URCap Software Development Tutorial Swing

Universal Robots A/S

Version 1.11.0

Abstract

URCaps make it possible to seamlessly extend any Universal Robot with customized functionality. Using the URCap Software Platform, a URCap developer can define customized installation screens and program nodes for the end user. These can, for example, encapsulate new complex robot programming concepts, or provide friendly hardware configuration interfaces. This tutorial explains how to use the URCap Software Platform version 1.11.0 to develop and deploy URCaps for PolyScope version 3.14.0/5.9.0 using Swing-based user interfaces.

Contents

1 Introduction

The first official version of the URCap Software Platform (version 1.0.0) was released with PolyScope version 3.3.0. This tutorial describes features supported in version 1.11.0 of the URCap Software Platform which is released together with PolyScope version 3.14.0/5.9.0.

This platform can be used to develop external contributions to PolyScope that are loaded when PolyScope starts up. This makes it possible for a URCap developer to provide customized functionality within PolyScope.

For example, a customized installation screen can serve the end user to comfortably configure a new tool. Similarly, a customized program node may serve as a way to perform complex tasks while hiding unnecessary detail.

The layout of a customized screen, the behaviour of a customized node, data persistence and script code generation is all implemented in Java. The URCap along with its resources is packaged and distributed as a single Java jar-file with the `.urcap` file extension. A URCap can be installed from the Setup screen in PolyScope.

The tutorial is organized in the following manner:

- Section 2 to 3 explain what you need to start developing URCaps.
- Section 4 to 6 guide you through the basic project setup including build and deployment.
- Section 7 to 9 introduces the concept behind URCaps and explains the different software components.
- Section 10 provides an overview of technical URCap examples distributed with the SDK that focus on specific features of the URCap API.
- Section 11 demonstrates how to create an empty URCap project. We recommend that you also have a look at the examples when you want to start from scratch.
- Section 14 describes different debugging and troubleshooting options. Also visit the support forum at `www.universal-robots.com/plus`.

To get started we use the *Hello World Swing* and the *My Daemon Swing* URCaps as running examples. These are very simple and basic URCaps.

1.1 Features in URCap Software Platform 1.11.0

The following entities can be contributed to PolyScope using a URCap:

- Customized installation nodes and corresponding screens
- Customized program nodes and corresponding screens
- Daemon executables that run as separate background processes on the control box.

The customized installation nodes support:

- Saving and loading of data underlying the customized installation node as part of the currently loaded installation in PolyScope.
- Script code that an installation node contributes to the preamble of a robot program.

The customized program nodes support:

- Saving and loading of data underlying the customized program nodes as part of the currently loaded PolyScope program.
- Script code that a program node contributes to the script code generated by the robot program.

2 Prerequisites

A working version of Java SDK 6 is required for URCap development along with Apache Maven 3.0.5. You will also need PolyScope version 3.14.0/5.9.0 in order to install the developed URCap, if it is using URCap API version 1.11.0. Previous versions of the API will have lower requirements for the PolyScope version. A UR3, UR5, or UR10 robot can be used for that purpose or the Universal Robots offline simulator software (URSim). PolyScope and the offline simulator can be found in the download area of the tech support website at:

`www.universal-robots.com/support`

Select the applicable version and follow the given installation instructions. The offline simulator is available for Linux and non-Linux operating systems through a virtual Linux machine.

The script language and pre-defined script functions are defined in the script manual, which can also be found in the download area of the tech support website.

The URCap SDK is freely available on the Universal Robots+ website at:

`www.universal-robots.com/plus`

It includes the sources for the URCap examples.

The *My Daemon Swing* example of this tutorial additionally requires either Python 2.5 (compatible) or the Universal Robots urtool3 cross-compiler toolchain. The urtool3 cross-compiler is included in the SDK.

The following section describes the content of the URCap SDK.

3 URCap SDK

The URCap SDK provides the basics to create a URCap. It contains a Java package with the API that the developer will program against, documentation, the Hello World Swing and other URCap examples, the urtool3 cross-compiler toolchain and a means of easily creating a new empty Maven-based template URCap project (See section 11).

The URCap SDK is distributed as a single ZIP file. Figure 1 shows the structure of the file.

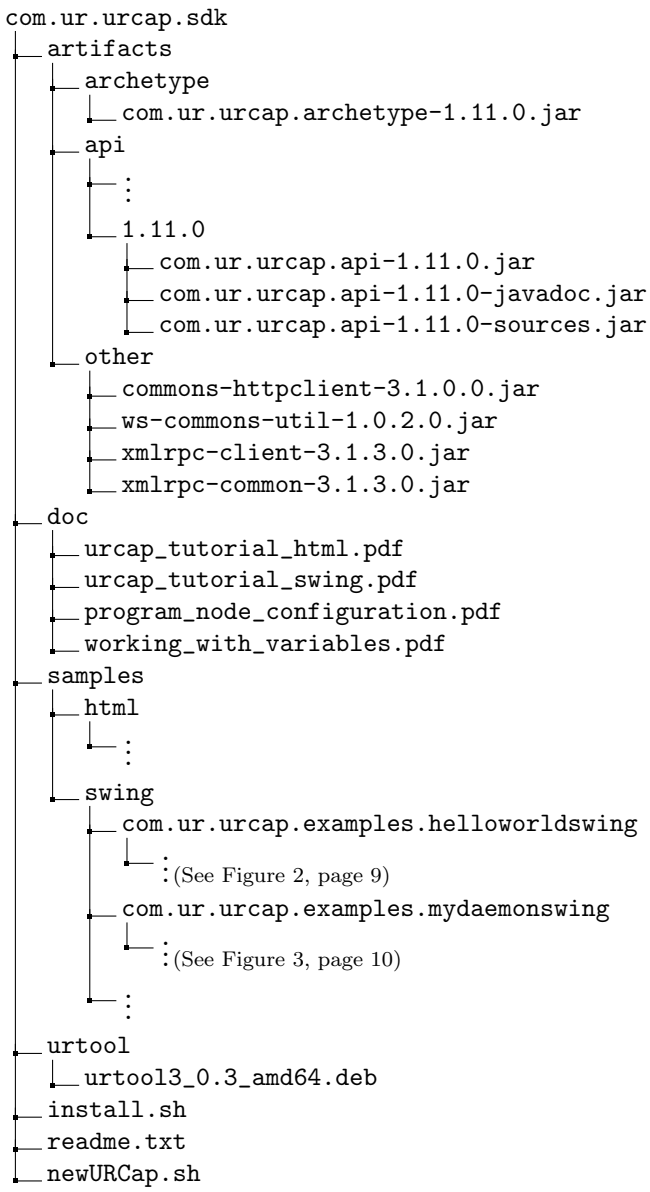


Figure 1: File structure of the URCaps SDK

A description of the directories and files contained in this file is given below:

`/artifacts/`: This directory holds all released versions of the URCap API in separate folders, the Maven archetype and other necessary files. Each folder named with a version number holds the Java packages, (e.g. `com.ur.urcap.api-1.11.0*.jar` files), that contains Java interfaces, Javadoc and sources of the URCap API that are necessary to implement the Java portion of a URCap. The Maven archetype folder holds the `com.ur.urcap.archetype-1.11.0.jar` file, that can be used to create a new empty template URCap project.

`/doc/`: The directory contains this tutorial (both in a HTML-based version and a Swing-based version) as well as a document describing how to configure child program nodes in a subtree and a document explaining how to work with variables. Included is also a document with a guide on how to convert an existing URCap with HTML-based user interface to a Swing-based one.

`/samples/`: A folder containing example projects demonstrating different features of the software framework. A description of the examples is found in section 10.

`/urtool/`: Contains the urtool3 cross-compiler toolchain that should be used when building C/C++ daemon executables for the CB3.0/3.1 and CB5.0 control boxes.

`install.sh`: A script which should be run as a first step to install the URCap SDK and urtool3 cross-compiler toolchain (see section 4). This will install all released versions of the URCap API and the Maven archetype in your local Maven repository as well as the cross-compiler toolchain in `/opt/urtool-3.0` (should you choose so).

`newURCap.sh`: A script which can be used to create a new empty Maven-based template URCap project in the current working directory (see section 11).

`readme.txt`: A *readme* file describing the content of the SDK.

4 Building and deploying URCaps

4.1 Building

To get started unzip the SDK zip file to a suitable location and run the install script inside the target location:

```
$ ./install.sh
```

This installs the SDK on your machine.

Next, enter the `samples/swing/com.ur.urcap.examples.helloworldswing` directory and compile the example by:

```
$ cd samples/swing/com.ur.urcap.examples.helloworldswing
$ mvn install
```

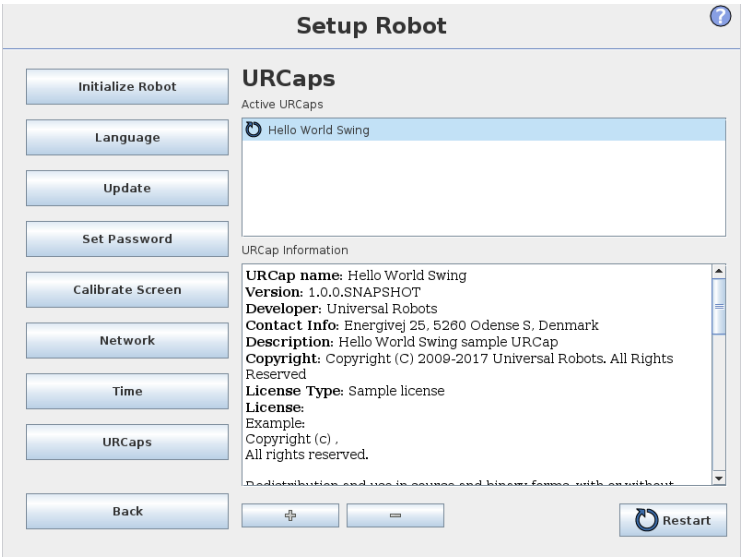
A new URCap with file name `target/helloworldswing-1.0-SNAPSHOT.urcap` has been born!

A similar procedure should be followed to compile the other URCap examples.

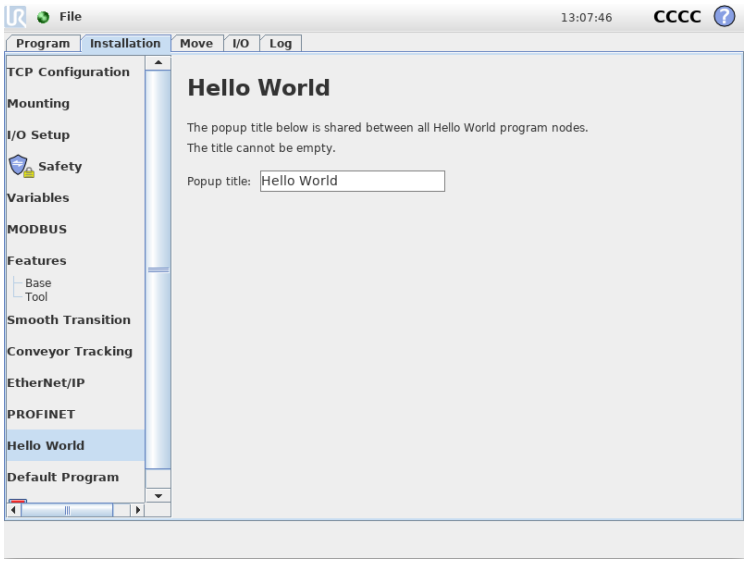
4.2 Manual Deployment

The URCap can be added to PolyScope with these steps:

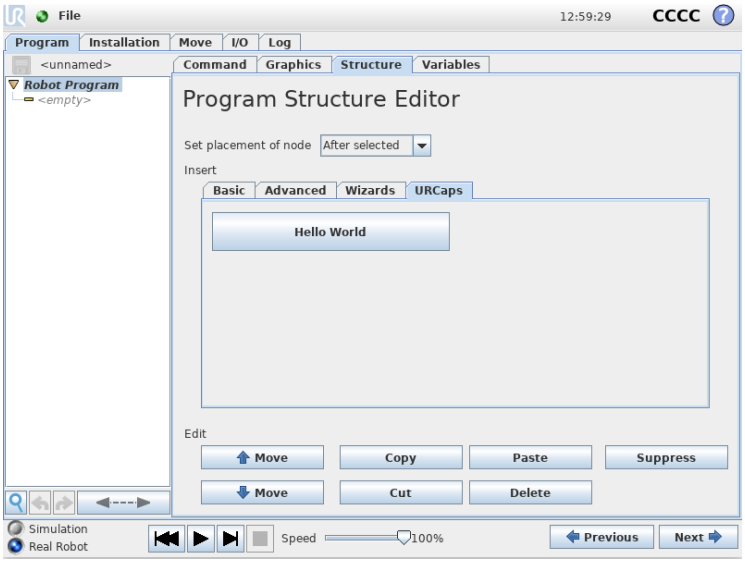
1. Copy the `helloworldswing-1.0-SNAPSHOT.urcap` file from above to your `programs` directory used by PolyScope or to a USB stick and insert it into a robot.
2. Tap the Setup Robot button from the PolyScope Robot User Interface Screen (the Welcome screen).
3. Tap the URCaps Setup button from the Setup Robot Screen.
4. Tap the + button.
5. Select a `.urcap` file, e.g. `helloworldswing-1.0-SNAPSHOT.urcap` and tap the Open button.
6. Restart PolyScope using the button in the bottom of the screen:



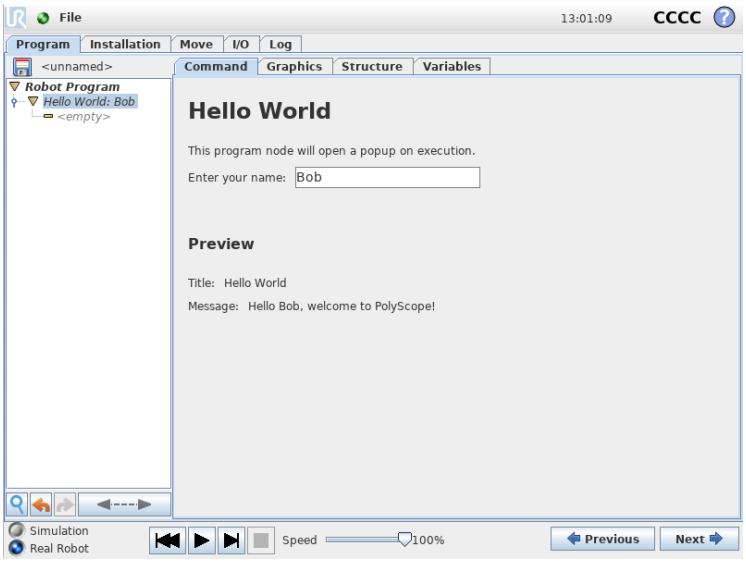
When the Hello World Swing URCap is deployed, the following installation screen is accessible from the Installation tab:



Furthermore the Hello World program node is visible within the Structure tab after selecting the URCaps tab:



The screen for the program node looks as follows:



When the program displayed above runs, a pop-up is shown with the title "Hello World" (configured in the installation screen) and message "Hello Bob, welcome to PolyScope!" (using the name defined in the program node).

