

# Assignment 1 / Object-Oriented Systems Design – A radio simulator

Kenan Esau

October 2000

Tutor: Mrs. C. Weiss  
Course: M.Sc Distributed Systems Engineering  
Lecturer: Mr. Prowse

## Contents

<b>1</b>	<b>Use Case analysis</b>	<b>3</b>
1.1	Classes of the display-component . . . . .	3
1.2	Classes of the radio-component . . . . .	4
<b>2</b>	<b>The design</b>	<b>5</b>
2.1	The GUI-/display-component . . . . .	5
2.1.1	The layout . . . . .	5
2.1.2	Static relations . . . . .	6
2.2	The radio-component . . . . .	7
2.2.1	Static relations . . . . .	7
<b>3</b>	<b>Collaboration diagrams</b>	<b>8</b>
3.1	System startup . . . . .	9
3.2	User action and display update . . . . .	9
<b>A</b>	<b>Source Code</b>	<b>10</b>
A.1	Display-Component . . . . .	10
A.2	Radio-Component . . . . .	17

# 1 Use Case analysis

In this part I will try to identify the classes which are needed to fulfill the requirements described in [1]. By analyzing the use cases it was only possible to find very few classes. The first important decision was that the radio simulator will be separated into two different components, – one component for the GUI and one component for the radio-functionality itself. The main class for the GUI-/display-component is called **Display**. The main class for the radio component is called **Radio**.

Since the two classes are implemented as two completely separated components, it will be possible to simply exchange the **Display**- or the **Radio**-class. An object of the **Radio**-class will be used to store all needed data like the frequencies and the volume level.

The classes **Radio** and **Display** will serve as facades [2] for their components (see section 2.2.1). The components are called the display- and the radio-component (see 1.1 and 1.2).

For the use case “Station Search” (see [1]) it would be required that the display of the frequency continuously increases but this is only possible if there would exist multiple threads. If there is only one thread the display will never be updated while the radio-component searches for a new frequency. Since the use of more threads is too complicated for this simple application it was decided to accept the effect that the display is not updated while the radio-component searches for a new station.

## 1.1 Classes of the display-component

To realize what kind of objects are needed to fulfill the requirements I looked at the tasks which should be possible to fulfill with the radio-simulator. It was also a great help to draw a sketch of the GUI to find out which kind of classes are needed. There are four main issues which should be covered by the GUI:

1. Display the current values of the frequency and the volume.
2. A button panel to save six stations.
3. A button panel to load six saved stations.
4. A panel which enables the user to control volume and frequency.

According to the four points mentioned above this component consists of the following classes:

- **Display**  
An object of this class acts as a communication partner for the

## 1.2 Classes of the radio-component

---

radio-component. It delegates the user-requests to its communication-partner of the radio-component – the class `Radio` – and receives the incoming results. It passes them on to the appropriate objects to display the results.

- `DisplayPanel`

The `DisplayPanel` is derived from `JPanel` and contains all subsequent panels. It acts as a kind of container for all other panels which are described below.

- `ControlPanel`

This panel is responsible for the control (see Point 4 above). It contains five buttons; two buttons to increase and decrease the volume as well as two buttons to increase and decrease the frequency and finally one button to mute the radio.

- `ViewPanel`

This class contains two text fields which are used to display the current values of the frequency and the volume.

- `ButtonPanel`

This class is the superclass of the two panels described below. This class summarizes the things which are common to the `Load-` and the `SaveButtonPanel`.

- `LoadButtonPanel`

This panel is derived from the `ButtonPanel`. It is responsible for creating the events to load a certain station from the radio-component.

- `SaveButtonPanel`

This panel is derived from the `ButtonPanel`. It is responsible for creating the events to save a certain station from the radio-component.

