Object-Oriented Software Engineering

Instructor : Huang, Chuen-Min

Teamwork1 ver.1

Group 2

ID	Name
B10821124	Leo
B10823011	Kousa
B10823015	Debby
B10823016	Kris
B10823018	Bob
B10823024	Michael
B10823029	Leo
B10823031	Andrew
B10823038	Joanne
B11023069	Young

Date 2021/11/3

Pattern and class diagram

First, we will explain all patterns that we have applied to our text editor. Because the original class diagram is huge and hard to interpret, so we will break down the whole class diagram to make sure that it can be better to understand each region of the class diagram.

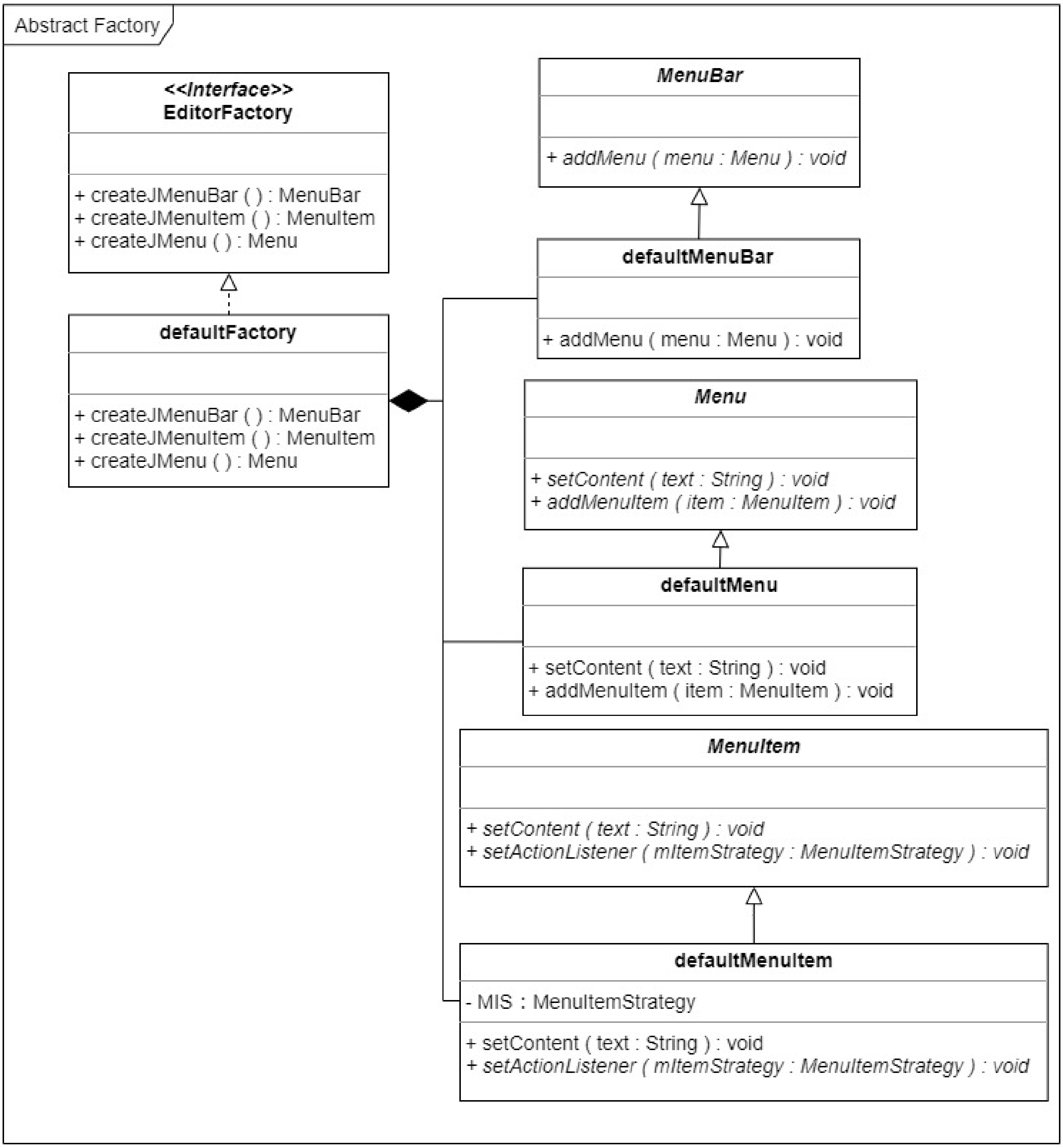
Abstract Factory pattern

Abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.
In our system, we use class “EditorFactory” to create the MenuBar, Menu and MenuItem. Every object is a class, and each object has one corresponding method to create it.

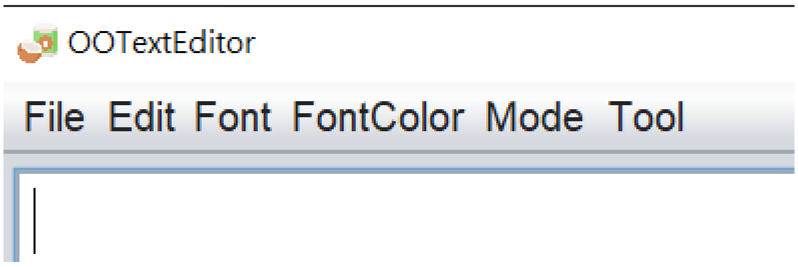
```java
public interface EditorFactory {
    //每個object都是一個類別，也都有一個對應的方法建立
    public MenuBar createJMenuBar();
    public MenuItem creatJMenuItem();
    public Menu createJMenu();
}
```

For those object, class “defaultFactory” provide the default version.

```java
//提供了預設版本
public class defaultFactory implements EditorFactory{
    public MenuBar createJMenuBar() {
        return new defaultMenuBar();
    }
    public MenuItem creatJMenuItem() {
        return new defaultMenuItem();
    }
    public Menu createJMenu() {
        return new defaultMenu();
    }

}
```


Abstract Factory
<<Interface>>
EditorFactory
+ createJMenuBar ( ) : MenuBar
+ createJMenuItem ( ) : MenuItem
+ createJMenu ( ) : Menu
defaultFactory
+ createJMenuBar ( ) : MenuBar
+ createJMenuItem ( ) : MenuItem
+ createJMenu ( ) : Menu
MenuBar
+ addMenu (menu : Menu) : void
defaultMenuBar
+ addMenu (menu : Menu) : void
Menu
+ setContent (text : String) : void
+ addMenuItem (item : MenuItem) : void
defaultMenu
+ setContent (text : String) : void
+ addMenuItem (item : MenuItem) : void
MenuItem
+ setContent (text : String) : void
+ setActionListener (mItemStrategy : MenuItemStrategy) : void
defaultMenuItem
- MIS : MenuItemStrategy
+ setContent (text : String) : void
+ setActionListener (mItemStrategy : MenuItemStrategy) : void

This is the application of the Abstract Factory pattern in our text editor, we use it to create the bar and the menu item.

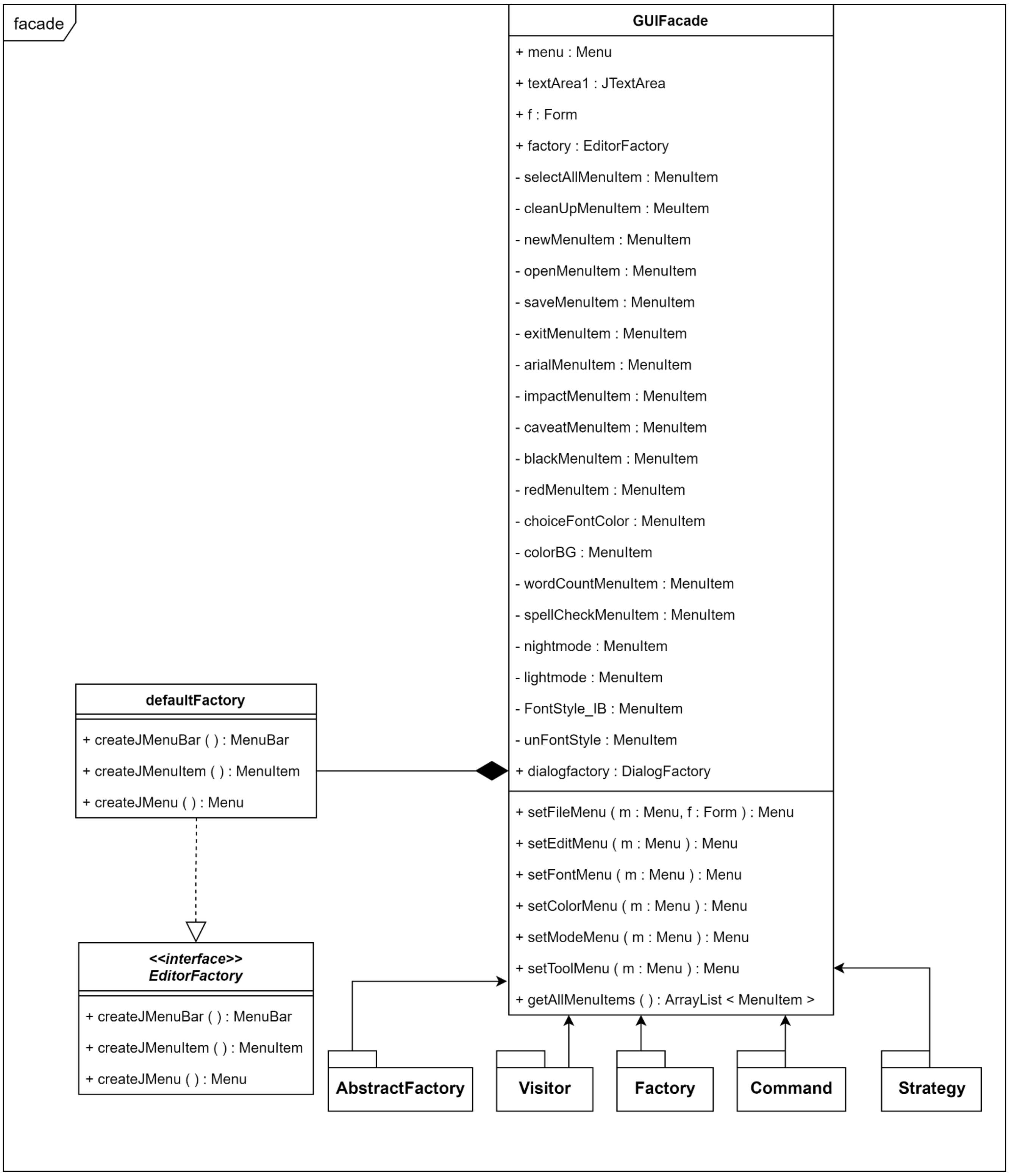


Facade Pattern

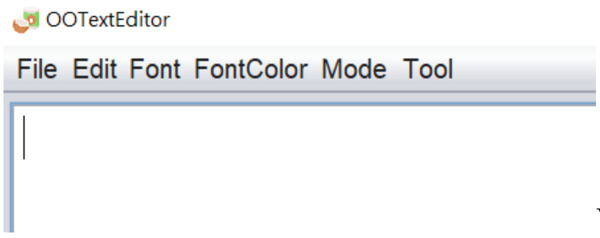
Facade pattern provides an interface to let us put all the interfaces into it.
In our system, class “GUIfacade” puts the MenuItem “New” into the Menu “File”; meanwhile, it can also trigger your actions. For example, when you click the MenuItem “New”, then it will create a new file. This pattern let us to use subsystems more easily.

```
public class GUIfacade {
    Menu menu;
    JTextArea textArea1;
    Form f;
    EditorFactory factory = new defaultFactory();
    private MenuItem selectAllMenuItem;
    private MenuItem cleanUpMenuItem;
    //some object//

  public Menu setFileMenu (Menu m , Form f) {
```

```
 public Menu setFileMenu (Menu m , Form f) {
     menu = m;
     this.f = f;
     this.textArea1 = f.getTextArea();
     menu.setContent("File");

     //---- newMenuItem ----
     newMenuItem = factory.creatJMenuItem();
     newMenuItem.setContent("New");
     newMenuItem.addActionListener(new newFile(textArea1));
     menu.add(newMenuItem);

     //---- openMenuItem ----
     openMenuItem = factory.creatJMenuItem();
     openMenuItem .setContent("Open");
     openMenuItem .addActionListener(new openFile(f));
     menu.add(openMenuItem );

     //---- saveMenuItem ----
     saveMenuItem = factory.creatJMenuItem();
     saveMenuItem.setContent("Save");
     saveMenuItem.addActionListener(new saveFile(f));
     menu.add(saveMenuItem);

     //---- exitMenuItem ----
     exitMenuItem = factory.creatJMenuItem();
     exitMenuItem.setContent("Exit");
     exitMenuItem.addActionListener(new exitApplication());
     menu.add(exitMenuItem);

     return menu;
 }
```


facade
GUIFacade
+ menu : Menu
+ textArea1 : JTextArea
+ f : Form
+ factory : EditorFactory
- selectAllMenuItem : MenuItem
- cleanUpMenuItem : MeuItem
- newMenuItem : MenuItem
- openMenuItem : MenuItem
- saveMenuItem : MenuItem
- exitMenuItem : MenuItem
- arialMenuItem : MenuItem
- impactMenuItem : MenuItem
- caveatMenuItem : MenuItem
- blackMenuItem : MenuItem
- redMenuItem : MenuItem
- choiceFontColor : MenuItem
- colorBG : MenuItem
- wordCountMenuItem : MenuItem
- spellCheckMenuItem : MenuItem
- nightmode : MenuItem
- lightmode : MenuItem
- FontStyle_IB : MenuItem
- unFontStyle : MenuItem
+ dialogfactory : DialogFactory
+ setFileMenu (m : Menu, f : Form) : Menu
+ setEditMenu (m : Menu) : Menu
+ setFontMenu (m : Menu) : Menu
+ setColorMenu (m : Menu) : Menu
+ setModeMenu (m : Menu) : Menu
+ setToolMenu (m : Menu) : Menu
+ getAllMenuItems ( ) : ArrayList < MenuItem >
defaultFactory
+ createJMenuBar ( ) : MenuBar
+ createJMenuItem ( ) : MenuItem
+ createJMenu ( ) : Menu
<<interface>>
EditorFactory
+ createJMenuBar ( ) : MenuBar
+ createJMenuItem ( ) : MenuItem
+ createJMenu ( ) : Menu
AbstractFactory
Visitor
Factory
Command
Strategy

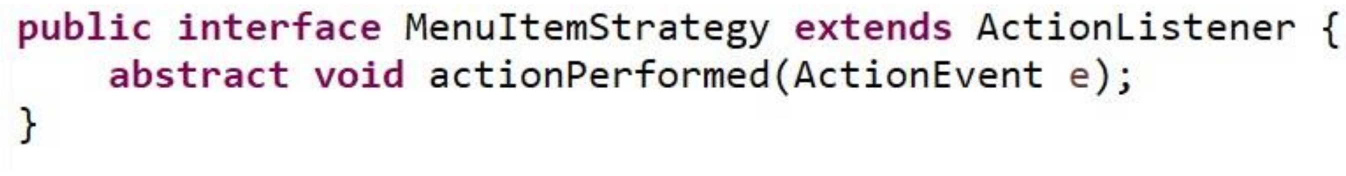
This is the application of the façade pattern in our text editor. It provides a simplified interface for our system, so we can easily to use these methods in our GUI.



