the rental;
    }
}
void printOverdueRentals() {
    yyy();
}
int countOverdueRentals() {
    int count;
    yyy();
    return count;
}
}

```

Because there are two implementations for the "process the rental" method, you should create an interface to allow for different implementations:

```

interface XXX {
    void process(Rental rental);
}
class BookRentals {
    ...
    void yyy(XXX obj) {
        for (i=0; i<countRentals(); i++) {
            BookRental rental = getRentalAt(i);
            if (rental.isOverdue())
                obj.process(rental);
        }
    }
    ...
}

```

Then, printOverdueRentals and countOverdueRentals need to provide their respective implementations:

```

class XXX1 implements XXX {
    ...
}
class XXX2 implements XXX {
    ...
}
class BookRentals {
    ...
    void yyy(XXX obj) {
        ...
    }
    void printOverdueRentals() {
        XXX1 obj=new XXX1();
        yyy(obj);
    }
    int countOverdueRentals() {
        int count;
        XXX2 obj=new XXX2();
    }
}

```

```
    yyy(obj);  
    return count;  
}  
}
```

Think of a good names for XXX, XXX1, XXX2 and yyy by considering what they do. If possible, make XXX1 and XXX2 anonymous classes (possible in Java, not in Delphi or C++).

5. Create sub-classes like USDCurrency, RMBCurrency and ESCUDOCurrency. Each sub-class should have a format method.
6. Create sub-classes like FOCPayment, TTPayment and etc. Each sub-class should have a getNominalPayment method. Only some of them should have a getBankName method.
7. Some comments can be removed by making the code as clear as the comments.

You should create a UI class for each payment class such as FOCPaymentUI, TTPaymentUI and etc. They should inherit/implement something like a PaymentUI. The PaymentUI have methods like tryToDisplayPayment(Payment payment) and makePayment(). The tryToDisplayPayment method tries to display the Payment object using the various components (text fields and etc.). The makePayment method uses the contents of those components to create a new Payment object. For example, TTPaymentUI checks if the Payment object is an TTPayment object. If yes, it will display the TTPayment's information in the components and return true. Otherwise, its tryToDisplayPayment method will return false. TTPaymentUI should create these components (including the panel holding the components) by itself. However, some components such as that for the payment date are shared by all UI objects and therefore should not be created by TTPaymentUI. Instead, create these shared components in the EditPaymentDialog and pass them into TTPaymentUI's constructor.

Each UI class should implement the toString method so that we can put the UI objects into the paymentType combo box directly (combo box in Java can contain Objects, not just Strings. Java will call the toString method of each object for visual display).

Using these classes, the long if-then-else-if in the displayPayment method and the makePayment method in the EditPaymentDialog class can be eliminated.

8. The run method will keep growing and growing. To stop it, you need to make the code checking for overheat look identical to the code checking for cooked rice (in a certain abstraction level). Any new code to be added to the run method in the future must also look identical. Consider the JButton and ActionListener in Java.

Sample solutions

1. Point out and remove the code smells in the code (this example was inspired by the one given by YK):

```
class FoodSalesReport {
    int q0; //how many items of rice sold?
    int q1; //how many items of noodle sold?
    int q2; //how many items of drink sold?
    int q3; //how many items of dessert sold?
    void LoadData(Connection conn) {
        PreparedStatement st = conn.prepareStatement("select "+
            "sum(case when foodType=0 then qty else 0 end) as totalQty0, "+
            "sum(case when foodType=1 then qty else 0 end) as totalQty1, "+
            "sum(case when foodType=2 then qty else 0 end) as totalQty2, "+
            "sum(case when foodType=3 then qty else 0 end) as totalQty3 "+
            "from foodSalesTable group by foodType");
        try {
            ResultSet rs = st.executeQuery();
            try {
                rs.next();
                q0 = rs.getInt("totalQty0");
                q1 = rs.getInt("totalQty1");
                q2 = rs.getInt("totalQty2");
                q3 = rs.getInt("totalQty3");
            } finally {
                rs.close();
            }
        } finally {
            st.close();
        }
    }
}
```

The variable names q0-q3 can be made better to eliminate the comments. The sum(...) SQL expression is duplicated.

```
class FoodSalesReport {
    int qtyRiceSold;
    int qtyNoodleSold;
    int qtyDrinkSold;
    int qtyDessertSold;
    void LoadData(Connection conn) {
        String sqlExprList = "";
        for (int i = 0; i <= 3; i++) {
            String separator = (i==0) ? "" : ",";
            sqlExprList = sqlExprList+
                separator+
                getSQLForSoldQtyForFoodType(i);
        }
        PreparedStatement st = conn.prepareStatement("select "+
            sqlExprList+
            "from foodSalesTable group by foodType");
        try {
            ResultSet rs = st.executeQuery();
            try {
```

```

        rs.next();
        qtyRiceSold = rs.getInt("totalQty0");
        qtyNoodleSold = rs.getInt("totalQty1");
        qtyDrinkSold = rs.getInt("totalQty2");
        qtyDessertSold = rs.getInt("totalQty3");
    } finally {
        rs.close();
    }
} finally {
    st.close();
}
}
String getSQLForSoldQtyForFoodType(int foodType) {
    return "sum(case when foodType="+foodType+
        " then qty else 0 end) as totalQty"+foodType;
}
}

```

If you are still concerned with the type code and the similarity between these food types, do something like:

```

class FoodType {
    int typeCode;
    FoodType(int typeCode) {
        ...
    }
    static FoodType foodTypeRice = new FoodType(0);
    static FoodType foodTypeNoodle = new FoodType(1);
    static FoodType foodTypeDrink = new FoodType(2);
    static FoodType foodTypeDessert = new FoodType(3);
    static FoodType knownFoodTypes[] =
        { foodTypeRice, foodTypeNoodle, foodTypeDrink, foodTypeDessert };
}

class FoodSalesReport {
    HashMap foodTypeToQtySold;
    void loadData(Connection conn) {
        FoodType knownFoodTypes[] = FoodType.knownFoodTypes;
        String sqlExprList = "";
        for (int i = 0; i < knownFoodTypes.length; i++) {
            String separator = (i==0) ? "" : ",";
            sqlExprList = sqlExprList+
                separator+
                getSQLForSoldQtyForFoodType(knownFoodTypes[i]);
        }
        PreparedStatement st = conn.prepareStatement("select "+
            sqlExprList+
            "from foodSalesTable group by foodType");
        try {
            ResultSet rs = st.executeQuery();
            try {
                rs.next();
                for (int i = 0; i < knownFoodTypes.length; i++) {
                    FoodType foodType = knownFoodTypes[i];
                    int qty = rs.getInt(getQtyFieldNameForFoodType(foodType));
                    foodTypeToQtySold.put(foodType, new Integer(qty));
                }
            }
        }
    }
}

```

```

        } finally {
            rs.close();
        }
    } finally {
        st.close();
    }
}
static String getQtyFieldNameForFoodType(FoodType foodType) {
    return "totalQty"+foodType.getCode();
}
String getSQLForSoldQtyForFoodType(FoodType foodType) {
    return "sum(case when foodType="+foodType.getCode()+
        " then qty else 0 end) as "+
        getQtyFieldNameForFoodType(foodType);
}
}

```

2. Point out and remove the code smells in the code (this example was inspired by the one given by YK):

```

class SurveyData {
    String path; //save the data to this file.
    boolean hidden; //should the file be hidden?
    //set the path to save the data according to the type of data (t).
    void setSavePath(int t) {
        if (t==0) { //raw data.
            path = "c:/application/data/raw.dat";
            hidden = true;
        } else if (t==1) { //cleaned up data.
            path = "c:/application/data/cleanedUp.dat";
            hidden = true;
        } else if (t==2) { //processed data.
            path = "c:/application/data/processed.dat";
            hidden = true;
        } else if (t==3) { //data ready for publication.
            path = "c:/application/data/publication.dat";
            hidden = false;
        }
    }
}

```

The folder "c:/application/data" is duplicated. The ".dat" is duplicated. The way the path is set is duplicated. The setting of hidden is duplicated. The comments should be turned into code. The type code should be eliminated. Each type code value should be replaced by an object.

```

class SurveyDataType {
    String baseFileName;
    boolean hideDataFile;
    SurveyDataType(String baseFileName, boolean hideDataFile) {
        this.baseFileName = baseFileName;
        this.hideDataFile = hideDataFile;
    }
    String getSavePath() {
        return "c:/application/data/"+baseFileName+".dat";
    }
}

```

```

static SurveyDataType rawDataType =
    new SurveyDataType("raw", true);
static SurveyDataType cleanedUpDataType =
    new SurveyDataType("cleanedUp", true);
static SurveyDataType processedDataType =
    new SurveyDataType("processed", true);
static SurveyDataType publicationDataType =
    new SurveyDataType("publication", false);
}

```

3. Point out and remove the code smells in the code (this example was inspired by the one given by Eugen):

```

class CustomersInDB {
    Connection conn;
    Customer getCustomer(String IDNumber) {
        PreparedStatement st = conn.prepareStatement(
            "select * from customer where ID=?");
        try {
            st.setString(1,
                IDNumber.replace('-', ' ').
                    replace('(', ' ').
                    replace(')', ' ').
                    replace('/', ' '));
            ResultSet rs = st.executeQuery();
            ...
        } finally {
            st.close();
        }
    }
    void addCustomer(Customer customer) {
        PreparedStatement st = conn.prepareStatement(
            "insert into customer values(?, ?, ?, ?)");
        try {
            st.setString(1,
                customer.getIDNumber().replace('-', ' ').
                    replace('(', ' ').
                    replace(')', ' ').
                    replace('/', ' '));
            st.setString(2, customer.getName());
            ...
            st.executeUpdate();
            ...
        } finally {
            st.close();
        }
    }
}

```

The processing of the ID number is duplicated. The calls to replace are similar.