## 1.2 Classes of the radio-component

This section describes the responsibilities of the classes of the radio-component. The radio-component consists of the following three classes:

- `Radio`

This class acts as the facade (see [2]) for the hold radio-component. An object of this class contains objects of the two classes described below. The functionality of the `FrequencyControl` and the `VolumeControl` classes could also be implemented into the `Radio`-class but since the `Radio`-class already acts as a facade and has to organize the updates of the GUI-component this functionality was implemented into two separated classes, which are described below.

- **FrequencyControl**  
This class is aggregated by the **Radio**-class. An object of this class has the task to control the six frequencies which can be stored and loaded by the user. It also controls the search for a new station.
- **VolumeControl**  
Again – aggregation is used. This class controls the actual volume and it is capable of increasing and decreasing the volume and it can mute the radio.

## 2 The design

In this section I will try to give an overview about the static relations between the classes. It is divided into two main parts – one for each component.

### 2.1 The GUI-/display-component

The following subsections explain what kind of layoutmanagers are used and where the event handling is implemented. Section 2.1.2 shows the relations between the java-classes and the classes introduced in section 1.1. Section 2.1.1 introduces the layout of the GUI.

#### 2.1.1 The layout

First the layout is introduced because the classes of the display-component can be easily derived from the layout of the GUI (see section 1.1). You will find that the sections in which the GUI is separated are the same as the classes which were chosen in section 1.1. Each section is encapsulated in its own class. There are two kinds of layout-managers used in this application:

1. **BorderLayout**
2. **GridBagLayout**

The **BorderLayout** is used as the top level layout. It is used by the class **DisplayPanel**. The two button panels which load and save stations are put to the **NORTH** and the **SOUTH** area of the **BorderLayout**. The controls for the volume and the frequency are in the **CENTER** area and the display of the current volume and frequency occurs in the **RIGHT** area of this panel. The **LEFT** area is empty.

All other panels are placed within the top level panel **DisplayPanel**. These panels all use the **GridBagLayout**. There are four subsequent panels which can be seen in figure 1. There is a panel with six buttons for loading the stations and one panel with six buttons for saving the

## 2.1 The GUI-/display-component

---

stations. One panel contains the five buttons to control volume and frequency and one panel with two textfields which display frequency and volume to the user. Each of this areas on the GUI is a separate class (see section 1.1).

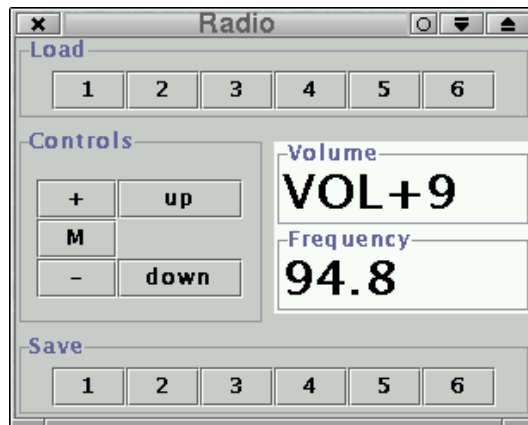


Figure 1: *GUI-Layout*

### 2.1.2 Static relations

The class `Display` acts as the facade for the GUI-component. This means every communication-partner which wants to do something with the display has to use this class as its interface. An object of this class delegates the tasks to the appropriate object. The class `Display` is derived from `JFrame` and contains only one panel which is the class `DisplayPanel`. The usage of the facade-pattern makes it easier to exchange one of the components. So if you want to use another implementation of the radio-component you could easily exchange them. Only the interfaces of the display- and the radio-component should be the same.

The class `DisplayPanel` is derived from `JPanel`. It is the top level panel which uses the `BorderLayout` (see section 2.1.1). It contains all subsequent panels which use the `GridBagLayout`. In this case aggregation is used.

In figure 2 you can see the relations between the java-classes and -interfaces and the classes of the display-component. As you can see all panels are derived from `JPanel`. Since the two button panels are very similar the things they have in common are summarized in their father class `ButtonPanel`.