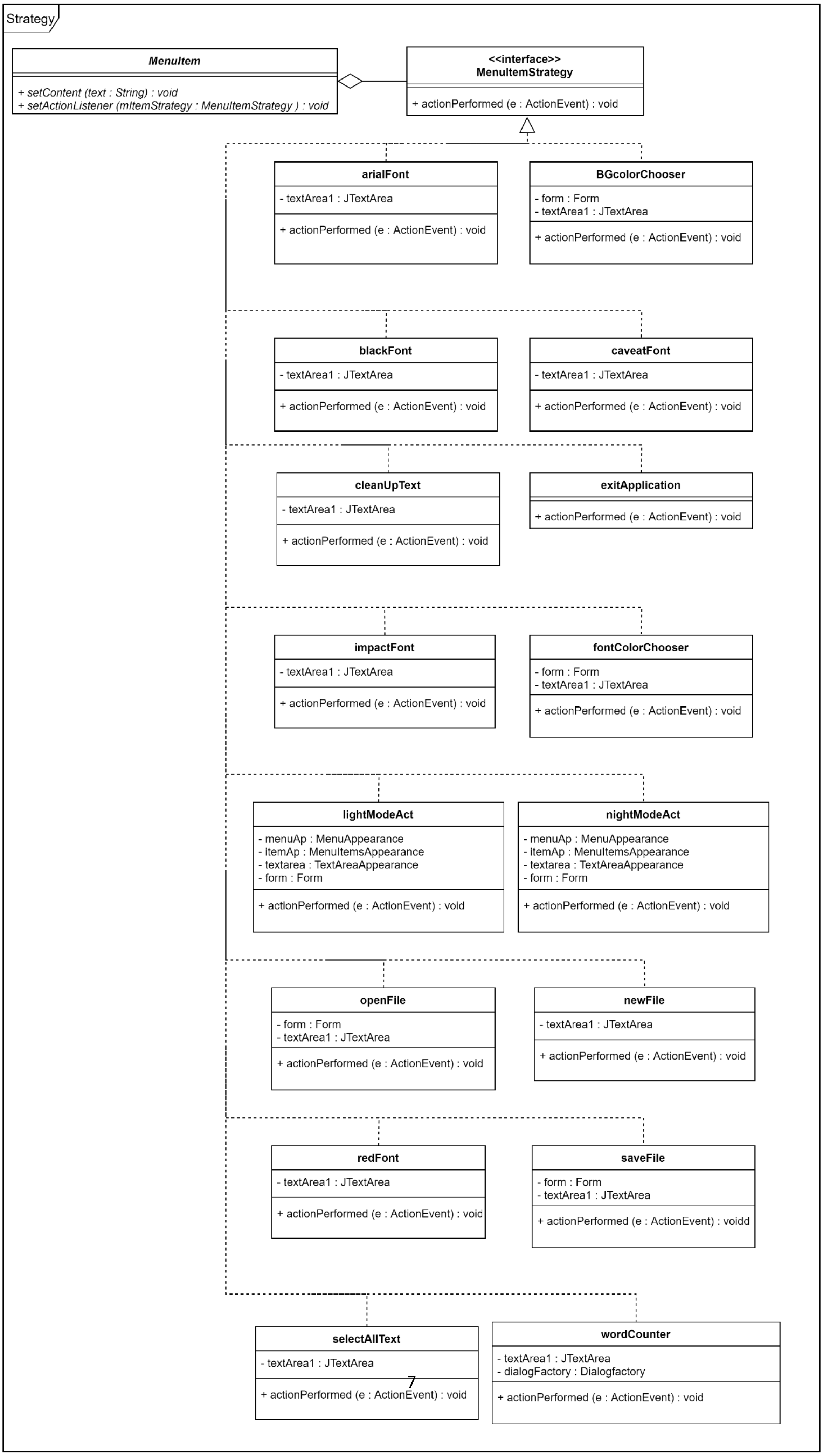
We will introduce all the methods in following patterns.

Strategy Pattern

Strategy pattern defines a family of algorithms, encapsulate each one, and make them interchangeable.
Strategy lets the algorithm vary independently from clients that use it.
In our system, we have 16 different strategies which can let us to choose the method that we want to perform. We use Strategy pattern to create methods in the text editor, and we can communicate with strategies (like arialFont, BGcolorChooser…, etc.) by using interface “MenuItemStrategy”.

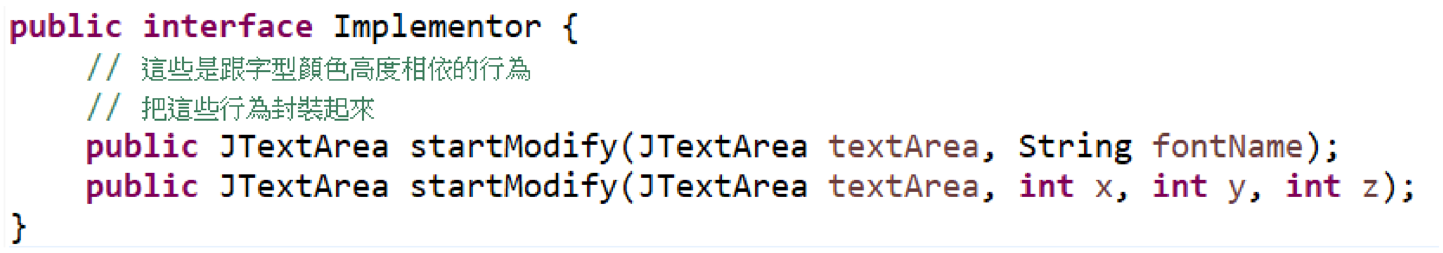


```java
public interface MenuItemStrategy extends ActionListener {
    abstract void actionPerformed(ActionEvent e);
}
```


Strategy
MenuItem
+ setContent (text : String) : void
+ setActionListener (mItemStrategy : MenuItemStrategy) : void
<<interface>>
MenuItemStrategy
+ actionPerformed (e : ActionEvent) : void
arialFont
- textArea1 : JTextArea
+ actionPerformed (e : ActionEvent) : void
BGcolorChooser
- form : Form
- textArea1 : JTextArea
+ actionPerformed (e : ActionEvent) : void
blackFont
- textArea1 : JTextArea
+ actionPerformed (e : ActionEvent) : void
caveatFont
- textArea1 : JTextArea
+ actionPerformed (e : ActionEvent) : void
cleanUpText
- textArea1 : JTextArea
+ actionPerformed (e : ActionEvent) : void
exitApplication
+ actionPerformed (e : ActionEvent) : void
impactFont
- textArea1 : JTextArea
+ actionPerformed (e : ActionEvent) : void
fontColorChooser
- form : Form
- textArea1 : JTextArea
+ actionPerformed (e : ActionEvent) : void
lightModeAct
- menuAp : MenuAppearance
- itemAp : MenuItemsAppearance
- textarea : TextAreaAppearance
- form : Form
+ actionPerformed (e : ActionEvent) : void
nightModeAct
- menuAp : MenuAppearance
- itemAp : MenuItemsAppearance
- textarea : TextAreaAppearance
- form : Form
+ actionPerformed (e : ActionEvent) : void
openFile
- form : Form
- textArea1 : JTextArea
+ actionPerformed (e : ActionEvent) : void
newFile
- textArea1 : JTextArea
+ actionPerformed (e : ActionEvent) : void
redFont
- textArea1 : JTextArea
+ actionPerformed (e : ActionEvent) : void
saveFile
- form : Form
- textArea1 : JTextArea
+ actionPerformed (e : ActionEvent) : voidd
selectAllText
- textArea1 : JTextArea
+ actionPerformed (e : ActionEvent) : void
wordCounter
- textArea1 : JTextArea
- dialogFactory : Dialogfactory
+ actionPerformed (e : ActionEvent) : void

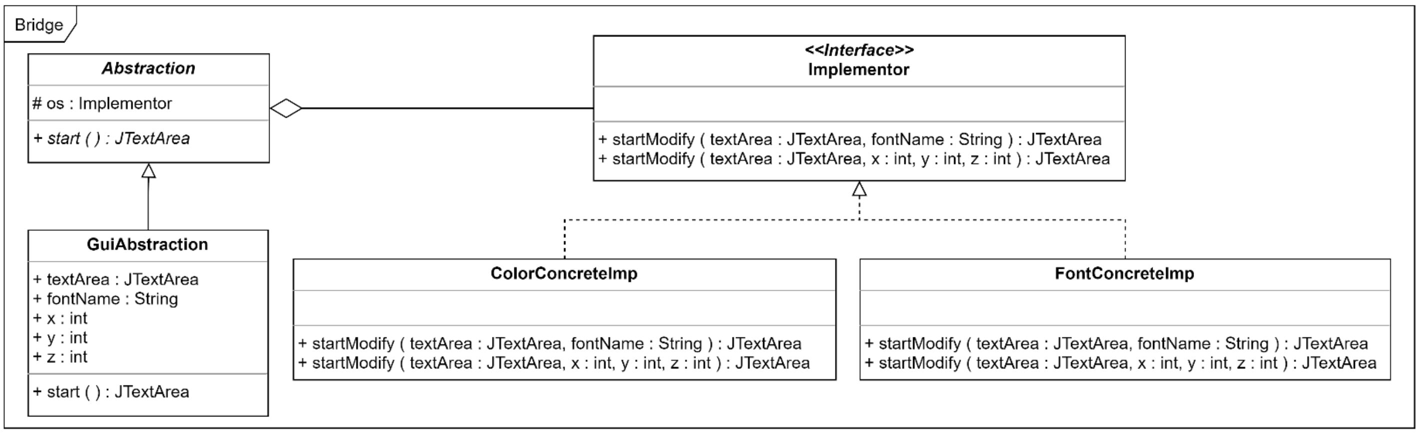
Bridge Pattern

We use Bridge pattern to separate Color and Fonts' abstraction from their implementation.
As you can see, we have class "Abstraction" and class "Implementor". The behavior in the Implementor class have the strong connection with Font and Color, we encapsulate these behaviors.

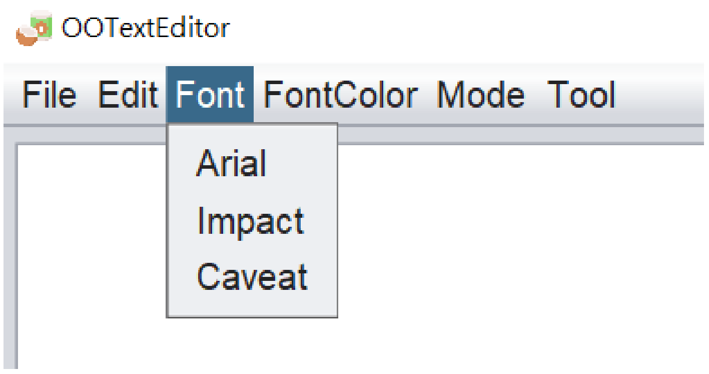


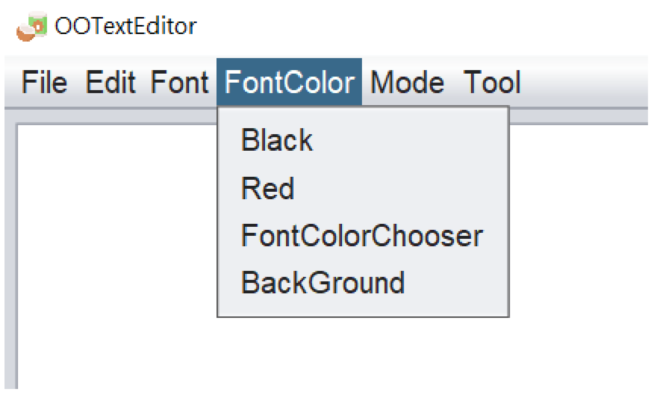
```
public interface Implementor {
    // 這些是跟字型顏色高度相依的行為
    // 把這些行為封裝起來
    public JTextArea startModify(JTextArea textArea, String fontName);
    public JTextArea startModify(JTextArea textArea, int x, int y, int z);
}
```

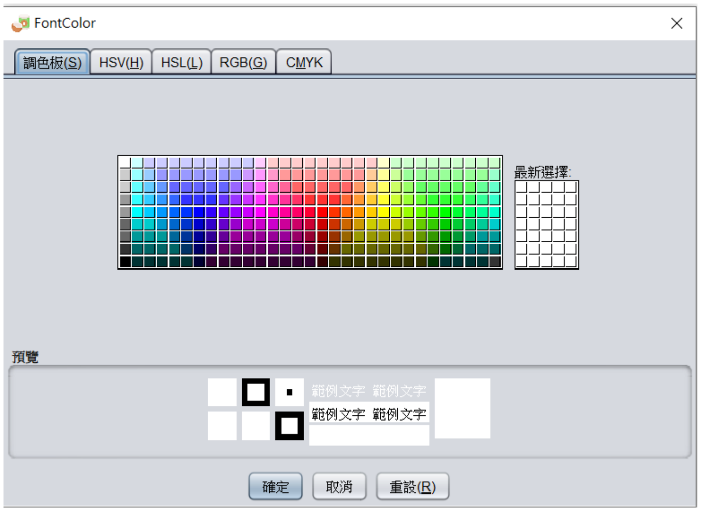
We use class "ColorConcreteImp" and class "FontConcreteImp" to implement class "Implementor", and Font and Color connect its own method so it can work correctly.
Then we use class "GuiAbstraction" to extends class "Abstraction", the behavior related with Font and Color are already encapsulate before, so class "GuiAbstraction" doesn't need to know how to implement.
Every implementor should be suitable for all abstraction which means that every implementor can implement any Abstraction.

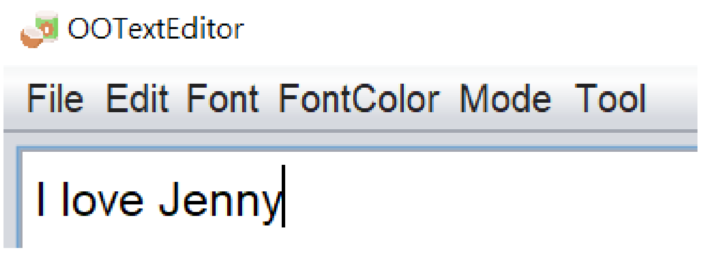

Bridge
Abstraction
# os : Implementor
+ start ( ) : JTextArea
<<Interface>>
Implementor
+ startModify (textArea : JTextArea, fontName : String) : JTextArea
+ startModify (textArea : JTextArea, x : int, y : int, z : int) : JTextArea
GuiAbstraction
+ textArea : JTextArea
+ fontName : String
+ x : int
+ y : int
+ z : int
+ start ( ) : JTextArea
ColorConcreteImp
+ startModify (textArea : JTextArea, fontName : String) : JTextArea
+ startModify (textArea : JTextArea, x : int, y : int, z : int) : JTextArea
FontConcreteImp
+ startModify (textArea : JTextArea, fontName : String) : JTextArea
+ startModify (textArea : JTextArea, x : int, y : int, z : int) : JTextArea

This is the application of the bridge pattern in our text editor. We have three fonts that can choose, and we can use the "FontColorChooser" to change the color of the text, and use the "BackGround" to change the color of the text area. "FontColorChooser" is a tool looks like a palette.

