



RADIO INFO (OU3)

Name: Gazi Md Rakibul Hasan
Application Development (Java)
CS Name: ens20ghn



FEBRUARY 5, 2024

Introduction

The assignment aims to create an application that displays information about Swedish Radio programmes to the user. To run the program, we utilised Java Swing and multiple design patterns to ensure the application's internal structure is organised and easy to maintain. Furthermore, we have used Swing Worker to ensure the application's robustness, responsiveness and user-friendliness.

User Usage

To run the application, the user needs to compile the application. Compilation can be accomplished using the tools provided by modern text editors, such as **IntelliJ**. The user will access the **File** section at the editor's top. Following that, the user will see the structure and layout of the project. The user will be presented with a section depicting artefacts by selecting the Project Structure. The user can select the artefact and click the "+" symbol.

Consequently, the user will be prompted to choose the **jar file**, and the user will select it and add it **from the modules with dependencies**. Subsequently, the user will be able to see the creation structure of the jar file. The user will be asked to specify the location of the main class within the project, and the **Radio Info** class will be chosen.

Moreover, the user will select the checkbox that signifies the inclusion of all tests and the project build. Subsequently, the user will click the **"Apply"** and **"OK"** buttons. Then, the user will navigate to the Build section and click on **Build Project**. The presence of the jar file will be located at **Radio_Info_OU3_\out\artifacts\Radio_Info_OU3__jar**. Executing the jar file will allow us to observe the active execution of the application. It is recommended that the user extends the JVM memory. Figure 1 illustrates when the application begins running without any errors.

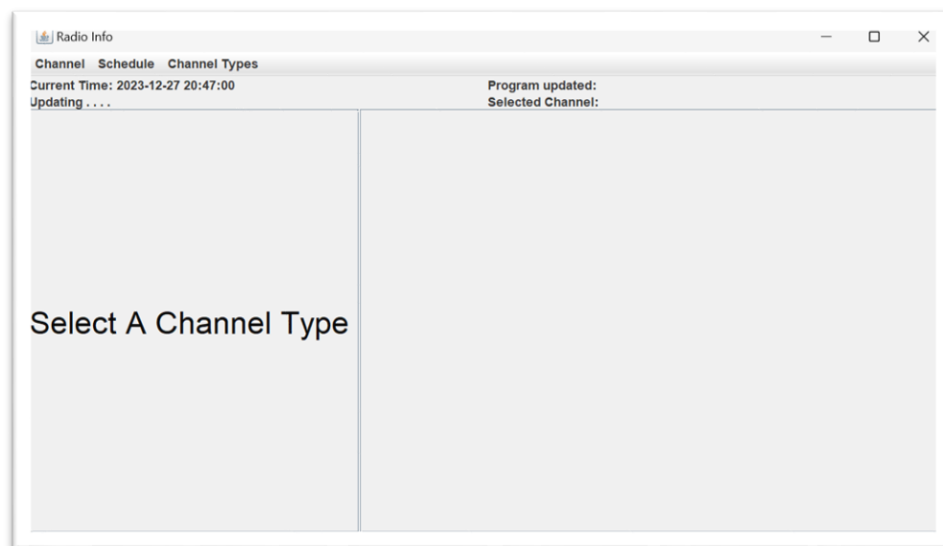


Figure 1: When the application starts to run successfully without any errors.

After running the application successfully, the user is ready to navigate through the application. Figure 2 shows a use case diagram illustrating how the user can operate with the application.