All `ButtonPanel`-classes and the class `ControlPanel` implement the `ActionListener`-interface. By implementing this interface the classes become capable of catching the events which are generated if a button

## 2.2 The radio-component

is pressed by the user. The panels immediately invoke the appropriate method of the `Radio`-class of the radio-component. This is possible because the `Radio` class is globally available since it is implemented as a singleton [2]. The singleton design pattern is described in more detail in section 2.2.1 – and of course in full length in [2].

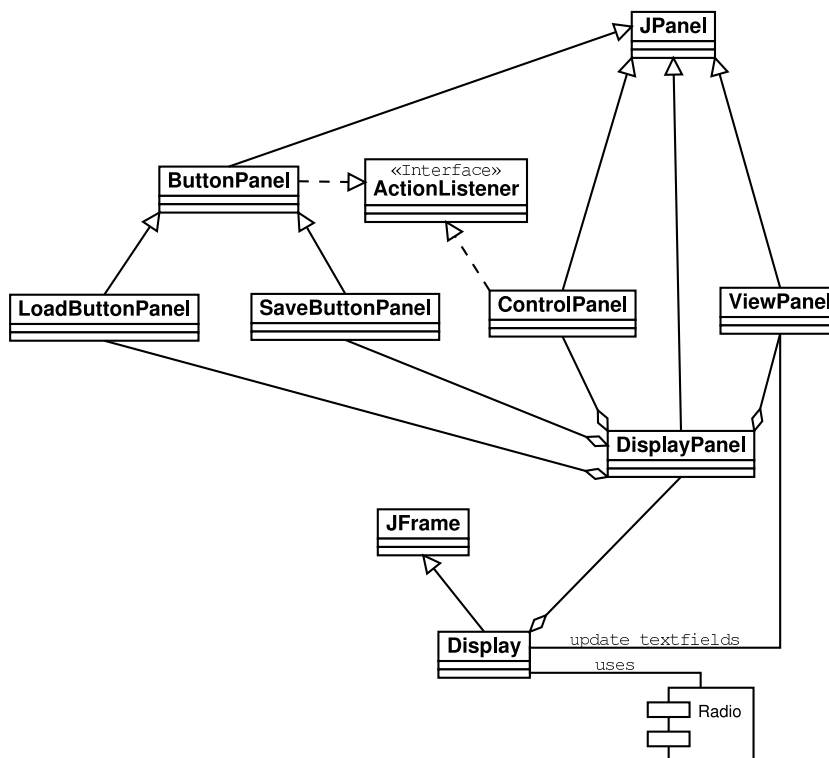


Figure 2: *Static class diagram of the display-component*

## 2.2 The radio-component

### 2.2.1 Static relations

The `Radio`-class is the counterpart of the `Display`-class of the GUI-component. The two components communicate with each other only by using these two facades. The `Radio`-class combines several design-patterns. The first is the facade, which means that the `Radio` class encapsulates a component which consists of many classes. If an object of another component wants to use an object of the actual component it has to call a method of the facade-object. The facade-object delegates the call to the appropriate object of the component (see [2]).

Additionally the `Radio`-class is observable. Which means that there can be multiple objects observing a `Radio`-object. If the data of the `Radio`-object changes the observing objects are informed about the change in

### 3 Collaboration diagrams

---

the data (see [2]). This pattern is also part of the MVC<sup>1</sup>-principle. In this case the display-component acts as the view and the controller and the radio-component contains the model which is controlled by the display.

Besides that the `Radio`-class is implemented as a singleton (see [2]) which means that there can be only one instance of this class at any time. This instance is globally available by the static method `getInstance()` of the `Radio`-class. By calling this method a `Display`-object can get a reference to the `Radio`-object. With this reference the `Display`-object can call every available method of the `Radio`-object (eg. `mute()`, `incVolume()`, ...)

Figure 3 shows – for sake of simplicity – only some of the methods of the classes and not all. For the complete details please refer to the source code in section A.2.

