

CHAPTER 6

Handling Inappropriate References

Example

This is a small program handling zip files. The user can input the path to the zip file such as c:\f.zip and the paths to the files that he would like to add to the zip file such as c:\f1.doc, c:\f2.doc and etc. in the main frame, then the program will compress f1.doc and f2.doc and create f.zip. When it is compressing each file, the status bar in the main frame will display the related message. For example, when it is compressing c:\f2.doc, the status bar will display "zipping c:\f2.zip".

The current code is:

```
class ZipMainFrame extends Frame {
    StatusBar sb;
    void makeZip() {
        String zipFilePath;
        String srcFilePaths[];
        //setup zipFilePath and srcFilePaths according to the UI.
        ...
        ZipEngine ze = new ZipEngine();
        ze.makeZip(zipFilePath, srcFilePaths, this);
    }
    void setStatusBarText(String statusText) {
        sb.setText(statusText);
    }
}

class ZipEngine {
    void makeZip(String zipFilePath, String srcFilePaths[], ZipMainFrame f) {
        //create zip file at the path.
        ...
        for (int i = 0; i < srcFilePaths.length; i++) {
            //add the file srcFilePaths[i] into the zip file.
            ...
            f.setStatusBarText("Zipping "+srcFilePaths[i]);
        }
    }
}
```

Now, suppose that we would like to reuse the ZipEngine class in another system (e.g., an inventory system) to perform backups. The files it needs to compress (backup) are the internal data files used by the system. Therefore, the paths are hard-coded and do not need to be input by the user. Suppose that it has a main frame too, and it also needs to display in its

status bar the path to each file as it is compressed.

However, there is a problem. We hope to reuse ZipEngine without rewriting the code. However, please read the code of ZipEngine above: It needs a ZipMainFrame to operate. Obviously, any other systems will not have a ZipMainFrame class. Therefore, this ZipEngine class can be used only in this small zip file handling program, but not in all the other systems (including this inventory system).

"Inappropriate" references make code hard to reuse

Because ZipEngine references ZipMainFrame, when we try to reuse ZipEngine, it will drag in ZipMainFrame. Because in the new environment it is impossible to have ZipMainFrame, ZipEngine cannot be reused.

Generally speaking, if a class A references another class B, when we would like to reuse A, A will drag in B. If B references another class C, B will in turn drag in C. When B or C has no meaning in the new environment or cannot exist in the new environment, we cannot reuse A. Therefore, "Inappropriate" references make the code hard to reuse.

In order to reuse ZipEngine, we must first make ZipEngine no longer reference ZipMainFrame. How to do that? Before answering this question, we need to answer another question first: Given any code fragment, how to check if the code contains "inappropriate" references? What is "inappropriate"?

How to check for "inappropriate" references

Method 1

If there are "inappropriate" references in the code, a simplest method to check is: We check if the code contains mutual references or circular references. For example, ZipMainFrame references ZipEngine, and ZipEngine references ZipMainFrame. Circular references is a code smell, suggesting that the references are "inappropriate".

This method is very simple. However, sometimes even there is an "inappropriate" reference, there are still no circular references. Therefore, this method may miss.

Method 2

Another method is subjective: We check the code and ask ourselves: Does it really need what it references? For ZipEngine, does it really need ZipMainFrame? The code of ZipEngine shows that it needs to display a message in the status bar of ZipMainFrame. Does it absolutely need a ZipMainFrame and cannot use anything else? Is there anything that can replace the ZipMainFrame? In fact, it doesn't really absolutely need a ZipMainFrame. All it needs is just a status bar to let it display the message. Therefore, we can change the code to:

```
class ZipEngine {
    void makeZip(String zipFilePath, String srcFilePaths[], StatusBar statusBar) {
        //create zip file at the path.
        ...
        for (int i = 0; i < srcFilePaths.length; i++) {
            //add the file srcFilePaths[i] into the zip file.
            ...
            statusBar.setText("Zipping "+srcFilePaths[i]);
        }
    }
}
```

Now, ZipEngine only references a StatusBar, not a ZipMainFrame. Because StatusBar is much more general (can exist in many more environments) than ZipMainFrame, the reusability of ZipEngine has been increased significantly. However, this method is too subjective. There is no fixed criterion, therefore it is very hard to apply. For example, we may say that ZipEngine doesn't really need a status, all it needs is a service to display messages (not necessarily a status bar).

