# URCap Software Development Tutorial

Universal Robots A/S

Version 1.2.56

**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 1.2.56 to develop and deploy URCaps for PolyScope version 3.5.0.

# 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 1.2.56 of the URCap Software Platform which is released together with PolyScope version 3.5.0.

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 is defined using a subset of HTML and CSS. The behaviour of a customized node, data persistence and script code generation is 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* and the *My Daemon* URCaps as running examples. These are very simple and basic URCaps.

## 1.1 Features in URCap Software Platform 1.2.56

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

- Customized installation nodes and corresponding screens with text, images, and input widgets (e.g. a text field).
- Customized program nodes and corresponding screens with text, images, and input widgets.
- 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.
- Behaviour for widgets and other elements on the customized screens.

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.
- Behaviour for widgets and other elements on the customized screens.

# 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 3.5.0 in order to install the developed URCap, if it is using URCap API version 1.2.56. 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* 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 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. A description of the directories and files contained in this file is given below:

`/artifacts/`: This directory holds all released versions of the API in seperate 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-1.2.56*.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-1.2.41.jar` file, that can be used to create a new empty template URCap project.

`/doc/`: The directory contains this tutorial as well as a document describing how to configure child program nodes in a sub-tree and a document explaining how to work with variables.

`/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 control boxes CB3.0 and CB3.1.

`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).

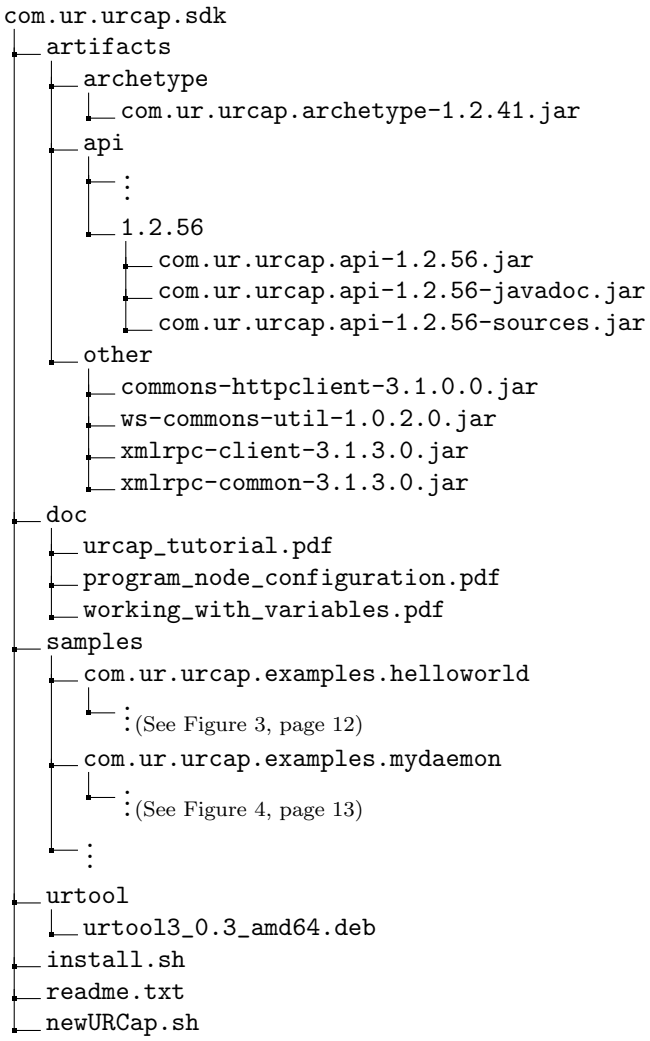



Figure 1: File structure of the URCaps SDK

`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/com.ur.urcap.examples.helloworld` directory and compile the example by:

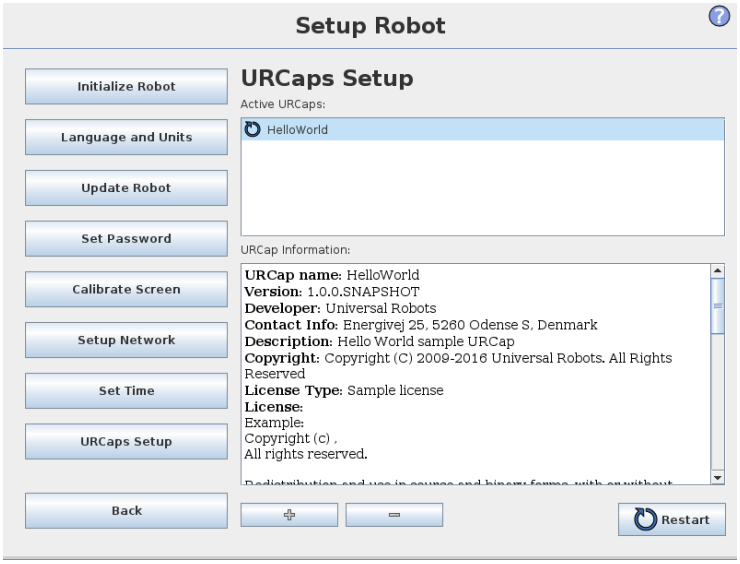
```
$ cd samples/com.ur.urcap.examples.helloworld
$ mvn install
```

A new URCap with file name `target/helloworld-1.0-SNAPSHOT.urcap` has been born! A similar procedure should be followed to compile the other URCap examples.

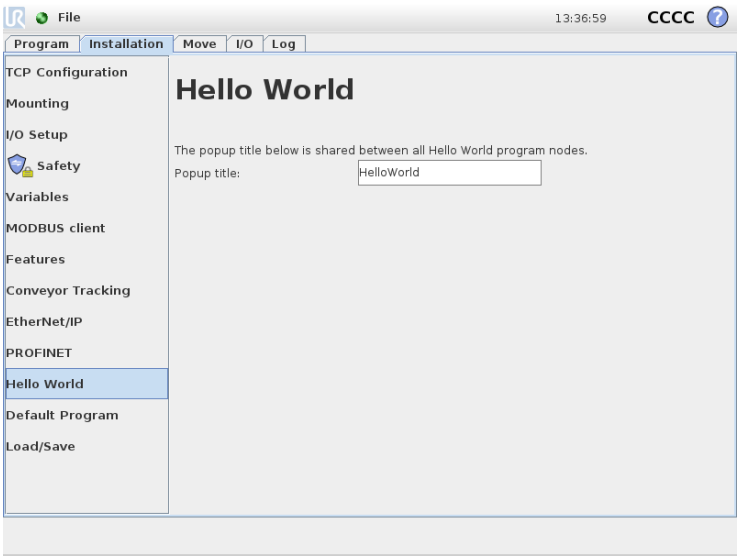
## 4.2 Manual deployment

The URCap can be added to PolyScope with these steps:

