URCap Software Development Tutorial

Universal Robots A/S

Version 1.0.0

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 1.0.0 to develop and deploy URCaps for PolyScope version 3.3.0.

This tutorial is for people with interest in URCap development. Some programming background in Java and familiarity with HTML and CSS is required.

Contents

1 Introduction

PolyScope version 3.3.0 includes the first official version (version 1.0.0) of the URCap Software Platform. 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 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.

In the remainder of this tutorial, we explain step by step how a URCap can be developed and deployed. To get started with URCap development we use the *Hello World* URCap as a running example. This is a minimal URCap that demonstrates the main features of the URCap Software Platform.

1.1 Features in URCap Software Platform 1.0.0

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.

The customized installation nodes 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 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 3.3.0 in order to install the developed URCap. A UR3, UR5, or UR10 robot can be used for that purpose or the Universal Robots offline simulator software (URSim). Go to the download area at:

`www.universal-robots.com/support`

and select the latest version and follow the given installation guidelines. The offline simulator is available for Linux and non-Linux operating systems through a virtual Linux machine.

Finally, a URCap developer will need to acquire the URCap SDK, which also includes the Hello World URCap (the focus of this tutorial). The URCap SDK is freely available at the download area of the web page shown above. The following section will describe the content of the URCap SDK.

3 URCap SDK

The URCap SDK enables a developer to create a URCap. It contains a Java package defining the API that the developer will program against, documentation, the Hello World sample URCap 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 `com.ur.urcap.sdk.zip` file. A description of the directories and files contained in this file is given below:

`/artifacts/`: This directory holds the Java package (`com.ur.urcap.api-1.0.0*.jar` files) that contains Java interfaces (the URCap API) necessary to implement the Java portion of a URCap together with the Maven archetype (`com.ur.urcap.archetype-1.0.0.jar` file) used when creating a new empty template URCap project.

`/doc/`: The directory contains this tutorial and a copy of the URScript manual.

`/samples/`: A folder containing the Hello World URCap Maven project. This includes the source code of a minimal URCap that demonstrates the main features of the URCap Software Platform, Maven configuration files used

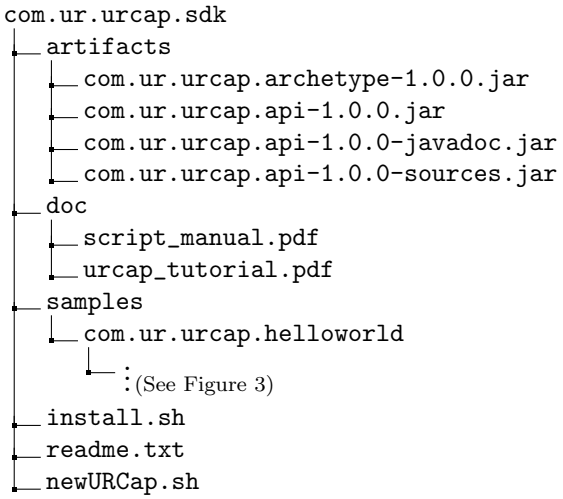


Figure 1: File structure of the URCaps SDK

in the build process and an example of a license file. This tutorial contains an in-depth description of the inner workings of this URCap.

`install.sh`: A script which should be run as a first step to install the URCap SDK (see section 4). This will install the URCap API and the Maven archetype in your local Maven repository.

`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 `com.ur.urcap.sdk.zip` file to a suitable location and run the install script inside the target location:

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

This installs the SDK on your machine.

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

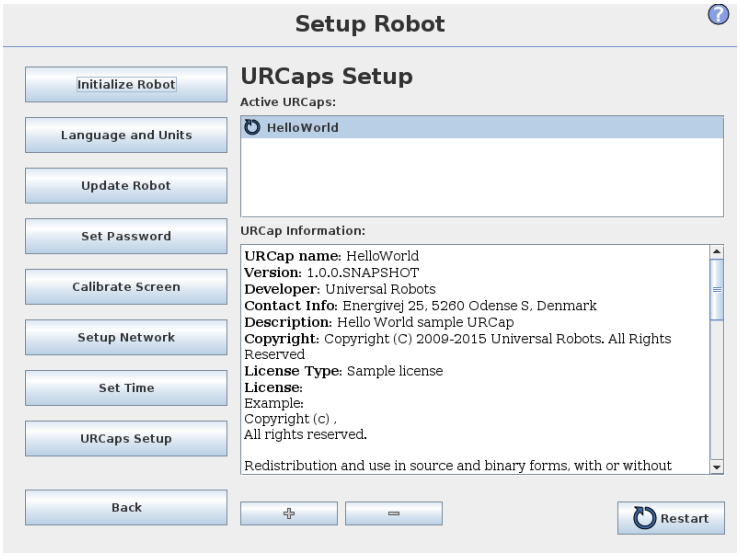
```
1    $ cd samples/com.ur.urcap.helloworld
2    $ mvn install
```