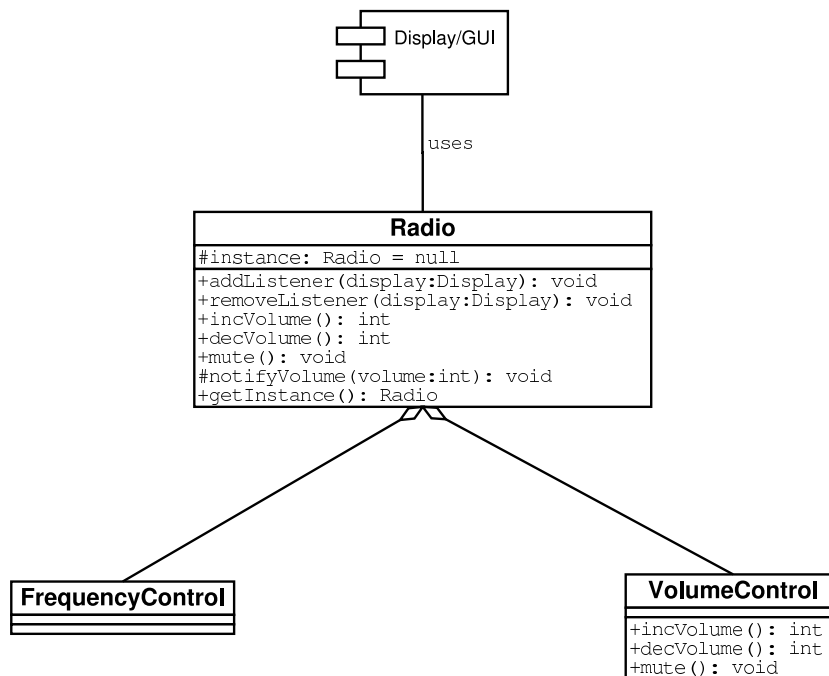


Figure 3: *Static class diagram of the radio-component*

### 3 Collaboration diagrams

In this section it is shown how the objects work together dynamically. The collaboration diagrams are based on the use cases. Normally the collaboration diagrams are used very early in the software development process but I think that they are also a very good way to explain how things work.

---

<sup>1</sup>Modell View Controller

### 3.1 System startup

First the process of starting the system is explained in detail. After the user started the program the `Radio`-class instantiates the `Display`-class (see figure 4 message 1.1). After that the `Display`-object calls the `addListener()`-method to subscribe itself to the list of listeners of the `Radio`-object. If any data of the `Radio`-object changes all subscribed listeners are informed of that change by a method-call from the `Radio`-object. At the moment there is only one listener per `Radio`-object but it will be easy to extend it to multiple listeners which means that there could be multiple GUIs connected to one radio.

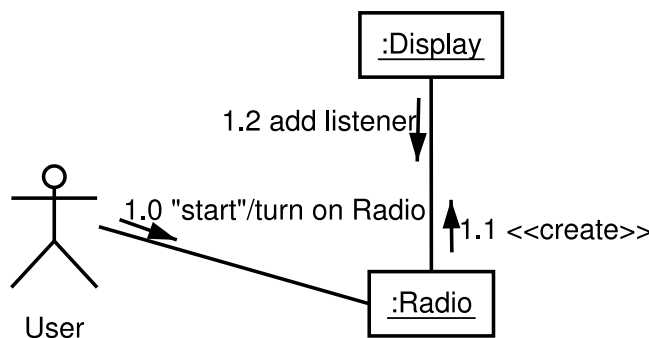


Figure 4: *Collaboration diagram of the startup*

### 3.2 User action and display update

Figure 5 is based on the use case “increase volume” but the collaboration diagram for “decrease volume”, “increase frequency” and the other use cases look nearly the same. Because of this, the use case “increase volume” is chosen to explain how the objects work together if a user pressed a button and how the “data” (eg volume) in the radio-component changes.