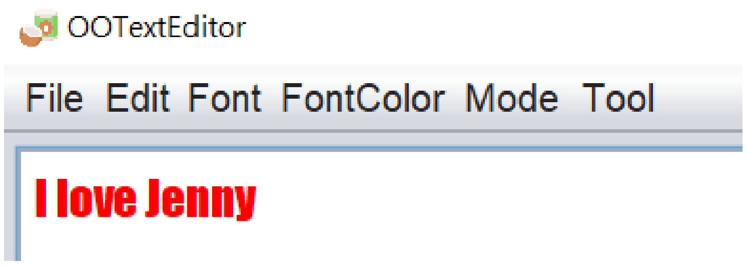




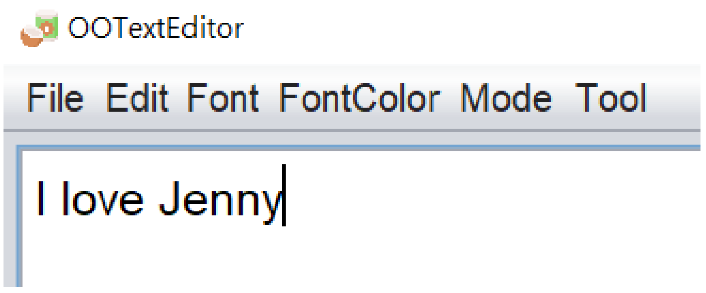




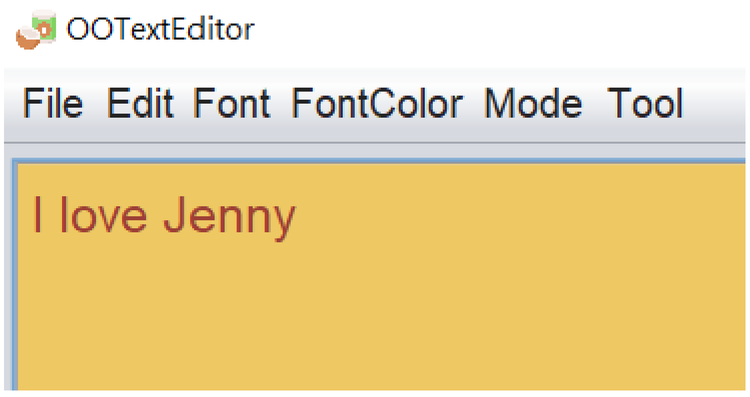
Before



After choose font color&type



Before

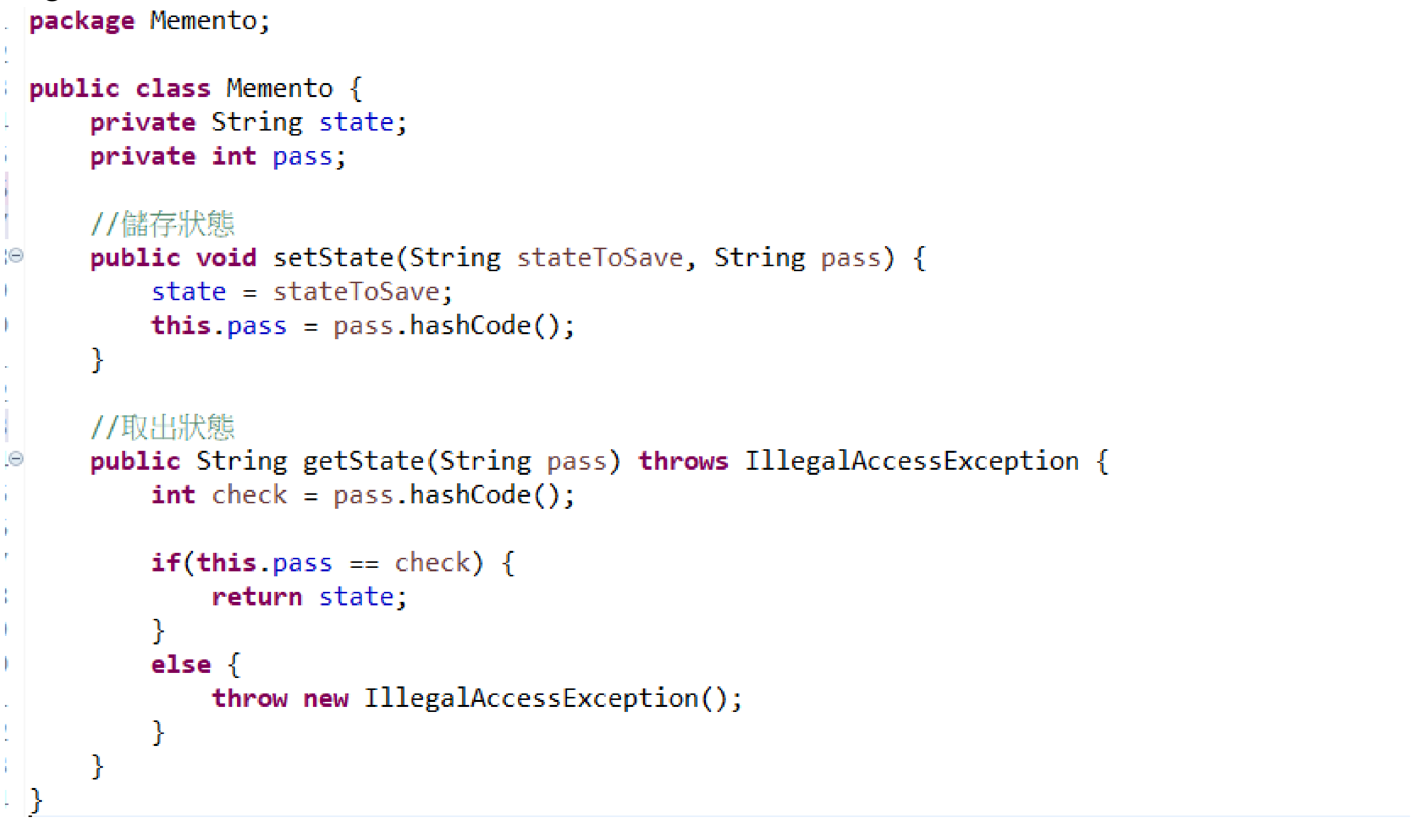


After choose background color

When the Abstraction needs to be changed in the future, we can also create a class "GUIAbstraction" without affecting implementation of method, because the method that will change are all moved to the Implementor.

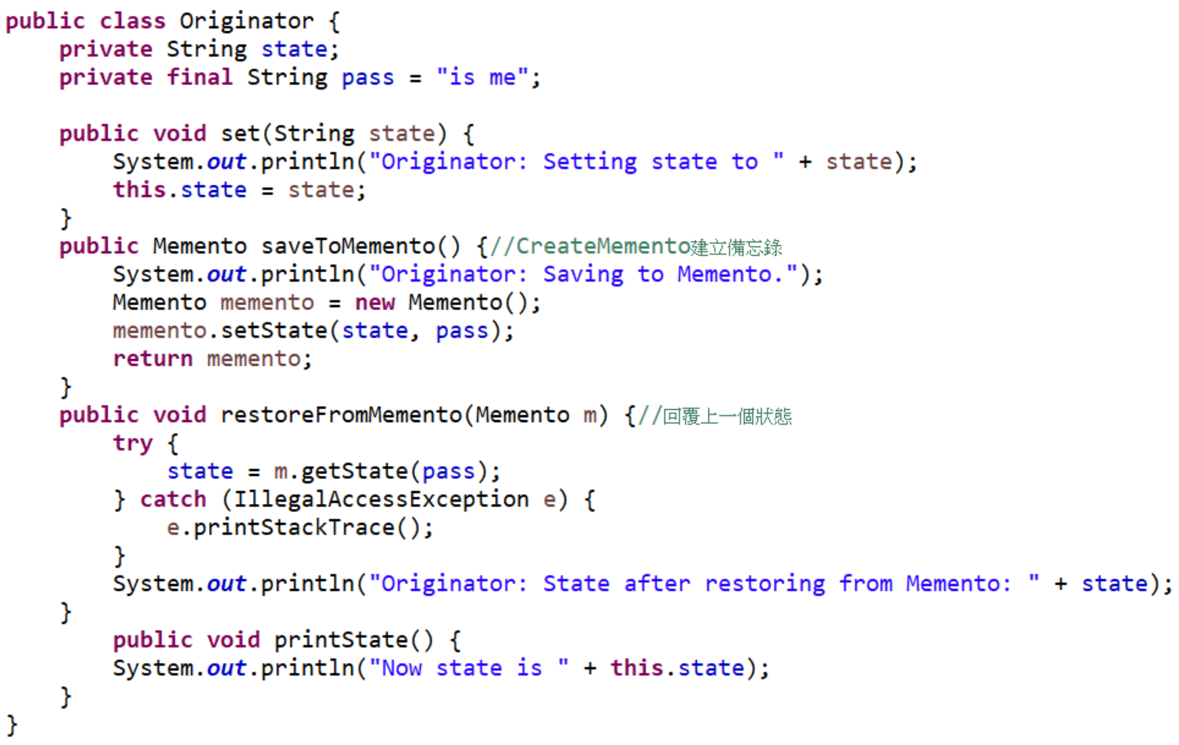
Memento pattern

Memento pattern can capture and externalize an object's internal state so that the object can be restored to this state later without violating encapsulation.
We use memento pattern to store the previous state, so that we can redo some action or go back to some state.
In class “Memento”, we use method setState() to Store the state. And we use method getState() to get the state.



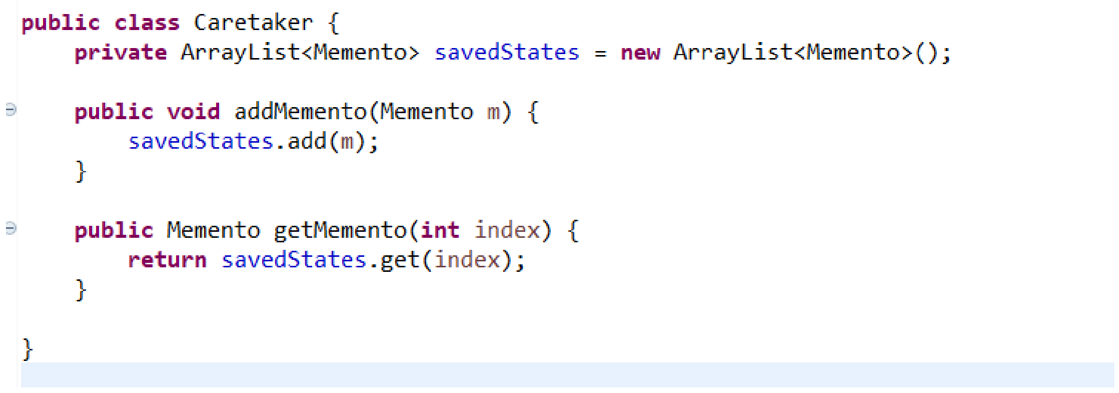
```java
package Memento;

public class Memento {
    private String state;
    private int pass;

    //儲存狀態
    public void setState(String stateToSave, String pass) {
        state = stateToSave;
        this.pass = pass.hashCode();
    }

    //取出狀態
    public String getState(String pass) throws IllegalAccessException {
        int check = pass.hashCode();

        if(this.pass == check) {
            return state;
        }
        else {
            throw new IllegalAccessException();
        }
    }
}
```

In class “Originator”, we create “state”, and use method set() to save the state.
Then use the method saveToMemento() to store state to memento.
In method restoreFromMemento(), we have a method getState() to get the pass/previous state, so that we can go back to the pass state.

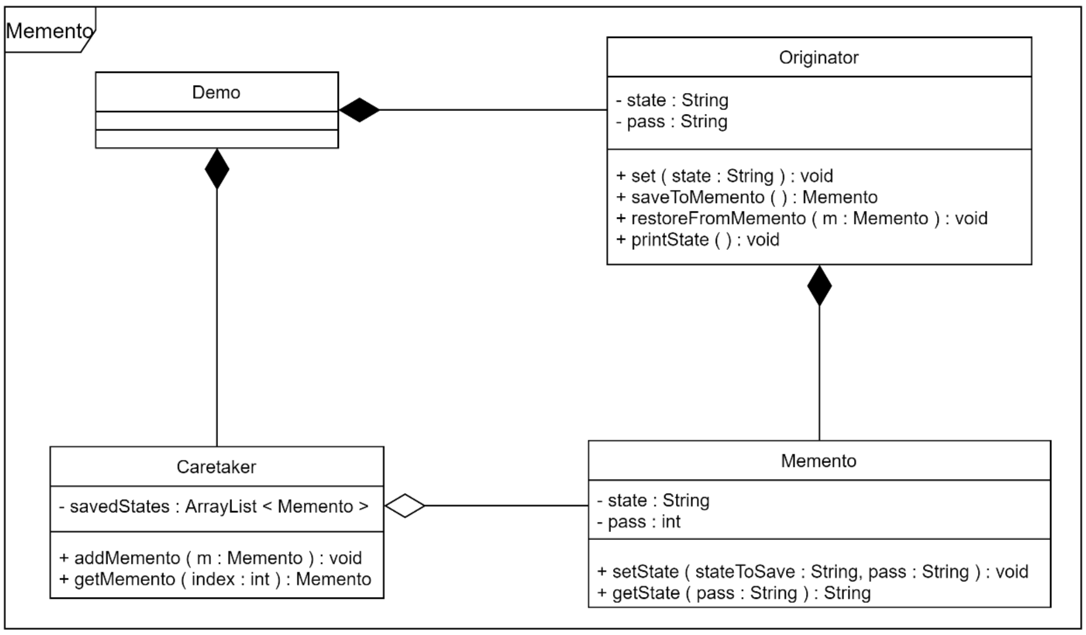


```
public class Originator {
    private String state;
    private final String pass = "is me";

    public void set(String state) {
        System.out.println("Originator: Setting state to " + state);
        this.state = state;
    }
    public Memento saveToMemento() {//CreateMemento建立備忘錄
        System.out.println("Originator: Saving to Memento.");
        Memento memento = new Memento();
        memento.setState(state, pass);
        return memento;
    }
    public void restoreFromMemento(Memento m) {//回覆上一個狀態
        try {
            state = m.getState(pass);
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
        System.out.println("Originator: State after restoring from Memento: " + state);
    }
        public void printState() {
        System.out.println("Now state is " + this.state);
    }
}
```

In class “Caretaker”, we create an arraylist call “savedSates” to store the memento. We use method addMemento() to add memento in the arraylist “savedStates”.

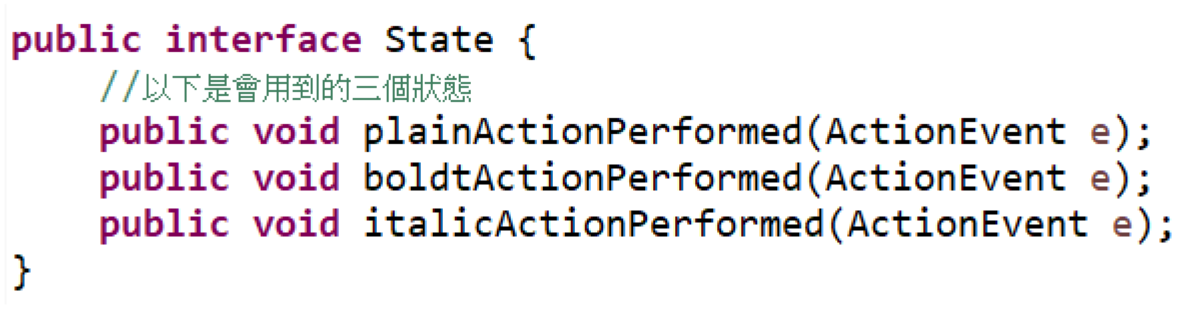


```
public class Caretaker {
    private ArrayList<Memento> savedStates = new ArrayList<Memento>();

    public void addMemento(Memento m) {
        savedStates.add(m);
    }

    public Memento getMemento(int index) {
        return savedStates.get(index);
    }

}
```


Memento
Demo
Originator
- state : String
- pass : String
+ set (state : String) : void
+ saveToMemento ( ) : Memento
+ restoreFromMemento (m : Memento) : void
+ printState ( ) : void
Caretaker
- savedStates : ArrayList < Memento >
+ addMemento (m : Memento) : void
+ getMemento (index : int) : Memento
Memento
- state : String
- pass : int
+ setState (stateToSave : String, pass : String) : void
+ getState (pass : String) : String

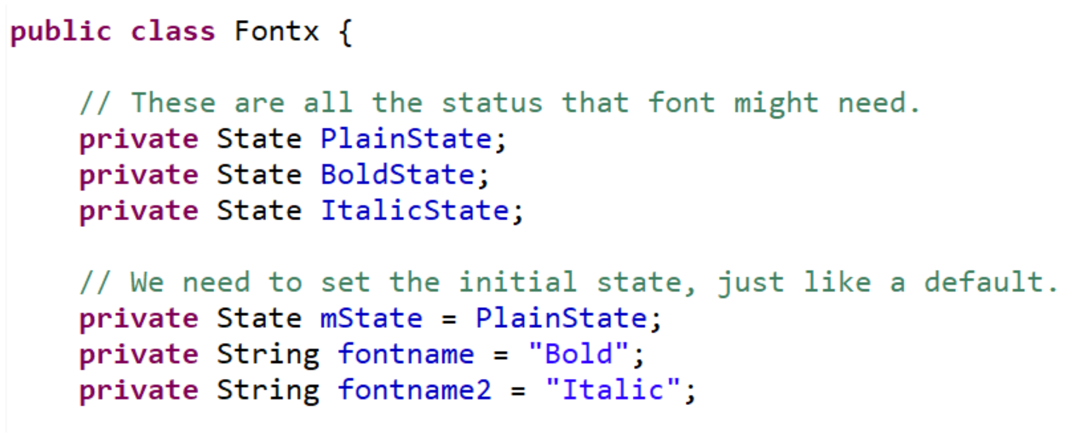
State Pattern

State pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
In our system, class “Fontx” delegates the jobs to interface “State”. Interface “State” has three concrete states: “BoldState”, “PlainState” and “ItalicState”, then we can change whatever states we want. The default state is “PlainState”, if we want to change the font into bold and italic, we can click the blod&italic button to change it.
First, we create an interface “State”, which is that all state class has to implement it.
And it use method plainActionPerformed(), boldActionPerformed(), italicActionPerformed() to define the thing that those states will do.



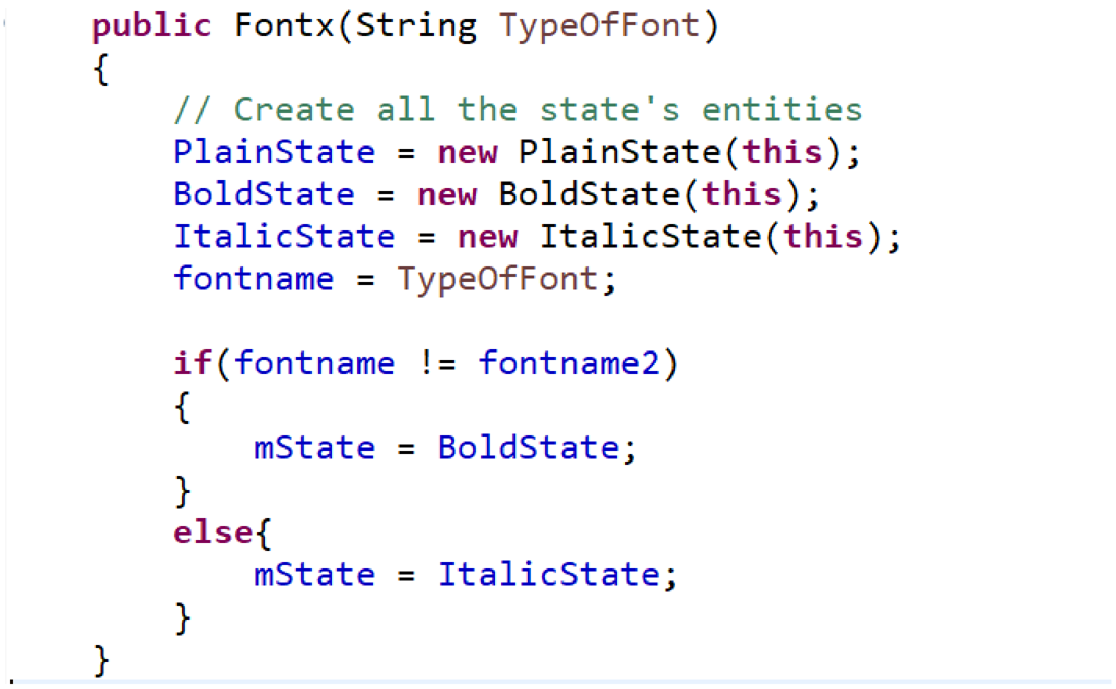
```
public interface State {
    //以下是會用到的三個狀態
    public void plainActionPerformed(ActionEvent e);
    public void boldtActionPerformed(ActionEvent e);
    public void italicActionPerformed(ActionEvent e);
}
```

The class “Fontx” creates all the status that font might need, and define some variable to keep the current state. We set “PlainState” to initial state.