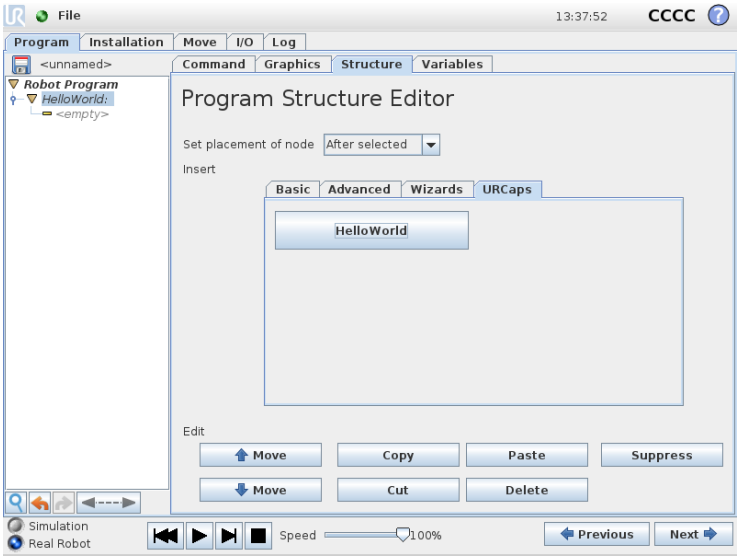
1. Copy the `helloworld-1.0-SNAPSHOT.urcap` file from above to your `programs` directory used by PolyScope or to a USB stick and insert it into a robot.
2. Tab the Setup Robot button from the PolyScope Robot User Interface Screen (the Welcome screen).
3. Tab the URCaps Setup button from the Setup Robot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



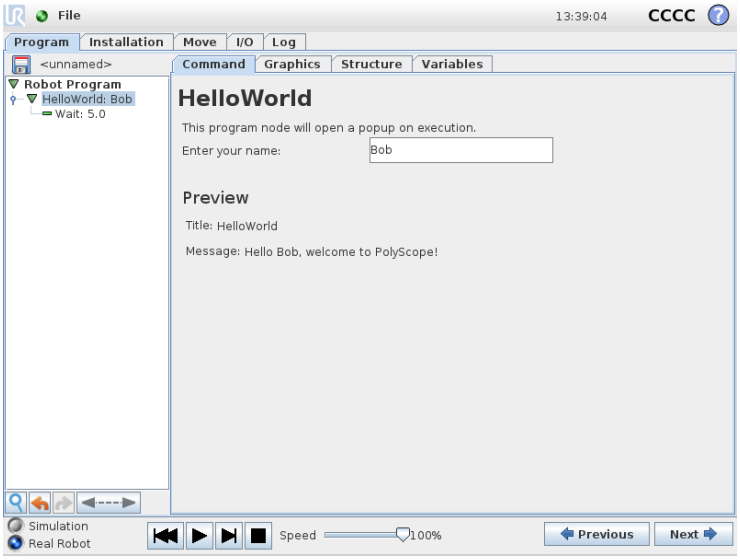

When the Hello World URCap is deployed, the following installation screen is accessible from the Installation tab:




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 "HelloWorld" (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 3, page 12, shows the structure of the Hello World URCap project. This project consists of the following parts:

1. A *view* part consisting of two screens with the layout specified in the files `installation.html` and `programnode.html`.
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 are shown to the user when the URCap is installed.
4. Maven configuration files `pom.xml` and `assembly.xml` for building the project.

The My Daemon URCap is an extended version of the Hello World URCap, that exemplifies the integration of an external daemon process. Figure 4, page 13, shows the structure of the My Daemon URCap project. Compared to the Hello World 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.

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 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 2 for the Hello World version of these properties.

```
<!--******************** BEGINNING OF URCAP META DATA
    ********************-->
<urcap.symbolicname>com.ur.urcap.examples.helloworld</
    urcap.symbolicname>
<urcap.vendor>Universal Robots</urcap.vendor>
<urcap.contactAddress>Energivej 25, 5260 Odense S, Denmark</
    urcap.contactAddress>
<urcap.copyright>Copyright (C) 2009-2017 Universal Robots. All
    Rights Reserved</urcap.copyright>
<urcap.description>Hello World sample URCap</urcap.description>
<urcap.licenseType>Sample license</urcap.licenseType>
<!--********************** END OF URCAP META DATA
    **********************-->
```

Figure 2: Section with meta-data properties inside the `pom.xml` file for the Hello World 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:

```
$ cd samples/com.ur.urcap.examples.helloworld
$ 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:

```
$ cd samples/com.ur.urcap.examples.helloworld
$ 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`.


Figure 3: Structure of the Hello World URCap project

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

```
$ cd samples/com.ur.urcap.examples.helloworld
$ mvn install -P ursim
```

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

```
$ cd samples/com.ur.urcap.examples.helloworld
$ 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.

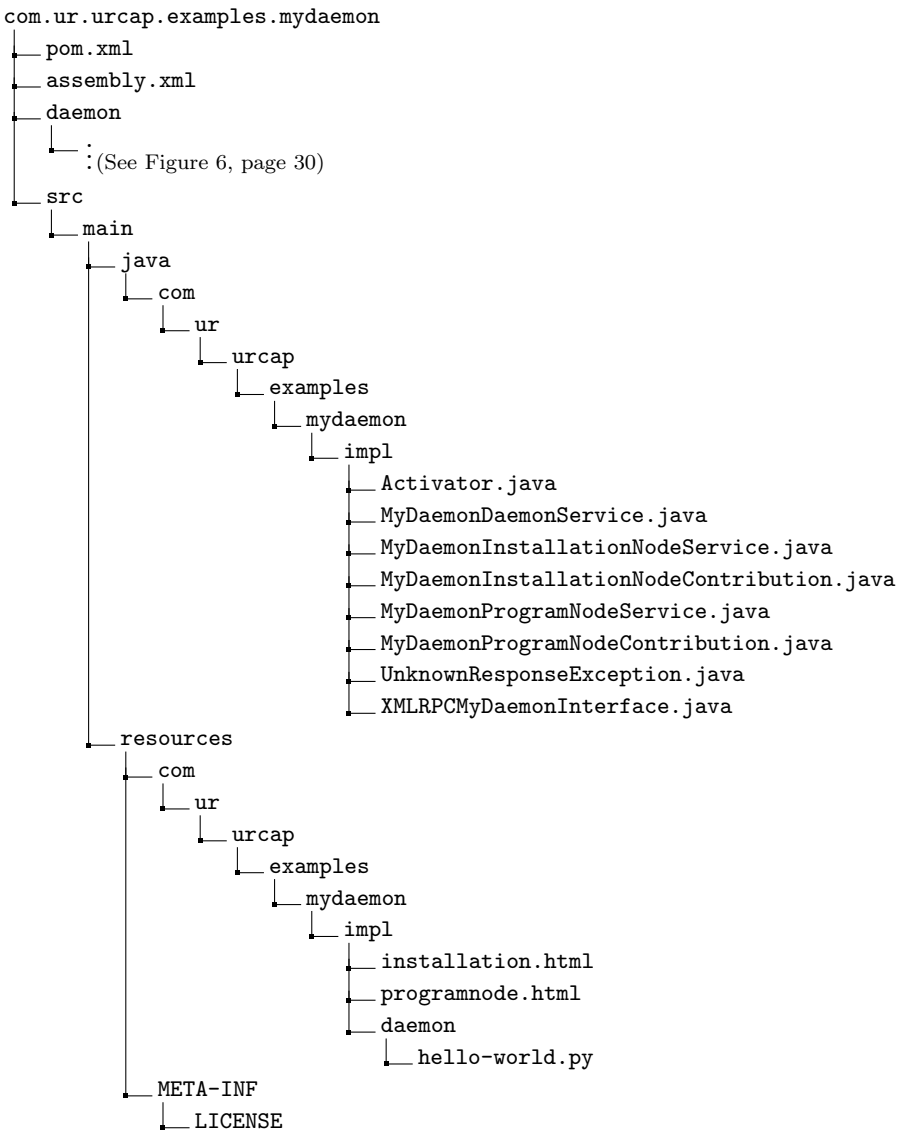



Figure 4: Structure of the My Daemon URCap project

# 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 Layout of the Installation Node

The layout of a customized installation node screen is specified using a HTML document. At the moment only a fragment of valid CSS styling properties and HTML are supported. The most important HTML elements that are supported are various form input elements, labels, headings, paragraphs, and divisions. For a detailed overview of supported HTML and CSS, consult appendix B.

Listing 1: HTML document that specifies the layout of the customized Hello World installation screen

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>Hello World</title>
5      <style>
6        label, input {
7          display: inline-block;
8          width: 200px;
9          height: 28px;
10       }
11
12       div.spacer {
13         padding-top: 10px;
14       }
15     </style>
16   </head>
17   <body>
18     <h1>Hello World</h1>
19     <div class="spacer"> </div>
20     <form>
21       <p>The popup title below is shared between all Hello World
             program nodes.</p><br \>
22       <p>The title cannot be empty.</p><br \>
23       <label>Popup title:</label><input id="popuptitle" type="
             text"/>
24     </form>
25   </body>
26 </html>
```