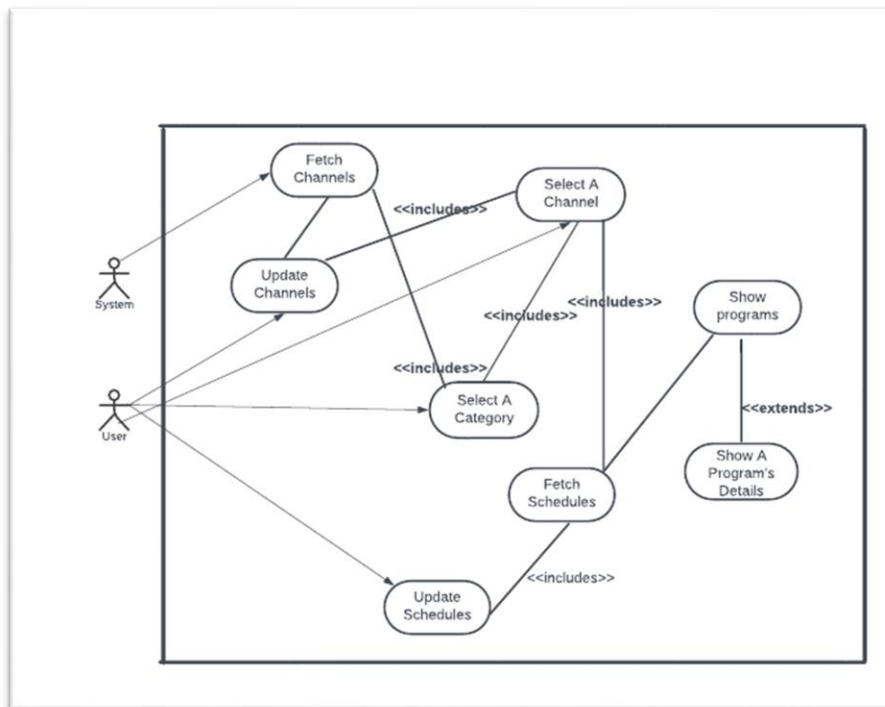


Figure 2: A use case diagram illustrating how users interact with the application.

Figure 2 illustrates that to choose a channel, we must select a category from the menu bar. Additionally, it is evident that the application initially retrieves channel data from the server (API). Once the data has been retrieved and parsed, the user can choose the channel category from the menu bar. After selecting the channel category from the menu bar, the user will see a list of channels. Figure 3 displays the outcome of choosing the " Rikskanal " category from the menu bar.

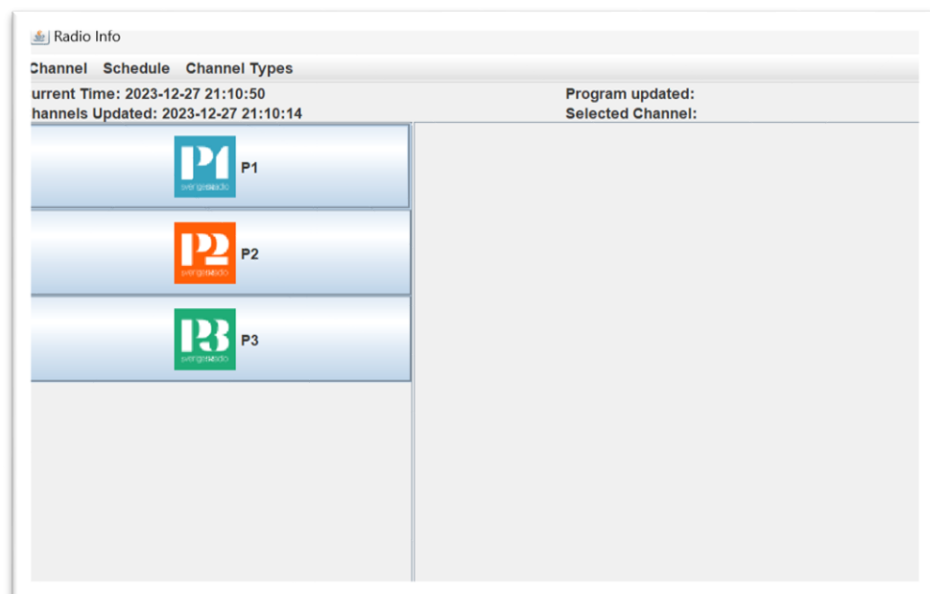



Figure 3: The result of selecting the category named "Rikskanal" As we can see, the channels P1, P2 and P3 display on the GUI.

Figure 2 also illustrates that the use case of selecting a channel involves choosing a specific channel category. By selecting a channel, the user will have the ability to view the programs associated with

that selected channel. To execute the use case, the user must choose a channel to display its programs. Figure 4 depicts when the channel "P1" is selected and its programs are displayed.