5 Structure of a URCap Project

A URCap is a Java Archive (`.jar`) file with the `.urcap` file extension. The Java file may contain a number of new installation nodes, program nodes, and daemon executables.

Figure 2 shows the structure of the Hello World Swing URCap project. This project consists of the following parts:

1. A Java *view* part consisting of two screens with the layout specified in the two files `HelloWorldProgramNodeView.java` and `HelloWorldInstallationNodeView.java`.

2. A Java *implementation* for the screens above, namely:
   (a) `HelloWorldInstallationNodeService.java` and `HelloWorldInstallationNodeContribution.java`
   (b) `HelloWorldProgramNodeService.java` and `HelloWorldProgramNodeContribution.java`
3. A *license* `META-INF/LICENSE` with the license information that is shown to the user when the URCap is installed.
4. Maven configuration files `pom.xml` and `assembly.xml` for building the project.

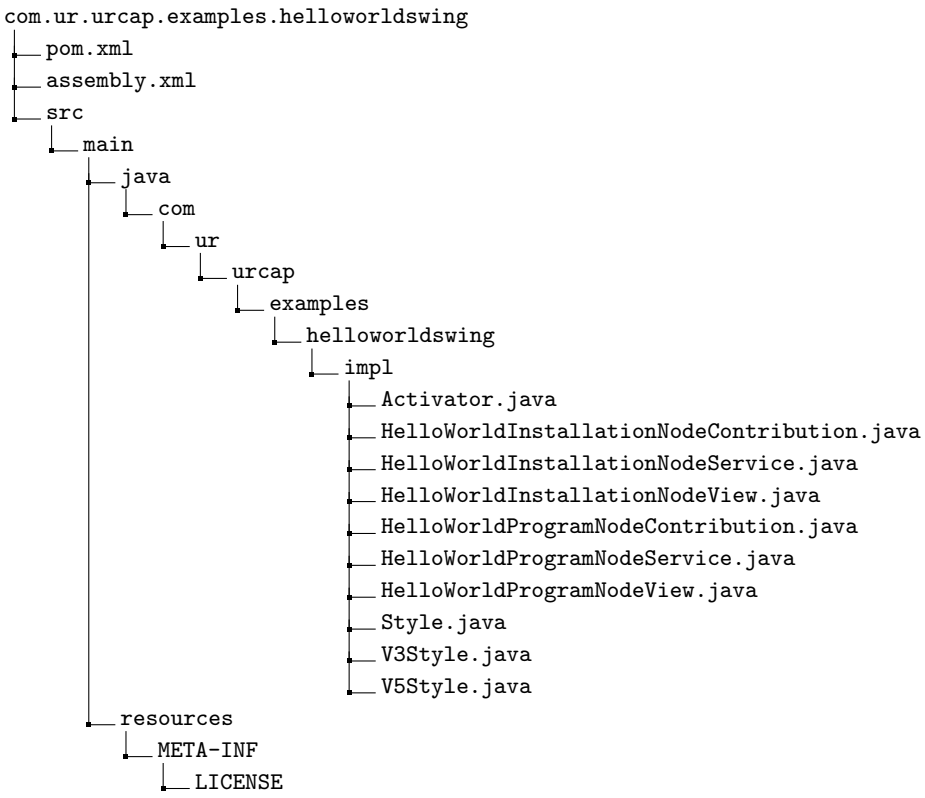


Figure 2: Structure of the Hello World Swing URCap project

The My Daemon Swing URCap is an extended version of the Hello World Swing URCap, that exemplifies the integration of an external daemon process. Figure 3 shows the structure of the My Daemon Swing URCap project. Compared to the Hello World Swing project it additionally offers the following parts:

1. A Python 2.5 *daemon* executable in the file `hello-world.py`.
2. C++ *daemon* sources in directory `daemon`.
3. A Java *implementation* `MyDaemonDaemonService.java` that defines and installs a daemon and makes it possible to control the daemon.

Figure 3: Structure of the My Daemon Swing URCap project

The Python and C++ daemons are alternatives that provide the same functionality.

The services:

- `HelloWorldInstallationNodeService.java`
- `HelloWorldProgramNodeService.java`

are registered in `Activator.java` and thereby a new installation node and program node are offered to PolyScope. The My Daemon Swing additionally registers its `MyDaemonDaemonService.java` service to make the daemon executable available to PolyScope.

The file `pom.xml` contains a section with a set of properties for the URCap with meta-data specifying the vendor, contact address, copyright, description, and short license information which will be displayed to the user when the URCap is installed in PolyScope. See Figure 4 for the Hello World Swing version of these properties.

```
1 <!--******************** BEGINNING OF URCAP META DATA ********************-->
2 <urcap.symbolicname>com.ur.urcap.examples.helloworldswing</urcap.symbolicname>
3 <urcap.vendor>Universal Robots</urcap.vendor>
4 <urcap.contactAddress>Energivej 25, 5260 Odense S, Denmark</
      urcap.contactAddress>
5 <urcap.copyright>Copyright (C) 2009-2020 Universal Robots. All Rights Reserved
      </urcap.copyright>
6 <urcap.description>Hello World Swing sample URCap</urcap.description>
7 <urcap.licenseType>Sample license</urcap.licenseType>
8 <!--************************ END OF URCAP META DATA ************************-->
```

Figure 4: Section with meta-data properties inside the `pom.xml` file for the Hello World Swing URCap

6 Deployment with Maven

In order to ease development, a URCap can be deployed using Maven.

Deployment to a robot with Maven Given the IP address of the robot, e.g. `10.2.128.64`, go to your URCap project folder and type:

```
1   $ cd samples/swing/com.ur.urcap.examples.helloworldswing
2   $ mvn install -Premote -Durcap.install.host=10.2.128.64
```

and the URCap is deployed and installed on the robot. During this process PolyScope will be restarted.

You can also specify the IP address of the robot via the property `urcap.install.host` inside the `pom.xml` file. Then you can deploy by typing:

```
1   $ cd samples/swing/com.ur.urcap.examples.helloworldswing
2   $ mvn install -Premote
```

Deployment to URSim If you are running Linux then URSim can be installed locally. Otherwise it needs to run in a virtual machine (VM). It is possible to deploy to both environments with Maven. As shown above parameters can be supplied either directly on the command line or in the `pom.xml` file.

- To deploy to a *locally running URSim* specify the path to the extracted URSim with the property `ursim.home`.
- To deploy to a *URSim running in a VM* specify the IP address of the VM using the property `ursimvm.install.host`.

Once the properties are configured you can deploy to a local URSim by using the `ursim` profile:

```
1   $ cd samples/swing/com.ur.urcap.examples.helloworldswing
2   $ mvn install -P ursim
```

or the URSim running in a VM using the `ursimvm` profile:

```
$ cd samples/swing/com.ur.urcap.examples.helloworldswing
$ mvn install -P ursimvm
```

Note, if you are using VirtualBox to run the VM you should make sure that the network of the VM is operating in bridged mode.

7 Contribution of an Installation Node

A URCap can contribute installation nodes. An installation node will support a customized installation node screen and customized functionality.

7.1 UI of the Installation Node View

The layout of a customized installation node screen is defined by a Java class implementing the `SwingInstallationNodeView` interface where the Swing GUI framework is used to create the user interface (UI). The implementation must specify the associated installation node contribution implementing the installation node's functionality (described in section 7.3) as a type variable.

Listing 1: The view (UI) of the customized Hello World installation screen

```
package com.ur.urcap.examples.helloworldswing.impl;

import com.ur.urcap.api.contribution.installation.swing.
    SwingInstallationNodeView;
import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardTextInput;

import javax.swing.BorderFactory;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.JTextPane;
import javax.swing.text.SimpleAttributeSet;
import javax.swing.text.StyleConstants;
import java.awt.Component;
import java.awt.Dimension;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

public class HelloWorldInstallationNodeView implements
    SwingInstallationNodeView<HelloWorldInstallationNodeContribution> {

  private final Style style;
  private JTextField jTextField;

  public HelloWorldInstallationNodeView(Style style) {
    this.style = style;
  }

  @Override
  public void buildUI(JPanel jPanel, final
      HelloWorldInstallationNodeContribution installationNode) {
    jPanel.setLayout(new BoxLayout(jPanel, BoxLayout.Y_AXIS));

    jPanel.add(createInfo());
    jPanel.add(createVerticalSpacing());
    jPanel.add(createInput(installationNode));
  }

  private Box createInfo() {
    Box infoBox = Box.createVerticalBox();
    infoBox.setAlignmentX(Component.LEFT_ALIGNMENT);
```

```
41      JTextPane pane = new JTextPane();
42      pane.setBorder(BorderFactory.createEmptyBorder());
43      SimpleAttributeSet attributeSet = new SimpleAttributeSet();
44      StyleConstants.setLineSpacing(attributeSet, 0.5f);
45      StyleConstants.setLeftIndent(attributeSet, 0f);
46      pane.setParagraphAttributes(attributeSet, false);
47      pane.setText("The␣popup␣title␣below␣is␣shared␣between␣all␣Hello␣World␣
            program␣nodes.\nThe␣title␣cannot␣be␣empty.");
48      pane.setEditable(false);
49      pane.setMaximumSize(pane.getPreferredSize());
50      pane.setBackground(infoBox.getBackground());
51      infoBox.add(pane);
52      return infoBox;
53    }
54
55    private Box createInput(final HelloWorldInstallationNodeContribution
          installationNode) {
56      Box inputBox = Box.createHorizontalBox();
57      inputBox.setAlignmentX(Component.LEFT_ALIGNMENT);
58
59      inputBox.add(new JLabel("Popup␣title:"));
60      inputBox.add(createHorizontalSpacing());
61
62      jTextField = new JTextField();
63      jTextField.setFocusable(false);
64      jTextField.setPreferredSize(style.getInputfieldSize());
65      jTextField.setMaximumSize(jTextField.getPreferredSize());
66      jTextField.addMouseListener(new MouseAdapter() {
67        @Override
68        public void mousePressed(MouseEvent e) {
69          KeyboardTextInput keyboardInput = installationNode.
                getInputForTextField();
70          keyboardInput.show(jTextField, installationNode.
                getCallbackForTextField());
71        }
72      });
73      inputBox.add(jTextField);
74
75      return inputBox;
76    }
77
78    private Component createHorizontalSpacing() {
79      return Box.createRigidArea(new Dimension(style.getHorizontalSpacing(), 0))
            ;
80    }
81
82    private Component createVerticalSpacing() {
83      return Box.createRigidArea(new Dimension(0, style.getVerticalSpacing()));
84    }
85
86    public void setPopupText(String t) {
87      jTextField.setText(t);
88    }
89  }
```

Listing 1 shows the content of the `HelloWorldInstallationNodeView.java` file which defines the layout of the screen used for the Hello World installation node. The class uses various Swing GUI components to construct the user interface. These components are added to the `JPanel` provided as argument in the `buildUI(JPanel, InstallationNodeContribution)` method.

The panel has a fixed size that cannot be changed. Margins are already added by PolyScope, so the entire area of the panel can be used for UI components. The title will also be set automatically to the value returned by `SwingInstallationNodeService.getTitle(Locale)` (described in section 7.2). It will have a Swing UI manager already set, meaning that components without

additional styling will look as native PolyScope ones including the correct font types. In order to resemble PolyScope the components should therefore only use limited styling, such as font sizes or input field sizes (if the applied ones are not suitable). Calling methods not supported by PolyScope will result in an exception being thrown.

The corresponding installation node contribution is passed as the second argument to the method `buildUI(JPanel, InstallationNodeContribution)` to enable the view and the contribution to communicate with each other in order to pass values and react to events.

This structure creates a model-view separation where the view is created in the aforementioned class and the model is handled in the contribution. The corresponding Java code is presented in the following two sections.

7.2 Making the customized Installation Node available to PolyScope

In order to make the layout specified in the view class and the customized installation nodes available to PolyScope, a Java class that implements the interface `SwingInstallationNodeService` must be defined. Listing 2 shows the Java code that makes the Hello World installation node available to PolyScope.