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

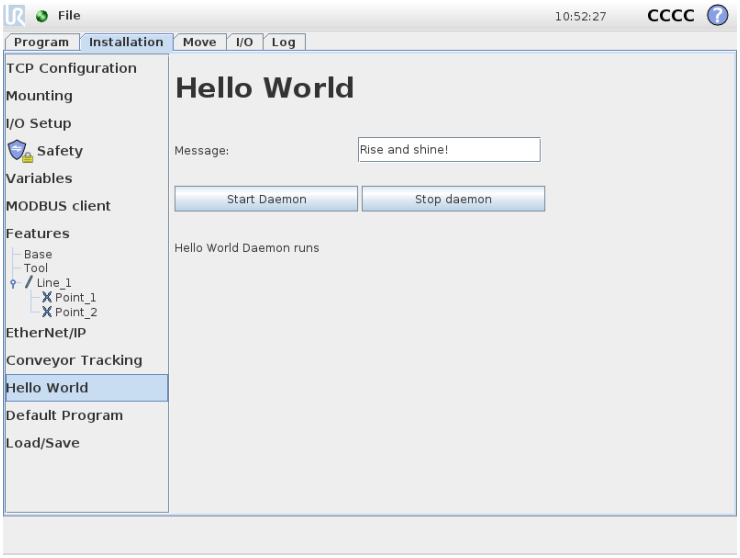
4.2 Manual deployment

The URCap can be added to PolyScope with these steps:

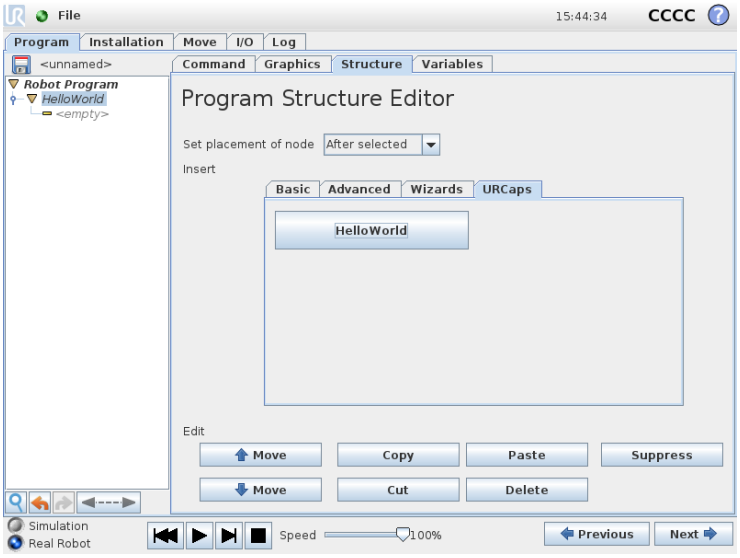
1. Copy the `helloworld-1.0-SNAPSHOT.urcap` file from above to your `programs` directory used by PolyScope or to a USB stick and insert it into a robot.
2. Tab the Setup Robot button from the PolyScope Robot User Interface Screen (the Welcome screen).
3. Tab the URCaps Setup button from the Setup Robot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