```

class CustomersInDB {
    Connection conn;
    String replaceSymbolsInID(String idNumber) {
        String symbolsToReplace = "-()/";
        for (int i = 0; i < symbolsToReplace.length(); i++) {
            idNumber = idNumber.replace(symbolsToReplace.charAt(i), ' ');
        }
    }
}

```

```

    }
    return idNumber;
}
Customer getCustomer(String IDNumber) {
    PreparedStatement st = conn.prepareStatement(
        "select * from customer where ID=?");
    try {
        st.setString(1, replaceSymbolsInID(IDNumber));
        ResultSet rs = st.executeQuery();
        ...
    } finally {
        st.close();
    }
}
void addCustomer(Customer customer) {
    PreparedStatement st = conn.prepareStatement(
        "insert into customer values(?, ?, ?, ?)");
    try {
        st.setString(1, replaceSymbolsInID(customer.getIDNumber()));
        st.setString(2, customer.getName());
        ...
        st.executeUpdate();
        ...
    } finally {
        st.close();
    }
}
}

```

4. The code below contains duplicate code: the loop in `printOverdueRentals` and in `countOverdueRentals`. If you need to remove this duplication at any cost, how do you do it?

```

class BookRentals {
    Vector rentals;
    int countRentals() {
        return rentals.size();
    }
    BookRental getRentalAt(int i) {
        return (BookRental) rentals.elementAt(i);
    }
    void printOverdueRentals() {
        int i;
        for (i=0; i<countRentals(); i++) {
            BookRental rental = getRentalAt(i);
            if (rental.isOverdue())
                System.out.println(rental.toString());
        }
    }
    int countOverdueRentals() {
        int i, count;
        count=0;
        for (i=0; i<countRentals(); i++)
            if (getRentalAt(i).isOverdue())
                count++;
        return count;
    }
}

```

First, make the code in both loop bodies look identical:

```
class BookRentals {
    ...
    void printOverdueRentals() {
        int i;
        for (i=0; i<countRentals(); i++) {
            BookRental rental = getRentalAt(i);
            if (rental.isOverdue())
                process the rental;
        }
    }
    int countOverdueRentals() {
        int i, count;
        count=0;
        for (i=0; i<countRentals(); i++)
            if (getRentalAt(i).isOverdue())
                process the rental;
        return count;
    }
}
```

As the two loops are identical, extract the loop into a method. Let the two existing methods call this new method:

```
class BookRentals {
    ...
    void yyy() {
        for (int i=0; i<countRentals(); i++) {
            BookRental rental = getRentalAt(i);
            if (rental.isOverdue())
                process the rental;
        }
    }
    void printOverdueRentals() {
        yyy();
    }
    int countOverdueRentals() {
        int count;
        yyy();
        return count;
    }
}
```

To see what is a good name for this method, consider what this method does. From the code in this method, it can be seen that it loops through each rental and process each overdue rental. So, "processOverdueRentals" should be a good name for this method:

```
class BookRentals {
    ...
    void processOverdueRentals() {
        for (int i=0; i<countRentals(); i++) {
            BookRental rental = getRentalAt(i);
            if (rental.isOverdue())
                process the rental;
        }
    }
}
```

```
}  
void printOverdueRentals() {  
    processOverdueRentals();  
}  
int countOverdueRentals() {  
    int count;  
    processOverdueRentals();  
    return count;  
}  
}
```

Because there are two implementations for the "process the rental" method, create an interface to allow for different implementations:

```
interface XXX {  
    void process(BookRental rental);  
}  
class BookRentals {  
    ...  
    void processOverdueRentals(XXX obj) {  
        for (int i=0; i<countRentals(); i++) {  
            BookRental rental = getRentalAt(i);  
            if (rental.isOverdue())  
                obj.process(rental);  
        }  
    }  
    ...  
}
```

To see what is a good name for this interface, consider what this interface does. From the code in this interface, it is clear that it has only one method and that method processes a rental. So, "RentalProcessor" should be a good name for this interface:

```
interface RentalProcessor {  
    void process(BookRental rental);  
}  
class BookRentals {  
    ...  
    void processOverdueRentals(RentalProcessor processor) {  
        for (int i=0; i<countRentals(); i++) {  
            BookRental rental = getRentalAt(i);  
            if (rental.isOverdue())  
                processor.process(rental);  
        }  
    }  
    ...  
}
```

printOverdueRentals and countOverdueRentals need to provide their respective implementations:

```
class RentalPrinter implements RentalProcessor {  
    void process(BookRental rental) {  
        System.out.println(rental.toString());  
    }  
}
```

```

}
class RentalCounter implements RentalProcessor {
    int count = 0;
    void process(BookRental rental) {
        count++;
    }
}
class BookRentals {
    ...
    void processOverdueRentals(RentalProcessor processor) {
        ...
    }
    void printOverdueRentals() {
        RentalPrinter rentalPrinter=new RentalPrinter();
        processOverdueRentals(rentalPrinter);
    }
    int countOverdueRentals() {
        RentalCounter rentalCounter=new RentalCounter();
        processOverdueRentals(rentalCounter);
        return rentalCounter.count;
    }
}

```

Use anonymous inner class or inner class if possible:

```

class BookRentals {
    ...
    void printOverdueRentals() {
        processOverdueRentals(new RentalProcessor() {
            void process(BookRental rental) {
                System.out.println(rental.toString());
            }
        });
    }
    int countOverdueRentals() {
        class RentalCounter implements RentalProcessor {
            int count = 0;
            void process(BookRental rental) {
                count++;
            }
        }
        RentalCounter rentalCounter = new RentalCounter();
        processOverdueRentals(rentalCounter);
        return rentalCounter.count;
    }
}

```

5. Point out and remove the code smells in the code:

```

class Currency {
    final public int USD=0;
    final public int RMB=1;
    final public int ESCUDO=2; //Portuguese currency
    private int currencyCode;
    public Currency(int currencyCode) {
        this.currencyCode=currencyCode;
    }
}

```

```
public String format(int amount) {
    switch (currencyCode) {
        case USD:
            //return something like $1,200
        case RMB:
            //return something like RMB1,200
        case ESCUDO:
            //return something like $1.200
    }
}
```

The use of type code is bad. Convert each type code into a class.