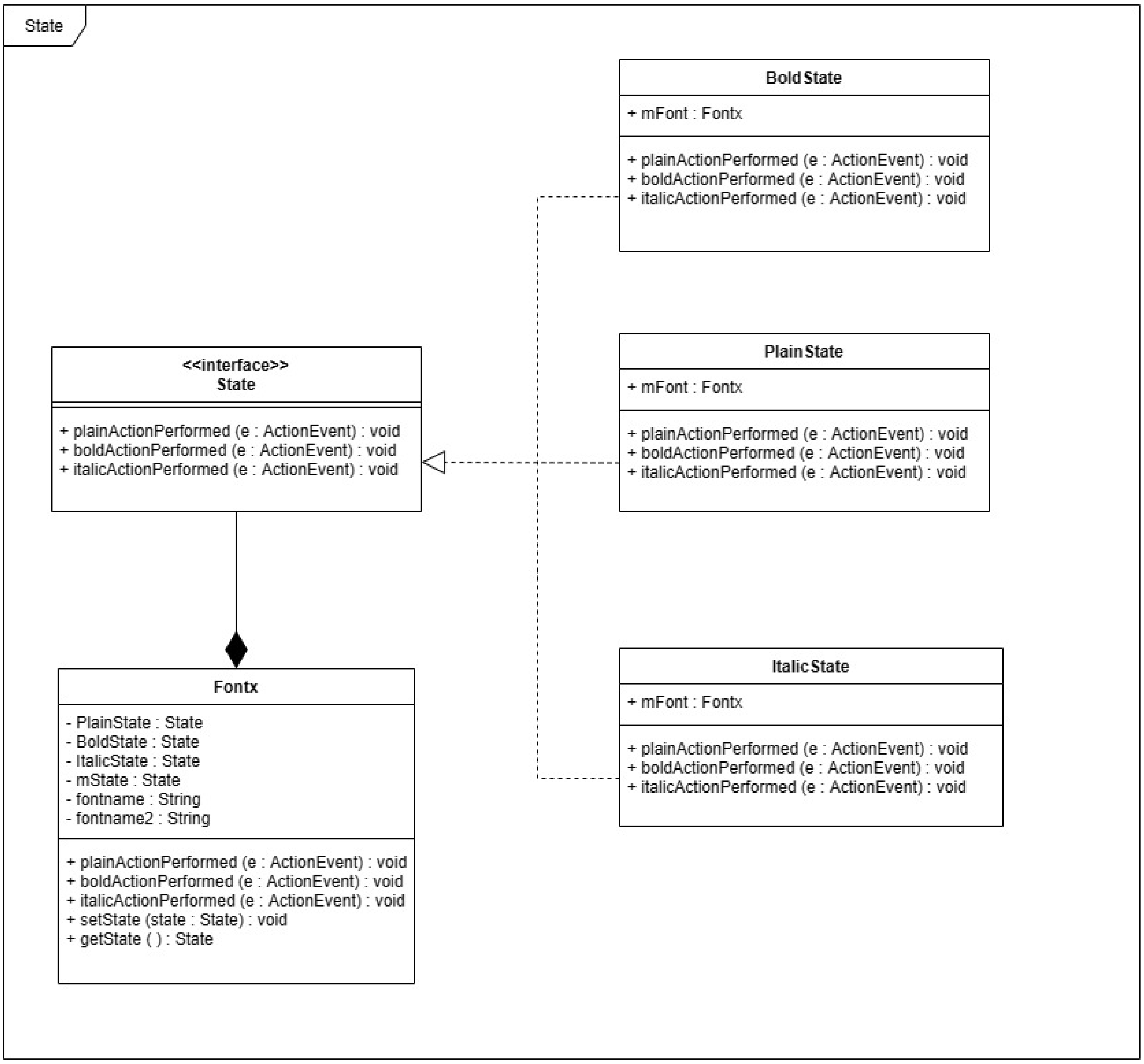
```
public class Fontx {

    // These are all the status that font might need.
    private State PlainState;
    private State BoldState;
    private State ItalicState;

    // We need to set the initial state, just like a default.
    private State mState = PlainState;
    private String fontname = "Bold";
    private String fontname2 = "Italic";
```

The class “Fontx” creates all the state’s entities, and determines the state we want to change.



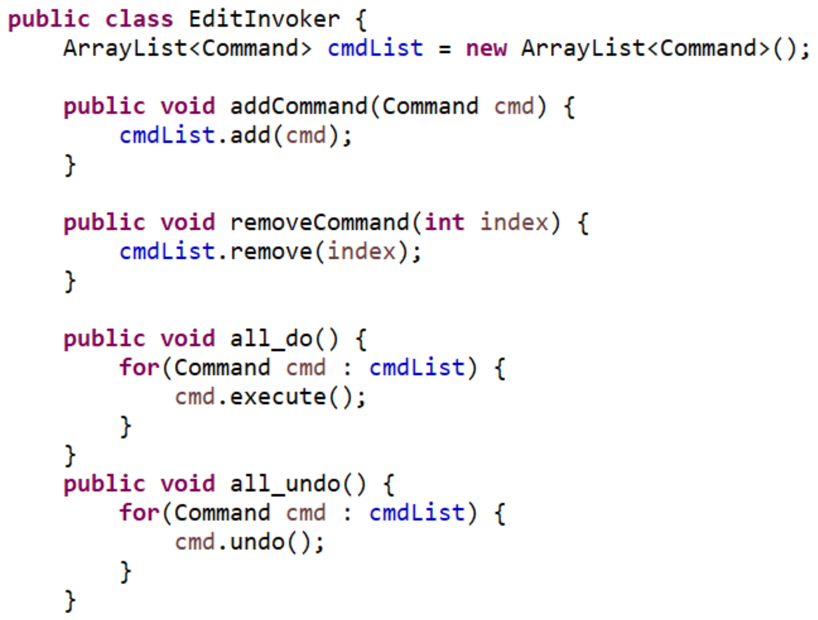
```
    public Fontx(String TypeOfFont)
    {
        // Create all the state's entities
        PlainState = new PlainState(this);
        BoldState = new BoldState(this);
        ItalicState = new ItalicState(this);
        fontname = TypeOfFont;

        if(fontname != fontname2)
        {
            mState = BoldState;
        }
        else{
            mState = ItalicState;
        }
    }
```

And class “PlainState”, “ItalicState”, “BoldState” implement the State.


State
Bold State
+ mFont : Fontx
+ plainActionPerformed (e : ActionEvent) : void
+ boldActionPerformed (e : ActionEvent) : void
+ italicActionPerformed (e : ActionEvent) : void
Plain State
+ mFont : Fontx
+ plainActionPerformed (e : ActionEvent) : void
+ boldActionPerformed (e : ActionEvent) : void
+ italicActionPerformed (e : ActionEvent) : void
Italic State
+ mFont : Fontx
+ plainActionPerformed (e : ActionEvent) : void
+ boldActionPerformed (e : ActionEvent) : void
+ italicActionPerformed (e : ActionEvent) : void
<<interface>>
State
+ plainActionPerformed (e : ActionEvent) : void
+ boldActionPerformed (e : ActionEvent) : void
+ italicActionPerformed (e : ActionEvent) : void
Fontx
- PlainState : State
- BoldState : State
- ItalicState : State
- mState : State
- fontname : String
- fontname2 : String
+ plainActionPerformed (e : ActionEvent) : void
+ boldActionPerformed (e : ActionEvent) : void
+ italicActionPerformed (e : ActionEvent) : void
+ setState (state : State) : void
+ getState ( ) : State

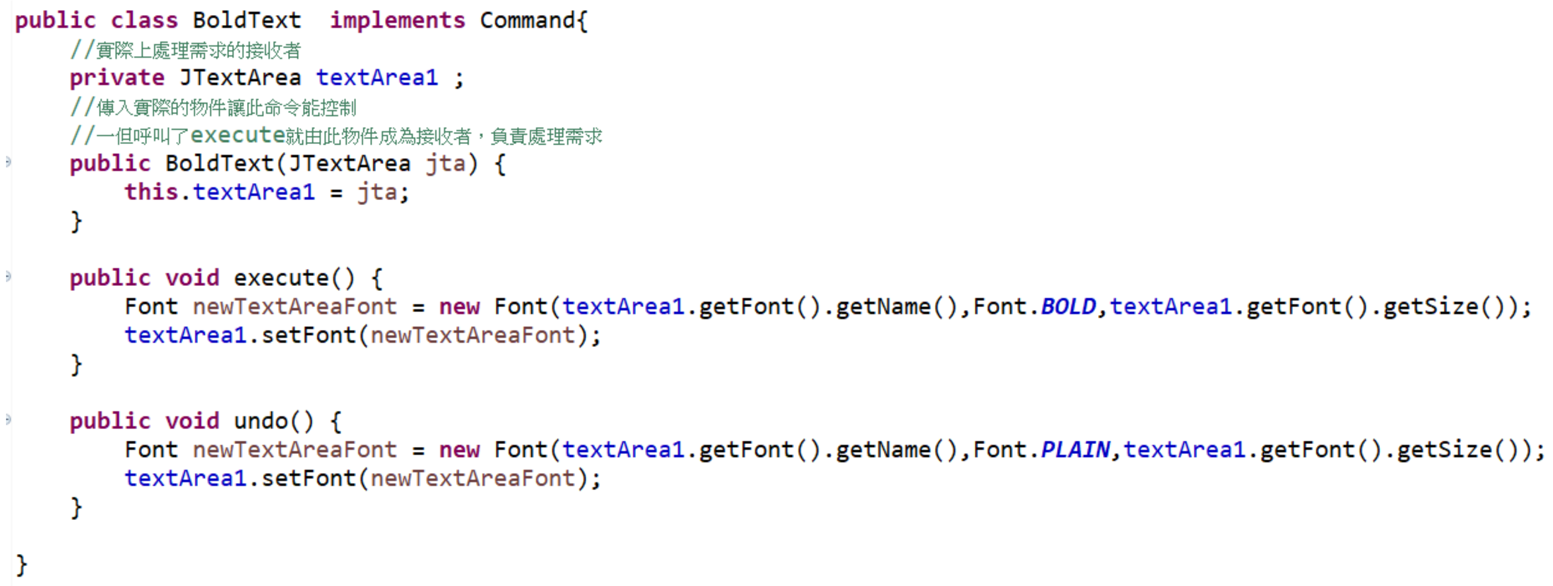
Command Pattern

Command Pattern encapsulates a request as an object which is “Command” in this diagram, thereby letting you parameterize other objects with different requests, like “BoldText” and “ItalicText” in our system.
We use Client to create a class “EditInvoker”.
Class “EditInvoker” is like a remote control, it creates a command that used to be executed.



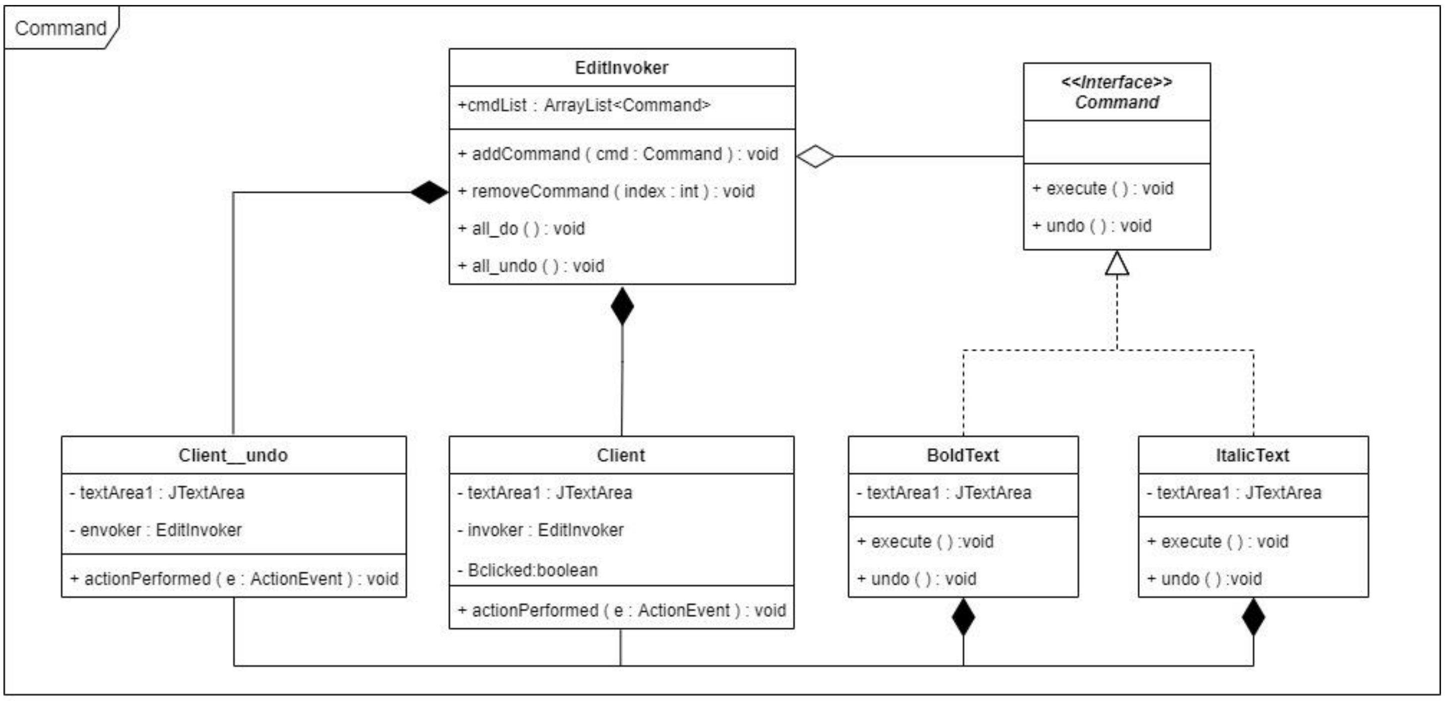
```
public class EditInvoker {
    ArrayList<Command> cmdList = new ArrayList<Command>();

    public void addCommand(Command cmd) {
        cmdList.add(cmd);
    }

    public void removeCommand(int index) {
        cmdList.remove(index);
    }

    public void all_do() {
        for(Command cmd : cmdList) {
            cmd.execute();
        }
    }
    public void all_undo() {
        for(Command cmd : cmdList) {
            cmd.undo();
        }
    }
```

We use class “BoldText” and class “ItalicText” to implement “Command”. Pass in the actual object “jta” to control. If we call execute, the object “textArea1” will be a receiver, responsible for handling the demand.