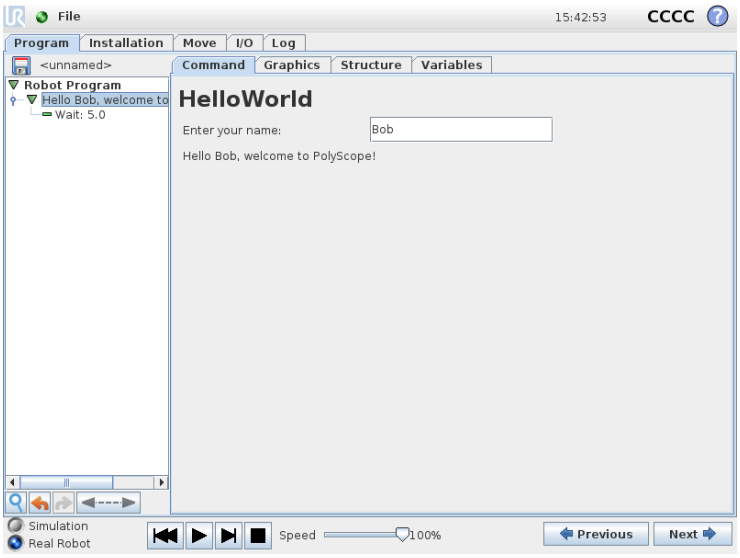
When the Hello World URCap is deployed, the following installation screen is accessible from the Installation tab:



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 is run, two pop-ups are shown. One says "Hello Bob" (i.e. the name defined in the program node) and the other says "Rise and shine!" (i.e. the message configured in the installation screen).

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 shows the structure of the Hello World URCap project. The 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 *daemon* executable in the file `hello-world-executable.py`.
4. A Java *implementation* `HelloWorldDaemonService.java` that defines and installs a daemon and makes it possible to control the daemon.
5. A *license* `META-INF/LICENSE` with the license information that are shown to the user when the URCap is installed.

6. Maven configuration files `pom.xml` and `assembly.xml` for building the project.

The services:

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

are registered in `Activator.java` and thereby a new installation node, program node, and daemon executable are offered 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.

```
1  <!--******************** BEGINNING OF URCAP META DATA
       ********************-->
2  <urcap.symbolicname>com.ur.urcap.helloworld</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) 2009-2016 Universal Robots. All
       Rights Reserved</urcap.copyright>
6  <urcap.description>Hello World sample URCap</urcap.description>
7  <urcap.licenseType>Sample license</urcap.licenseType>
8  <!--********************** 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. `10.1.1.64`, go to your URCap project folder and type:

```
1   $ cd samples/com.ur.urcap.helloworld
2   $ mvn install -Premote -Durcap.install.host=10.1.1.64
```

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:

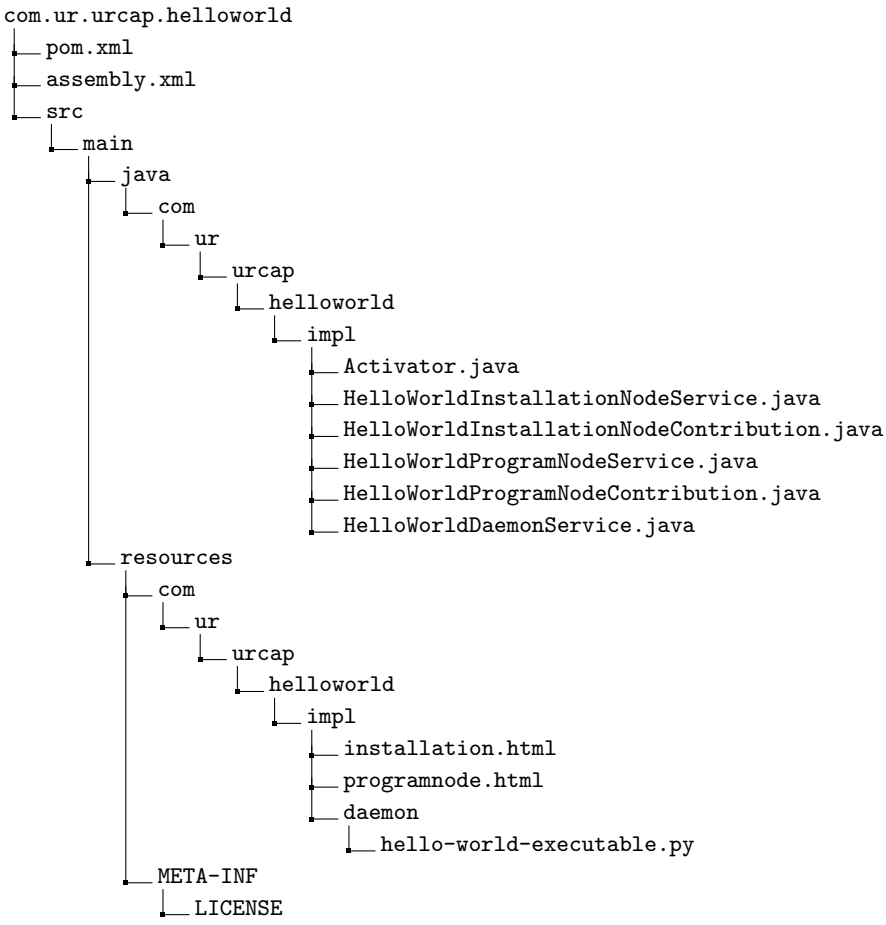


Figure 3: Structure of the Hello World URCap project