Method 3

The third method is also subjective: We check the code and ask ourselves: Can it be reused in another environment? If no, we consider it contains "inappropriate" references. For example, we ask: Can ZipEngine be reused in another environment? Because it references ZipMainFrame, it obviously cannot. Therefore, it really should not reference ZipMainFrame. This method is also subjective and is very hard to apply. For example: What is the new environment like? It is totally subject to our imagination without limits.

Method 4

The fourth method is a simpler and objective method. When we find that we need to reuse the code, but cannot do it because it references something, then we consider it contains "inappropriate" references. For example, when we really need to reuse ZipEngine in an inventory system, we find that it cannot be reused. So we consider it contains "inappropriate" references (to ZipMainFrame).

This is a "lazy", "passive" method, but it is effective.

Summary on the checking methods

To check if the code contains "inappropriate" references, there are four methods:

1. Look: Are there circular references?
2. Think: What it really needs?
3. Think: Can it be reused in a new environment?
4. Look: Now, I need to reuse it in this environment, can it be done?

Method 1 and 4 are the easiest to use and therefore recommended for beginners. With more design experience, method 2 and 3 will be easier.

How to make ZipEngine no longer reference ZipMainFrame

Now we will see how to make ZipEngine no longer reference ZipMainFrame. In fact, when we introduced method 2, through contemplation we have found the real need of ZipEngine and found a solution. Because method 2 is relatively hard to apply, we do not use it here. Assume that we use method 4 and find that ZipEngine cannot be reused in a text mode system (there is no status bar, but we would like to display the messages on the screen). How to solve this problem?

Because we cannot reuse ZipEngine, we first use Copy and Paste, and then modify the resulting code:

```
class TextModeApp {
    void makeZip() {
        String zipFilePath;
        String srcFilePaths[];
        ...
        ZipEngine ze = new ZipEngine();
        ze.makeZip(zipFilePath, srcFilePaths);
    }
}
class ZipEngine {
    void makeZip(String zipFilePath, String srcFilePaths[]) {
        //create zip file at the path.
        ...
        for (int i = 0; i < srcFilePaths.length; i++) {
            //add the file srcFilePaths[i] into the zip file.
            ...
        }
    }
}
```

```

        System.out.println("Zipping "+srcFilePaths[i]);
    }
}

```

Compare this code to the original ZipEngine:

```

class ZipEngine {
    void makeZip(String zipFilePath, String srcFilePaths[], ZipMainFrame f) {
        //create zip file at the path.
        ...
        for (int i = 0; i < srcFilePaths.length; i++) {
            //add the file srcFilePaths[i] into the zip file.
            ...
            f.setStatusBarText("Zipping "+srcFilePaths[i]);
        }
    }
}

```

Obviously there is a lot of duplicate code (code smell). To remove the code smell, we need to make these two code fragments identical (on a certain abstraction level). For example, we may change them to:

```

class ZipEngine {
    void makeZip(String zipFilePath, String srcFilePaths[]) {
        //create zip file at the path.
        ...
        for (int i = 0; i < srcFilePaths.length; i++) {
            //add the file srcFilePaths[i] into the zip file.
            ...
            show the message;
        }
    }
}

```

Because this "show the message" method need two different implementations, we need to create an interface declaring this method and then create two implementation classes to implement this interface, each of which will provide one method implementation. Because this method is called "show the message", this interface may be called "MessageDisplay" or "MessageSink" and etc.:

```

interface MessageDisplay {
    void showMessage(String msg);
}

```

ZipEngine is changed to:

```

class ZipEngine {
    void makeZip(String zipFilePath, String srcFilePaths[], MessageDisplay
msgDisplay) {
        //create zip file at the path.
        ...
        for (int i = 0; i < srcFilePaths.length; i++) {
            //add the file srcFilePaths[i] into the zip file.
            ...

