URCap Converting to Swing

Universal Robots A/S

Version 1.13.0

Abstract

As of URCap API version 1.3 support for Swing-based user interfaces (UIs) was introduced. This document will describe how to convert an existing HTML-based URCap into a Swing-based one. The document will not give an introduction to the URCap API. For more information on the URCap API see the URCap tutorial.

Contents

1 Introduction

For advanced user interfaces (UIs) it can be an advantage to have full control. This can be achieved by creating a Swing-based UI. If this is to be applied to an existing URCap with HTML-based UI, a number of steps must be performed to convert the URCap.

In the following sections the steps will be described using a program node contribution (implementing the `ProgramNodeContribution` interface) as a starting point. The same steps apply for an installation node contribution (implementing the `InstallationNodeContribution` interface).

2 Updates to pom.xml

First of all the `pom.xml` file must be updated to use the new URCap API supporting Swing-based UIs. The new version must be specified in two places, the dependency section and the import section in the bundle plugin.

2.1 Dependency Section

Search for the `<dependencies>` section where the URCap API dependency can be found. Update the version number of the API to 1.3 included in the URCap SDK or newer as shown in Listing 1.

Listing 1: Dependency section in the `pom.xml` file

```
<dependency>
  <groupId>com.ur.urcap</groupId>
  <artifactId>api</artifactId>
  <version>1.3.0</version>
  <scope>provided</scope>
</dependency>
```

2.2 Import-Package

Search for the `<Import-Package>` section where the packages required for runtime resolve of dependencies are described. Update the lower part of the version range to the specific version of the URCap API 1.13 included in the URCap SDK or newer as shown in Listing 2.

Listing 2: Import-Package section in the `pom.xml` file

```
<Import-Package>
  com.ur.urcap.api*,
  *
```

3 Recreating the User Interface

Create a new class that implements the `SwingProgramNodeView` interface (the *View*) and specify the existing implementation of the `ProgramNodeContribution` interface (the *Contribution*) as the type variable. In the `buildUI(JPanel, ContributionProvider<ProgramNodeContribution>)` method create the UI using regular Swing UI components and layout managers while using the supplied `JPanel` as the base of the UI.

Do not add the title in the UI as this is handled by PolyScope using the string returned by the `getTitle(Locale)` method of the `SwingProgramNodeService` (or `SwingInstallationNodeService` in case of an installation node) to ensure correct layout of this part.

Attach listeners to each component the end user should be able to interact with. The listeners should delegate the event to the `ProgramNodeContribution` implementation. For UI components that need to show a virtual keyboard or keypad (e.g. an input field), request the keyboard from the *Contribution* in a preconfigured state. Let the *View* be responsible for showing the keyboard/keypad while specifying the component it should attach to. In case of a keyboard this will allow the keyboard to be shown simultaneously with the associated component. In case of a keypad this will allow the keypad to be shown next to the associated component.

Add setter methods for all fields that should be able to show variable values so the *Contribution* can update the *View* according to the data model content.

Finally, remove the html and css files from the resources.

4 Implementing Swing-based Interfaces

The class implementing the `ProgramNodeService` interface must no longer implement said interface, but its Swing counterpart instead. The `SwingProgramNodeService` interface uses two generics to tie the *View* and *Contribution* together at compile time. Specify which *View* and *Contribution* classes or interfaces are being used as type variables. An example of an implementation of the `SwingProgramNodeService` is shown in listing 3.

Listing 3: Implementing the `SwingProgramNodeService` interface

```
1  public class MyProgramNodeService implements SwingProgramNodeService<
       MyProgramNodeContribution, MyProgramNodeView> {
2
3    @Override
4    public String getId() {
5      return "MyID";
6    }
7
8    @Override
9    public void configureContribution(ProgramNodeContributionConfiguration
         configuration) {
10     configuration.setChildrenAllowed(true);
11   }
12
13   @Override
14   public String getTitle() {
15     return "My␣Program␣Node";
16   }
17
18   @Override
19   public MyProgramNodeView createView() {
20     return new MyProgramNodeView();
21   }
22
23   @Override
24   public MyProgramNodeContribution createNode(URCapAPI api, MyProgramNodeView
         view, DataModel model, ProgramNodeCreationContext context) {
25     return new MyProgramNodeContribution(api, view, model);
26   }
27 }
```

Also update the `Activator` implementation to register the class as a `SwingProgramNodeService`-based service. The implementation of the `createView(ViewAPIProvider)` method in the interface `SwingProgramNodeService` must return an instance of the new *View* class mentioned in the previous section. The `createNode(ProgramAPIProvider, SwingProgramNodeView, DataModel, CreationContext)` method must return the *Contribution* as previously, but should pass along the *View* instance,

so the *Contribution* can interact with its *View*.

The configuration of a Swing-based program node contribution has been moved to the method `configureContribution(ContributionConfiguration)`. In the implementation of this method use the configuration argument to configure the contribution. If the original `ProgramNodeService` implementation also implemented the `NonUserInsertable` interface, it should no longer do so, but rather call the `setUserInsertable()` method on the configuration argument and set it to `false`.

In case of an installation node contribution, the `SwingInstallationNodeService` interface should be used instead, but otherwise the same guidelines should be followed.