Listing 1 shows the content of the `installation.html` file which defines the layout of the screen used for the Hello World installation node. The listing begins with CSS styling properties (defined within the `<style>` tag). After that follows a simple document with a heading, a paragraph, a label and a single text input widget.

In order to connect the HTML GUI specification to your Java implementation an `id` attribute is specified for the text field to access its value.
This creates a model-view separation where the `id` is used to wire a Java object with the HTML widgets. In this way, methods in the Java implementation can be used to define behaviour for widgets identified by `id` attributes. 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 HTML and the customized installation nodes available to PolyScope, a Java class that implements the interface `InstallationNodeService` 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.helloworld.impl;
2
3  import com.ur.urcap.api.contribution.
       InstallationNodeContribution;
4  import com.ur.urcap.api.contribution.InstallationNodeService;
5  import com.ur.urcap.api.domain.URCapAPI;
6
7  import java.io.InputStream;
8
9  import com.ur.urcap.api.domain.data.DataModel;
10
11 public class HelloWorldInstallationNodeService implements
       InstallationNodeService {
12
13   public HelloWorldInstallationNodeService() { }
14
15   @Override
16   public InstallationNodeContribution createInstallationNode(
         URCapAPI api, DataModel model) {
17     return new HelloWorldInstallationNodeContribution(model);
18   }
19
20   @Override
21   public String getTitle() {
22     return "Hello World";
23   }
24
25   @Override
26   public InputStream getHTML() {
27     InputStream is = this.getClass().getResourceAsStream("/com/
           ur/urcap/examples/helloworld/impl/installation.html");
28     return is;
29   }
30 }
```

The `InstallationNodeService` interface requires the following methods to be defined:

- `getHTML()` returns an input stream with the HTML which is passed into PolyScope.
- `getTitle()` 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()` method would be translated into the language specified by standard Java localization, based on the locale provided by `Locale.getDefault()`.
- `createInstallationNode(URCapAPI, DataModel)` is called by PolyScope when it needs to create an instance of the installation node. Both a `URCapAPI` and a `DataModel` object is passed in as arguments. The first gives access to the domain of PolyScope and the second provides a data model with automatic persistence. The constructor used in the implementation of the method `createInstallationNode(...)` is discussed in section 7.3.

## 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 the HTML widgets to concrete field variables, gives access to a data model with automatic persistence, listening for GUI events 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.helloworld.impl;
2
3  import com.ur.urcap.api.contribution.
       InstallationNodeContribution;
4  import com.ur.urcap.api.domain.data.DataModel;
5  import com.ur.urcap.api.domain.script.ScriptWriter;
6  import com.ur.urcap.api.ui.annotation.Input;
7  import com.ur.urcap.api.ui.component.InputEvent;
```

```
8  import com.ur.urcap.api.ui.component.InputTextField;
9
10 public class HelloWorldInstallationNodeContribution implements
       InstallationNodeContribution {
11
12   private static final String POPUPTITLE_KEY = "popuptitle";
13   private static final String DEFAULT_VALUE = "HelloWorld";
14
15   private DataModel model;
16
17   public HelloWorldInstallationNodeContribution(DataModel model)
         {
18     this.model = model;
19   }
20
21   @Input(id = POPUPTITLE_KEY)
22   private InputTextField popupTitleField;
23
24   @Input(id = POPUPTITLE_KEY)
25   public void onMessageChange(InputEvent event) {
26     if (event.getEventType() == InputEvent.EventType.ON_CHANGE)
           {
27       setPopupTitle(popupTitleField.getText());
28     }
29   }
30
31   @Override
32   public void openView() {
33     popupTitleField.setText(getPopupTitle());
34   }
35
36   @Override
37   public void closeView() { }
38
39   public boolean isDefined() {
40     return !getPopupTitle().isEmpty();
41   }
42
43   @Override
44   public void generateScript(ScriptWriter writer) {
45     // Store the popup title in a global variable so it is
           globally available to all HelloWorld program nodes.
46     writer.assign("hello_world_popup_title", "\"" +
           getPopupTitle() + "\"");
47   }
48
49   public String getPopupTitle() {
50     if (!model.isSet(POPUPTITLE_KEY)) {
51       resetToDefaultValue();
52     }
53     return model.get(POPUPTITLE_KEY, DEFAULT_VALUE);
54   }
55
56   private void setPopupTitle(String message) {
57     if ("".equals(message)) {
58       resetToDefaultValue();
59     } else {
```

```
60          model.set(POPUPTITLE_KEY, message);
61        }
62      }
63
64      private void resetToDefaultValue() {
65        popupTitleField.setText(DEFAULT_VALUE);
66        model.set(POPUPTITLE_KEY, DEFAULT_VALUE);
67      }
68    }
```

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.