```
interface Currency {
    public String format(int amount);
}
class USDCurrency implements Currency {
    public String format(int amount) {
        //return something like $1,200
    }
}
class RMBCurrency implements Currency {
    public String format(int amount) {
        //return something like RMB1,200
    }
}
class ESCUDOCurrency implements Currency {
    public String format(int amount) {
        //return something like $1.200
    }
}
```

6. Point out and remove the code smells in the code:

```
class Payment {
    final static String FOC = "FOC"; //free of charge.
    final static String TT = "TT"; //paid by telegraphic transfer.
    final static String CHEQUE = "Cheque"; //paid by cheque.
    final static String CREDIT_CARD = "CreditCard"; //paid by credit card.
    final static String CASH = "Cash"; //paid by cash.
    //type of payment. Must be one of the above constant.
    String paymentType;
    Date paymentDate; //if FOC, the date the fee is waived.
    int actualPayment; //if FOC, it is 0.
    int discount; //if FOC, the amount that is waived.
    String bankName; //if it is by TT, cheque or credit card.
    String chequeNumber; //if it is by cheque.
    //if it is by credit card.
    String creditCardType;
    String creditCardHolderName;
    String creditCardNumber;
    Date creditCardExpiryDate;
    int getNominalPayment() {
        return actualPayment+discount;
    }
    String getBankName() {
```

```
    if (paymentType.equals(TT) ||
        paymentType.equals(CHEQUE) ||
        paymentType.equals(CREDIT_CARD)) {
        return bankName;
    }
    else {
        throw new Exception("bank name is undefined for this payment type");
    }
}
}
```

The use of type code is bad. Most attributes are not always used. We should convert each type code into a class. The actualAmount and discount are duplicated in several payment types, so extract them to form a common parent class. Some comments can be turned into code.

```
abstract class Payment {
    Date paymentDate;
    abstract int getNominalPayment();
}
class FOCPayment extends Payment {
    int amountWaived;
    int getNominalPayment() {
        return amountWaived;
    }
}
class RealPayment extends Payment {
    int actualPayment;
    int discount;
    int getNominalPayment() {
        return actualPayment+discount;
    }
}
class TTPayment extends RealPayment {
    String bankName;
    String getBankName() {
        return bankName;
    }
}
class ChequePayment extends RealPayment {
    String bankName;
    String chequeNumber;
    String getBankName() {
        return bankName;
    }
}
class CreditCardPayment extends RealPayment {
    String bankName;
    String cardType;
    String cardHolderName;
    String cardNumber;
    Date expiryDate;
    String getBankName() {
        return bankName;
    }
}
```

```
}  
class CashPayment extends RealPayment {  
}
```

Note that the bankName is still duplicated in several payment types. We may extract it to create a parent class, but it is quite difficult to think of a good name for this parent class (BankPayment? PaymentThroughBank?). As it is just a single variable, the motivation to extract it to form a class is not very strong either. So it is up to you to decide to extract it or not.

7. In the above application, there is a dialog to allow the user edit a payment object. The user can choose the payment type in a combo box. Then the related components for that payment type will be displayed. This function is provided by the CardLayout in Java. The code is shown below. Update the code accordingly.

```
class EditPaymentDialog extends JDialog {  
    Payment newPayment; //new payment to be returned.  
    JPanel sharedPaymentDetails;  
    JPanel uniquePaymentDetails;  
    JTextField paymentDate;  
    JComboBox paymentType;  
    JTextField discountForFOC;  
    JTextField bankNameForTT;  
    JTextField actualAmountForTT;  
    JTextField discountForTT;  
    JTextField bankNameForCheque;  
    JTextField chequeNumberForCheque;  
    JTextField actualAmountForCheque;  
    JTextField discountForCheque;  
    ...  
  
    EditPaymentDialog() {  
        //setup the components.  
        Container contentPane = getContentPane();  
        String comboBoxItems[] = { //available payment types.  
            Payment.FOC,  
            Payment.TT,  
            Payment.CHEQUE,  
            Payment.CREDIT_CARD,  
            Payment.CASH  
        };  
        //setup the components for the details shared by all types of payment.  
        sharedPaymentDetails = new JPanel();  
        paymentDate = new JTextField();  
        paymentType = new JComboBox(comboBoxItems);  
        sharedPaymentDetails.add(paymentDate);  
        sharedPaymentDetails.add(paymentType);  
        contentPane.add(sharedPaymentDetails, BorderLayout.NORTH);  
        //setup the unique components for each type of payment.  
        uniquePaymentDetails = new JPanel();  
        uniquePaymentDetails.setLayout(new CardLayout());  
        //setup a panel for FOC payment.  
        JPanel panelForFOC = new JPanel();  
        discountForFOC = new JTextField();  
        panelForFOC.add(discountForFOC);  
        uniquePaymentDetails.add(panelForFOC, Payment.FOC);  
    }  
}
```