After the user pressed the button to increase the volume (Message 1.0) an object of the class `Display` calls the static method to get a reference to the `Radio`-object. After the `Display`-object got this reference it can call a method. In figure 5 it calls the method which increases the volume. The `Radio`-object delegates the task of increasing the volume to an object of the class `VolumeController` (Message 2.1). The `VolumeController`-object increases the volume and notifies the `Radio`-object to inform every listener that the volume has changed (Message 3.0). The `Radio`-object notifies every listener by iterating over its internal list of references to listeners and notifies them about the data change via method call. At the moment the internal list contains only one entry.

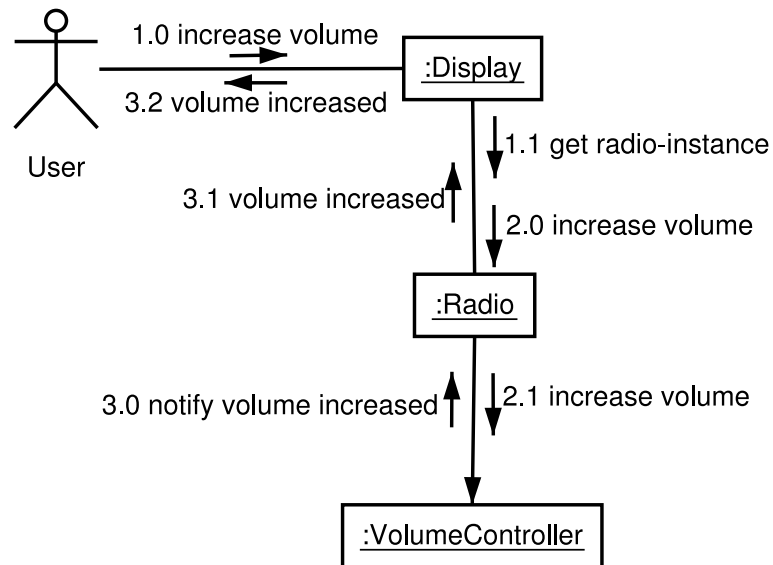


Figure 5: *Collaboration diagram of the use case “increase volume”*

## A Source Code

### A.1 Display-Component

```

import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;

class DisplayPanel extends JPanel
{
    public DisplayPanel(Display disp)
    {
        setLayout(new BorderLayout(5, 5));

        add(new LoadButtonPanel(), BorderLayout.NORTH);
        add(new SaveButtonPanel(), BorderLayout.SOUTH);
        add(new ControlPanel(), BorderLayout.CENTER);
        add(new ViewPanel(disp), BorderLayout.EAST);
    }
}

abstract class ButtonPanel extends JPanel
{
    private JButton oneButton;
    private JButton twoButton;
    private JButton threeButton;
    private JButton fourButton;
}

```

## A.1 Display-Component

---

```
private JButton fiveButton;
private JButton sixButton;

public ButtonPanel()
{
    GridBagConstraints c = new GridBagConstraints();
    GridBagLayout gridBag = new GridBagLayout();
    setLayout(gridBag);

    c.fill = GridBagConstraints.BOTH;
    c.anchor = GridBagConstraints.EAST;

    Dimension dim = new Dimension(100,25);
    Component sep = Box.createRigidArea(new Dimension\
        (10,200));

    oneButton = new JButton("1");
    oneButton.setPreferredSize(dim);
    c.gridx=0;
    c.gridy=0;
    gridBag.setConstraints(oneButton, c);
    oneButton.setActionCommand("1");
    oneButton.addActionListener(this);
    add(oneButton);

    twoButton = new JButton("2");
    twoButton.setPreferredSize(dim);
    c.gridx=1;
    c.gridy=0;
    gridBag.setConstraints(twoButton, c);
    twoButton.setActionCommand("2");
    twoButton.addActionListener(this);
    add(twoButton);

    threeButton = new JButton("3");
    threeButton.setPreferredSize(dim);
    c.gridx=2;
    c.gridy=0;
    gridBag.setConstraints(threeButton, c);
    threeButton.setActionCommand("3");
    threeButton.addActionListener(this);
    add(threeButton);

    fourButton = new JButton("4");
    fourButton.setPreferredSize(dim);
```

