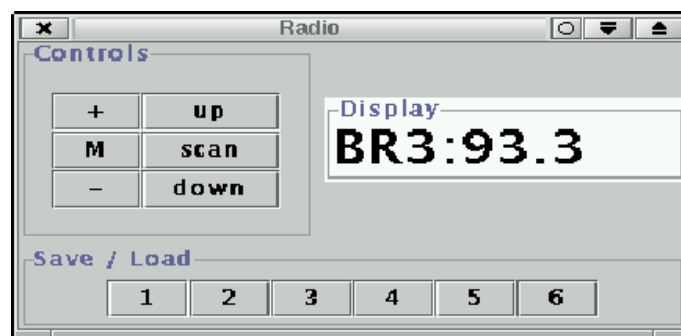


Assignment 2 Part 2/ Object Oriented Systems Design – Threads

Kenan Esau

November 2000



Tutor: Mrs. C. Weiss

Course: M.Sc Distributed Systems Engineering

Lecturer: Mr. Prowse

Contents

1	Introduction	3
2	Java Threads	3
2.1	Derive from the class Thread	3
2.2	Implementation of the interface Runnable	4
2.3	Mutual exclusion in Java	4
3	New design of the radio simulator	5
3.1	The new classes of the simulator	5
3.2	Static Relations of the Classes	6
4	Implementation of the new design	7
4.1	Implementation of the multifunction keys	7
4.2	Implementation of the multifunction display	8
4.3	Implementation of the search function	9
5	Conclusions	11

1 Introduction

This is the second part of the assignments dealing with the radio-simulator. The first part introduced the design for a single-threaded simulator. This part deals especially with the changes which were needed to make the simulator multithreaded and to introduce the new “station-search”-feature as described in [1]. As you can see in figure 1 the GUI was reduced to only one button panel to load and store stations and one display-area. The functionality remains the same the buttons of the panel can now load and store stations. If the volume is changed the new volume is displayed for two second and then the display is switched back to the frequency. In contrast to the first part this part is less theoretical because there are a lot of little pieces of code which – hopefully – explain what I want so say. The software-design for the radio-simulator remains more or less the same. The third (and last) part of the radio-simulator assignments will deal with incremental software development and design patterns.

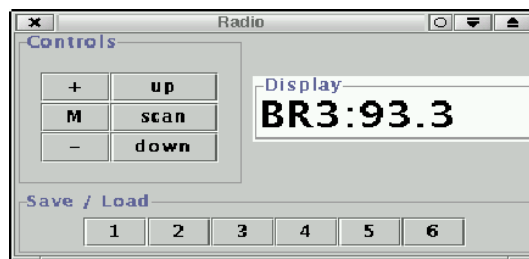


Figure 1: *New GUI-layout*

2 Java Threads

In Java there are two possibilities of creating a thread. They are discussed in the two sections 2.1 and 2.2. Section 2.3 deals with mutual exclusion and how it is implemented in Java.

2.1 Derive from the class Thread

This is the more simple solution. If you want to create a thread just derive from the class `java.lang.Thread` and overwrite the method `run()`. The method `run()` is executed during the state running of the thread this state is reached if you call the method `start()`. This solution has the advantage that it is very simple. The disadvantage is that, since Java lacks the feature of multiple inheritance, the class you created can only derive from the class `Thread`. An alternative to this solution is the one discussed in the next section.

2.2 Implementation of the interface Runnable

By implementing the interface `Runnable` it is also possible to create a thread. This solution has got the advantage that it is possible to derive from other classes. Starting this kind of thread is a little bit different from the solution discussed in section 2.1. In this case you can't just call the method `start()` to "execute" a thread since the new class is no thread – it just implements the interface `Runnable`. In this case a new thread can be started by creating an object of the class thread and passing its constructor a reference to a object of the new class implementing the interface `Runnable`. This solution would look like this:

```
Thread myThread = new Thread (new myThreadClass());
myThread.start();
```

Assuming that the class `myThreadClass` implements the interface `Runnable` and there for the method `run()`.

2.3 Mutual exclusion in Java

Mutual exclusion in Java is assured by monitors. Monitors are based on semaphores. A monitor encapsulates the data and the critical section which works on that data in one construct. The functionality of semaphores and monitors are the same [3].

Often it is not enough to have some simple critical sections. There are very often some conditions which have to be recognized. If the execution of a critical section depends on many conditions it gets often very confusing for the programmer if he uses semaphores.