Listing 2: Hello World Installation node service

```
1  package com.ur.urcap.examples.helloworldswing.impl;
2
3  import com.ur.urcap.api.contribution.ViewAPIProvider;
4  import com.ur.urcap.api.contribution.installation.ContributionConfiguration;
5  import com.ur.urcap.api.contribution.installation.CreationContext;
6  import com.ur.urcap.api.contribution.installation.InstallationAPIProvider;
7  import com.ur.urcap.api.contribution.installation.swing.
       SwingInstallationNodeService;
8  import com.ur.urcap.api.domain.SystemAPI;
9  import com.ur.urcap.api.domain.data.DataModel;
10
11 import java.util.Locale;
12
13 public class HelloWorldInstallationNodeService implements
       SwingInstallationNodeService<HelloWorldInstallationNodeContribution,
       HelloWorldInstallationNodeView> {
14
15   @Override
16   public void configureContribution(ContributionConfiguration configuration) {
17   }
18
19   @Override
20   public String getTitle(Locale locale) {
21     return "Hello World";
22   }
23
24   @Override
25   public HelloWorldInstallationNodeView createView(ViewAPIProvider apiProvider
         ) {
26     SystemAPI systemAPI = apiProvider.getSystemAPI();
27     Style style = systemAPI.getSoftwareVersion().getMajorVersion() >= 5 ? new
           V5Style() : new V3Style();
28     return new HelloWorldInstallationNodeView(style);
29   }
30
31   @Override
32   public HelloWorldInstallationNodeContribution createInstallationNode(
         InstallationAPIProvider apiProvider, HelloWorldInstallationNodeView view
         , DataModel model, CreationContext context) {
```

```
33      return new HelloWorldInstallationNodeContribution(apiProvider, model, view
            );
34    }
35  }
```

The `SwingInstallationNodeService` interface requires two type variables for the specific implementations of the `InstallationNodeContribution` and `SwingInstallationNodeView` interface, respectively, as well as the following methods to be defined:

- `getTitle(Locale)` returns the title for the node, to be shown on the left side of the Installation tab to access the customized installation screen. For simplicity, the title is specified simply as `"Hello World"`. In a more realistic example, the return value of the `getTitle(Locale)` method would be translated into the language specified by standard Java localization, based on the provided `Locale` argument. The title is also used as the fixed header for the installation node screen in PolyScope. The method is only called once.
- `createView(ViewAPIProvider)` returns an instance of the view (described in section 7.1). Use the `ViewAPIProvider` to access information about the software version, language settings (can be used to provide a translated UI), etc.
- `createInstallationNode(InstallationAPIProvider, SwingInstallationNodeView, DataModel, CreationContext)` is called by PolyScope when it needs to create an instance of the installation node. The arguments are:
  - `InstallationAPIProvider`: provides access to various PolyScope domain APIs relevant for an installation node
  - `SwingInstallationNodeView`: the view instance created by the `createView(ViewAPIProvider)` described above
  - `DataModel`: the model for the installation node with automatic persistence support
  - `CreationContext`: the context in which the `createInstallationNode(...)` is called

  The constructor used in the implementation of the method `createInstallationNode(...)` is discussed in section 7.3. All modifications to the supplied data model from the installation node constructor are ignored when existing installation is loaded. This means that ideally the installation node constructor should not set anything in the data model.
- `configureContribution(ContributionConfiguration)` is called once after the service has been registered. Use the argument supplied to configure the contribution if its default values are not appropriate (see default values in the Javadoc). If the default values are appropriate, leave this method empty.

7.3 Functionality of the Installation Node

The functionality behind a customized installation node must be defined in a Java class that implements the `InstallationNodeContribution` interface. Listing 3 shows the Java code that defines the functionality of the Hello World installation screen. An instance of this class is returned by the `createInstallationNode(...)` method in the `HelloWorldInstallationNodeService` class described in previous section.

In essence, the `InstallationNodeContribution` interface requires the following to be defined:

1. What happens when the user enters and exits the customized installation screen.
2. Script code that should be added to the preamble of any program when run with this URCap installed.

In addition, the class contains code that links to the view (mentioned in section 7.1), gives access to a data model with automatic persistence and UR-Script generation associated with the node.

Listing 3: Java class defining functionality for the Hello World installation node

```
1  package com.ur.urcap.examples.helloworldswing.impl;
2
3  import com.ur.urcap.api.contribution.InstallationNodeContribution;
4  import com.ur.urcap.api.contribution.installation.InstallationAPIProvider;
5  import com.ur.urcap.api.domain.data.DataModel;
6  import com.ur.urcap.api.domain.script.ScriptWriter;
7  import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardInputCallback;
8  import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardInputFactory;
9  import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardTextInput;
10
11 public class HelloWorldInstallationNodeContribution implements
       InstallationNodeContribution {
12
13   private static final String POPUPTITLE_KEY = "popuptitle";
14   private static final String DEFAULT_VALUE = "Hello World";
15   private final HelloWorldInstallationNodeView view;
16   private final KeyboardInputFactory keyboardFactory;
17
18   private DataModel model;
19
20   public HelloWorldInstallationNodeContribution(InstallationAPIProvider
         apiProvider, DataModel model, HelloWorldInstallationNodeView view) {
21     this.keyboardFactory = apiProvider.getUserInterfaceAPI().
           getUserInteraction().getKeyboardInputFactory();
22     this.model = model;
23     this.view = view;
24   }
25
26   @Override
27   public void openView() {
28     view.setPopupText(getPopupTitle());
29   }
30
31   @Override
32   public void closeView() {
33
34   }
35
36   public boolean isDefined() {
37     return !getPopupTitle().isEmpty();
38   }
39
40   @Override
41   public void generateScript(ScriptWriter writer) {
42     // Store the popup title in a global variable so it is globally available
           to all Hello World program nodes.
43     writer.assign("hello_world_swing_popup_title", "\"" + getPopupTitle() + "
           \"");
44   }
45
46   public String getPopupTitle() {
47     return model.get(POPUPTITLE_KEY, DEFAULT_VALUE);
48   }
49
50   public void setPopupTitle(String message) {
51     if ("".equals(message)) {
52       resetToDefaultValue();
53     } else {
54       model.set(POPUPTITLE_KEY, message);
55     }
56   }
57
```

```
58    private void resetToDefaultValue() {
59      view.setPopupText(DEFAULT_VALUE);
60      model.set(POPUPTITLE_KEY, DEFAULT_VALUE);
61    }
62
63    public KeyboardTextInput getInputForTextField() {
64      KeyboardTextInput keyboardInput = keyboardFactory.
            createStringKeyboardInput();
65      keyboardInput.setInitialValue(getPopupTitle());
66      return keyboardInput;
67    }
68
69    public KeyboardInputCallback<String> getCallbackForTextField() {
70      return new KeyboardInputCallback<String>() {
71        @Override
72        public void onOk(String value) {
73          setPopupTitle(value);
74          view.setPopupText(value);
75        }
76      };
77    }
78  }
```

The data model which was mentioned in section 7.2 is passed into the constructor through a `DataModel` object. All data that needs to be saved and loaded along with a robot installation must be stored in and retrieved from this model object.

When the user interacts with the text input field defined in the view, the listener defined there will request a keyboard from the contribution and when the user accepts, delegate the call to `getCallbackForTextField()`. The code within that method takes care of storing the contents of the text input widget in the data model under the key `POPUPTITLE_KEY` whenever the user accepts what is typed using the keyboard. By saving and loading the robot installation you will notice that values are stored and read again from and back to the view.

The `openView()` method is called whenever the user enters the screen. It sets the contents of the text input field defined in the view to the value stored in the data model. The `closeView()` method is called when the user leaves the screen.

Finally, the preamble of each program run with this URCap installed will contain an assignment in its preamble, as specified in the implementation of the `generateScript(ScriptWriter)` method. In the assignment, the script variable named `"hello_world_swing_popup_title"` is assigned to a string that contains the popup title stored within the data model object.

7.4 Life Cycle of Contributions and Views

Each time a new installation is created or a different installation is loaded, the `createView(...)` method in the `SwingInstallationNodeService` interface (see section 7.2) is also called and a new view instance should be returned. The `createInstallationNode(...)` method in the interface `SwingInstallationNodeService` is also called to pass in the new `DataModel` instance.

This means that Java garbage collection has a chance to clean up previous instances in case any listeners have been forgotten or other potential memory leaks have been created. This also means that no references to the view instance or contribution instance should be kept outside these classes. Respecting this will protect PolyScope from running out of memory.

8 Contribution of a Program Node

A URCap can contribute program nodes. A node is supplied by a customized view part and a part with the customized functionality.

8.1 UI of the Program Node View

The layout for customized program node screens is defined similarly as the layout of customized installation node screens (see section 7.1) by implementing the `SwingProgramNodeView` interface. The implementation must specify the associated program node contribution implementing the functionality of the program nodes (described in section 8.4) as a type variable. The panel provided to the `buildUI(JPanel, ContributionProvider<ProgramNodeContribution>)` method has the same restrictions and properties as described in section 7.1.

Listing 4 shows the definition of the layout of a simple program node. It contains a single input text field where the user can type a name and two labels that provide a preview of the popup that will be displayed at runtime. A label defined in the view will be used to display the title set in the installation. The name that is entered by the end user in the Hello World program node will be used to construct a customized popup message. This message will also be shown in the preview label with name `"previewMessage"`.

In order to communicate with the corresponding program node contribution implementing the functionality of the program node itself, a provider is passed as argument to the method `buildUI(JPanel, ContributionProvider<ProgramNodeContribution>)`. Calling the `get()` method on the `ContributionProvider` object will return the currently selected program node. The provider has a type variable which corresponds to the associated contribution.

The linking of the view and the program node contributions is presented in the following section.

Listing 4: The view (UI) of the customized Hello World program screen

```
1  package com.ur.urcap.examples.helloworldswing.impl;
2
3  import com.ur.urcap.api.contribution.ContributionProvider;
4  import com.ur.urcap.api.contribution.program.swing.SwingProgramNodeView;
5  import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardTextInput;
6
7  import javax.swing.Box;
8  import javax.swing.BoxLayout;
9  import javax.swing.JLabel;
10 import javax.swing.JPanel;
11 import javax.swing.JTextField;
12 import java.awt.Component;
13 import java.awt.Dimension;
14 import java.awt.Font;
15 import java.awt.event.MouseAdapter;
16 import java.awt.event.MouseEvent;
17
18 public class HelloWorldProgramNodeView implements SwingProgramNodeView<
       HelloWorldProgramNodeContribution>{
19
20   private final Style style;
21   private JTextField jTextField;
22   private JLabel previewTitle;
23   private JLabel previewMessage;
24
25   public HelloWorldProgramNodeView(Style style) {
26     this.style = style;
27   }
28
```

```
29    @Override
30    public void buildUI(JPanel jPanel, final ContributionProvider<
          HelloWorldProgramNodeContribution> provider) {
31      jPanel.setLayout(new BoxLayout(jPanel, BoxLayout.Y_AXIS));
32
33      jPanel.add(createInfo());
34      jPanel.add(createVerticalSpacing(style.getVerticalSpacing()));
35      jPanel.add(createInput(provider));
36      jPanel.add(createVerticalSpacing(style.getExtraLargeVerticalSpacing()));
37      jPanel.add(createPreview());
38    }
39
40    private Box createInfo() {
41      Box infoBox = Box.createHorizontalBox();
42      infoBox.setAlignmentX(Component.LEFT_ALIGNMENT);
43      infoBox.add(new JLabel("This␣program␣node␣will␣open␣a␣popup␣on␣execution."
            ));
44      return infoBox;
45    }
46
47    private Box createInput(final ContributionProvider<
          HelloWorldProgramNodeContribution> provider) {
48      Box inputBox = Box.createHorizontalBox();
49      inputBox.setAlignmentX(Component.LEFT_ALIGNMENT);
50      inputBox.add(new JLabel("Enter␣your␣name:"));
51      inputBox.add(createHorizontalSpacing());
52
53      jTextField = new JTextField();
54      jTextField.setFocusable(false);
55      jTextField.setPreferredSize(style.getInputfieldSize());
56      jTextField.setMaximumSize(jTextField.getPreferredSize());
57      jTextField.addMouseListener(new MouseAdapter() {
58        @Override
59        public void mousePressed(MouseEvent e) {
60          KeyboardTextInput keyboardInput = provider.get().
                getKeyboardForTextField();
61          keyboardInput.show(jTextField, provider.get().getCallbackForTextField
                ());
62        }
63      });
64
65      inputBox.add(jTextField);
66      return inputBox;
67    }
68
69    private Box createPreview() {
70      Box previewBox = Box.createVerticalBox();
71      JLabel preview = new JLabel("Preview");
72      preview.setFont(preview.getFont().deriveFont(Font.BOLD, style.
            getSmallHeaderFontSize()));
73
74      Box titleBox = Box.createHorizontalBox();
75      titleBox.setAlignmentX(Component.LEFT_ALIGNMENT);
76      titleBox.add(new JLabel("Title:"));
77      titleBox.add(createHorizontalSpacing());
78      previewTitle = new JLabel("my␣title");
79      titleBox.add(previewTitle);
80
81      Box messageBox = Box.createHorizontalBox();
82      messageBox.setAlignmentX(Component.LEFT_ALIGNMENT);
83      messageBox.add(new JLabel("Message:"));
84      messageBox.add(createHorizontalSpacing());
85      previewMessage = new JLabel("my␣message");
86      messageBox.add(previewMessage);
87
88      previewBox.add(preview);
89      previewBox.add(createVerticalSpacing(style.getLargeVerticalSpacing()));
```

```
90        previewBox.add(titleBox);
91        previewBox.add(createVerticalSpacing(style.getVerticalSpacing()));
92        previewBox.add(messageBox);
93
94        return previewBox;
95      }
96
97      private Component createVerticalSpacing(int height) {
98        return Box.createRigidArea(new Dimension(0, height));
99      }
100
101     private Component createHorizontalSpacing() {
102       return Box.createRigidArea(new Dimension(style.getHorizontalSpacing(), 0))
              ;
103     }
104
105     public void setPopupText(String popupText) {
106       jTextField.setText(popupText);
107     }
108
109     public void setMessagePreview(String message) {
110       previewMessage.setText(message);
111     }
112
113     public void setTitlePreview(String title) {
114       previewTitle.setText(title);
115     }
116   }
```

8.2 Linking View and Contribution

The view (implementing the `SwingProgramNodeView` interface) and the program node contribution (implementing the `ProgramNodeContribution` interface) 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 supplied provider must be used. The `get()` method on the `ContributionProvider` will automatically return the `ProgramNodeContribution` instance representing the currently selected program node and in turn its associated data model.

The contribution object on the other hand was instantiated with the one and only view instance and can call methods on this instance without any further ado.

8.3 Making the customized Program Nodes available to PolyScope

To make the Hello World program node available to PolyScope, a Java class that implements the `SwingProgramNodeService` interface is required. Listing 5 shows the Java code that makes the Hello World program node available to PolyScope.

The `getId()` method returns the unique identifier for this type of program node. The identifier will be used when storing programs that contain these program nodes. This method is called once. Do not change the return value of this method in released URCaps, since it will break backwards compatibility for existing programs. URCap program nodes in such existing programs will not be loaded properly and the program can not run anymore.

Its `getTitle(Locale)` method supplies the text for the button in the Structure Tab that corresponds to this type of program node. It is also used as the title on the Command tab screen

for such nodes. Use the provided `Locale` if translated titles should be supported. This method is called once.