## A.1 Display-Component

---

```
        c.gridx=3;
        c.gridy=0;
        gridBag.setConstraints(fourButton, c);
        fourButton.setActionCommand("4");
        fourButton.addActionListener(this);
        add(fourButton);

        fiveButton = new JButton("5");
        fiveButton.setPreferredSize(dim);
        c.gridx=4;
        c.gridy=0;
        gridBag.setConstraints(fiveButton, c);
        fiveButton.setActionCommand("5");
        fiveButton.addActionListener(this);
        add(fiveButton);

        sixButton = new JButton("6");
        sixButton.setPreferredSize(dim);
        c.gridx=5;
        c.gridy=0;
        gridBag.setConstraints(sixButton, c);
        sixButton.setActionCommand("6");
        sixButton.addActionListener(this);
        add(sixButton);
    }
}

class LoadButtonPanel extends ButtonPanel
{
    public LoadButtonPanel(){
        setBorder(new TitledBorder("Load"));
    }

    public void actionPerformed(ActionEvent e)
    {
        int command = new Integer(e.getActionCommand()).\
            intValue();

        Radio.getInstance().loadStation(command);
    }
}

class SaveButtonPanel extends ButtonPanel
```

## A.1 Display-Component

---

```
{

    public SaveButtonPanel(){
        setBorder(new TitledBorder("Save"));
    }

    public void actionPerformed(ActionEvent e)
    {
        int command = new Integer(e.getActionCommand()).\
            intValue();

        Radio.getInstance().saveStation(command);
    }
}

class ControlPanel extends JPanel implements ActionListener
{
    protected JButton incVolButton;
    protected JButton decVolButton;
    protected JButton muteButton;
    protected JButton incFreqButton;
    protected JButton decFreqButton;

    public ControlPanel(){
        GridBagConstraints c = new GridBagConstraints();
        GridBagLayout gridBag = new GridBagLayout();
        setLayout(gridBag);

        c.fill = GridBagConstraints.BOTH;
        c.anchor = GridBagConstraints.NORTH;

        Dimension dim = new Dimension(100,25);
        Component sep = Box.createRigidArea(new Dimension\
            (10,200));

        incVolButton = new JButton("+");
        incVolButton.setPreferredSize(dim);
        c.gridx=0;
        c.gridy=0;
        gridBag.setConstraints(incVolButton, c);
        incVolButton.setActionCommand("incVol");
        incVolButton.addActionListener(this);
        add(incVolButton);
```

## A.1 Display-Component

---

```
muteButton = new JButton("M");
muteButton.setPreferredSize(dim);
c.gridx=0;
c.gridy=1;
gridBag.setConstraints(muteButton, c);
muteButton.setActionCommand("mute");
muteButton.addActionListener(this);
add(muteButton);

decVolButton = new JButton("-");
decVolButton.setPreferredSize(dim);
c.gridx=0;
c.gridy=2;
gridBag.setConstraints(decVolButton, c);
decVolButton.setActionCommand("decVol");
decVolButton.addActionListener(this);
add(decVolButton);

incFreqButton = new JButton("up");
incFreqButton.setPreferredSize(dim);
c.gridx=1;
c.gridy=0;
gridBag.setConstraints(incFreqButton, c);
incFreqButton.setActionCommand("incFreq");
incFreqButton.addActionListener(this);
add(incFreqButton);

decFreqButton = new JButton("down");
decFreqButton.setPreferredSize(dim);
c.gridx=1;
c.gridy=2;
gridBag.setConstraints(decFreqButton, c);
decFreqButton.setActionCommand("decFreq");
decFreqButton.addActionListener(this);
add(decFreqButton);

setBorder(new TitledBorder("Controls"));
}

public void actionPerformed(ActionEvent e)
{
    String command = e.getActionCommand();
    if (command.equals("incFreq")) Radio.getInstance().\
        incFrequency();
    if (command.equals("decFreq")) Radio.getInstance().\
```