Program Name	Start Time	End Time
Ring P1!	2023-12-27 09:30	2023-12-27 10:00
Ekot senaste nytt	2023-12-27 10:00	2023-12-27 10:04
Plånboken	2023-12-27 10:04	2023-12-27 10:35
Känsligt läge	2023-12-27 10:35	2023-12-27 11:00
Ekot senaste nytt	2023-12-27 11:00	2023-12-27 11:03
Språket	2023-12-27 11:03	2023-12-27 11:33
Ekot senaste nytt	2023-12-27 11:33	2023-12-27 11:35
Ljudböcker från Radiofö...	2023-12-27 11:35	2023-12-27 12:00
Tolvslaget	2023-12-27 12:00	2023-12-27 12:01
Dagens dikt	2023-12-27 12:01	2023-12-27 12:09
Vetenskapsradion Fors...	2023-12-27 12:09	2023-12-27 12:30
Ekot 12:30	2023-12-27 12:30	2023-12-27 12:55
Land- och sjövärdar	2023-12-27 12:55	2023-12-27 13:00

Figure 4: By selecting the channel named P1, we can see the programs of channel P1 and each program's start and end times.

Figure 4 shows the start and end times of the channel's programs. One use case involves displaying the specific details of a program. This use case is an extension of displaying a channel's programs on the GUI. After selecting a program's name, the user can see the specific program's details. Figure 5 displays the result by clicking the program name "Ring P1!".

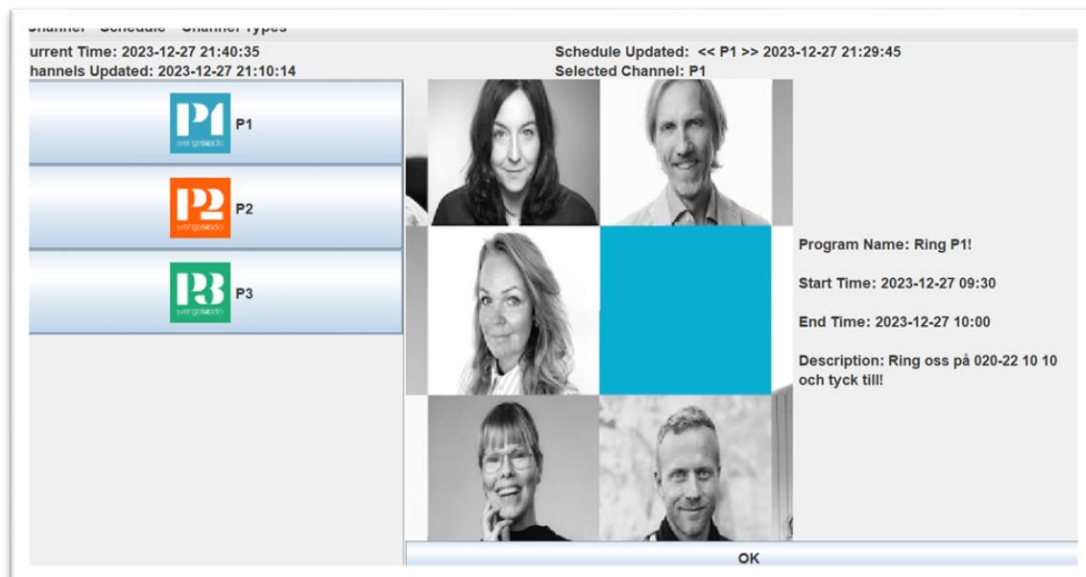


Figure 5: After clicking the program "Ring P1," we can see the program's start and end times. At the same time, we can see the description of the program.

Additional use cases are updating the channels. A channel's program schedules are shown in Figure 2. Updating channels and schedules involves updating all the channels and their respective programs.

We will use a sequence diagram to comprehensively describe our application's system and behaviour. The diagram will be pivotal in showing how our application's classes operate. Figure 6 illustrates the sequence diagram, which illustrates how our application flows and how each class cooperate with other classes throughout the application.

whenever the channels' parsing is completed. The Controller will manage the fetched data using the observer class. As shown in Figure 6, these fetched data will be used to create channels based on their categories in Step 5. Then, the user can select a specific category from the menu bar. Afterwards, the UI Manager will create and show those channels on the GUI.