Use the argument supplied in the method `configureContribution(ContributionConfiguration)` to configure the program node contribution. This method is called once after the service has been registered. Use the argument supplied to configure the contribution if its default values are not appropriate (see default values in the Javadoc). If the default values are appropriate, leave this method empty. The following properties can be configured:

- Calling `setDeprecated()` with `true` makes it impossible to create new program nodes of this type, but still support loading program nodes of this type in existing programs.
- Calling `setChildrenAllowed()` with `true`, signals that it is possible for the program node to contain other (child) program nodes.
- Calling `setUserInsertable()` with `false` makes the program node programmatically insertable only (i.e. the end user cannot insert it).
- Calling `getProgramDebuggingSupport()` returns `ProgramDebuggingSupport` object that can be used to customize debugging capabilities of the Program Node and child nodes in subtree. Debugging features are available on e-Series platform only. Refer to User Manual for detailed description.
  - `setAllowBreakpointOnNode()` determines if the end user is allowed to set a breakpoint on or single step this program node.
  - `setAllowBreakpointOnChildNodesInSubtree()` determines if the end user is allowed to set breakpoint on or single step any child in this program node subtree (only if the child node itself allows breakpoints).
  - `setAllowStartFromNode()` determines if the end user can start the program directly from this program node.
  - `setAllowStartFromChildNodesInSubtree()` determines if the end user can start the program directly from the selected child in this program node subtree (only if the child node itself allows it).

Finally, `createNode(ProgramAPIProvider, SwingProgramNodeView, DataModel, CreationContext)` creates program nodes. The arguments are:

- `ProgramAPIProvider`: provides access to various APIs relevant for a program node.
- `SwingProgramNodeView`: this is the view instance (described in section 8.1) created by the `createView(ViewAPIProvider)` method
- `DataModel`: this gives the user a data model with automatic persistence
- `CreationContext`: the context in which this program node is created

The `createNode(...)` method creates a new Java object for each node of this type occurring in the program tree. The returned object is used when interacting with the view on the customized program node screen for the particular node selected in the program tree. It must use the supplied data model object to retrieve and store data that should be saved and loaded within the robot program along with the corresponding node occurrence. Please note that only data related to the current configuration of that particular program node instance should be stored in the data model, i.e. no global or shared state, state irrevelant to this node instance, etc. should be stored.

The constructor used in the implementation of the `createNode(...)` method is discussed in section 8.4. The `createNode(...)` method call during program load is discussed in section 8.6

Listing 5: Java class defining how Hello World program nodes are created

```
1  package com.ur.urcap.examples.helloworldswing.impl;
2
3  import com.ur.urcap.api.contribution.ViewAPIProvider;
4  import com.ur.urcap.api.contribution.program.ContributionConfiguration;
5  import com.ur.urcap.api.contribution.program.CreationContext;
6  import com.ur.urcap.api.contribution.program.ProgramAPIProvider;
7  import com.ur.urcap.api.contribution.program.swing.SwingProgramNodeService;
8  import com.ur.urcap.api.domain.SystemAPI;
9  import com.ur.urcap.api.domain.data.DataModel;
10
11 import java.util.Locale;
12
13 public class HelloWorldProgramNodeService implements SwingProgramNodeService<
       HelloWorldProgramNodeContribution, HelloWorldProgramNodeView> {
14
15   @Override
16   public String getId() {
17     return "HelloWorldSwingNode";
18   }
19
20   @Override
21   public void configureContribution(ContributionConfiguration configuration) {
22     configuration.setChildrenAllowed(true);
23   }
24
25   @Override
26   public String getTitle(Locale locale) {
27     String title = "Hello␣World";
28     if ("ru".equals(locale.getLanguage())) {
29       title = "            ␣      ";
30     } else if ("de".equals(locale.getLanguage())) {
31       title = "Hallo␣Welt";
32     }
33     return title;
34   }
35
36   @Override
37   public HelloWorldProgramNodeView createView(ViewAPIProvider apiProvider) {
38     SystemAPI systemAPI = apiProvider.getSystemAPI();
39     Style style = systemAPI.getSoftwareVersion().getMajorVersion() >= 5 ? new
           V5Style() : new V3Style();
40     return new HelloWorldProgramNodeView(style);
41   }
42
43   @Override
44   public HelloWorldProgramNodeContribution createNode(
45       ProgramAPIProvider apiProvider,
46       HelloWorldProgramNodeView view,
47       DataModel model,
48       CreationContext context) {
49     return new HelloWorldProgramNodeContribution(apiProvider, view, model);
50   }
51 }
```

8.4 Functionality of the Program Node

The functionality of the Hello World program node is implemented in the Java class shown in Listing 6. This class implements the `ProgramNodeContribution` interface and instances of this class are returned by the method `createNode(ProgramAPIProvider, SwingProgramNodeView, DataModel, CreationContext)` of the `HelloWorldProgramNodeService` class described in the previous section.

Listing 6: Java class defining functionality for the Hello World program node

```
1  package com.ur.urcap.examples.helloworldswing.impl;
2
3  import com.ur.urcap.api.contribution.ProgramNodeContribution;
4  import com.ur.urcap.api.contribution.program.ProgramAPIProvider;
5  import com.ur.urcap.api.domain.ProgramAPI;
6  import com.ur.urcap.api.domain.data.DataModel;
7  import com.ur.urcap.api.domain.script.ScriptWriter;
8  import com.ur.urcap.api.domain.undoredo.UndoRedoManager;
9  import com.ur.urcap.api.domain.undoredo.UndoableChanges;
10 import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardInputCallback;
11 import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardInputFactory;
12 import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardTextInput;
13
14 public class HelloWorldProgramNodeContribution implements
       ProgramNodeContribution {
15   private static final String NAME = "name";
16
17   private final ProgramAPI programAPI;
18   private final UndoRedoManager undoRedoManager;
19   private final KeyboardInputFactory keyboardFactory;
20
21   private final HelloWorldProgramNodeView view;
22   private final DataModel model;
23
24   public HelloWorldProgramNodeContribution(ProgramAPIProvider apiProvider,
         HelloWorldProgramNodeView view, DataModel model) {
25     this.programAPI = apiProvider.getProgramAPI();
26     this.undoRedoManager = apiProvider.getProgramAPI().getUndoRedoManager();
27     this.keyboardFactory = apiProvider.getUserInterfaceAPI().
           getUserInteraction().getKeyboardInputFactory();
28
29     this.view = view;
30     this.model = model;
31   }
32
33   @Override
34   public void openView() {
35     view.setPopupText(getName());
36     updatePopupMessageAndPreview();
37   }
38
39   @Override
40   public void closeView() {
41   }
42
43   @Override
44   public String getTitle() {
45     return "Hello␣World:␣" + (model.isSet(NAME) ? getName() : "");
46   }
47
48   @Override
49   public boolean isDefined() {
50     return getInstallation().isDefined() && !getName().isEmpty();
51   }
52
53   @Override
54   public void generateScript(ScriptWriter writer) {
55     // Directly generate this Program Node's popup message + access the popup
           title through a global variable
56     writer.appendLine("popup(\"" + generatePopupMessage() + "\",␣
           hello_world_swing_popup_title,␣False,␣False,␣blocking=True)");
57     writer.writeChildren();
58   }
59
60   public KeyboardTextInput getKeyboardForTextField() {
61     KeyboardTextInput keyboardInput = keyboardFactory.
           createStringKeyboardInput();
```

```
62      keyboardInput.setInitialValue(getName());
63      return keyboardInput;
64    }
65
66    public KeyboardInputCallback<String> getCallbackForTextField() {
67      return new KeyboardInputCallback<String>() {
68        @Override
69        public void onOk(String value) {
70          setPopupTitle(value);
71          view.setPopupText(value);
72        }
73      };
74    }
75
76    public void setPopupTitle(final String value) {
77      undoRedoManager.recordChanges(new UndoableChanges() {
78        @Override
79        public void executeChanges() {
80          if ("".equals(value)) {
81            model.remove(NAME);
82          } else {
83            model.set(NAME, value);
84          }
85        }
86      });
87
88      updatePopupMessageAndPreview();
89    }
90
91    private String generatePopupMessage() {
92      return model.isSet(NAME) ? "Hello␣" + getName() + ",␣welcome␣to␣PolyScope!
            " : "No␣name␣set";
93    }
94
95    private void updatePopupMessageAndPreview() {
96      view.setMessagePreview(generatePopupMessage());
97      view.setTitlePreview(getInstallation().isDefined() ? getInstallation().
            getPopupTitle() : "No␣title␣set");
98    }
99
100   private String getName() {
101     return model.get(NAME, "");
102   }
103
104   private HelloWorldInstallationNodeContribution getInstallation() {
105     return programAPI.getInstallationNode(
            HelloWorldInstallationNodeContribution.class);
106   }
107
108 }
```

The `openView()` and `closeView()` methods specify what happens when the user selects and unselects the underlying program node in the program tree.

The `getTitle()` method defines the text which is displayed in the program tree for the node. The text of the node in the program tree is updated when values are written to the `DataModel`.

The `isDefined()` method serves to identify whether the node is completely defined (green) or still undefined (yellow). Note that a node, which can contain other program nodes (see section 8.3), remains undefined as long as it has a child that is undefined. The `isDefined()` method is called when values are written to the `DataModel` to ensure that the program tree reflects the proper state of the program node.

Finally, `generateScript(ScriptWriter)` is called to add script code to the spot where the underlying node occurs in the robot program.

As the user interacts with the text input field, the constructed message is displayed on the screen using the view instance. Each Hello World node is defined (green) if both the Hello World installation node is defined and the name in the program node is non-empty. When executed, it shows a simple popup dialog with the title defined in the installation and the message constructed from the name.

The popup title is the value of the script variable `hello_world_swing_popup_title`. This variable is initialized by the script code contributed by the Hello World installation node. Thus, the script variable serves to pass data from the contributed installation node to the contributed program node. Another approach to pass information between these two objects is by directly requesting the installation object through the `ProgramAPI` interface. The Hello World program node utilizes this approach in its `updatePopupMessageAndPreview()` method.

8.5 Updating the data model

8.5.1 Restrictions

It is not allowed to modify the supplied data model in the implementation of any overridden method defined in the program node contribution interface (described in section 8.4), which PolyScope calls. The data model should rather be updated due to a user-initiated event, e.g. a button click.

Furthermore, if the URCap contains an installation node contribution, the data model of the installation contribution not be modified from the program node contribution. If such modifications occur in the call to `openView()` while a program is running, PolyScope will automatically stop the program every time the URCap program node is encountered in the program tree, because the installation was changed. It is, however, allowed to read from the data model in the installation node contribution.

8.5.2 Undo/redo Functionality

All user-initiated data 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 end user. Below is a small code snippet demonstrating this.

Listing 7: Code snippet for undo/redo

```
1 UndoRedoManager manager = apiProvider.getProgramAPI().getUndoRedoManager();
2 manager.recordChanges(new UndoableChanges() {
3   @Override
4   public void executeChanges() {
5     model.set(NAME, "my name");
6     insertChildNodes();
7   }
8 });
```

Only if the end user initiated the action (e.g. clicked a button) should the changes be recorded as an undoable action. Otherwise the end user will be able to undo something he did not do and be confused as to what he is undoing.

Failing to record changes inside an `UndoableChanges` will throw a `IllegalStateExeption` exception.

As mentioned in section 8.3, remember to only store data related to the current configuration of the particular program node instance in the model, i.e. no global or shared state, state irrevelant to this node instance, etc. should be stored there.

Undoable actions only apply to a program nodes. Changes to a data model in an installation node do not have undo/redo support and will not throw a `IllegalStateException` exception.

When a user clicks undo, the previous values are restored in the data model as well as the program tree and a call to the `openView()` method if the program node is currently selected, for an opportunity to display the new values.

This also means that no values should be cached in member variables, but always retrieved from the data model, as there is no guarantee that things have not changed. Also keep in mind, that the user might not select the Command tab of the URCap, so there is no guaranteed call to `openView()`. This can be the case when loading a program that has already been setup.

8.6 Loading Programs with Program Node Contributions

Program node contributions contain a data model and an interface to manipulate the sub-tree (introduced in URCap API version 1.1.0).

When a contributable program node is created in PolyScope by the user, the data model object given to the `createNode(ProgramAPIProvider, SwingProgramNodeView, DataModel, CreationContext)` method is empty and the provided `CreationContext` object will also reflect this.

When a program is loaded, the method `createNode(...)` is called for each persisted program node contribution to re-create the program tree. In contrast to newly creating the program node, the data model now contains the data from the persisted node and the `CreationContext` object will reflect this situation also. All modifications to the data model from the program node constructor are ignored. This means that ideally the program node constructor should not set anything in the data model.

For creating sub-trees a program model can be used. In some of the URCap examples in Chapter 10 it is demonstrated how a sub-tree can be generated programmatically. The program model provides the interface `TreeNode` to create and manipulate the sub-tree. When a contributable program node is created in PolyScope by the user, the tree node has no children. The program model can be requested through the `ProgramAPI` interface.

When a program is loaded each program node is deserialized on its own, this includes sub-trees previously created through the program model. Also now, the tree node requested through the `ProgramModel` is empty. The program node factory returned by `getProgramNodeFactory()` in the `ProgramModel` interface will return program nodes without any functionality. In particular, the method `createURCapProgramNode(Class<? extends URCapProgramNodeService>)` does not call the `createNode(...)` method in the specified service. Therefore, modifications are ignored during the `createNode(...)` call.

8.7 Life Cycle of Contributions and Views

Each time a new program is created or a different program is loaded, the `createView(...)` method in the `SwingProgramNodeService` interface (see section 8.3) is also called and a new view instance should be returned. The `createNode(...)` method in the `SwingProgramNodeService` interface is also

called to pass in the new `DataModel` object.

This means that Java garbage collection has a chance to clean up previous instances in case any listeners have been forgotten or other potential memory leaks have been created. This also means that no references to the view instance or contribution instance should be kept outside these classes. Respecting this will protect PolyScope from running out of memory.

9 Contribution of a Daemon

A daemon can be any executable script or binary file that runs on the control box. The My Daemon Swing URCap serves as the running example for explaining this functionality and is an extension of the Hello World Swing example. The My Daemon Swing example offers the same functionality from the user's point of view as the Hello World Swing example.

