



CHAPTER 5

Take Care to Inherit



Example

This is a conference management system. It is used to manage various types of information of conference participants. Each participant corresponds to a record in the Participants table in the database. Sometimes the users delete a participant by mistake. Therefore, when deleting a participant, the system simply sets a delete flag in the record to true without actually deleting it. The system will delete all the records whose delete flag is true within 24 hours. If the users change their mind before 24 hours, the system can bring back the participant by setting the delete flag to false.

Please read the current code carefully:

```
public class DBTable {
    protected Connection conn;
    protected tableName;
    public DBTable(String tableName) {
        this.tableName = tableName;
        this.conn = ...;
    }
    public void clear() {
        PreparedStatement st = conn.prepareStatement("DELETE FROM "+tableName);
        try {
            st.executeUpdate();
        }finally{
            st.close();
        }
    }
    public int getCount() {
        PreparedStatement st = conn.prepareStatement("SELECT COUNT(*) FROM
"+tableName);
        try {
            ResultSet rs = st.executeQuery();
            rs.next();
            return rs.getInt(1);
        }finally{
            st.close();
        }
    }
}
public class ParticipantsInDB extends DBTable {
    public ParticipantsInDB() {
        super("participants");
    }
    public void addParticipant(Participant part) {
        ...
    }
}
```

```

public void deleteParticipant(String participantId) {
    setDeleteFlag(participantId, true);
}
public void restoreParticipant(String participantId) {
    setDeleteFlag(participantId, false);
}
private void setDeleteFlag(String participantId, boolean b) {
    ...
}
public void reallyDelete() {
    PreparedStatement st = conn.prepareStatement(
        "DELETE FROM "+
        tableName+
        " WHERE deleteFlag=true");

    try {
        st.executeUpdate();
    }finally{
        st.close();
    }
}
public int countParticipants() {
    PreparedStatement st = conn.prepareStatement(
        "SELECT COUNT(*) FROM "+
        tableName+
        " WHERE deleteFlag=false");

    try {
        ResultSet rs = st.executeQuery();
        rs.next();
        return rs.getInt(1);
    }finally{
        st.close();
    }
}
}
}

```

Note that `countParticipants` only counts those records whose `deleteFlags` are false. That is, it does not count the deleted participants.

The above code looks fine but it contains a severe problem. What is the problem? Read the code below:

```

ParticipantsInDB partsInDB = ...;
Participant kent = new Participant(...);
Participant paul = new Participant(...);
partsInDB.clear();
partsInDB.addParticipant(kent);
partsInDB.addParticipant(paul);
partsInDB.deleteParticipant(kent.getId());
System.out.println("There are "+partsInDB.getCount()+ "participants");

```

The last line should print "There are 1 participants", right? No! It will print "There are 2 participants"! This is because the last line calls the `getCount` method in `DBTable`, not the `countParticipants` method in `ParticipantsInDB`. Because `getCount` knows nothing about the delete flag and simply counts records, it doesn't know how to count the effective (not deleted) participants.

Having inherited inappropriate (or useless) features

ParticipantsInDB has inherited the methods of DBTable such as clear and getCount. For ParticipantsInDB, clear should be useful: It deletes all the participants. But getCount makes little sense: For ParticipantsInDB, what getCount does is to count the total number of participants including those effective and those deleted. Generally speaking, nobody should need to know this number. Even if someone does, this method really should not be called getCount, because this name can easily be associated with "counting the number of (effective) participants".

Therefore, ParticipantsInDB should really not inherit this getCount method. What should we do?

Is there really an inheritance relationship?

When we have inherited something that we don't want, we need to think twice: There is really an inheritance relationship between them? Must ParticipantsInDB be a DBTable? Would ParticipantsInDB like the others to know that it is a DBTable?