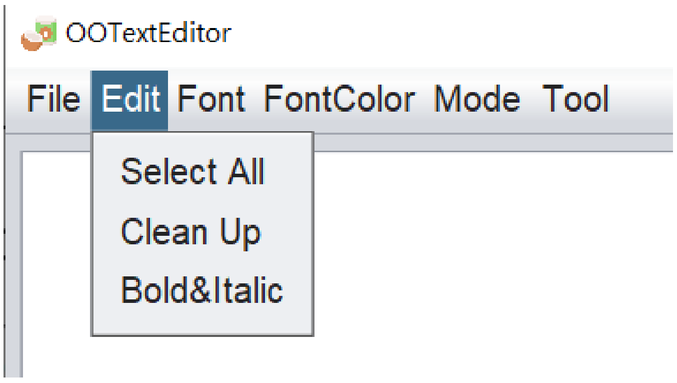


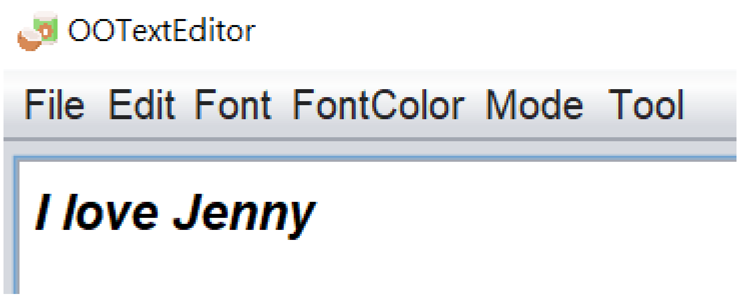
```
public class BoldText  implements Command{
    //實際上處理需求的接收者
    private JTextArea textArea1 ;
    //傳入實際的物件讓此命令能控制
    //一但呼叫了execute就由此物件成為接收者，負責處理需求
    public BoldText(JTextArea jta) {
        this.textArea1 = jta;
    }

    public void execute() {
        Font newTextAreaFont = new Font(textArea1.getFont().getName(),Font.BOLD,textArea1.getFont().getSize());
        textArea1.setFont(newTextAreaFont);
    }

    public void undo() {
        Font newTextAreaFont = new Font(textArea1.getFont().getName(),Font.PLAIN,textArea1.getFont().getSize());
        textArea1.setFont(newTextAreaFont);
    }

}
```

```
import javax.swing.JTextArea;

public class ItalicText  implements Command{
    private JTextArea textArea1 ;

    public ItalicText(JTextArea jta) {
        this.textArea1 = jta;
    }

    public void execute() {
        Font newTextAreaFont = new Font(textArea1.getFont().getName(),Font.ITALIC,textArea1.getFont().getSize());
        textArea1.setFont(newTextAreaFont);
    }

    public void undo() {
        Font newTextAreaFont = new Font(textArea1.getFont().getName(),Font.PLAIN,textArea1.getFont().getSize());
        textArea1.setFont(newTextAreaFont);
    }

}
```



This is the application of the Command pattern in our text editor. We can let the text be thicker and oblique.





Visitor pattern

Visitor pattern represents an operation to be performed on elements of an object structure.
In our system, interface "AppearenceVisitor" lets you define a new operation without changing the classes of the elements on which it operates, like element "MenuItemsAppearence", "MenuAppearence" and "TextAreaAppearence".
Simply put, it separated the behavior of classes and classes.
All the element classes have to extends class "Items" to get the visitor. The thing that will actually operate.

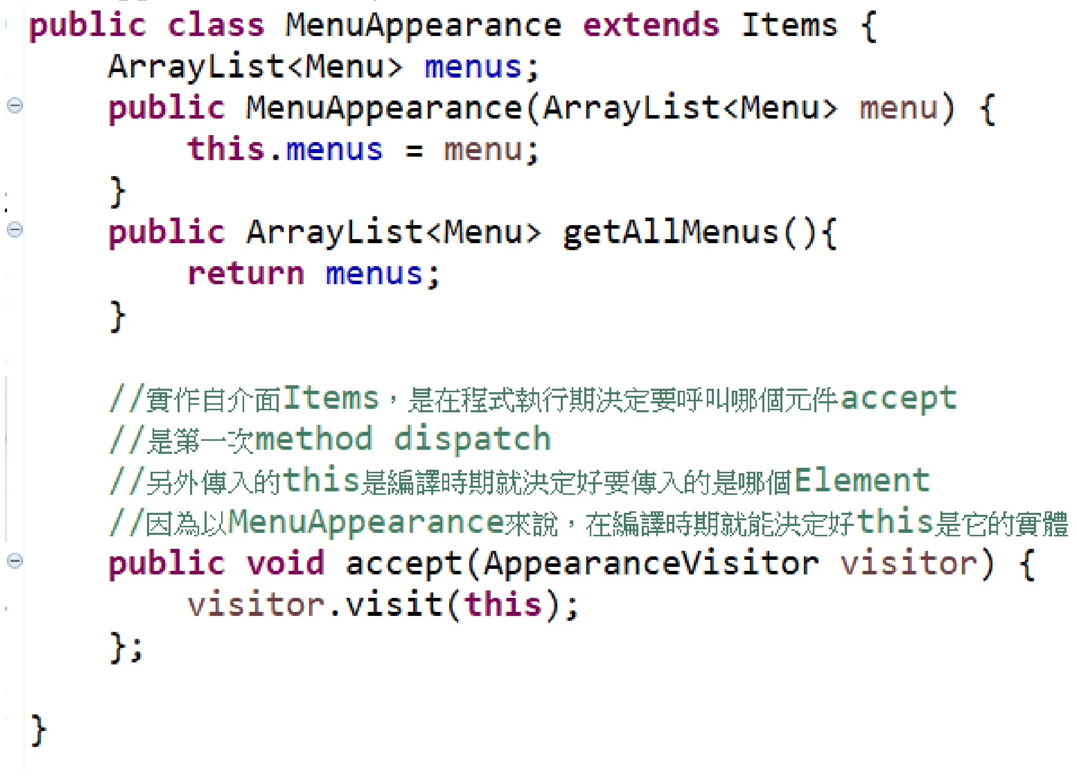
```
//所有elemant的類別都要繼承這個
public abstract class Items {

    public abstract void accept(AppearanceVisitor visitor);
}
```

In interface "AppearanceVisitor", we have three methods because we have three elements to visit.

```
public interface AppearanceVisitor {
    //因為有三個要訪問的元件  所以有三個方法
    public void visit(MenuAppearance menus);

    public void visit(MenuItemsAppearance items);

    public void visit(TextAreaAppearance textArea);
}
```

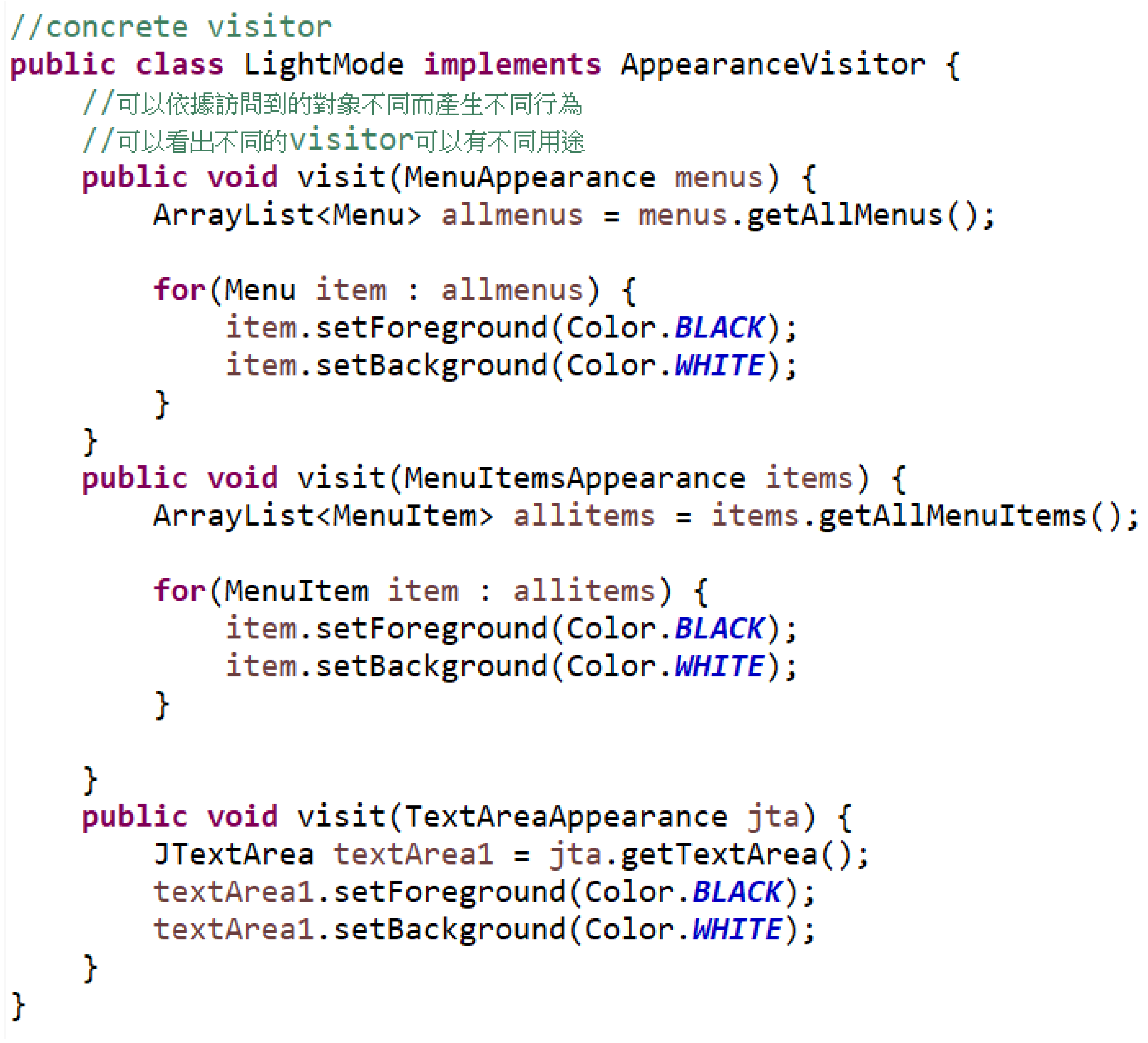
Class "MenuAppearance", "MenuItemsAppearance" and "TextAreaAppearance" are the elements that we actually operate, that is, the elements that visitors will visit.

Take the class “MenuAppearance” for example, the method accept() is implemented from interface “Items”, so it decide which element's method accept() to call during the system execution. This is the first method dispatch. Also, each concrete visitor implements interface “Visitor”; therefore, method accept() decides which visit of Visitor will be called during the system execution. This is the second method dispatch.
Visit imports “this”(MenuAppearance) to be the parameter, this let it looks like Polymorphism, but actually it’s method overloading.
Besides, the imported “this” is already decided which element to import during the compiling. Because take class “MenuAppearance” for example, it can determine that “this” is MenuAppearence’s entity.

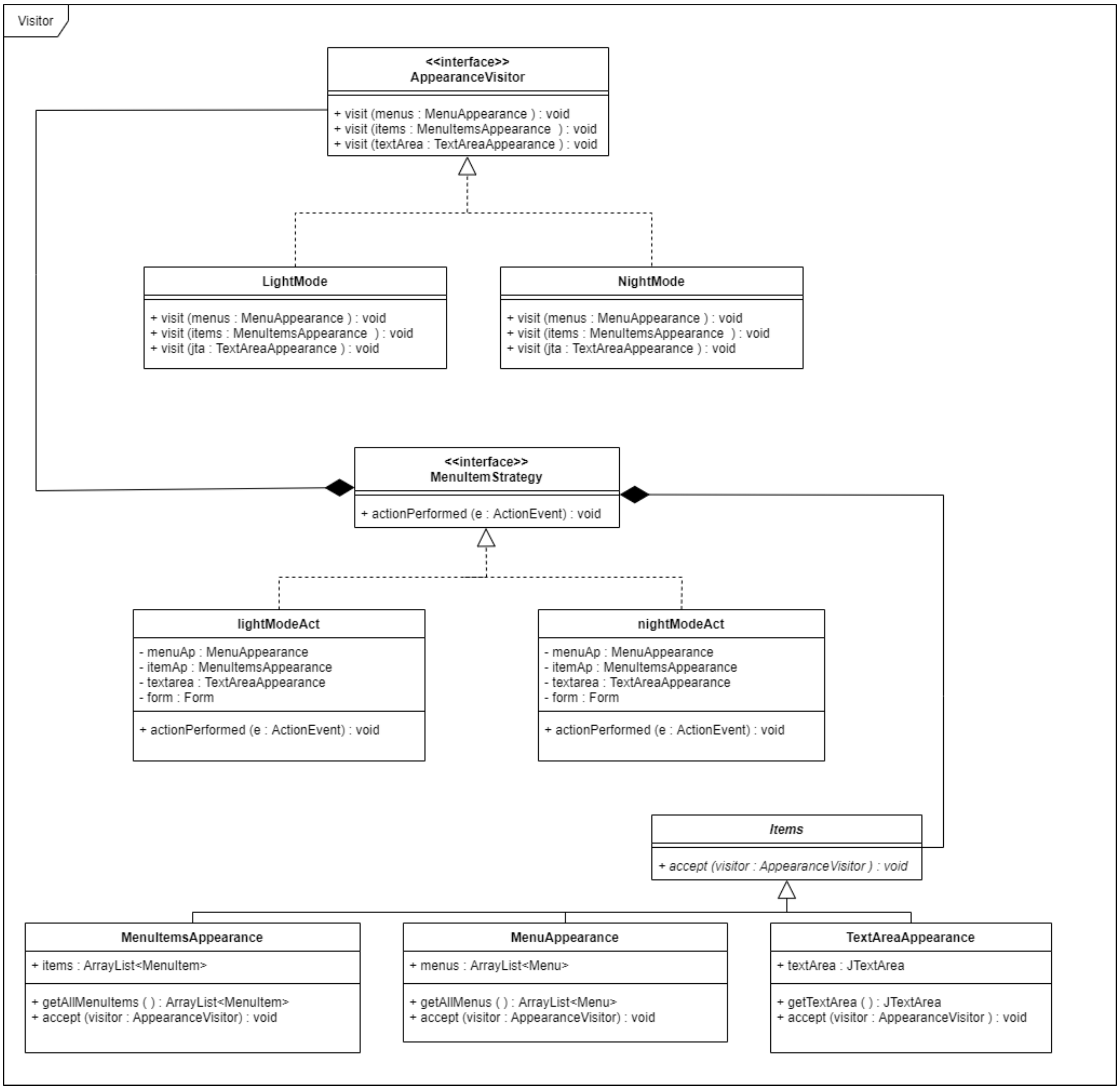


```java
public class MenuAppearance extends Items {
    ArrayList<Menu> menus;
    public MenuAppearance(ArrayList<Menu> menu) {
        this.menus = menu;
    }
    public ArrayList<Menu> getAllMenus(){
        return menus;
    }

    //實作自介面Items，是在程式執行期決定要呼叫哪個元件accept
    //是第一次method dispatch
    //另外傳入的this是編譯時期就決定好要傳入的是哪個Element
    //因為以MenuAppearance來說，在編譯時期就能決定好this是它的實體
    public void accept(AppearanceVisitor visitor) {
        visitor.visit(this);
    };

}
```

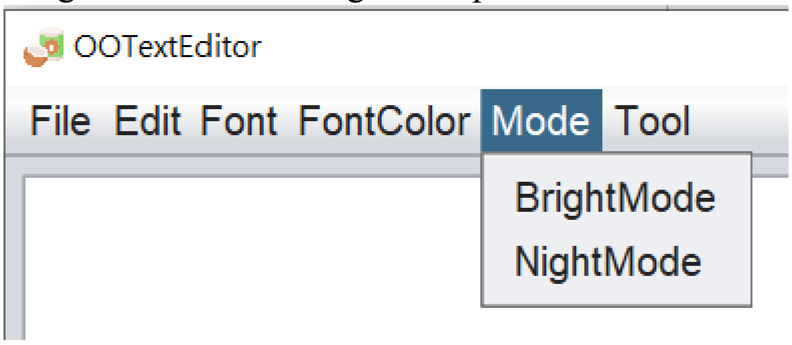
Class "LightMode" and "NightMode" are concrete visitor, they implement interface "AppearanceVisitor". Different behaviors can be generated based on the different visited objects. Take class "LightMode" for example, you can see different visitor has different use.