The HTML text input widget has an `id` attribute equal to `"popuptitle"`. This attribute maps the HTML widget to an object of type `InputTextField` which permits basic operations on text fields. This is achieved using the annotation `@Input` by specifying its argument `id`.

Particularly, when the user interacts with the widget by, e.g., entering some text, the method `onMessageChange(InputEvent)` is called. Its argument indicates what kind of interaction has occurred. 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 content of the widget changes. By saving and loading the robot installation you will notice that values are stored and read again from and back to the `popupTitleField`.

The `openView()` method is called whenever the user enters the screen. It sets the contents of the text input widget to the current value stored within the member field `message`. 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 `generateScript(ScriptWriter)` method. In the assignment, the script variable named `"hello_world_popup_title"` is assigned to a string that contains the popup title stored within the data model object.

# 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 Layout of the Program Node

The layout for customized program node screens is defined similarly as the layout of customized installation node screens (see section 7.1). Listing 4 defines the layout of a simple program node. It contains a single input widget where the

user can type a name and two labels that provide a preview of the popup that will be displayed at runtime. The label with id `"titlePreviewLabel"` 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 id `"messagePreviewLabel"`. The Java code that underlies these widgets is presented in the following two sections.

Listing 4: Layout of the customized Hello World program node

```
<!DOCTYPE html>
<html>
  <head>
    <title>HelloWorld</title>
    <style>
      label, input {
        display: inline-block;
        width: 200px;
        height: 28px;
      }

      #preview {
        display: block;
        padding: 20px 20px 20px 0px;
      }
    </style>
  </head>
  <body>
    <form>
      <p>This program node will open a popup on execution.</p><
          br \>
      <label>Enter your name:</label> <input id="yourname" type=
          "text"/>
      <br/>
      <div id="preview">
        <h3>Preview</h3><br \>
        Title: <label id="titlePreviewLabel" style="width: 400px
            ;"/> <br \>
        Message: <label id="messagePreviewLabel" style="width: 
            400px;"/>
      </div>
    </form>
  </body>
</html>
```

## 8.2 Making the customized program nodes available to PolyScope

To make the Hello World program node available to PolyScope, a Java class that implements the `ProgramNodeService` 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.

Its `getTitle()` 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 heading on the Command tab for such nodes.

Letting the method `isDeprecated()` return `true` makes it impossible to create new program nodes of this type, but still support loading program nodes of this type in existing programs.

If the method `isChildrenAllowed()` returns `true`, it signals that it is possible for the program node to contain other (child) program nodes.

Finally, `createNode(URCapAPI, DataModel)` creates program nodes with references to the `URCapAPI` and the `DataModel`. The first gives access to the domain of PolyScope and the second gives the user a data model with automatic persistence. 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 widgets 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.

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

```
1  package com.ur.urcap.examples.helloworld.impl;
2
3  import com.ur.urcap.api.contribution.ProgramNodeContribution;
4  import com.ur.urcap.api.contribution.ProgramNodeService;
5  import com.ur.urcap.api.domain.URCapAPI;
6  import com.ur.urcap.api.domain.data.DataModel;
7
8  import java.io.InputStream;
9
10 public class HelloWorldProgramNodeService implements
       ProgramNodeService {
11
12   public HelloWorldProgramNodeService() {
13   }
14
15   @Override
16   public String getId() {
17     return "HelloWorldNode";
18   }
19
20   @Override
21   public String getTitle() {
22     return "HelloWorld";
```

```
23    }
24
25    @Override
26    public InputStream getHTML() {
27      InputStream is = this.getClass().getResourceAsStream("/com/
            ur/urcap/examples/helloworld/impl/programnode.html");
28      return is;
29    }
30
31    @Override
32    public boolean isDeprecated() {
33      return false;
34    }
35
36    @Override
37    public boolean isChildrenAllowed() {
38      return true;
39    }
40
41    @Override
42    public ProgramNodeContribution createNode(URCapAPI api,
          DataModel model) {
43      return new HelloWorldProgramNodeContribution(api, model);
44    }
45  }
```

## 8.3 Functionality of the Program Node

The functionality of the Hello World program node is implemented in the Java class in Listing 6. This class implements the `ProgramNodeContribution` interface and instances of this class are returned by the `createNode(URCapAPI, DataModel)` 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.helloworld.impl;
2
3  import com.ur.urcap.api.contribution.ProgramNodeContribution;
4  import com.ur.urcap.api.domain.URCapAPI;
5  import com.ur.urcap.api.domain.data.DataModel;
6  import com.ur.urcap.api.domain.script.ScriptWriter;
7  import com.ur.urcap.api.ui.annotation.Input;
8  import com.ur.urcap.api.ui.annotation.Label;
9  import com.ur.urcap.api.ui.component.InputEvent;
10 import com.ur.urcap.api.ui.component.InputTextField;
11 import com.ur.urcap.api.ui.component.LabelComponent;
12
13 public class HelloWorldProgramNodeContribution implements
       ProgramNodeContribution {
14   private static final String NAME = "name";
15
16   private final DataModel model;
17   private final URCapAPI api;
18
```

```
19    public HelloWorldProgramNodeContribution(URCapAPI api,
          DataModel model) {
20      this.api = api;
21      this.model = model;
22    }
23
24    @Input(id = "yourname")
25    private InputTextField nameTextField;
26
27    @Label(id = "titlePreviewLabel")
28    private LabelComponent titlePreviewLabel;
29
30    @Label(id = "messagePreviewLabel")
31    private LabelComponent messagePreviewLabel;
32
33    @Input(id = "yourname")
34    public void onInput(InputEvent event) {
35      if (event.getEventType() == InputEvent.EventType.ON_CHANGE)
            {
36        setName(nameTextField.getText());
37        updatePopupMessageAndPreview();
38      }
39    }
40
41    @Override
42    public void openView() {
43      nameTextField.setText(getName());
44      updatePopupMessageAndPreview();
45    }
46
47    @Override
48    public void closeView() {
49    }
50
51    @Override
52    public String getTitle() {
53      return "HelloWorld: " + (model.isSet(NAME) ? getName() : "")
            ;
54    }
55
56    @Override
57    public boolean isDefined() {
58      return getInstallation().isDefined() && !getName().isEmpty()
            ;
59    }
60
61    @Override
62    public void generateScript(ScriptWriter writer) {
63      // Directly generate this Program Node's popup message +
            access the popup title through a global variable
64      writer.appendLine("popup(\"" + generatePopupMessage() + "\",
            hello_world_popup_title, False, False, blocking=True)")
            ;
65      writer.writeChildren();
66    }
67
68    private String generatePopupMessage() {
```