In fact, ParticipantsInDB represents a set of participants. It can be represented by one, two or three tables in a database. It can also be represented by two tables in two different databases. Therefore, it has no fixed relationship between a database table. Therefore, there is not really an inheritance relationship between them. It means that we should make ParticipantsInDB not to inherit DBTable. As a result, it no longer inherits the getCount method. However, ParticipantsInDB still needs to use the other functions of DBTable, so we let ParticipantsInDB use a DBTable instead:

```
public class DBTable {
    private Connection conn;
    private String tableName;
    public DBTable(String tableName) {
        this.tableName = tableName;
        this.conn = ...;
    }
    public void clear() {
        PreparedStatement st = conn.prepareStatement("DELETE FROM "+tableName);
        try {
            st.executeUpdate();
        }finally{
            st.close();
        }
    }
    public int getCount() {
        PreparedStatement st = conn.prepareStatement("SELECT COUNT(*) FROM
"+tableName);
        try {
            ResultSet rs = st.executeQuery();
            rs.next();
```

```

        return rs.getInt(1);
    }finally{
        st.close();
    }
}
}
public String getTableName() {
    return tableName;
}
public Connection getConn() {
    return conn;
}
}
}
public class ParticipantsInDB {
    private DBTable table;
    public ParticipantsInDB() {
        table = new DBTable("participants");
    }
    public void addParticipant(Participant part) {
        ...
    }
    public void deleteParticipant(String participantId) {
        setDeleteFlag(participantId, true);
    }
    public void restoreParticipant(String participantId) {
        setDeleteFlag(participantId, false);
    }
    private void setDeleteFlag(String participantId, boolean b) {
        ...
    }
    public void reallyDelete() {
        PreparedStatement st = table.getConn().prepareStatement(
            "DELETE FROM "+
            table.getTableName()+
            " WHERE deleteFlag=true");
        try {
            st.executeUpdate();
        }finally{
            st.close();
        }
    }
    public void clear() {
        table.clear();
    }
    public int countParticipants() {
        PreparedStatement st = table.getConn().prepareStatement(
            "SELECT COUNT(*) FROM "+
            table.getTableName()+
            " WHERE deleteFlag=false");
        try {
            ResultSet rs = st.executeQuery();
            rs.next();
            return rs.getInt(1);
        }finally{
            st.close();
        }
    }
}
}
}

```

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

ParticipantsInDB no longer inherits DBTable. Instead, it uses a variable to refer to a DBTable object and calls its methods such as clear, getConn, getTableName and etc. In particular, the clear method of ParticipantsInDB consists of only one statement that simply calls the clear method of DBTable. That is, a ParticipantsInDB object passes on its own work to a DBTable object to finish. This kind of passing is called "delegation".

Now, the original code which contained the bug will not compile:

```
ParticipantsInDB partsInDB = ...;
Participant kent = new Participant(...);
Participant paul = new Participant(...);
partsInDB.clear();
partsInDB.addParticipant(kent);
partsInDB.addParticipant(paul);
partsInDB.deleteParticipant(kent.getId());
//Compile error: there is no getCount method in ParticipantsInDB!
System.out.println("There are "+partsInDB.getCount()+ "participants");
```

In summary, we find that there is not really an inheritance relationship between ParticipantsInDB and DBTable. Then we replace delegation for inheritance to solve the problem. The advantage of delegation is that we can selectively "make public" the methods of DBTable (e.g., the clear method). If we use inheritance, we have no choice but to accept all the public methods of DBTable (clear and getCount) as our own public methods.

Take out non-essential features

Now let's see another example. Suppose that a Component represents a GUI object such as a button or text field. Please read the code below carefully:

```
abstract class Component {
    boolean isVisible;
    int posXInContainer;
    int posYInContainer;
    int width;
    int height;
    ...
    abstract void paint(Graphics graphics);
    void setWidth(int newWidth) {
        ...
    }
    void setHeight(int newHeight) {
        ...
    }
}
class Button extends Component {
    ActionListener listeners[];
    ...
    void paint(Graphics graphics) {
        ...
    }
}
```

```
class Container {
    Component components[];
    void add(Component component) {
        ...
    }
}
```

Suppose that now you would like to write a clock component. It looks like a round clock and will update its hour-hand and minute-hand to display the current time. As it is a component, it should inherit the Component class:

```
class ClockComponent extends Component {
    ...
    void paint(Graphics graphics) {
        //draw the clock showing the time.
    }
}
```

Now we have a problem: It should be round, but it inherits the attributes width and height from Component and the setWidth and setHeight methods. These things are meaningless to a round component.