```

```

        msgDisplay.showMessage("Zipping "+srcFilePaths[i]);
    }
}

```

The two implementation classes implementing MessageDisplay may be:

```

class ZipMainFrameMessageDisplay implements MessageDisplay {
    ZipMainFrame f;
    ZipMainFrameMessageDisplay(ZipMainFrame f) {
        this.f = f;
    }
    void showMessage(String msg) {
        f.setStatusBarText(msg);
    }
}
class SystemOutMessageDisplay implements MessageDisplay {
    void showMessage(String msg) {
        System.out.println(msg);
    }
}

```

The two systems need to change accordingly:

```

class ZipMainFrame extends Frame {
    StatusBar sb;
    void makeZip() {
        String zipFilePath;
        String srcFilePaths[];
        //setup zipFilePath and srcFilePaths according to the UI.
        ...
        ZipEngine ze = new ZipEngine();
        ze.makeZip(zipFilePath, srcFilePaths, new ZipMainFrameMessageDisplay(this));
    }
    void setStatusBarText(String statusText) {
        sb.setText(statusText);
    }
}
class TextModeApp {
    void makeZip() {
        String zipFilePath;
        String srcFilePaths[];
        ...
        ZipEngine ze = new ZipEngine();
        ze.makeZip(zipFilePath, srcFilePaths, new SystemOutMessageDisplay());
    }
}

```

The improved code

The improved code is shown below. To make the code clearer, we use inner classes in Java:

```

interface MessageDisplay {

```

```
    void showMessage(String msg);
}
class ZipEngine {
    void makeZip(String zipFilePath, String srcFilePaths[], MessageDisplay
msgDisplay) {
        //create zip file at the path.
        ...
        for (int i = 0; i < srcFilePaths.length; i++) {
            //add the file srcFilePaths[i] into the zip file.
            ...
            msgDisplay.showMessage("Zipping "+srcFilePaths[i]);
        }
    }
}
class ZipMainFrame extends Frame {
    StatusBar sb;
    void makeZip() {
        String zipFilePath;
        String srcFilePaths[];
        //setup zipFilePath and srcFilePaths according to the UI.
        ...
        ZipEngine ze = new ZipEngine();
        ze.makeZip(zipFilePath, srcFilePaths, new MessageDisplay() {
            void showMessage(String msg) {
                setStatusBarText(msg);
            }
        });
    }
    void setStatusBarText(String statusText) {
        sb.setText(statusText);
    }
}
class TextModeApp {
    void makeZip() {
        String zipFilePath;
        String srcFilePaths[];
        ...
        ZipEngine ze = new ZipEngine();
        ze.makeZip(zipFilePath, srcFilePaths, new MessageDisplay() {
            void showMessage(String msg) {
                System.out.println(msg);
            }
        });
    }
}
}
```

References

- The Dependency Inversion Principle states: A high level module shouldn't depend on a low level module. If it really happens, we should extract an abstract concept and let them both depend on that concept. For more details, please see:
 - <http://www.objectmentor.com/resources/articles/dip.pdf>.

- <http://c2.com/cgi/wiki?DependencyInversionPrinciple>.



Chapter exercises

Introduction

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

Problems

1. Point out the problem in the code below. Further suppose that you need to reuse the file copier in a text mode file copying application that will display the progress in its text console as an integer. What should you do?

```
class MainApp extends JFrame {
    ProgressBar progressBar;
    void main() {
        FileCopier fileCopier = new FileCopier(this);
        fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));
    }
    void updateProgressBar(int noBytesCopied, int sizeOfSource) {
        progressBar.setPercentage(noBytesCopied*100/sizeOfSource);
    }
}

class FileCopier {
    MainApp app;
    FileCopier(MainApp app) {
        this.app = app;
    }
    void copyFile(File source, File target) {
        int sizeOfSource = (int)source.length();
        for (int i = 0; i < sizeOfSource; ) {
            //read n (<= 512) bytes from source.
            //write n bytes to target.
            i += n;
            app.updateProgressBar(i, sizeOfSource);
        }
    }
}
```

2. In the above case, suppose that you need to develop another text mode file copying application that will display a "*" for each 10% of progress in its text console. What should you do?
3. Point out the problem in the code below. Further suppose that you need to reuse the fax machine code in another application. What should you do?

```
class MainApp {
    String faxNo;
    void main() {
        FaxMachine faxMachine = new FaxMachine(this);
        faxMachine.sendFax("783675", "hello");
    }
}
```

```

}
class FaxMachine {
    MainApp app;
    FaxMachine(MainApp app) {
        this.app = app;
    }
    void sendFax(String toFaxNo, String msg) {
        FaxMachineHardware hardware = ...;
        hardware.setStationId(app.getFaxNo());
        hardware.setRecipientFaxNo(toFaxNo);
        hardware.start();
        try {
            do {
                Graphics graphics = hardware.newPage();
                //draw the msg into the graphics.
            } while (more page is needed);
        } finally {
            hardware.done();
        }
    }
}