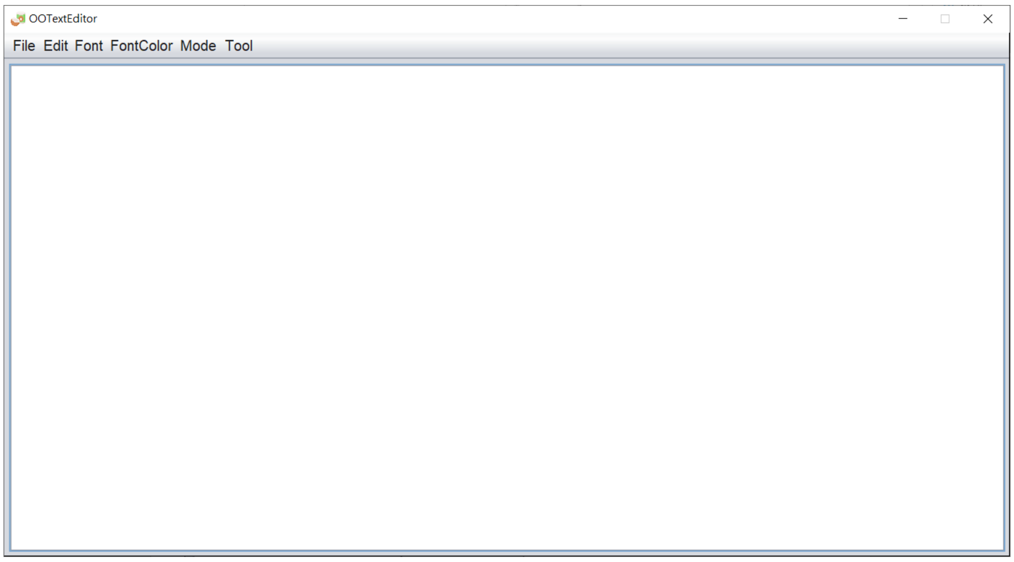


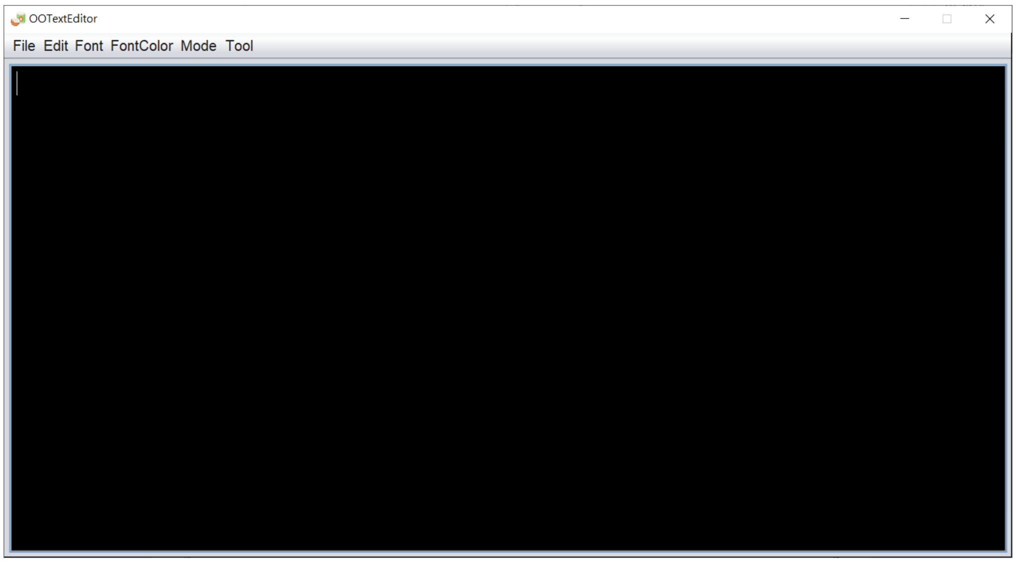
```java
//concrete visitor
public class LightMode implements AppearanceVisitor {
    //可以依據訪問到的對象不同而產生不同行為
    //可以看出不同的visitor可以有不同用途
    public void visit(MenuAppearance menus) {
        ArrayList<Menu> allmenus = menus.getAllMenus();

        for(Menu item : allmenus) {
            item.setForeground(Color.BLACK);
            item.setBackground(Color.WHITE);
        }
    }
    public void visit(MenuItemsAppearance items) {
        ArrayList<MenuItem> allitems = items.getAllMenuItems();

        for(MenuItem item : allitems) {
            item.setForeground(Color.BLACK);
            item.setBackground(Color.WHITE);
        }

    }
    public void visit(TextAreaAppearance jta) {
        JTextArea textArea1 = jta.getTextArea();
        textArea1.setForeground(Color.BLACK);
        textArea1.setBackground(Color.WHITE);
    }
}
```


Visitor
<<interface>>
AppearanceVisitor
+ visit (menus : MenuAppearance) : void
+ visit (items : MenuItemsAppearance) : void
+ visit (textArea : TextAreaAppearance) : void
LightMode
+ visit (menus : MenuAppearance) : void
+ visit (items : MenuItemsAppearance) : void
+ visit (jta : TextAreaAppearance) : void
NightMode
+ visit (menus : MenuAppearance) : void
+ visit (items : MenuItemsAppearance) : void
+ visit (jta : TextAreaAppearance) : void
<<interface>>
MenuItemStrategy
+ actionPerformed (e : ActionEvent) : void
lightModeAct
- menuAp : MenuAppearance
- itemAp : MenuItemsAppearance
- textarea : TextAreaAppearance
- form : Form
+ actionPerformed (e : ActionEvent) : void
nightModeAct
- menuAp : MenuAppearance
- itemAp : MenuItemsAppearance
- textarea : TextAreaAppearance
- form : Form
+ actionPerformed (e : ActionEvent) : void
Items
+ accept (visitor : AppearanceVisitor) : void
MenuItemsAppearance
+ items : ArrayList<MenuItem>
+ getAllMenuItems ( ) : ArrayList<MenuItem>
+ accept (visitor : AppearanceVisitor) : void
MenuAppearance
+ menus : ArrayList<Menu>
+ getAllMenus ( ) : ArrayList<Menu>
+ accept (visitor : AppearanceVisitor) : void
TextAreaAppearance
+ textArea : JTextArea
+ getTextArea ( ) : JTextArea
+ accept (visitor : AppearanceVisitor) : void

This is the application of visitor pattern in our text editor. We can choose “BrightMode” or “NightMode” according to our preference.







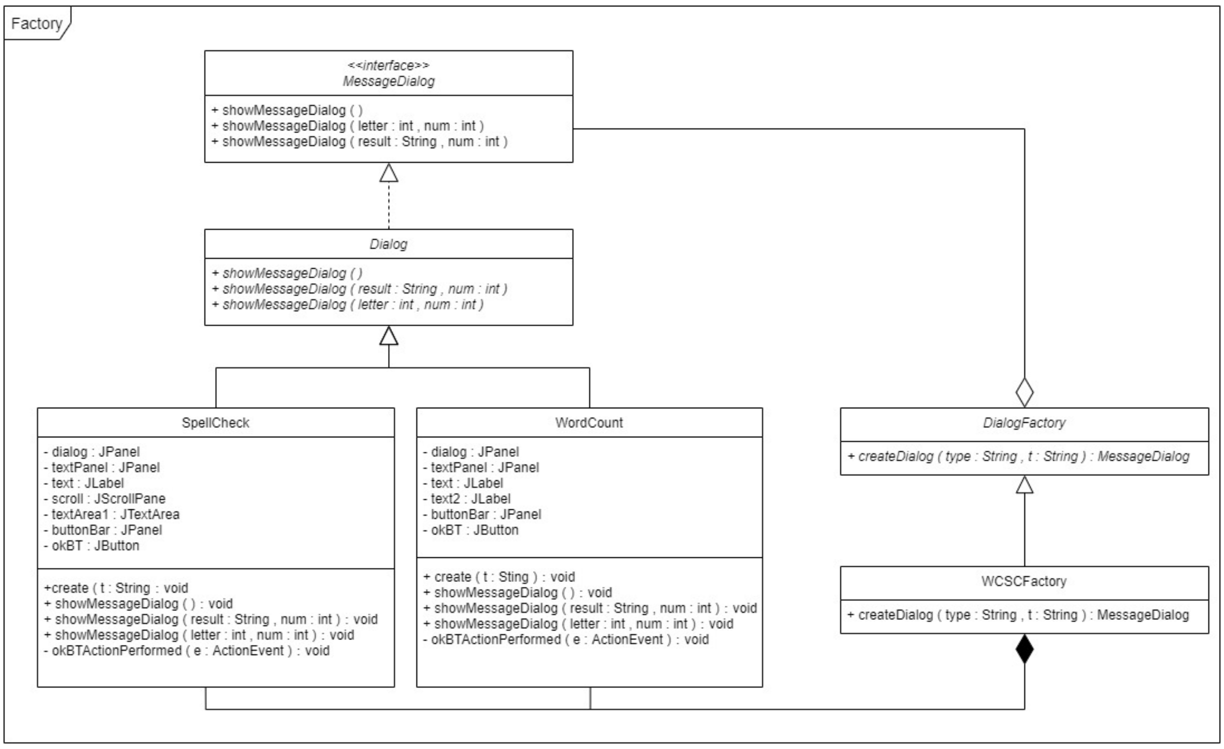
Factory Pattern

Factory pattern defines an interface for creating an object, but let subclasses decide which class to instantiate. It lets a class defer instantiation to subclasses.
In our system, it defines an interface “DialogFactory” for creating an object.

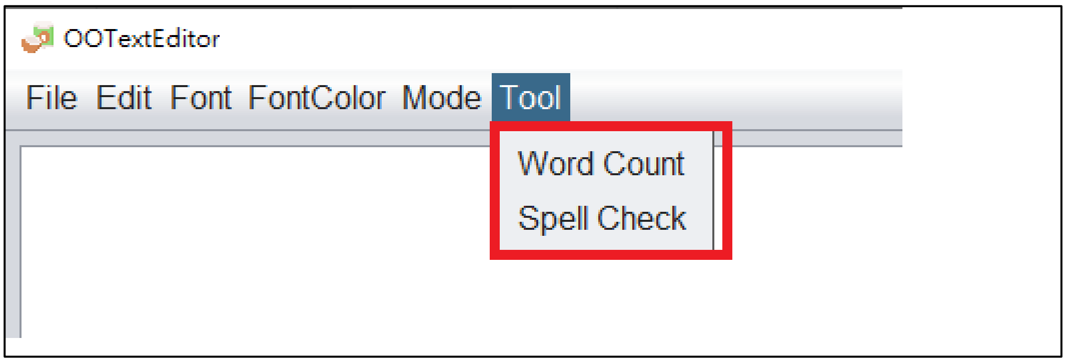
```java
public abstract class DialogFactory {
    //回傳 MessageDialog type選擇拼字、算字 t設定title
    public abstract MessageDialog createDialog(String type,String t);
}
```

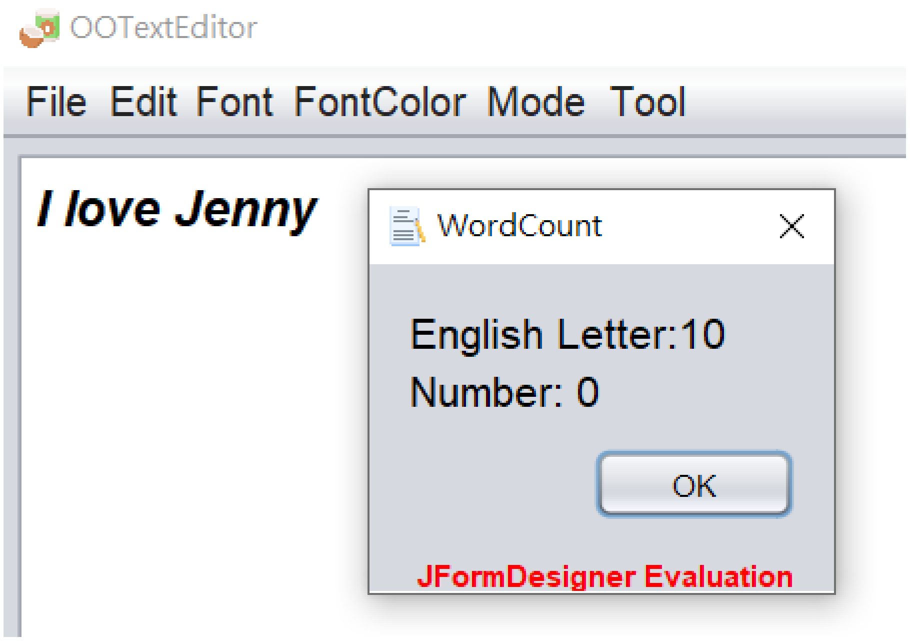
When we order it to create a Dialog, some subclasses like “WCSCFactory” will create it. This approach can separate the method in the superclass like method createDialog() from the method that actually creates the object, such as Dialog.

```java
public class WCSCFactory extends DialogFactory {
    public  MessageDialog createDialog(String type, String t) {
        if(type == "Word Count") {
            return new WordCount(t);
        }
        else if(type == "Spell Cheeck") {
            return new SpellCheck(t);
        }
        else
            return null;

    }
```


Factory
<<interface>>
MessageDialog
+ showMessageDialog ( )
+ showMessageDialog (letter : int , num : int)
+ showMessageDialog (result : String , num : int)
Dialog
+ showMessageDialog ( )
+ showMessageDialog (result : String , num : int)
+ showMessageDialog (letter : int , num : int)
SpellCheck
- dialog : JPanel
- textPanel : JPanel
- text : JLabel
- scroll : JScrollPane
- textArea1 : JTextArea
- buttonBar : JPanel
- okBT : JButton
+create (t : String : void
+ showMessageDialog ( ) : void
+ showMessageDialog (result : String , num : int) : void
+ showMessageDialog (letter : int , num : int) : void
- okBTActionPerformed (e : ActionEvent) : void
WordCount
- dialog : JPanel
- textPanel : JPanel
- text : JLabel
- text2 : JLabel
- buttonBar : JPanel
- okBT : JButton
+ create (t : Sting) : void
+ showMessageDialog ( ) : void
+ showMessageDialog (result : String , num : int) : void
+ showMessageDialog (letter : int , num : int) : void
- okBTActionPerformed (e : ActionEvent) : void
DialogFactory
+ createDialog (type : String , t : String) : MessageDialog
WCSCFactory
+ createDialog (type : String , t : String) : MessageDialog

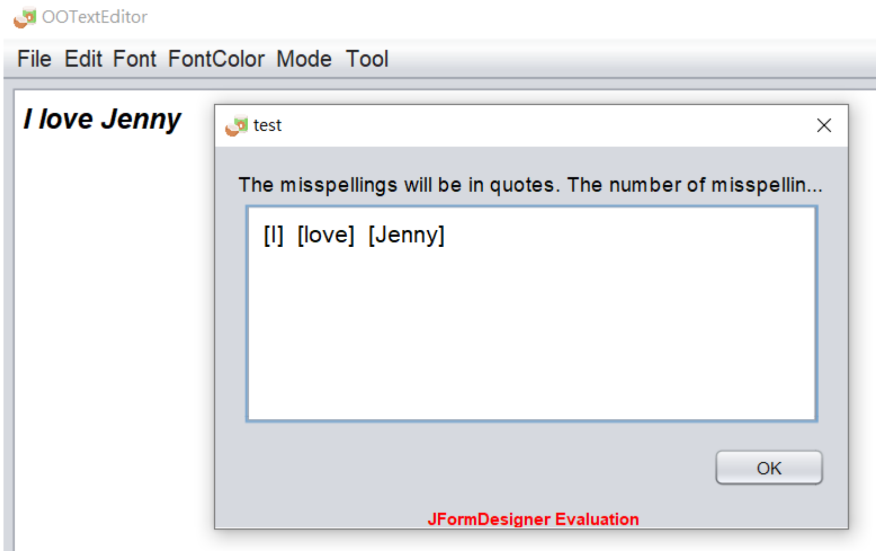
This is the application of the Factory pattern in our text editor. We can use "Word Count" to count how many English letter and number there are. And we use "Spell Check" to check the spell accuracy.





I love Jenny

English Letter:10
Number: 0



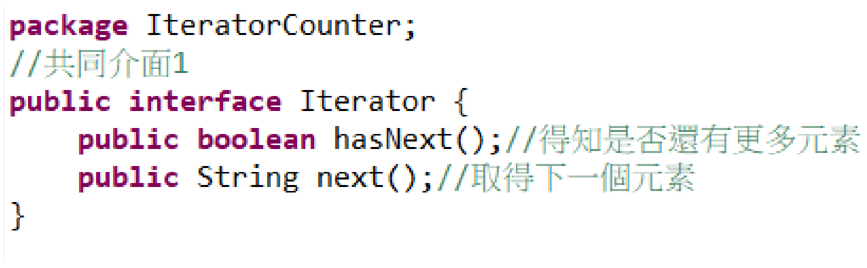
I love Jenny

The misspellings will be in quotes. The number of misspellin...

[I] [love] [Jenny]

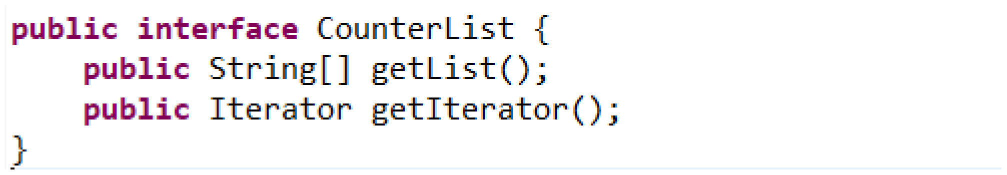
Iterator pattern

Iterator pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Interface “Iterator” is the common interface. And we have method hasNext() to check if there is another element. We also have method next() to get the next element.