```
69      return model.isSet(NAME) ? "Hello␣" + getName() + ",␣welcome
            ␣to␣PolyScope!" : "No␣name␣set";
70    }
71
72    private void updatePopupMessageAndPreview() {
73      messagePreviewLabel.setText(generatePopupMessage());
74      titlePreviewLabel.setText(getInstallation().isDefined() ?
            getInstallation().getPopupTitle() : "No␣title␣set");
75    }
76
77    private String getName() {
78      return model.get(NAME, "");
79    }
80
81    private void setName(String name) {
82      if ("".equals(name)){
83        model.remove(NAME);
84      }else{
85        model.set(NAME, name);
86      }
87    }
88
89    private HelloWorldInstallationNodeContribution getInstallation
          () {
90      return api.getInstallationNode(
            HelloWorldInstallationNodeContribution.class);
91    }
92
93  }
```

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 8.2), 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 widget, the constructed message is displayed on the screen in the label with id `"messagePreviewLabel"`. 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_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 `URCapAPI` class. The Hello World program node utilizes this approach in its `updatePopupMessageAndPreview()` method.

## 8.4 Undo/redo functionality

The supplied data model automatically supports undo/redo functionality in the sense that every bulk change to the data model is recorded on the undo stack. A bulk change is every change that happens to the data model in the scope of a user-initiated event to the URCap. Changes to the sub-tree and its properties done in the scope of the event are also part of the bulk change.

If changes made to properties of nodes in the sub-tree happen outside the scope of a user-initiated event, these changes will not be recorded on the undo stack.

For example, if two changes are made to the data model and a node with a property set is added to the sub-tree as a result of the user clicking a button in the user interface, this counts as one undo action. 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 is made, 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.

## 8.5 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 given to the method `createNode(URCapAPI, DataModel)` is empty.

When a program is loaded, the method `createNode(URCapAPI, DataModel)` 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.

For creating sub-trees a program model can be used. In Chapter 10 it is exemplified 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 `URCapAPI`.

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

# 9 Contribution of a daemon

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

However, the My Daemon 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 4, page 13, shows the structure of the My Daemon URCap project.

## 9.1 Daemon Service

A URCap can contribute any number of daemon executables through implementation of the `DaemonService` interface (see Listing 7):

- 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"`.

## 9.2 Interaction with the daemon

The My Daemon installation screen is shown in Figure 5, page 28, and the code can be found in Listing 14, page 43, in Appendix C.

Two buttons have been added to the installation screen to enable and disable the daemon. The button widgets are accessible in Java in a similar manner as described in Section 7.3, i.e. through annotations with `id` attributes (`btnEnableDaemon` and `btnDisableDaemon` respectively). 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 a `java.util.Timer`. 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 `closeView()`) to conserve computing resources.

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

Listing 7: The My Daemon Service

```
1  package com.ur.urcap.examples.mydaemon.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/mydaemon/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/mydaemon/impl/
             daemon/hello-world.py"); // Python executable
30       // return new URL("file:com/ur/urcap/examples/mydaemon/
             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 }
```

- 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

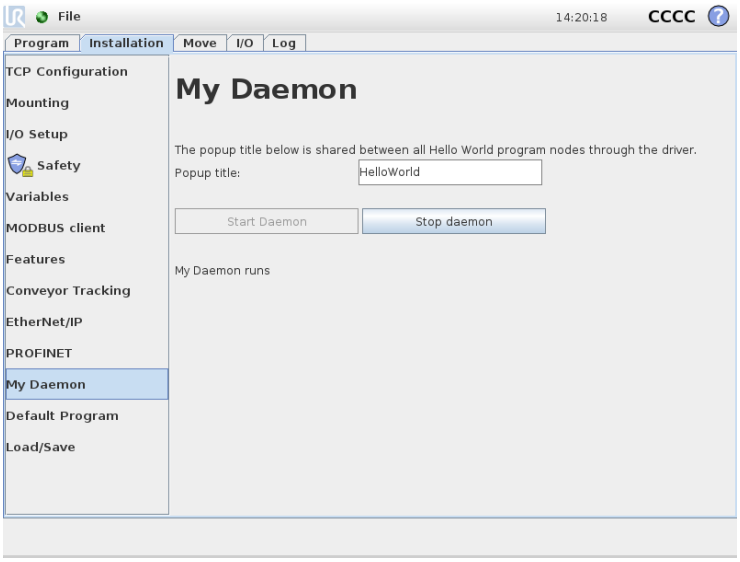



Figure 5: My Daemon installation screen

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 8: URScript XML-RPC example

```
1 global mydaemon = rpc_factory("xmlrpc", "http://127.0.0.1:40404/
    RPC2")
2 global mydaemon_message = mydaemon.get_message("Bob")
3 popup(mydaemon_message, "My Title", False, False, blocking=True)
```

Listing 8 shows a small URScript example for making a XML-RPC call to a XML-RPC server. The `hello-world.py` example daemon (see Listing 17, page 51) can be used as XML-RPC test server. Simply start the daemon in the My Daemon URCap and run the URScript in a `Script`-node.

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 `mydaemon` 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 URCap example also includes a Java XML-RPC client example, see the combination of the `MyDaemonProgramNodeContribution` and `XMLRPCMyDaemonInterface` classes (Listing 15, page 47 and listing 16, page 49 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 and CB3.1 control boxes 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 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 7, page 27), 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, 30. 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 HelloWorld binary.

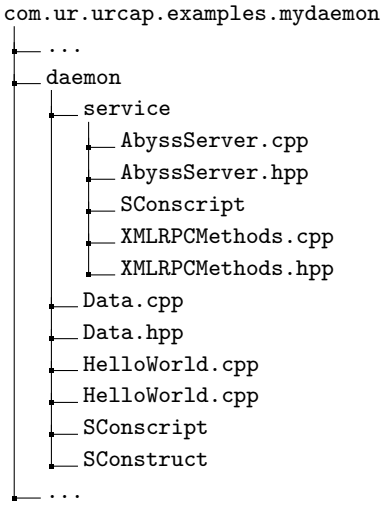



Figure 6: Structure of the C++ daemon of the My Daemon URCap project

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.mydaemon/daemon
2 scons release=1
```

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

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 both CB3.0 and CB3.1 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 installation node, program node, and daemon executable are registered and offered to PolyScope through the code in Listing 9. 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 instance of the `MyDaemonInstallationNodeContribution` installation node is created with the `createInstallationNode(URCapAPI, DataModel)` method. In this way, the daemon executable can be controlled from the installation node.