When we have inherited something that we don't want, we need to think twice: Is there really an inheritance relationship between them? Must ClockComponent be a Component? Would ClockComponent like the others to know that it is a Component?

In contrast to the case of ParticipantsInDB, ClockComponent should indeed be a Component, otherwise it cannot be put into a Container along with some other Components. Therefore, we should indeed use inheritance here (can't use delegation).

As it must inherit Component but doesn't want width, height, setWidth and setHeight, we must take out these four features from Component. In fact, the current situation already proves that these features are not something that all Components must have (at least our ClockComponent doesn't need them). Taking them out is a very reasonable thing to do.

However, if we take out these features from Component, the existing classes like Button will lose the width and height, right? For the Button classes, these features are indeed needed (assuming Buttons must be rectangular).

A possible solution is to create a RectangularComponent to hold these features and let Button inherit RectangularComponent:

```
abstract class Component {
    boolean isVisible;
    int posXInContainer;
    int posYInContainer;
    ...
    abstract void paint(Graphics graphics);
}
abstract class RectangularComponent extends Component {
    int width;
```

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

```
    int height;
    void setWidth(int newWidth) {
        ...
    }
    void setHeight(int newHeight) {
        ...
    }
}
class Button extends RectangularComponent {
    ActionListener listeners[];
    ...
    void paint(Graphics graphics) {
        ...
    }
}
class ClockComponent extends Component {
    ...
    void paint(Graphics graphics) {
        //draw the clock showing the time.
    }
}
}
```

This is not the only possible solution. Another possible solution is to create a `RectangularDimension` class to hold features and let `Button` delegates to it:

```
abstract class Component {
    boolean isVisible;
    int posXInContainer;
    int posYInContainer;
    ...
    abstract void paint(Graphics graphics);
}
class RectangularDimension {
    int width;
    int height;
    void setWidth(int newWidth) {
        ...
    }
    void setHeight(int newHeight) {
        ...
    }
}
class Button extends Component {
    ActionListener listeners[];
    RectangularDimension dim;
    ...
    void paint(Graphics graphics) {
        ...
    }
    void setWidth(int newWidth) {
        dim.setWidth(newWidth);
    }
    void setHeight(int newHeight) {
        dim.setHeight(newHeight);
    }
}
class ClockComponent extends Component {
    ...
}
```

```
void paint(Graphics graphics) {  
    //draw the clock showing the time.  
}  
}
```

Summary

When we would like to let one class inherit another, we need to check carefully: Will the sub-class inherit some features (attributes or methods) that it doesn't want? If yes, we need to think carefully: Is there really an inheritance relationship between them? If no, use delegation. If yes, take out those unwanted features and put them into an appropriate location.

References

- The Liskov Substitution Principle states: We should be able to use an object of a sub-class to replace an object of a base class, without causing any difference to the client code using the base class. Although the main point of this chapter is different from that of the Liskov Substitution Principle, it is still a very good reference:
 - <http://www.objectmentor.com/resources/articles/lsp.pdf>.
 - <http://c2.com/cgi/wiki?LiskovSubstitutionPrinciple>.
- Design By Contract is something related to the Liskov Substitution Principle. It states that we should test our assumption about our code. For the details, see:
 - <http://c2.com/cgi/wiki?DesignByContract>.



Chapter exercises

Introduction

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

Problems

1. The following code contains duplication. If you need to remove the duplication at any cost, how do you do that?

```
interface GridListener {
    void onRowAppended();
    void onRowMoved(int existingIdx, int newIdx);
}
class Grid {
    GridListener listeners[];
    void appendRow() {
        //append a row at the end of the grid.
        for (int i = 0; i < listeners.length; i++) {
            listeners[i].onRowAppended();
        }
    }
    void moveRow(int existingIdx, int newIdx) {
        //move the row.
        for (int i = 0; i < listeners.length; i++) {
            listeners[i].onRowMoved(existingIdx, newIdx);
        }
    }
}
```