## A.1 Display-Component

---

```
        decFrequency();
    if (command.equals("incVol")) Radio.getInstance().\
        incVolume();
    if (command.equals("decVol")) Radio.getInstance().\
        decVolume();
    if (command.equals("mute")) Radio.getInstance().mute();
    }
}
```

```
class ViewPanel extends JPanel
{

    protected JTextField volDisplay;
    protected JTextField freqDisplay;

    public ViewPanel(Display display)
    {
        GridBagConstraints c = new GridBagConstraints();
        GridBagLayout gridBag = new GridBagLayout();
        setLayout(gridBag);

        c.fill = GridBagConstraints.BOTH;
        c.anchor = GridBagConstraints.EAST;

        Dimension dim = new Dimension(100,50);
        Component sep = Box.createRigidArea(new Dimension\
            (10,200));

        Dimension dim2 = new Dimension (150,50);

        volDisplay = new JTextField();
        volDisplay.setPreferredSize(dim2);
        volDisplay.setFont(new Font("Arial", Font.BOLD, 28));
        c.gridx=0;
        c.gridy=0;
        gridBag.setConstraints(volDisplay, c);
        volDisplay.setActionCommand("decFreq");
        volDisplay.setBorder(new TitledBorder("Volume"));
        add(volDisplay);

        freqDisplay = new JTextField();
        freqDisplay.setPreferredSize(dim2);
        freqDisplay.setFont(new Font("Arial", Font.BOLD, 28));
        c.gridx=0;
```

## A.1 Display-Component

---

```
c.gridy=1;
gridBag.setConstraints(freqDisplay, c);
freqDisplay.setActionCommand("decFreq");
freqDisplay.setBorder(new TitledBorder("Frequency"));
add(freqDisplay);

display.setVolumeField(volDisplay);
display.setFrequencyField(freqDisplay);
}
}

public class Display extends JFrame
{
    static protected JTextField volume = null;
    static protected JTextField frequency = null;

    public void setVolumeField(JTextField field){volume = field;}
    protected JTextField getVolumeField(){return volume;}
    public void setFrequencyField(JTextField field){frequency = \
        field;}

    public Display()
    {
        super("Radio");
        setBounds(30, 30, 300, 220);

        //Look & Feel des Betriebssystems übernehmen
        try{
            UIManager.setLookAndFeel(UIManager.\
                getSystemLookAndFeelClassName());
        }
        catch (Exception e){
            System.err.println("Konnte L&F nicht laden:" +e.\
                getMessage());
        }

        Radio.getInstance().addListener(this);
        System.out.println("Listener set");

        Display listener = this;
        addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        })
    }
}
```

## A.2 Radio-Component

---

```
    });

    getContentPane().add(new DisplayPanel(this));

    Radio.getInstance().incVolume();
    Radio.getInstance().incFrequency();
    setVisible(true);
}

public void setFrequency(int freq){
    frequency.setText( (new Double ((double)freq/10)).toString\
        ( ) );
}

public void setVolume(int vol){
    volume.setText( "VOL+"+(new Integer (vol)).toString() );
}

public void setMute(){
    volume.setText("MUTE");
}
}
```

## A.2 Radio-Component

```
import java.util.*;

class FrequencyControl
{
    private int frequency;

    private int [] stations = new int[10];
    private int [] savedStations = new int[6];

    public FrequencyControl(){
        for (int i=0; i<stations.length; i++){
            stations [i]=(int)(900+(Math.random()*200));
            System.out.println(stations [i] );
        }

        for (int i=0; i<savedStations.length; i++){
            savedStations[i]=(int)(900+(Math.random()*200));
            System.out.println(savedStations[i] );
        }
    }
}
```