Listing 9: Tying different URCap contributions together

```
1  package com.ur.urcap.examples.mydaemon.impl;
2
3  import org.osgi.framework.BundleActivator;
4  import org.osgi.framework.BundleContext;
5  import com.ur.urcap.api.contribution.InstallationNodeService;
6  import com.ur.urcap.api.contribution.ProgramNodeService;
7  import com.ur.urcap.api.contribution.DaemonService;
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(InstallationNodeService.class,
           installationNodeService, null);
16     context.registerService(ProgramNodeService.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

Below is a short summary of each of the URCap examples included with the URCaps SDK.

**Hello World** serves as the primary example throughout this tutorial and introduces all the core concepts of a URCap. This includes contributions to PolyScope by program nodes and installation nodes that seamlessly hook into both the UI, the persistence of program and installation files and the creation and execution of programs.

- *Available from:*
  - URCap API version 1.0.0.
  - PolyScope version 3.3.0.
- *Main API interfaces:* InstallationNodeContribution, ProgramNodeContribution.

**My Daemon** is an extension to the Hello World 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.

- *Available from:*
  - URCap API version 1.0.0.
  - PolyScope version 3.3.0.
- *Main API interfaces:* DaemonContribution, DaemonService.

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

- *Available from:*
  - URCap API version 1.1.0.
  - PolyScope version 3.4.0.
- *Main API interfaces:* Function, FunctionModel

**Pick or Place** is a toy example that shows how to make changes to the program tree through the TreeNode API. The marker interface NonUserInsertable is used for program contributions that can only be inserted into the program tree by a URCap and not from the UI of PolyScope.

- *Available from:*

  - URCap API version 1.1.0.
  - PolyScope version 3.4.0.
- *Main API interfaces:* ProgramModel, TreeNode, ProgramNodeFactory

**Ellipse** demonstrates how to obtain a Pose using the Move Tab. In this example, the Pose is used as a center point for an ellipse like movement, which is achieved by inserting a configured MoveP program node containing preconfigured waypoints.

- *Available from:*
  - URCap API version 1.2.*.
  - PolyScope version 3.5.0.
- *Main API interfaces:* UserInteraction, WaypointNodeConfig, MovePMoveNodeConfig, PoseFactory

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

- *Available from:*
  - URCap API version 1.2.*.
  - PolyScope version 3.5.0.
- *Main API interfaces:* Variable, VariableFactory, ExpressionBuilder

**Idle Time** 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.

- *Available from:*
  - URCap API version 1.2.*.
  - PolyScope version 3.5.0.
- *Main API interfaces:* ProgramNodeVisitor, WaitNodeConfig

**Coordinate Map** demonstrates how to capture click or touch coordinates when the user interacts with an image.

- *Available from:*
  - URCap API version 1.2.*.

  - PolyScope version 3.5.0.
- *Main API interfaces:* TouchEvent

**Localization** demonstrates how to implement localization in URCaps. PolyScope localization settings is accessed through the SystemSettings API. Note that the script writer currently does not support Russian, Chinese, Japanese and Korean characters. This means that when one of these languages is selected, the Localization URCap popup message will display the text translated into English.

- *Available from:*
  - URCap API version 1.2.*.
  - PolyScope version 3.5.0.
- *Main API interfaces:* SystemSettings, Localization, Translatable, UnitType, ValueFactory

# 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 folder 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 2 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 the `<import-package>` element under the `maven-bundle-plugin` (without the build number part). See listings 10 and 13 for examples of this.

Listing 10: Specifying URCap API 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.

| API version | Min. PolyScope version |
|---|---|
| 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 `RuntimeException` will be thrown).

As an example you could have a dependency on API version 1.1.0 and run it on PolyScope version 3.3.0 (officially only supporting API version 1.0.0), but in the actual execution path you can only use types present in API version 1.0.0.

Listing 11: AdvancedFeature class showing advanced compatiblity

```
1  import com.ur.urcap.api.domain.URCapAPI;
2  import com.ur.urcap.api.domain.function.FunctionException;
3
4  public class AdvancedFeature {
5    private final URCapAPI api;
6
7    public AdvancedFeature(URCapAPI api) {
8      this.api = api;
9    }
10
11   public void addFunction() {
12     try {
13       api.getFunctionModel().addFunction("myFunction");
14     } catch (FunctionException e) {
15       e.printStackTrace();
16     }
17   }
18 }
```

Listing 12: Usage of AdvancedFeature class

```
1  if (api.getSoftwareVersion().getMinorVersion() >= 4) {
2      new AdvancedFeature(api).addFunction();
3  }
```

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 13: Excerpt of pom.xml for advanced compatibility

```
1  ...
2  <Import-Package>
3      com.ur.urcap.api*;version="[1.0.0,2.0.0)";resolution:=
           optional,
4      *
5  </Import-Package>
6  ...
```

This will make the URCap start up if the supported API version is between 1.0.0 and 2.0.0 (latter not inclusive) 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 11 and 12 for an example of how to structure this. In listing 11 all code related to unsupported API functionality is located and in 12 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. It will also contain a section showing the stack trace from the exception in the URCap code.

# 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:

```
1  $ nc 127.0.0.1 6666
2
3  Felix Remote Shell Console:
4  ============================
5
6  ->
```

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:

```
1  -> ps
2  START LEVEL 1
3     ID   State         Level  Name
4  [   0] [Active     ] [    0] System Bundle (5.2.0)
5  [   1] [Active     ] [    1] aopalliance (1.0)
6  [   2] [Active     ] [    1] org.aspectj.aspectjrt (1.8.2)
7  [   3] [Active     ] [    1] org.netbeans.awtextra (1.0)
8  [   4] [Active     ] [    1] net.java.balloontip (1.2.4)
9  [   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] HelloWorld (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 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: HelloWorld, 1.0.0.SNAPSHOT, Universal Robots
4  #   Type: Hello World
5  hello_world_popup_title = "HelloWorld"
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 HelloWorld program node.

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

# B CSS and HTML support

A strict subset of CSS and HTML is supported for defining the layout of customized program and installation node screens. Besides the basic HTML elements `<html>`, `<head>`, `<style>`, `<title>`, `<body>` the following HTML elements are the only supported:

1. The `<form>` element, when appearing as a child of `<body>`.
2. The `<div>` element, when appearing as a child of `<body>`, `<form>` or another `<div>`.
3. Any of the elements `<h1>`, `<h2>`, `<h3>`, `<p>`, and `<img>`, when appearing as a child of `<body>`, `<form>` or `<div>`.
4. Any of the elements `<label>`, `<input>`, and `<select>`, when appearing inside a `<form>` as a child of `<form>`, `<div>` or `<p>`.
5. The element `<option>`, when appearing as a child of `<select>`.
6. The elements `<span>`, `<em>`, `<b>`, `<i>`, `<br>`, and `<hr>`.

As for standard HTML attributes, only the `id` and `style` and `src` attributes are supported. Data input widgets correspond to HTML elements as follows:

- Text input field: `<input type="text"/>`.
- Number input field: `<input type="number"/>`.
  - A range can be specified with the attributes `min` and `max`.
  - The number input field can be restricted to the integers by setting the property `step` to the value 1, i.e. `step="1"`
- Button: `<input type="button"/>`.
- Check box: `<input type="checkbox"/>`.
- Radio button: `<input type="radio"/>`.
- Drop-down menu: `<select/>`.

Styling of HTML elements can be customized using the following CSS properties:

`display`: Modifies the layout of an element w.r.t. other elements. Allowed values are `inline`, `inline-block` and `block`.