```
1    $ cd samples/com.ur.urcap.helloworld
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 either using Maven. As above it is possible to supply parameters either directly on the command line or in the `pom.xml` file. To deploy to a locally running URSim you must specify the path to the extracted URSim using the property `ursim.home`. To deploy to a URSim running in a VM you must 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:

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

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

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

7 Contribution of an installation node

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

7.1 Layout of the Installation Screen

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 A.

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

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

      div.spacer {
        padding-top: 10px;
      }
    </style>
  </head>
  <body>
    <h1>Hello World</h1>
    <div class="spacer"> </div>
    <form>
      <label>Message:</label><input id="message" type="text"/>
      <div class="spacer"> </div>
      <input id="btnEnableDaemon" type="button" />
      <input id="btnDisableDaemon" type="button" />
      <div class="spacer"> </div>
      <label id="lblDaemonStatus">Status of daemon</label>
    </form>
  </body>
</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 label and a single text input widget. It furthermore defines two buttons which can be used to enable and disable the daemon (Section 9 describes daemons in detail). A label will display the current

run status of the daemon.

In order to connect the HTML GUI specification to your Java implementation an `id` attribute is specified for:

- the text field to enable reading of its value
- the buttons in order to capture clicked events
- the label to allow updating it with the current run status of the daemon

This creates a clear 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.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   private final HelloWorldDaemonService helloWorldDaemonService;
14
15   public HelloWorldInstallationNodeService(
         HelloWorldDaemonService helloWorldDaemonService) {
16     this.helloWorldDaemonService = helloWorldDaemonService;
17   }
18
19   @Override
20   public InstallationNodeContribution createInstallationNode(
         URCapAPI api, DataModel model) {
21     return new HelloWorldInstallationNodeContribution(
           helloWorldDaemonService, model);
22   }
23
```