5 Linking View and Contribution

The *View* instance and the *Contribution* must be able to communicate in order to pass values and react to events. Only one *View* instance exists and many instances of the *Contribution* could exist. In order for the *View* to call methods on the currently selected program node's underlying *Contribution*, the `ContributionProvider` object supplied to the `createNode(...)` method must be used. The `get()` method on the `ContributionProvider` will automatically return the *Contribution* instance representing the currently selected program node and in turn its associated data model.

The *Contribution* on the other hand was instantiated with the one and only *View* instance and can call methods on this instance without any further ado. The *Contribution* class needs to be updated to reflect that it now has a *View* and automatic binding to annotated (HTML) UI component fields no longer occurs. This means that instead of using the annotated fields directly, a method on the *View* instance should be called instead. Repeat this for all interactions with annotated fields in the *Contribution* class.

Similarly, for annotated methods representing UI events. These should be the methods called from the *View* when responding to a Swing event from a listener mentioned in section 3. Recreating the User Interface. In the case of an input field that brings up the virtual keyboard, the annotated method should be converted into a `KeyboardInputCallback` implementation. This callback is used as an argument when calling `show(JComponent, KeyboardInputCallback<T>)` on the `KeyboardTextInput` interface. An additional method should be created which will define how the keyboard configuration should be constructed.

A small example of an HTML-based event converted to a Swing-based one is shown in listing 4.

Listing 4: Code snippet for text field conversion

```
1  /// HTML-based text input field ///
2  @Input(id = "yourname")
3  public void onInput(InputEvent event) {
4    if (event.getEventType() == InputEvent.EventType.ON_CHANGE) {
5      setName(nameTextField.getText());
6      updatePopupMessageAndPreview();
7    }
8  }
9
10 ////////////////////////////////////////
11 /// Converted to a Swing-based approach ///
12 ////////////////////////////////////////
13
14 /// View excerpt ///
15 jTextField = new JTextField();
16 jTextField.setFocusable(false);
17 jTextField.setPreferredSize(style.getInputfieldSize());
18 jTextField.setMaximumSize(jTextField.getPreferredSize());
```

```
19  jTextField.addMouseListener(new MouseAdapter() {
20    @Override
21    public void mousePressed(MouseEvent e) {
22      KeyboardTextInput keyboardInput = provider.get().getKeyboardForTextField()
            ;
23      keyboardInput.show(jTextField, provider.get().getCallbackForTextField());
24    }
25  });
26
27
28  /// Contribution excerpt ///
29  public KeyboardTextInput getKeyboardForTextField() {
30      KeyboardInputFactory keyboardFactory = apiProvider.getUserInterfaceAPI().
            getUserInteraction().getKeyboardInputFactory();
31    KeyboardTextInput keyboardInput = keyboardFactory.createStringKeyboardInput
          ();
32    keyboardInput.setInitialValue(getName());
33    return keyboardInput;
34  }
35
36  public KeyboardInputCallback getCallbackForTextField() {
37    return new KeyboardInputCallback<String>() {
38      @Override
39      public void onOk(String value) {
40        setPopupTitle(value);
41        view.setPopupText(value);
42      }
43    };
44  }
45
46  public void setPopupTitle(final String value) {
47    undoRedoManager.recordChanges(new UndoableChanges() {
48      @Override
49      public void executeChanges() {
50        if ("".equals(value)){
51          model.remove(NAME);
52        } else {
53          model.set(NAME, value);
54        }
55      }
56    });
57  }
58
59  private String getName() {
60    return model.get(NAME, "");
61  }
```

All that remains is to remove the annotations from the methods as well as HTML-related arguments and to remove the class fields having annotations from the URCap API.

In case of an installation node contribution, only one instance of the *Contribution* exists, so there is no need for a provider.

6 Keyboards and Keypads

As mentioned in the previous section it is now required to programmatically bring up a virtual keyboard or keypad when needed. This is done through the `KeyboardInputFactory` interface by calling the appropriate create method. The create method returns a `KeyboardTextInput` or `KeyboardNumberInput<T>` instance where an error validator or an initial value optionally can be set. The `KeyboardInputFactory` instance can be accessed through the `UserInteraction` interface provided by the interface `UserInterfaceAPI`.

The `KeyboardTextInput` or `KeyboardNumberInput<T>` instance can afterwards be used to show the keyboard or keypad by calling the `show(JComponent, KeyboardInputCallback<T>)` method which takes the UI component to attach to and a callback when the user either clicks the OK or Cancel button.

The callback type is an abstract class that must implement the `onOK()` method and optionally implement the `onCancel()` method. Typically the `onOK()` method would contain code that sets a key and the value parameter in the data model.

The undo/redo of this model change must now be handled manually which will be described in the next section.

7 Handling Undo/Redo Support

All model or program tree changes must happen inside the scope of an `UndoableChanges` object using the `UndoRedoManager` interface to record the changes. Multiple changes to the model or program tree can happen inside a single `UndoableChanges` object and this means that all said changes will be undoable as a single action by the user. Below is a small code snippet demonstrating this.

Listing 5: Code snippet for undo/redo support

```
1 api.getUndoRedoManager().recordChanges(new UndoableChanges() {
2   @Override
3   public void executeChanges() {
4     model.set(NAME, "my name");
5     insertChildNodes();
6   }
7 });
```

Failing to record changes inside an `UndoableChanges` will throw a `IllegalStateExeption` exception. It is possible to execute code not related to the model or tree inside the scope of the `UndoableChanges` if need be.

Undoable actions only apply to a program node. Changes to a data model in an installation node do not have undo/redo support.