```

//setup a panel for TT payment.
JPanel panelForTT = new JPanel();
bankNameForTT = new JTextField();
actualAmountForTT = new JTextField();
discountForTT = new JTextField();
panelForTT.add(bankNameForTT);
panelForTT.add(actualAmountForTT);
panelForTT.add(discountForTT);
uniquePaymentDetails.add(panelForTT, Payment.TT);
//setup a panel for cheque payment.
JPanel panelForCheque = new JPanel();
bankNameForCheque = new JTextField();
chequeNumberForCheque = new JTextField();
actualAmountForCheque = new JTextField();
discountForCheque = new JTextField();
panelForCheque.add(bankNameForCheque);
panelForCheque.add(chequeNumberForCheque);
panelForCheque.add(actualAmountForCheque);
panelForCheque.add(discountForCheque);
uniquePaymentDetails.add(panelForCheque, Payment.CHEQUE);
//setup a panel for credit card payment.
...
//setup a panel for cash payment.
...
contentPane.add(uniquePaymentDetails, BorderLayout.CENTER);
}
Payment editPayment(Payment payment) {
    displayPayment(payment);
    setVisible(true);
    return newPayment;
}
void displayPayment(Payment payment) {
    paymentDate.setText(payment.getDateAsString());
    paymentType.setSelectedItem(payment.getType());
    if (payment.getType().equals(Payment.FOC)) {
        discountForFOC.setText(Integer.toString(payment.getDiscount()));
    }
    else if (payment.getType().equals(Payment.TT)) {
        bankNameForTT.setText(payment.getBankName());
        actualAmountForTT.setText(
            Integer.toString(payment.getActualAmount()));
        discountForTT.setText(Integer.toString(payment.getDiscount()));
    }
    else if (payment.getType().equals(Payment.CHEQUE)) {
        bankNameForCheque.setText(payment.getBankName());
        chequeNumberForCheque.setText(payment.getChequeNumber());
        actualAmountForCheque.setText(
            Integer.toString(payment.getActualAmount()));
        discountForCheque.setText(Integer.toString(payment.getDiscount()));
    }
    else if (payment.getType().equals(Payment.CREDIT_CARD)) {
        //...
    }
    else if (payment.getType().equals(Payment.CASH)) {
        //...
    }
}
//when the user clicks OK.
void onOK() {
    newPayment = makePayment();
}

```

```

        dispose();
    }
    //make a payment from the components.
    Payment makePayment() {
        String paymentTypeString = (String) paymentType.getSelectedItemAt();
        Payment payment = new Payment(paymentTypeString);
        payment.setDateAsText(paymentDate.getText());
        if (paymentTypeString.equals(Payment.FOC)) {
            payment.setDiscount(Integer.parseInt(discountForFOC.getText()));
        }
        else if (paymentTypeString.equals(Payment.TT)) {
            payment.setBankName(bankNameForTT.getText());
            payment.setActualAmount(
                Integer.parseInt(actualAmountForTT.getText()));
            payment.setDiscount(
                Integer.parseInt(discountForTT.getText()));
        }
        else if (paymentTypeString.equals(Payment.CHEQUE)) {
            payment.setBankName(bankNameForCheque.getText());
            payment.setChequeNumber(chaqueNumberForCheque.getText());
            payment.setActualAmount(
                Integer.parseInt(actualAmountForCheque.getText()));
            payment.setDiscount(
                Integer.parseInt(discountForCheque.getText()));
        }
        else if (paymentTypeString.equals(Payment.CREDIT_CARD)) {
            //...
        }
        else if (paymentTypeString.equals(Payment.CASH)) {
            //...
        }
        return payment;
    }
}

```

Some comments can be turned into code. The long if-then-else-if in the displayPayment method and the makePayment method in the EditPaymentDialog class are no good.

To eliminate these if-then-else-ifs, we should create a UI class for each payment class such as FOCPaymentUI, TTPaymentUI and etc. They should inherit/implement something like a PaymentUI. The PaymentUI have methods like tryToShowPayment (Payment payment) and makePayment(). The PaymentUI constructor should take the paymentDate JTextField as an argument because all sub-classes need it. Each sub-class should implement the toString method so that we can replace the String array by a PaymentUI array for the paymentType combo box (combo box in java can contain Objects, not just Strings).