In Java a monitor is built with the keyword `synchronized`. If there is a class called `Class1` with two methods `method1()` and `method2()` which are both synchronized the code would like in the example below. Assume that the two methods operate on the same data. There can never be more than one thread in those methods. If a second thread wants to enter an area which is marked as `synchronized` it has to wait until the first thread leaves this area.

```
public class Class1 {

    public synchronized void method1() {
        ...
    }

    public synchronized void method2() {
        ...
    }
}
```

Besides the possibility to mark methods as synchronized there is also the possibility to mark a single block of a method as synchronized. With this possibility you have to use a key-value which acts like a semaphore. It is grabbed if a thread enters the synchronized block and it is released again if the thread leaves the block. When the key is released again a new thread can enter the synchronized block. A key can be a reference to any object.

3 New design of the radio simulator

To introduce multi threading to the radio simulator there are several new classes needed. These new classes are introduced in section 3.1. Section 3.2 deals with the relations between the classes and the new design of the radio simulator.

3.1 The new classes of the simulator

- **SaveLoadButtonPanel**
According to the task description in [1] it is required to combine the functionality of the `SaveButtonPanel` and the class `LoadButtonPanel`. This combination is the class `SaveLoadButtonPanel` which implements the multifunction keys. For details of the implementation of the multifunction keys see section 4.1
- **FrequencyQueue**
This class provides a queue for objects of the classes `FrequencyQueueWorker` and `StationUpdateWorker`. In this queue there can be a maximum of four entries. This queue and the two worker-thread-classes are exclusively for the new “station-search”- functionality which is described in detail in section 4.3.
- **FrequencyQueueWorker**
This class describes a worker-thread which is responsible for filling the queue. If the queue is full it waits until another thread removes an entry from the queue.
- **StationUpdateWorker**
This class describes the worker-thread which removes entries from the queue. It waits until there are entries in the queue and then it “scrolls” the frequencies in 0.1-steps towards the value which it has taken from the queue. Between these steps it waits for 100 milliseconds. If it has reached the value it has taken from the queue it waits for 10 seconds and then it starts all over again.

The following classes are listed here only because of completeness. Those classes are not necessary for the threads-

3.2 Static Relations of the Classes

assignment. They are discussed in detail in [5].

- **StationEntry**
Objects of this class are capable of storing data for radio stations. This means in this class there is the station-name and the frequency stored. This class implements the interface **AbstractEntry** so it is possible to store **StationEntries** within an **StorageSysImpl**
- **StorageSysImpl**
This is a new class which is capable of storing entries which implement the interface **AbstractEntry** (see [5] for more details).
- **PersistentStorageSysImpl**
This class is derived from **StorageSysImpl** and is additionally persistent. Every change in the data stored by an object of this class is immediately written to the disk.
- **StationStorageSys**
Object of this class use one of the storage systems described above to store **StationEntry**-Objects (see [5]).

3.2 Static Relations of the Classes

As you can see in figure 2 the class diagram gets more and more complex. The new classes dealing with the various kinds of storage systems (eg. **StationStorageSys**, **StorageSysImpl**, ...) do not matter for this assignment but they are in the picture because of completeness. The bridge pattern which is used to introduce more flexibility in dealing with different kinds of storage systems is also described more closely in [5] for the special implementation used with the radio-simulator and in [2] for a more general description. The major changes done in this assignment cope with the display component (eg. changes in the button panels) and in the radio component with the class **FrequencyQueue** and the two threads which fill and empty the queue with entries.

The **FrequencyQueue**-object has to be accessed from a thread which needs information from the **FrequencyController** about all available stations. This thread is implemented with the class **FrequencyQueueWorker** which is responsible for filling the queue. But the queue has to be accessed from a thread which has to do the update of the display too. Because of this constellation of responsibilities these classes were placed within the **FrequencyController**. If the **Radio**-object calls **startStationSearch()** the **FrequencyController** starts a **FrequencyQueueWorker** and a **StationUpdateWorker**.