The UI Manager class will utilise the components of the Programme View class to create these channels. When the user selects a channel to retrieve the program's information, the system will verify whether the data is stored in the cache. Suppose the data of the selected channel is stored in the cache. In that case, the Controller uses the cache class to retrieve the cached data. Subsequently, the cached data will be displayed on the GUI as the information for the channel programs. The application retrieves the data from the API if the selected channel's data is not stored in the cache. The Schedule Parser class will parse and analyse the data from the API. Due to the time-consuming nature of the task, we will utilise Swing Worker, which uses background threads to ensure the GUI remains responsive to the user. Following completion of the task of data parsing through the utilisation of background threads, the Controller class will eventually get notified. The Controller class will transmit the data to the UI Manager to update the GUI by populating the table with the relevant details regarding the program's name, start time, and end time. If a program name is clicked, the details will be displayed on the GUI, and the UI Manager class will be responsible for that task. When the user chooses to update a channel or a channel schedule, the updates will be made by retrieving data from the API using the Swing Worker class.

Design Patterns

We used an architecture design pattern called MVC, which refers to the Model, View and Controller. The Model contains information about the data that the application will use. The View contains those classes that will be used to construct GUI-related classes. The Controller will be the bridge between the View and the Model. In Figure 7, we illustrated the package diagram and showed how those packages can work together to implement the application.

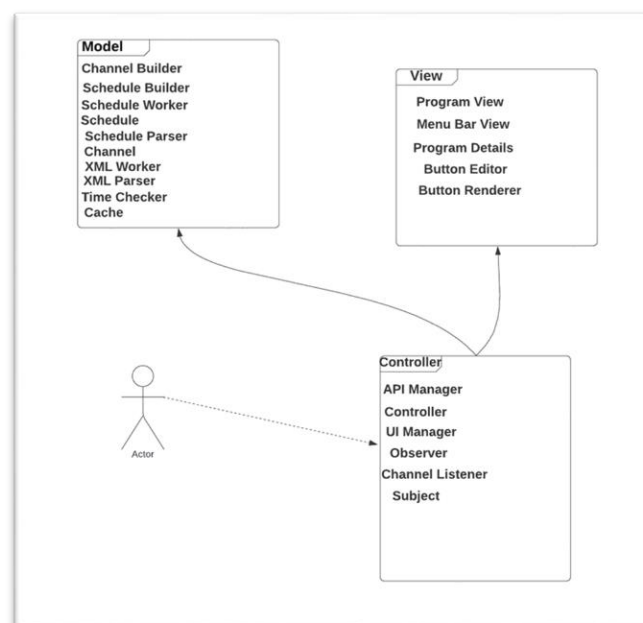


Figure 7: Package diagram that shows how MVC has been utilised in the application. Furthermore, it also includes those classes that are included in each package.

The observer pattern has been applied to our application. That design pattern is utilised when we fetch data from the API and notify the observers. We will utilise Swing Workers to parse and process data from the API. After the Swing Worker completes its task, it will inform the API Manager class (the observer) that the data has been retrieved. The XML Parser Worker class serves as the subject in the observer pattern. The role of the subject class is to notify the observer when specific scenarios emerge. The API Manager class serves as the observer class.

Similarly, we have implemented the observer pattern when the user chooses a specific channel. When the user chooses a channel from the GUI, the Controller class is notified to retrieve the relevant data about the channel's programs from the API. If the selected channel data is cached, it will not retrieve it from the API. Instead, it will update the UI based on the cached data. On the other hand, it will retrieve the data from the API. Again, the Swing Worker class will parse and store data using background threads. The Swing Worker class will serve as the subject. It will register and notify the observer class when a channel's program data is parsed. Afterwards, the API Manager class will get notified from the Schedule Worker class and transmit the data to the Controller class.

The builder design pattern has been utilised in the application. Given that certain classes in the application require multiple parameters to instantiate. Therefore, using the builder pattern to instantiate objects of the Channel and Schedule classes was crucial. We can construct Channel and Schedule classes using the Channel Builder and Schedule Builder classes.