However, the My Daemon Swing URCap performs its tasks through an executable, which acts as a sort of driver or server. The executable is implemented as Python 2.5 script and C++ binary. The executables communicate with the Java front-end and URScript executor through XML encoded Remote Procedure Calls (XML-RPC). Figure 3, page 10, shows the structure of the My Daemon Swing URCap project.

9.1 Daemon Service

A URCap can contribute any number of daemon executables through implementation of the `DaemonService` interface (see Listing 8):

- The `init(DaemonContribution)` method will be called by PolyScope with a `DaemonContribution` object which gives the URCap developer the control to install, start, stop, and query the state of the daemon. An example of how to integrate start, stop, and query a daemon will be discussed in Section 9.2.
- The `installResource(URL url)` method in the `DaemonContribution` interface takes an argument that points to the source inside the URCap Jar file (`.urcap` file). This path may point to a single executable daemon or a directory that contains a daemon and additional files (e.g. dynamic linked libraries or configuration files).
- The implementation of `getExecutable()` provides PolyScope with the path to the executable that will be started.

The `/etc/service` directory contains links to the URCap daemon executables currently running. If a daemon executable has a link present but is in fact not running, the `ERROR` state will be returned upon querying the daemon's state. The links to daemon executables follow the lifetime of the encapsulating URCap and will be removed when the URCap is removed. The initial state for a daemon is `STOPPED`, however if it is desired, auto-start can be achieved by calling `start()` in the `init(DaemonContribution)` method right after the daemon has had its resources installed.
Log information with respect to the process handling of the daemon executable are saved together with the daemon executable (follow the symbolic link of the daemon executable in `/etc/service` to locate the log directory).

Note, that script daemons must include an interpreter directive at the first line to help select the right program for interpreting the script. For instance, Bash scripts use `"#!/bin/bash"` and Python scripts use `"#!/usr/bin/env␣python"`.

Listing 8: The My Daemon Service

```
1  package com.ur.urcap.examples.mydaemonswing.impl;
2
3  import com.ur.urcap.api.contribution.DaemonContribution;
4  import com.ur.urcap.api.contribution.DaemonService;
5
6  import java.net.MalformedURLException;
7  import java.net.URL;
8
9
10 public class MyDaemonDaemonService implements DaemonService {
11
12   private DaemonContribution daemonContribution;
13
14   public MyDaemonDaemonService() {
15   }
16
17   @Override
18   public void init(DaemonContribution daemonContribution) {
19     this.daemonContribution = daemonContribution;
20     try {
21       daemonContribution.installResource(new URL("file:com/ur/urcap/examples/
             mydaemonswing/impl/daemon/"));
22     } catch (MalformedURLException e) { }
23   }
24
25   @Override
26   public URL getExecutable() {
27     try {
28       // Two equivalent example daemons are available:
29       return new URL("file:com/ur/urcap/examples/mydaemonswing/impl/daemon/
             hello-world.py"); // Python executable
30 //      return new URL("file:com/ur/urcap/examples/mydaemonswing/impl/daemon/
       HelloWorld"); // C++ executable
31     } catch (MalformedURLException e) {
32       return null;
33     }
34   }
35
36   public DaemonContribution getDaemon() {
37     return daemonContribution;
38   }
39
40 }
```

9.2 Interaction with the Daemon

The My Daemon installation screen is shown in Figure 5 and the code can be found in Listing 16, page 50, in Appendix B.

Two buttons have been added to the installation screen to enable and disable the daemon. In this example the daemon is enabled by default when a new installation is created, and future changes to the desired run state will be stored in the data model.

The daemon runs in parallel with PolyScope and can in principle change its state independently. Therefore, the label below the buttons displays the current run status of the daemon. This label is updated with a 1 Hz frequency, utilizing the `java.util.Timer` class. Since the UI update is initiated from a different thread than the Java AWT thread, the timer task must utilize the `EventQueue.invokeLater` functionality. Note, the `Timer` is added when the My Daemon Installation screen is opened (see `openView()`) and removed when the user moves away from the screen (see

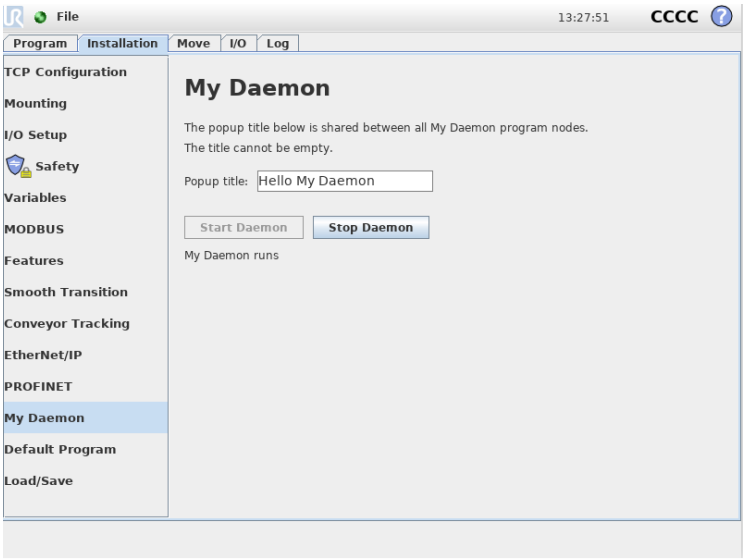


Figure 5: My Daemon installation screen

`closeView()`) to conserve computing resources.

Two options are available for Java and URScript to communicate with the daemon:

- TCP/IP sockets can be used to stream data.
- XML encoded Remote Procedure Calls (XML-RPC) can be used for configuration tasks (e.g. camera calibration) or service execution (e.g. locating the next object).

The advantage of XML-RPC over sockets is that no custom protocol or encoding needs to be implemented. The URScript XML-RPC implementation supports all URScript data types. Moreover, a RPC will only return when the function execution has been completed. This is desirable when the next program step relies on data retrieved from the daemon service. Plain sockets are on the other hand more efficient for data streaming, since there is no encoding overhead. Both methods can be complimentary applied and are available for Java ↔ daemon and URScript ↔ daemon communication.

Listing 9 shows a small URScript example for making a XML-RPC call to a XML-RPC server. The `hello-world.py` example daemon (see Listing 19, page 57) can be used as XML-RPC test server. Simply start the daemon in the My Daemon and run the URScript in a `Script` node.

Listing 9: URScript XML-RPC example

```
1 global mydaemon_swing = rpc_factory("xmlrpc", "http://127.0.0.1:40405/RPC2")
2 global mydaemon_message = mydaemon_swing.get_message("Bob")
3 popup(mydaemon_message, "My Title", False, False, blocking=True)
```

The intention of this URScript example is to retrieve a message from the daemon to display during runtime (similar to the My Daemon program node). The `rpc_factory` script function creates a connection to the XML-RPC server in the daemon. The new connection is stored in the global `my_daemon_swing` variable and serves as a handle. The next line then requests the XML-RPC server in the daemon to execute the `get_message(...)` function with the string argument

"Bob" and return the result. The return value of the RPC call is stored in the mydaemon_message variable for further processing in the popup(...) script function.

Note, making XML-RPC calls from URScript does not require any additional function stubs or pre-definitions of the remote function to be executed in URScript. Until the XML-RPC returns this URScript thread is automatically blocked (i.e. no sync nor Wait is needed). The standard XML-RPC protocol does not allow void return values and XML-RPC extensions enabling this are not always compatible.

The My Daemon Swing example also includes a Java XML-RPC client example, see the combination of the MyDaemonProgramNodeContribution and XMLRPCMyDaemonInterface classes (Listing 17, page 53 and listing 18, page 55 respectively). Note, the execution of the XML-RPC calls is not on the main Java AWT thread, but offloaded to a separate thread.

9.3 C/C++ Daemon Executables

The CB3.0/3.1 and CB5.0 control boxes all run a minimal Debian 32-bit Linux operating system. To guarantee binary compatibility all C/C++ executables should be compiled with the urtool3 cross-compiler under Linux. The urtool3 cross-compiler is included in the SDK installation.

To test if the urtool3 is properly installed type the following in a terminal:

```
1  echo $URTOOL_ROOT; i686-unknown-linux-gnu-g++ --version
```

The correct output is:

```
1  /opt/urtool-3.0
2  i686-unknown-linux-gnu-g++ (GCC) 4.1.2
3  Copyright (C) 2006 Free Software Foundation, Inc.
4  This is free software; see the source for copying conditions.  There is NO
5  warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

If the first line is not printed directly after installing the SDK, please reboot your PC for the environment variables to be updated.

The My Daemon Swing URCap comes with a fully functional C++ XML-RPC server example that is equivalent to the hello-world.py Python daemon. Simply switch the comments in the getExecutable() function in the MyDaemonDaemonService class (Listing 8, page 28), and recompile to use the C++ daemon implementation. The popup title should now be appended with "(C++)" instead of "(Python)" during execution of the URCap.

The C++ daemon directory structure is shown in Figure 6, page 31. For managing the software construction process of the C++ daemon a tool called Scons is used. The SConstruct file among other things contains the main configuration, the urtool3 cross-compiler, and libxmlrpc-c integration. The SConscript files are used to define the compilation targets, e.g. the Hello World binary.

For the example URCap, the daemon will be build as part of the URCap build process by maven. However, the daemon can also be compiled manually by typing the following in a terminal:

```
1  cd com.ur.urcap.examples.mydaemonswing/daemon
2  scons release=1
```

This will build a release version of the daemon. Using release=0 will build an executable with debugging symbols.

```
com.ur.urcap.examples.mydaemonswing
├── ...
├── daemon
│   ├── service
│   │   ├── AbyssServer.cpp
│   │   ├── AbyssServer.hpp
│   │   ├── SConscript
│   │   ├── XMLRPCMethods.cpp
│   │   └── XMLRPCMethods.hpp
│   ├── Data.cpp
│   ├── Data.hpp
│   ├── HelloWorld.cpp
│   ├── HelloWorld.cpp
│   ├── SConscript
│   └── SConstruct
└── ...
```

Figure 6: Structure of the C++ daemon of the My Daemon Swing project

The XML-RPC functionality in the C++ daemon relies on the open-source library libxmlrpc-c (`http://xmlrpc-c.sourceforge.net`). This library is by default available on the CB3.0/3.1 and CB5.0 control boxes. The `service` directory contains all relevant XML-RPC code. The AbyssServer is one of the XML-RPC server implementations supported by libxmlrpc-c. Please look in the C++ code for more programming hints and links to relevant documentation.

9.4 Tying the different Contributions together

The new My Daemon Swing URCap installation node, program node, and daemon executable are registered and offered to PolyScope through the code in Listing 10.

Three services are registered:

- `MyDaemonInstallationNodeService`
- `MyDaemonProgramNodeService`
- `MyDaemonDaemonService`

The `MyDaemonInstallationNodeService` class has visibility to an instance of the `MyDaemonDaemonService` class. This instance is passed in the constructor when a new installation node instance of the type `MyDaemonInstallationNodeContribution` is created with the `createInstallationNode(...)` method. In this way, the daemon executable can be controlled from the installation node.

Listing 10: Tying different URCap contributions together