`width`, `height`: Serve to specify the size of an element with display attribute different from `inline`. The accepted values are numbers, optionally followed by "px", or percentages of the respective dimension of the parent element.

`padding`, `padding-top`, `padding-right`, `padding-bottom`, `padding-left`: Used to specify spacing around an element. The accepted values are numbers, optionally followed by "px". The attribute `padding` may take 1, 2 or 4 values. All the other attributes take a single value.

`font-size`, `font-style`, `font-weight`: For modifying the size and type of font used within the element. Allowed values for `font-size` are numbers, optionally followed by "px", or percentages of the font size of the parent element. Allowed values for `font-style` are `normal` and `italic`. Allowed values for `font-weight` are `normal` and `bold`.

`vertical-align`: Sets the vertical aligment of text within a label element. Allowed values for `vertical-align` are `top`, `middle` and `bottom`.

`text-align`: Sets the horizontal aligment of text within a label element. Allowed values for `text-align` are `left`, `center` and `right`.

# C My Daemon Program and Installation Node

Listing 14: Java class defining functionality for the My Daemon installation node

```
1  package com.ur.urcap.examples.mydaemon.impl;
2
3  import com.ur.urcap.api.contribution.DaemonContribution;
4  import com.ur.urcap.api.contribution.
       InstallationNodeContribution;
5  import com.ur.urcap.api.domain.data.DataModel;
6  import com.ur.urcap.api.domain.script.ScriptWriter;
7  import com.ur.urcap.api.ui.annotation.Input;
8  import com.ur.urcap.api.ui.annotation.Label;
9  import com.ur.urcap.api.ui.component.InputButton;
10 import com.ur.urcap.api.ui.component.InputEvent;
11 import com.ur.urcap.api.ui.component.InputTextField;
12 import com.ur.urcap.api.ui.component.LabelComponent;
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";
22   private static final String ENABLED_KEY = "enabled";
23   private static final String DEFAULT_VALUE = "HelloWorld";
24
25   private DataModel model;
26   private final MyDaemonDaemonService daemonService;
27   private XmlRpcMyDaemonInterface xmlRpcDaemonInterface;
28   private Timer uiTimer;
29
30   public MyDaemonInstallationNodeContribution(
         MyDaemonDaemonService daemonService, DataModel model) {
31     this.daemonService = daemonService;
32     this.model = model;
33     xmlRpcDaemonInterface = new XmlRpcMyDaemonInterface("
           127.0.0.1", 40404);
34     applyDesiredDaemonStatus();
35   }
36
37   @Input(id = POPUPTITLE_KEY)
38   private InputTextField popupTitleField;
39
40   @Input(id = "btnEnableDaemon")
41   private InputButton enableDaemonButton;
42
43   @Input(id = "btnDisableDaemon")
44   private InputButton disableDaemonButton;
45
46   @Label(id = "lblDaemonStatus")
```

```
47    private LabelComponent daemonStatusLabel;
48
49    @Input(id = POPUPTITLE_KEY)
50    public void onMessageChange(InputEvent event) {
51      if (event.getEventType() == InputEvent.EventType.ON_CHANGE)
          {
52        setPopupTitle(popupTitleField.getText());
53      }
54    }
55
56    @Input(id = "btnEnableDaemon")
57    public void onStartClick(InputEvent event) {
58      if (event.getEventType() == InputEvent.EventType.ON_CHANGE)
          {
59        setDaemonEnabled(true);
60        applyDesiredDaemonStatus();
61      }
62    }
63
64    @Input(id = "btnDisableDaemon")
65    public void onStopClick(InputEvent event) {
66      if (event.getEventType() == InputEvent.EventType.ON_CHANGE)
          {
67        setDaemonEnabled(false);
68        applyDesiredDaemonStatus();
69      }
70    }
71
72    @Override
73    public void openView() {
74      enableDaemonButton.setText("Start␣Daemon");
75      disableDaemonButton.setText("Stop␣daemon");
76      popupTitleField.setText(getPopupTitle());
77
78      //UI updates from non-GUI threads must use EventQueue.
          invokeLater (or SwingUtilities.invokeLater)
79      uiTimer = new Timer(true);
80      uiTimer.schedule(new TimerTask() {
81        @Override
82        public void run() {
83          EventQueue.invokeLater(new Runnable() {
84            @Override
85            public void run() {
86              updateUI();
87            }
88          });
89        }
90      }, 0, 1000);
91    }
92
93    private void updateUI() {
94      DaemonContribution.State state = getDaemonState();
95
96      if (state == DaemonContribution.State.RUNNING) {
97        enableDaemonButton.setEnabled(false);
98        disableDaemonButton.setEnabled(true);
99      } else {
```

```
100         enableDaemonButton.setEnabled(true);
101         disableDaemonButton.setEnabled(false);
102       }
103
104       String text = "";
105       switch (state) {
106       case RUNNING:
107         text = "My Daemon runs";
108         break;
109       case STOPPED:
110         text = "My Daemon stopped";
111         break;
112       case ERROR:
113         text = "My Daemon failed";
114         break;
115       }
116       daemonStatusLabel.setText(text);
117     }
118
119     @Override
120     public void closeView() {
121       if (uiTimer != null) {
122         uiTimer.cancel();
123       }
124     }
125
126     public boolean isDefined() {
127       return !getPopupTitle().isEmpty() && getDaemonState() ==
              DaemonContribution.State.RUNNING;
128     }
129
130     @Override
131     public void generateScript(ScriptWriter writer) {
132       writer.globalVariable(XMLRPC_VARIABLE, "rpc_factory(\"xmlrpc
              \", \"http://127.0.0.1:40404/RPC2\")");
133       // Apply the settings to the daemon on program start in the
              Installation pre-amble
134       writer.appendLine(XMLRPC_VARIABLE + ".set_title(\"" +
              getPopupTitle() + "\")");
135     }
136
137     public String getPopupTitle() {
138       if (!model.isSet(POPUPTITLE_KEY)) {
139         resetToDefaultValue();
140       }
141       return model.get(POPUPTITLE_KEY, DEFAULT_VALUE);
142     }
143
144     private void setPopupTitle(String title) {
145       if ("".equals(title)) {
146         resetToDefaultValue();
147       } else {
148         model.set(POPUPTITLE_KEY, title);
149         // Apply the new setting to the daemon for real-time
                preview purposes
150         // Note this might influence a running program, since the
                actual state is stored in the daemon.
```

```
151         setDaemonTitle(title);
152       }
153     }
154
155     private void resetToDefaultValue() {
156       popupTitleField.setText(DEFAULT_VALUE);
157       model.set(POPUPTITLE_KEY, DEFAULT_VALUE);
158       setDaemonTitle(DEFAULT_VALUE);
159     }
160
161     private void setDaemonTitle(String title) {
162       try {
163         xmlRpcDaemonInterface.setTitle(title);
164       } catch(Exception e){
165         System.err.println("Could not set the title in the daemon
              process.");
166       }
167     }
168
169     private void applyDesiredDaemonStatus() {
170       if (isDaemonEnabled()) {
171         // Download the daemon settings to the daemon process on
              initial start for real-time preview purposes
172         try {
173           awaitDaemonRunning(5000);
174           xmlRpcDaemonInterface.setTitle(getPopupTitle());
175         } catch(Exception e){
176           System.err.println("Could not set the title in the
                daemon process.");
177         }
178       } else {
179         daemonService.getDaemon().stop();
180       }
181     }
182
183     private void awaitDaemonRunning(long timeOutMilliSeconds)
          throws InterruptedException {
184       daemonService.getDaemon().start();
185       long endTime = System.nanoTime() + timeOutMilliSeconds *
            1000L * 1000L;
186       while(System.nanoTime() < endTime && (daemonService.
            getDaemon().getState() != DaemonContribution.State.
            RUNNING || !xmlRpcDaemonInterface.isReachable())) {
187         Thread.sleep(100);
188       }
189     }
190
191     private DaemonContribution.State getDaemonState(){
192       return daemonService.getDaemon().getState();
193     }
194
195     private Boolean isDaemonEnabled() {
196       return model.get(ENABLED_KEY, true); //This daemon is
            enabled by default
197     }
198
199     private void setDaemonEnabled(Boolean enable) {
```

```
200       model.set(ENABLED_KEY, enable);
201     }
202
203     public String getXMLRPCVariable(){
204       return XMLRPC_VARIABLE;
205     }
206
207     public XmlRpcMyDaemonInterface getXmlRpcDaemonInterface() {
            return xmlRpcDaemonInterface; }
208   }
```