2. Java provides a class called "HashMap". You can add a key-value pair to such a map by the "put" method. Later, you can retrieve the value by calling the "get" method and providing the key. The key and value can be any type. It has a "size" method that returns the number of pairs in the map. See the sample code below.

```
HashMap m=new HashMap();
m.put("x", new Integer(123));
m.put("y", "hello");
m.put(new Double(120.33), "hi");
System.out.println(m.get("x")); // print 123
System.out.println(m.get("y")); // print hello
System.out.println(m.size()); // print 3
```

Point out and remove the problem in the code below:

```
public class CourseCatalog extends HashMap {
    public void addCourse(Course c) {
        put(c.getTitle(), c);
    }
}
```

```

public Course findCourse(String title) {
    return (Course) get(title);
}
public int countCourses() {
    return size();
}
}

```

3. Point out and remove the problem in the code below. You are NOT allowed to use the collection classes in Java.

```

public class Node {
    private Node nextNode;
    public Node getNextNode() {
        return nextNode;
    }
    public void setNextNode(Node nextNode) {
        this.nextNode = nextNode;
    }
}
public class LinkList {
    private Node firstNode;
    public void addNode(Node newNode) {
        ...
    }
    public Node getFirstNode() {
        return firstNode;
    }
}
public class Employee extends Node {
    String employeeId;
    String name;
    ...
}
public class EmployeeList extends LinkList {
    public void addEmployee(Employee employee) {
        addNode(employee);
    }
    public Employee getFirstEmployee() {
        return (Employee) getFirstNode();
    }
    ...
}
}

```

4. Suppose that in general a teacher can teach many students. However, a graduate student can be taught by a graduate teacher only. Point out and remove the problem in the code:

```

class Student {
    String studentId;
    ...
}
class Teacher {
    String teacherId;
    Vector studentsTaught;
    public String getId() {
        return teacherId;
    }
    public void addStudent(Student student) {

```

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

```
        studentsTaught.add(student);
    }
}
class GraduateStudent extends Student {
}
class GraduateTeacher extends Teacher {
}
```

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

```
public class Button {
    private Font labelFont;
    private String labelText;
    ...
    public void addActionListener(ActionListener listener) {
        ...
    }
    public void paint(Graphics graphics) {
        //draw the label text on the graphics using the label's font.
    }
}
public class BitmapButton extends Button {
    private Bitmap bitmap;
    public void paint(Graphics graphics) {
        //draw the bitmap on the graphics.
    }
}
```

6. A property file is a regular text file, but its content has a particular format. For example, it may look like:

```
java.security.enforcePolicy=true
java.system.lang=en
conference.abc=10
xyz=hello
```

That is, every line in a property file has a key (a string), then an equal sign and finally a value.

In order to make it easier to create this kind of property file, you have written the following code, using the `FileWriter` and its `write` method in Java:

```
class PropertyFileWriter extends FileWriter {
    PropertyFileWriter(File file) {
        super(file);
    }
    void writeEntry(String key, String value) {
        super.write(key+"="+value);
    }
}
class App {
    void makePropertyFile() {
        PropertyFileWriter fw = new PropertyFileWriter("fl.properties");
        try {
            fw.writeEntry("conference.abc", "10");
            fw.writeEntry("xyz", "hello");
        }
    }
}
```

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

Point out and remove the problem in the code.

7. Come up with some code that misuses inheritance and correct the problem.

Hints

1. Refer to the RentalProcessor example.
2. You don't want to inherit methods like put.
3. You don't want to inherit methods like getNextNode and addNode. Stop Employee from inheriting Node. Create a new class like EmployeeNode. Let EmployeeList contains a LinkedList instead. When adding an Employee, it should create an EmployeeNode object.