## A.2 Radio-Component

---

```
        frequency=(int)((900+Math.random()*200));
    }

    public void incFrequency(){
        do {
            frequency = frequency + 1;
            if (frequency > 1100) frequency = 900;

            Radio.getInstance().notifyFrequency(frequency);
        } while (!stationFound());
    }

    public void decFrequency(){
        do {
            frequency = frequency - 1;
            if (frequency < 900) frequency = 1100;

            Radio.getInstance().notifyFrequency(frequency);
        } while (!stationFound());
    }

    protected boolean stationFound() {
        boolean found = false;

        for (int i=0; i<stations.length; i++){
            if (frequency==stations[i]) found = true;
            System.out.println(found+" station: FM+" +stations.\
                length);
        }
        System.out.println(" after for");
        return found;
    }

    public void loadStation(int i){
        frequency = savedStations[i-1];
        Radio.getInstance().notifyFrequency(frequency);
    }
    public void saveStation(int i){savedStations[i-1]=frequency;}
```

## A.2 Radio-Component

---

```
}

class VolumeControl{
    private int volume=8;
    private int oldVolume=8;
    private boolean isMute=false;

    public VolumeControl(){

    public int incVolume(){
        if (!isMute){
            volume++;
            if (volume > 20) volume = 20;
            Radio.getInstance().notifyVolume(volume);
        }
        return volume;
    }
    public int decVolume(){
        if (!isMute){
            volume--;
            if (volume < 0) volume = 0;
            Radio.getInstance().notifyVolume(volume);
        }
        return volume;
    }

    public void mute(){
        if (isMute){
            isMute=false;
            volume=oldVolume;
            Radio.getInstance().notifyVolume(volume);
        }
        else {
            isMute=true;
            oldVolume=volume;
            volume=0;
            Radio.getInstance().notifyMute();
        }
    }
}

public class Radio
{
    public static Radio instance = null;

    private Vector listeners = new Vector();
```

## A.2 Radio-Component

---

```
private FrequencyControl freqCtrl = new FrequencyControl();
private VolumeControl volCtrl = new VolumeControl();

public static Radio getInstance()
{
    if (instance==null) {
        instance = new Radio();
    }
    return instance;
}

protected Radio(){}

public int incVolume(){return volCtrl.incVolume();}
public int decVolume(){return volCtrl.decVolume();}
public void mute(){volCtrl.mute();}

public void saveStation(int stationNumber){freqCtrl.saveStation\
(stationNumber);}
public void loadStation(int stationNumber){freqCtrl.loadStation\
(stationNumber);}
public int incFrequency(){
    freqCtrl.incFrequency();
    return 1;
}
public int decFrequency(){
    freqCtrl.decFrequency();
    return 1;
}

public void addListener(Display display){
    listeners .add (display);
}
public void removeListener(Display display){
    listeners .remove(display);
}
protected void notifyFrequency(int frequency){

    for (int i=0; i<listeners . size (); i++) {
        System.out.println(" notifying ͡ listeners ͡" + listeners . \
            size ());
        ((Display)( listeners .elementAt(i))).setFrequency(\
            frequency);
    }
}
```

## A.2 Radio-Component

---

```
    }  
  }  
  
  protected void notifyVolume(int volume){  
    for (int i=0; i<listeners.size(); i++) {  
      System.out.println("notifying " + listeners.  
        size());  
      ((Display)(listeners.elementAt(i))).setVolume(volume);  
    }  
  }  
  
  protected void notifyMute(){  
    for (int i=0; i<listeners.size(); i++) {  
      System.out.println("notifying " + listeners.  
        size());  
      ((Display)(listeners.elementAt(i))).setMute();  
    }  
  }  
  
  public static void main (String[] args)  
  {  
    new Display();  
  }  
}
```

### References

- [1] Description of the task for the assignment for workshop AW1
- [2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides Design Patterns
- [3] Goll, Weiß, Rothländer Java als erste Programmiersprache