Listing 15: Java class defining functionality for the My Daemon program node

```
1   package com.ur.urcap.examples.mydaemon.impl;
2
3   import com.ur.urcap.api.contribution.ProgramNodeContribution;
4   import com.ur.urcap.api.domain.URCapAPI;
5   import com.ur.urcap.api.domain.data.DataModel;
6   import com.ur.urcap.api.domain.script.ScriptWriter;
7   import com.ur.urcap.api.ui.annotation.Input;
8   import com.ur.urcap.api.ui.annotation.Label;
9   import com.ur.urcap.api.ui.component.InputEvent;
10  import com.ur.urcap.api.ui.component.InputTextField;
11  import com.ur.urcap.api.ui.component.LabelComponent;
12
13  import java.awt.*;
14  import java.util.Timer;
15  import java.util.TimerTask;
16
17  public class MyDaemonProgramNodeContribution implements
        ProgramNodeContribution {
18    private static final String NAME = "name";
19
20    private final DataModel model;
21    private final URCapAPI api;
22    private Timer uiTimer;
23
24    public MyDaemonProgramNodeContribution(URCapAPI api, DataModel
          model) {
25      this.api = api;
26      this.model = model;
27    }
28
29    @Input(id = "yourname")
30    private InputTextField nameTextField;
31
32    @Label(id = "titlePreviewLabel")
33    private LabelComponent titlePreviewLabel;
34
35    @Label(id = "messagePreviewLabel")
36    private LabelComponent messagePreviewLabel;
37
38    @Input(id = "yourname")
39    public void onInput(InputEvent event) {
40      if (event.getEventType() == InputEvent.EventType.ON_CHANGE)
            {
```

```
41        setName(nameTextField.getText());
42        updatePreview();
43      }
44    }
45
46    @Override
47    public void openView() {
48      nameTextField.setText(getName());
49
50      //UI updates from non-GUI threads must use EventQueue.
            invokeLater (or SwingUtilities.invokeLater)
51      uiTimer = new Timer(true);
52      uiTimer.schedule(new TimerTask() {
53        @Override
54        public void run() {
55          EventQueue.invokeLater(new Runnable() {
56            @Override
57            public void run() {
58              updatePreview();
59            }
60          });
61        }
62      }, 0, 1000);
63    }
64
65    @Override
66    public void closeView() {
67      uiTimer.cancel();
68    }
69
70    @Override
71    public String getTitle() {
72      return "MyDaemon: " + (model.isSet(NAME) ? getName() : "");
73    }
74
75    @Override
76    public boolean isDefined() {
77      return getInstallation().isDefined() && !getName().isEmpty()
            ;
78    }
79
80    @Override
81    public void generateScript(ScriptWriter writer) {
82      // Interact with the daemon process through XML-RPC calls
83      // Note, alternatively plain sockets can be used.
84      writer.assign("mydaemon_message", getInstallation().
            getXMLRPCVariable() + ".get_message(\"" + getName() + "
            \")");
85      writer.assign("mydaemon_title", getInstallation().
            getXMLRPCVariable() + ".get_title()");
86      writer.appendLine("popup(mydaemon_message, mydaemon_title, 
            False, False, blocking=True)");
87      writer.writeChildren();
88    }
89
90    private void updatePreview() {
91      String title = "";
```

```
92      String message = "";
93      try {
94        // Provide a real-time preview of the daemon state
95        title = getInstallation().getXmlRpcDaemonInterface().
              getTitle();
96        message = getInstallation().getXmlRpcDaemonInterface().
              getMessage(getName());
97      } catch (Exception e) {
98        System.err.println("Could not retrieve essential data from
              the daemon process for the preview.");
99        title = message = "<Daemon disconnected>";
100     }
101
102     titlePreviewLabel.setText(title);
103     messagePreviewLabel.setText(message);
104   }
105
106   private String getName() {
107     return model.get(NAME, "");
108   }
109
110   private void setName(String name) {
111     if ("".equals(name)){
112       model.remove(NAME);
113     }else{
114       model.set(NAME, name);
115     }
116   }
117
118   private MyDaemonInstallationNodeContribution getInstallation()
          {
119     return api.getInstallationNode(
            MyDaemonInstallationNodeContribution.class);
120   }
121
122 }
```

Listing 16: Java class for XML-RPC communication

```
1  package com.ur.urcap.examples.mydaemon.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 17: hello-world.py Python 2.5 daemon example

```
1  #!/usr/bin/env python
2
3  import time
4  import sys
5
6  import xmlrpclib
7  from SimpleXMLRPCServer import SimpleXMLRPCServer
8
9  title = ""
10
11 def set_title(new_title):
12   global title
13   title = new_title
14   return title
15
16 def get_title():
17   tmp = ""
18   if str(title):
19     tmp = title
20   else:
21     tmp = "No title set"
22   return tmp + " (Python)"
23
24 def get_message(name):
25   if str(name):
26     return "Hello " + str(name) + ", welcome to PolyScope!"
27   else:
28     return "No name set"
29
30 sys.stdout.write("MyDaemon daemon started")
31 sys.stderr.write("MyDaemon daemon started")
32
33 server = SimpleXMLRPCServer(("127.0.0.1", 40404))
34 server.register_function(set_title, "set_title")
35 server.register_function(get_title, "get_title")
36 server.register_function(get_message, "get_message")
37 server.serve_forever()
```