```

abstract class PaymentUI {
    JTextField paymentDate;
    PaymentUI(JTextField paymentDate) {
        ...
    }
    void displayPaymentDate(Payment payment) {
        paymentDate.setText(
            new SimpleDateFormat().format(payment.getPaymentDate()));
    }
}

```

```

    }
    Date makePaymentDate() {
        //parse the text in paymentDate and return a date;
    }
    abstract boolean tryToDisplayPayment(Payment payment);
    abstract Payment makePayment();
    abstract JPanel getPanel();
}
abstract class RealPaymentUI extends PaymentUI {
    JTextField actualPayment;
    JTextField discount;
    RealPaymentUI(JTextField paymentDate) {
        super(paymentDate);
        actualPayment = new JTextField();
        discount = new JTextField();
    }
    void displayActualPayment(RealPayment payment) {
        actualPayment.setText(Integer.toString(payment.getActualPayment()));
    }
    void displayDiscount(RealPayment payment) {
        discount.setText(Integer.toString(payment.getDiscount()));
    }
    int makeActualPayment() {
        //parse the text in actualPayment and return an int;
    }
    int makeDiscount() {
        //parse the text in discount and return an int;
    }
}
class TTPaymentUI extends RealPaymentUI {
    JPanel panel;
    JTextField bankName;
    TTPaymentUI(JTextField paymentDate) {
        super(paymentDate);
        panel = ...;
        bankName = ...;
        //add bankName, actualPayment, discount to panel.
    }
    boolean tryToDisplayPayment(Payment payment) {
        if (payment instanceof TTPayment) {
            TTPayment ttpayment = (TTPayment)payment;
            displayPaymentDate(payment);
            displayActualPayment(ttpayment);
            displayDiscount(ttpayment);
            bankName.setText(ttpayment.getBankName());
            return true;
        }
        return false;
    }
    Payment makePayment() {
        return new TTPayment(makePaymentDate(),
            makeActualPayment(),
            makeDiscount(),
            bankName.getText());
    }
    String toString() {

```

```
        return "TT";
    }
    JPanel getPanel() {
        return panel;
    }
}
class FOCPaymentUI extends PaymentUI {
    ...
}
class ChequePaymentUI extends RealPaymentUI {
    ...
}
class EditPaymentDialog extends JDialog {
    Payment newPaymentToReturn;
    JPanel sharedPaymentDetails;
    JTextField paymentDate;
    JComboBox paymentType;
    PaymentUI paymentUIs[];

    EditPaymentDialog() {
        setupComponents();
    }
    void setupComponents() {
        setupComponentsSharedByAllPaymentTypes();
        setupPaymentUIs();
        setupPaymentTypeIndicator();
        setupComponentsUniqueToEachPaymentType();
    }
    void setupComponentsSharedByAllPaymentTypes() {
        paymentDate = new JTextField();
        sharedPaymentDetails = new JPanel();
        sharedPaymentDetails.add(paymentDate);
        Container contentPane = getContentPane();
        contentPane.add(sharedPaymentDetails, BorderLayout.NORTH);
    }
    void setupPaymentUIs() {
        paymentUIs[0] = new TTPaymentUI(paymentDate);
        paymentUIs[1] = new FOCPaymentUI(paymentDate);
        paymentUIs[2] = new ChequePaymentUI(paymentDate);
        ...
    }
    void setupPaymentTypeIndicator() {
        paymentType = new JComboBox(paymentUIs);
        sharedPaymentDetails.add(paymentType);
    }
    void setupComponentsUniqueToEachPaymentType() {
        JPanel uniquePaymentDetails = new JPanel();
        uniquePaymentDetails.setLayout(new CardLayout());
        for (int i = 0; i < paymentUIs.length; i++) {
            PaymentUI UI = paymentUIs[i];
            uniquePaymentDetails.add(UI.getPanel(), UI.toString());
        }
        Container contentPane = getContentPane();
        contentPane.add(uniquePaymentDetails, BorderLayout.CENTER);
    }
    Payment editPayment(Payment payment) {
        displayPayment(payment);
    }
}
```

```

        setVisible(true);
        return newPaymentToReturn;
    }
    void displayPayment(Payment payment) {
        for (int i = 0; i < paymentUIs.length; i++) {
            PaymentUI UI = paymentUIs[i];
            if (UI.tryToDisplayPayment(payment)) {
                paymentType.setSelectedItem(UI);
            }
        }
    }
    void onOK() {
        newPaymentToReturn = makePayment();
        dispose();
    }
    Payment makePayment() {
        PaymentUI UI = (PaymentUI)paymentType.getSelectedItem();
        return UI.makePayment();
    }
}

```

8. This is an embedded application controlling a cooker. In every second, it will check if the cooker is over-heated (e.g., short-circuited). If yes it will cut itself off the power and make an alarm using its built-in speaker. It will also check if the moisture inside is lower than a certain threshold (e.g., the rice is cooked). If yes, it will turn its built-in heater to 50 degree Celsius just to keep the rice warm. In the future, you expect that some more things will need to be done in every second.

Point out and remove the problem in the code.