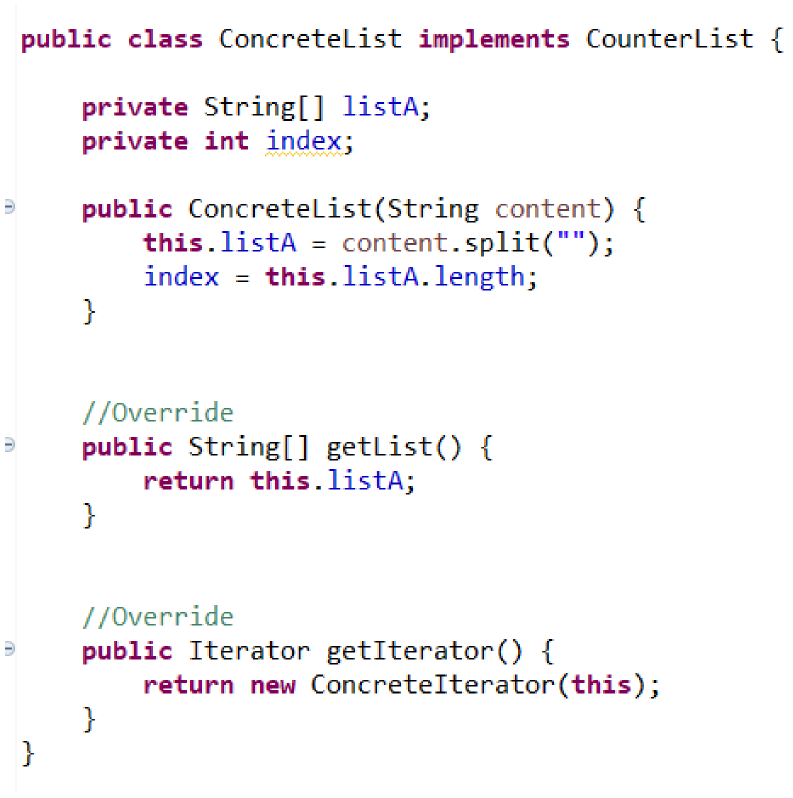
```java
package IteratorCounter;
//共同介面1
public interface Iterator {
    public boolean hasNext();//得知是否還有更多元素
    public String next();//取得下一個元素
}
```

“CounterList” is a simple interface to get the Iterator.



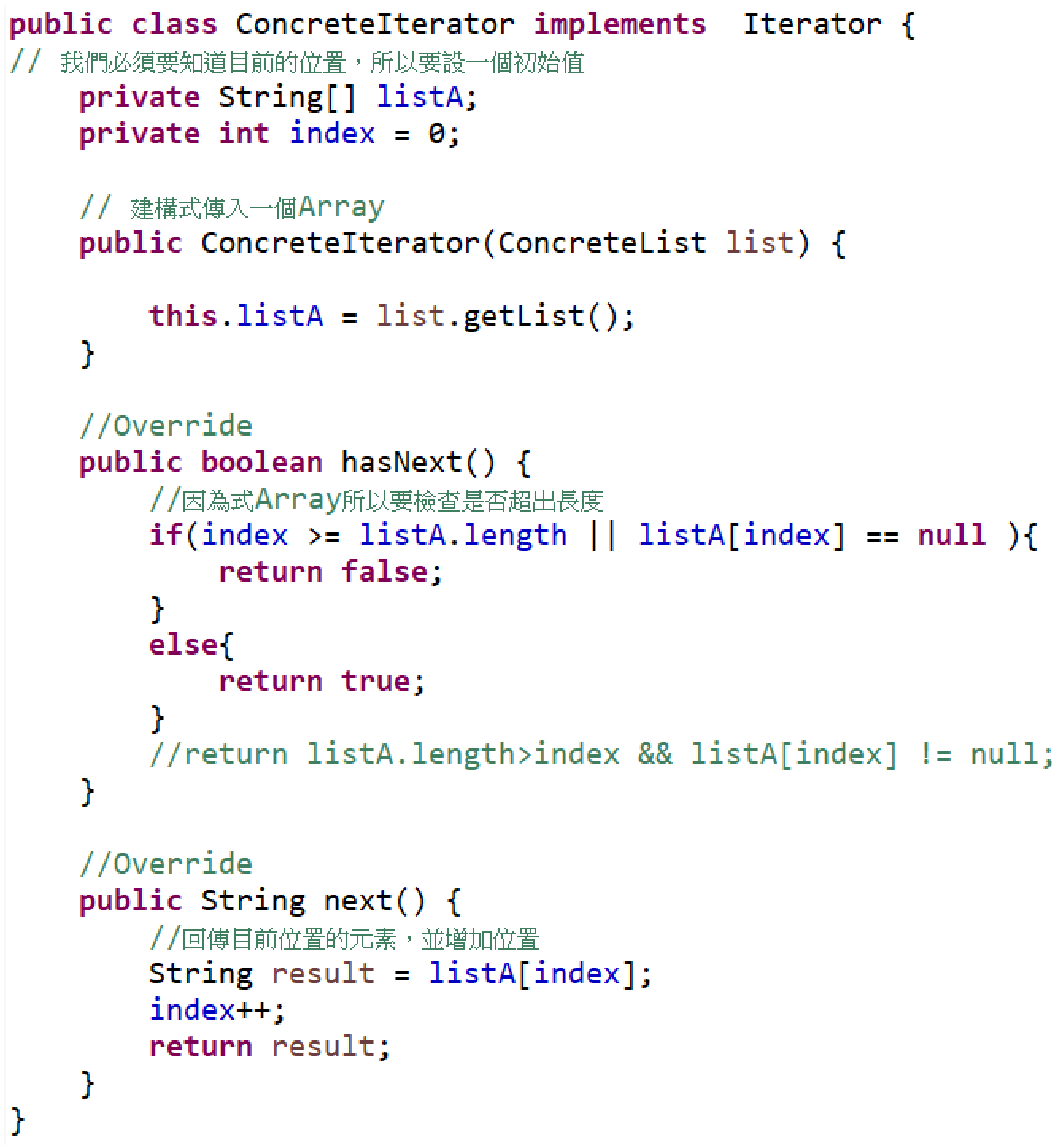
```java
public interface CounterList {
    public String[] getList();
    public Iterator getIterator();
}
```

In class “ConcreteList”, we can use method getIterator() to create a new Iterator.

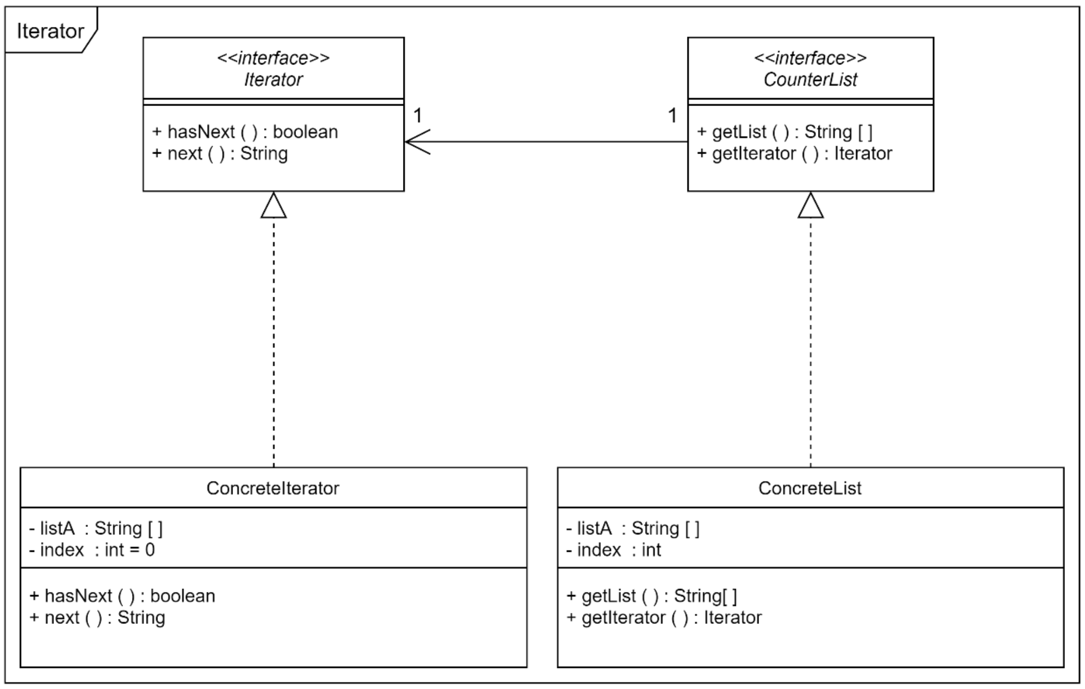


```java
public class ConcreteList implements CounterList {

    private String[] listA;
    private int index;

    public ConcreteList(String content) {
        this.listA = content.split("");
        index = this.listA.length;
    }


    //Override
    public String[] getList() {
        return this.listA;
    }


    //Override
    public Iterator getIterator() {
        return new ConcreteIterator(this);
    }
}
```

In class “ConcreteIterator”, we need to record the current location; therefore, we use index to set the initial value.
Then we use the method ConcreteIterator() to give the constructor an Array.
The method hasNext() can help us to check is there have another object; therefore, we have to check the length of the Array exceeds or not.
Finally, we use method next() to return where the current location is, and add the new location, so the next value is able to access.



```
public class ConcreteIterator implements  Iterator {
// 我們必須要知道目前的位置，所以要設一個初始值
    private String[] listA;
    private int index = 0;

    // 建構式傳入一個Array
    public ConcreteIterator(ConcreteList list) {

        this.listA = list.getList();
    }

    //Override
    public boolean hasNext() {
        //因為式Array所以要檢查是否超出長度
        if(index >= listA.length || listA[index] == null ){
            return false;
        }
        else{
            return true;
        }
        //return listA.length>index && listA[index] != null;
    }

    //Override
    public String next() {
        //回傳目前位置的元素，並增加位置
        String result = listA[index];
        index++;
        return result;
    }
}
```

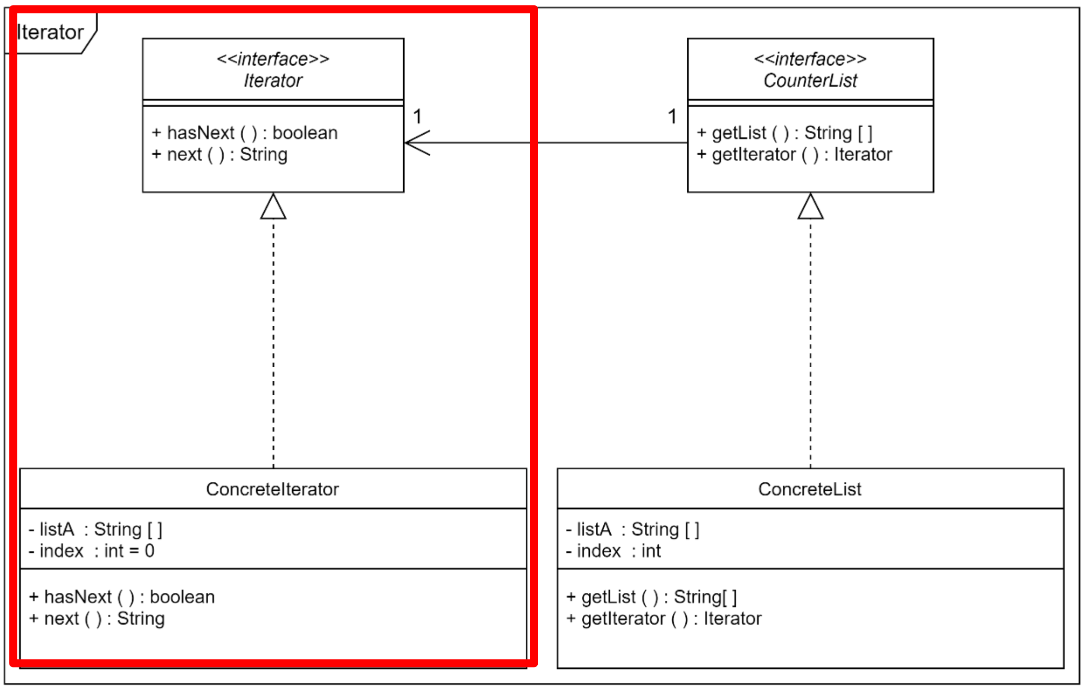

Iterator
<<interface>>
Iterator
+ hasNext ( ) : boolean
+ next ( ) : String
1
1
<<interface>>
CounterList
+ getList ( ) : String [ ]
+ getIterator ( ) : Iterator
ConcreteIterator
- listA : String [ ]
- index : int = 0
+ hasNext ( ) : boolean
+ next ( ) : String
ConcreteList
- listA : String [ ]
- index : int
+ getList ( ) : String[ ]
+ getIterator ( ) : Iterator

SOLID Principle

Next we will introduce which parts of our text editor comply with which “SOLID” principle.

Single Responsibility Principle(SRP)

We use Iterater Pattern to explain SRP. Interface “Iterator” was implemented by class “ConcreteIterator”. Interface “Iterator” has two methods : hasNext() and next(). There just have one reason for a class to change that is the method getIerator() in class “ConcreteList”, it will return the listA to class “ConcreteIterator”.
The following figure uses Iterater Pattern as examples.

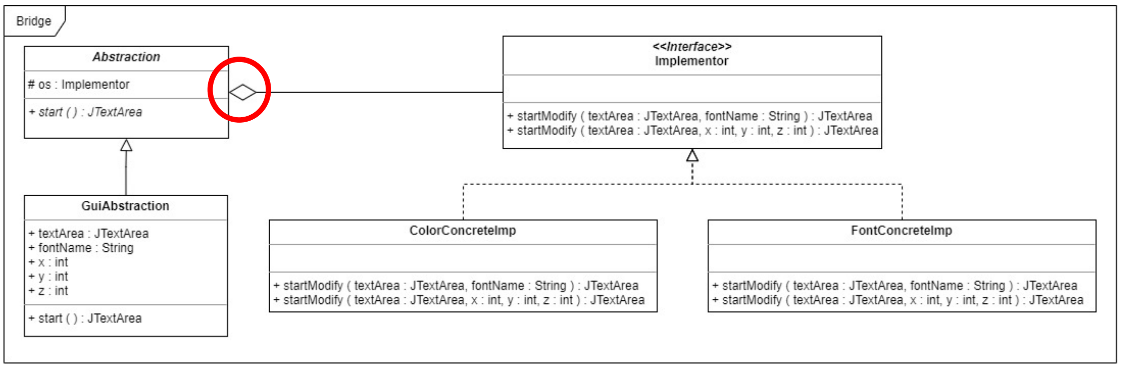


```
public interface Iterator {
    public boolean hasNext();
    public String next();
}
```

```
public class ConcreteList implements CounterList {
    private String[] listA;
    private int index;
    public ConcreteList(String content) {
        this.listA = content.split("");
        setIndex(this.listA.length);
    }
    //Override
    public String[] getList() {
        return this.listA;
    }
    //Override
    public Iterator getIterator() {
        return new ConcreteIterator(this);
    }
```

Open/Closed Principle(OCP)

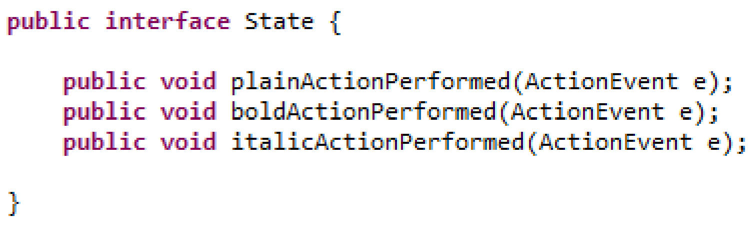
We use bridge pattern to explain OCP. Interface "Implementor" and the abstract class "Abstraction" establish the connection between the two classes through aggregation, rather than inheritance, which conforms to Open(open extension).
The abstract class is separated from its concrete implementation part, so that they can change independently and comply with Closed(closed modification).
The following figure uses Bridge Pattern as examples.