Sample solutions

1. The following code contains duplication. If you need to remove the duplication at any cost, how do you do that?

```
interface GridListener {
    void onRowAppended();
    void onRowMoved(int existingIdx, int newIdx);
}
class Grid {
    GridListener listeners[];
    void appendRow() {
        //append a row at the end of the grid.
        for (int i = 0; i < listeners.length; i++) {
            listeners[i].onRowAppended();
        }
    }
    void moveRow(int existingIdx, int newIdx) {
        //move the row.
        for (int i = 0; i < listeners.length; i++) {
            listeners[i].onRowMoved(existingIdx, newIdx);
        }
    }
}
```

First, make the code look identical:

```
interface GridListener {
    void onRowAppended();
    void onRowMoved(int existingIdx, int newIdx);
}
class Grid {
    GridListener listeners[];
    void appendRow() {
        //append a row at the end of the grid.
        for (int i = 0; i < listeners.length; i++) {
            notify listeners[i];
        }
    }
    void moveRow(int existingIdx, int newIdx) {
        //move the row.
        for (int i = 0; i < listeners.length; i++) {
            notify listeners[i];
        }
    }
}
```

Then extract the duplicate code into a method:

```
interface GridListener {
    void onRowAppended();
    void onRowMoved(int existingIdx, int newIdx);
}
class Grid {
    ...
    void appendRow() {
```

```

    //append a row at the end of the grid.
    ???();
}
void moveRow(int existingIdx, int newIdx) {
    //move the row.
    ???();
}
void ???() {
    for (int i = 0; i < listeners.length; i++) {
        notify listeners[i];
    }
}
}

```

To find a good name for the method, see what the code of the method actually does. In this case, It notifies all the listeners one by one. So "notifyListeners" should be a good name:

```

interface GridListener {
    void onRowAppended();
    void onRowMoved(int existingIdx, int newIdx);
}
class Grid {
    ...
    void appendRow() {
        //append a row at the end of the grid.
        notifyListeners();
    }
    void moveRow(int existingIdx, int newIdx) {
        //move the row.
        notifyListeners();
    }
    void notifyListeners() {
        for (int i = 0; i < listeners.length; i++) {
            notify listeners[i];
        }
    }
}
}

```

As there are two ways to implement "notify listeners[i]", we need to create an interface so that later we can create one implementation class for each implementation:

```

interface GridListener {
    void onRowAppended();
    void onRowMoved(int existingIdx, int newIdx);
}
interface ??? {
    void notify(GridListener listener);
}
class Grid {
    ...
    void appendRow() {
        //append a row at the end of the grid.
        notifyListeners();
    }
    void moveRow(int existingIdx, int newIdx) {
        //move the row.
    }
}

```

```

    notifyListeners();
}
void notifyListeners(??? someObject) {
    for (int i = 0; i < listeners.length; i++) {
        someObject.notify(listeners[i]);
    }
}
}

```

To find a good name for this interface, see what this interface does. As it has only one method (notify) which notifies the specified GridListener, "GridListenerNotifier" should be a good name:

```

interface GridListener {
    void onRowAppended();
    void onRowMoved(int existingIdx, int newIdx);
}
interface GridListenerNotifier {
    void notify(GridListener listener);
}
class Grid {
    ...
    void appendRow() {
        //append a row at the end of the grid.
        notifyListeners();
    }
    void moveRow(int existingIdx, int newIdx) {
        //move the row.
        notifyListeners();
    }
    void notifyListeners(GridListenerNotifier notifier) {
        for (int i = 0; i < listeners.length; i++) {
            notifier.notify(listeners[i]);
        }
    }
}

```

Provide a different implementation class for each implementation. Make them inner classes if possible:

```

interface GridListener {
    void onRowAppended();
    void onRowMoved(int existingIdx, int newIdx);
}
interface GridListenerNotifier {
    void notify(GridListener listener);
}
class Grid {
    ...
    void appendRow() {
        //append a row at the end of the grid.
        notifyListeners(new GridListenerNotifier() {
            void notify(GridListener listener) {
                listener.onRowAppended();
            }
        });
    }
}

```

```

void moveRow(final int existingIdx, final int newIdx) {
    //move the row.
    notifyListeners(new GridListenerNotifier() {
        void notify(GridListener listener) {
            listener.onRowMoved(existingIdx, newIdx);
        }
    });
}
void notifyListeners(GridListenerNotifier notifier) {
    for (int i = 0; i < listeners.length; i++) {
        notifier.notify(listeners[i]);
    }
}
}