```

class Scheduler extends Thread {
    Alarm alarm;
    HeatSensor heatSensor;
    PowerSupply powerSupply;
    MoistureSensor moistureSensor;
    Heater heater;
    public void run() {
        for (;;) {
            Thread.sleep(1000);
            //check if it is overheated.
            if (heatSensor.isOverHeated()) {
                powerSupply.turnOff();
                alarm.turnOn();
            }
            //check if the rice is cooked.
            if (moistureSensor.getMoisture() < 60) {
                heater.setTemperature(50);
            }
        }
    }
}

```

The run method will keep growing and growing. To stop it, we should make the code checking for overheat look identical to the code checking for cooked rice (in a certain abstraction level):

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

```
class Scheduler extends Thread {
    public void run() {
        for (;;) {
            Thread.sleep(1000);
            //check if it is overheated.
            do some task;
            //check if the rice is cooked.
            do some task;
        }
    }
}
```

Define an interface such as Task and two implementation classes doing the task differently:

```
interface Task {
    void doIt();
}
class Scheduler extends Thread {
    Task tasks[];
    void registerTask(Task task) {
        //add task to tasks;
    }
    public void run() {
        for (;;) {
            Thread.sleep(1000);
            for (int i = 0; i < tasks.length; i++) {
                tasks[i].doIt();
            }
        }
    }
}
class OverHeatCheckTask implements Task {
    Alarm alarm;
    PowerSupply powerSupply;
    HeatSensor heatSensor;
    void doIt() {
        if (heatSensor.isOverHeated()) {
            powerSupply.turnOff();
            alarm.turnOn();
        }
    }
}
class RickCookedCheckTask implements Task {
    Heater heater;
    MoistureSensor moistureSensor;
    void doIt() {
        if (moistureSensor.getMoisture() < 60) {
            heater.setTemperature(50);
        }
    }
}
class Cooker {
    public static void main(String args[]) {
        Scheduler scheduler = new Scheduler();
        scheduler.registerTask(new OverHeatCheckTask());
        scheduler.registerTask(new RickCookedCheckTask());
    }
}
```

```

    }
}

```

9. This application is concerned with the training courses. The schedule of a course can be expressed in three ways (as of now): weekly, range or list. A weekly schedule is like "every Tuesday for 5 weeks starting from Oct. 22". A range schedule is like "Every day from Oct. 22 to Nov. 3". A list schedule is like "Oct. 22, Oct. 25, Nov. 3, Nov. 10". In this exercise we will ignore the time and just assume that it is always 7:00pm-10:00pm. It is expected that new ways to express the schedule may be added in the future.

Point out and remove the code smells in the code:

```

class Course {
    static final int WEEKLY=0;
    static final int RANGE=1;
    static final int LIST=2;
    String courseTitle;
    int scheduleType; // WEEKLY, RANGE, or LIST
    int noWeeks; // For WEEKLY.
    Date fromDate; // for WEEKLY and RANGE.
    Date toDate; // for RANGE.
    Date dateList[]; // for LIST.

    int getDurationInDays() {
        switch (scheduleType) {
            case WEEKLY:
                return noWeeks;
            case RANGE:
                int msInOneDay = 24*60*60*1000;
                return (int)((toDate.getTime()-fromDate.getTime())/msInOneDay);
            case LIST:
                return dateList.length;
            default:
                return 0; // unknown schedule type!
        }
    }
    void printSchedule() {
        switch (scheduleType) {
            case WEEKLY:
                //...
            case RANGE:
                //...
            case LIST:
                //...
        }
    }
}

```

The use of type code is bad. Some instance variables are not used all the time. We should turn each type code value into a sub-class. We are talking about different types of schedules, not really different types of courses.

```

interface Schedule {
    int getDurationInDays();
    void print();
}

```

```

}
class WeeklySchedule implements Schedule {
    int noWeeks;
    Date fromDate;
    Date toDate;
    int getDurationInDays() {
        return noWeeks;
    }
    void print() {
        ...
    }
}
class RangeSchedule implements Schedule {
    Date fromDate;
    Date toDate;
    int getDurationInDays() {
        int msInOneDay = 24*60*60*1000;
        return (int)((toDate.getTime()-fromDate.getTime())/msInOneDay);
    }
    void print() {
        ...
    }
}
class ListSchedule implements Schedule {
    Date dateList[];
    int getDurationInDays() {
        return dateList.length;
    }
    void print() {
        ...
    }
}
class Course {
    String courseTitle;
    Schedule schedule;
    int getDurationInDays() {
        return schedule.getDurationInDays();
    }
    void printSchedule() {
        schedule.print();
    }
}
}

```

10. This application is concerned with training courses. A course has a title, a fee and a list of sessions. However, sometimes a course can consist of several modules, each of which is a course. For example, there may be a compound course "Fast track to becoming a web developer" which consists of three modules: a course named "HTML", a course named "FrontPage" and a course named "Flash". It is possible for a module to consist of some other modules. If a course consists of modules, its fee and schedule are totally determined by that of its modules and thus it will not maintain its list of sessions.

Point out and remove the code smells in the code:

```

class Session {
    Date date;

```

```
int startHour;
int endHour;
int getDuration() {
    return endHour-startHour;
}
}
class Course {
    String courseTitle;
    Session sessions[];
    double fee;
    Course modules[];

    Course(String courseTitle, double fee, Session sessions[]) {
        //...
    }
    Course(String courseTitle, Course modules[]) {
        //...
    }
    String getTitle() {
        return courseTitle;
    }
    double getDuration() {
        int duration=0;
        if (modules==null)
            for (int i=0; i<sessions.length; i++)
                duration += sessions[i].getDuration();
        else
            for (int i=0; i<modules.length; i++)
                duration += modules[i].getDuration();
        return duration;
    }
    double getFee() {
        if (modules==null)
            return fee;
        else {
            double totalFee = 0;
            for (int i=0; i<modules.length; i++)
                totalFee += modules[i].getFee();
            return totalFee;
        }
    }
    void setFee(int fee) throws Exception {
        if (modules==null)
            this.fee = fee;
        else
            throw new Exception("Please set the fee of each module one by one");
    }
}
```

Some instance variables are not used all the time. There are actually two types of courses: those containing modules and those not containing modules. The if-then-else type checking is duplicated in many methods. We should separate each type into a sub-class. Some methods like `setFee` is applicable to one type only and should be declared in that sub-class.