```

4. Point out the problem in the code below. Further suppose that you need to reuse the heat sensor code in another application. What should you do?

```

class Cooker {
    HeatSensor heatSensor;
    Speaker speaker;
    void alarm() {
        speaker.setFrequency(Speaker.HIGH_FREQUENCY);
        speaker.turnOn();
    }
}
class HeatSensor {
    Cooker cooker;
    HeatSensor(Cooker cooker) {
        this.cooker = cooker;
    }
    void checkOverHeated() {
        if (isOverHeated()) {
            cooker.alarm();
        }
    }
    boolean isOverHeated() {
        ...
    }
}

```

5. This is a word processor application. It can let the user select the font. Before the user confirms to change the font, it will let the user preview the effect of the change. The current code is shown below. Point out the code smell. If we need to reuse the ChooseFontDialog in a GUI application that doesn't support this preview functionality, how should you change the code?

```

class WordProcessorMainFrame extends JFrame {
    void onChangeFont() {
        ChooseFontDialog chooseFontDialog = new ChooseFontDialog(this);
    }
}

```

```
        if (chooseFontDialog.showOnScreen()) {
            Font newFont = chooseFontDialog.getSelectedFont();
            //show the contents using this new font.
        } else {
            //show the contents using the existing font.
        }
    }
    void previewWithFont(Font font) {
        //show the contents using this preview font.
    }
}
class ChooseFontDialog extends JDialog {
    WordProcessorMainFrame mainFrame;
    Font selectedFont;
    ChooseFontDialog(WordProcessorMainFrame mainFrame) {
        this.mainFrame = mainFrame;
    }
    boolean showOnScreen() {
        ...
    }
    void onSelectedFontChange() {
        selectedFont = getSelectedFontFromUI();
        mainFrame.previewWithFont(selectedFont);
    }
    Font getSelectedFontFromUI() {
        ...
    }
    Font getSelectedFont() {
        return selectedFont;
    }
}
```

6. Come up with some code that contains inappropriate references and correct the problem.

Hints

1. Circular references.
2. No hint for this one.
3. Circular references. No need to create an interface and implementation classes. The differences should be represented using different objects, not classes.

Sample solutions

1. Point out the problem in the code below. Further suppose that you need to reuse the file copier in a text mode file copying application that will display the progress in its text console as an integer. What should you do?

```
class MainApp extends JFrame {
    ProgressBar progressBar;
    void main() {
        FileCopier fileCopier = new FileCopier(this);
        fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));
    }
    void updateProgressBar(int noBytesCopied, int sizeOfSource) {
        progressBar.setPercentage(noBytesCopied*100/sizeOfSource);
    }
}
class FileCopier {
    MainApp app;
    FileCopier(MainApp app) {
        this.app = app;
    }
    void copyFile(File source, File target) {
        int sizeOfSource = (int)source.length();
        for (int i = 0; i < sizeOfSource; ) {
            //read n (<= 512) bytes from source.
            //write n bytes to target.
            i += n;
            app.updateProgressBar(i, sizeOfSource);
        }
    }
}
```

Currently FileCopier depends on MainApp, making it un reusable in this text mode application. To solve this problem, we can make a copy of FileCopier in the text mode application, modify it and then make its code look identical to the FileCopier in MainApp:

```
interface CopyMonitor {
    void updateProgress(int noBytesCopied, int sizeOfSource);
}
class FileCopier {
    CopyMonitor copyMonitor;
    FileCopier(CopyMonitor copyMonitor) {
        this.copyMonitor = copyMonitor;
    }
    void copyFile(File source, File target) {
        int sizeOfSource = (int)source.length();
        for (int i = 0; i < sizeOfSource; ) {
            //read n (<= 512) bytes from source.
            //write n bytes to target.
            i += n;
            copyMonitor.updateProgress(i, sizeOfSource);
        }
    }
}
```