```

2. Java provides a class called "HashMap". You can add a key-value pair to such a map by the "put" method. Later, you can retrieve the value by calling the "get" method and providing the key. The key and value can be any type. It has a "size" method that returns the number of pairs in the map. See the sample code below.

```

HashMap m=new HashMap();
m.put("x", new Integer(123));
m.put("y", "hello");
m.put(new Double(120.33), "hi");
System.out.println(m.get("x")); // print 123
System.out.println(m.get("y")); // print hello
System.out.println(m.size()); // print 3

```

Point out and remove the problem in the code below:

```

public class CourseCatalog extends HashMap {
    public void addCourse(Course c) {
        put(c.getTitle(), c);
    }
    public Course findCourse(String title) {
        return (Course)get(title);
    }
    public int countCourses() {
        return size();
    }
}

```

CourseCatalog does not want to inherit the put, get and size methods. Yes, it needs to use these methods, but it does not want others to call these methods on it. For example, the following code will cause trouble:

```

CourseCatalog courseCatalog = new CourseCatalog();
courseCatalog.put("Hello", "World");
courseCatalog.findCourse("Hello"); //trouble: "World" is a string, not Course

```

As there is no code that needs to use a CourseCatalog as a HashMap, we should use delegation instead of inheritance:

```

public class CourseCatalog {
    HashMap map;
}

```

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

```
public void addCourse(Course c) {
    map.put(c.getTitle(), c);
}
public Course findCourse(String title) {
    return (Course)map.get(title);
}
public int countCourses() {
    return map.size();
}
}
```

3. Point out and remove the problem in the code below. You are NOT allowed to use the collection classes in Java.

```
public class Node {
    private Node nextNode;
    public Node getNextNode() {
        return nextNode;
    }
    public void setNextNode(Node nextNode) {
        this.nextNode = nextNode;
    }
}
public class LinkedList {
    private Node firstNode;
    public void addNode(Node newNode) {
        ...
    }
    public Node getFirstNode() {
        return firstNode;
    }
}
public class Employee extends Node {
    String employeeId;
    String name;
    ...
}
public class EmployeeList extends LinkedList {
    public void addEmployee(Employee employee) {
        addNode(employee);
    }
    public Employee getFirstEmployee() {
        return (Employee)getFirstNode();
    }
    ...
}
```

EmployeeList does not want to inherit the addNode method. Yes, it needs to use this method, but it does not want others to call this method on it. For example, the following code will cause trouble:

```
EmployeeList employeeList = new EmployeeList();
Node someNode = new Node();
employeeList.addNode(someNode);
employeeList.getFirstEmployee();//trouble: someNode is a Node, not Employee
```

Employee does not want to inherit getNextNode either.

As there is no code that needs to use an `EmployeeList` as a `List`, we should use delegation instead of inheritance:

```
public class Node {
    ...
}
public class LinkedList {
    ...
}
public class Employee {
    String employeeId;
    String name;
    ...
}
public class EmployeeNode extends Node {
    Employee employee;
}
public class EmployeeList {
    LinkedList list;
    public void addEmployee(Employee employee) {
        list.addNode(new EmployeeNode(employee));
    }
    public Employee getFirstEmployee() {
        return ((EmployeeNode)list.getFirstNode()).getEmployee();
    }
    ...
}
```

4. Suppose that in general a teacher can teach many students. However, a graduate student can be taught by a graduate teacher only. Point out and remove the problem in the code:

```
class Student {
    String studentId;
    ...
}
class Teacher {
    String teacherId;
    Vector studentsTaught;
    public String getId() {
        return teacherId;
    }
    public void addStudent(Student student) {
        studentsTaught.add(student);
    }
}
class GraduateStudent extends Student {
}
class GraduateTeacher extends Teacher {
}
```

Currently, it is possible to let a non-graduate teacher teach a graduate student. The problem is caused by the `addStudent` method in the `Teacher` class. This method accepts any `Student`. Is it true that a `Teacher` can teach any `Student`? It is not true. Therefore, this method should not exist here. Instead, we should move it `GraduateTeacher`:

```
class Student {
    String studentId;
    ...
}
class Teacher {
    String teacherId;
    Vector studentsTaught;
    public String getId() {
        return teacherId;
    }
}
class GraduateStudent extends Student {
}
class GraduateTeacher extends Teacher {
    public void addStudent(Student student) {
        studentsTaught.add(student);
    }
}
```