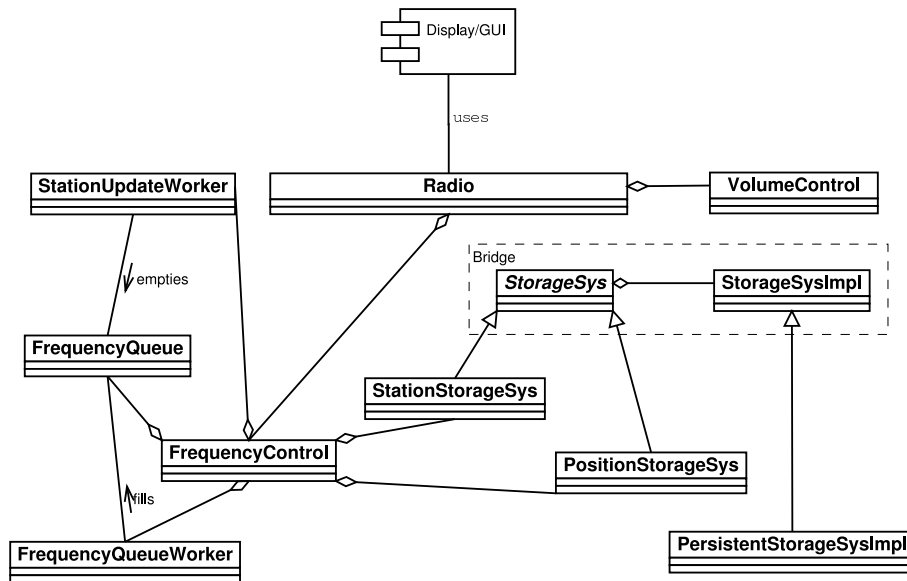


Figure 2: *The new class diagram*

4 Implementation of the new design

The following sections deal with the implementation of the various changes dealing with multi threading as demanded in [1].

4.1 Implementation of the multifunction keys

The two classes `LoadButtonPanel` and `SaveButtonPanel` are not needed anymore because the functionality of both panels should be implemented into one. The new class `SaveLoadButtonPanel` implements the methods for the event listeners `ActionListener` and `MouseListener`. The implementations for the methods `mouseClicked()`, `mouseEntered()` and `mouseExited()` are empty because they are not needed.

The methods `mousePressed()`, `mouseReleased()` and `actionPerformed()` are implemented. In the method `mousePressed()` the current `System-time` is saved. If the mouse button is released again the `mouseReleased()`-method will be called. In this method the time is measured again and if the difference of the both measured times is longer than two seconds the `saveStation()`-method of the `Radio`-class will be called otherwise the `loadStation()`-method will be called.

To accomplish this solution the super class `ButtonPanel` had to be changed in a way that it implements the `MouseListener`-interface additional to the `ActionListener`-interface. This has to be done in the super class because each button on the panel has to be registered with the `MouseListener` and because the buttons are created in the class `ButtonPanel` the registration has to take place here too.

4.2 Implementation of the multifunction display

In the code-snippet below you can see the implementation like it is described above. As you can see the solution is very simple. This is the reason why I decided not to use threads for this.

```
public void mousePressed(MouseEvent e){
    time = System.currentTimeMillis();
}

public void mouseReleased(MouseEvent e){
    if ( !( Radio.getInstance().isMute() ) ) {
        if ( System.currentTimeMillis()-time <= 2000){
            //pressed Button for only a short Time
            Radio.getInstance().loadStation(command);
        }
        else {
            //Button pressed longer than 2 sec
            Radio.getInstance().saveStation(command);
        }
    }
}

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

To implement the same functionality with threads it would need at least one thread which gets started and measures the time, if the mouse button is pressed and then, if the mouse button is released again the time has to be measured a second time, the thread has to be notified and then the two time values would have to be compared. As you can see it's pretty much the same as in the upper solution – a comparison of two time values. The solution using thread is only a little bit more complicated than needed.

4.2 Implementation of the multifunction display

In contrast to the implementation of the multifunction keys described in section 4.1 this solution is implemented using thread. Since the class `Display` is responsible for updating the display it has to implement the new functionality of displaying the volume for two seconds and then switching back to the frequency. Therefore it has to implement the interface `Runnable`. As you can see below in the little piece of code if the method `setVolume()` is called it displays the volume and then starts a thread which will switch back to the frequency after two seconds.

The variable `thread` is a member of the class `Display`. As you can see the constructor of the new created thread gets a reference to the `Display`-object itself. After this it is started and therefore the `run()`-method

4.3 Implementation of the search function

shown below is executed. The access to the counter named `threadCount` is protected by a `synchronized`-block. This way it is assured that if the user presses a volume button more than once the access to the variable `threadCount` is safe. For every click on an volume-button (increase/decrease volume) the user creates a new thread. Each of these threads increases the counter `threadCount` and then goes to sleep for two seconds. After a thread begins running again it checks whether it is the last thread or not (check if `threadCount` equals one). If this is the case it refreshes the display with the frequency. Before a thread dies it decreases the counter again.