```
class Session {
    Date date;
    int startHour;
```

```
int endHour;
int getDuration() {
    return endHour-startHour;
}
}
abstract class Course {
    String courseTitle;
    Course(String courseTitle) {
        ...
    }
    String getTitle() {
        return courseTitle;
    }
    abstract double getFee();
    abstract double getDuration();
}
class SimpleCourse extends Course {
    Session sessions[];
    double fee;
    SimpleCourse(String courseTitle, double fee, Session sessions[]) {
        ...
    }
    double getFee() {
        return fee;
    }
    double getDuration() {
        int duration=0;
        for (int i=0; i<sessions.length; i++) {
            duration += sessions[i].getDuration();
        }
        return duration;
    }
    void setFee(int fee) {
        this.fee = fee;
    }
}
class CompoundCourse extends Course {
    Course modules[];
    CompoundCourse(String courseTitle, Course modules[]) {
        ...
    }
    double getFee() {
        double totalFee = 0;
        for (int i=0; i<modules.length; i++) {
            totalFee += modules[i].getFee();
        }
        return totalFee;
    }
    double getDuration() {
        int duration=0;
        for (int i=0; i<modules.length; i++) {
            duration += modules[i].getDuration();
        }
        return duration;
    }
}
}
```

11. Point out and remove the code smells in the code:

```
class BookRental {
    String bookTitle;
    String author;
    Date rentDate;
    Date dueDate;
    double rentalFee;
    boolean isOverdue() {
        Date now=new Date();
        return dueDate.before(now);
    }
    double getTotalFee() {
        return isOverdue() ? 1.2*rentalFee : rentalFee;
    }
}
class MovieRental {
    String movieTitle;
    int classification;
    Date rentDate;
    Date dueDate;
    double rentalFee;
    boolean isOverdue() {
        Date now=new Date();
        return dueDate.before(now);
    }
    double getTotalFee() {
        return isOverdue() ? Math.max(1.3*rentalFee, rentalFee+20) : rentalFee;
    }
}
```

Some instance variables and some methods are duplicated in both classes. They should be extracted into a parent class like `Rental`. The `getTotalFee` method in both classes have the same structure. Make them look identical and then extract it into the parent class.

```
abstract class Rental {
    Date rentDate;
    Date dueDate;
    double rentalFee;
    boolean isOverdue() {
        Date now=new Date();
        return dueDate.before(now);
    }
    double getTotalFee() {
        return isOverdue() ? getFeeWhenOverdue() : rentalFee;
    }
    abstract double getFeeWhenOverdue();
}
class BookRental extends Rental {
    String bookTitle;
    String author;
    double getFeeWhenOverdue() {
        return 1.2*rentalFee;
    }
}
class MovieRental extends Rental {
    String movieTitle;
    int classification;
    double getFeeWhenOverdue() {
```

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

```

    return Math.max(1.3*rentalFee, rentalFee+20);
}
}

```

12. Point out and remove the code smells in the code:

```

class Customer {
    String homeAddress;
    String workAddress;
}
class Order {
    String orderId;
    Restaurant restaurantReceivingOrder;
    Customer customerPlacingOrder;
    //"H": deliver to home address of the customer.
    //"W": deliver to work address of the customer.
    //"O": deliver to the address specified here.
    String addressType;
    String otherAddress; //address if addressType is "O".
    HashMap orderItems;

    public String getDeliveryAddress() {
        if (addressType.equals("H")) {
            return customerPlacingOrder.getHomeAddress();
        } else if (addressType.equals("W")) {
            return customerPlacingOrder.getWorkAddress();
        } else if (addressType.equals("O")) {
            return otherAddress;
        } else {
            return null;
        }
    }
}
}

```

The Order class contains a type code. The instance variables otherAddress is not always used. We should turn each type code value into a class:

```

class Customer {
    String homeAddress;
    String workAddress;
}
interface DeliveryAddress {
}
class HomeAddress implements DeliveryAddress {
    Customer customer;
    HomeAddress(Customer customer) {
        ...
    }
    String toString() {
        return customer.getHomeAddress();
    }
}
class WorkAddress implements DeliveryAddress {
    Customer customer;
    WorkAddress(Customer customer) {
        ...
    }
}

```

```
String toString() {
    return customer.getWorkAddress();
}
}
class SpecifiedAddress implements DeliveryAddress {
    String addressSpecified;
    SpecifiedAddress(String addressSpecified) {
        ...
    }
    String toString() {
        return addressSpecified;
    }
}
class Order {
    String orderId;
    Restaurant restaurantReceivingOrder;
    Customer customerPlacingOrder;
    DeliveryAddress deliveryAddress;
    HashMap orderItems;

    public String getDeliveryAddress() {
        return deliveryAddress.toString();
    }
}
```