```
1  package com.ur.urcap.examples.mydaemonswing.impl;
2
3  import com.ur.urcap.api.contribution.DaemonService;
4  import com.ur.urcap.api.contribution.installation.swing.
       SwingInstallationNodeService;
5  import com.ur.urcap.api.contribution.program.swing.SwingProgramNodeService;
6  import org.osgi.framework.BundleActivator;
7  import org.osgi.framework.BundleContext;
8
9  public class Activator implements BundleActivator {
10   @Override
11   public void start(final BundleContext context) throws Exception {
12     MyDaemonDaemonService daemonService = new MyDaemonDaemonService();
13     MyDaemonInstallationNodeService installationNodeService = new
           MyDaemonInstallationNodeService(daemonService);
14
15     context.registerService(SwingInstallationNodeService.class,
           installationNodeService, null);
16     context.registerService(SwingProgramNodeService.class, new
           MyDaemonProgramNodeService(), null);
17     context.registerService(DaemonService.class, daemonService, null);
18   }
19
20   @Override
21   public void stop(BundleContext context) throws Exception {
22   }
23 }
```

10 URCap Examples Overview

This section contains a short description of each of the URCap examples included with the URCaps SDK.

10.1 Regular Samples

Hello World Swing serves as the primary example throughout this tutorial and introduces all the core concepts of a URCap. This includes contributions to PolyScope of program nodes and installation nodes that seamlessly hook into:

- The UI
- Persistence of program and installation files
- Creation and execution of programs
- Program undo/redo functionality

Information:

- *Available from:*
  - URCap API version 1.3.0.
  - PolyScope version 3.6.0/5.0.4.
- *Main API interfaces:* SwingProgramNodeService, ProgramNodeContribution, SwingProgramNodeView, ContributionProvider, SwingInstallationNodeService, InstallationNodeContribution, SwingInstallationNodeView, UndoRedoManager, DataModel, ScriptWriter.

My Daemon Swing is an extension to the Hello World Swing URCap and demonstrates how a Python 2.5 or C++ daemon can be integrated with the URCap Software Platform. This is useful when a URCap depends on e.g. a driver or server which is not implemented in Java. Furthermore, the URCap shows how the XML-RPC protocol can be used to communicate with the daemon from an installation node and in the script code generated by a program node.

Information:

- *Available from:*
  - URCap API version 1.3.0.
  - PolyScope version 3.6.0/5.0.4.
- *Main API interfaces:* DaemonContribution, DaemonService.

Script Function Swing demonstrates how to add functions to the list of available script functions in the Expression Editor. Script functions often used by end users of a URCap should be added to this list.

Information:

- *Available from:*
  - URCap API version 1.3.0.

  - PolyScope version 3.6.0/5.0.4.
- *Main API interfaces:* Function, FunctionModel

Pick or Place Swing is a toy example that shows how to make changes to the program tree through the TreeNode API. The program node service (SwingProgramNodeService interface) is configured in such a way that it creates program node contributions that can only be inserted into the program tree by a URCap and not from the UI of PolyScope by the end user.

Information:

- *Available from:*
  - URCap API version 1.3.0.
  - PolyScope version 3.6.0/5.0.4.
- *Main API interfaces:* ProgramModel, TreeNode, ProgramNodeFactory, SwingProgramNodeService, ContributionConfiguration

Ellipse Swing is a toy example, where a pose is used to define the center point for an ellipse-like movement. The movement is achieved by inserting a pre-configured MoveP program node containing pre-defined and named Waypoint nodes. This example demonstrates how to:

- Obtain a pose for the robot position by requesting the end user to define it using the Move Tab
- Name waypoints
- Request the end user to move the robot to a given target position
- Allow the end user to use the builtin PolyScope support for starting from and pausing/breaking on a selected program node in the program tree. In this case, the end user can start from or break on a specific Waypoint child node under the Ellipse (URCap) program node.

Note:

- The functionality of assigning the Waypoint nodes custom names is only available from URCap API version 1.4.0 (released with PolyScope version 3.7.0/5.1.0)
- Requesting the user to move the robot to a defined center point is only available from URCap API version 1.5.0 (released with PolyScope version 3.8.0/5.2.0).
- From URCap API version 1.6.0 (released with PolyScope version 3.9.0/5.3.0) the use of the deprecated move node config factory (MoveNodeConfigFactory interface) has been replaced with the equivalent builder and the TCP selection of the MoveP node is pre-configured.
- Support for allowing the end user to start from and break on child nodes is only available from URCap API version 1.9.0 (released with PolyScope version 5.6.0).
- From URCap API version 1.11.0 (released with PolyScope 3.14.0/5.9.0) the use of the deprecated method `getUserDefinedRobotPosition(RobotPositionCallback)` has been replaced with the equivalent `getUserDefinedRobotPosition(RobotPositionCallback2)` method. Furthermore, the use of the deprecated factory method for creating fixed position Waypoint node configurations (`createFixedPositionConfig(...)` without TCP offset) has been replaced with the equivalent method taking the TCP offset as parameter as well.

Information:

- *Available from:*
  - URCap API version 1.3.0.
  - PolyScope version 3.6.0/5.0.4.
- *Main API interfaces:* UserInteraction, RobotPositionCallback2, RobotMovement, RobotMovementCallback, WaypointNodeConfig, MovePMoveNodeConfig, MoveNodeConfigBuilders, MovePConfigBuilder, PoseFactory, Pose, SimpleValueFactory, JointPositions

Cycle Counter Swing demonstrates how to work with variables. In this example, the chosen variable will be incremented each time the program node is executed.

This sample also demonstrates how to allow the end user to use the builtin PolyScope support for starting from and pausing/breaking on a selected program node in the program tree. In this case, the end user can start from or break on any child node under the Cycle Counter (URCap) program node.

Note:

- Support for allowing the end user to start from and break on child nodes is only available from URCap API version 1.9.0 (released with PolyScope version 5.6.0)

Information:

- *Available from:*
  - URCap API version 1.3.0.
  - PolyScope version 3.6.0/5.0.4.
- *Main API interfaces:* Variable, VariableFactory, ExpressionBuilder

Idle Time Swing demonstrates how to work with the ProgramNodeVisitor to traverse all program nodes in a sub-tree. In this example, all Wait nodes will be visited. If a Wait node is configured to wait for an amount of time, that amount of idle time (in seconds) will accumulate in the selected variable.

Information:

- *Available from:*
  - URCap API version 1.3.0.
  - PolyScope version 3.6.0/5.0.4.
- *Main API interfaces:* ProgramNodeVisitor, WaitNodeConfig

Localization Swing demonstrates how to implement localization in URCaps. PolyScope localization settings is accessed through the SystemSettings API.

Information:

- *Available from:*
  - URCap API version 1.3.0.

  - PolyScope version 3.6.0/5.0.4.
- *Main API interfaces:* SystemSettings, Localization, Unit, SimpleValueFactory

User Input demonstrates how to work with the virtual on-screen keyboard/keypad and user input validation.

Information:

- *Available from:*
  - URCap API version 1.3.0.
  - PolyScope version 3.6.0/5.0.4.
- *Main API interfaces:* KeyboardInput, InputValidationFactory

My Toolbar demonstrates how to implement a PolyScope toolbar contribution.

Information:

- *Available from:*
  - URCap API version 1.3.0.
  - PolyScope version 5.0.4.
- *Main API interfaces:* SwingToolbarService, SwingToolbarContribution.

Node Ordering Swing demonstrates how to define a specific sort order in PolyScope for the program node contributions from a URCap.

Information:

- *Available from:*
  - URCap API version 1.5.0.
  - PolyScope version 3.8.0/5.2.0.
- *Main API interfaces:* ContributionConfiguration, SwingProgramNodeService.

Tool Changer Swing is a toy example that shows how contribute TCPs to PolyScope as well as access the list of available TCPs in PolyScope.

In the installation node contribution, the user can define a tool change position, enable/disable different tool TCPs and define a translational offset between the tool flange and all the enabled tool TCPs.

In the program node contribution, the user can select a TCP for the new tool from the list of all available TCPs in PolyScope. When the program node is executed, the robot will move to the user-defined tool change position, change the tool (simulated by a waiting period) and finally change the active TCP to the selected TCP.

Information:

- *Available from:*
  - URCap API version 1.5.0.

  - PolyScope version 3.8.0/5.2.0.
- *Main API interfaces:* TCPContributionModel, TCPModel.

Note:

- From URCap API version 1.11.0 (released with PolyScope 3.14.0/5.9.0) the use of the deprecated method `getUserDefinedRobotPosition(RobotPositionCallback)` has been replaced with the equivalent `getUserDefinedRobotPosition(RobotPositionCallback2)` method.

Move Until Detection Swing demonstrates how to work with the Distance and Until program nodes through the URCap API.

The program node contribution creates a Direction node that moves the robot downwards, until a sensor is triggered (through an input), or until a maximum distance is reached. If the sensor is triggered, a connected physical device (e.g. a gripper) is activated by setting an output high. If the maximum distance is travelled before the sensor is triggered, a popup is generated to display an error to the user. The user can specify the maximum distance through a text field in the Move Until Detection node.

Information:

- *Available from:*
  - URCap API version 1.6.0.
  - PolyScope version 3.9.0/5.3.0.
- *Main API interfaces:* DirectionNode, DirectionNodeConfigBuilder, UntilNode, UntilNodeConfigFactory.

Tool I/O Control Swing demonstrates how to work with the resource model and request exclusive control of the Tool I/O Interface through the URCap API. It also demonstrates how to check, if a specific capability is available on the underlying robot system (in this case the Tool Communication Interface (TCI) and Tool Output Mode features).

For further details about resource control, please see the separate *Resource Control* document. The concept of system capabilities and checking their availability is described in the separate *Capabilities* document.

Information:

- *Available from:*
  - URCap API version 1.7.0.
  - PolyScope version 3.10.0/5.4.0.
- *Main API interfaces:* ResourceModel, ControllableResourceModel, ToolIOInterfaceController, ToolIOInterface, ToolIOInterfaceControllable, CapabilityManager.

Grip and Release Swing demonstrates how to use gripper devices in a template program node.

In the program node contribution, the user can select a gripper among the registered grippers available in PolyScope. When a gripper has been selected, the selected gripper can be applied to the template node. This will insert two Gripper program nodes for

the selected gripper: one node configured for a grip action and one node configured for a release action.

This example demonstrates how to:

- Get the list of grippers available in PolyScope
- Insert a Gripper program node for a specific gripper device in the program tree
- Configure a Gripper program node for grip and release actions

Note: To have grippers to select from in the program node, some of the gripper driver URCap samples (e.g. Simple Gripper) can be installed.

For further details about how to use devices in template nodes, please see the separate *Using a Device in a Template* document.

Information:

- *Available from:*
  - URCap API version 1.9.0.
  - PolyScope version 3.12.0/5.6.0.
- *Main API interfaces:* GripperManager, GripperProgramNodeFactory, GripperNode, GripConfigBuilder, ReleaseConfigBuilder, GripperDevice.

Create Feature Swing is an example that shows how to contribute a feature to PolyScope as well as how to modify and remove it.

In the installation contribution, the user can define a feature and subsequently update and remove it. The created feature is stored in data model of the installation contribution.

In the program node contribution, the user will be asked to create a feature from the installation contribution, if not done already. When the feature is created, the user can press a button to create a movement relative to the created feature. The movement is achieved by inserting a pre-configured MoveL program node containing pre-defined Waypoint nodes. When the program node is executed, the robot will perform a square movement centered at the created feature.

Information:

- *Available from:*
  - URCap API version 1.9.0.
  - PolyScope version 3.12.0/5.6.0.
- *Main API interfaces:* FeatureContributionModel.

Note:

- From URCap API version 1.11.0 (released with PolyScope 3.14.0/5.9.0) the use of the deprecated method `getUserDefinedRobotPosition(RobotPositionCallback)` has been replaced with the equivalent `getUserDefinedRobotPosition(RobotPositionCallback2)` method. Furthermore, the use of the deprecated factory method for creating fixed position Waypoint node configurations (`createFixedPositionConfig(...)` without TCP offset) has been replaced with the equivalent method taking the TCP offset as parameter as well. The tool flange is used as the TCP offset.

10.2 Driver Samples

Custom User Inputs demonstrates how to use different types of user inputs (e.g. combo box and checkbox inputs) and other (non-user input) UI elements (e.g text components) when defining a custom UI for a driver contribution.

The URCap also shows how to use the filler UI element for controlling/grouping the layout and how to add a custom input validator for detecting errors for enterable user inputs, in this case an IP address user input (text field).

Information:

- *Available from:*
  - URCap API version 1.7.0.
  - PolyScope version 3.11.0/5.5.0.
- *Main API interfaces:* CustomUserInputConfiguration.

Simple Gripper demonstrates how to create a gripper driver contribution for a basic gripper that only supports the mandatory "default" grip and release actions. The URCap uses digital outputs in the script code generation to trigger grip and release actions.

For further details about gripper driver contributions, please see the separate *Gripper Driver* document.

Information:

- *Available from:*
  - URCap API version 1.8.0.
  - PolyScope version 3.11.0/5.5.0.
- *Main API interfaces:* GripperContribution.

Advanced Gripper demonstrates how to create a gripper driver contribution for a more advanced gripper that supports some of the optional gripper capabilities and controls the Output Voltage setting of the Tool I/O Interface resource. The URCap shows how to:

- Configure gripper capabilities for, e.g. width, speed, force, vacuum and feedback for grip and release detection
- Request exclusive control of the Tool I/O Interface resource
- Configure the Output Voltage I/O setting of the Tool I/O Interface

Additional information:

- For details about gripper driver contributions, please see the separate *Gripper Driver* document.
- For information about system resource control, please see the separate *Resource Control* document.
- To see an example of how some of the other settings of the Tool I/O Interface can be configured, see the *Tool I/O Control Swing* URCap example.

Note:

- Feedback capabilities of the gripper is only available from URCap API version 1.9.0 (released with PolyScope version 3.12.0/5.6.0)

Information:

- *Available from:*
  - URCap API version 1.8.0.
  - PolyScope version 3.11.0/5.5.0.
- *Main API interfaces:* GripperContribution, GripperCapabilities, SystemConfiguration, GripActionParameters, ReleaseActionParameters, GripperFeedbackCapabilities.

Custom Gripper Setup demonstrates how to create a gripper driver contribution which defines a custom UI in the installation node for setting up the gripper as well as adds a TCP for the gripper to PolyScope.

For further details about gripper driver contributions, please see the separate *Gripper Driver* document.

Information:

- *Available from:*
  - URCap API version 1.8.0.
  - PolyScope version 3.11.0/5.5.0.
- *Main API interfaces:* GripperContribution, TCPConfiguration, CustomUserInputConfiguration.

Dual Zone Gripper demonstrates how to create a gripper driver contribution for a multi-gripper that supports multiple permanently enabled, individual grippers as well as dynamically adjusts the parameters of a capability (after it has been registered) for all individual grippers.

In this example, the gripper is a vacuum dual gripper with two independent physical grippers (zones) "built into" the gripping device. The URCap provides the user the option to select from three individual zones (grippers), named *Zone A*, *Zone B* and *Zone A+B*. The user can operate *Zone A* and *Zone B* independently of one another or choose to use both zones at the same by selecting *Zone A+B*.

A custom UI is defined in the installation node which allows the user to enable the "Use Fragile Handling" option. When enabled, this will restrict the maximum vaccum level available for a grip action. This is achieved by reducing the maximum value and default value parameters of the registered vacuum capability simultaneously for all zones.

The URCap shows how to:

- Configure the multi-gripper capability to support multiple permanently enabled, individual grippers
- Add a dedicated TCP for each individual gripper (zone)

- Dynamically update the parameter values of a registered (parameter-based) capability for all individual grippers (zones). In this case, the registered capability is the grip vaccum capability.

For further details about gripper driver contributions, please see the separate *Gripper Driver* document.

Information:

- *Available from:*
  - URCap API version 1.11.0.
  - PolyScope version 3.14.0/5.9.0.
- *Main API interfaces:* GripperContribution, GripperCapabilities, GripperListProvider, GripperListBuilder, SelectableGripper, SystemConfiguration, TCPConfiguration, GripVacuumCapability

Dynamic Multi-Gripper demonstrates how to create a gripper driver contribution that supports both a single gripper setup and multi-gripper setup as well as dynamically adjusts the parameters of a capability (after it has been registered) exclusively for each individual gripper.

In this example, the gripper driver provides the user the option of choosing between using a setup with only a single gripper mounted on the robot, or one where two separate, identical grippers are mounted on the robot at the same time. Each individual gripper supports moving to a user configurable position (open/close to a configurable width).

A custom UI is defined in the installation node which allows the user to select how many gripper are mounted on the robot. The available options are *Single* and *Dual*. Selecting the *Single* option will disable the secondary gripper, *Gripper 2*, and only the standard gripper, *Gripper 1*, will be available to the user. Choosing the *Dual* option will enable both grippers making both of them available to the user. Depending on the option selected, the offset of TCP for the standard gripper (*Gripper 1*) is updated accordingly.

For each individual gripper (*Gripper 1* and *Gripper 2*), the user can configure the type of fingertips attached to the gripper (in the installation node). There are two options, *Standard* and *Extended*. The *Standard* option is the default whereas the *Extended* option can be selected, when fingertips with wide range is attached. Selecting the *Extended* option will increase the maximum value and default value parameters of the registered width capability exclusively for the specific individual gripper (independently of the other gripper).

The URCap shows how to:

- Configure the multi-gripper capability to support multiple individual grippers where some of the grippers are initially disabled
- Dynamically enable and disable individual grippers
- Add a dedicated TCP for each individual gripper
- Dynamically update the parameter values of a registered (parameter-based) capability exclusively for an individual gripper (independently of the other grippers). In this case, the registered capability is the width capability.

For further details about gripper driver contributions, please see the separate *Gripper Driver* document.

Information:

- *Available from:*
  - URCap API version 1.11.0.
  - PolyScope version 3.14.0/5.9.0.
- *Main API interfaces:* GripperContribution, GripperCapabilities, GripperListProvider, GripperListBuilder, SelectableGripper, MultiGripperCapability, SystemConfiguration, TCPConfiguration, WidthCapability

Simple Screwdriver demonstrates how to create a screwdriver driver contribution for a basic screwdriver that only supports the mandatory start and stop screwdriver operations. The URCap uses digital outputs in the script code generation to start and stop the screwdriver.

For further details about screwdriver driver contributions, please see the separate *Screwdriver Driver* document.

Information:

- *Available from:*
  - URCap API version 1.9.0.
  - PolyScope version 5.6.0.

Advanced Screwdriver demonstrates how to create a screwdriver driver contribution that supports some of the optional operation and operation feedback capabilities, e.g. Program Selection, Feed Screw, Drive Screw OK and Screwdrive Ready. The URCap primarily uses I/Os for the implementation of the capabilities.

For further details about screwdriver driver contributions, please see the separate *Screwdriver Driver* document.

Information:

- *Available from:*
  - URCap API version 1.9.0.
  - PolyScope version 5.6.0.

Custom Screwdriver demonstrates how to create a screwdriver contribution driver which defines a custom UI in the Screwdriving installation node for setting up the screwdriver as well as adds a default TCP for the screwdriver to PolyScope.

For further details about screwdriver driver contributions, please see the separate *Screwdriver Driver* document.

Information:

- *Available from:*
  - URCap API version 1.9.0.
  - PolyScope version 5.6.0.

11 Creating new thin Projects using a Maven Archetype

There are different ways to get started with URCap development. One is to start with an existing URCap project and modify that. When you have got a hang of it you may want to start with an empty skeleton with the basic Maven structure. So enter the directory of the URCaps SDK and type:

```
$ ./newURCap.sh
```

This prompts you with a dialog box where you select a group and artifact-id for your new URCap. An example could be `com.yourcompany` as group-id and `thenewapp` as artifact-id. Consult best practices naming conventions for Java group-ids. You must also specify the target URCap API version. Choosing an earlier version of the API will make your URCap compatible with earlier PolyScope versions, but also limit the functionality accessible through the API. Pressing Ok creates a new Maven project under the sub-folder `./com.yourcompany.thenewapp`. This project can easily be imported into an IDE for Java, e.g. Eclipse, Netbeans, or IntelliJ.

Notice that the generated `pom.xml` file has a section with a set of properties for the new URCap with meta-data for vendor, contact address, copyright, description, and short license information which will be displayed to the user when the URCap is installed in PolyScope. Update this section with the data relevant for the new URCap. See Figure 4 for an example of how this section might look.

Should you need to change the version of the URCap API to depend upon after your project has been setup, this can be done in the `pom.xml` file in your project. Here you must update to the desired version in the URCap API dependency under the `<dependencies>` section of the `pom.xml`-file as well as modify the `<import-package>` element under the `maven-bundle-plugin` if version information is present. See listings 11, 12 and 15 for examples of this

Listing 11: Modify URCap API runtime dependency in `pom.xml`

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

Listing 12: Specifying URCap API compile time dependency in `pom.xml`

```
...
<dependencies>
  ...
  <dependency>
    <groupId>com.ur.urcap</groupId>
    <artifactId>api</artifactId>
    <version>1.0.0.30</version>
    <scope>provided</scope>
  </dependency>
  ...
</dependencies>
...
```

12 Compatibility

When developing URCaps you must specify a dependency on a version of the URCap API to compile against. Using the `newURCap.sh` script mentioned in previous section, this is handled automatically for you. A given version of the API is compatible with a specific version of PolyScope (see table below). PolyScope will remain backwards compatible with earlier versions of the API. This means that if you choose to use the newest API, customers using your URCap must be running at least the version of PolyScope with which the given API was released.

It is not a problem if the customer is running a newer (future) version of PolyScope. However, it is not possible for the customer to use your URCap if he is running an earlier version of PolyScope than the one the API was released with. A good rule of thumb is thus to choose the earliest possible version of the API that supports the functionality you wish to use. This will target the broadest audience.

For instance, if you specify a dependency on the API version 1.1.0, your URCap will only run on PolyScope version 3.4.0 or newer. If you wish to target the broadest possible audience, you must use version 1.0.0 of the API and the customers must be running PolyScope version 3.3.0 or newer.

URCap API version	Min. PolyScope version
1.11.0	3.14.0/5.9.0
1.10.0	3.13.0/5.8.0
1.9.0	3.12.0/5.6.0
1.8.0	3.11.0/5.5.0
1.7.0	3.10.0/5.4.0
1.6.0	3.9.0/5.3.0
1.5.0	3.8.0/5.2.0
1.4.0	3.7.0/5.1.0
1.3.0	3.6.0/5.0.4
1.2.56	3.5.0
1.1.0	3.4.0
1.0.0	3.3.0

Figure 7: API versions and PolyScope version requirements

12.1 Advanced compatibility

Contrary to what is described above, it is possible to load a URCap depending on a newer API than what is officially supported by PolyScope. By configuring your URCap to resolve its dependencies runtime rather than install time, PolyScope will start your URCap regardless of the URCap API version dependency specified. Care must be taken to have a code execution path that does not use anything not available in the API that the given version of PolyScope supports (otherwise a `NoSuchMethodError` or `NoClassDefFoundError` will be thrown).

As an example you could have a dependency on API version 1.5.0 and run it on PolyScope version 3.7.0/5.1.0 (officially only supporting API version 1.4.0), but in the actual execution path you can only use types present in API version 1.4.0.

Listing 13: `AdvancedFeature` class showing advanced compatiblity

```
1 import com.ur.urcap.api.contribution.installation.InstallationAPIProvider;
```

```
2  import com.ur.urcap.api.domain.InstallationAPI;
3  import com.ur.urcap.api.domain.value.Pose;
4  import com.ur.urcap.api.domain.value.PoseFactory;
5  import com.ur.urcap.api.domain.value.simple.Angle;
6  import com.ur.urcap.api.domain.value.simple.Length;
7
8  public class AdvancedFeature {
9    private final InstallationAPIProvider apiProvider;
10
11   public AdvancedFeature(InstallationAPIProvider apiProvider) {
12     this.apiProvider = apiProvider;
13   }
14
15   public void addTCP() {
16     InstallationAPI installationAPI = apiProvider.getInstallationAPI();
17     PoseFactory poseFactory = installationAPI.getValueFactoryProvider().
           getPoseFactory();
18     Pose pose = poseFactory.createPose(0, 0, 100, 0, 0, 0, Length.Unit.MM,
           Angle.Unit.RAD);
19     apiProvider.getInstallationAPI().getTCPContributionModel().addTCP("
           GRIPPER_TCP", "Gripper", pose);
20   }
21 }
```

Listing 14: Usage of `AdvancedFeature` class

```
1      SoftwareVersion softwareVersion = apiProvider.getSystemAPI().
           getSoftwareVersion();
2      if ((softwareVersion.getMajorVersion() == 3 && softwareVersion.
           getMinorVersion() >= 8) ||
3        softwareVersion.getMajorVersion() == 5 && softwareVersion.
             getMinorVersion() >= 2) {
4        new AdvancedFeature(apiProvider).addTCP();
5      }
```

To have the URCap resolve at runtime the `pom.xml` must have the option `;resolution:=optional` appended to the URCap API entry in the `<import-package>` section. The full `<import-package>` section could look like this:

Listing 15: Excerpt of `pom.xml` for advanced compatibility

```
1  ...
2  <Import-Package>
3      com.ur.urcap.api*;resolution:=optional,
4      *
5  </Import-Package>
6  ...
```

This will make the URCap start up regardless of what the actual dependency states.

As mentioned you must also structure your code so no code referring unsupported API functionality is executed. Also no `import` or `catch` clauses referring to unsupported types can be present in classes that will execute. See listings 13 and 14 for an example of how to structure this. In listing 13 all code related to unsupported API functionality is located and in 14 a check for PolyScope version number is performed before creating an instance of the `AdvancedFeature` class.

If you choose to use this advanced feature, you must test your URCap carefully on all PolyScope versions you wish to support making sure all code execution paths are tested.

13 Exception Handling

All exceptions thrown and not caught in a URCap will be caught by PolyScope. If this happens when the end user selects either an installation node or a program node from the URCap, the UI provided by the URCap will be replaced by a screen displaying information about the error. In all other cases, a dialog will be shown to the end user.

The error screen and dialog will show that an exception has happened in the URCap along with meta information about the URCap. This occurs if the call stack originates from PolyScope (i.e. if the exception occurred when PolyScope called a method defined in a URCap API interface implemented by the URCap) or happens inside PolyScope (e.g. due to illegal arguments passed in an API method call). In this case, it will also contain a section showing the stack trace from the exception.

If an uncaught exception (e.g. a `NullPointerException`) happens in the source code of a URCap, PolyScope will try to identify the failing URCap. If successful, the error dialog will be shown. If PolyScope fails to identify the failing URCap, a general error dialog will be shown without meta information.

14 Troubleshooting

Internally in PolyScope, a URCap is installed as an OSGi bundle with the Apache Felix OSGi framework. For the purpose of debugging problems, it is possible to inspect various information about bundles using the Apache Felix command shell.

You can establish a shell connection to the running Apache Felix by opening a TCP connection on port 6666. Access the Apache Felix shell console by typing:

```
$ nc 127.0.0.1 6666

Felix Remote Shell Console:
============================

->
```

Note that you need to use the `nc` command, since the `telnet` command is not available on the robot, and `127.0.0.1` because `localhost` does not work on a robot.

To view a list of installed bundles and their state type the following command:

```
-> ps
START LEVEL 1
   ID   State         Level  Name
[   0] [Active     ] [    0] System Bundle (5.2.0)
[   1] [Active     ] [    1] aopalliance (1.0)
[   2] [Active     ] [    1] org.aspectj.aspectjrt (1.8.2)
[   3] [Active     ] [    1] org.netbeans.awtextra (1.0)
[   4] [Active     ] [    1] net.java.balloontip (1.2.4)
[   5] [Active     ] [    1] cglib (2.2)
[   6] [Active     ] [    1] com.ur.dashboardserver (3.3.0.SNAPSHOT)
[   7] [Active     ] [    1] com.ur.domain (3.3.0.SNAPSHOT)
...
[  56] [Active     ] [    1] com.thoughtworks.xstream (1.3.1)
[  57] [Active     ] [    1] helloworldswing (1.0.0.SNAPSHOT)
->
```

Inside the shell you can type `help` to see the list of the available commands:

```
-> help
uninstall
sysprop
bundlelevel
find
version
headers
refresh
start
obr
inspect
ps
stop
shutdown
help
update
install
log
cd
startlevel
resolve

Use 'help <command-name>' for more information.
->
```

For example, the `headers` command can be executed to display different properties of the individual installed bundles.

A URCaps and Generated Script Code

An URCap may generate script code for installation and program nodes. To aid debugging, the generated script code for both node types is annotated with URCap information.

The following example is taken from a small program using the Hello World Swing URCap.

To see the generated script code for an URCap, it is required to save the program. Assuming the program is called `hello.urp`, the corresponding script code can be found in `hello.script`.

The script code of the installation node will be surrounded with begin/end URCap comments and information about the source of the URCap and its type:

```
1  ...
2  # begin: URCap Installation Node
3  #   Source: Hello World Swing, 1.0.0.SNAPSHOT, Universal Robots
4  #   Type: Hello World Swing
5  hello_world_swing_popup_title = "Hello␣World"
6  # end: URCap Installation Node
7  ...
```

Similarly for program nodes, script code is surrounded with begin/end URCap comments and information about the source of the URCap and its type as shown below. The program node generated by PolyScope is the first label after the `# begin: URCap Program Node`, here `$ 2`. The remaining labels until `# end : URCap Program Node`, here the statement `$ 3`, are the nodes inserted under the Hello World program node.