```
public void setVolume(int vol){
    display.setText( "VOL+"+(new Integer (vol)).toString() );
    thread=new Thread(this);
    thread.start ();
}

public void run(){
    try {
        if (!Radio.getInstance().isMute()){
            //If Radio is NOT mute ...
            synchronized (key) {
                threadCount++;
            }
            thread.sleep(2000);
            synchronized (key) {
                if (threadCount==1) {
                    //After displaying the volume for 2 sec display freq. \
                    again
                    int freq = Radio.getInstance().getFrequency();
                    Radio.getInstance().notifyFrequency(freq);
                }
                threadCount--;
            }
        }
    }
    catch (InterruptedException e) {
        System.out.println (e);
    }
}
```

4.3 Implementation of the search function

The class `ControlPanel` was extended with a button which can invoke the station-search-function. By clicking this button once the function `startStationSearch()` of the `Radio`-class is invoked. By pressing this button twice the method `stopStationSearch()` of the `Radio`-class is in-

4.3 Implementation of the search function

voked.

```
if (command.equals("scan") && !(Radio.getInstance().isMute())\
    ){\
    if (scanning) {\
        Radio.getInstance().stopStationSearch();\
        System.out.println("called_stopStationSearch");\
        scanning=false;\
    }\
    else {\
        scanning=true;\
        Radio.getInstance().startStationSearch();\
        System.out.println("called_startStationSearch");\
    }\
}
```

The method `startStationSearch()` creates a new object of the class `FrequencyQueue` and after that it invokes the `FrequencyQueueWorker-Thread` and the `StationUpdateWorker-thread`.

As you can see in figure 3 the `FrequencyQueueWorker-thread` has to fill at least one entry into the queue before the `StationUpdateWorker` can start removing entries from it. Since the thread which is responsible for updating the display is always slower than the thread which inserts entries into the queue it can never happen to the `StationUpdateWorker-thread` that there are no entries in the queue.

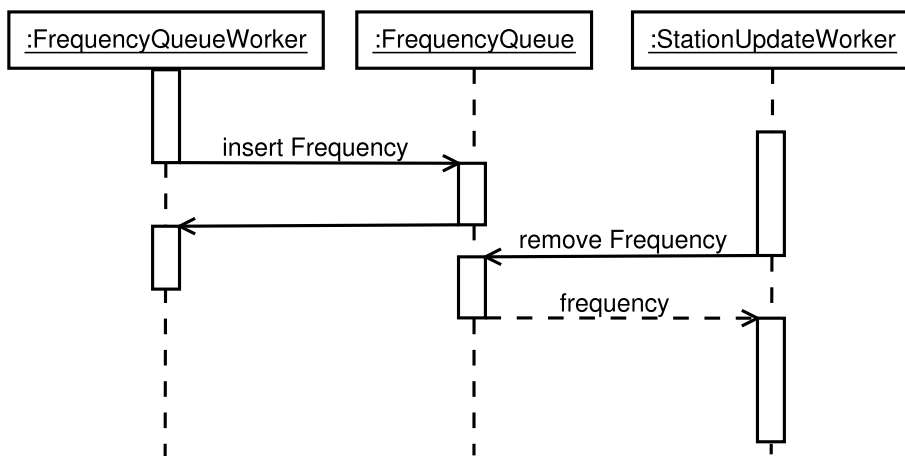


Figure 3: *Frequency-Queue*

Besides this the queue is designed in a way that the thread which inserts something in it has to wait if the queue is full. If a thread tries to remove something from an empty queue it has to wait too. This synchronization with sequence is achieved by a monitor around the two methods `insert()` and `remove()`.

5 Conclusions

As you can see it is quite simple to implement threads in Java. This assignment shows how threads can be implemented in Java and how mutual exclusion and synchronization with sequence is assured. It is also shown that there are only few changes needed for an architecture if you want to implement multithreading if you have had a good design before.

The concept of having lightweight processes which are part of the programming language makes the development of multithreaded applications a lot of easier but nevertheless Java still lacks the high performance which is offered by C or C++. For the C and C++-programming languages there are some very good libraries which can make the development of multithreaded applications a lot easier. To overcome the performance-problem with Java there are already some very interesting developments in compiler technology which make it possible to compile Java code to native code. GCJ is one of those compilers – you can even mix native code and Java-byte code – this gives an enormous boost in performance.

References

- [1] “Description of the task for the assignment for workshop AW2”
- [2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides “Design Patterns”
- [3] Goll, Weiß, Rothländer “Java als erste Programmiersprache”
- [4] Esau “Assignment Object-Oriented Systems Design – A Radio Simulator”
- [5] Esau “Assignment 2 Part 2 Object-Oriented Systems Design – Design Patterns and Evolution of Software-Architectures”