In addition, we have implemented the strategy design pattern in the application. Both the XML Parser and Schedule Parser classes retrieve data from the API. However, they differ in terms of their algorithms. Therefore, the strategy design pattern is ideal in that particular scenario. We constructed an interface named the Data Fetch Strategy. Both the XML Parser and Schedule Parser will implement the interface.

Thread Safety

We have used Swing Workers (background threads) to execute time-consuming tasks such as parsing and fetching data from the API. The XML Parser Worker class extends from the Swing Worker class. It will override two inherited methods from its parent class. The `doInBackground` method will fetch the channel data from the API and parse it using the background threads. After completing its task, it will provide a list of channels retrieved and processed from the API. The `done` method will invoke the `get` method to retrieve the result from the `doInBackground` method since the `done` method in the Swing Worker class executes on the EDT, ensuring thread safety. Following that, we will transmit the notification to the observers. Figures 8 and 9 illustrate the implementation of those methods, the notification process of the API Manager class, and the data processing procedure.

```

@Override
protected ArrayList<Channel> doInBackground() {
    DataFetchStrategy <Channel> parser = new XMLParser();
    return parser.fetchData();
}

/**
 * That method will be called from the EDT, and it will get
 * of the background thread, and it categorize the channels
 */
Rakib
@Override
protected void done() {
    try {
        this.channels = get();
        notifyObservers();
    } catch (InterruptedException e) {
        JOptionPane.showMessageDialog( parentComponent: null,
    } catch (ExecutionException e) {
        JOptionPane.showMessageDialog( parentComponent: null,
    }
}

```

Figure 8: XML Parser Worker class will fetch data and parse channels. The channels will be stored in a list. It will notify the observers when the task is completed.

```

/**
 * That method invoked when channels have fetched from the API.
 * Thereafter, it notifies the Controller with the data.
 * @param channels The list of channels.
 */
1 usage Rakib
@Override
public void channelUpdate(ArrayList<Channel> channels) {
    this.channels = channels;
    HashSet<String> channelCategory = new HashSet<>();
    HashMap<String, ArrayList<Channel>> channelWithCategory = new HashMap<>();
    creatingChannelWithCategory(channelWithCategory);
    getTotalCategory(channelCategory);
    this.channelWithCategory = channelWithCategory;
    controller.updatedChannels(channelCategory, channelWithCategory);
}

```

Figure 9: It shows what happens when the observer class gets notified from the subject where the subject notified the observer with a list of channels. It will process the channels further to map the channel by its corresponding category.

Figure 10 shows how the channel and its corresponding category are updated on the GUI when data is passed to the Controller class. All GUI-related component updates will occur in the EDT thread. The *SwingUtilities.invokeLater* method ensures that all updates associated with the GUI components occur in the EDT. The XML Parser is exclusively utilised in the XML Parser Worker class and not

elsewhere in the application. We can deduce that we have ensured the thread safety while fetching channels from the API.

§

```
1 usage: rakib *  
public void updatedChannels (HashSet<String> types, HashMap<String,ArrayList<Channel>> channelWithType) {  
    channelWithTypeForTesting = channelWithType;  
    cache.clearCache();  
    SwingUtilities.invokeLater(() -> {  
        if(!types.isEmpty()){  
            uiManager.setupChannelButtons(types,channelWithType);  
            uiManager.setChannelUpdatedLabel();  
            resetAutomaticUpdates();  
        }  
    });  
}
```

Figure 10: The GUI components update occurs within the *SwingUtilities—invoke-later* method.