```
1  ...
2  # begin: URCap Program Node
3  #   Source: Hello World Swing, 1.0.0.SNAPSHOT, Universal Robots
4  #   Type: Hello World Swing
5  $ 2 "Hello␣World␣Swing:␣My␣node"
6  popup("Hello␣My␣node,␣welcome␣to␣PolyScope!", hello_world_swing_popup_title,
       False, False, blocking=True)
7  $ 3 "Wait:␣0.01"
8  sleep(0.01)
9  # end: URCap Program Node
10 ...
```

B My Daemon Program and Installation Node

Listing 16: Java class defining functionality for the My Daemon installation node

```
1  package com.ur.urcap.examples.mydaemonswing.impl;
2
3  import com.ur.urcap.api.contribution.DaemonContribution;
4  import com.ur.urcap.api.contribution.InstallationNodeContribution;
5  import com.ur.urcap.api.contribution.installation.CreationContext;
6  import com.ur.urcap.api.contribution.installation.InstallationAPIProvider;
7  import com.ur.urcap.api.domain.data.DataModel;
8  import com.ur.urcap.api.domain.script.ScriptWriter;
9  import com.ur.urcap.api.domain.userinteraction.inputvalidation.
       InputValidationFactory;
10 import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardInputCallback;
11 import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardInputFactory;
12 import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardTextInput;
13
14 import java.awt.*;
15 import java.util.Timer;
16 import java.util.TimerTask;
17
18 public class MyDaemonInstallationNodeContribution implements
       InstallationNodeContribution {
19   private static final String POPUPTITLE_KEY = "popuptitle";
20
21   private static final String XMLRPC_VARIABLE = "my_daemon_swing";
22   private static final String ENABLED_KEY = "enabled";
23   private static final String DEFAULT_VALUE = "Hello␣My␣Daemon";
24   public static final int PORT = 40405;
25
26   private DataModel model;
27
28   private final MyDaemonInstallationNodeView view;
29   private final MyDaemonDaemonService daemonService;
30   private XmlRpcMyDaemonInterface xmlRpcDaemonInterface;
31   private Timer uiTimer;
32   private boolean pauseTimer = false;
33
34   private KeyboardInputFactory keyboardInputFactory;
35   private final InputValidationFactory inputValidationFactory;
36
37   public MyDaemonInstallationNodeContribution(InstallationAPIProvider
         apiProvider, MyDaemonInstallationNodeView view, DataModel model,
         MyDaemonDaemonService daemonService, CreationContext context) {
38     keyboardInputFactory = apiProvider.getUserInterfaceAPI().
           getUserInteraction().getKeyboardInputFactory();
39     inputValidationFactory = apiProvider.getUserInterfaceAPI().
           getUserInteraction().getInputValidationFactory();
40     this.view = view;
41     this.daemonService = daemonService;
42     this.model = model;
43     xmlRpcDaemonInterface = new XmlRpcMyDaemonInterface("127.0.0.1", PORT);
44     if (context.getNodeCreationType() == CreationContext.NodeCreationType.NEW)
           {
45       model.set(POPUPTITLE_KEY, DEFAULT_VALUE);
46     }
47     applyDesiredDaemonStatus();
48   }
49
50   @Override
51   public void openView() {
52     view.setPopupText(getPopupTitle());
53
```

```
54      //UI updates from non-GUI threads must use EventQueue.invokeLater (or
            SwingUtilities.invokeLater)
55      uiTimer = new Timer(true);
56      uiTimer.schedule(new TimerTask() {
57        @Override
58        public void run() {
59          EventQueue.invokeLater(new Runnable() {
60            @Override
61            public void run() {
62              if (!pauseTimer) {
63                updateUI();
64              }
65            }
66          });
67        }
68      }, 0, 1000);
69    }
70
71    @Override
72    public void closeView() {
73      if (uiTimer != null) {
74        uiTimer.cancel();
75      }
76    }
77
78    @Override
79    public void generateScript(ScriptWriter writer) {
80      writer.assign(XMLRPC_VARIABLE, "rpc_factory(\"xmlrpc\",␣\"http
            ://127.0.0.1:" + PORT + "/RPC2\")");
81      // Apply the settings to the daemon on program start in the Installation
            pre-amble
82      writer.appendLine(XMLRPC_VARIABLE + ".set_title(\"" + getPopupTitle() + "
            \")");
83    }
84
85    private void updateUI() {
86      DaemonContribution.State state = getDaemonState();
87
88      if (state == DaemonContribution.State.RUNNING) {
89        view.setStartButtonEnabled(false);
90        view.setStopButtonEnabled(true);
91      } else {
92        view.setStartButtonEnabled(true);
93        view.setStopButtonEnabled(false);
94      }
95
96      String text = "";
97      switch (state) {
98      case RUNNING:
99        text = "My␣Daemon␣runs";
100       break;
101     case STOPPED:
102       text = "My␣Daemon␣stopped";
103       break;
104     case ERROR:
105       text = "My␣Daemon␣failed";
106       break;
107     }
108
109     view.setStatusLabel(text);
110   }
111
112   public void onStartClick() {
113     model.set(ENABLED_KEY, true);
114     applyDesiredDaemonStatus();
115   }
116
```

```
117     public void onStopClick() {
118       model.set(ENABLED_KEY, false);
119       applyDesiredDaemonStatus();
120     }
121
122     private void applyDesiredDaemonStatus() {
123       new Thread(new Runnable() {
124         @Override
125         public void run() {
126           if (isDaemonEnabled()) {
127             // Download the daemon settings to the daemon process on initial
                    start for real-time preview purposes
128             try {
129               pauseTimer = true;
130               awaitDaemonRunning(5000);
131               xmlRpcDaemonInterface.setTitle(getPopupTitle());
132             } catch(Exception e){
133               System.err.println("Could␣not␣set␣the␣title␣in␣the␣daemon␣process.
                      ");
134             } finally {
135               pauseTimer = false;
136             }
137           } else {
138             daemonService.getDaemon().stop();
139           }
140         }
141       }).start();
142     }
143
144     private void awaitDaemonRunning(long timeOutMilliSeconds) throws
          InterruptedException {
145       daemonService.getDaemon().start();
146       long endTime = System.nanoTime() + timeOutMilliSeconds * 1000L * 1000L;
147       while(System.nanoTime() < endTime && (daemonService.getDaemon().getState()
            != DaemonContribution.State.RUNNING || !xmlRpcDaemonInterface.
            isReachable())) {
148         Thread.sleep(100);
149       }
150     }
151
152     public String getPopupTitle() {
153       return model.get(POPUPTITLE_KEY, DEFAULT_VALUE);
154     }
155
156     public KeyboardTextInput getInputForTextField() {
157       KeyboardTextInput keyboardInput = keyboardInputFactory.
            createStringKeyboardInput();
158       keyboardInput.setErrorValidator(inputValidationFactory.
            createStringLengthValidator(1, 255));
159       keyboardInput.setInitialValue(getPopupTitle());
160       return keyboardInput;
161     }
162
163     public KeyboardInputCallback<String> getCallbackForTextField() {
164       return new KeyboardInputCallback<String>() {
165         @Override
166         public void onOk(String value) {
167           setPopupTitle(value);
168           view.setPopupText(value);
169         }
170       };
171     }
172
173     private void setPopupTitle(String title) {
174       model.set(POPUPTITLE_KEY, title);
175       // Apply the new setting to the daemon for real-time preview purposes
```

```
176       // Note this might influence a running program, since the actual state is
              stored in the daemon.
177       try {
178         xmlRpcDaemonInterface.setTitle(title);
179       } catch(Exception e){
180         System.err.println("Could not set the title in the daemon process.");
181       }
182     }
183
184     public boolean isDefined() {
185       return model.isSet(POPUPTITLE_KEY) && getDaemonState() ==
              DaemonContribution.State.RUNNING;
186     }
187
188     private DaemonContribution.State getDaemonState() {
189       return daemonService.getDaemon().getState();
190     }
191
192     private Boolean isDaemonEnabled() {
193       return model.get(ENABLED_KEY, true); //This daemon is enabled by default
194     }
195
196     public String getXMLRPCVariable(){
197       return XMLRPC_VARIABLE;
198     }
199
200     public XmlRpcMyDaemonInterface getXmlRpcDaemonInterface() {
201       return xmlRpcDaemonInterface;
202     }
203   }
```

Listing 17: Java class defining functionality for the My Daemon program node

```
1
2   package com.ur.urcap.examples.mydaemonswing.impl;
3
4   import com.ur.urcap.api.contribution.ProgramNodeContribution;
5   import com.ur.urcap.api.contribution.program.ProgramAPIProvider;
6   import com.ur.urcap.api.domain.data.DataModel;
7   import com.ur.urcap.api.domain.script.ScriptWriter;
8   import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardInputCallback;
9   import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardInputFactory;
10  import com.ur.urcap.api.domain.userinteraction.keyboard.KeyboardTextInput;
11
12  import java.awt.*;
13  import java.util.Timer;
14  import java.util.TimerTask;
15
16  public class MyDaemonProgramNodeContribution implements
        ProgramNodeContribution {
17    private static final String NAME = "name";
18
19    private final ProgramAPIProvider apiProvider;
20    private final MyDaemonProgramNodeView view;
21    private final DataModel model;
22
23    private Timer uiTimer;
24    private KeyboardInputFactory keyboardInputFactory;
25
26    public MyDaemonProgramNodeContribution(ProgramAPIProvider apiProvider,
          MyDaemonProgramNodeView view, DataModel model) {
27      this.apiProvider = apiProvider;
28      keyboardInputFactory = apiProvider.getUserInterfaceAPI().
            getUserInteraction().getKeyboardInputFactory();
29      this.view = view;
30      this.model = model;
```

```
31    }
32
33    @Override
34    public void openView() {
35      view.setNameText(getName());
36
37      //UI updates from non-GUI threads must use EventQueue.invokeLater (or
          SwingUtilities.invokeLater)
38      uiTimer = new Timer(true);
39      uiTimer.schedule(new TimerTask() {
40        @Override
41        public void run() {
42          EventQueue.invokeLater(new Runnable() {
43            @Override
44            public void run() {
45              updatePreview();
46            }
47          });
48        }
49      }, 0, 1000);
50    }
51
52    @Override
53    public void closeView() {
54      uiTimer.cancel();
55    }
56
57    @Override
58    public String getTitle() {
59      return "My Daemon: " + getName();
60    }
61
62    @Override
63    public boolean isDefined() {
64      return getInstallation().isDefined() && !getName().isEmpty();
65    }
66
67    @Override
68    public void generateScript(ScriptWriter writer) {
69      // Interact with the daemon process through XML-RPC calls
70      // Note, alternatively plain sockets can be used.
71      writer.assign("mydaemon_message", getInstallation().getXMLRPCVariable() +
          ".get_message(\"" + getName() + "\")");
72      writer.assign("mydaemon_title", getInstallation().getXMLRPCVariable() + ".
          get_title()");
73      writer.appendLine("popup(mydaemon_message, mydaemon_title, False, False,
          blocking=True)");
74      writer.writeChildren();
75    }
76
77    private void updatePreview() {
78      String title;
79      String message;
80      try {
81        // Provide a real-time preview of the daemon state
82        title = getInstallation().getXmlRpcDaemonInterface().getTitle();
83        message = getInstallation().getXmlRpcDaemonInterface().getMessage(
              getName());
84      } catch (Exception e) {
85        System.err.println("Could not retrieve essential data from the daemon
              process for the preview.");
86        title = message = "<Daemon disconnected>";
87      }
88
89      view.setTitlePreview(title);
90      view.setMessagePreview(message);
91    }
```

```
92
93    public KeyboardTextInput getInputForTextField() {
94      KeyboardTextInput keyboardTextInput = keyboardInputFactory.
            createStringKeyboardInput();
95      keyboardTextInput.setInitialValue(getName());
96      return keyboardTextInput;
97    }
98
99    public KeyboardInputCallback<String> getCallbackForTextField() {
100     return new KeyboardInputCallback<String>() {
101       @Override
102       public void onOk(String value) {
103         setName(value);
104         view.setNameText(value);
105         updatePreview();
106       }
107     };
108   }
109
110   private String getName() {
111     return model.get(NAME, "");
112   }
113
114   private void setName(String name) {
115     if ("".equals(name)){
116       model.remove(NAME);
117     }else{
118       model.set(NAME, name);
119     }
120   }
121
122   private MyDaemonInstallationNodeContribution getInstallation(){
123     return apiProvider.getProgramAPI().getInstallationNode(
            MyDaemonInstallationNodeContribution.class);
124   }
125 }
```

Listing 18: Java class for XML-RPC communication

```
1  package com.ur.urcap.examples.mydaemonswing.impl;
2
3  import org.apache.xmlrpc.XmlRpcException;
4  import org.apache.xmlrpc.client.XmlRpcClient;
5  import org.apache.xmlrpc.client.XmlRpcClientConfigImpl;
6
7  import java.net.MalformedURLException;
8  import java.net.URL;
9  import java.util.ArrayList;
10
11 public class XmlRpcMyDaemonInterface {
12
13   private final XmlRpcClient client;
14
15   public XmlRpcMyDaemonInterface(String host, int port) {
16     XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
17     config.setEnabledForExtensions(true);
18     try {
19       config.setServerURL(new URL("http://" + host + ":" + port + "/RPC2"));
20     } catch (MalformedURLException e) {
21       e.printStackTrace();
22     }
23     config.setConnectionTimeout(1000); //1s
24     client = new XmlRpcClient();
25     client.setConfig(config);
26   }
27
```

```
28    public boolean isReachable() {
29      try {
30        client.execute("get_title", new ArrayList<String>());
31        return true;
32      } catch (XmlRpcException e) {
33        return false;
34      }
35    }
36
37    public String getTitle() throws XmlRpcException, UnknownResponseException {
38      Object result = client.execute("get_title", new ArrayList<String>());
39      return processString(result);
40    }
41
42    public String setTitle(String title) throws XmlRpcException,
          UnknownResponseException {
43      ArrayList<String> args = new ArrayList<String>();
44      args.add(title);
45      Object result = client.execute("set_title", args);
46      return processString(result);
47    }
48
49    public String getMessage(String name) throws XmlRpcException,
          UnknownResponseException {
50      ArrayList<String> args = new ArrayList<String>();
51      args.add(name);
52      Object result = client.execute("get_message", args);
53      return processString(result);
54    }
55
56    private boolean processBoolean(Object response) throws
          UnknownResponseException {
57      if (response instanceof Boolean) {
58        Boolean val = (Boolean) response;
59        return val.booleanValue();
60      } else {
61        throw new UnknownResponseException();
62      }
63    }
64
65    private String processString(Object response) throws
          UnknownResponseException {
66      if (response instanceof String) {
67        return (String) response;
68      } else {
69        throw new UnknownResponseException();
70      }
71    }
72  }
```

Listing 19: hello-world.py Python 2.5 daemon example

```
#!/usr/bin/env python

import sys

from SimpleXMLRPCServer import SimpleXMLRPCServer
from SocketServer import ThreadingMixIn

title = ""

def set_title(new_title):
  global title
  title = new_title
  return title

def get_title():
  tmp = ""
  if str(title):
    tmp = title
  else:
    tmp = "No title set"
  return tmp + " (Python)"

def get_message(name):
  if str(name):
    return "Hello " + str(name) + ", welcome to PolyScope!"
  else:
    return "No name set"

sys.stdout.write("MyDaemon daemon started")
sys.stderr.write("MyDaemon daemon started")

class MultithreadedSimpleXMLRPCServer(ThreadingMixIn, SimpleXMLRPCServer):
  pass

server = MultithreadedSimpleXMLRPCServer(("127.0.0.1", 40405))
server.RequestHandlerClass.protocol_version = "HTTP/1.1"
server.register_function(set_title, "set_title")
server.register_function(get_title, "get_title")
server.register_function(get_message, "get_message")
server.serve_forever()
```