```
24    @Override
25    public String getTitle() {
26      return "Hello World";
27    }
28
29    @Override
30    public InputStream getHTML() {
31      InputStream is = this.getClass().getResourceAsStream("/com/
            ur/urcap/helloworld/impl/installation.html");
32      return is;
33    }
34  }
```

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 (i.e. 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.

The field variable `helloWorldDaemonService` is used to create visibility to the daemon service inside the installation node. We will describe this in greater detail later on.

7.3 Functionality of the Installation Screen

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.helloworld.impl;
2
3  import com.ur.urcap.api.contribution.DaemonContribution;
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.annotation.Label;
8  import com.ur.urcap.api.ui.component.InputButton;
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 import com.ur.urcap.api.contribution.
       InstallationNodeContribution;
13
14 import java.awt.*;
15 import java.util.Timer;
16 import java.util.TimerTask;
17
18 public class HelloWorldInstallationNodeContribution implements
       InstallationNodeContribution {
19
20   private static final String MESSAGE_KEY = "message";
21
22   private final HelloWorldDaemonService helloWorldDaemonService;
23   private DataModel model;
24   private Timer uiTimer;
25
26   private String getMessage() {
27     return model.get(MESSAGE_KEY, "");
28   }
29
30   private void setMessage(String message) {
31     model.set(MESSAGE_KEY, message);
32   }
33
34   public HelloWorldInstallationNodeContribution(
         HelloWorldDaemonService helloWorldDaemonService, DataModel
          model) {
35     this.helloWorldDaemonService = helloWorldDaemonService;
36     this.model = model;
37   }
38
39   @Input(id = "message")
40   private InputTextField messageField;
41
42   @Input(id = "btnEnableDaemon")
43   private InputButton enableDaemonButton;
44
```

```
45    @Input(id = "btnDisableDaemon")
46    private InputButton disableDaemonButton;
47
48    @Label(id = "lblDaemonStatus")
49    private LabelComponent daemonStatusLabel;
50
51    @Input(id = "message")
52    public void onMessageChange(InputEvent event) {
53      if (event.getEventType() == InputEvent.EventType.ON_CHANGE)
          {
54        setMessage(messageField.getText());
55      }
56    }
57
58    @Input(id = "btnEnableDaemon")
59    public void onStartClick(InputEvent event) {
60      if (event.getEventType() == InputEvent.EventType.ON_CHANGE)
          {
61        helloWorldDaemonService.getDaemonContribution().start();
62      }
63    }
64
65    @Input(id = "btnDisableDaemon")
66    public void onStopClick(InputEvent event) {
67      if (event.getEventType() == InputEvent.EventType.ON_CHANGE)
          {
68        helloWorldDaemonService.getDaemonContribution().stop();
69      }
70    }
71
72    @Override
73    public void openView() {
74      messageField.setText(getMessage());
75      enableDaemonButton.setText("Start␣Daemon");
76      disableDaemonButton.setText("Stop␣daemon");
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    @Override
94    public void closeView() {
95      uiTimer.cancel();
96    }
97
```

```
98    private void updateUI() {
99      String text = "";
100     DaemonContribution.State state = helloWorldDaemonService.
            getDaemonContribution().getState();
101     switch (state){
102     case RUNNING:
103       text = "Hello World Daemon runs";
104       break;
105     case STOPPED:
106       text = "Hello World Daemon stopped";
107       break;
108     case ERROR:
109       text = "Hello World Daemon failed";
110     }
111     daemonStatusLabel.setText(text);
112   }
113
114   public boolean isDefined() {
115     return !getMessage().isEmpty();
116   }
117
118   @Override
119   public void generateScript(ScriptWriter writer) {
120     writer.assign("hello_world_message", "\"" + getMessage() + "
            \"");
121   }
122
123
124 }
```

Notice the constructor which has visibility to a daemon through an object of type `HelloWorldDaemonService`. 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 `"message"`. 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`. The same technique is used for the two buttons, and a label that displays the current run status of the daemon.

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 data model under the key `MESSAGE_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 `messageField`.

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` and it updates the GUI with the status of the daemon. The `closeView()` method, called whenever the user leaves the screen, is left empty.

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 this assignment, the script variable with name `"hello_world_message"` is assigned to a string that contains the message 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

Layout for customized program node screens is defined similarly as 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. The name that is typed will be shown in the label with id `"greetinglabel"`. 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;
      }
    </style>
  </head>
  <body>
    <form>
      <label>Enter your name:</label> <input id="yourname" type=
          "text"/>
      <br/>
      <label id="greetinglabel" style="width: 400px;"/>
    </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.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/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.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    private String getName() {
20      return model.get(NAME, "");
21    }
22
23    private void setName(String name) {
24      if ("".equals(name)){
25        model.remove(NAME);
26      }else{
27        model.set(NAME, name);
28      }
29    }
30
31    @Input(id = "yourname")
32    private InputTextField nameTextField;
33
34    @Label(id = "greetinglabel")
35    private LabelComponent greetingLabel;
36
37    public HelloWorldProgramNodeContribution(URCapAPI api,
          DataModel model) {
38      this.api = api;
39      this.model = model;
40    }
41
42    @Input(id = "yourname")
43    public void onInput(InputEvent event) {
44      if (event.getEventType() == InputEvent.EventType.ON_CHANGE)
            {
45        setName(nameTextField.getText());
46        setGreetingLabel();
47      }
48    }
49
50    private void setGreetingLabel() {
51      greetingLabel.setText(model.isSet(NAME) ? "Hello␣" + getName
            () + ",␣welcome␣to␣PolyScope!" : "No␣name␣set");
52    }
53
54    @Override
55    public void openView() {
56      nameTextField.setText(getName());
57      setGreetingLabel();
58    }
59
60    @Override
61    public void closeView() {
62    }
63
64    @Override
65    public String getTitle() {
```