Figure 11 shows the implications of the user's channel selection. After choosing a channel, the Swing Worker class will retrieve the associated programs of the selected channel from the API. The observer class will be notified upon retrieving the data. The user can select several channels at the same time. Based on the selected channel, we will get/update the schema of the selected channel. Thus, we must apply the synchronisation mechanism here. In that case, we have utilised the Atomic Reference, which will hold the reference of the selected channel in a thread-safe manner. There will be two types of updates to the channel's schedule: the user initiates an update request and the automatic update. An automatic update is initiated when manually deciding to update all channels. In the application, an inbuilt system ensures that the application will update the cache's content every hour by fetching data from the API. We have used a boolean flag to distinguish between these two. In Figure 14, we have shown the instantiation of the Schedule Worker class. We created a new instance of the Schedule Worker for each channel associated with the boolean flag to mark whether it is an automatic or user-initiated update request.

```

@Override
protected ArrayList<Schedule> doInBackground() {
    DataFetchStrategy<Schedule> parser = new ScheduleParser(channel);
    return parser.fetchData();
}

Rakib
@Override
protected void done() {
    try {
        schedules = get();
        notifyObservers();
    } catch (InterruptedException e) {
        JOptionPane.showMessageDialog(parentComponent: null, message: "I
    } catch (ExecutionException e) {
        JOptionPane.showMessageDialog(parentComponent: null, message: "Ex
    }
}

1 usage Rakib
@Override
public void scheduleUpdate(Channel channel, ArrayList<Schedule> schedules, boolean isAutomaticUpdate) {
    if(!isAutomaticUpdate) {
        controller.processChannelAndSchedule(channel, schedules);
    }
    else {
        controller.processChannelAndScheduleForAutomaticUpdate(channel, schedules);
    }
}

```

Figure 11: That class will fetch and parse the selected channel's program schedules using the Swing Worker. It also displays how the API Manager class transmit the data to the Controller class.

Consequently, the observer class (API Manager) will transmit the retrieved data to the Controller class. The Controller class is responsible for modifying the GUI components associated with Swing. In our application, the user can choose channels while awaiting the completion of a channel's data retrieval task. Hence, it is essential to process only one thread at a time. Hence, a synchronised mechanism is necessary to prevent the race condition. To ensure that all GUI-related updates execute in the EDT, we utilised the `SwingUtilities.invokeLater` method. Inside this method, we have wrapped all GUI update methods. Figure 12 represents the method with synchronised mechanism and GUI-related updates that occur in the EDT. The exact process is deduced when the user selects a channel, and the data of that channel is already stored in the cache. After retrieving cached data, the GUI-related components will get updated in the EDT. To ensure that we have used the `SwingUtilities.invokeLater` method to wrap all GUI updates in it.

Figure 13 illustrates the method invoked when the automatic update is initiated. Since there could be many channels at the same time, they may get updated and want to update the cache simultaneously. Therefore, we must implement the synchronised mechanism to prevent race conditions and ensure thread safety.

```

3 usages  Rakib *
public synchronized void processChannelAndSchedule(Channel channel, ArrayList<Schedule> schedules) {
    uiManager.getChannelUpdateStatus().put(channel.getId(), Boolean.FALSE);
    cache.addSchedules(channel, schedules);
    SwingUtilities.invokeLater(() -> {
        selectedChannel.set(channel);
        uiManager.updateProgramTable(channel, schedules);
        uiManager.setScheduleUpdatedLabel(channel.getChannelName());
        menuBarView.updateScheduleListener();
    });
}

```

Figure 12: In the Controller class, GUI-related updates occur in the EDT, which was ensured using the *SwingUtilities.invokeLater*.

```

/**
 * That method will cache the given channel and its program schedules.
 * @param channel the channel.
 * @param schedules the given channel's programs schedules.
 */
1 usage  Rakib
public synchronized void processChannelAndScheduleForAutomaticUpdate(Channel channel, ArrayList<Schedule> schedules){
    cache.addSchedules(channel, schedules);
}

```

Figure 13: In the Controller class, when the automatic update gets triggered, that method will update the cache information by adding new data(schedule) to the corresponding channel (keys).

```

/**
 * Initiates the process of fetching schedule data for a specific channel.
 * It creates and executes a ScheduleWorker to perform the task in the background.
 * @param channel The channel for which the schedule data is to be fetched.
 */
3 usages  Rakib
public void fetchScheduleForChannel(Channel channel, boolean isAutomaticUpdate) {
    ScheduleWorker scheduleWorker = new ScheduleWorker(channel, isAutomaticUpdate);
    scheduleWorker.registerObserver(o: this);
    scheduleWorker.execute();
}

```

Figure 14: Schedule worker's class instance is created for each channel associated with the boolean flag, which marks whether it is an initiate update(when the user selects a channel on the GUI) or an automatic update. This instantiation methodology applies to maintain thread safety.