In each of the two applications, we need to create a class to implement the CopyMonitor interface:

```
class MainApp extends JFrame {
    ProgressBar progressBar;
    void main() {
        FileCopier fileCopier = new FileCopier(new CopyMonitor() {
            void updateProgress(int noBytesCopied, int sizeOfSource) {
                updateProgressBar(noBytesCopied, sizeOfSource);
            }
        });
        fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));
    }
    void updateProgressBar(int noBytesCopied, int sizeOfSource) {
        progressBar.setPercentage(noBytesCopied*100/sizeOfSource);
    }
}

class TextApp {
    void main() {
        FileCopier fileCopier = new FileCopier(new CopyMonitor() {
            void updateProgress(int noBytesCopied, int sizeOfSource) {
                System.out.println(noBytesCopied*100/sizeOfSource);
            }
        });
        fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));
    }
}
```

Since both implementation classes need to calculate the percentage, we may just pass the percentage instead:

```
class MainApp extends JFrame {
    ProgressBar progressBar;
    void main() {
        FileCopier fileCopier = new FileCopier(new CopyMonitor() {
            void updateProgress(int completionPercentage) {
                updateProgressBar(completionPercentage);
            }
        });
        fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));
    }
    void updateProgressBar(int percentage) {
        progressBar.setPercentage(percentage);
    }
}

class TextApp {
    void main() {
        FileCopier fileCopier = new FileCopier(new CopyMonitor() {
            void updateProgress(int completionPercentage) {
                System.out.println(completionPercentage);
            }
        });
        fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));
    }
}

interface CopyMonitor {
    void updateProgress(int completionPercentage);
}
```

```

}
class FileCopier {
    CopyMonitor copyMonitor;
    FileCopier(CopyMonitor copyMonitor) {
        this.copyMonitor = copyMonitor;
    }
    void copyFile(File source, File target) {
        int sizeOfSource = (int)source.length();
        for (int i = 0; i < sizeOfSource; ) {
            //read n (<= 512) bytes from source.
            //write n bytes to target.
            i += n;
            copyMonitor.updateProgress(i*100/sizeOfSource);
        }
    }
}

```

2. In the above case, suppose that you need to develop another text mode file copying application that will display a "*" for each 10% of progress in its text console. What should you do?

Just create another class to implement CopyMonitor:

```

class AnotherTextApp {
    void main() {
        FileCopier fileCopier = new FileCopier(new CopyMonitor() {
            int noStarsPrinted = 0;
            void updateProgress(int completionPercentage) {
                int noStarsToPrint = completionPercentage/10;
                while (noStarsPrinted < noStarsToPrint) {
                    System.out.println("*");
                    noStarsToPrint++;
                }
            }
        });
        fileCopier.copyFile(new File("f1.doc"), new File("f2.doc"));
    }
}

```

3. Point out the problem in the code below. Further suppose that you need to reuse the fax machine code in another application. What should you do?

```

class MainApp {
    String faxNo;
    void main() {
        FaxMachine faxMachine = new FaxMachine(this);
        faxMachine.sendFax("783675", "hello");
    }
}
class FaxMachine {
    MainApp app;
    FaxMachine(MainApp app) {
        this.app = app;
    }
    void sendFax(String toFaxNo, String msg) {
        FaxMachineHardware hardware = ...;
    }
}

```

```

hardware.setStationId(app.getFaxNo());
hardware.setRecipientFaxNo(toFaxNo);
hardware.start();
try {
    do {
        Graphics graphics = hardware.newPage();
        //draw the msg into the graphics.
    } while (more page is needed);
} finally {
    hardware.done();
}
}
}

```

Currently FaxMachine depends on MainApp, making it un reusable in another application. Actually, what FaxMachine really needs is just the fax number to serve as the station ID. To solve this problem, we can just pass the fax number instead of the MainApp into the FaxMachine:

```

class MainApp {
    String faxNo;
    void main() {
        FaxMachine faxMachine = new FaxMachine(faxNo);
        faxMachine.sendFax("783675", "hello");
    }
}
class FaxMachine {
    String stationId;
    FaxMachine(String stationId) {
        this.stationId = stationId;
    }
    void sendFax(String toFaxNo, String msg) {
        FaxMachineHardware hardware = ...;
        hardware.setStationId(stationId);
        hardware.setRecipientFaxNo(toFaxNo);
        hardware.start();
        try {
            do {
                Graphics graphics = hardware.newPage();
                //draw the msg into the graphics.
            } while (more page is needed);
        } finally {
            hardware.done();
        }
    }
}