```
66      return model.isSet(NAME) ? "Hello␣" + getName() + ",␣welcome
            ␣to␣PolyScope!" : "HelloWorld";
67    }
68
69
70    @Override
71    public boolean isDefined() {
72      HelloWorldInstallationNodeContribution c = api.
            getInstallationNode(
            HelloWorldInstallationNodeContribution.class);
73      return c.isDefined() && !getName().isEmpty();
74    }
75
76    @Override
77    public void generateScript(ScriptWriter writer) {
78      writer.appendLine("popup(\"Hello␣" + getName() + "\",␣\"Info
            \",␣False,␣False,␣blocking=True)");
79      writer.appendLine("popup(hello_world_message,␣\"Info\",␣
            False,␣False,␣blocking=True)");
80      writer.writeChildren();
81    }
82  }
```

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.

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 entered text is displayed on the screen in the label with id `"greetinglabel"`. Each Hello World node is defined (green) if both the message in the Hello World installation node and the name in the program node are non-empty. When executed, it shows two simple popup dialogs containing the message and the name, respectively.

Note that the message shown within the first popup is the value of the script variable `"hello_world_message"`. 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. A more universal way of passing 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 `isDefined()` method.

9 Contribution of a daemon

A daemon executable can be any script or binary file that can be directly executed in the shell on the control box. A URCap can contribute any number of daemon executables through implementation of the `DaemonService` interface:

Listing 7: The Hello World Daemon Service

```
1  package com.ur.urcap.helloworld.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 HelloWorldDaemonService implements DaemonService {
11
12   private DaemonContribution daemonContribution;
13
14   public HelloWorldDaemonService() {
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/helloworld/impl/daemon/"));
22     } catch (MalformedURLException e) { }
23   }
24
25   @Override
26   public URL getExecutable() {
27     try {
28       return new URL("file:com/ur/urcap/helloworld/impl/daemon/
             hello-world-executable.py");
29     } catch (MalformedURLException e) {
30       return null;
31     }
32   }
33
34   public DaemonContribution getDaemonContribution() {
35     return daemonContribution;
36   }
37
38 }
```

- 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 start, stop, and query a daemon can be found in Listing 3 in section 7.3.
- 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.

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

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).

10 Tying the different contributions together

The new Hello World installation node, program node, and daemon executable are registered and offered to PolyScope through the code in Listing 8. Three services are registered:

- `HelloWorldInstallationNodeService`
- `HelloWorldProgramNodeService`
- `HelloWorldDaemonService`

The `HelloWorldInstallationNodeService` class has visibility to a instance of the `HelloWorldDaemonService` class. This instance is passed in the constructor when a new instance of the `HelloWorldInstallationNodeContribution` installation node is created with the `createInstallationNode(URCapAPI, DataModel)` method. In this way, the daemon executable can be controlled from the installation node.

Listing 8: Tying different URCap contributions together

```
1  package com.ur.urcap.helloworld.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     HelloWorldDaemonService helloWorldDaemonService = new
           HelloWorldDaemonService();
13     HelloWorldInstallationNodeService
           helloWorldInstallationNodeService = new
           HelloWorldInstallationNodeService(
           helloWorldDaemonService);
14
15     context.registerService(InstallationNodeService.class,
           helloWorldInstallationNodeService, null);
16     context.registerService(ProgramNodeService.class, new
           HelloWorldProgramNodeService(), null);
17     context.registerService(DaemonService.class,
           helloWorldDaemonService, null);
18   }
19
20   @Override
21   public void stop(BundleContext context) throws Exception {
22   }
23 }
```

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:

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

This promts 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. 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.

12 Troubleshooting

Internally in PolyScope, a URCap is installed as an OSGi bundle using 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)
10 [   6] [Active     ] [    1] com.ur.dashboardserver (3.3.0.
       SNAPSHOT)
11 [   7] [Active     ] [    1] com.ur.domain (3.3.0.SNAPSHOT)
12 ...
13 [  56] [Active     ] [    1] com.thoughtworks.xstream (1.3.1)
14 [  57] [Active     ] [    1] HelloWorld (1.0.0.SNAPSHOT)
15 ->
```

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

```
1  -> help
2  uninstall
3  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 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`.