Bridge
Abstraction
# os : Implementor
+ start ( ) : JTextArea
<<Interface>>
Implementor
+ startModify (textArea : JTextArea, fontName : String) : JTextArea
+ startModify (textArea : JTextArea, x : int, y : int, z : int) : JTextArea
GuiAbstraction
+ textArea : JTextArea
+ fontName : String
+ x : int
+ y : int
+ z : int
+ start ( ) : JTextArea
ColorConcreteImp
+ startModify (textArea : JTextArea, fontName : String) : JTextArea
+ startModify (textArea : JTextArea, x : int, y : int, z : int) : JTextArea
FontConcreteImp
+ startModify (textArea : JTextArea, fontName : String) : JTextArea
+ startModify (textArea : JTextArea, x : int, y : int, z : int) : JTextArea

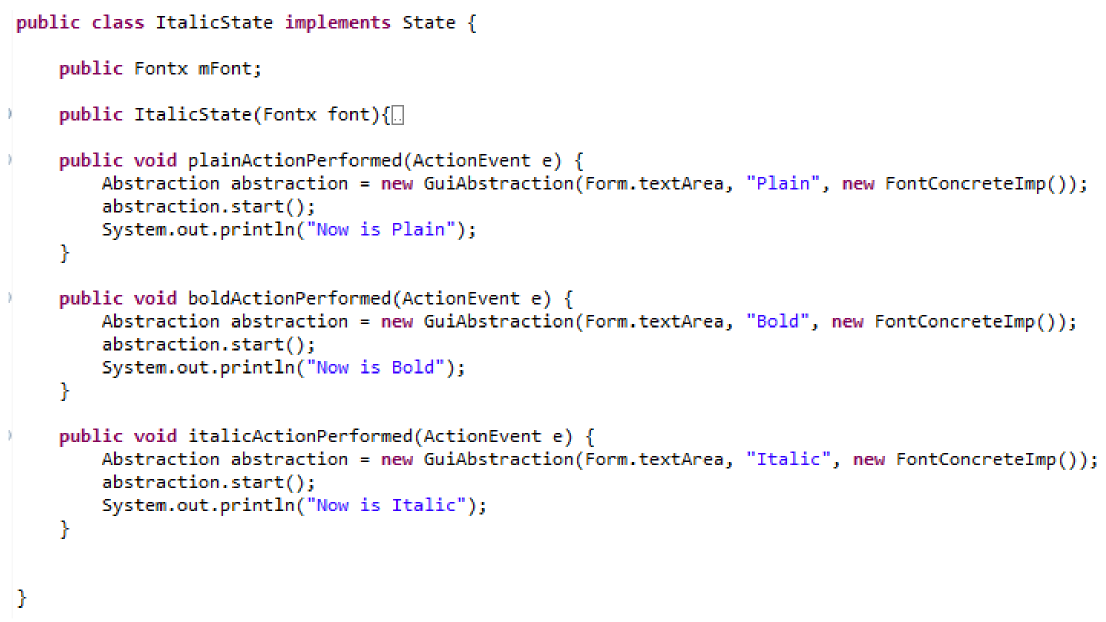
```java
public abstract class Abstraction {

    protected Implementor os = null;

    public Abstraction(Implementor os) {
        this.os = os;
    }

    public abstract JTextArea start();
}
```

Liskov Substitution Principle(LSP)

We use State Pattern to explain LSP. In interface “state”, there are three methods:plainActionPerformed(),boldActionPerformed(),italicAction-Performed().
Class “BoldState”, class “PlainState”, class “ItalicStatethree” methods of state, and they don’t change the content of the method.
The following figure uses interface “State” and class “ItalicState” as examples.



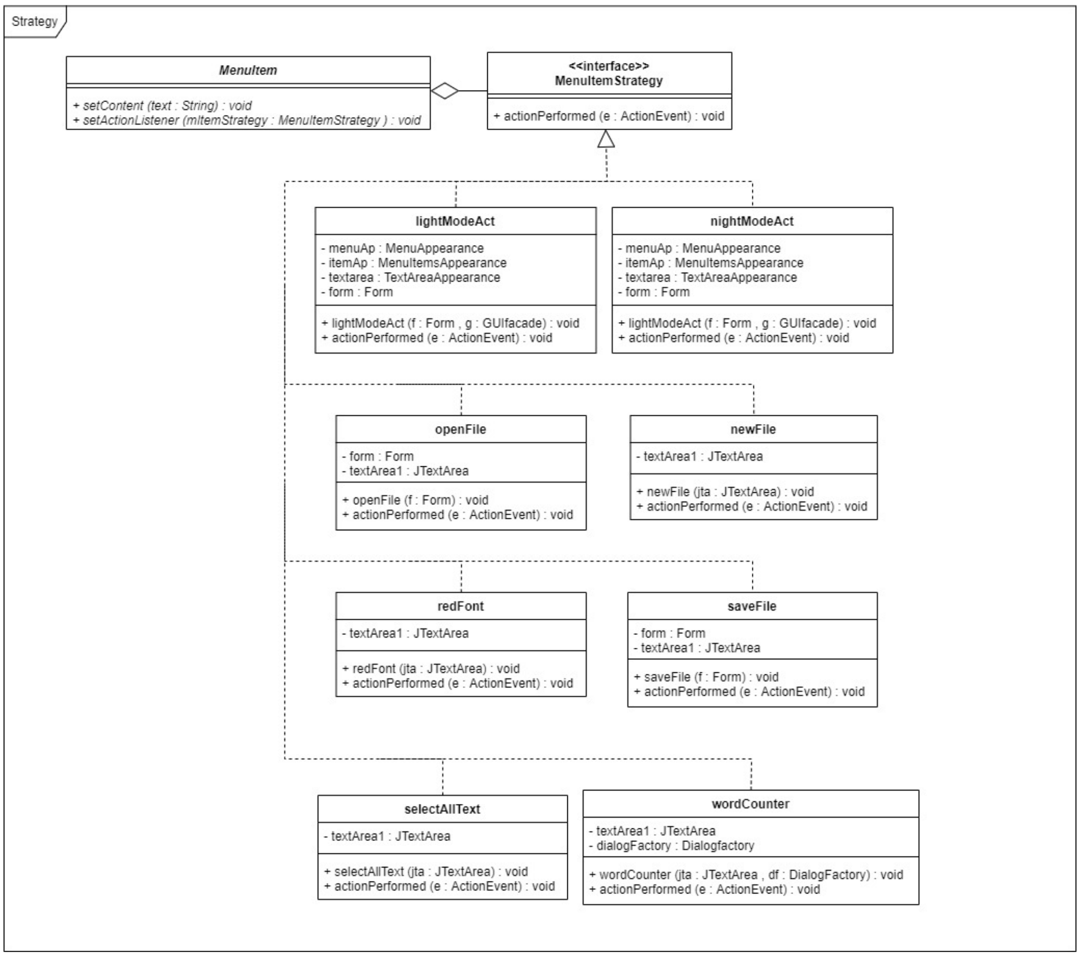
```java
public interface State {

    public void plainActionPerformed(ActionEvent e);
    public void boldActionPerformed(ActionEvent e);
    public void italicActionPerformed(ActionEvent e);

}
```

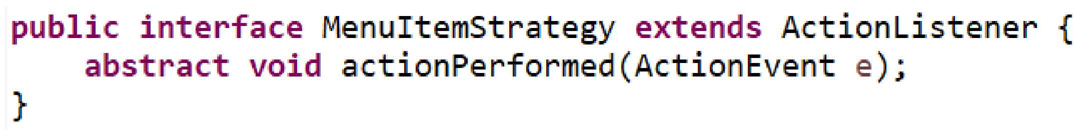


```java
public class ItalicState implements State {

    public Fontx mFont;

    public ItalicState(Fontx font){□

    public void plainActionPerformed(ActionEvent e) {
        Abstraction abstraction = new GuiAbstraction(Form.textArea, "Plain", new FontConcreteImp());
        abstraction.start();
        System.out.println("Now is Plain");
    }

    public void boldActionPerformed(ActionEvent e) {
        Abstraction abstraction = new GuiAbstraction(Form.textArea, "Bold", new FontConcreteImp());
        abstraction.start();
        System.out.println("Now is Bold");
    }

    public void italicActionPerformed(ActionEvent e) {
        Abstraction abstraction = new GuiAbstraction(Form.textArea, "Italic", new FontConcreteImp());
        abstraction.start();
        System.out.println("Now is Italic");
    }


}
```

Interface Segregation Principle(ISP)

We use Strategy Pattern to introduce the ISP. There is an interface class called “MenuItemStrategy”. It only has one method actionPerformed(). In this class, we keep the interface small and use the less interface to do more things.The following figure uses Strategy Pattern as examples.

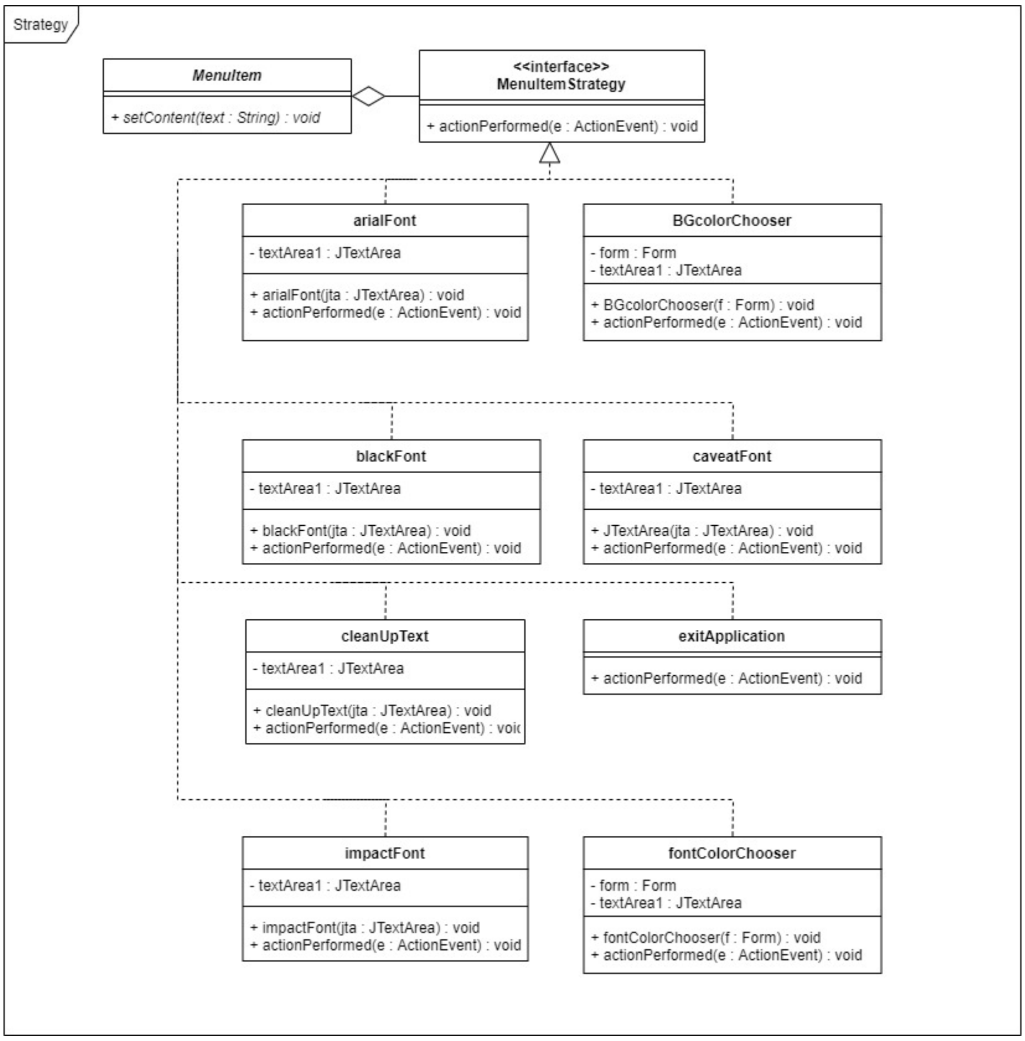

Strategy
MenuItem
+ setContent (text : String) : void
+ setActionListener (mItemStrategy : MenuItemStrategy) : void
<<interface>>
MenuItemStrategy
+ actionPerformed (e : ActionEvent) : void
lightModeAct
- menuAp : MenuAppearance
- itemAp : MenuItemsAppearance
- textarea : TextAreaAppearance
- form : Form
+ lightModeAct (f : Form , g : GUIfacade) : void
+ actionPerformed (e : ActionEvent) : void
nightModeAct
- menuAp : MenuAppearance
- itemAp : MenuItemsAppearance
- textarea : TextAreaAppearance
- form : Form
+ lightModeAct (f : Form , g : GUIfacade) : void
+ actionPerformed (e : ActionEvent) : void
openFile
- form : Form
- textArea1 : JTextArea
+ openFile (f : Form) : void
+ actionPerformed (e : ActionEvent) : void
newFile
- textArea1 : JTextArea
+ newFile (jta : JTextArea) : void
+ actionPerformed (e : ActionEvent) : void
redFont
- textArea1 : JTextArea
+ redFont (jta : JTextArea) : void
+ actionPerformed (e : ActionEvent) : void
saveFile
- form : Form
- textArea1 : JTextArea
+ saveFile (f : Form) : void
+ actionPerformed (e : ActionEvent) : void
selectAllText
- textArea1 : JTextArea
+ selectAllText (jta : JTextArea) : void
+ actionPerformed (e : ActionEvent) : void
wordCounter
- textArea1 : JTextArea
- dialogFactory : Dialogfactory
+ wordCounter (jta : JTextArea , df : DialogFactory) : void
+ actionPerformed (e : ActionEvent) : void



```java
public interface MenuItemStrategy extends ActionListener {
    abstract void actionPerformed(ActionEvent e);
}
```

Dependency inversion principle(DIP)

We use Strategy Pattern to introduce DIP. We put the methods of the text editor in the interface “MenuItemStrategy” to be implemented. Let class “MenuItem” depend on it instead of letting them all be associated to class “MenuItem” to implements methods. In this way, when we add a new function, we don’t need to modify the class “MenuItem”. Just add a new class for this function which can implement interface “MentItemStrategy” and let the “GUIFacade” call it.
The following figure shows the Strategy Pattern class diagram, the code of “MenuItem” and a part of the code of “GUIFacade”.


Strategy
MenuItem
+ setContent(text : String) : void
<<interface>>
MenuItemStrategy
+ actionPerformed(e : ActionEvent) : void
arialFont
- textArea1 : JTextArea
+ arialFont(jta : JTextArea) : void
+ actionPerformed(e : ActionEvent) : void
BGcolorChooser
- form : Form
- textArea1 : JTextArea
+ BGcolorChooser(f : Form) : void
+ actionPerformed(e : ActionEvent) : void
blackFont
- textArea1 : JTextArea
+ blackFont(jta : JTextArea) : void
+ actionPerformed(e : ActionEvent) : void
caveatFont
- textArea1 : JTextArea
+ JTextArea(jta : JTextArea) : void
+ actionPerformed(e : ActionEvent) : void
cleanUpText
- textArea1 : JTextArea
+ cleanUpText(jta : JTextArea) : void
+ actionPerformed(e : ActionEvent) : voic
exitApplication
+ actionPerformed(e : ActionEvent) : void
impactFont
- textArea1 : JTextArea
+ impactFont(jta : JTextArea) : void
+ actionPerformed(e : ActionEvent) : void
fontColorChooser
- form : Form
- textArea1 : JTextArea
+ fontColorChooser(f : Form) : void
+ actionPerformed(e : ActionEvent) : void

```java
package AbstractFactory;

import javax.swing.*;

import Strategy.MenuItemStrategy;

public abstract class MenuItem extends JMenuItem {
    public abstract void setContent(String text);
    public abstract void setActionListener(MenuItemStrategy mItemStrategy);

}
```

```java
public class GUIfacade {
    Menu menu;
    JTextArea textArea1;
    Form f;
    EditorFactory factory = new defaultFactory();
    private MenuItem selectAllMenuItem;
    private MenuItem cleanUpMenuItem;
    private MenuItem newMenuItem;
    private MenuItem openMenuItem ;
    private MenuItem saveMenuItem;
    private MenuItem exitMenuItem;
    private MenuItem arialMenuItem;
    private MenuItem impactMenuItem;
    private MenuItem caveatMenuItem;
    private MenuItem blackMenuItem;
    private MenuItem redMenuItem;
    private MenuItem choiceFontColor;
    private MenuItem colorBG;
    private MenuItem wordCountMenuItem;
    private MenuItem spellCheckMenuItem;
    private MenuItem nightmode;
    private MenuItem lightmode;
    private MenuItem FontStyle_IB;
    private MenuItem unFontStyle;
    private DialogFactory dialogfactory = new WCSCFactory();

 public Menu setFontMenu (Menu m ) {
     menu = m;
     menu.setContent("Font");

     //---- arialMenuItem ----
     arialMenuItem = factory.creatJMenuItem();
     arialMenuItem.setContent("Arial");
     arialMenuItem.addActionListener(new arialFont(textArea1));
     menu.add(arialMenuItem);

     //---- impactMenuItem ----
     impactMenuItem = factory.creatJMenuItem();
     impactMenuItem.setContent("Impact");
     impactMenuItem.addActionListener(new impactFont(textArea1));
     menu.add(impactMenuItem);

     //---- caveatMenuItem ----
     caveatMenuItem = factory.creatJMenuItem();
     caveatMenuItem.setContent("Caveat");
     caveatMenuItem.addActionListener(new caveatFont(textArea1));
     menu.add(caveatMenuItem);

     return menu;
 }
```

Teamwork Participation

Student ID	English name	Responsibility (Percentage) 負責項目/ 分工比例 ** 若同個項目有分工請記得寫 percentage	Score
B10821124	Leo	Coding(25%) Design Pattern(25%)	100%
B10823011	Kousa	Coding(25%) Design Pattern(25%)	100%
B10823015	Debby	Explain Pattern(33%) Class Diagram(16.6%)	100%
B10823016	Kris	Explain Pattern(33%) Class Diagram(16.6%)	100%
B10823018	Bob	Explain Pattern(33%) Class Diagram(16.6%)	100%
B10823024	Michael	SOLID (33%) Class Diagram(16.6%)	100%
B10823029	Leo	Coding(25%) Design Pattern(25%)	100%
B10823031	Andrew	SOLID (33%) Class Diagram(16.6%)	100%
B0823038	Joanne	SOLID(33%) Class Diagram(16.6%)	100%
B11023069	Young	Coding(25%) Design Pattern(25%)	100%