However, as the AddStudent is no longer in the Teacher class, how can the other (non-graduate) teachers teach the non-graduate students? We can create a NonGraduateTeacher class and a NonGraduateStudent class for that:

```
class Student {
    String studentId;
    ...
}
class Teacher {
    String teacherId;
    Vector studentsTaught;
    public String getId() {
        return teacherId;
    }
}
class GraduateStudent extends Student {
}
class GraduateTeacher extends Teacher {
    public void addStudent(Student student) {
        studentsTaught.add(student);
    }
}
class NonGraduateStudent extends Student {
}
class NonGraduateTeacher extends Teacher {
    public void addStudent(NonGraduateStudent student) {
        studentsTaught.add(student);
    }
}
```

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

```
public class Button {
    private Font labelFont;
    private String labelText;
    ...
    public void addActionListener(ActionListener listener) {
        ...
    }
}
```

```

    }
    public void paint(Graphics graphics) {
        //draw the label text on the graphics using the label's font.
    }
}
public class BitmapButton extends Button {
    private Bitmap bitmap;
    public void paint(Graphics graphics) {
        //draw the bitmap on the graphics.
    }
}
}

```

The `labelFont` and `labelText` in the `Button` class make no sense in the `BitmapButton` class. As every button needs a font or some text, they should not be in the `Button` class. Extract them into a new sub-class such as `LabelButton`. The `paint` method in the `Button` class thus should become abstract because there is no more text to draw:

```

public abstract class Button {
    ...
    public void addActionListener(ActionListener listener) {
        ...
    }
    abstract public void paint(Graphics graphics);
}
public class LabelButton extends Button {
    private Font labelFont;
    private String labelText;
    public void paint(Graphics graphics) {
        //draw the label text on the graphics using the label's font.
    }
}
public class BitmapButton extends Button {
    private Bitmap bitmap;
    public void paint(Graphics graphics) {
        //draw the bitmap on the graphics.
    }
}
}

```

6. A property file is a regular text file, but its content has a particular format. For example, it may look like:

```

java.security.enforcePolicy=true
java.system.lang=en
conference.abc=10
xyz=hello

```

That is, every line in a property file has a key (a string), then an equal sign and finally a value.

In order to make it easier to create this kind of property file, you have written the following code, using the `FileWriter` and its `write` method in Java:

```

class PropertyFileWriter extends FileWriter {
    PropertyFileWriter(String path) {
        super(new File(path));
    }
}

```

```
}
void writeEntry(String key, String value) {
    super.write(key+"="+value);
}
}
class App {
    void makePropertyFile() {
        PropertyFileWriter fw = new PropertyFileWriter("fl.properties");
        try {
            fw.writeEntry("conference.abc", "10");
            fw.writeEntry("xyz", "hello");
        } finally {
            fw.close();
        }
    }
}
```

Point out and remove the problem in the code.

PropertyFileWrite does not want to inherit the write method. Yes, it needs to use this method, but it does not want others to call this method on it. For example, the following code will cause trouble:

```
PropertyFileWriter propertyFileWriter = new PropertyFileWriter(...);
propertyFileWriter.write("this is not a valid entry line");
```

As there is no code that needs to use a PropertyFileWriter as a FileWriter, we should use delegation instead of inheritance:

```
class PropertyFileWriter {
    FileWriter fileWriter;
    PropertyFileWriter(String path) {
        fileWriter = new FileWriter(new File(path));
    }
    void writeEntry(String key, String value) {
        fileWriter.write(key+"="+value);
    }
    void close() {
        fileWriter.close();
    }
}
class App {
    void makePropertyFile() {
        PropertyFileWriter fw = new PropertyFileWriter("fl.properties");
        try {
            fw.writeEntry("conference.abc", "10");
            fw.writeEntry("xyz", "hello");
        } finally {
            fw.close();
        }
    }
}
```