```

4. Point out the problem in the code below. Further suppose that you need to reuse the heat sensor code in another application. What should you do?

```

class Cooker {
    HeatSensor heatSensor;
    Speaker speaker;
    void alarm() {
        speaker.setFrequency(Speaker.HIGH_FREQUENCY);
        speaker.turnOn();
    }
}

```

```

}
class HeatSensor {
    Cooker cooker;
    HeatSensor(Cooker cooker) {
        this.cooker = cooker;
    }
    void checkOverHeated() {
        if (isOverHeated()) {
            cooker.alarm();
        }
    }
    boolean isOverHeated() {
        ...
    }
}

```

Currently HeatSensor depends on Cooker, making it un reusable in another application. Actually, what HeatSensor really needs is just an alarm, not the Cooker. To solve this problem, we can just pass the alarm instead of the Cooker into the HeatSensor:

```

interface Alarm {
    void turnOn();
}
class Cooker {
    HeatSensor heatSensor;
    Speaker speaker;
    Alarm getAlarm() {
        return new Alarm() {
            void turnOn() {
                speaker.setFrequency(Speaker.HIGH_FREQUENCY);
                speaker.turnOn();
            }
        };
    }
}
class HeatSensor {
    Alarm alarm;
    HeatSensor(Alarm alarm) {
        this.alarm = alarm;
    }
    void checkOverHeated() {
        if (isOverHeated()) {
            alarm.turnOn();
        }
    }
    ...
}

```

5. This is a word processor application. It can let the user select the font. Before the user confirms to change the font, it will let the user preview the effect of the change. The current code is shown below. Point out the code smell. If we need to reuse the ChooseFontDialog in a GUI application that doesn't support this preview functionality, how should you change the code?

```

class WordProcessorMainFrame extends JFrame {
    void onChangeFont() {

```

```
        ChooseFontDialog chooseFontDialog = new ChooseFontDialog(this);
        if (chooseFontDialog.showOnScreen()) {
            Font newFont = chooseFontDialog.getSelectedFont();
            //show the contents using this new font.
        } else {
            //show the contents using the existing font.
        }
    }
    void previewWithFont(Font font) {
        //show the contents using this preview font.
    }
}
class ChooseFontDialog extends JDialog {
    WordProcessorMainFrame mainFrame;
    Font selectedFont;
    ChooseFontDialog(WordProcessorMainFrame mainFrame) {
        this.mainFrame = mainFrame;
    }
    boolean showOnScreen() {
        ...
    }
    void onSelectedFontChange() {
        selectedFont = getSelectedFontFromUI();
        mainFrame.previewWithFont(selectedFont);
    }
    Font getSelectedFontFromUI() {
        ...
    }
    Font getSelectedFont() {
        return selectedFont;
    }
}
```

The code smell is the circular references between `WordProcessorMainFrame` and `ChooseFontDialog`. In order to use `ChooseFontDialog` in a GUI application that does not support font change preview, we need to stop `ChooseFontDialog` from referencing `WordProcessorMainFrame`. Actually, what `ChooseFontDialog` really needs is to notify another object of the temporary font change, so that the other object may let the user preview the effect of the font change.

```
interface FontChangeListener {
    void onFontChanged(Font newFont);
}
class WordProcessorMainFrame extends JFrame {
    void onChangeFont() {
        ChooseFontDialog chooseFontDialog = new ChooseFontDialog(
            new FontChangeListener() {
                void onFontChanged(Font newFont) {
                    previewWithFont(newFont);
                }
            });
        if (chooseFontDialog.showOnScreen()) {
            Font newFont = chooseFontDialog.getSelectedFont();
            //show the contents using this new font.
        } else {
            //show the contents using the existing font.
        }
    }
}
```

```
}
void previewWithFont(Font font) {
    //show the contents using this preview font.
}
}
class ChooseFontDialog extends JDialog {
    FontChangeListener fontChangeListener;
    Font selectedFont;
    ChooseFontDialog(FontChangeListener fontChangeListener) {
        this.fontChangeListener = fontChangeListener;
    }
    boolean showOnScreen() {
        ...
    }
    void onSelectedFontChange() {
        selectedFont = getSelectedFontFromUI();
        fontChangeListener.onFontChanged(selectedFont);
    }
    Font getSelectedFontFromUI() {
        ...
    }
    Font getSelectedFont() {
        return selectedFont;
    }
}
```